PostgreSQL 8.3

Pavel Stěhule
Department of Mapping and Cartography, Faculty of Civil Engineering
Czech Technical University in Prague
stehule kix.fsv.cvut.cz

Klíčová slova: Database systems, Open Source

Abstrakt

Práce na Open Source databázích pokračují nezadržitelným tempem. Vývojáři se musí vyrovnat s rostoucími požadavky uživatelů na objem dat ukládaných do databází, na náročnější požadavky na odezvu atd. Zatím nedostižnou metou je implementace celého standardu ANSI SQL 200x. Všechny databáze z velké trojky (Firebird, MySQL a PostgreSQL) používají multigenerační architekturu, cenově orientované hledání optimálního prováděcího plánu, write ahead log atd. MySQL se profiluje jako SQL databáze schopná používat specializované databázové backendy schopné maximální efektivity pro určité konkrétní prostředí. PostgreSQL je široce použitelná databáze, těžící z vynikající stability, s perfektní rozšiřitelností a komfortním prostředím. Konečně Firebird je vynikající embeded databáze, která se osvědčuje v tisících instalacích na desktopech.

Podle původního plánu mělo dojít k uvolnění verze 8.3 koncem léta – mělo jít o verzi obsahující patche dokončené pro 8.2, ale v té době nedostatečně otestované. Nakonec se ukázalo, že ty nejdůležitější patche je třeba dopracovat. Jednalo se o tak atraktivní vlastnosti, že se rozhodlo s vydáním nové verze počkat. 8.3 obsahuje integrovaný fulltext, podporu opožděného potvrzování (asynchronní commit), synchronizované sekvenční čtení datových souborů, úspornější ukládání dynamických datových typů (kratších 256byte), HOT updates a sofistikovanější aktualizaci indexů (hot indexes). Z patchů připravených pro 8.2 se v 8.3 neobjeví podpora bitmapových indexů a podpora aktualizovatelných pohledů. Původní řešení založené na pravidlech (rules) bylo příliš komplikované. 8.3 obsahuje podporu aktualizovatelných kurzorů, a je docela dobře možné, že aktualizovatelné pohledy budou ve verzi 8.4 implementovány právě s pomocí této třídy kurzorů.

Vývoj pokračuje implementací dalších modulů SQL. Ve verzi 8.3 je to konkrétně SQL/XML (rozšíření ANSI SQL), která umožňuje operace s XML dokumenty přímo v databázi a zjednodušuje generování XML dokumentů. Zásadní (interní) změnou je zkrácení hlavičky řádku z 28 bajtů na 24 bajtů. Další změnou, která by měla vést k minimalizaci velikosti uložených dat je diverzifikace typu varlena. Tento typ se v PostgreSQL používá pro serializaci hodnot všech typů s variabilní délkou. Trochu připomíná string v Pascalu. První byty nesou informaci o délce, další nesou obsah. Starší verze PostgreSQL znaly jen typ varlena s 4byte informací o

délce. 8.3 podporuje také typ varlena s 1byte záhlavím. Úspora by se měla projevit hlavně u typu NUMERIC a krátkých řetězců. K překladu PostgreSQL lze počínaje touto verzí použít jak gcc, MINGW tak Microsoft Visual C++ (na platformě Microsoft Windows).

Integrace TSearch2

Integrace TSearch2 do jádra PostgreSQL je výsledkem dlouholetého úsilí Olega Bartunova a Teodora Sigaeva. Díky integraci se zjednoduší konfigurace fulltextu a pro určité jazyky (pro které existuje podpora v projektu Snowball) lze fulltext používat hned po instalaci databáze. Čeština bohužel mezi tyto jazyky nepatří – je potřeba provést několik dalších operací. Předně převést Open Office slovníky do kódování UTF8 a zkopírovat je do příslušného podadresáře PostgreSQL. Dále zaregistrovat slovník a provést tzv. konfiguraci. Kromě konfigurace jsou rozdíly mezi integrovaným fulltextem a TSearch2 (z verze 8.2) spíše kosmetické.

```
CREATE TEXT SEARCH DICTIONARY cspell1(
   template=ispell,
   dictfile = czech, afffile=czech, stopwords=czech);

CREATE TEXT SEARCH CONFIGURATION cs (copy=english);

ALTER TEXT SEARCH CONFIGURATION cs
   ALTER MAPPING FOR word, lword  WITH cspell, simple;

postgres=# select * from ts_debug('cs','Příliš žluťoučký kůň se napil žluté vody');
 Alias |  Description  |   Token   |  Dictionaries   |    Lexized token
-------+---------------+-----------+-----------------+---------------------
 word  | Word          | Příliš    | {cspell,simple} | cspell: {příliš}
 blank | Space symbols |           | {}              |
 word  | Word          | žluťoučký | {cspell,simple} | cspell: {žluťoučký}
 blank | Space symbols |           | {}              |
 word  | Word          | kůň       | {cspell,simple} | cspell: {kůň}
 blank | Space symbols |           | {}              |
 lword | Latin word    | se        | {cspell,simple} | cspell: {}
 blank | Space symbols |           | {}              |
 lword | Latin word    | napil     | {cspell,simple} | cspell: {napít}
 blank | Space symbols |           | {}              |
 word  | Word          | žluté     | {cspell,simple} | cspell: {žlutý}
 blank | Space symbols |           | {}              |
 lword | Latin word    | vody      | {cspell,simple} | cspell: {voda}
(13 rows)
```

Podporu fulltextu nad konkrétním sloupcem můžeme aktivovat např. vytvořením funkcionálního GIN indexu.

```
CREATE INDEX data_poznamka_ftx ON data
   USING gin(to_tsvector('cs', poznamka))
```

a vyhledávat operátorem @@ (fulltextové vyhledávání)

```
SELECT * FROM data
  WHERE to_tsvector('cs',poznamka) @@ to_tsquery('žlutá & voda')
```

Podpora SQL/XML

Zásadně se změnila podpora XML. To co v předchozích verzích se neohrabaně řešilo přes doplňky se nyní dostalo přímo do jádra. Jednak jsou k dispozici funkce generující XML (xmlelement, xmlforest, ...) jednak jsou tu funkce mapující obsah tabulky do XML. Výsledkem může být XML schéma (použitelné pro validaci nebo pro přenos definice tabulky), XML dokument s integrovaným schématem, nebo samotný XML dokument. Jelikož je výstupní

formát standardizován v SQL/XML, neměl by být přenos těchto tabulek problémem (mezi těmi databázemi, které SQL/XML podporují).

```
pavel=# create table a(a date, b varchar(10));
CREATE TABLE
pavel=# insert into a values(current_date, 'něco'),(current_date+1, NULL);
INSERT 0 2
pavel=# select table_to_xml_and_xmlschema('a', true, false, '');
                                  table_to_xml_and_xmlschema
-----------------------------------------------------------------------------------------------
 <a xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:noNamespaceSchemaLocation="#">

 <xsd:schema
     xmlns:xsd="http://www.w3.org/2001/XMLSchema">

 <xsd:simpleType name="DATE">
   <xsd:restriction base="xsd:date">
     <xsd:pattern value="\p{Nd}{4}-\p{Nd}{2}-\p{Nd}{2}"/>
   </xsd:restriction>
 </xsd:simpleType>

 <xsd:simpleType name="VARCHAR">
 </xsd:simpleType>

 <xsd:complexType name="RowType.root.public.a">
   <xsd:sequence>
     <xsd:element name="a" type="DATE" nillable="true"></xsd:element>
     <xsd:element name="b" type="VARCHAR" nillable="true"></xsd:element>
   </xsd:sequence>
 </xsd:complexType>

 <xsd:complexType name="TableType.root.public.a">
   <xsd:sequence>
     <xsd:element name="row" type="RowType.root.public.a" minOccurs="0" maxOccurs="unbounded"/>
   </xsd:sequence>
 </xsd:complexType>

 <xsd:element name="a" type="TableType.root.public.a"/>

 </xsd:schema>

 <row>
   <a>2007-02-19</a>
   <b>něco</b>
 </row>

 <row>
   <a>2007-02-20</a>
   <b xsi:nil='true'/>
 </row>

 </a>
```

Stejného výsledku dosáhneme pomocí funkcí generujících XML. Jejich použití je univerzálnější, a o trochu komplikovanější:

```
pavel=# SELECT xmlelement(name a,
xmlagg(
xmlelement(name row,
xmlforest(a,b))))
FROM a;
                                xmlelement
--------------------------------------------------------------------------
 <a><row><a>2007-02-19</a><b>něco</b></row><row><a>2007-02-20</a></row></a>
(1 řádka)
```

V PostgreSQL stále chybí COLLATE. Podařilo se alespoň rozšířit klauzuli ORDER a to

v pozicování řádků s hodnotou NULL (ORDER BY .. NULLS FIRST/LAST). Adekvátně tomu se rozšířili parametry u btree indexů. Refaktorizací kódu se docílila podpora NULL v indexech. Starší verze nedokázaly indexovat hodnotu NULL.

```
postgres=# explain SELECT count(*) FROM fx WHERE a IS NULL;
                            QUERY PLAN
------------------------------------------------------------------
 Aggregate  (cost=8.28..8.29 rows=1 width=0)
   ->  Index Scan using bb on fx  (cost=0.00..8.28 rows=1 width=0)
         Index Cond: (a IS NULL)
(3 rows)
```

Nové datové typy a rozšíření možností stávajících datových typů

Ve verzi 8.2 PostgreSQL podporuje několik nových datových typů: XML zajišťující validitu obsahu, UUID (universal unique identifier) dle RFC 4122. Vlastní generátor je v contribu uuid-ossp (je potřeba doinstalovat package uuid a uuid-devel). K dispozici je deset různých způsobů generování jednoznačných univerzálních identifikátorů. Dále je tu možnost používat vlastní výčtové typy (zjevně inspirováno MySQL). Na rozdíl od MySQL v PostgreSQL je nutné před použitím vytvořit pro určitý seznam hodnot vlastní typ. Jeho použití je ovšem podstatně širší než v MySQL.

```
-- klasické řešení výčtu
CREATE TABLE foo(varianta char(2) CHECK (varianta IN ('a1','a2','a3')));

-- pouziti typu enum
CREATE TYPE vycet_variant AS ENUM('a1','a2','a3','a4','a5');
CREATE TABLE foo(varianta vycet_variant);
-- enum lze pouzit i v poli
SELECT '{a1,a3}'::vycet_variant[] as pripustne_varianty;
```

Rozsah hodnot získáme voláním funkce enum_range. Pokud funkci předáme parametr NULL, získáme úplný výčet hodnot.

```
postgres=# SELECT enum_range('a2'::va, 'a4'::vycet_variant);
 enum_range
------------
 {a2,a3,a4}
(1 row)

postgres=# SELECT enum_range(null::vycet_variant);
    enum_range
------------------
 {a1,a2,a3,a4,a5}
(1 row)
```

Kromě opravy několika chyb v PL/pgSQL (chyběla kontrola NOT NULL domén), došlo již k níže zmíněnému rozšíření příkazu RETURN o tabulkový výraz, a konečně lze i v PL/pgSQL používat scrollable kurzory. Ty PostgreSQL podporuje delší dobu, z PL/pgSQL je však nebylo možné používat. Kromě scrollable kurzorů lze v PL/pgSQL (ale i vně) používat updatable kurzory podle ANSI SQL 92 (oproti ANSI SQL 2003 přísnější omezení). U SRF funkcí můžeme upřesnit jejich náročnost a předpoklad počtu vrácených řádků (atributy COST a ROWS). V předchozích verzích se při hledání optimálního prováděcího plánu předpokládalo, že SRF funkce vrátí vždy 1000 řádků, což nebyla pokaždé být pravda (výsledkem byl neoptimální prováděcí plán).

Vedlejším efektem implementace subsystému pro kešování prováděcích plánů bylo odstranění problémů s neplatnými prováděcími plány v PL/pgSQL. Tyto problémy se projevovaly hlavně při použití dočasných tabulek, které se nesměly odstraňovat. Jinak docházelo, při použití SQL

příkazu vázaného na zrušenou a opětovně vytvořenou tabulku, k chybě. Nyní se cache čistí v závislosti na rušení databázových objektů. Samozřejmě, že funkce volané v cyklu budou prováděny efektivně jen tehdy, pokud nebude docházet k regenerování prováděcích plánů.

V 8.3 můžeme pole vytvářet i ze složených typů – v podstatě můžeme uložit tabulku jako jednu hodnotu). Stále však chybí podpora domén (a vkládaný záznam je nutné explicitně typovat):

```
postgres=# CREATE TYPE at AS (a integer, b integer);
CREATE TYPE
postgres=# CREATE TABLE foo(a at[]);
CREATE TABLE
postgres=# INSERT INTO foo VALUES(ARRAY[(10,20)::at]);
INSERT 0 1
postgres=# INSERT INTO foo VALUES(ARRAY[(10,20)::at, (20,30)::at]);
INSERT 0 1
postgres=# SELECT * FROM foo;
           a
-----------------------
 {"(10,20)"}
 {"(10,20)","(20,30)"}
(2 rows)

postgres=# SELECT a[1] FROM foo;
    a
---------
 (10,20)
 (10,20)
(2 rows)

postgres=# SELECT a[1].a FROM foo;
 a
----
 10
 10
(2 rows)
```

Optimalizace

Ve verzi 8.3 došlo k celé řada změn a úprav, které by měly vést k rychlejšímu zpracování SQL příkazů. Zrychlit by mělo načítání dat příkazem COPY. U tohoto příkazu není žádný důvod, proč by mělo docházet k zápisu do write ahead logu (ten je základem procesu obnovy po pádu PostgreSQL) a tato verze dokáže u tohoto příkazu obcházet zápis so WAL (COPY musí být v transakci). Nově PostgreSQL efektivněji provádí dotazy s ORDER BY c LIMIT n, kdy znatelně zrychlí výběr prvních n řádků řazených podle sloupce c, pokud nad sloupcem c není index (nedochází k seřazení celé tabulky). Příkaz EXPLAIN ANALYZE nyní poskytuje další informace o řazení (typ, spotřeba paměti):

```
t=# explain analyze SELECT * FROM foo ORDER BY a LIMIT 12;
                                                                  QUERY PLAN
-----------------------------------------------------------------------------------------------------------
 Limit  (cost=3685.48..3685.51 rows=12 width=4) (actual time=290.549..290.588 rows=12 loops=1)
   ->  Sort  (cost=3685.48..3935.48 rows=100000 width=4) (actual time=290.544..290.557 rows=12
             loops=1)
         Sort Key: a
         Sort Method:  top-N heapsort  Memory: 17kB
         ->  Seq Scan on foo  (cost=0.00..1393.00 rows=100000 width=4) (actual
                              time=0.036..153.526 rows=100000 loops=1)
 Total runtime: 290.658 ms
(6 rows)

t=# explain analyze SELECT * from foo order by a;
```

```
                                                        QUERY PLAN
-----------------------------------------------------------------------------------------------------
 Sort  (cost=10894.82..11144.82 rows=100000 width=4) (actual time=520.528..683.190
        rows=100000 loops=1)
   Sort Key: a
   Sort Method:  external merge  Disk: 1552kB
   ->  Seq Scan on foo  (cost=0.00..1393.00 rows=100000 width=4) (actual time=0.022..159.028
                         rows=100000 loops=1)
 Total runtime: 800.065 ms
(5 rows)
```

V předchozích verzích neexistovala hash funkce pro typ NUMERIC. Proto se pro spojování tabulek skrz sloupce typu NUMERIC nedala použít metoda HASHJOIN, která patří k nejrychlejším.

Zrychlit by měla i operace LIKE, zvlášť když se použije více bajtové kódování – použil se jiný algoritmus na porovnání řetězců. Nyní se již neporovnávají znaky, ale bajty, což ušetří jednu konverzi z UTF8 do UTF16. Na zkušební tabulce o sta tisících žlutých koních sekvenční čtení tabulky se zrychlilo z 169ms na 105ms.

Pokud se zjistí, že dochází k souběžnému sekvenčnímu čtení jedné tabulky z více obslužných procesů, tak se systém pokusí tyto procesy sesynchronizovat (pokud dojde k sekvenčnímu čtení, tak se ve velké většině případů čte celý datový soubor). Sekvenční čtení všech procesů je přibližně stejně rychlé, a tak je šance, že všechny procesy budou chtít v jednu chvíli stejnou datovou stránku, a je mnohem vyšší pravděpodobnost, že ji najdou ve vyrovnávací paměti. Pokud nedochází k synchronizaci procesu, tak pravděpodobnost, že požadovaná stránka je ve vyrovnávací paměti je mnohem menší, čímž se zvyšuje pravděpodobnost požadavku na fyzické čtení datové stránky se souboru. Tato optimalizace má smysl při větším počtu současně pracujících uživatelů, kdy je vyšší pravděpodobnost, že dojde k synchronizaci, a také kdy je větší tlak na vyrovnávací paměť.

Podpora asynchronního commitu je méně nebezpečnou obdobou nedoporučované konfigurace fsync=off. Při asynchronním commitu je zaručena konzistence databáze, nicméně při havárii hrozí riziko ztráty několika posledních transakcí. Parametr synchronous_commit je vázán na session, takže vývojář může na základě své úvahy zvolit méně bezpečný, nicméně rychlejší způsob řešení transakcí. Testy ukazují, že má smysl uvažovat o tomto parametru v případě málo zatížené databáze, kdy nedochází ke sdílení zápisu do transakčního logu – typicky při administraci databáze, kdy ostatní uživatelé nemají přístup k databázi a s databází pracuje pouze DBA.

8.3 obsahuje sofistikovanější metodu pro vytváření nových verzí označovanou jako HOT (Heap Only Tuples). Starší verze při jakékoliv operaci UPDATE modifikovaly relevantní indexy, a to i přesto, že nedošlo k modifikaci oindexovaných sloupců. Tzv. horký UPDATE je podmíněný dostatkem volného prostoru na datové stránce a změnou pouze neoindexovaných sloupců. Pokud tyto podmínky nejsou splněny, provede se klasický "studený" UPDATE. HOT UPDATE je vůči klasické implementaci operace mnohem úspornější a tudíž i rychlejší. Navíc tato nová metoda dokáže využít prostor na datové stránce obsazený nedostupnými verzemi (které byly vytvořeny také touto metodou) bez potřeby spuštění operace VACUUM.

Spuštění operace VACUUM ve více procesech

V 8.3 je automatické vacuování implementováno s podporou více procesů, tj. pokud trvá vacuum jedné databáze příliš dlouho, vytvoří se nový pracovní proces (worker), který zajišťuje vacuování dalších databází (smyslem není urychlit díky paralelizaci operaci VACUUM, ale zajistit, že v daném časovém okně se provede spravedlivě VACUUM všech databází). Úkolem pracovního procesu je projet provozní statistiky všech tabulek v databázi a vybrat tabulky určené k vacuování. Pracovní proces je na úrovni databáze sekvenční, paralelizace je na úrovni clusteru. Nově vacuum také zajišťuje samočinné volání VACUUM FREEZE coby ochrany před přetečením rozsahu identifikátorů verzí řádků.

Rozšíření PL/pgSQL – RETURN QUERY, lokální systémové proměnné

Předchozí verze neumožňovaly vrátit množinu záznamů jako výsledek SRF funkce. Jediným řešením bylo volání příkazu RETURN NEXT pro každý řádek výsledku dotazu. V podstatě totéž (ale na nižší úrovni, tudíž efektivněji) provádí příkaz RETURN QUERY. Jeho parametrem je SQL dotaz, jehož výsledek se připojí k výstupu. Podobně jako RETURN NEXT neukončuje provádění funkce.

```
CREATE OR REPLACE FUNCTION dirty_series(m integer, n integer)
RETURNS SETOF integer AS $$
BEGIN
  RETURN QUERY SELECT * FROM generate_series(1,m) g(i)
                WHERE i % n = 0;
  RETURN;
END; $$ LANGUAGE plpgsq;
```

Další novou vlastností je možnost modifikovat systémové proměnné lokálně pro určitou funkci. Podobně se chová T-SQL nebo MySQL, kde se ukládá aktuální nastavení systémových proměnných v čase registrace funkce. V PostgreSQL žádný podobný mechanismus nebyl. Tato vlastnost řeší zabezpečení SECURITY DEFINER funkcí, kterým bylo možné podvrhnout útočníkův kód změnou systémové proměnné search_path. Zápis je zřejmý z následujícího příkladu:

```
CREATE FUNCTION report_guc(text) RETURNS TEXT AS
  $$ SELECT current_setting($1) $$ LANGUAGE sql
  SET regex_flavor = basic;

ALTER FUNCTION report_guc(text)
  RESET search_path
  SET regex_flavor = extended;
```

Podpora režimu Warm Standby, prototyp replikace založené na transakčním logu

PostgreSQL 8.3 umožňuje nakonfigurovat a používat dva PostgreSQL servery tak, že první slouží jako výkonný server a druhý jako záložní, kdy změny v datech na prvním serveru jsou na druhý server replikovány exportem transakčního logu. Tato konfigurace se používá pouze v náročných aplikací, kde časté klasické zálohování z důvodu objemu není možné a kde ztráta dat není akceptovatelná – kde zákazník vyžaduje průběžné zálohování. Nejde o multi-master replikaci jako v případě MySQL. Druhý systém je až do signálu nedostupný, tudíž tímto způsobem replikace nelze rozložit zátěž serveru. Pro usnadnění konfigurace verze 8.3 obsahuje příkaz pg_standby (ve stejnojmenném contrib adresáři), který zajistí udržení druhé instance PostgreSQL v režimu Warm Standby.

```
Master (postgresql.conf):
archive_command = 'cp %p ../archive/%f'
archive_timeout = 20

Warm Standby (recovery.conf)
restore_command = 'pg_standby -l -d -k 255 -r 2 -s 2 -w 0 -t /tmp/stop /usr/local/pgsql/archive %f %p \
                  2>> standby.log'
```

Po modifikaci konfiguračních souborů stačí spustit oba servery. Záložní server zůstane v recovery režimu, kdy pg_standby postupně podvrhuje segmenty transakčního logu (sleduje exportované segmenty) a nedovolí dokončit obnovu záložní databáze. Teprve po signalizaci, pg_standby (existencí předem určeného souboru (v příkladu /tmp/stop)) oznámí serveru, že se jednalo o poslední segment transakčního logu a dovolí dokončení obnovy a tím i přepnutí do stavu, kdy záložní server je schopen přijímat požadavky. Signální soubor musí vygenerovat uživatel postgres, tak aby jej pg_standby mohlo odstranit. Pokud tento soubor nelze odstranit, replikace zhavaruje. V praxi toto řešení není příliš použitelné, neboť pg_standby nedokáže zachytit výjimku a tudíž ji ani nedokáže korektně obsloužit (aniž by nebyl nevratně přerušen proces replikace spojený se ztrátou dat na záložním serveru).

```
4916 ?        S      0:00 /usr/local/pgsql/bin/postmaster -D /usr/local/pgsql2/data
4918 ?        Ss     0:00 postgres: startup process
5226 ?        S      0:00 sh -c pg_standby -l -d -t
/tmp/aaa /usr/local/pgsql/archive 000000010000000000000018 pg_xlog/RECOVERYXLOG 2>> standby.log
5227 ?        S      0:00 pg_standby -l -d -t
/tmp/aaa /usr/local/pgsql/archive 000000010000000000000018 pg_xlog/RECOVERYXLOG
```

Záložní cluster musí být klonem zálohovaného clusteru. Musí být vytvořen zkopírováním adresáře databázového clusteru – nevytváří se příkazem initdb.

Regulární výrazy

Podpora reg. výrazů není v PostgreSQL novinkou. V 8.3 se objevily nové funkce regexp_matches a dvojice regexp_split_to_array a regexp_split_to_table. Pro řadu úloh nyní nemusíme používat plperl. V následujícím příkladu je z XML dokumentu separován seznam identifikačních čísel, který je následně indexován a použit k vyhledávání. Ke stejnému účelu by bylo možné použít i funkce podporující XPath výrazy. Toto řešení je řádově rychlejší než intuitivní (a velice pomalé) řešení s LIKE.

```
objednava_v_xml LIKE '%<id>hledane_id</id>%'
```

Pole NEW.objednavka_id_produktu je aktualizované v triggeru:

```
NEW.objednavka_id_produktu := ARRAY(SELECT i[1] FROM
regexp_matches(NEW.objednavka_v_xml,'<id>(\\d+)</id>','g' r(i));
```

Funkčně srovnatelný predikát výše uvedenému LIKE je:

```
objednavka_id_produktu @> ARRAY[hledane_id]
```

Ostatní změny

Nezanedbatelného rozšíření se dočkalo prostředí ecpg. Vylepšuje podporu prepared statements, nabízí auto prepare mód, pozicované proměnné. Jedná o zásadní změny – došlo ke změně verze z 4.4 na 6.0. Jednou z prvních backportů z EnterpriseDB je debugger a profiler PL/pgSQL. Nová verze pgAdminIII obsahuje grafické rozhraní debuggeru, které je zpřístupněno, pokud je v PostgreSQL nainstalován plugin s debuggerem (ke stažení na pgfoundry). Ve srovnání s moderními debuggery obsahuje PL/pgSQL pouze základní funkce.

```
pavel=# LOAD '$libdir/plugins/plugin_profiler';
LOAD
pavel=# SET plpgsql.profiler_tablename = 'profilerstats';
SET
pavel=# SELECT line_number, sourcecode, time_total, exec_count, func_oid::regproc
          FROM profilerstats
          ORDER BY 1;
 line_number |      sourcecode       | time_total | exec_count | func_oid
-------------+-----------------------+------------+------------+----------
           0 |                       |          0 |          0 | x
           1 | begin                 |          0 |          0 | x
           2 |   for i in 1..4 loop  |   0.000315 |          1 | x
           3 |     return next i;    |    9.8e-05 |          4 | x
           4 |   end loop;           |          0 |          0 | x
           5 |   return;             |      3e-06 |          1 | x
(6 rows)
```

Index advisor

Index advisor je plugin plánovače dotazů. Jedná se o prototyp navržený Tomem Lanem za účelem demonstrace monitorovacího rozhraní návrhu a optimalizace prováděcích plánů. Pokud se aktivuje, tak optimalizátor bere v úvahu, kromě existujících indexů, hypotetické indexy vytvořené nad každým sloupcem:

```
regression=# load '/home/tgl/pgsql/advisor';
LOAD
regression=# explain select * from fooey order by unique2,unique1;
                                    QUERY PLAN
-----------------------------------------------------------------------------------
 Sort  (cost=809.39..834.39 rows=10000 width=8)
   Sort Key: unique2, unique1
   ->  Seq Scan on fooey  (cost=0.00..145.00 rows=10000 width=8)

 Plan with hypothetical indexes:
 Index Scan using <hypothetical index> on fooey  (cost=0.00..376.00 rows=10000 width=8)
(6 rows)

regression=# explain select * from fooey where unique2 in (1,2,3);
                                   QUERY PLAN
--------------------------------------------------------------------------------
 Seq Scan on fooey  (cost=0.00..182.50 rows=3 width=8)
   Filter: (unique2 = ANY ('{1,2,3}'::integer[]))

 Plan with hypothetical indexes:
 Bitmap Heap Scan on fooey  (cost=12.78..22.49 rows=3 width=8)
   Recheck Cond: (unique2 = ANY ('{1,2,3}'::integer[]))
   ->  Bitmap Index Scan on <hypothetical index>  (cost=0.00..12.77 rows=3 width=0)
         Index Cond: (unique2 = ANY ('{1,2,3}'::integer[]))
(8 rows)
```

Vývoj v následujících letech

Největší slabinou PostgreSQL je chybějící podpora replikaci. V této oblasti nikoliv nezaslouženě dominují komerční systémy. Dále PostgreSQL nemá dořešenou internacionalizaci, tzv. COLLATES, které nabízí jak MySQL, tak Firebird. Konečně třetí oblastí, kterou je nyní třeba intenzivně se zabývat je podpora OLAP databází. Nelze předpokládat, že by PostgreSQL v brzké době podporoval OLAP databáze, nicméně existují určité indicie, že hlavním tématem následující verze (8.4) bude podpora analytických a rekurzivních dotazů. V delším

časovém horizontu je možné očekávat zařazení podpory zpracování tzv. proudových dat, jelikož platforma PostgreSQL byla použita k vytvoření experimentálních prototypů proudových databází a kromě toho, část týmu vývojářů se touto problematikou aktivně zabývá.

Odkazy

1. Přehled vlastností jednotlivých verzí[1]
2. Přehled plánovaných rozšíření v příští verzi[2]

[1] http://developer.postgresql.org/index.php/Feature_Matrix
[2] http://developer.postgresql.org/index.php/Todo:WishlistFor84