

PostgreSQL ve verzi 13-16

| |
|---------------------------------|
| createdb jménodb |
| Vytvoří novou databázi |
| dropdb jménodb |
| odstraní existující databázi |
| psql jménodb ¹ |
| spustí SQL konzoli |
| 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.

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

Konfigurace konzole

| |
|---|
| Soubor <i>.psqlrc</i> |
| <pre>\set QUIET on \setenv PAGER less² \setenv LESS '-iMSx4 -RSFX -e' \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 off³ \set QUIET off</pre> |

- ¹ Nastavením systémové proměnné PGDATABASE lze určit implicitní databázi
- ² Alternativním pagerem může být pager *pspg*. <https://github.com/okbob/pspg>
- ³ Doporučeno pro produkci – vynucuje potvrzení změn explicitním COMMITem. Po vypnutí *autocommitu* se psql bude chovat podobně jako konzole *Oracle*.

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 databáze⁴.

| | |
|---------------------------------|---|
| <code>-f</code> | specifikuje cílový soubor |
| <code>-a</code> | exportuje pouze data |
| <code>-s</code> | exportuje pouze definice |
| <code>-c</code> | odstraní objekty před jejich importem |
| <code>-C</code> | vloží příkaz pro vytvoření nové databáze |
| <code>-t</code> | exportuje pouze jmenovanou tabulku |
| <code>-T</code> | neexportuje uvedenou tabulku |
| <code>--disable-triggers</code> | během importu blokuje trigger |
| <code>--inserts</code> | generuje příkazy INSERT místo COPY |
| <code>-Fc</code> | záloha je průběžně komprimovaná s dodatečnými meta informacemi ⁵ |

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).

| |
|---|
| <code>shared_buffers= 2GB</code> |
| velikost paměti pro uložení datových stránek (1/5..1/3 RAM ⁶) |
| <code>work_mem = 10MB</code> |
| limit paměti pro běžnou manipulaci s daty (10..100MB) |
| <code>maintenance_work_mem = 200MB</code> |
| limit paměti pro údržbu (100MB ..) |
| <code>effective_cache_size = 6GB</code> |
| odhad objemu dat cache (2/3 RAM) |
| <code>max_connections = 100</code> |

max počet přihlášených uživatelů (často zbytečně vysoké)

- ⁴ 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.
- ⁵ Pro obnovu je nutné použít `pg_restore`, obnovit lze i každou vybranou tabulku.
- ⁶ Doporučené hodnoty platí pro tzv dedikovaný server – tj počítač, který je vyhrazen primárně pro provoz databáze s 8GB RAM.

Mělo by platit⁷:

| |
|--|
| <code>shared_buffers + 2 * work_mem * max_connection <= 2/3 RAM</code> |
| <code>shared_buffers + 2 * maintenance_work_mem <= 1/2 RAM</code> |
| <code>max_connections <= 10 * (počet_CPU)</code> |
| Pokud dochází k intenzivnímu zápisu, může mít smysl zvýšit hodnotu <i>max_wal_size</i> . Pokud velikost transakčního logu přesáhne tuto hranici, dojde k provedení CHECKPOINTU. Vyšší hodnota znamená nižší frekvenci checkpointů a naopak. Výchozí hodnota 1GB je pro obvyklé použití dostatečná. |
| <code>max_wal_size = 1GB</code> |

Po CHECKPOINTU lze zahodit transakční logy vztahené k času před CHECKPOINTem. Za optimální frekvenci CHECKPOINTů se považuje 5 – 15 min.

```
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;
```

LIMIT

Pomocí klauzule LIMIT můžeme omezit počet řádků výsledné relace. Kromě proprietární klauzule LIMIT je podporován ANSI `FETCH FIRST`. U tohoto zápisu můžeme použít frázi `WITH TIES`, která zajistí doplnění výsledku o řádky, které mají stejnou hodnotu ve výrazu `ORDER BY` jako poslední řádek určený `FETCH FIRST n`. Pokud použijete `FETCH FIRST` a `OFFSET` dohromady, pak klauzule `OFFSET` musí být uvedena před klauzulí `FETCH FIRST`:

- ⁷ Jedná se o orientační hodnoty určené pro počáteční konfiguraci “typického použití” databáze.
- 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 * FROM obce
ORDER BY pocet_muzu + pocet_zen
OFFSET 0 FETCH FIRST 10 ROWS WITH TIES
```

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:

```
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 zpracovala 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 množinou

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(0.510) 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 obých 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 relací¹¹ JOIN

Příkaz **JOIN** spojuje relace (tabulky) vedle sebe a to na základě stejných hodnot v jednom nebo více atributech (sloupcích). Každé spojení specifikuje dvě relace (spojkou je klíčové slovo **JOIN**) a podmínku, která určuje, jak se tyto relace budou spojoovat (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 (zaměstnanec) – 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 NULL**¹². 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 tabulka¹³.

- 11 Tabulka je relací. Výsledek SQL dotazu je relací. Tudíž příkaz **SELECT** můžeme aplikovat na tabulku nebo i na výsledek jiného příkazu **SELECT**.
- 12 Pro hodnotu **NULL** není možné použít operátor **=**.
- 13 **SELECT** ze **SELECTu**

I derivovanou tabulku lze spojoovat 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 **LATERAL** 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;
```

LATERAL join lze využít pro efektivní provedení úlohy nalezení top **N** pro každou skupinu:

```
SELECT *
FROM okresy,
      LATERAL (SELECT *
               FROM obce
               WHERE obce.okres_id = okresy.id
               ORDER BY pocet_obyvatel DESC
               LIMIT 3);
```

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:

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

```
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
```

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 - lag(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 w14 AS (PARTITION BY dbname,
ORDER BY time)
```

Počítání klouzavých průměrů, součtů

Díky analytickým funkcím není komplikované počítat klouzavou agregovanou hodnotu. Rozsah (okno) může být určeno klíčovými slovy: RANGE (maximálním rozdílem hodnot¹⁵), ROWS (počtem řádků) a GROUPS (počet unikátních hodnot):

```
SELECT ti,
       count(*) OVER (ORDER BY ti
                     RANGE BETWEEN '30min' PRECEDING
                               AND '30min' FOLLOWING
                     FROM data;
```

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 upsert17 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 rekurze):

```
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;
```

Za CTE výraz můžeme přidat klauzuli *CYCLE*¹⁸ se seznamem sloupců, na kterých se má detekovat cyklus (nalezení duplicitních hodnot), klauzulí *SET*, ve které můžeme nastavit libovolný sloupec při detekci cyklu, a klauzulí *USING*, kterou definujeme název sloupce, ve které se drží historie průchodů:

```
WITH RECURSIVE dest AS (
  CYCLE departure, arrival SET is_cycle USING path
SELECT * FROM dest
```

Obdobně jako klauzulí *CYCLE* vytváříme hodnotu použitou k detekci cyklů, tak klauzulemi *SEARCH DEPTH FIRST* a *SEARCH BREADTH FIRST* si můžeme vytvořit hodnotu, a pokud podle ní budeme řadit, tak dostaneme výstup odpovídající prohledávání grafu do hloubky nebo do šířky:

```
WITH RECURSIVE dest AS (
  SEARCH BREADTH FIRST BY departure, arrival SET ordercol
SELECT * FROM dest ORDER BY ordercol
```

GROUPING SETS

Klauzule *GROUPING SETS* zajistí vícenásobnou agregaci podle daného seznamu. klauzule *CUBE* vytvoří všechny kombinace z daného seznamu, klauzule *ROLLUP*¹⁹ vytvoří agregace implementující drilování dat podle zadaného seznamu.

```
SELECT a,b, sum(x) FROM foo GROUP BY20 GROUPING SETS(a,b,())
```

je ekvivalentem dotazu

```
SELECT a, NULL, sum(x) FROM foo GROUP BY a
UNION ALL SELECT NULL, b, sum(x) FROM foo GROUP BY b
UNION ALL SELECT NULL, NULL, sum(x)
```

CUBE a *ROLLUP* se převádějí na *GROUPING SETS*:

```
CUBE(a, b) GROUPING SETS((a,b), a, b, ())
ROLLUP(a, b) GROUPING SETS((a,b), a, ())
```

Zobrazí prodeje podle lokality a názvu, podle lokality a prodeje celkem:

```
SELECT lokalita, nazev, sum(prodej)
FROM data_prodeje
GROUP BY ROLLUP(lokality, nazev)
```

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 ctid21 <> (SELECT ctid
                 FROM lidi
                 WHERE prijmeni=l.prijmeni
                 AND jmeno=l.jmeno
                 LIMIT 1);
```

²⁰ Od verze 14 je zde možné použít klíčové slovo *DISTINCT* pro redukci duplicitních agregáčních výrazů.

²¹ *Ctid* je fyzický identifikátor záznamu – v podstatě je to pozice záznamu v datovém souboru. Hodí se pouze pro některého úlohy, neboť po aktualizaci má záznam jiné *ctid*.

¹⁴ Funkce **lag** vrací předchozí hodnotu atributu v podmnožině.

¹⁵ Příklad obsahuje ukázkou sdílené definice okna (podmnožiny) **w**.

¹⁶ Vhodné pro typ *timestamp* - okno může být definováno např. 1h, 30min, ...

¹⁷ V případě, že záznam existuje, provede *UPDATE*, jinak *INSERT*.

¹⁸ Od verze 14

¹⁹ Implementace této klauzule je velice úspěšná

INSERT ON CONFLICT DO

Pomocí klauzule ON CONFLICT DO příkazu INSERT můžeme propojit příkazy INSERT a UPDATE do jednoho příkazu. Touto klauzulí se zavádí nový alias EXCLUDED pro kolizní vkládaný řádek.

Následující příkaz vloží obsah tabulky boo do tabulky foo. Neudělá nic, pokud se vložená hodnota x neliší od již existující:

```
INSERT INTO foo
SELECT * FROM boo
ON CONFLICT (id) DO
UPDATE foo SET x = excluded.x
WHERE x IS DISTINCT FROM excluded.x;
```

MERGE

Příkazem MERGE můžeme aplikovat obsah jedné tabulky (nebo výsledek dotazu) na druhou tabulku.

```
MERGE INTO foo
USING boo ON foo.id = boo.id
WHEN MATCHED AND boo.is_active THEN UPDATE SET c = boo.c
WHEN MATCHED AND NOT boo.is_active THEN DELETE
WHEN NOT MATCHED THEN INSERT VALUES(id, c, is_active);
```

Rozšiřující statistiky

Rozšiřující statistiky vytváříme příkazem CREATE STATISTICS. Aktuálně jsou podporovány vícestloupcové statistiky, a funkcionální statistiky²².

```
CREATE STATISTICS s123 ON (pocet_muzu + pocet_zen) FROM obce;
CREATE STATISTICS s224 ON pocet_muzu, pocet_zen FROM obce;
```

Často používané funkce a operátory

| | |
|-------------------------------|-------------------------------------|
| substring('ABC' FROM 1 FOR 2) | vrátí podřetězec |
| upper('ahoj') | převede text na velká písmena |
| lower('AH0J') | převede text na malá písmena |
| to_char(now(), 'DD.MM.YY') | formátuje datum |
| to_char(now(), 'HH24:MI:SS') | formátuje čas |
| trim(' aa ') | odstraní krajní mezery |
| EXTRACT(dow FROM now()) | vratí den v týdnu |
| EXTRACT(day FROM now()) | vratí den v měsíci |
| EXTRACT(month FROM now()) | vratí měsíc |
| EXTRACT(year FROM now()) | vratí rok |
| date_trunc('month', now()) | vratí nejbližší začátek období |
| COALESCE(a,b,c) | vratí první ne NULL hodnotu |
| array_lower(a, 1) | vratí spodní index pole nté dimenze |
| array_upper(a,1) | vratí horní index pole nté dimenze |
| random() | vratí pseudonáhodné číslo [0..1) |
| generate_series(l,h) | generuje posloupnost od l do h |
| array_to_string(a, ',') | serializuje pole |
| string_to_array(a, ',') | parsuje řetězec do pole |
| string_agg(a, ',') | agreguje do seznamu hodnot |
| concat('A',NULL,'B') | spojuje řetězce, ignoruje NULL |
| concat_ws(',','A',NULL,'B') | spojuje řetězce daným separátorem |
| start_with('Ahoj', 'Ah') | test prefixu |

| | |
|----------------------------|-------------------------------------|
| 'Hello' 'World' | spojuje řetězce (citlivé na NULL) |
| 10 IS NULL | test na NULL |
| 10 IS NOT NULL | negace testu na NULL |
| 10 IS DISTINCT FROM 20 | NULL bezpečný test na neekvivalenci |
| 10 IS NOT DISTINCT FROM 20 | NULL bezpečný test na ekvivalenci |

| | |
|----------------|---|
| row_number() | číslo řádku v podmnožině |
| rank() | pořadí v podmn. – nesouvislá řada |
| dense_rank() | pořadí v podmn. – souvislá řada |
| lag(a, 1, -1) | n-tá předchozí hodnota v podmn. |
| lead(a, 1, -1) | n-tá následující hodnota v podmn. |
| ntile(10) | vrací číslo podmnožiny z n skupin ²⁵ |

| | |
|-------------------|---|
| nazev ~ 'xx\$' | test na regulární výraz (citlivé na velikost písmen) |
| název ~* 'XX\$' | test na regulární výraz (bez ohledu na velikost písmen) |
| název ^@ 'prefix' | test prefixu |

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 dotazů²⁶, popřípadě monitorování událostí, které jsou obvykle spojeny s výkonnostními 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 1MB²⁷

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;
```

²⁵ Rozdělí množinu do n podobně velkých podmnožin. Lze použít pro orientační určení mediánu a kvantilů.

²⁶ Pro analýzu pomalých dotazů lze použít **pgFouine** nebo **pgbadger**. K monitorování lze použít extenze **auto_explain** (zapiše do logu prováděcí plán pomalého dotazu)

²⁷ Velké množství dočasných souborů může signalizovat nízkou hodnotu **work_mem**.

Využití tabulek²⁸ (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 nízkouřovňovou 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 transakcí²⁹.

```
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('Stehule', true);
SELECT novy_zamestnanec(jmeno => 'Stehule', 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 NOTICE30 '%', 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:

²² Statistky nad výrazy

²³ Funkcionální statistika od verze Postgres 14

²⁴ Vícestloupcová statistika v tomto případě obsahující ndistinct, korelace (funkční závislosti) a více dimenzionální MCV (Most Common Values)

²⁸ Pro indexy - pg_stat_user_indexes

²⁹ Používají se pouze subtransakce (implicitní) a to k zajištění ošetření zachycení výjimky.

³⁰ Zobrazí text na ladicí výstupu.


```
BEGIN
  IF EXISTS(SELECT 1
             FROM zamestnanci z
             WHERE z.jmeno = _jmeno
             FOR UPDATE31)
  THEN
    ...
  END IF;
END;
```

Ošetření chyby

PL/pgSQL vytváří subtransakci pro každý chráněný blok³² – 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 (případně ANYCOMPATIBLE, ANYCOMPATIBLEARRAY, ANYCOMPATIBLENONARRAY, ANYCOMPATIBLERANGE).

Generická funkce *myleast* by mohla vypadat následujícím způsobem:

```
CREATE OR REPLACE FUNCTION myleast(VARIADIC ANYCOMPATIBLEARRAY)
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 aktivoval³³ (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

Triggrem 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 triggery).

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. Modifikaci 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 záznamů³⁴.

³³ 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.

³⁴ AFTER triggery 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 triggery – musí se udržívat fronta nevyhodnocených AFTER triggerů.

```
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();
```

Statement triggerry

Ve verzi 10 už je možné prakticky používat statement triggerry díky tzv přechodovým (transition) tabulkám. V nich jsou k dispozici změny, které provedl příkaz, který nastartoval trigger. Přechodové tabulky jsou k dispozici pouze pro AFTER triggerry.

```
CREATE OR REPLACE FUNCTION audit()
RETURNS trigger AS $$
BEGIN
  INSERT INTO audit SELECT * FROM new_table;
  RETURN NULL;
END;
```

```
CREATE TRIGGER audit_trg
AFTER INSERT ON tab
REFERENCING NEW TABLE AS new_table
FOR EACH STATEMENT EXECUTE PROCEDURE audit();
```

Event triggerry

Event trigger je trigger, který je aktivován změnou systémového katalogu (např. přidáním tabulky, přidáním funkce, odstraněním uživatele). U těchto triggerů jsou následující události: ddl_comand_start, ddl_command_end³⁵, table_rewrite a sql_drop³⁶.

```
CREATE OR REPLACE FUNCTION drop_trg_func()
RETURNS event_trigger AS $$
DECLARE r RECORD;
BEGIN
  FOR r IN
    SELECT * FROM pg_event_trigger_dropped_objects()
  LOOP
    RAISE NOTICE 'dropped object: %', r;
  END LOOP;
END;
$$ LANGUAGE plpgsql;
```

```
CREATE EVENT TRIGGER on_drops_trg
ON SQL_DROP
EXECUTE PROCEDURE drop_trg_func();
```

k dispozici jsou automatické proměnné tg_tag a tg_event.

Procedury

Novinkou v PostgreSQL 11 jsou procedury aktivované příkazem CALL. Díky tomu, že nejsou volány příkazem SELECT, který musí běžet v rámci transakce, tak v proceduře můžeme³⁷ explicitně ukončovat transakce příkazy COMMIT a ROLLBACK.

```
CREATE OR REPLACE PROCEDURE foo(INOUT a int)
AS $$
```

³⁵ Změny katalogu vrací tabulková funkce pg_event_trigger_ddl_commands().

³⁶ Seznam rušených objektů vrací funkce pg_event_trigger_dropped_objects().

³⁷ Za předpokladu, že procedura není volána z funkce, která je spuštěna příkazem SELECT.

³¹ Pozor na případnou RACE CONDITION.

³² Vytvoření subtransakce má určitou režii – pozor na použití v cyklu, a nepoužívat, když není nezbytně nutné

```
BEGIN
a := 1;
INSERT INTO tab1 VALUES(1);
COMMIT;
INSERT INTO tab1 VALUES(a + 1);
ROLLBACK;
END;
$$ LANGUAGE plpgsql;

CALL foo(10);
```

Procedury mohou modifikovat INOUT parametry.

Kontrola SQL identifikátorů

Při registraci funkce se provede syntaktická kontrola vložených SQL příkazů. Nekontroluje se správnost identifikátorů³⁹. Pro kontrolu identifikátorů musíme funkci (proceduru) spustit, a dotazy nechat vykonat. V mnoha případech postačí statická analýza pomocí extenze `plpgsql_check`³⁹:

```
CREATE EXTENSION plpgsql_check;
SELECT * FROM plpgsql_check_function('novy_zamestnanec');
```

Kromě jiného statická analýza provedená touto extenzí může identifikovat problémové operace z hlediska výkonu, bezpečnosti (detekce SQL injection), nebo nepoužívané proměnné nebo mrtvý kód.

Profiling

U pomalejších nebo častěji používaných funkcí bychom měli vědět o úzkých hrdlech kódu funkce (procedury). Nejjednodušším nástrojem pro profilování kódu je statistika četností a doby volání funkcí v pohledu `pg_stat_user_functions`⁴⁰.

Pro detailnější pohled (na úrovni jednotlivých příkazů) je nutné nainstalovat externí profiler. K dispozici jsou dva – `plProfiler`⁴¹ a integrovaný profiler v `plpgsql_check`.

Profiler z extenze `plpgsql_check` se aktivuje nastavením konfigurační proměnné `plpgsql_check.profiler`. Po vykonání libovolné funkce v `PlpgSQL` se můžeme podívat na její profil:

```
SET plpgsql_check.profiler TO ON;
SELECT novy_zamestnanec('Stehule', true);
SELECT * FROM plpgsql_profiler_function('novy_zamestnanec');
```

Partitioning

Partitioning umožňuje rozdělit data v relaci do definovaných fyzicky oddělených disjunktích 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 podděděním. Například z relace *lidé* (*jméno*, *příjmení*) poddělím 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 (lídé);
CREATE TABLE zamestnanci(zarazení text) INHERITS(lídé);
```

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* automaticky⁴² - musíme si je vytvořit manuálně.
- ✗ Umístění záznamů do odpovídajících *partitions* **se neprovádí** automaticky⁴³ - musíme si napsat distribuční trigger⁴⁴.
- ✗ Počet *partitions* **není omezen** – neměl by ovšem přesáhnout 100 *partitions* jedné tabulky⁴⁵.

Redistribuční trigger

Úkolem tohoto triggeru je přesun záznamu z rodičovské tabulky do odpovídající podděšené tabulky⁴⁶ (pro větší počet *partitions* – cca nad 20 je praktické použití *dynamického* SQL).

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

- 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.
- 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.
- Od verze 10 není nutné.
- Při velkém počtu *partitions* je problém s paměťovými nároky optimalizátoru.
- Také aplikace může přesněji cílit a zapisovat do tabulek, které odpovídají *partitions* a nikoliv do rodičovské tabulky – tím se ušetří volání redistribučního triggeru a INSERT bude rychlejší.

```
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)
```

Deklarativní partitioning

V případě deklarativního *partitioningu* není nutné psát redistribuční trigger. Tento typ *partitioningu* může být založený na disjunktích intervalech (ranges):

```
CREATE TABLE data(a text, vlozeno date)
PARTITION BY RANGE(vlozeno);
CREATE TABLE data_2016 PARTITION OF data
FOR VALUES FROM ('2016-01-01') TO ('2017-01-01');
CREATE TABLE data_2017 PARTITION OF data
FOR VALUES FROM ('2017-01-01') TO ('2018-01-01');
CREATE TABLE data_other PARTITION OF data DEFAULT47;
```

Další možností je *partitioning* založený na seznamech:

```
CREATE TABLE data(a text, vlozeno date)
PARTITION BY LIST(EXTRACT(YEAR FROM vlozeno));
CREATE TABLE data_2016 PARTITION OF data FOR VALUES IN (2016);
CREATE TABLE data_2017 PARTITION OF data FOR VALUES IN (2017);
```

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'";
done
```

Oblíbeným 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 bloků⁴⁸ – kódu v `plpgsql`.

```
SQL=$(cat <<'EOF'
SELECT set_config('custom.owner', :'owner', false);
DO $$
```

38 V `PlpgSQL` musí být SQL identifikátor validní až v okamžiku běhu dotazu.

39 Lze instalovat z komunitního repozitáře.

40 Aktualizace tohoto pohledu se zapíná nastavením `track_function` na „pl“ nebo „all“. Toto nastavení může provést pouze superuser.

41 Umí html report a flame grafy

47 Default partition od verze 11

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

```
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 'database=%', 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
```

Počínaje verzí 10 je možné v psql použít jednoduchý skriptovací jazyk:

```
SELECT pg_is_in_recovery() as is_slave \gset
\if :is_slave
  \set PROMPT1 '\nslave %x$ '
\else
  \set PROMPT1 '\nmaster %x$ '
\endif
```

Paralelní vykonání příkazu

Častou úlohou může být provedení určitého příkazu pro každou databázi. Takové příkazy lze obvykle dobře paralelizovat a to jednoduše na unixových systémech díky příkazu *xargs*:

```
psql -At -c "SELECT datname FROM pg_database
WHERE NOT datistemplate AND dataallowcomm" postgres |
xargs -n 1 -P 249 psql -c "vacuum full"
```

Row Level Security

Každá bezpečnostní politika přidává filtr, který se aplikuje pro vybrané uživatele (případně pro všechny uživatele). Uživatel vidí obsah⁵⁰, pokud filtr vrácí hodnotu true (klauzule USING).n Klauzule WITH CHECK⁵¹ se uplatní u příkazů INSERT a UPDATE. V případě, že výraz v této klauzuli není pravdivý, potom příkaz selže.

```
CREATE TABLE foo(s text, owner regrole);
GRANT ALL ON foo TO public;
ALTER TABLE foo ENABLE ROW LABEL SECURITY;
```

```
CREATE POLICY owner_policy ON foo
USING (owner = current_user::regrole);
```

Výše uvedená politika způsobí, že uživatel vidí a může editovat záznamy, které sám vložil.

Výchozí politiky nejsou restriktivní – v případě, že k tabulce máme více politik, tak stačí jedna splněná politika, aby uživatel měl zpřístupněna data. Tzv restriktivní politiku vytvoříme pomocí klauzule RESTRICTIVE (musí být vždy splněné):

```
CREATE POLICY admin_local_only52 ON passwd
AS RESTRICTIVE TO admin
USING (pg_catalog.inet_client_addr() IS NULL);
```

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 tzv *archive_command*⁵³:

```
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 logy⁵⁴.

Obnova ze zálohy

Rozbalení poslední plné zálohy

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

Nastavte 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:

52 Nedovolí přístup k tabulce *passwd*, pokud se uživatel přihlásil vzdáleně.

53 Vždy při naplnění segmentu transakčního logu nebo vypršení časového intervalu PostgreSQL volá *archive_command*, jehož úkolem je zajistit zápis segmentu na bezpečné médium.

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

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

PostgreSQL implicitně⁵⁵ provádí obnovu do okamžiku, ke kterému dohledá 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/00000001000000000000000A':
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 aktivní databázi replication (přístup se povoluje v souboru *pg_hba.conf*).

```
wal_level = replica
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).

49 Příkaz VACUUM bude použit ve dvou paralelních procesech.

50 Předpokladem jsou odpovídající práva k tabulce.

51 Pokud tato klauzule chybí, použije se pro stejný účel klauzule USING.

Úpravy konfigurace – master

```
wal_level = replica
max_wal_senders = 10

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

Úpravy konfigurace – slave⁵⁶

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

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

Na slave v `postgresql.conf` v sekci Standby servers:

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

Před startem repliky vymažte log a **pid** file. **Vytvořte prázdný soubor `standby.signal`**. Po startu by měl log obsahovat záznam:

```
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 |
| <code>pg_is_in_recovery()</code> ⁵⁷ | vrátí true na standby serveru |

Logická replikace

Fyzická replikace replikuje instanci Postgresu. Pokud chceme replikovat jen vybrané tabulky, pak musíme použít tzv *logickou replikaci*⁵⁸. Pro logickou replikaci je nutné pouze nastavit:

```
wal_level = logical
```

Dále je nutné vybrané tabulky zveřejnit:

```
CREATE TABLE foo(id int primary key, a int);
CREATE PUBLICATION test_pub FOR TABLE foo;
INSERT INTO foo VALUES(1, 200);
```

a na druhé straně aktivovat odběr zveřejněných tabulek:

```
CREATE TABLE foo(id int primary key, a int)59;
CREATE SUBSCRIPTION60 test_sub
```

```
CONNECTION 'port=5432' PUBLICATION test_pub
WITH (streaming=on);
```

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%';
```

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) @@61
       to_tsquery('simple','skal:*62 & !česká');
```

Vlastní konfigurace se vytvářejí kopií a následnou úpravou některé stávající. Následující konfigurace zahrnuje použití funkce *unaccent*⁶³.

```
CREATE TEXT SEARCH CONFIGURATION simple_unaccent
                                ( COPY = simple );
ALTER TEXT SEARCH CONFIGURATION simple_unaccent
  ALTER MAPPING FOR hword, hword_part, word
  WITH unaccent64, 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','svatý');
```

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 ten následující⁶⁵ dokáže využít index.

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

K optimalizaci dotazů s predikátem **LIKE** (i **ILIKE**) lze použít extenzi *pg_trgm*, která obsaguje 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 `'~*'`⁶⁶.

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

Také vyhledávání prostřednictvím regulárních výrazů může být urychleno trigramovým indexem⁶⁷.

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

57 Od verze 14 se můžeme podívat na systémovou proměnnou `in_hot_standby`

58 V řadě ohledů je podobná fyzické replikaci – přenáší se změny v datech, používá se transakční log.

59 Nereplikují se DDL změny

60 Pokud se použije parametr *streaming=ON*, tak dochází k průběžné odesílání dat k odběrateli (nečeká se na dokončení transakce, změny dat se potvrdí s potvrzením zdrojové transakce).

61 Fulltextový operátor

62 Hledání prefixu „skal“.

63 Vyzaduje extenzi *unaccent*.

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

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

66 Case insensitive varianta

67 Za předpokladu, že je určen kompletní trigram (tři znaky)

pg_rman

*pg_rman*⁶⁸ je jednoduchá aplikace příkazového řádku pro zejména lokální zálohování a management záloh využívající mechanismus exportu transakčního logu. Požadavkem je přímý přístup k datovému adresáři, přístup k adresáři, kde budou uloženy zálohy a přístup k adresáři, kde se exportují segmenty transakčního logu.

Konfigurace

Postgres musí mít aktivní export segmentů transakčního logu – viz konfigurace: `archive_command`, `archive_mode`. K cílovému adresáři musí mít *pg_rman* přístup⁶⁹. Dále musí mít přístup k datovému adresáři postgresu a k adresáři, kde budou umístěny zálohy. Tyto adresáře jsou identifikovány pomocí přepínačů nebo systémovými proměnnými PGDATA a BACKUP_PATH⁷⁰.

Adresář pro uložení záloh musí být prázdný – inicializuje se příkazem

```
pg_rman init
```

Tento příkaz vytvoří v zadaném adresáři konfigurační soubor *pg_rman.ini*, kde lze ještě nastavit:

| | |
|---------------------------|--|
| BACKUP_MODE = F | výchozí režim zálohování |
| COMPRESS_DATA = YES | aktivovat komprimaci |
| KEEP_ARCLOG_FILES = 10 | retence počtu exportovaných segmentů WAL |
| KEEP_ARCLOG_DAYS = 2 | retence stáří exportovaných segmentů WAL ⁷¹ |
| KEEP_DATA_GENERATIONS = 4 | retence počtu úplných záloh |
| KEEP_DATA_DAYS = 30 | retence stáří záloh |

Základní příkazy

Zobrazí seznam a podrobnosti provedených záloh⁷².

```
pg_rman show [ ( detail | čas zálohy ) ]
```

Vytvoří zálohu

```
pg_rman backup -backup_mode=incr
```

Validace zálohy – pouze z validovaných záloh lze obnovovat, a pouze vůči validovaným zálohám lze vytvořit inkrementální zálohu

```
pg_rman validate
```

Zrušení všech zbytných záloh starších než zadané datum

```
pg_rman delete datum
```

Z katalogu provedených záloh odstraní záznamy o zrušených zálohách

```
pg_rman purge
```

Obnova ze zálohy

```
pg_rman restore [ ( --recovery-target-time |  
                  --recovery-target-xid |  
                  --recovery-target-timeline ) bod obnovy ]
```

Barman

*Barman*⁷³ 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 serverech musí být nainstalovaná stejná verze Postgresu⁷⁴, Python a psycopg2 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 serveru⁷⁵:

```
[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  
backup_method = rsync  
backup_options = concurrent_backup  
archiver = on
```

Dále je nutné nakonfigurovat zálohovaný PostgreSQL⁷⁶:

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

Základní příkazy

Verifikace konfigurace

```
barman check78 dbserver01
```

Vytvoření kompletní zálohy serveru (všech serverů)

68 *pg_rman* zvládá plnou zálohu, inkrementální zálohu (redukovaná plná záloha), zálohu transakčních logů, zálohu logu Postgresu, retenci záloha a retenci exportovaných transakčních logů.

69 konfigurační proměnná ARCLOG_PATH.

70 Doporučuje se je nastavit v profilu

71 Pro odstranění souborů je nutné splnit vždy obě podmínky.

72 Čas vytvoření zálohy je zároveň jejím identifikátorem

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

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

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

76 *postgresql.conf*

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

78 Před spuštěním testu je nutné exportovat alespoň jeden segment transakčního logu. Např příkazem `barman switch-xlog -force -archive dbserver01`

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

Výpis seznamu záloh

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

Lokální⁷⁹ obnova ze zálohy

```
barman recover dbserver01 20140419T23552480 ~/xxx
```

Informace k záloze

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

Explicitní odstranění zálohy

```
barman delete dbserver01 oldest
```

Repmgr

*repmgr*⁸¹ 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 mastera⁸².

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 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):

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

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

81 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*.

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

```
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'
```

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

vynucené 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 pool⁸⁴. 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*⁸⁵, 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
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
autodb_idle_timeout = 3600
```

recovery_target_timeline='latest'

⁸⁴ Počet otevřených spojení v poolu lze omezit. V případě nedostatku volných spojení PgBouncer umí požadavek o spojení podržet předdefinovanou dobu.

⁸⁵ aplikace vyžaduje *psycopy2*

```
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 téže portu.

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

Monitoring

Databáze pgbounce 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"
```

pspg

pspg⁸⁶ je unixový pager navržený s ohledem na prohlížení tabulek. Umí zafixovat řádek se jmény sloupců a prvních n sloupců s identifikátory řádků. Lze jej také použít pro prohlížení CSV a TSV souborů. Ovládání pspg vychází z ovládání pageru less, které opět v mnohém respektuje ovládání editoru v.i. Pspg lze také ovládat pomocí menu (klikem myši nebo stiskem F9).

Klávesové zkratky

| | |
|--------------------|---|
| 0..9 | fixace sloupců |
| q, F10, ESC 0 | ukončení |
| KEY_UP, k | kurzor o řádek nahoru |
| KEY_DOWN, j | kurzor o řádek dolů |
| KEY_LEFT, h | o sloupec doleva |
| KEY_RIGHT, l | o sloupec doprava |
| CTRL HOME, g | kurzor na první řádek |
| CTRL END, G | kurzor na poslední řádek |
| HOME, ^ | první sloupec |
| END, \$ | poslední sloupec |
| PG_DOWN, ^f, space | skok na další stránku |
| PG_UP, ^b | skok na předchozí stránku |
| s | ulož obsah do souboru |
| / | zadat a hledat řetězec ve směru textu |
| ? | zadat a hledat řetězec proti směru textu |
| n | hledat další výskyt řetězce v daném směru |
| N | hledat další výskyt řetězce v opačném směru |
| ALT k | definovat záložku (bookmark) |
| ALT j | skok na další záložku |
| ALT i | skok na předchozí záložku |
| a | řadit vzestupně podle vertikálního kurzoru |
| d | řadit sestupně podle vertikálního kurzoru |
| u | nastavit původní pořadí řádků |
| ALT c | přepnout zobrazení kurzoru |
| ALT m | přepnout ovládání myši (vlastní, terminál) |
| ALT n | přepnout zobrazení čísel řádků |
| ALT v | přepnout zobrazení vertikálního kurzoru |
| F9, ESC 9 | aktivovat menu |
| CTRL o | dočasně zobrazí primární obrazovku |

⁸⁶ Nachází se v repozitářích většiny linuxových distribucí, případně v komunitním repozitáři.

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:

povyšení slave na master

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

přesměrování slave na nového mastera⁸³

⁸³ Pro správnou funkci je nutné alespoň 9.3 a v *recovery.conf*

Nerelační datové typy⁸⁷

S použitím typů HStore, JSON, JSONB a XML můžeme emulovat nerelační databáze. V JSONB jsou data uložena binárně, ostatní typy se ukládá jako text⁸⁸. 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*⁸⁹ 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);
```

Vrátí jména všech osob, jejichž příjmení je „stěhule“

```
SELECT ostatni->'jmeno'
FROM lide
WHERE ostatni @> 'prijmeni => stěhule';
```

Přidá atribut zaměstnání

```
UPDATE lide
SET ostatni = ostatni || 'zamestnani=>programator'
WHERE rc = 7307150888;
```

Vrátí všechny záznamy, které obsahují atribut zaměstnání – výsledkem je JSON

```
SELECT hstore_to_json(ostatni)
FROM lide
WHERE ostatni ? 'zamestnani';
```

Vytvoření funkcionálního indexu nad atributem zaměstnání a jeho použití:

```
CREATE INDEX ON lide ((ostatni->'zamestnani'));
SELECT *
FROM lide
WHERE ostatni->'zamestnani' = 'programator';
```

Operátory a funkce

| | |
|---------------------|--|
| hstore -> text | získání hodnoty |
| hstore -> text[] | získání pole hodnot |
| hstore hstore | spojení dvou hodnot typu hstore |
| hstore ? text | test, zda-li obsahuje klíč |
| hstore ?& text[] | test, zda-li obsahuje všechny klíče |
| hstore ? text[] | test, zda-li obsahuje některý klíč |
| hstore @> hstore | test, zda-li levý op. obsahuje pravý operand |
| hstore #= hstore | změna vybraných klíčů |
| hstore - text | odstraní klíč |
| hstore - hstore | rozdíl dvou hodnot typu hstore |
| hstore(record) | konstruktor z kompozitního typu |
| hstore(text, text) | konstruktor klíč, hodnota |
| hstore_to_matrix(h) | převede na 2D pole |
| hstore_to_json(h) | převede na JSON |

| | |
|-----------------------|---------------------------------|
| slice(h, text[]) | vrátí vyjmenované klíče |
| each(h) | převede na tabulku klíč/hodnota |
| populate_record(t, h) | převede na záznam typu t |

JSON

Data jsou uložena v textovém formátu – při vyhledávání uvnitř dokumentu je nutné vždy dokument parsovat⁹⁰. Pro indexaci položek je možné použít funkcionální index.

```
SELECT row_to_json(row(1, 'foo'));
```

Operátory a funkce

| | |
|-----------------|---|
| json -> text | získání atributu |
| json -> int | získání prvku pole |
| json ->> text | získání atributu jako textu |
| json ->> int | získání prvku pole jako textu |
| json #> text[] | získání atributu určeného cestou |
| json #>> text[] | získání atributu určeného cestou jako textu |

| | |
|---------------------------|--|
| array_to_json(a) | převede pole na JSON |
| row_to_json(r) | převede kompozitní typ na JSON |
| hstore_to_json(h) | převede HStore (vše text) na JSON |
| hstore_to_json_loose(h) | převede HStore na JSON s ohledem na typy |
| to_json(anyelement) | převede hodnotu na validní JSON hodnotu |
| json_each(json) | rozvine JSON na tabulku klíč/hodnota |
| json_each_text(json) | rozvine JSON na tabulku klíč/hodnota jako text |
| json_populate_recordset() | převede JSON na řádek určeného typu |
| json_array_elements(json) | rozvine pole JSON na tabulku |
| json_build_object() | vytvoří nticí dvojic (klíč, hodnota) |
| json_build_array() | vytvoří posloupnost hodnot |
| json_strip_null(json) | redukuje NULL hodnoty |
| json_pretty(json) | formátuje JSON |

```
CREATE TYPE x AS (a int, b int);
SELECT *
FROM json_populate_recordset(null::x,
'[{ "a":1, "b":2 }, { "a":3, "b":4 } ] ' );
SELECT json_build_object('foo',1,'boo',2);
SELECT json_build_array(1,2,3,'Hi',4);
```

jsonb

jsonb vychází z typu HStore – data jsou uložena binárně (při hledání v dokumentu nedochází k parsování) a podporuje rekurzi – *jsonb* může obsahovat další vložené JSONB dokumenty. Na vstupu a výstupu se používá formát JSON.

```
SELECT '[1, 2, "foo", null]':jsonb;
SELECT '{"bar": "baz", "balance": 7.77,
"active":false}':jsonb;
```

Kromě podpory B-Tree funkcionálního indexu existuje podpora jsonb GIN indexu. **Pozor: zanořené tagy nejsou indexovány!**

```
CREATE INDEX idxgin ON api USING GIN (jdoc);
CREATE INDEX idxginh ON api USING GIN (jdoc jsonb_hash_ops91);
SELECT jdoc->'guid', jdoc->'name'
FROM api
WHERE jdoc @> '{"company": "MagnaFone"}';
```

Existující operátory a funkce pro typ jsonb je mix operátorů a funkcí typů HStore a JSON. Navíc jsou funkce (analogické funkcím pro JSON): *jsonb_each*, *jsonb_each_text*,

jsonb_populate_record, *jsonb_populate_recordset*, *jsonb_array_elements*, *jsonb_array_elements_text* atd.

Od verze 15 jsou k dispozici nové SQL/JSON funkce *JSON_EXISTS* (vrací true nebo false, pokud JSONPath výraz vrátí alespoň jednu hodnotu), *JSON_VALUE* (vrací hodnotu), *JSON_QUERY* (vrací JSON objekt nebo pole), *JSON_TABLE* (vrací tabulku) a operátor *IS JSON {OBJECT | ARRAY | SCALAR}*.

```
JSON_VALUE(jsonb "ahoj", '$ RETURNING text)
JSON_VALUE(jsonb "2017-02-20", '$ RETURNING date)
JSON_VALUE(jsonb '{"a": 1}', '$.a RETURNING int)
JSON_VALUE(jsonb '{"a": 1}', '$.b RETURNING int DEFAULT 10 ON EMPTY)
JSON_VALUE(jsonb '{"a": 1}', '$.b RETURNING int
ERROR ON EMPTY ERROR ON ERROR)
```

```
JSON_QUERY(jsonb '[10,20,30]', '$[*] ? (@ > 20)' WITH WRAPPER);
```

```
JSON_TABLE(jsonb '1,2,3', '$[*]' COLUMNS(a int PATH '$'))
JSON_TABLE(jsonb '{"a": 1, "b": {"x": 1, "y":2}}',
'$ COLUMNS (a text,
NESTED PATH '$.c' COLUMNS (x int, y int)))
```

JSON hodnotu lze vytvořit konstruktor funkcemi *JSON⁹²*, *JSON_OBJECT*, *JSON_SCALAR*, *JSON_ARRAY*, *JSON_ARRAYAGG*, *JSON_OBJECT*, *JSON_OBJECTAGG*:

```
JSON_SCALAR('ahoj')
JSON_OBJECT('a':ARRAY[10,20], 'b': 'nazdar');
JSON_ARRAY(10,203,40);
JSON_ARRAY(10,203,40, null NULL ON NULL);
```

SQL/JSON Path language (JSONPath)

Počínaje PostgreSQL 12 můžeme pro vyhledávání v JSON a jsonb používat dotazovací jazyk *JSONPath*:

| | |
|---------------|--|
| . | přístup k položce struktury |
| [] | přístup k prvku pole (pole začínají nulou) |
| \$ | hodnota |
| \$name | pojmenovaná hodnota |
| @ | proměnná reprezentující výsledek |
| .key | přístup k položce |
| ."\$name" | přístup k položce prostřednictvím pojmenované proměnné |
| .* | použije všechny položky struktury |
| ** | použije všechny položky struktury rekurzivně (nepoužívat v lax módu) |
| [*] | všechny položky pole |
| && | operátor AND |
| | operátor OD |
| ==, <, >, ... | ostatní matematické operátory |

Pro filtrování se dva základní operátory:

| | |
|----|---|
| @? | true, když výsledek výběru není prázdný |
| @@ | vrací výsledek logického výrazu nebo NULL |

Pro zobrazení lze použít funkci *jsonb_path_query* :

```
SELECT '[1,2,3]':jsonb @@? '$[*]' ? (@ >= 2)';
SELECT '[1,2,3]':jsonb @@ '$[*]' >= 2)';
SELECT jsonb_path_query('[1,2,3]', '$[*]' ? (@ >= 2)');
```

87 Mezi nerelační datové typy patří i pole a typy range a multirange, sloužící jako kolekce (ukládají data v nativním formátu (binárně)).

88 Ve většině případů bez negativního vlivu na výkon.

89 Entity Attribute Value model

90 Lze vyřešit funkcionálním indexem.

91 GIN HASH podporuje pouze operátor @>. Hash index by měl být menší.

92 Bohužel tyto funkce aktuálně vrací JSON (nikoliv jsonb)

XML

Opět data jsou uložena v textovém formátu – dokumenty nad 2KB jsou efektivně komprimovány díky TOAST. Největší výhodou tohoto typu jsou uživatelsky přívětivé a silné funkce pro generování XML dokumentů dotazem respektující ANSI SQL/XML:

XMLCOMMENT, *XMLCONCAT*, *XMLELEMENT*, *XMLFOREST*, *XMLPI*, *XMLROOT*, *XMLAGG*.

```
SELECT
XMLROOT (
  XMLELEMENT( NAME gazonk,
    XMLATTRIBUTES ( 'val' AS name, 1 + 1 AS num ),
    XMLELEMENT ( NAME qux, 'foo' ) ),
  VERSION '1.0',
  STANDALONE YES );
```

Velice praktická funkce je *XMLFOREST*:

```
SELECT XMLFOREST( first_name AS "FName", last_name AS "LName",
  title AS "Title", region AS "Region")
FROM employees;
```

Dotazy ve kterých se používá SQL/XML funkcionalita nemusí být dobře čitelné, lze si pomoci funkcemi:

```
CREATE OR REPLACE FUNCTION cast_to_xml(date)
RETURNS xml AS $$
  SELECT xmlelement(NAME "date", to_char($1, 'YYYY-MM-DD'));
$$ LANGUAGE sql;
```

Celou tabulku nebo dotaz lze vyexportovat do jednoduchého XML dokumentu funkcemi:

```
table_to_xml(tbl regclass, nulls boolean,
  tableforest boolean, targetns text)
query_to_xml(query text, nulls boolean,
  tableforest boolean, targetns text)
```

Pro vyhledávání lze použít funkci XPATH:

```
SELECT xpath('/my:a/text()',
  '<my:a xmlns:my="http://example.com">test</my:a>',
  ARRAY[ARRAY['my', 'http://example.com']]);

SELECT (xpath('/gazonk/qux/text()', xmlcol))[0]93;
```

Pro parsování (převod XML na tabulku) můžeme použít funkci XMLTABLE:

```
SELECT xmltable.*
FROM xmldata,
XMLTABLE('/ROWS/ROW'
  PASSING data
  COLUMNS id int PATH '@id',
  ordinality FOR ORDINALITY,
  "COUNTRY_NAME" text,
  country_id text PATH 'COUNTRY_ID',
  size_sq_km float
  PATH 'SIZE[@unit = "sq_km"]',
  size_other text PATH
'concat(SIZE[@unit!="sq_km"], " ", SIZE[@unit="sq_km"])/@unit',
  premier_name text
  PATH 'PREMIER_NAME'
  DEFAULT 'not specified') ;
```

Integrace v RedHatu

Řízení služeb (včetně PostgreSQL) se v Linuxu vyvíjelo od startup skriptů k aktuálně nejrozšířenějšímu subsystému pro správu služeb *systemd* používaným jak v RH tak v Debianu.

Pokud chcete používat jiný než výchozí adresář pro databázi je nutné upravit unit file

```
sudo systemctl edit94 postgresql-1495 96
```

pro změny⁹⁷ v unit souborech (po vytvoření nových unit souborů) může být nezbytné refreshnout *systemd*:

```
systemctl daemon-reload
```

Poté můžeme vytvořit databázový cluster a nastartovat službu

```
PGSETUP_INITDB_OPTIONS="-k" postgresql-14-setup initdb
systemctl enable postgresql-14
```

```
sudo systemctl start postgresql-14
sudo systemctl stop postgresql-14
sudo systemctl restart postgresql-14
sudo systemctl status postgresql-14
```

Příkazy pro výpis seznamu units, nastavení a vlastní výpis unity

```
sudo systemctl list-units
systemctl show postgresql-14|grep PGDATA
systemctl cat postgresql-14
```

Pro investigaci ať už výkonostních nebo jiných problémů je nutné nainstalovat debug symboly postgresu:

```
yum debuginfo-install postgresql14-server
```

Pozor – promote se provede skrze *pg_ctl* *promote*.

Pro upgrade z předchozí major verze⁹⁸ lze použít skript *postgresql-14-setup upgrade* (databáze se musí vytvořit *postgresql-14-setup initdb*)⁹⁹.

Poznámky

94 Lze použít přepínač –full, kterým vytvoří modifikovatelná kopie celého unit souboru.

95 Musí změnit proměnná Environment v sekci [Service]. Viz systemctl cat ... Vytvoří se soubor /etc/systemd/system/postgresql-14.service.d/override.conf

96 Také lze zkopírovat unit file v adresáři /usr/lib/systemd/system. Kopie by měla být umístěna v adresáři /etc/systemd/system/

97 Změny v unit souborech zobrazí příkaz systemd-delta

98 Ve skriptu lze modifikovat proměnnou PREVMAJORVERSION

99 Volá pg_upgrade

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

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 16 tis. Kč (včetně DPH) pro 4 osob plus příplatek 2000 Kč za každého další účastníka (32 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 cestovné výdaje. V ceně jsou vytištěné 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 databázi. 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 **připravované** školení je určeno pokročilejším uživatelům PostgreSQL. V rámci školení se účastníci seznámí s možnostmi zálohování a také si prakticky vyzkouší konfiguraci vestavěné replikace.

- 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 databází. 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

Moderní SQL v PostgreSQL

Toto školení je určeno IT profesionálům a pokročilým uživatelům. V posledních několika letech vývojáři PostgreSQL implementovali většinu rozšíření SQL, které vychází z ANSI SQL 2001. Některé dotazy, které dříve bylo nutné řešit aplikačně nebo pomocí uložených procedur, lze nyní napsat jednoduše a čitelně v SQL – což přináší úsporu času, redukuje kód a zvyšuje jeho čitelnost.

- Analytické (window) funkce
- Common Table Expression – rekurzivní dotazy a dočasné pohledy
- Agregáčn

Autor: [Pavel Stěhule](#)
Kontakt: pavel.stehule@gmail.com, tel: 724 191 000
Profil: [cz.linkedin.com/in/stehule/](https://www.linkedin.com/in/stehule/) stackexchange.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, migrace z Oracle