

PostgreSQL ve verzi 9.2, 9.3 a 9.4

createdb jménodb

Vytvoří novou databázi

dropdb jménodb

odstraní existující databázi

psql jménodb

spustí SQL konzole

pg_dump jménodb > jméno_souboru

Vytvoří zálohu databáze

SQL konzole – psql

Umožní zadání SQL příkazu a zobrazí jeho výsledek.

Přehled důležitých příkazů

Každý příkaz začíná zpětným lomítkem “\” a není ukončen středníkem.

\c jménodb	přepnutí do jiné databáze
\l	zobrazí seznamu databází
\d objekt	zobrazí popis objektu (tabulky, pohledu)
\dt+	zobrazí seznamu tabulek
\dv	zobrazí seznam pohledů
\df *filtr*	zobrazí seznam funkcí
\sf funkce	zobrazí zdrojový kód funkce
\i	importuje soubor
\h SQL	zobrazí syntaxi SQL příkazu
\?	zobrazí seznam psql příkazů
\q	ukončí konzolu
\x	přepíná řádkové a sloupcové zobrazení
\timing on	zapíná měření času zpracování dotazu

Konfigurace konzole

Soubor .bashrc

```
export PAGER=less
export LESS="-iMSx4 -RSFX -e"
```

Soubor .psqlrc

```
\pset pager always
\pset linestyle unicode
\pset null 'NULL'
\set FETCH_COUNT 1000
\set HISTSIZE 5000
\timing
\set HISTFILE ~/.psql_history-:DBNAME
\set HISTCONTROL ignoredups
\set PROMPT1 '%(n@%M:%>) [%/] > '
\set ON_ERROR_ROLLBACK on
\set AUTOCOMMIT off1
```

Export a import dat

Příkaz COPY

Pomocí příkazu COPY můžeme číst a zapisovat soubory na serveru (pouze superuser) nebo číst ze **stdin** a zapisovat na **stdout**. Podobný příkaz \copy v **psql** umožňuje číst a zapisovat soubory na klientském počítači.

Export tabulky zaměstnanci do CSV souboru

```
COPY zamestnanci TO '/tmp/zam.csv'
CSV HEADER
DELIMITER ',' FORCE QUOTE *;
```

Import tabulky zaměstnanci z domovského adresáře uživatele (v konzoli)

```
\copy zamestnanci from ~/zamestnanci.dta
```

pg_dump – zajímavé parametry

Příkaz pg_dump slouží k jednoduchému zálohování databáze².

-f	specifikuje cílový soubor
-a	exportuje pouze data
-s	exportuje pouze definice
-c	odstraní objekty před jejich importem
-C	vloží příkaz pro vytvoření nové databáze
-t	exportuje pouze jmenovanou tabulku
-T	neexportuje uvedenou tabulku
--disable-triggers	během importu blokuje triggeru
--inserts	generuje příkazy INSERT místo COPY
--Fc	záloha je průběžně komprimovaná s dodatečnými meta informacemi ³

Základní konfigurace PostgreSQL

Soubor postgresql.conf

Po instalaci PostgreSQL je nutné nastavit několik málo konfiguračních parametrů, které ovlivňují využití operační paměti (výchozí nastavení je zbytečně úsporné).

```
shared_buffers*= 2GB
```

velikost paměti pro uložení datových stránek (1/5..1/3 RAM⁵)

```
work_mem = 10MB
```

limit paměti pro běžnou manipulaci s daty (10..100MB)

```
maintenance_work_mem = 200MB
```

limit paměti pro údržbu (100MB ..)

```
effective_cache_size = 6GB
```

odhad objemu dat cache (2/3 RAM)

```
max_connections = 100
```

max počet přihlášených uživatelů (často zbytečně vysoké)

- ² Příkaz pg_dump nezalohuje uživatele. K tomuto účelu se používá příkaz pg_dumpall s parametrem -r. Zálohování příkazem pg_dump je vhodné pro databáze do velikosti cca 50GB. Pro větší databáze je praktičtější použít jiné metody zálohování.
- ³ Pro obnovu je nutné použít pg_restore, obnovit lze i každou vybranou tabulku.
- ⁴ Nastavení shared_buffers nad 28MB typicky vyžaduje úpravu SHMMAX (na os. Linux)
- ⁵ Doporučené hodnoty platí pro tzv dedikovaný server – tj počítač, který je vyhrazen primárně pro provoz databáze.

Mělo by platit⁶:

```
shared_buffers + 2 * work_mem * max_connection <= 2/3 RAM
shared_buffers + 2 * maintenance_work_mem <= 1/2 RAM
max_connections <= 10 * (počet_CPU)
```

checkpoint_segments udává počet 16MB segmentů transakčního logu po jejichž naplnění dojde k tzv CHECKPOINTu (databáze zapíše modifikované datové stránky (cache) na disk).

```
checkpoint_segments = 32
```

Po CHECKPOINTU lze zahodit transakční logy vztažené k času před CHECKPOINTem. Za optimální frekvenci CHECKPOINTů se považuje 5 – 15 min. Obvykle se zvyšuje na 32 až 256⁷ (pro vytížené servery).

```
wal_buffers = 256kB
```

Velikost bufferu transakčního logu by měl být větší než je velikost běžné transakce – pak se provede pouze jeden zápis do transakčního logu během transakce (256kB..1MB). Aktuálně výchozí hodnotou je -1, což znamená zhruba 3% shared_buffers (max 16MB).

```
listen_addresses = '*'
```

A pro vzdálený přístup povolit TCP

SQL

Nejdůležitějším SQL příkazem je příkaz SELECT. Při zápisu je nutné dodržovat pořadí jednotlivých klauzulí:

```
SELECT AVG(a.sloupec1), b.sloupec4
FROM tabulka1 a
JOIN tabulka2 b
ON a.sloupec1 = b.sloupec2
WHERE b.sloupec3 = 'něco'
GROUP BY b.sloupec4
HAVING AVG(a.sloupec1) > 100
ORDER BY 1
LIMIT 10
```

Sjednocení, průnik, rozdíl relací

Pro relace (tabulky) existují operace sjednocení (UNION), průnik (INTERSECT) a rozdíl (EXCEPT). Častou operací je sjednocení relací – výsledků dvou příkazů SELECT – operace sloučí řádky (a zároveň odstraní případné duplicitní řádky). Podmínkou je stejný počet sloupců a konvertibilní datové typy slučovaných relací.

Vybere 10 nejstarších zaměstnanců bez ohledu zdali se jedná o interního nebo externího zaměstnance:

```
SELECT jmeno, prijmeni, vek
FROM zamestnanci
UNION8
SELECT jmeno, prijmeni, vek
FROM externi_zamestnanci
ORDER BY vek DESC9
LIMIT 10;
```

CASE

Konstrukce CASE se používá pro transformace hodnot – zobrazení, bez nutnosti definovat vlastní funkce. Existují dva zápisy – první hledá konstantu, v druhém se hledá platný výraz:

¹ Doporučeno pro produkci – vynucuje potvrzení změn explicitním COMMITem. Po vypnutí autocommitu se psql bude chovat podobně jako konzole Oracle.

⁶ Jedná se o orientační hodnoty určené pro počáteční konfiguraci “typického použití” databáze.
⁷ Příliš vysoká hodnota může zvýšit dobu obnovy po havárii, kdy se kontroluje transakční log.
⁸ Při použití UNION ALL nedochází k odstranění duplicitních řádků – což může zrychlit vykonání dotazu.
⁹ Klauzule ORDER BY se aplikuje na výsledek algebraických operací

```
SELECT CASE sloupec WHEN 0 THEN 'NE'
              WHEN 1 THEN 'ANO' END
FROM tabulka;

SELECT CASE WHEN sloupec = 0 THEN 'NE'
              WHEN sloupec = 1 THEN 'ANO' END
FROM tabulka;
```

V případě, že se nenajde hledaná konstanta a nebo že žádný výraz není pravdivý, tak je výsledkem hodnota za klíčovým slovem ELSE – nebo NULL, pokud chybí ELSE.

Agregační funkce s definovaným pořadím

Výsledek novějších agregačních funkcí – string_agg, array_agg závisí na pořadí ve kterém se zpracovávala agregovaná data. Proto je možné přímo v agregační funkci určit v jakém pořadí bude agregační funkce načítat hodnoty. **Klauzule ORDER BY musí být za posledním argumentem agregační funkce.**

Vrátí seznam zaměstnanců v každém oddělení řazený podle příjmení:

```
SELECT sekce_id, string_agg(prijmeni, ',' ORDER BY prijmeni)
FROM zamestnanci
GROUP BY sekce_id
```

Agregační funkce nad uspořádanou množinou¹⁰

Tato speciální syntax se používá pouze pro funkce, jejichž výpočet vyžaduje seřazená data (např. výpočet percentilů). Následující dotaz zobrazí medián (50% percentil) mzdy zaměstnanců.

```
SELECT percentile_cont(0.511) WITHIN GROUP (ORDER BY mzda)
FROM zamestnanci
```

Poddotazy

Příkaz SELECT může obsahovat vnořené příkazy SELECT. Vnořený příkaz SELECT se nazývá **poddotaz** a vkládá se do obklíčkových závorek. Poddotazy se mohou použít i u dalších SQL příkazů.

Poddotaz ve WHERE

Používá se pro filtrování – následující dotaz zobrazí obce z okresu Benešov:

```
SELECT nazev
FROM obce o
WHERE o.okres_id = (SELECT id
                   FROM okresy
                   WHERE kod = 'BN')
```

Korelované poddotazy

Poddotaz se může odkazovat na výsledek, který produkuje vnější dotaz.

Pro každého zaměstnance zobrazí seznam jeho dětí:

```
SELECT jmeno, prijmeni
      (SELECT string_agg(jmeno, ',')
       FROM deti d
       WHERE d.zamestnanec_id = z.id)
FROM zamestnanci z
```

Zobrazí zaměstnance, kteří mají děti:

```
SELECT jmeno, prijmeni
```

```
FROM zamestnanci z
WHERE EXISTS(SELECT id
             FROM deti d
             WHERE d.zamestnanec_id = z.id)
```

Zobrazí z každého oddělení dva nejstarší zaměstnance (více násobné použití tabulky)

```
SELECT jmeno, prijmeni
FROM zamestnanci z1
WHERE vek IN (SELECT vek
             FROM zamestnanci z2
             WHERE z2.sekce_id = z1.sekce_id
             ORDER BY vek DESC
             LIMIT 2)
```

Spojení relací¹² JOIN

Příkaz JOIN spojuje relace (tabulky) vedle sebe a to na základě stejných hodnot v jednom nebo více atributu (sloupci). Každé spojení specifikuje dvě relace (spojkou je klíčové slovo JOIN) a podmínku, která určuje, jak se tyto relace budou spojovat (zapsanou za klíčovým slovem ON).

Vnitřní spojení relací – INNER JOIN

Nejčastější varianta – do výsledku se zahrnou pouze řádky, které se podařilo dohledat v obou relacích (stejně hodnota/hodnoty) se našly v obou tabulkách.

Zobrazí jméno dítěte a jméno rodiče (zaměstnance) – v případě, že má zaměstnanec více dětí, tak jeho jméno bude uvedeno opakovaně:

```
SELECT d.jmeno, d.prijmeni, z.jmeno, z.prijmeni
FROM deti d
JOIN zamestnanci z
ON d.zamestnanec_id = z.id
```

Vnější spojení relací – OUTER JOIN

Jedná se o rozšíření vnitřního spojení – kromě řádků, které se spárovaly se do výsledku zařadí i nespárované řádky z tabulky nalevo od slova JOIN (LEFT JOIN) nebo napravo od slova JOIN (RIGHT JOIN). Chybějící hodnoty se nahradí hodnotou NULL.

Často se používá dohromady s testem na hodnotu NULL – operátorem IS NULL¹³. Tím se vyberou nespárované řádky – např. pro zobrazení zaměstnanců, kteří nemají děti, lze použít dotaz:

```
SELECT z.jmeno, z.prijmeni
FROM zamestnanci z
LEFT JOIN deti d
ON z.id = d.zamestnanec_id
WHERE d.id IS NULL.
```

Použití derivované tabulky

Poddotaz se může objevit i v klauzuli FROM – pak jej označujeme jako derivovaná tabulka¹⁴. I derivovanou tabulku lze spojovat s běžnými tabulkami (obojí je relací).

Následující příklad zobrazí seznam nejstarších zaměstnanců z každého oddělení:

```
SELECT z.jmeno, z.prijmeni
FROM zamestnanci z
JOIN (SELECT sekce_id, MAX(vek) AS vek
      FROM zamestnanci
      GROUP BY sekce_id) s
```

```
ON z.sekce_id = s.sekce_id
AND z.vek = s.vek
```

Dotazy s LATERAL relacemi

Klauzule LATERAL¹⁵ umožňuje ke každému záznamu relace X připojit výsledek poddotazu (derivované tabulky), uvnitř kterého je možné použít referenci na relaci X. Místo derivované tabulky lze použít funkci, která vrátí tabulku, a pak atribut(y) z relace X může být argumentem této funkce.

Pro každý záznam z tabulky a vrátí všechny záznamy z tabulky b, pro které platí, že atribut a je větší než dvojnásobek atributu b.

```
SELECT *
FROM a,
      LATERAL (SELECT *
              FROM b
              WHERE a.a > 2 * b.b) x;
```

Pro každou hodnotu vrátí součet všech kladných celých čísel menší rovno této hodnotě:

```
SELECT a, sum(i)
FROM a,
      LATERAL generate_series(1, a) g(i)
GROUP BY a
ORDER BY 1;
```

Analytické (window) funkce

Analytické funkce se počítají pro každý prvek definované podmnožiny, např. pořadí prvku v podmnožině. Na rozdíl od agregačních funkcí se podmnožiny nedefinují klauzulí GROUP BY, ale klauzulí PARTITION hned za voláním analytické funkce (v závorce za klíčovým slovem OVER). Mezi nejčastěji používané analytické funkce bude patřit funkce row_number (číslo řádku) nebo ranking (pořadí hodnoty), případně dense_rank a percent_rank.

Pozor – pro analytické funkce nelze použít klauzuli HAVING – filtrování hodnot se řeší použitím derivované tabulky.

Následující dotaz vybere deset nejdéle zaměstnaných pracovníků (na základě porovnání osobních čísel):

```
SELECT jmeno, prijmeni
FROM (SELECT rank() OVER (ORDER BY id),
         jmeno, prijmeni
      FROM zamestnanci
      WHERE ukonceni_prac_pomeru IS NULL) s
WHERE s.rank <= 10
```

Zobrazení dvou nejstarších zaměstnanců z každého oddělení:

```
SELECT jmeno, prijmeni
FROM (SELECT rank() OVER (PARTITION BY sekce_id
                        ORDER BY vek DESC),
         jmeno, prijmeni
      FROM zamestnanci) s
WHERE s.rank <= 2
```

Seznam tří nejlépe hodnocených pracovníků z každého oddělení:

```
SELECT jmeno, prijmeni
FROM (SELECT rank() OVER (PARTITION BY sekce_id
                        ORDER BY hodnoceni),
         jmeno, prijmeni, hodnoceni, sekce_id
      FROM zamestnanci) s
WHERE s.rank <= 2
ORDER BY sekce_id, hodnoceni
```

10 v PostgreSQL 9.4

11 Rozšíření vůči ANSI/SQL umožňuje zadat více parametrů jako pole – výsledkem je opět pole.

12 Tabulka je relací. Výsledek SQL dotazu je relací. Tudíž příkaz SELECT můžeme aplikovat na tabulku nebo i na výsledek jiného příkazu SELECT.

13 Pro hodnotu NULL není možné použít operátor =.

14 SELECT ze SELECTu

15 V PostgreSQL 9.3

O síle analytických funkcí nás může přesvědčit následující příkaz. V tabulce *statistics* se ukládají aktuální hodnoty čítačů množství vložených, aktualizovaných a odstraněných řádek. Odečet čítačů se provádí každých 5 minut. Následující dotaz zobrazí počet vložených, aktualizovaných a odstraněných řádek pro každý interval:

```
SELECT dbname, time,
       tup_inserted - lag16(tup_inserted) OVER w as tup_inserted,
       tup_updated - lag(tup_updated) OVER w as tup_updated,
       tup_deleted - lag(tup_deleted) OVER w as tup_deleted,
FROM statistics WINDOW w17 AS (PARTITION BY dbname,
                                ORDER BY time)
```

Common Table Expressions – CTE

Pomocí CTE můžeme dočasně (v rámci jednoho SQL příkazu) definovat novou relaci a na tuto relaci se můžeme opakovaně odkazovat.

Nerekurzivní CTE

CTE klauzule umožňuje řetězení (pipelining) SQL příkazů (archivuje zrušené záznamy):

```
WITH t1 AS (DELETE FROM tabulka RETURNING *),
     t2 AS (INSERT INTO archiv SELECT * FROM t1 RETURNING *)
SELECT * FROM t2;
```

Vrací čísla dělitelná 2 a 3 beze zbytku z intervalu 1 až 20 (zabíráje opakovanému výpočtu):

```
WITH iterator AS (SELECT i FROM generate_series(1,20) g(i))
SELECT * FROM iterator WHERE i % 2 = 0
UNION
SELECT * FROM iterator WHERE i % 3 = 0
ORDER BY 1;
```

V PostgreSQL mohou relace vzniknout i na základě DML příkazů (INSERT, UPDATE, DELETE).

```
WITH upsert18 AS (UPDATE target t SET c = s.c
                  FROM source s
                  WHERE t.id = s.id
                  RETURNING s.id)
INSERT INTO target
SELECT *
FROM source s
WHERE s.id NOT IN (SELECT id
                  FROM upsert)
```

Rekurzivní CTE

Lokální relace vzniká jako výsledek iniciálního SELECTu *S1*, který vrací kořen a opakovaného volání SELECTu *S2*, který vrací všechny potomky uzlů, které byly dohledány v předchozí iteraci. Rekurse končí, pokud výsledkem *S2* je prázdná relace:

```
WITH RECURSIVE ti
AS (SELECT S1
   UNION ALL
   SELECT S2
   FROM tabulka t
   JOIN ti
   ON t.parent = ti.id)
SELECT *
FROM ti;
```

Zobrazí seznam všech zaměstnanců, kteří jsou přímo nebo nepřímo podřízeni zaměstnanci s *id = 1* (včetně hloubky rekurse):

```
WITH RECURSIVE os
AS (SELECT , 1 AS hloubka
   FROM zamestnanci
   WHERE id = 1
   UNION ALL
   SELECT z.*, hloubka + 1
   FROM zamestnanci z
   JOIN os
   ON z.nadrizeny = os.id)
SELECT *
FROM os;
```

Ostatní SQL příkazy

INSERT

Jednoduchý INSERT s vložením defaultní hodnoty

```
INSERT INTO tab1(id, t) VALUES(DEFAULT, '2012-12-16');
```

Vícenásobný INSERT

```
INSERT INTO tab2(a, b) VALUES(10,20), (30,40)
```

INSERT SELECT – vloží výsledek dotazu včetně aktuálního času

```
INSERT INTO statistics
SELECT CURRENT_TIMESTAMP, *
FROM pg_stat_user_tables
```

UPDATE

Aktualizace na základě dat z jiné tabulky

```
UPDATE zamestnanci z
SET mzda = n.mzda
FROM novy_vymer n
WHERE z.id = n.id
```

DELETE

Příkaz DELETE odstraňuje záznamy z tabulky

```
DELETE FROM produkty
WHERE id IN (SELECT id
            FROM ukoncene_produkty)
```

Častou úlohou je odstranění duplicitních řádek:

```
DELETE FROM lidi l
WHERE ctid19 <> (SELECT ctid
                 FROM lidi
                 WHERE prijmeni=l.prijmeni
                 AND jmeno=l.jmeno
                 LIMIT 1);
```

Často používané funkce a operátory

```
substring('ABC' FROM 1 FOR 2)
upper('ahoj')
lower('AH0J')
to_char(now(), 'DD.MM.YY')
to_char(now(), 'HH24:MI:SS')
trim(' aa ')
EXTRACT(dow FROM now())
```

vrátí podřetězec
převede text na velká písmena
převede text na malá písmena
formátuje datum
formátuje čas
odstraní krajní mezery
vrátí den v týdnu

```
EXTRACT(day FROM now())
EXTRACT(month FROM now())
EXTRACT(year FROM now())
date_trunc('month', now())
COALESCE(a,b,c)
array_lower(a, 1)
array_upper(a,1)
random()
generate_series(1,h)
array_to_string(a, ',')
string_to_array(a, ',')
string_agg(a, ', ')
concat('A',NULL,'B')
concat_ws(',', 'A',NULL,'B')
```

```
'Hello' || 'World'
10 IS NULL
10 IS NOT NULL
10 IS DISTINCT FROM 20
10 IS NOT DISTINCT FROM 20
```

vrátí den v měsíci
vrátí měsíc
vrátí rok
vrátí nejbližší začátek období
vrátí první ne NULL hodnotu
vrátí spodní index pole nté dimenze
vrátí horní index pole nté dimenze
vrátí pseudonáhodné číslo [0..1)
generuje posloupnost od l do h
serializuje pole
parsuje řetězec do pole
agreguje do seznamu hodnot
spojuje řetězce, ignoruje NULL
spojuje řetězce daným separátorem

```
row_number()
rank()
dense_rank()
lag(a, 1, -1)
lead(a, 1, -1)
ntile(10)
```

číslo řádku v podmnožině
pořadí v podmn. – nesouvislá řada
pořadí v podmn. – souvislá řada
n-tá předchozí hodnota v podmn.
n-tá následující hodnota v podmn.
vrací číslo podmnožiny z n skupin²⁰

```
nazev ~ 'xx$'      test na regulární výraz (citlivé na velikost písmen)
název ~* 'xx$'     test na regulární výraz (bez ohledu na velikost písmen)
```

Přibližné dohledání mediánu:

```
SELECT max(a)
FROM (SELECT a, ntile(2) OVER (ORDER BY a)
      FROM a) x
WHERE ntile = 1;
```

Monitoring

Offline

Základní úkolem je monitorování pomalých dotazů²¹, popřípadě monitorování událostí, které jsou obvykle spojeny s výkonovými problémy.

```
log_min_duration_statement = 200
```

zapiše dotaz, který běžel déle než 200 ms

```
log_lock_waits = on
```

zaloguje čekání na zámek delší než detekce deadlocku (1 sec)

```
log_temp_files = 1MB
```

zaloguje vytvoření dočasného souboru většího než 1MB²²

²⁰ Rozdělí množinu do n podobně velkých podmnožin. Lze použít pro orientační určení mediánu a kvantilů.

²¹ Pro analýzu pomalých dotazů lze použít **pgFouine** nebo **pgbadger**. K monitorování lze použít extenze **auto_explain** (zapiše do logu prováděcí plán pomalého dotazu)

²² Velké množství dočasných souborů může signalizovat nízkou hodnotu **work_mem**.

¹⁶ Funkce **lag** vrací předchozí hodnotu atributu v podmnožině.

¹⁷ Příklad obsahuje ukázkou sdílené definice okna (podmnožiny) **w**.

¹⁸ V případě, že záznam existuje, provede UPDATE, jinak INSERT.

¹⁹ *Ctid* je fyzický identifikátor záznamu – v podstatě je to pozice záznamu v datovém souboru. Hodí se pouze pro některé úlohy, neboť po aktualizaci má záznam jiné *ctid*.

Online

Dotazy do systémových tabulek můžeme zjistit aktuální stav a provoz databáze, případně využití jednotlivých databázových objektů.

Stav otevřených spojení (přihlášených uživatelů do db)

```
SELECT * FROM pg_stat_activity;
```

Přerušení všech dotazů běžících déle než 5 min

```
SELECT pg_cancel_backend(pid)
FROM pg_stat_activity
WHERE current_timestamp - query_start > interval '5 min';
```

Využití jednotlivých db (včetně aktuálně přihlášených uživatelů k db)

```
SELECT * FROM pg_stat_database;
```

Využití tabulek²³ (počet čtení, počet zápisů, ...)

```
SELECT * FROM pg_stat_user_tables;
```

Využití IO, cache vztažené k tabulkám

```
SELECT * FROM pg_statio_user_tables;
```

Po instalaci doplňku pg_buffercache můžeme monitorovat obsah PostgreSQL cache. Funkce z doplňku pgstattuple umožňují provést nízkouřvňovou diagnostiku datových souborů tabulek a indexů.

PL/pgSQL

PL/pgSQL je jednoduchý programovací jazyk vycházející z PL/SQL (Oracle) a potažmo ze zjednodušeného programovacího jazyka ADA. Je těsně spjat s prostředím PostgreSQL – k dispozici jsou pouze datové typy, které nabízí PostgreSQL a operátory a funkce pro tyto typy. Je to ideální lepidlo pro SQL příkazy, které mohou být vykonány na serveru, čímž se odbourávají latence způsobené sítí a protokolem.

Základní funkce

Funkce slouží k získání výsledku nebo provedení nějaké operace nad daty. Funkce v PostgreSQL mohou vracet skalární hodnotu (jeden atribut), záznam (více atributů), pole, případně tabulku. Uvnitř funkcí nelze používat explicitně řízení transakcí²⁴.

```
CREATE OR REPLACE FUNCTION novy_zamestnanec(jmeno text,
                                             plny_uvazek boolean)
RETURNS void AS $$
BEGIN
    IF plny_uvazek THEN
        INSERT INTO zamestnanci
            VALUES(novy_zamestnanec.jmeno);
    ELSE
        INSERT INTO externisti
            VALUES(novy_zamestnanec.jmeno);
    END IF;
END;
$$ LANGUAGE plpgsql;

SELECT novy_zamestnanec('Stěhule', true);
SELECT novy_zamestnanec(jmeno := 'Stěhule', true);
```

Iterace nad výsledkem dotazu

V některých případech potřebujeme zpracovat výsledek dotazu – iterace FOR SELECT nám umožňuje provést určitý proces nad každým záznamem vrácené relace (pozor – v případě, že lze iteraci nahradit jedním čitelným SQL příkazem, měli bychom preferovat jeden SQL příkaz):

```
DECLARE r record;
BEGIN
    FOR r IN SELECT * FROM pg_database
    LOOP
        RAISE NOTICE25 '%', r;
    END LOOP;
END;
```

Provedení akce pokud hodnota existuje

Jedná se o typický vzor, kde je začátečnickou chybou rozhodovat nad počtem záznamů – což může být řádově dražší úloha než test na existenci hodnoty.

```
BEGIN
    IF EXISTS(SELECT 1
               FROM zamestnanci z
               WHERE z.jmeno = _jmeno
               FOR UPDATE26)
    THEN
        ...
    END IF;
END;
```

Ošetření chyby

PL/pgSQL vytváří subtransakci pro každý chráněný blok – v případě zachycení výjimky je tato subtransakce automaticky odvolána:

```
CREATE OR REPLACE FUNCTION fx(a int, b int)
RETURNS int AS $$
BEGIN
    RETURN a / b;
EXCEPTION WHEN division_by_zero THEN
    RAISE EXCEPTION 'dělení nulou';
END;
$$ LANGUAGE plpgsql IMMUTABLE STRICT;
```

Funkce s defaultními parametry

PostgreSQL podporuje defaultní hodnoty parametrů funkce – při volání funkce, lze parametr, který má přiřazenou defaultní hodnotu vynechat.

Následující funkce vrátí tabulku existujících databází – a v případě, že parametr vynecháme, tak tabulku databází aktuálního uživatele:

```
CREATE OR REPLACE FUNCTION dblist(username text
                                  DEFAULT CURRENT_USER)
RETURNS SETOF text AS $$
BEGIN
    RETURN QUERY SELECT datname::text
                  FROM pg_database d
                  WHERE pg_catalog.pg_get_userbyid(d.datdba)
                      = username;
    RETURN;
END;
$$ LANGUAGE plpgsql;

SELECT * FROM dblist('postgres');
SELECT * FROM dblist(username := 'postgres');
SELECT * FROM dblist();
```

Variadické funkce

Variadická funkce je funkce s proměnlivým počtem parametrů. Posledním parametrem této funkce je tzv variadický parametr typu pole.

Následující ukázka je vlastní implementace funkce *least* – získání minimální hodnoty ze seznamu hodnot:

```
CREATE OR REPLACE FUNCTION myleast(VARIADIC numeric[])
RETURNS numeric AS $$
    SELECT MIN(v)
    FROM unnest($1) g(v);
$$ LANGUAGE sql;
```

Zde se nejedná o PL/pgSQL funkci, ale o SQL funkci – pro triviální funkce je vhodnější používat tento jazyk:

```
SELECT myleast(10,1,2);
SELECT myleast(VARIADIC ARRAY[10,1,2])
```

Polymorfní funkce

Polymorfní funkce jsou generické funkce, navrženy tak, aby byly funkční s libovolným datovým typem. Místo konkrétního typu parametru použijeme generický typ – ANYELEMENT, ANYARRAY, ANYNONARRAY, ANYRANGE a ANYENUM.

Generická funkce *myleast* by mohla vypadat následujícím způsobem:

```
CREATE OR REPLACE FUNCTION myleast(VARIADIC ANYARRAY)
RETURNS ANYELEMENT AS $$
    SELECT MIN(v)
    FROM unnest($1) g(v);
$$ LANGUAGE sql;
```

SECURITY DEFINER funkce

Kód funkce v PostgreSQL běží s právy uživatele, který danou funkci aktivoval²⁷ (podobné je to i u triggerů). Toto chování lze změnit – pomocí atributu funkce SECURITY DEFINER. Tato technika se používá v situacích, kdy dočasně musíme zpřístupnit data, ke kterým běžně není přístup.

Následující funkci musí zaregistrovat (tím se stane jejím vlastníkem) uživatel s přístupem k tabulce *users*:

```
CREATE OR REPLACE FUNCTION verify_login(username text,
                                         password text)
RETURNS boolean AS $$
BEGIN
    IF EXISTS(SELECT *
              FROM users u
              WHERE u.passwd = md5(verify_login.password)
              AND u.name = verify_login.username)
    THEN
        RETURN true;
    ELSE
        RAISE WARNING 'unsuccessful login: %', username;
        PERFORM pg_sleep(random() * 3);
        RETURN false;
    END;
END;
$$ SECURITY DEFINER
LANGUAGE plpgsql;
```

Výhodou tohoto řešení je skutečnost, že i když útočník dokáže kompromitovat účet běžného uživatele, nezíská přístup k tabulce *users*.

Triggery

Triggerem se v PostgreSQL myslí vazba mezi určitou událostí a jednou konkrétní funkcí. Pokud ta událost nastane, tak se vykoná dotyčná funkce. Triggerem můžeme sledovat změny dat v tabulkách (klasické BEFORE, AFTER triggery), pokus o změnu dat v pohledu (INSTEAD OF triggery), případně změny v systémovém katalogu (EVENT triggery²⁸).

²³ Pro indexy - pg_stat_user_indexes

²⁴ Používají se pouze subtransakce (implicitní) a to k zajištění ošetření zachycení výjimky.

²⁵ Zobrazí text na ladicí výstup.

²⁶ Pozor na případnou RACE CONDITION.

²⁷ Toto chování je podobné přístupu k uživatelským právům v Unixu. Pozor – prakticky ve všech ostatních db (včetně ANSI SQL) je to jinak – kód uvnitř funkce je vykonáván s právy vlastníka funkce.

²⁸ V PostgreSQL 9.3

Nejčastěji používané jsou BEFORE, AFTER triggery volané po operacích INSERT, UPDATE a DELETE. Vybrané funkce se mohou spouštět pro každý příkazem dotčený řádek (row trigger) nebo jednou pro příkaz (statement trigger). U řádkových triggerů máme k dispozici proměnnou NEW a OLD, obsahující záznam před provedením a po provedení příkazu. Modifikací proměnné NEW můžeme záznam měnit (v BEFORE triggeru). V době provedení funkcí BEFORE triggerů je dotčený záznam ještě v nezměněné podobě. Funkce AFTER triggerů se volají v době, kdy tabulka obsahuje nové verze všech záznamů²⁹.

```
CREATE OR REPLACE FUNCTION pridej_razitko()
RETURNS trigger AS $$
BEGIN
    NEW.vlozeno := CURRENT_TIMESTAMP;
    NEW.provedl := SESSION_USER;
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER orazitkuj_zmenu_zamestnanci
BEFORE INSERT OR UPDATE ON zamestnanci
FOR EACH ROW
EXECUTE PROCEDURE pridej_razitko();
```

Partitioning

Partitioning umožňuje rozdělit data v relaci do definovaných fyzicky oddělených disjunktních podmnožin. Později, při zpracování dotazu se použijí pouze ty *partitions*, které jsou pro zpracování dotazu nezbytné.

Dědičnost relací

Velice specifickou vlastností PostgreSQL je částečná podpora OOP – podpora dědičnosti. Relace může být vytvořena děděním jiné relace. Relace potomka obsahuje atributy rodiče a případně další. Relace rodiče obsahuje všechny záznamy relací, které vznikly jejím přímým nebo nepřímým podděním. Například z relace *lidé* (*jméno*, *příjmení*) podděním relace *studenti* (*jméno*, *příjmení*, *obor*) a *zaměstnanci* (*jméno*, *příjmení*, *zařazení*). Dotaz do relace *lidé* zobrazí jak všechny studenty tak všechny zaměstnance.

```
CREATE TABLE lidé(jmeno text, prijmeni text);
CREATE TABLE studenti(obor text) INHERITS (lidé);
CREATE TABLE zamestnanci(zarazeni text) INHERITS(lidé);
```

Partition je v PostgreSQL podděšená relace (tabulka) s definovaným omezením. Toto omezení by mělo časově invariantní (tj neměl bych se snažit o *partition* pro „posledních 30 dní“)

```
CREATE TABLE objednavka(vlozeno date, castka numeric(12,2));
CREATE TABLE objednavka_2012
    (CHECK(EXTRACT(year FROM vlozeno) = 2012))
    INHERITS (objednavka);
CREATE TABLE objednavka_2011
    (CHECK(EXTRACT(year FROM vlozeno) = 2011))
    INHERITS (objednavka);
```

Omezení

✗ *Partitions se nevytváří* automaticky³⁰ - musíme si je vytvořit manuálně.

✗ Umístění záznamů do odpovídajících *partitions se neprovádí* automaticky³¹ - musíme si napsat distribuční trigger.

29 AFTER triggergy používáme, když potřebujeme vidět změny v tabulce. Provádějí se až po vložení, aktualizaci, odstranění všech řádků realizovaných jedním SQL příkazem a jsou proto o něco málo náročnější než BEFORE triggergy – musí se udržovat fronta nevyhodnocených AFTER triggerů.

30 Lze je vytvářet uvnitř triggerů, ale to nedoporučuji – hrozí **race condition** nebo ztráta výkonu z důvodu čekání na zámek. Nejjednodušší a nejpraktičtější je vyrobit *partitions* na rok dopředu.

31 Základem je distribuční BEFORE INSERT trigger nad rodičovskou tabulkou. V případě, že dochází při UPDATE k přesunu mezi *partitions* je nutný BEFORE UPDATE trigger nad každou podděšenou tabulkou.

✗ Počet *partitions* není omezen – neměl by ovšem přesáhnout 100 *partitions* jedné tabulky³².

Redistribuční trigger

Úkolem tohoto triggeru je přesun záznamu z rodičovské tabulky do odpovídající podděšené tabulky³³ (pro větší počet *partitions* – cca nad 20 je praktické použití *dynamického SQL*).

```
CREATE OR REPLACE FUNCTION public.objednavka_bi()
RETURNS trigger AS $$
BEGIN
    CASE EXTRACT(year FROM NEW.vlozeno)
        WHEN 2011 THEN
            INSERT INTO objednavka_2011 VALUES(NEW.*);
        WHEN 2012 THEN
            INSERT INTO objednavka_2012 VALUES(NEW.*);
        ELSE
            RAISE EXCEPTION 'chybejici partition pro rok %',
                EXTRACT(year FROM NEW.vlozeno);
    END CASE;
    RETURN NULL;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER objednavka_before_insert_trg
BEFORE INSERT ON objednavka
FOR EACH ROW EXECUTE PROCEDURE public.objednavka_bi();
```

Použití

Při plánování dotazu se provádí identifikace *partitions*, které lze bezpečně vyjmout z plánování neboť obsahují pouze řádky, které 100% nevyhovují podmínkám, a tyto *partitions* se při zpracování dotazu nepoužijí. U každého dotazu, kde předpokládáme aplikaci *partitioningu si ověřujeme* (příkaz EXPLAIN), že dotaz je napsán tak, že *planner* z něj dokáže detekovat nepotřebné *partitions*.

```
postgres=# EXPLAIN SELECT * FROM objednavka
           WHERE EXTRACT(year from vlozeno) > 2012;
           QUERY PLAN

Result  (cost=0.00..77.05 rows=1087 width=20)
-> Append  (cost=0.00..77.05 rows=1087 width=20)
-> Seq Scan on objednavka
    Filter: (date_part('year', vlozeno) > 2012)
-> Seq Scan on objednavka_2013
    Filter: (date_part('year', vlozeno) > 2012)
(6 rows)
```

Kombinace bash a psql

psql lze použít i pro jednodušší skriptování (automatizaci) v kombinaci s Bashem. V jednodušších případech stačí použít parametr -c "SQL příkaz". Ten ovšem nelze použít, když chceme použít dotaz parametrizovat pomocí psql proměnných.

Ukázka využívá psql proměnných, heredoc zápis a binární ASCII *unit* separator :

```
SQL=$(cat <<EOF
SELECT datname, pg_catalog.pg_get_userbyid(d.datdba)
FROM pg_database d
WHERE pg_catalog.pg_get_userbyid(d.datdba) = :'owner'
EOF
)
echo $SQL | psql postgres -q -t -A -v owner=$1 -F '$\x1f' | \
while IFS=$'\x1f' read -r a b;
do
    echo -e "datname='$a'\towner='$b'";
```

32 Při velkém počtu *partitions* je problém s paměťovými nároky optimizátoru.

33 Také aplikace může přesněji čítat a zapisovat do tabulek, které odpovídají *partitions* a nikoliv do rodičovské tabulky – tím se ušetří volání redistribučního triggeru a INSERT bude rychlejší.

```
done

Obilíbeným trikem je vygenerování DDL příkazů v psql, které se pošlou jiné instanci psql, kde se provedou. Následující skript odstraní všechny databáze vybraného uživatele:
```

```
SQL=$(cat <<EOF
SELECT format('DROP DATABASE %I;', datname)
FROM pg_database d
WHERE pg_catalog.pg_get_userbyid(d.datdba) = :'owner'
EOF
)

echo $SQL | psql postgres -q -t -A -v owner=$1 | \
psql -e postgres
```

Jednodušší skripty můžeme napsat pomocí tzv online bloků³⁴ – kódu v plpgsql.

```
SQL=$(cat <<'EOF'
SELECT set_config('custom.owner', :'owner', false);
DO $$
DECLARE name text;
BEGIN
    FOR name IN SELECT d.*
FROM pg_database d
WHERE pg_catalog.pg_get_userbyid(d.datdba)
= current_setting('custom.owner')

    LOOP
        RAISE NOTICE 'dabaze=%', name;
    END LOOP;
END;
$$
EOF
)

echo $SQL | psql postgres -v owner=$1
```

Zajímavým trikem je generování obsahu ve formátu vhodném pro příkaz COPY, který se pomocí roury natlačí do další instance konzole. Následující příkaz uloží aktuální stav provozních statistik a uloží je do souhrnné tabulky v databázi *postgres*.

```
SQL_stat=$(cat <<'EOF'
SELECT current_database(), current_timestamp::timestamp(0),
sum(n_tup_ins) tup_inserted,
sum(n_tup_upd) tup_updated,
sum(n_tup_del) tup_deleted,
FROM pg_stat_user_tables
GROUP BY substring(relname from 1 for 2);
EOF
)

for d in `psql -At -c "select datname from pg_database where
pg_get_userbyid(datdba) <> 'postgres'"`
do
    echo $SQL_stat | psql -At $d -F'$\t' | \
    psql postgres -c "COPY statistics FROM stdin"
done
```

Online fyzické zálohování

Kontinuální

Při kontinuálním zálohování archivujeme segmenty transakčního logu. Na základě obsahu transakčního logu jsme schopni zrekonstruovat stav databáze v libovolném okamžiku od vytvoření kompletní zálohy do okamžiku získání posledního validního segmentu transakčního logu.

Konfigurace

Pro vytvoření zálohy musíme povolit export segmentů transakčního logu a nastavit

34 Předávání parametrů dovnitř online bloku je o něco málo komplikovanější..

tzv `archive_command`³⁵:

```
archive_mode = on
archive_command = 'cp %p /var/backup/xlogs/%f'
archive_timeout = 300
```

Vytvoření zálohy

Vynucení checkpointu a nastavení štitku (label) plné zálohy

```
SELECT pg_start_backup(current_timestamp::text);
```

Záloha datového adresáře – bez transakčních logů

```
cd /usr/local/pgsql
tar -cjf pgdata.tar.bz2 --exclude='pg_xlog' data/*
```

Ukončení plné zálohy (full backup)

```
SELECT pg_stop_backup();
```

Adresář `/var/backup/xlogs` se začne plnit transakčními logy³⁶.

Obnova ze zálohy

Rozbalení poslední plné zálohy

```
cd /usr/local/pgsql
tar xvfj pgdata.tar.bz2
```

V datovém adresáři vytvořte soubor **recovery.conf**, ve kterém definujete tzv `restore_commad` (analogicky k `archive_command`):

```
restore_command = 'cp /var/backup/xlogs/%f %p'
```

Pokud je čitelný adresář s transakčními logy původního serveru, tak můžeme tento adresář zkopírovat do datového adresáře obnoveného serveru. Jinak vytvoříme prázdný adresář

```
mkdir pg_xlog
```

Nastartujeme server. Po úspěšném startu by měl být soubor `recovery.conf` přejmenován na **recovery.done** a v logu bychom měli najít záznam:

```
LOG:  archive recovery complete
LOG:  database system is ready to accept connections
```

PostgreSQL implicitně³⁷ provádí obnovu do okamžiku, ke kterému dohlédá poslední validní segment transakčního logu. Záznam v logu referuje o postupu hledání segmentů:

```
LOG:  restored log file "000000010000000000000007" from archive
LOG:  restored log file "000000010000000000000008" from archive
LOG:  restored log file "000000010000000000000009" from archive
cp: cannot stat `/var/backup/xlogs/0000000100000000000000A':
No such file or directory LOG:  could not open file
"pg_xlog/000000010000000000000000A": No such file or directory
```

Jednorázové

Jednorázovým zálohováním se miní vytvoření klonu běžící databáze. Základem této metody je časově omezená replikace záznamů transakčních logů. Výhodou je jednoduchost použití – rychlost zálohování a obnovy ze zálohy je limitována rychlostí IO.

Konfigurace

Tato metoda vyžaduje úpravu konfiguračního souboru a uživatele s oprávněním `REPLICATION` a přístupem k fiktivní databázi `replication` (přístup se povoluje v souboru `pg_hba.conf`).

```
wal_level = archive
max_wal_senders = 1
```

```
# v případě větších db zvýšit
wal_keep_segments = 100
```

Úprava `pg_hba.conf`:

local	replication	backup	md5
-------	-------------	--------	-----

Vytvoření uživatele `backup`:

```
CREATE ROLE backup LOGIN REPLICATION;
ALTER ROLE backup PASSWORD 'heslo';
```

Tato změna konfigurace vyžaduje restart databáze.

Vlastní zálohování

Spustíme příkaz `pg_basebackup`, kde uvedeme adresář, kde chceme mít uložený klon.

```
[pavel@diana ~]$ /usr/local/pgsql91/bin/pg_basebackup -D \
zaloha9 -U backup -v -P -x -c fast
Password:
xlog start point: 0/21000020
50386/50386 kB (100%), 1/1 tablespace

xlog end point: 0/21000094
pg_basebackup: base backup completed
```

Obnova ze zálohy

Obsah adresáře zálohy zkopírujeme do adresáře clusteru PostgreSQL a nastartujeme server. Pozor - vlastníkem souborů bude uživatel, pod kterým byl spuštěn `pg_basebackup`, což pravděpodobně nebude uživatel `postgres`, a proto je nutné nejprve hromadně změnit vlastníka souborů.

Fyzická replikace

Potřebujeme opět uživatele s právem `REPLICATION` a přístupem k db `replication`. Základem sekundárního (ro) serveru je klon primárního serveru (rw).

Úpravy konfigurace – master

```
wal_level = hot_standby
max_wal_senders = 1
```

```
# v případě větších db zvýšit
wal_keep_segments = 100
```

Úpravy konfigurace – slave³⁸

Pozor, po naklonování se `slave` nikdy nesmí spustit jako samostatný server. Pokud možno, klonujte s konfigurací `wal_level = hot_standby` na `masteru`.

```
hot_standby = on
hot_standby_feedback = on# pro zajištění pomalých dotazů na sl.
```

Vytvořte soubor **recovery.conf**, který uložíte do cluster adresáře serveru `slave`:

```
standby_mode='on'
primary_conninfo='host=localhost user=backup password=heslo'
```

Před startem repliky vymažte log a **pid** file. Po startu by měl log obsahovat záznam:

³⁸ Upravuje se `postgresql.conf` na počítači použitým jako `slave`. Dále se zde musí vytvořit konfigurační soubor `recovery.conf`.

```
LOG:  entering standby mode
LOG:  consistent recovery state reached at 0/300014C
LOG:  record with zero length at 0/300014C
LOG:  database system is ready to accept read only connections
LOG:  streaming replication successfully connected to primary
```

Po startu je `slave` v **read only** režimu. Signálem jej lze přepnout do role `master`. Pozor – tato **změna je nevratná**. Nový `slave` se vytvoří kopií nového `masteru`.

```
su postgres
pg_ctl -D /usr/local/pgsql/data/repl/ promote
```

synchronizaci lze na každé replikovaném serveru dočasně blokovat – a během té doby můžeme provést fyzickou zálohu (zkopírování datového adresáře – *full backup*). K řízení replikace slouží následující funkce:

<code>pg_xlog_replay_pause()</code>	pozastaví replikaci
<code>pg_xlog_replay_resume()</code>	obnoví replikaci
<code>pg_is_xlog_replay_paused()</code>	vrátí true v případě pozastavené replikace

Využití systémového katalogu

V PostgreSQL jsou všechna data potřebná pro provoz databáze uložena v systémových tabulkách. Orientace v systémových tabulkách a pohledech není jednoduchá. Lze ovšem využít jeden trik – většina dotazů do těchto objektů je pokryta příkazy v **psql**. A pokud se `psql` pustí s parametrem `-E`, tak dojde k zobrazení všech SQL příkazů, které se posílají do DB – a tedy i dotazů do systémového katalogu.

```
bash-4.2$ psql -E postgres
psql (9.3devel)
Type "help" for help.
```

```
postgres=# \l
***** QUERY *****
SELECT d.datname as "Name",
       pg_catalog.pg_get_userbyid(d.datdba) as "Owner",
       pg_catalog.pg_encoding_to_char(d.encoding) as "Encoding",
       d.datcollate as "Collate",
       d.datctype as "Ctype",
       pg_catalog.array_to_string(d.datacl, E'\n') AS "privileges"
FROM pg_catalog.pg_database d
ORDER BY 1;
*****
```

Běžně se v systémovém katalogu dohledává seznam tabulek, databází, uživatelů. Systémový katalog můžeme využít k zobrazení tabulek obsahující určité sloupce nebo uložených procedur, které obsahují hledaný řetězec.

Zobrazí tabulky obsahující hledaný sloupec:

```
SELECT attrelid::regclass
FROM   pg_catalog.pg_attribute a
WHERE  a.attname = 'jmeno' AND NOT a.attisdropped;
```

Místo systémového katalogu lze použít standardizované `information_schema`:

```
SELECT table_name
FROM   information_schema.columns
WHERE  column_name = 'jmeno';
```

Zobrazí funkce, které ve zdrojovém kódu obsahují hledaný řetězec:

```
SELECT oid::regprocedure
FROM   pg_proc
WHERE  prosrc ILIKE '%hello%';
```

³⁵ Vždy při naplnění segmentu transakčního logu nebo vypršení časového intervalu PostgreSQL volá `archive command`, jehož úkolem je zajistit zápis segmentu na bezpečné médium.

³⁶ Transakční logy lze velice dobře komprimovat – např. asynchronně (viz *BARMAN*)

³⁷ Nastavením `recovery_target_time` v `recovery.conf` lze určit okamžik, kdy se má s přehráváním transakčních logů skončit – například před okamžik, kdy došlo k odstranění důležitých dat.

Vyhledávání v textu

Fulltext

Fulltext umožňuje case insensitive vyhledávání slov (případně prefixů slov) v textu. S drobnými úpravami lze vyhledávat *lexémy* a nebo lze při vyhledávání ignorovat diakritiku. Každé slovo se při fulltextovém zpracování definovaným způsobem transformuje. Seznam těchto transformací (každá třída slov může mít jinou transformaci) pro určitý jazyk nazýváme konfigurací. Nejjednodušší konfigurací je konfigurace `simple`. Pro urychlení fulltextového vyhledávání potřebujeme *fulltextový* index (*Gist*, *GIN* *funkcionální* index)

```
CREATE INDEX ON obce
    USING gist ((to_tsvector('simple',navez)));
```

S tímto indexem lze efektivně fulltextově vyhledávat:

```
SELECT *
FROM obce
WHERE to_tsvector('simple',navez) @@39
    to_tsquery('simple','skal:40 & !česká');
```

Vlastní konfigurace se vytvářejí kopií a následnou úpravou některé stávající. Následující konfigurace zahrnuje použití funkce *unaccent*⁴¹.

```
CREATE TEXT SEARCH CONFIGURATION simple_unaccent
    ( COPY = simple );
ALTER TEXT SEARCH CONFIGURATION simple_unaccent
    ALTER MAPPING FOR hword, hword_part, word
    WITH unaccent42, simple;

CREATE INDEX ON obce USING gist
    ((to_tsvector('simple_unaccent',navez)));

SELECT *
FROM obce
WHERE to_tsvector('simple_unaccent',navez) @@
    to_tsquery('simple_unaccent','svaty');
```

LIKE

Predikát s `LIKE`, kdy je žolík `'%'` za písmeny, lze urychlit vytvořením indexu s volbou `varchar_pattern_ops`.

```
CREATE INDEX ON obce(navez varchar_pattern_ops);
```

Dotaz jako je následující⁴³, pak dokáže využít index.

```
SELECT *
FROM obce
WHERE navez LIKE 'S%'
```

K optimalizaci dotazů s predikátem `LIKE` (i `LIKE`) lze použít extenzi *pg_trgm*, která obsahuje podporu pro *trigramový* index (index nad množinou tří písmenných kombinací z řetězce). Index je nutné vytvořit s volbou `gist_trgm_ops` nebo `gin_trgm_ops`

```
CREATE INDEX ON obce
    USING GiST (navez gist_trgm_ops)
```

Tento typ indexu dokáže podporovat i dotazy, kde se hledá libovolný umístěný podřetězec:

```
SELECT *
FROM obce
WHERE navez ILIKE 'Ska%'
```

Regulární výrazy

Pro vyhledávání lze použít i regulární výrazy – operátor `~` nebo `~*`⁴⁴.

```
SELECT navez
FROM obce
WHERE navez ~ '^Sk[aáo]';
```

Také vyhledávání prostřednictvím regulárních výrazů může být urychleno trigramovým indexem⁴⁵.

Barman

Barman⁴⁶ je aplikační nadstavba nad vestavěným replikačním a zálohovacím systémem v PostgreSQL umožňující hromadnou administraci zálohování, evidenci a management záloh (komprimaci), řízení retenční politiky a samozřejmě obnovu ze zálohy do určeného adresáře.

Konfigurace

Je požadována obousměrná ssh spojení mezi zálohovaným a zálohovacím serverem. Na obouh serverech musí být nainstalovaná stejná verze Postgresu⁴⁷, Python a pycpg2 a rsync.

```
# zálohovaný systém @10.0.0.4
su - postgres
ssh-keygen -t rsa -N "" -f ~/.ssh/id_rsa
ssh-copy-id -i ~/.ssh/id_rsa.pub barman@10.0.0.8
# ssh barman@10.0.0.8
```

```
# zálohovací systém @10.0.0.8
su - barman
ssh-keygen -t rsa -N "" -f ~/.ssh/id_rsa
ssh-copy-id -i ~/.ssh/id_rsa.pub postgres@10.0.0.8
# ssh postgres@10.0.0.4
```

Dále musí být umožněn přístup k zálohované databázi uživateli postgres z zálohovacího serveru (úprava *pg_hba.conf*). Následující příkaz musí fungovat

```
[barman]$ psql -c 'SELECT version()' -U postgres -h 10.0.0.4
```

S právy roota se na zálohovacím serveru vytvoří adresář pro uložení záloh:

```
barman$ sudo mkdir /var/lib/barman
barman$ sudo chown barman:barman /var/lib/barman
```

Vlastní konfigurace je v */etc/barman/barman.conf* – nutné přidat popis zálohovaného serveru⁴⁸:

```
[dbserver01]
description = "PostgreSQL Database Server 01"
ssh_command = ssh postgres@10.0.0.4
conninfo = host=10.0.0.4 user=postgres
minimum_redundancy = 1
```

Dále je nutné nakonfigurovat zálohovaný PostgreSQL⁴⁹:

```
wal_level = 'archive' # For PostgreSQL >= 9.0
archive_mode = on
archive_command = 'rsync
    -a %p barman@backup:dbserver01/incoming50/%f'
```

Základní příkazy

Verifikace konfigurace

```
barman check dbserver01
```

Vytvoření kompletní zálohy serveru (všech serverů)

```
barman backup [--immediate-checkpoint] ( all | dbserver01 )
```

Výpis seznamu záloh

```
barman list-backup (all | dbserver0 )
```

Lokální⁵¹ obnova ze zálohy

```
barman recover dbserver01 20140419T23552452 ~/xxx
```

Informace k záloze

```
barman show-backup dbserver01 latest
```

Explicitní odstranění zálohy

```
barman delete dbserver01 oldest
```

Repmgr

*repmgr*⁵³ je aplikační nadstavba nad vestavěnou replikací v PostgreSQL zjednodušující management a monitoring clusteru master/multi slave implementující failover. Doporučuje se symetrická architektura – každý uzel může dlouhodobě převzít roli mastera⁵⁴.

Konfigurace

Repmgr vyžaduje obousměrné ssh spojení bez nutnosti zadávání hesla pro uživatele postgres na všech serverech zapojených do clusteru (nastavení viz konfigurace Barmanu). Dále repmgr musí být nainstalován na všech uzlech

Server sloužící ve výchozí pozici jako master musí být nakonfigurován jako master hot-standby stream replikace (v *postgresql.conf*):

```
listen_addresses='*'
wal_level = 'hot_standby'
archive_mode = on
archive_command = 'cd .' # just does nothing
max_wal_senders = 10
wal_keep_segments = 5000      # 80 GB required on pg_xlog
hot_standby = on
```

Vytvoříme uživatele repmgr správem REPLICATION a SUPERUSER a povolíme mu přístup z IP používaných pro provoz slave serverů. Čistě z praktických důvodů (není nezbytně nutné) vytvoříme aplikačního uživatele repmgr na všech uzlech (*useradd*). Databázový

⁴⁹ *postgresql.conf*

⁵⁰ musí souhlasit s položkou `incoming_wals_directory` zobrazené příkazem `barman show-server dbserver01`

⁵¹ s volbou `--remote-ssh-command` `COMMAND` lze obnovu provést na vzdáleném serveru. Přepínačem `--target-time` `TARGET_TIME` lze nastavit bod obnovy.

⁵² Zálohu lze také specifikovat klíčovými slovy „oldest“ nebo „latest“

⁵³ Pokud ji nenaleznete ve své distribuci, pak se překládá a instaluje jako *contrib* modul Postgresu.

Dále *pg_ctl* a *pg_config* musí být v `PATH`.

⁵⁴ I z toho důvodu se nedoporučuje používat v názvu instance slova master nebo slave.

³⁹ Fulltextový operátor

⁴⁰ Hledání prefixu „skal“.

⁴¹ Vyžaduje extenzi *unaccent*.

⁴² Každé slovo se transformuje slovníkem – slovník *unaccent* odstraňuje diakritiku, slovník *simple* nedělá nic -pro každou třídu slov můžeme mít definovanou posloupnost slovníků.

⁴³ Varchar_pattern_ops indexem je podporován pouze LIKE, který je case sensitive (nikoliv case insensitive ILIKE).

⁴⁴ Case insensitive varianta

⁴⁵ V PostgreSQL 9.3 – za předpokladu, že je určen kompletní trigram (tři znaky)

⁴⁶ Barman je OS aplikace napsaná v Pythonu ke stažení z <http://www.pgbarman.org>

⁴⁷ Barman sám Postgres nepoužívá, ale Postgres je nutný pro start lokálně obnovené databáze.

⁴⁸ poté by již měl být funkční příkaz `barman check dbserver01`

uživatel repmgr musí mít přístup k explicitně vytvořené databázi repmgr na masteru i lokálně ze všech uzlů.

```
psql
-c "CREATE ROLE repmgr LOGIN SUPERUSER REPLICATION" postgres
```

v *pg_hba.conf*

host	repmgr	repmgr	10.0.0.8/32	trust
host	repmgr	repmgr	10.0.0.4/32	trust
host	replication	repmgr	10.0.0.8/32	trust

Ze slave bych se měl dokázat připojit k masteru jako uživatel repmgr

```
psql -U repmgr -h 10.0.0.4 repmgr
```

Následující příkaz vytvoří klon (parametr -R obsahuje uživatele pro rsync, -U uživatele databáze):

```
repmgr -D /usr/local/pgsql/data -d repmgr -p 5432 -U repmgr
-R postgres --verbose standby clone 10.0.0.4
```

V každém uzlu se vytvoří konfigurační soubor */usr/local/pgsql/repmgr/repmgr.conf*:

```
cluster=test
node=1
node_name=dell
conninfo='host=10.0.0.4 user=repmgr dbname=repmgr'
pg_bindir=/usr/local/pgsql/bin
master_response_timeout=60
reconnect_attempts=6
reconnect_interval=10
failover=automatic
priority=-1
promote_command='repmgr standby promote
-f '/usr/local/pgsql/repmgr/repmgr.conf'
follow_command='repmgr standby follow
-f '/usr/local/pgsql/repmgr/repmgr.conf -w'
```

registrace konfigurace na masteru a start repmgrd:

```
repmgr -f /usr/local/pgsql/repmgr/repmgr.conf
--verbose master register
repmgrd -f /usr/local/pgsql/repmgr/repmgr.conf --verbose
--monitoring-history > /usr/local/pgsql/repmgr/repmgr.log 2>&1
```

a totéž na slave

```
cluster=test
node=2
node_name=lenovo
conninfo='host=10.0.0.8 user=repmgr dbname=repmgr'
pg_bindir=/usr/local/pgsql/bin
master_response_timeout=60
reconnect_attempts=6
reconnect_interval=10
failover=automatic
priority=-1
promote_command='repmgr standby promote
-f '/usr/local/pgsql/repmgr/repmgr.conf'
follow_command='repmgr standby follow
-f '/usr/local/pgsql/repmgr/repmgr.conf -w'
```

Dále je nutné nastartovat repmgr démona, který zároveň zaregistruje slave

```
repmgrd -f /usr/local/pgsql/repmgr/repmgr.conf --verbose
--monitoring-history > /usr/local/pgsql/repmgr/repmgr.log 2>&1
```

Podpora failover vyžaduje nainstalovanou extenzi *repmgr_func*.

```
psql -U repmgr repmgr <
/usr/local/pgsql/share/contrib/repmgr_funcs.sql
```

a preload této extenze (v *postgresql.conf*)

```
shared_preload_libraries = 'repmgr_funcs'
```

Použití

Při správné konfiguraci by následující příkazy měly vypsat status uzlů v clusteru:

```
psql -x -d repmgr -c "SELECT * FROM repmgr_test.repl_status"
repmgr -f /usr/local/pgsql/repmgr/repmgr.conf cluster show
```

Spuštěním příkazu repmgr na příslušném uzlu můžeme dosáhnout:

povýšení slave na master

```
repmgr -f /usr/local/pgsql/repmgr/repmgr.conf standby promote
```

přesměrování slave na nového mastera⁵⁵

```
repmgr -f /usr/local/pgsql/repmgr/repmgr.conf standby follow
```

vy nucené klonování – změna mastera na slave

```
repmgr -D /usr/local/pgsql/data -d repmgr -p 5432 -U repmgr
-R postgres --force --verbose standby clone 10.0.0.4
```

PgBouncer

PgBouncer vytváří cache (*pool*) spojení do PostgreSQL. Jedno nebo více spojení do konkrétní databáze pod konkrétním uživatelem se v *PgBounceru* označuje jako pool⁵⁶. Specifikem *PgBounceru* je fiktivní databáze *pgbouncer* umožňující základní administraci a monitoring.

Konfigurace

Vytvořte si systémový účet *pgbouncer*. Tento účet bude mít jako jediný přístup k hashům hesel databázových účtů a poběží pod ním aplikace *pgbouncer*.

V tomto adresáři je také skript *mkauth.py*⁵⁷, který zkopíruje md5 hashe hesel účtů v *postgresu* do zadaného souboru. Pro tyto účty je nutné nastavit (v *pg_hba.conf*) md5 ověřování.

```
su - postgres -c '/etc/pgbouncer/mkauth.py
/var/tmp/userlist.txt "host=localhost dbname=postgres"'
```

```
mv /var/tmp/userlist.txt /etc/pgbouncer/userlist.txt
chown pgbouncer:pgbouncer /etc/pgbouncer/userlist.txt
```

```
mkdir /var/log/pgbouncer
chown pgbouncer:pgbouncer /var/log/pgbouncer
mkdir /var/run/pgbouncer
chown pgbouncer:pgbouncer /var/run/pgbouncer
```

Do */etc/pgbouncer/pgbouncer.ini* zkopírovat minimální konfiguraci (s dynamickými pooly):

```
[databases]
* = host=10.0.0.4 port=5434

[pgbouncer]
logfile = /var/log/pgbouncer/pgbouncer.log
pidfile = /var/run/pgbouncer/pgbouncer.pid

listen_addr = 127.0.0.1
```

55 Pro správnou funkci je nutné alespoň 9.3 a v *recovery.conf* *recovery_target_timeline='latest'*

56 Počet otevřených spojení v poolu lze omezit. V případě nedostatku volných spojení *PgBouncer* umí požadavek o spojení podržet předdefinovanou dobu.

57 aplikace vyžaduje *psycopy2*

```
listen_port = 6432

auth_type = md5
auth_file = /etc/pgbouncer/userlist.txt
```

```
admin_users = postgres
stats_users = pavel, postgres
```

```
pool_mode = session
server_reset_query = DISCARD ALL
```

```
max_client_conn = 100
default_pool_size = 20
```

```
server_lifetime = 1200
server_idle_timeout = 60
server_connect_timeout = 15
server_login_retry = 15
client_idle_timeout = 3600
autodbg_idle_timeout = 3600
```

```
tcp_keepalive = 1
```

Pod uživatelem *pgbouncer* spustíme aplikaci *pgbouncer*:

```
su - pgbouncer
pgbouncer /etc/pgbouncer/pgbouncer.ini
```

Nyní se můžeme přihlásit k libovolné databázi na portu 6432 nebo k databázi *pgbouncer* na též portu.

```
psql -U postgres -p 6432 postgres
```

Monitoring

Databáze *pgbouncer* umožňuje přístup ke statistikám a základní administraci. Pozor k této databázi přistupujeme pomocí *psql*, ale nepoužíváme SQL (příkaz *SHOW HELP*, *SHOW STATS*):

```
psql -U postgres -p 6432 pgbouncer -c "SHOW STATS"
```

Nerelační datové typy

S použitím typů *HStore*, *JSON*, *JSONB* a *XML* můžeme emulovat nerelační databáze. V *JSONB* jsou data uložena binárně, ostatní typy se ukládá jako text⁵⁸. *XML* a *JSON* se používají primárně pro uložení a výstup dat ve formátu, který je průmyslovým standardem. *HStore* a *JSONB* pak umožňují manipulaci a vyhledávání v datech v těchto formátech uložených v databázi.

HStore

Typ *HStore* je emulace hash array. Lze jej použít coby efektivnější náhradu *EAV*⁵⁹ a je podporován *GiST* a *GIN* indexy. Ukládané hodnoty mohou být pouze texty nebo čísla, které se ukládají vždy v textovém formátu.

```
CREATE EXTENSION hstore;
CREATE TABLE lide(rc numeric PRIMARY KEY, ostatni hstore);
INSERT INTO lide VALUES(7307150888, 'jmeno=>Pavel,
prijmeni=>stěhule');
CREATE INDEX ON lide USING gist (ostatni);
```

58 Ve většině případů bez negativního vlivu na výkon.

59 Entitý Attribute Value model

Inhouse školení PostgreSQL

Vyberte si z naší nabídky jednodenní školení pro začátečníky i pokročilé. Z těchto jednodenních školení je možné (na základě poptávky) kombinovat vícedenní školení. Tato školení vede a organizuje [Pavel Stěhule](#), který se také podílí na vývoji PostgreSQL a je dlouholetým uživatelem a propagátorem této databáze. Již pro tři Vaše zaměstnance jsou tato školení levnější (bez ohledu na úsporu času) než školení organizovaná počítačovými školami. Pokud byste měli zájem o in-house školení nebo se chcete informovat o nejbližším termínu, obraťte se, prosím, přímo na Pavla Stěhuleho ([kontakt](#)).

Cena za jeden den in-house školení je 12 tis. Kč (včetně DPH) pro 4 osob plus příplatek 1000 Kč za každého další účastníka (20 tis za max 12 osob). (veřejná školení se vypisují na základě poptávky více než 8 účastníků, cena je 4000 Kč za osobu). Pro bližší informace ohledně nejbližších termínů kontaktujte [Pavla Stěhuleho](#) pavel.stehule@gmail.com, mob: 724 191 000. V případě školení mimo Prahu jsou účtovány cestovní výdaje. V ceně jsou vytisknuté školící materiály.

Všeobecné základy

Školení je určeno začátečníkům a středně pokročilým uživatelům, kteří se během osmi hodinového kurzu dozvědí vše potřebné k efektivnímu používání tohoto databázového systému. K dispozici jsou [školící materiály](#). Školení předpokládá obecné znalosti SQL a IT problematiky u posluchačů (např. není vysvětlován pojem databáze, relace, SQL DML DDL příkazy atd). Účastníci školení by měli získat přehled o možnostech PostgreSQL a měli by být následně schopni efektivně používat PostgreSQL.

- Podpora PostgreSQL na internetu
- Instalace ve zkratce
- Porovnání o.s. SQL RDBMS Firebird, PostgreSQL, MySQL a SQLite
- Minimální požadavky na databázi, ACID kritéria
- Charakteristické prvky PostgreSQL MGA, TOAST
- Datové typy bez limitů - TOAST
- Spolehlivost a výkon - WAL
- Nutné zlo, příkaz VACUUM
- Rozšiřitelnost
- Základní příkazy pro správu PostgreSQL
- Export, import dat
- Efektivní SQL, indexy, optimalizace dotazů
- Funkce generate_series

Programování v PL/pgSQL

Tento kurz je určen především vývojářům, kteří chtějí zvládnout efektivní vývoj nad PostgreSQL, který není bez uložených procedur myslitelný. PostgreSQL podporuje jak SQL procedury tak tzv. externí procedury. K dispozici je několik jazyků od SQL až po PL/Perl. Každý jazyk nabízí jiné možnosti a po absolvování kurzu by se vývojář měl dokázat rozhodnout pro jeden konkrétní jazyk, který pro dané zadání nabízí největší možnosti. Školení je osmi hodinové - důraz je kladen na procvičení vyložené látky. K dispozici jsou [podklady](#) pro toto školení.

- Uložené procedury, kdy a proč
- Inline procedury v SQL
- Úvod do PL/pgSQL
- Syntaxe příkazu CREATE FUNCTION
- Blokovaný diagram PL/pgSQL

- Příkazy PL/pgSQL
- Dynamické SQL
- Použití dočasných tabulek v PL/pgSQL
- Triggery v PL/pgSQL
- Typy pro vývoj PL/pgSQL
- Příloha, Transakce

Administrace

Z názvu je patrné, že toto školení je určené jak začínajícím tak i pokročilým administrátorům, které připravuje na každodenní správu PostgreSQL databází. Po absolvování kurzu by mělo být absolventům jasné, proč se provádí určité činnosti (pravidelné nebo nahodilé), a na co, při správě PostgreSQL, klást důraz. Školení je šesti hodinové. K dispozici jsou [podklady](#) pro toto školení.

- Omezení přístupu k databázi
- Údržba databáze
- Správa uživatelů
- Export, import dat
- Zálohování, obnova databáze
- Konfigurace databáze
- Monitorování databáze
- Instalace doplňků
- Postup při přechodu na novou verzi

High performance

Tento kurz je určen pokročilejším uživatelům a vývojářům, kteří používají PostgreSQL. Zabývá se obecnější otázkou výkonu datově orientovaných aplikací postavených nad relační databází. K dispozici jsou [podklady](#) pro toto školení.

- Základní faktory ovlivňující výkon databáze
- Aplikační vrstvy
- CPU, RAM, IO, NET
- Konfigurace PostgreSQL
- Identifikace hrdel
- Použití cache a materializovaných pohledů
- Použití indexů a psaní index friendly aplikací
- Cost based optimizer, projevy chyb v odhadech a jejich řešení
- Monitoring
- Doporučení

Zálohování a replikace

Toto **přípravované** školení je určeno pokročilejším uživatelům PostgreSQL. V rámci školení se účastníci seznámí s možnostmi zálohování a také si prakticky vyzkouší konfiguraci vestavěné replikace. **Školení lze objednat na jaře 2014.**

- Úvod - zálohování, replikace
- Konfigurace exportu transakčního logu
- pg_basebackup

- Barman a repmgr
- Konfigurace vestavěné replikace
- Kombinace replikace a exportu transakčního logu

Základy SQL

Toto školení je určeno především začátečníkům (z ne IT oborů), kteří chtějí využít SQL pro tvorbu vlastních reportů. Během kurzu jsou vysvětleny základní pojmy z teorie a praxe relačních databází. Dvě třetiny času osmihodinového školení je věnováno procvičování dotazů (od nejjednodušších ke středně složitým), tak aby po absolventu školení dokázal samostatně (pro svou praxi) získávat zajímavá data z SQL databázi. K dispozici jsou [školící materiály](#).

- Příkaz SELECT - spojování tabulek, filtrování, projekce, řazení
- Ostatní databázové objekty - sekvence, pohledy, indexy
- Zajištění referenční a doménové integrity - primární a cizí klíče, domény, trigger

Autor: [Pavel Stěhule](#)

Kontakt: pavel.stehule@gmail.com, 724 191 000

Profil: [cz.linkedin.com/in/stehule/](https://www.linkedin.com/in/stehule/) stackoverflow.com/users/176171/pavel-stehule
<http://www.root.cz/autori/pavel-stehule/>

Inhouse školení PostgreSQL – instalace, konfigurace, používání a administrace

Inhouse školení PL/pgSQL – vývoj uložených procedur

Inhouse i veřejné školení SQL

Konzultace, konfigurace PostgreSQL, audit produkčních PostgreSQL serverů

Komerční podpora PostgreSQL