PostGIS pro vývojáře

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, GIS, development

Abstrakt

Systémy GIS první generace ukládaly svá data do souborů v proprietárních formátech. Další generace těchto systémů dokázaly spolupracovat s databázemi (zejména čerpat data s databází). Soudobé GIS systémy se již zcela spoléhají na databáze. Tyto požadavky GIS systémů musely reflektovat databázové systémy. Nejstarší SQL systémy vůbec s prostorovými daty nepočítaly.

Úvod

Až standard ANSI SQL2000 v části věnované podpoře multimediálních dat obsahuje popis prostorových (spatial) dat. Z komerčních databází je na prvním místě, co se týče podpory prostorových dat, databázový systém fy. Oracle. Počínaje verzí Oracle 7 existuje rozšíření Oracle, samostatně distribuované, řešící podporu prostorových dat. Toto rozšíření se nazývá Oracle spatial. Odpovědí o.s. světa byla implementace podpory prostorových dat pro RDBMS PostgreSQL a to tak, jak předpokládá standard OpenGIS.

Jednou z charakteristik PostgreSQL je právě jeho rozšiřitelnost. V PostgreSQL lze relativně jednoduše navrhnout vlastní datové typy, vlastní operace a operandy nad těmito typy. Touto vlastností byl systém PostgreSQL mezi o.s. databázovými systémy výjimečný. Proto celkem logicky byl PostgreSQL použit pro o.s. implementaci standardu OpenGIS. Část standardu OpenGIS (chronologicky předchází SQL2000) je zaměřena na SQL databáze, které by měly sloužit jako uložiště prostorových dat. Vychází z SQL92 rozšířeného o geometrické typy (SQL92 with Geometry Types), definuje metadata popisující funkcionalitu systému co se týká geometrických dat, a definuje datové schéma. Databáze, jejichž datový model respektuje normu OpenGIS, může sloužit jako datový server libovolné GIS aplikaci, která vychází z tohoto standardu. Díky tomu může, v celé řadě případů, PostgreSQL zastoupit komerční db systémy. Implementace standardu OpenGIS pro PostgreSQL se nazývá PostGIS. PostgreSQL základní geometrické typy má, úkolem PostGISu je hlavně jejich obalení do specifického (určeného normou) datového modelu. SQL/MM-Spatial sice vychází z OpenGIS nicméně není kompatibilní. PostGIS je certifikován pro OpenGIS, částečně také implementuje SQL/MM.

PostGIS obsahuje:

- nové datové typy (geometry),
- nové operátory (&& průnik, @kompletně obsažen),
- nové funkce (distance, transform),
- nové tabulky (Geometry_columns, Spatial_ref_sys) slouží jako systémové tabulky, poskytují prostor pro metadata.

Standard OpenGIS je množina dokumentů detailně popisující aplikační rozhraní a datové formáty v oblasti GIS systémů. Tato dokumentace je určena vývojářům a jejím cílem je dosažení interoperability aplikací vyvíjených členy konsorcia Open Geospatial Consortium (www stránky OGC). Oblast, kterou pokrývá PostGIS je popsána v dokumentu "OpenGIS® Simple Features Specification For SQL"

Názvy datových typů v SQL/MM odvozených z OpenGISu vznikly spojením prefixu ST_ (spatial type) a názvu datového typu v OpenGISu. OpenGIS je obecnější, počítá s minimálně dvěma variantami implementace geometrických typů, a k tomu potřebnému zázemí. SQL/MM-Spatial se dá s jistou mírou tolerance chápat jako podmnožina OpenGISu.

Implementace vlastních funkcí v PostgreSQL

Vzhledem k faktu, že vlastní datové typy lze implementovat pouze v prg. jazyce C a že PostGIS je implementován v C, bude popsán návrh modulu pouze s využitím jazyka C.

Při návrhu funkcí v C se prakticky všude používají makra z PostgreSQL knihovny. Důvodů je několik:

- používání univerzálního typu Varlena pro typy s variabilní délkou, který je v případě, že je delší než 2K transparentně (jak pro uživatele, tak pro vývojáře) komprimován.
- odlišný způsob předávání parametrům (v PostgreSQL nepřímo prostřednictvím určené hodnoty typu struct obsahující ukazatel na pole parametrů, pole s informací, který parametr je NULL, a počtem parametrů).
- tento způsob volání není závislý na programovacím jazyku C

Ukázka implementace funkce concat_text spojující dva řetězce dohromady:

```
01 PG_FUNCTION_INFO_V1(concat_text);
02
03 Datum
04 concat_text(PG_FUNCTION_ARGS)
05 {
06     text  *arg1 = PG_GETARG_TEXT_P(0);
07     text  *arg2 = PG_GETARG_TEXT_P(1);
08     int32 new_text_size = VARSIZE(arg1) + VARSIZE(arg2) - VARHDRSZ;
09     text *new_text = (text *) palloc(new_text_size);
10
11     VARATT_SIZEP(new_text) = new_text_size;
12     memcpy(VARDATA(new_text), VARDATA(arg1), VARSIZE(arg1) - VARHDRSZ);
13     memcpy(VARDATA(new_text) + (VARSIZE(arg1) - VARHDRSZ),
14            VARDATA(arg2), VARSIZE(arg2) - VARHDRSZ);
15     PG_RETURN_TEXT_P(new_text);
16 }
```

Komentáře:

01 Registrace funkce s volací konvencí 1.

03 Každá funkce dosažitelná z SQL musí vracet univerzální datový typ Datum (společný typ pro datové typy s fixní velikostí (menší než 64bite) a datový typ varlena).

06 Získání prvního parametru (resp. ukazatele na něj a korektní přetypování, případně dekomprimace).

07 Získání prvního parametru (resp. ukazatele na něj a korektní přetypování, případně dekomprimace).

08 Datový typ varlena je podobný stringu v Pascalu, první čtyři byte obsahují údaj o délce s rozdílem, že údaj obsahuje velikost celé hodnoty včetně záhlaví. Velikost hlavičky je uložena v konst. VARHDRSZ. Výpočet velikosti vrácené hodnoty typu datum (součet velikostí obou řetězců + velikost hlavičky).

09 PostgreSQL má vlastní správu paměti, tudíž se paměť nealokuje voláním funkce malloc, ale palloc. PostgreSQL přiděluje paměť z tzv. paměťových kontextů (persistentní, transakce, volání funkce). Při dokončení operace se uvolňuje odpovídající paměťový kontext. Explicitní volání pfree má smysl jen u funkcí, které by bez explicitního uvolnění paměti si nárokovali příliš paměti. Použití paměťových kontextů snižuje riziko tzv. memory leaku, tj. že vývojář zapomene vrátit alokovanou paměť. Také snižují náročnost vrácení paměti. Místo několikanásobného uvolnění malých bloků paměti se volá jednorázová operace. Dalším příjemným efektem je nižší fragmentace paměti.

15 Přetypování z typu text na datum (případná komprimace).

Každá interní funkce se musí před vlastním použitím zaregistrovat pro její použití v SQL.

```
01 CREATE FUNCTION concat_text(text, text) RETURNS text
02      AS 'DIRECTORY/funcs', 'concat_text'
03      LANGUAGE C STRICT;
```

Komentáře:

01 Funkce concat_text má dva parametry typu text a vrací text.

02 Tuto funkci PostgreSQL nalezne v knihovně (souboru) 'DIRECTORY/funcs' pod názvem 'concat_text'.

03 Jedná se o binární knihovnu. V případě, že jeden parametr je NULL, výsledkem volání funkce je hodnota NULL (atribut STRICT).

Pro volání existujících PostgreSQL funkcí (pod volající konvencí 1) musíme použít specifický způsob jejich volání, resp. předání parametrů. Často používanou funkcí je textin, což je funkce, která slouží pro převod z-řetězce (klasického řetězce v jazyce C ukončeného nulou) na řetězec typu varlena. Následující funkce vrátí konstantní řetězec "Hello World!".

```
01 PG_FUNCTION_INFO_V1(hello_world);
02
03 Datum
04 hello_word(PG_FUNCTION_ARGS)
05 {
06 PG_RETURN_DATUM(DirectFunctionCall1(textin, CStringGetDatum("Hello World!")));
07 }
```

Komentáře:

06 CStringGetDatum provádí pouze přetypování

Bez použití funkce DirectFunctionCall1(1 na konci názvu funkce má význam jeden argument. Tato funkce existuje ve variantách pro jeden až devět argumentů.) by výše zmíněná funkce vypadala následovně (z ukázky je vidět předávání parametrů v V1 volající konvenci):

```
01 PG_FUNCTION_INFO_V1(hello_world);
02
03 Datum
04 hello_word(PG_FUNCTION_ARGS)
05 {
06 FunctionCallInfoData locfcinfo;
07
08 InitFunctionCallInfoData(locfcinfo, fcinfo->flinfo, 1, NULL, NULL);
09
10 locfcinfo.arg[0] = CStringGetDatum("Hello World!")
11 locfcinfo.argnull[0] = false;
12
13 PG_RETURN_DATUM(textin(&locfinfo));
14 }
```

Komentáře:

08 Datová struktura locfcinfo je inicializována pro jeden argument.

13 Přímé volání funkce textin. Jelikož tato funkce vrací typ Datum, který je výsledný typ funkce hello_world, nedochází k přetypování.

Implementace vlastních datových typů v PostgreSQL

V PostgreSQL je každý datový typ určen svojí vstupní a výstupní, a sadou operátorů a funkcí, které tento typ podporují. Vstupní funkce je funkce, která převádí řetězec na binární hodnotu. Výstupní funkce inverzně převádí binární hodnotu na řetězec. Podporované operátory a funkce pak pracují s binární hodnotou. V případě vkládání nových záznamů se vstupní funkce volají automaticky, v případě výrazů je nutné v některých případech volat explicitní konverzi.

Explicitní konverze se v PostgreSQL provede třemi různými způsoby:

- zápisem typu následovaný řetězcem
- použitím ANSI SQL operátoru CAST
- použitím binárního operátoru pro přetypování '::' (není standardem)

Ukázka:

```
SELECT rodne_cislo '7307150xxx';
SELECT CAST('730715xxx' AS rodne_cislo);
SELECT '730715xxx'::rodne_cislo;
```

OpenGIS, coby nezávislý standard, přidává vlastní způsob zápisu. V OpenGISu je počítáno i s variantou, že data jsou uložená textově, v případě že databázový systém nelze rozšířit o geometrické typy (což není případ PostgreSQL). Nicméně PostGIS nepoužívá geometrické typy PostgreSQL. Typ se zapisuje přímo do řetězce následovaný vlastními daty, které jsou uzavřené v závorkách. Tento zápis se označuje jako 'Well-Known Text (WKT)'. Pro přečtení hodnoty tohoto typu se používá funkce GeometryFromText.

Ukázka:

```
INSERT INTO SPATIALTABLE ( THE_GEOM, THE_NAME ) VALUES
       ( GeomFromText('POINT(-126.4 45.32)', 312), 'A Place');
```

Kromě textového formátu je v OpenGISu ještě definován binární formát 'Well-Know Binary (WKB)'. Konverzi z binárního formátu do textového (čitelná podoba) formátu provádí funkce asewkt (astext). Interně PostGIS ukládá data v binárním formátu WKB.

Pokud máme vstupní a výstupní funkci k dispozici, můžeme zaregistrovat nový datový typ příkazem CREATE TYPE.

```
01 CREATE OR REPLACE FUNCTION rodne_cislo_in (cstring)
02   RETURNS rodne_cislo AS 'rc.so','rodne_cislo_in' LANGUAGE 'C';
03
04 CREATE OR REPLACE FUNCTION rc_out (rodne_cislo)
05   RETURNS cstring AS 'rc.so', 'rodne_cislo_out' LANGUAGE 'C';
06
07 CREATE TYPE rodne_cislo (
08   internallength = 8,
09   input          = rc_in,
10   output         = rc_out
11 );
```

Komentáře:

08 Jedná se o datový typ s pevnou délkou osmi bajtů.

Pokud chceme datovým typ použít v databázi, musíme implementaci datového typu rozšířit o implementaci základních binárních operátorů. Poté bude možné použít vlastní datový typ v klauzuli WHERE a ORDER BY.

```
01 CREATE OR REPLACE FUNCTION rodne_cislo_eq (rodne_cislo, rodne_cislo)
02  RETURNS bool AS 'rc.so','rodne_cislo_eq' LANGUAGE 'C' STRICT;
03
04 CREATE OR REPLACE FUNCTION rodne_cislo_ne (rodne_cislo, rodne_cislo)
05   RETURNS bool AS 'rc.so', 'rodne_cislo_ne' LANGUAGE 'C' STRICT;
06
07 CREATE OR REPLACE FUNCTION rodne_cislo_lt (rodne_cislo, rodne_cislo)
08   RETURNS bool AS 'rc.so','rodne_cislo_lt' LANGUAGE 'C' STRICT;
09
10 CREATE OR REPLACE FUNCTION rodne_cislo_le (rodne_cislo, rodne_cislo)
11  RETURNS bool AS 'rc.so', 'rodne_cislo_le' LANGUAGE 'C' STRICT;
12
13 CREATE OPERATOR = (
14   leftarg   = rodne_cislo,
15   rightarg  = rodne_cislo,
16   procedure = rodne_cislo_eq
17   commutator = =,
18   negator    = <>,
19   restrict   = eqsel,
20   join       = eqjoinsel
21 );
22
23 CREATE OPERATOR <> (
24   leftarg   = rodne_cislo,
25   rightarg  = rodne_cislo,
26   procedure = rodne_cislo_ne
27 );
28
29 CREATE OPERATOR <(
30   leftarg   = rodne_cislo,
31   rightarg  = rodne_cislo,
32   procedure = rodne_cislo_le
33 );
34
```

```
35 CREATE OPERATOR <= (
36   leftarg   = rodne_cislo,
37   rightarg  = rodne_cislo,
38   procedure = rodne_cislo_le
39 );
```

Komentáře:

13 Registrace binárního operátoru rovno.

14 Levý argument je typu rodné číslo.

15 Pravý argument je typu rodné číslo.

16 Název funkce, která zajišťuje operaci porovnání pro datový typ rodne_cislo.

17 Rovná se je komutátorem sama sebe, neboť platí že $x = y <=> y = x$.

18 Platí, že $x = y <=> NOT(x <> y)$.

19 Operátor je silně restriktivní v případě, že jedním argumentem je konstanta, tj. výsledkem je malá podmnožina tabulky.

20 Operátoru se přiřazuje funkce odhadu selektivity.

Výše uvedené operátory stále nestačí k tomu, aby se nad sloupcem s vlastním typem mohl vytvořit index. Každý datový typ musí mít definovanou alespoň jednu třídu operátorů, což je v podstatě seznam operátorů doplněný o jejich sémantický význam. Kromě operátorů je potřeba určit tzv. podpůrnou funkci $F(a, b)$, jejíž parametry jsou klíče a výsledkem celé číslo $(a > b => F(a, b) = 1; a = b => F(a, b) = 0; a < b => F(a, b) = -1)$.

```
01 CREATE OR REPLACE FUNCTION rodne_cislo_cmp (rodne_cislo, rodne_cislo)
02   RETURNS int AS 'rc.so', 'rodne_cislo__cmp' LANGUAGE 'C' STRICT;
03
04 CREATE OPERATOR CLASS rodne_cislo_ops
05   DEFAULT FOR TYPE rodne_cislo USING btree AS
06     OPERATOR  1    <,
07     OPERATOR  2    <=,
08     OPERATOR  3    = ,
09     OPERATOR  4    >=,
10     OPERATOR  5    >,
11     FUNCTION  1    rodne_cislo_cmp (rodne_cislo, rodne_cislo);
```

Komentáře:

06 Strategie 1 má význam menší než.

07 Strategie 2 má význam menší rovno než.

08 Strategie 3 má význam rovno.

09 Strategie 4 má význam vetší rovno než.

10 Strategie 5 má význam větší než.

11 Určení podpůrné funkce.

Ukázka použití PostGISu

OpenGIS předpokládá uložení dat do klasických databázových tabulek. Nicméně k tomu, aby tyto tabulky dokázali přečíst GIS aplikace je nutné do datového schématu přidat dvě systémové (z pohledu OpenGIS) tabulky.

geometry_columns obsahuje informace o sloupcích geometrii geoprvků,

spatial_ref_sys obsahuje informace o souřadnicových systémech používaných systémem.

```
01 create table user_locations (gid int4, user_name varchar);
02 select AddGeometryColumn ('db_mapbender','user_locations','the_geom','4326','POINT',2);
03 insert into user_locations values ('1','Morissette',GeometryFromText('POINT(-71.060316 48.432044)',
4326));
```

Komentáře:

01 vytvoření tabulky faktů (feature table)

02 do tabulky user_location přidá sloupec s názvem the_geom (následně přidá do tabulky geometry_columns nový řádek s metadaty o sloupci the_geom)

03 Plnění tabulky daty

Kromě plnění datových tabulek pomocí SQL příkazů PostGIS obsahuje nástroj pro import datových (shape) souborů, tzv. shape loader. Díky němu je možné importovat data v několika formátech. Namátkou podporované formáty jsou Shape, MapInfo, DGN, GML.

K urychlení operací prováděných nad prostorovými daty lze použít prostorový index. PostgreSQL podporuje několik typů indexů, pro GIS lze použít (ve verzi 8.2) formáty GIST a GIN.

```
01 CREATE INDEX <indexname>
02 ON <tablename>
03 USING GIST ( <geometrycolumn> [ GIST_GEOMETRY_OPS ] );
```

Komentáře:

03 GIST_GEOMETRY_OPS určuje výše zmíněnou třídu operátorů.

Analýza obsahu distribuce PostGIS 1.2.1

Struktura adresáře:

- ./ – Sestavovací a instalační skripty
- ./lwgeom – Zdrojový kód knihoven
- ./java/ejb – Podpora EJB Java
- ./java/jdbc – JDBC ovladač pro PostgreSQL rozšířený o podporu GIS objektů
- ./java/pljava – PostgreSQL PL/Java rozšířená o prostorové objekty
- ./doc – Dokumentace
- ./loader – Programy zajišťující konverzi ESRI Shape souborů do SQL (resp. PostGIS) a inverzní transformaci PostGIS dat do Shape souborů (pgsql2shp a shp2pgsql)

- ./topology – Počáteční implementace modelu topology
- ./utils – Pomocné skripty (aktualizace, profilace)
- ./extras – Kód, který se nedostal do hlavního stromu (WFS_locks, ukázka wkb parseru)
- ./regress – Regresní testy

Závislosti:

- proj4 - knihovna realizující transformace mezi projekcemi
- geos – knihovna implementující topologické testy (PostgreSQL je třeba překládat s podporou C++)

Typ LWGEOM nahrazuje původní typ GEOMETRY PostGISu. Oproti němu je menší (data jsou uložená binárně – pro POINT(0,0) je to úspora z 140 bajtů na 21 bajtů), podporuje 1D, 2D, 3D a 4D souřadnice, interně vychází z OGC WKB typu a také jeho textová prezentace je OGC WKB. Typ LWGEOM nahradil předchozí typ GEOMETRY ve verzi 1.0. LW znamená Light Weight (vylehčený). Interně v PostgreSQL se používá identifikátor PG_LWGEOM. Hodnoty se serializují rekurzivně, za hlavičkou specifikující typ a atributy se serializují vlastní data.

Jádro PostGISu je schované v implementaci typu LWGEOM. Jako parser je, v prostředí UNIX obvyklý, použitý generátor překladačů Lex a yacc (konkrétně jejich implementace Bison). Syntaxe je určena v souborech wktparse.lex (lexikální elementy, klíčová slova, čísla) a wktparse.y (syntaktické elementy)

```
01 /* MULTIPOINT */
02
03 geom_multipoint :
04         MULTIPOINT { alloc_multipoint(); } multipoint  { pop(); }
05         |
06         MULTIPOINTM { set_zm(0, 1); alloc_multipoint(); } multipoint {pop(); }
07
08 multipoint :
09         empty
10         |
11         { alloc_counter(); } LPAREN multipoint_int RPAREN { pop(); }
12
13 multipoint_int :
14         mpoint_element
15         |
16         multipoint_int COMMA mpoint_element
17
18 mpoint_element :
19         nonempty_point
20         |
21         /* this is to allow MULTIPOINT(0 0, 1 1) */
22         { alloc_point(); } a_point { pop(); }
```

Komentáře:

03 Povoleným zápisem je MULTIPOINT seznam nebo MULTIPOINTM seznam.

08 Seznam může být prázdný nebo je posloupností čísel uzavřený v závorkách.

13 Seznam je buďto o jednom prvku nebo seznam a čárkou oddělený element.

16 Rekurzivní definice seznamu.

22 a_point je 2D 3D 4D hodnota zapsaná jako posloupnost n čísel oddělených mezerou.

Serializace a deserializace načteného syntaktického stromu je řešena v souboru lwgeom_inout.c.

```
01 CREATE FUNCTION geometry_in(cstring)
02         RETURNS geometry
03         AS '@MODULE_FILENAME@','LWGEOM_in'
04         LANGUAGE 'C' _IMMUTABLE_STRICT; -- WITH (isstrict,iscachable);
05
06 CREATE FUNCTION geometry_out(geometry)
07         RETURNS cstring
08         AS '@MODULE_FILENAME@','LWGEOM_out'
09         LANGUAGE 'C' _IMMUTABLE_STRICT; -- WITH (isstrict,iscachable);
10
11 CREATE TYPE geometry (
12         internallength = variable,
13         input = geometry_in,
14         output = geometry_out,
15 );
```

Definice typu geometry je v souboru lwpostgis.sql.in spolu s definicemi dalších desítek databázových objektů. Zcela zásadní jsou tabulky spatial_ref_sys a geometry_columns.

```
01 ----------------------------------------------------------------------
02 -- SPATIAL_REF_SYS
03 ----------------------------------------------------------------------
04 CREATE TABLE spatial_ref_sys (
05          srid integer not null primary key,
06          auth_name varchar(256),
07          auth_srid integer,
08          srtext varchar(2048),
09          proj4text varchar(2048)
10 );
11
12 ----------------------------------------------------------------------
13 -- GEOMETRY_COLUMNS
14 ----------------------------------------------------------------------
15 CREATE TABLE geometry_columns (
16         f_table_catalog varchar(256) not null,
17         f_table_schema varchar(256) not null,
18         f_table_name varchar(256) not null,
19         f_geometry_column varchar(256) not null,
20         coord_dimension integer not null,
21         srid integer not null,
22         type varchar(30) not null,
23         CONSTRAINT geometry_columns_pk primary key (
24                 f_table_catalog,
25                 f_table_schema,
26                 f_table_name,
27                 f_geometry_column )
28 ) WITH OIDS;
```

Řada funkcí PostGISu jsou realizována v jazyce PL/pgSQL. Což je jazyk SQL procedur v prostředí PostgreSQL vycházející z PL/SQL fy. Oracle (který vychází z prg. jazyka ADA). Je to celkem logické, díky integraci SQL jsou SQL příkazy zapsány úsporně a čitelně.

```
001 ----------------------------------------------------------------------
002 -- ADDGEOMETRYCOLUMN
003 --   <catalogue>, <schema>, <table>, <column>, <srid>, <type>, <dim>
004 ----------------------------------------------------------------------
005 --
006 -- Type can be one of geometry, GEOMETRYCOLLECTION, POINT, MULTIPOINT, POLYGON,
007 -- MULTIPOLYGON, LINESTRING, or MULTILINESTRING.
008 --
009 -- Types (except geometry) are checked for consistency using a CHECK constraint
010 -- uses SQL ALTER TABLE command to add the geometry column to the table.
011 -- Addes a row to geometry_columns.
```

```
012 -- Addes a constraint on the table that all the geometries MUST have the same
013 -- SRID. Checks the coord_dimension to make sure its between 0 and 3.
014 -- Should also check the precision grid (future expansion).
015 -- Calls fix_geometry_columns() at the end.
016 --
017 -----------------------------------------------------------------------
018 CREATEFUNCTION AddGeometryColumn(varchar,varchar,varchar,varchar,integer,varchar,integer)
019  RETURNS text
020  AS
021 '
```

Komentáře:

018 Tento zdrojový kód se v PostGISu zpracovává ještě preprocesorem, takže na první pohled neplatné klíčové slovo CREATEFUNCTION je správné. Důvodem je potřeba jedné verze zdrojových kódů použitelných pro různé verze PostgreSQL, kdy mezi nejstarší verzí 7.4 a nejnovější 8.2 je zřetelný rozdíl v možnostech a i v zápisu uložených procedur. Jinak tyto verze od sebe dělí tři roky. Přestože oficiálně nejstarší podporovaná verze je 7.4, v kodu je řada odkazů na verzi 7.2.

```
022 DECLARE
023  catalog_name alias for $1;
024  schema_name alias for $2;
025  table_name alias for $3;
026  column_name alias for $4;
027  new_srid alias for $5;
028  new_type alias for $6;
029  new_dim alias for $7;
030  rec RECORD;
031  schema_ok bool;
032  real_schema name;
033 BEGIN
034
035  IF ( not ( (new_type =''GEOMETRY'') or
036     (new_type =''GEOMETRYCOLLECTION'') or
037     (new_type =''POINT'') or
038     (new_type =''MULTIPOINT'') or
039     (new_type =''POLYGON'') or
040     (new_type =''MULTIPOLYGON'') or
041     (new_type =''LINESTRING'') or
042     (new_type =''MULTILINESTRING'') or
043     (new_type =''GEOMETRYCOLLECTIONM'') or
044     (new_type =''POINTM'') or
045     (new_type =''MULTIPOINTM'') or
046     (new_type =''POLYGONM'') or
047     (new_type =''MULTIPOLYGONM'') or
048     (new_type =''LINESTRINGM'') or
049     (new_type =''MULTILINESTRINGM'') or
050                     (new_type = ''CIRCULARSTRING'') or
051                     (new_type = ''CIRCULARSTRINGM'') or
052                     (new_type = ''COMPOUNDCURVE'') or
053                     (new_type = ''COMPOUNDCURVEM'') or
054                     (new_type = ''CURVEPOLYGON'') or
055                     (new_type = ''CURVEPOLYGONM'') or
056                     (new_type = ''MULTICURVE'') or
057                     (new_type = ''MULTICURVEM'') or
058                     (new_type = ''MULTISURFACE'') or
059                     (new_type = ''MULTISURFACEM'')) )
060  THEN
061  RAISE EXCEPTION ''Invalid type name - valid ones are:
062  GEOMETRY, GEOMETRYCOLLECTION, POINT,
063  MULTIPOINT, POLYGON, MULTIPOLYGON,
064  LINESTRING, MULTILINESTRING,
065                         CIRCULARSTRING, COMPOUNDCURVE,
```

```
066                         CURVEPOLYGON, MULTICURVE, MULTISURFACE,
067  GEOMETRYCOLLECTIONM, POINTM,
068  MULTIPOINTM, POLYGONM, MULTIPOLYGONM,
069  LINESTRINGM, MULTILINESTRINGM
070                         CIRCULARSTRINGM, COMPOUNDCURVEM,
071                         CURVEPOLYGONM, MULTICURVEM or MULTISURFACEM'';
072  return ''fail'';
073  END IF;
074
075  IF ( (new_dim >4) or (new_dim <0) ) THEN
076  RAISE EXCEPTION ''invalid dimension'';
077  return ''fail'';
078  END IF;
079
080  IF ( (new_type LIKE ''%M'') and (new_dim!=3) ) THEN
081
082  RAISE EXCEPTION ''TypeM needs 3 dimensions'';
083  return ''fail'';
084  END IF;
085
086  IF ( schema_name != '''' ) THEN
087  schema_ok = ''f'';
088  FOR rec IN SELECT nspname FROM pg_namespace WHERE text(nspname) = schema_name LOOP
089  schema_ok := ''t'';
090  END LOOP;
091
092  if ( schema_ok <> ''t'' ) THEN
093  RAISE NOTICE ''Invalid schema name - using current_schema()'';
094  SELECT current_schema() into real_schema;
095  ELSE
096  real_schema = schema_name;
097  END IF;
098
099  ELSE
100  SELECT current_schema() into real_schema;
101  END IF;
102
103  -- Add geometry column
104
105  EXECUTE ''ALTER TABLE '' ||
106  quote_ident(real_schema) || ''.'' || quote_ident(table_name)
107  || '' ADD COLUMN '' || quote_ident(column_name) ||
108  '' geometry '';
109
```

Komentáře:

105 Prostřednictvím dynamického SQL přidává sloupec do cílové tabulky faktů.

```
110  -- Delete stale record in geometry_column (if any)
111
112  EXECUTE ''DELETE FROM geometry_columns WHERE
113  f_table_catalog = '' || quote_literal('''') ||
114  '' AND f_table_schema = '' ||
115  quote_literal(real_schema) ||
116  '' AND f_table_name = '' || quote_literal(table_name) ||
117  '' AND f_geometry_column = '' || quote_literal(column_name);
118
119  -- Add record in geometry_column
120
121  EXECUTE ''INSERT INTO geometry_columns VALUES ('' ||
122  quote_literal('''') || '','' ||
123  quote_literal(real_schema) || '','' ||
124  quote_literal(table_name) || '','' ||
125  quote_literal(column_name) || '','' ||
126  new_dim || '','' || new_srid || '','' ||
```

```
127  quote_literal(new_type) || ''')'';
128
```

Komentáře:

112 Prostřednictvím dynamického SQL ruší sloupec, pokud byl takový, v tabulce metadat geometry_columns.

121 Prostřednictvím dynamického SQL vkládá metadata o sloupci do tabulky geometry_columns.

```
129  -- Add table checks
130
131  EXECUTE ''ALTER TABLE '' ||
132  quote_ident(real_schema) || ''.'' || quote_ident(table_name)
133  || '' ADD CONSTRAINT ''
134  || quote_ident(''enforce_srid_'' || column_name)
135  || '' CHECK (SRID('' || quote_ident(column_name) ||
136  '') = '' || new_srid || '')'' ;
137
138  EXECUTE ''ALTER TABLE '' ||
139  quote_ident(real_schema) || ''.'' || quote_ident(table_name)
140  || '' ADD CONSTRAINT ''
141  || quote_ident(''enforce_dims_'' || column_name)
142  || '' CHECK (ndims('' || quote_ident(column_name) ||
143  '') = '' || new_dim || '')'' ;
144
145  IF (not(new_type = ''GEOMETRY'')) THEN
146  EXECUTE ''ALTER TABLE '' ||
147  quote_ident(real_schema) || ''.'' || quote_ident(table_name)
148  || '' ADD CONSTRAINT ''
149  || quote_ident(''enforce_geotype_'' || column_name)
150  || '' CHECK (geometrytype('' ||
151  quote_ident(column_name) || '')='' ||
152  quote_literal(new_type) || '' OR ('' ||
153  quote_ident(column_name) || '') is null)'';
154  END IF;
155
156  return
157  real_schema || ''.'' ||
158  table_name || ''.'' || column_name ||
159  '' SRID:'' || new_srid ||
160  '' TYPE:'' || new_type ||
161  '' DIMS:'' || new_dim || chr(10) || '' '';
162 END;
163 '
164 LANGUAGE 'plpgsql' _VOLATILE_STRICT; -- WITH (isstrict);
165
166 -----------------------------------------------------------------------
167 -- ADDGEOMETRYCOLUMN ( <schema>, <table>, <column>, <srid>, <type>, <dim> )
168 -----------------------------------------------------------------------
169 --
170 -- This is a wrapper to the real AddGeometryColumn, for use
171 -- when catalogue is undefined
172 --
173 -----------------------------------------------------------------------
174 CREATEFUNCTION AddGeometryColumn(varchar,varchar,varchar,integer,varchar,integer) RETURNS text AS '
175 DECLARE
176  ret  text;
177 BEGIN
178  SELECT AddGeometryColumn('''',$1,$2,$3,$4,$5,$6) into ret;
179  RETURN ret;
180 END;
181 '
182 LANGUAGE 'plpgsql' _STABLE_STRICT; -- WITH (isstrict);
```

```
183
184 ----------------------------------------------------------------------------
185 -- ADDGEOMETRYCOLUMN ( <table>, <column>, <srid>, <type>, <dim> )
186 ----------------------------------------------------------------------------
187 --
188 -- This is a wrapper to the real AddGeometryColumn, for use
189 -- when catalogue and schema are undefined
190 --
191 ----------------------------------------------------------------------------
192 CREATEFUNCTION AddGeometryColumn(varchar,varchar,integer,varchar,integer) RETURNS text AS '
193 DECLARE
194  ret  text;
195 BEGIN
196  SELECT AddGeometryColumn('''','''',$1,$2,$3,$4,$5) into ret;
197  RETURN ret;
198 END;
199 '
200 LANGUAGE 'plpgsql' _VOLATILE_STRICT; -- WITH (isstrict);
```

Komentáře:

174 Přetížení funkcí (tj. existuje více funkcí stejného jména s různými parametry) se v PostgreSQL (dle standardu ANSI) používá také k náhradě nepodporovaných volitelných parametrů. Funkce definované na řádcích 174 a 192 se používají v případě, že chybí hodnoty parametrů katalog a schéma. V ANSI SQL se nepoužívá termín databáze, ale katalog, který obsahuje jemnější dělení na jednotlivá schémata.

Zdrojový kód ESRI ArcSDE podporovaná podmnožiny SQL/MM funkcí je v souboru sqlmm.sql.

```
01 -- PostGIS equivalent function: ndims(geometry)
02 CREATEFUNCTION ST_CoordDim(geometry)
03     RETURNS smallint
04     AS '@MODULE_FILENAME@', 'LWGEOM_ndims'
05     LANGUAGE 'C' _IMMUTABLE_STRICT; -- WITH (iscachable,isstrict);
06
07 -- PostGIS equivalent function: GeometryType(geometry)
08 CREATEFUNCTION ST_GeometryType(geometry)
09     RETURNS text
10     AS '
11     DECLARE
12         gtype text := geometrytype($1);
13     BEGIN
14         IF (gtype IN (''POINT'', ''POINTM'')) THEN
15             gtype := ''Point'';
16         ELSIF (gtype IN (''LINESTRING'', ''LINESTRINGM'')) THEN
17             gtype := ''LineString'';
18         ELSIF (gtype IN (''POLYGON'', ''POLYGONM'')) THEN
19             gtype := ''Polygon'';
20         ELSIF (gtype IN (''MULTIPOINT'', ''MULTIPOINTM'')) THEN
21             gtype := ''MultiPoint'';
22         ELSIF (gtype IN (''MULTILINESTRING'', ''MULTILINESTRINGM'')) THEN
23             gtype := ''MultiLineString'';
24         ELSIF (gtype IN (''MULTIPOLYGON'', ''MULTIPOLYGONM'')) THEN
25             gtype := ''MultiPolygon'';
26         ELSE
27             gtype := ''Geometry'';
28         END IF;
29         RETURN ''ST_'' || gtype;
30     END
31         '
32         LANGUAGE 'plpgsql' _IMMUTABLE_STRICT; -- WITH (isstrict,iscachable);
```

Komentáře:

02 Vytvoření synonyma pro PostGIS funkci.

08 Zapouzdření PostGIS funkce kódem v plpgsql. V tomto případě se stírá rozdíl mezi typy MULTIPOINT a MULTIPOINTM (analogicky u dalších typů).

Řešení výkonných funkcí v PostGISu

Kromě vlastní implementace datových typů PostGIS obsahuje implementaci pomocných funkcí nad prostorovými daty. Následující příklady jsou funkce z lwgeom_functions_basic.c, které mohou sloužit jako vzor pro vytváření vlastních funkcí.

Funkce LWGEOM_makepoint se používá pro vytvoření 2D bodu na základě zadaných souřadnic.

```
01 PG_FUNCTION_INFO_V1(LWGEOM_makepoint);
02 Datum LWGEOM_makepoint(PG_FUNCTION_ARGS)
03 {
04         double x,y,z,m;
05         LWPOINT *point;
06         PG_LWGEOM *result;
07
08         x = PG_GETARG_FLOAT8(0);
09         y = PG_GETARG_FLOAT8(1);
10
11         if ( PG_NARGS() == 2 ) point = make_lwpoint2d(-1, x, y);
12         else if ( PG_NARGS() == 3 ) {
13                 z = PG_GETARG_FLOAT8(2);
14                 point = make_lwpoint3dz(-1, x, y, z);
15         }
16         else if ( PG_NARGS() == 4 ) {
17                 z = PG_GETARG_FLOAT8(2);
18                 m = PG_GETARG_FLOAT8(3);
19                 point = make_lwpoint4d(-1, x, y, z, m);
20         }
21         else {
22                 elog(ERROR, "LWGEOM_makepoint: unsupported number of args: %d",
23                         PG_NARGS());
24                 PG_RETURN_NULL();
25         }
26
27         result = pglwgeom_serialize((LWGEOM *)point);
28
29         PG_RETURN_POINTER(result);
30 }
```

Komentáře:

01, 02 Standardní záhlaví funkce pro v1 volající konvenci

08, 09 Získání prvních dvou argumentů typu float8

11 Pokud počet argumentů funkce je roven dvěma, volá se externí funkce make_lwpoint2d, jinak se zjišťuje počet argumentů a podle něj se volá odpovídající verze externí funkce.

22 Funkce elog se používá pro vyvolání výjimky, pokud je level ERROR. V případě, že level je NOTICE, zobrazí ladící hlášení.

24 Kód za elog(ERROR,...) se již neprovádí. V tomto případě PG_RETURN_NULL() slouží k utišení překladače ohledně zobrazení varování.

27 Serializace objektu do typu PG_LWGEOM.

29 Výstupem z funkce je ukazatel na serializovanou hodnotu objektu, provede se konverze na typ Datum.

SQL registrace této funkce vypadá následovně:

```
01 CREATEFUNCTION makePoint(float8, float8)
02         RETURNS geometry
03         AS '@MODULE_FILENAME@', 'LWGEOM_makepoint'
04         LANGUAGE 'C' _IMMUTABLE_STRICT; -- WITH (iscachable,isstrict);
05
06 CREATEFUNCTION makePoint(float8, float8, float8)
07         RETURNS geometry
08         AS '@MODULE_FILENAME@', 'LWGEOM_makepoint'
09         LANGUAGE 'C' _IMMUTABLE_STRICT; -- WITH (iscachable,isstrict);
10
11 CREATEFUNCTION makePoint(float8, float8, float8, float8)
12         RETURNS geometry
13         AS '@MODULE_FILENAME@', 'LWGEOM_makepoint'
14         LANGUAGE 'C' _IMMUTABLE_STRICT; -- WITH (iscachable,isstrict);
```

Komentáře:

01, 06, 11 Tato implementace je ukázkou přetížené funkce (s různým počtem parametrů), kdy všechny varianty této přetížené funkce sdílí jednu funkci implementovanou v jazyce C.

Funkce LWGEOM_inside_circle_point slouží k určení, zda-li je bod uvnitř nebo vně kruhu.

```
01 PG_FUNCTION_INFO_V1(LWGEOM_inside_circle_point);
02 Datum LWGEOM_inside_circle_point(PG_FUNCTION_ARGS)
03 {
04         PG_LWGEOM *geom;
05         double cx = PG_GETARG_FLOAT8(1);
06         double cy = PG_GETARG_FLOAT8(2);
07         double rr = PG_GETARG_FLOAT8(3);
08         LWPOINT *point;
09         POINT2D pt;
10
11         geom = (PG_LWGEOM *)PG_DETOAST_DATUM(PG_GETARG_DATUM(0));
12         point = lwpoint_deserialize(SERIALIZED_FORM(geom));
13         if ( point == NULL ) {
14                 PG_FREE_IF_COPY(geom, 0);
15                 PG_RETURN_NULL(); /* not a point */
16         }
17
18         getPoint2d_p(point->point, 0, &pt);
19
20         PG_FREE_IF_COPY(geom, 0);
21
22         PG_RETURN_BOOL(lwgeom_pt_inside_circle(&pt, cx, cy, rr));
23 }
```

Komentáře:

11 Z TOAST hodnoty musíme získat serializovanou hodnotu typu PG_LWGEOM. TOAST je pro uživatele databáze (nikoliv pro vývojáře) transparentní mechanismus zajišťující kompresi a uložení serializovaných řetězců delších než 2KB. PostgreSQL interně používá datové stránky o 8KB a žádná do databáze uložená hodnota (vyjma tzv. BLOBu) nemůže tuto velikost přesáhnout. Toto omezení se obchází právě metodou nazvanou TOAST, kdy se delší hodnoty dělí a ukládají do speciální tabulky do více řádků po maximálně 2KB).

12 Deserializace typu point.

14 Bezpečné uvolnění paměti (celá řada typů se přenáší hodnotou, a tudíž je nelze chápat jako ukazatele a nelze dealokovat paměť na kterou by se odkazovaly).

18 Konverze typu LWPOINT na typ POINT2D.

22 Vrácení návratové hodnoty jako výsledku volání funkce lwgeom_pt_inside_circle.

Definice funkce lwgeom_pt_inside_circle (measures.c):

```
01 lwgeom_pt_inside_circle(POINT2D *p, double cx, double cy, double rad)
02 {
03         POINT2D center;
04
05         center.x = cx;
06         center.y = cy;
07
08         if ( distance2d_pt_pt(p, &center) < rad ) return 1;
09         else return 0;
10 }
```

Funkce LWGEOM_inside_circle_point je registrována SQL příkazem:

```
01 CREATEFUNCTION point_inside_circle(geometry,float8,float8,float8)
02         RETURNS bool
03         AS '@MODULE_FILENAME@', 'LWGEOM_inside_circle_point'
04         LANGUAGE 'C' _IMMUTABLE_STRICT; -- WITH (isstrict);
```

Podpora indexu typu GiST v PostGISu

Indexy typu R-tree jsou specifické právě pro prostorová vícedimenzionální data (a pro ně byly navrženy). Aktuální verze PostgreSQL nabízí již další generaci této třídy databázových indexů a to tzv. GiST (Generalized Search Tree) indexy. Jejich princip je stejný, širší je ale jejich uplatnění. GiST indexy se v PostgreSQL používají pro fulltext, indexování obsahu polí, vlastní podporu hierarchických dat a také samozřejmě pro geometrické typy.

Jak již bylo zmíněno, R-tree index předpokládá, že indexovaná data budou mít minimálně dvě dimenze. Index má stromovou strukturu, a každý nekoncový uzel obsahuje jednak odkaz na své potomky a hlavně geometrii nejmenšího pravoúhlého n-rozměrného tělese obsahujícího všechny potomky.

GiST je aplikační rozhraní, které umožňuje implementaci libovolného typu indexu: B-tree, R-tree. Výhodou GiST indexů je možnost vytvoření doménově specifických indexů vázaných na vlastní typy vývojářům znalým doménové oblasti bez toho, aby se nutně staly databázovými specialisty (Rozhodně ale implementace GiST indexu nepatří mezi triviální programování). Ukázkovým příkladem je použití GiST indexu v PostGISu. Kritériem, které se použije pro rozhodování, zda-li použít B-tree index nebo GiST index jsou operace, které chceme urychlit indexem. Pokud nám postačuje množina binárních operátorů $<, =, >$, pak je na místě uvažovat o B-tree indexu. V opačném případě nezbývá než použít GiST index, který je obecnější než B-tree index.

Prostorové indexy se dají použít i pro klasická data. Jednodimenzionální data se ale musí předtím převést na vícedimenzionální. Ukázkovým případem selhání jednodimenzionálního případu je následující příklad. Mějme databázi událostí popsanou časem zahájení (start_time) a časem ukončení (end_time). Pokud budeme chtít vypsat události, které probíhaly v určitém čase napíšeme dotaz s podmínkou

```
01 WHERE star_time < t AND end_time > (t + n)
```

Indexace sloupců start_time a end_time zcela jistě pomůže, nicméně v tomto případě mají indexy malou selektivitu (v průměru oba vrací polovinu řádků z tabulky). start_time a end_time jsou dvě jednodimenzionální řady, takže se celkem přirozeně nabízí chápat je jako jednu dvou dimenzionální řadu a předchozí podmínku transformovat do tvaru postaveného nad geometrickými operátory.

```
01 WHERE box(point(start_time - t, start_time - t),
02                      point(end_time - (t + n), end_time - (t + n)) @
03 box(point(start_time, start_time),
04      point(end_time, end_time))
```

Komentáře:

01 Zápis není validní. Z důvodu čitelnosti neobsahuje nezbytnou konverzi z typu Date na celá čísla.

02 Operátor @ má význam kompletně obsažen.

Popis a použití GiST indexu

GiST index je vyvážený strom obsahující vždy dvojice klíč (predikát), ukazatel na data na hraničních uzlech stromu (listech) a dvojice tvrzení, ukazatel na potomky ve vnitřních uzlech stromu. Dvojice tvrzení, ukazatel se označuje jako záznam indexu. Každý uzel může obsahovat více záznamů indexu. Tvrzení (predicates) je vždy platné pro všechny klíče dostupné z daného uzlu. To je koncept, který se objevuje ve všech na stromech založených indexech. U již zmíněného R-tree indexu je tvrzením ohraničující obdélník obsahující všechny body (R-tree je index navržený pro prostorová data), které jsou dostupné z vnitřního uzlu.

Každý GiST index je definován následujícími operacemi:

Operace nad klíči – tyto metody jsou specifické pro danou třídu objektů a defacto určují konfiguraci GiST indexu (Key Methods): Consistent, Union, Compress, Decompress, Penalty, PickSplit

Operace nad stromem – obecné operace, které volají operace nad klíči (Tree Methods): Search (Consistent), Insert (Penalty, PickSplit), Delete (Consistent).

Operace nad klíči (v závorce smysl operace v případě prostorových dat):

Základní datovou strukturou používanou ve funkcích implementujících GiST index je GISTENTRY:

```
01 /*
02  * An entry on a GiST node.  Contains the key, as well as its own
03  * location (rel,page,offset) which can supply the matching pointer.
04  * leafkey is a flag to tell us if the entry is in a leaf node.
05  */
06 typedef struct GISTENTRY
07 {
08     Datum       key;
09     Relation    rel;
10     Page        page;
11     OffsetNumber offset;
12     bool        leafkey;
13 } GISTENTRY;
```

Komentáře:

consistent	Pokud v záznamu indexu je zaručeno, že tvrzení nevyhovuje dotazu s danou hodnotou, pak vrací logickou hodnotu nepravda. Jinak vrací logickou hodnotu pravda. Pokud operace nesprávně vrátí log. hodnotu pravda, pak tato chyba nemá vliv na výsledek, pouze ovlivní efektivitu algoritmu (true, pokud dochází k překryvu, jinak false).
union	Pro danou množinu záznamů indexu vrací takové tvrzení, které je platné pro všechny záznamy v množině.
compress	Převádí data do vhodného formátu pro fyzické uložení na stránce indexu (V případě prostorových dat se určí hraniční obdélník).
decompress	Opak metody compress. Převádí binární reprezentaci indexu tak, aby s ní mohlo API manipulovat (načte se hraniční obdélník).
penalty	Vrací hodnotu ve významu ceny za vložení nové položky do konkrétní části stromu. Položka je vložena do té části stromu, kde je tato hodnota (penalty) nejnižší (Zjišťuje se, o kolik by se zvětšila plocha hraničního obdélníku).
picksplit	Ve chvíli, kdy je nutné rozdělit stránku indexu, tato funkce určuje, které položky zůstanou na původní stránce, a které se přesunou na novou stranu indexu.
same	Vrací logickou hodnotu pravda pokud jsou dvě položky identické.

09 Identifikátor tabulky

10 Identifikace datové stránky

11 Pozice na datové stránce

a

GistEntryVector

```
01 /*
02  * Vector of GISTENTRY structs; user-defined methods union and pick
03  * split takes it as one of their arguments
04  */
05 typedef struct
06 {
07     int32       n;              /* number of elements */
08     GISTENTRY   vector[1];
09 } GistEntryVector;
10
11 #define GEVHDRSZ    (offsetof(GistEntryVector, vector))
```

Následující příklad je ukázkou metody union, která vrací nejmenší možný obdélník pro všechny zadané body:

```
01 PG_FUNCTION_INFO_V1(LWGEOM_gist_union);
02 Datum LWGEOM_gist_union(PG_FUNCTION_ARGS)
03 {
04         GistEntryVector *entryvec = (GistEntryVector *) PG_GETARG_POINTER(0);
05     int                *sizep = (int *) PG_GETARG_POINTER(1);
06         int                        numranges,
07                                    i;
08         BOX2DFLOAT4 *cur,
09                             *pageunion;
10
11         numranges = entryvec->n;
12         cur = (BOX2DFLOAT4 *) DatumGetPointer(entryvec->vector[0].key);
13
14         pageunion = (BOX2DFLOAT4 *) palloc(sizeof(BOX2DFLOAT4));
```

```
15          memcpy((void *) pageunion, (void *) cur, sizeof(BOX2DFLOAT4));
16
17          for (i = 1; i < numranges; i++)
18          {
19                  cur = (BOX2DFLOAT4*) DatumGetPointer(entryvec->vector[i].key);
20
21                  if (pageunion->xmax < cur->xmax)
22                          pageunion->xmax = cur->xmax;
23                  if (pageunion->xmin > cur->xmin)
24                          pageunion->xmin = cur->xmin;
25                  if (pageunion->ymax < cur->ymax)
26                          pageunion->ymax = cur->ymax;
27                  if (pageunion->ymin > cur->ymin)
28                          pageunion->ymin = cur->ymin;
29          }
30
31          *sizep = sizeof(BOX2DFLOAT4);
32
33          PG_RETURN_POINTER(pageunion);
34 }
```

Komentáře:

04 První argument obsahuje ukazatel na GistEntryVector

05 Druhý argument obsahuje ukazatel na velikost vrácené datové struktury

14, 15 Vytvoření prostoru pro výstupní strukturu pageunion a její naplnění prvním prvkem vektoru.

12, 19 Naplnění struktury cur (iterace po prvcích GiST vektoru, který obsahuje prvky typu Datum (v tomto případě ukazatele na typ BOX2DFLOAT4)

21-28 Hledání nejmenšího možného obdélníka obsahujícího všechny zadané body

33 Předání ukazatele na výstupní strukturu

Každá funkce podporující GiST index se musí nejdříve zaregistrovat jako PostgreSQL funkce a potom všechny relevantní funkce ještě jednou objeví v registraci GiST indexu:

```
CREATEFUNCTION LWGEOM_gist_union(bytea, OPAQUE_TYPE)
        RETURNS OPAQUE_TYPE
        AS '@MODULE_FILENAME@' ,'LWGEOM_gist_union'
        LANGUAGE 'C';

01 CREATE OPERATOR CLASS gist_geometry_ops
02          DEFAULT FOR TYPE geometry USING gist AS
03          OPERATOR        1       <<      RECHECK,
04          OPERATOR        2       &<      RECHECK,
05          OPERATOR        3       &&      RECHECK,
06          OPERATOR        4       &>      RECHECK,
07          OPERATOR        5       >>      RECHECK,
08          OPERATOR        6       ~=      RECHECK,
09          OPERATOR        7       ~       RECHECK,
10          OPERATOR        8       @       RECHECK,
11          OPERATOR        9       &<|     RECHECK,
12          OPERATOR        10      <<|     RECHECK,
13          OPERATOR        11      |>>     RECHECK,
14          OPERATOR        12      |&>     RECHECK,
15          FUNCTION        1       LWGEOM_gist_consistent (internal, geometry, int4),
16          FUNCTION        2       LWGEOM_gist_union (bytea, internal),
17          FUNCTION        3       LWGEOM_gist_compress (internal),
18          FUNCTION        4       LWGEOM_gist_decompress (internal),
19          FUNCTION        5       LWGEOM_gist_penalty (internal, internal, internal),
20          FUNCTION        6       LWGEOM_gist_picksplit (internal, internal),
```

```
21        FUNCTION        7        LWGEOM_gist_same (box2d, box2d, internal); \
01 CREATE OPERATOR CLASS gist_geometry_ops
```

Závěr

Cílem této práce bylo připravit podklady umožňující snažší orientaci v implementaci standardu OpenGIS v prostředí o.s. databázového systémy PostgreSQL – PostGIS. Nejdůležitější komponenty systému PostGIS byly popsány, zbytek nikoliv. Což ani nebylo cílem práce. Ačkoliv není pravděpodobné, že by někdo mohl navrhnout vlastní rozšíření PostGISu bez předchozích znalostí PostgreSQL, C a vlastních GIS aplikací, doufám, že díky této práci mohou vznikat další rozšíření postavené nad tímto, poměrně velice úspěšným produktem.

Literatura

- Správa časoprostorových dat v prostředí PostgreSQL/PostGIS, Antonín ORLÍK
- http://postgis.refractions.net/docs/
- Návrh a realizace UDF v C pro PostgreSQL[1]
- Access Methods for Next-Generation Database Systems[2]
- Spatial Data Management[3]

[1] http://www.pgsql.cz/index.php/N%C3%A1vrh_a_realizace_UDF_v_c_pro_PostgreSQL#N \
.C3.A1vrh_vlastn.C3.ADch_datov.C3.BDch_typ.C5.AF

[2] http://citeseer.ist.psu.edu/rd/0%2C448594%2C1%2C0.25%2CDownload/http://citeseer.ist \
.psu.edu/cache/papers/cs/22615/http:zSzzSzs2k-ftp.cs.berkeley.edu:8000zSz%7Emar \
celzSzdisszSzdiss.pdf/access-methods-for-next.pdf

[3] http://www.mapbender.org/presentations/Spatial_Data_Management_Arnulf_Christl/Spati \
al_Data_Management_Arnulf_Christl.pdf