VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Nadstandardní vlastnosti PostgreSQL a jejich praktické využití

Non-standard Abilities of PostgreSQL

2015 Adam Folwarczny

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Zadání bakalářské práce

Student: **Adam Folwarczny**

Studijní program: B2647 Informační a komunikační technologie

Studijní obor: 2612R025 Informatika a výpočetní technika

Téma: Nadstandardní vlastnosti PostgreSQL a jejich praktické využití
Non-standard Abilities of PostgreSQL

Zásady pro vypracování:

Cílem je najít vlastnosti, které má aktuální PostgreSQL nad rámec SQL (resp. konkurečních databází) a navrhnout v jakých situacích se dají dobře uplatnit.

1. Prozkoumejte nadstandardní (dle jejich označení v oficiální dokumentaci) vlastnosti a schopnosti aktuální stabilní verze PostgreSQL .
2. Vyberte a zdokumentujte ty, u kterých shledáte mimořádný praktický přínos obtížně nahraditelný standardními vlastnostmi SQL. Popište u nich typ problému, který dobře řeší a jak. Tato část by měla představovat 50% - 75% textu práce.
3. Prozkoumejte postupy pro uživatelské rozšíření PostgreSQL, vyberte jednu oblast rozšíření, definujte vhodný ukázkový praktický problém a naimplementujte rozšíření, které jej řeší. Popište podrobněji, co bylo potřeba naimplementovat, v tomto vybraném případě.
4. Srovnejte stručně s možnostmi, resp. postupy pro rozšíření dvou dalších populárních DBMS.

Seznam doporučené odborné literatury:

[1] Momjian, Bruce: PostgreSQL : praktický průvodce. Computer Press. 2003.
[2] Molinaro, Anthony: SQL Cookbook (SQL: kuchařka programátora). O'Reilly Media. 2005.
[3] PostgreSQL 9.3.5 Documentation: Extending SQL.
<http://www.postgresql.org/docs/9.3/static/extend.html>
[4] PostgreSQL 9.3.5 Documentation: Packaging Related Objects into an Extension.
<http://www.postgresql.org/docs/9.3/static/extend-extensions.html>

Formální náležitosti a rozsah bakalářské práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí bakalářské práce: **Ing. Jakub Macek**

Datum zadání: 01.09.2014

Datum odevzdání: 07.05.2015

doc. Dr. Ing. Eduard Sojka
vedoucí katedry

prof. RNDr. Václav Snášel, CSc.
děkan fakulty

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě 7. května 2015

.....

Rád bych na tomto místě poděkoval svému vedoucímu bakalářské práce Ing. Jakubovi Mackovi za odborné vedení a rady při její tvorbě. Dále bych rád poděkoval i své rodině za trpělivost a morální podporu.

Abstrakt

Cílem bakalářské práce je prozkoumat nadstandardní vlastnosti databázového systému PostgreSQL nad rámec standardu SQL, respektive konkurečních databázových systémů a následně pro ně najít situace, ve kterých se dají vhodně uplatnit. V práci jsou nejprve postupně vyčteny možnosti uživatelské rozšířitelnosti PostgreSQL databáze a následně vybrány, zdokumentovány a zdůvodněny právě ty, které jsou jen obtížně nahraditelné vlastnostmi standardu SQL. Další část práce se soustředí na postup vytvoření jednoho praktického uživatelského rozšíření datábaze. Tato část podrobně popisuje, co všechno bylo při implementaci tohoto rozšíření nutno provést. Závěrečná část práce se pak stručně zabývá srovnáním dvou konkurečních SŘBD z hlediska rozšířitelnosti.

Klíčová slova: PostgreSQL, nadstandardní vlastnosti, uživatelská rozšířitelnost, SŘBD, bakalářská práce

Abstract

The goal of this bachelor thesis is to explore non-standard abilities of PostgreSQL database system beyond the SQL standard respectively beyond other competitive database systems and to find situations of practical applications for these non-standard abilities subsenquently. As first there are listed user-defined extensibility options of the PostgreSQL database and then there are picked those that are difficult to replace with abilities of SQL standard. Next part of the thesis is focused on process of creating one practical user-defined extension of the database system. This part in detail describes what was needed for the implementation of that user-defined extension. Final part then briefly compares PostgreSQL database with two other database managements systems from the point of extensibility.

Keywords: PostgreSQL, non-standard abilities, user-defined extensibility, DBMS, thesis

Seznam použitých zkratek a symbolů

SQL	–	Structured English Query Language
pgSQL	–	PostgreSQL
DBMS	–	Database management system
SŘBD	–	Systém řízení báze dat
PL	–	Procedural Language
Tcl	–	Tool Command Language
sh	–	Unix shell
XML	–	Extensible Markup Language
XSD	–	XML Schema Definition
VS	–	Microsoft Visual Studio
CLR	–	Common Language Runtime
T-SQL	–	Transact-SQL
OOP	–	Objektově orientované programování
UDF	–	User-defined functions
API	–	Application Programming Interface

Obsah

Seznam výpisů zdrojového kódu

1 Úvod

Hlavní naplní této bakalářské práce je především prozkoumání, zdokumentování a odůvodnění jedinečnosti některých nadstandardních vlastností databázového systému PostgreSQL. Bakalářská práce obsahuje výčet možností rozšíření databáze a to především v podobě různých typů funkcí, které PostgreSQL nabízí. Celá práce se týká PostgreSQL databáze ve verzi 9.4.1.

Je známo, že původní standard SQL neumožňuje některé operace jako jsou cykly, ukládání do proměných, vytváření funkcí či procedur, práci se soubory a některé další. Avšak konkrétní databázové systémy, jako jsou například Oracle, MySQL, SQL Server, PostgreSQL a mnoho dalších, již dnes nabízí rozšíření nad tento standard a umožňují nám tak pracovat s daty v tabulkách databází více jako v nějakém starším jednoduchém programovacím jazyku (C, Pascal, ...). Je tedy možné vytvářet funkce, díky kterým můžeme manipulovat s daty pomocí cyklů, podmínek a tak dále. Směřuje to tedy ke zrychlení přístupu a manipulace s daty v databázích. Výhoda spočívá především ve schopnosti psát části serverové logiky aplikací přímo na straně databázového serveru. To vede k rychlejším operacím, lepší přenositelnosti aplikací atd. Nejčastěji je tato možnost rozšíření zajištěna prostřednictvím nejrůznějších procedurálních jazyků, ať už se jedná například o PL/SQL (Oracle Database), T-SQL (Microsoft SQL Server) či právě o jeden z mnoha procedurálních jazyků, které nabízí pgSQL. Některé SŘBD však nabízejí možnost psát serverovou logiku i pomocí některých nízkoúrovňových programovacích jazyků s využitím specifických funkcí, knihoven a datových typů určeným k zajištění kompatibility s jádrem databázového systému. U PostgreSQL je tak možné psát uživatelské funkce a rozšíření přímo v jazyce C. Dalším open-source SŘBD, který má obdobnou možnost vytváření UDF v C/C+, je například MySQL. Komerční databázový systém SQL Server firmy Microsoft pak umožňuje vytváření uživatelských funkcí nad celou virtuální platformou CLR architektury .NET. Je zde tedy možné využívat OOP prostřednictvím jazyků Visual Basic či C#.

Kapitola 2 nejprve popisuje SQL standard ve spojistosti s PostgreSQL a vysvětluje na jakém základě a jak vlastně rozšířitelnost v databázovém systému PostgreSQL funguje. Následně kapitola obsahuje popis systému datových typů databáze. Nejdůležitější část pak ale ukazuje výčet všech možných uživatelsky vytvořitelných funkcí, typů, doplňků a také některé jiné nadstadardní vlastnosti databáze PostgreSQL.

Následující kapitola 3 obsahuje výběr vlastností, které jsou jedinečné a obtížně nahraditelné v rámci standardu SQL. Kapitola dokumentuje, proč jsou tyto vlastnosti či uživatelská rozšíření jedinečná a ukazuje jejich použití prostřednictvím jednoduchých příkladů.

V kapitole 4 se dovíme o postupu řešení a implementaci uživatelské C funkce nad databází. Popis zahrňuje, jakým způsobem je třeba využívat C knihovny, které PostgreSQL pro uživatelské C rozšíření nabízí. Dále se zde dočteme, jaké rozšíření jsem implementoval, jakou mělo funkci a co všechno k vytvoření bylo zapotřebí implementovat, nainstalovat či nastavit.

V poslední kapitole 5 jsem srovnával možnosti rozšíření PostgreSQL se dvěma dalšími databazovými systémy. Jedná se o dva populární open-source DBMS MySQL a SQlite.

2 Nadstandardní vlastnosti PostgreSQL

2.1 PostgreSQL a SQL standard

Následující část stručně popisuje, do jaké míry vyhovuje databáze PostgreSQL aktuálnímu SQL standardu. ISO/IEC 9075:2011 či jednoduše SQL:2011 představuje označení aktuální verze SQL standardu ustanoveného roku 2011. Navazuje a rozšiřuje dřívější verze SQL:2008, SQL:2003, SQL:1999 a SQL-92. Každá verze tak nahrazuje předcházející. Není tedy cílem, aby funkcionalita databáze odpovídala starším verzím standardu. Proto také aktuální verze PostgreSQL směřuje svůj vývoj ke standardu SQL:2011.

2.1.1 Vývoj standardu

Standard SQL-92 definoval tři základní balíčky: vstupní (Entry), pokročilý (Intermediate) a kompletní (Full). Většina databázových systému dodržuje Entry balíček z důvodu velkého objemu a komplexnosti funkcionalit sad Intermediate a Full. Vydání standardu SQL:1999 následně definovalo velké množství jednotlivých vlastností a funkcionalit namísto třech obsahlých úrovní standardu SQL-92. Velká část těchto ustanovených vlastností je označena jako „Core features“ (hlavní vlastnosti/funkcionality). Všechny implementace databázových systémů, které chtějí vyhovět standardu, musí tyto „Core features“ splňovat. Podpora zbývajících vlastností je dobrovolná.

2.1.2 SQL:2011 versus PostgreSQL

Většina vlastností vyžadovaných posledním SQL standardem je v rámci PostgreSQL zabudována. Z 179 standardizovaných Core vlastností, v současné době splňuje PostgreSQL nejméně 160. K tomu také navíc implementuje velké množství volitelných vlastností. Některá použití syntaxí se ovšem mohou mírně lišit. S průběhem času lze v rámci vývoje PostgreSQL očekávat kroky směřující k větší shodě se standardem. Za zmínku také stojí to, že žádná současná verze jakéhokoli databázového systému nesplňuje celou sadu Core vlastností SQL:2011. [4]

2.1.3 Kategorie SQL standardu

Poslední SQL standard (SQL:2011) je rozdělen do devíti základních okruhů. Které z těchto okruhů databáze PostgreSQL řeší, popřípadě jakým způsobem je řeší, uvádí následující seznam.

- **ISO/IEC 9075-1 Framework (SQL/Framework)**
  Implementováno v rámci jádra pgSQL.
- **ISO/IEC 9075-2 Foundation (SQL/Foundation)**
  Implementováno v rámci jádra pgSQL.
- **ISO/IEC 9075-3 Call Level Interface (SQL/CLI)**
  Tato specifikace je řešena prostřednictvím ovladače ODBC.

- **ISO/IEC 9075-4 Persistent Stored Modules (SQL/PSM)**
  V současné době pgSQL tuto specifikaci neimplementuje.
- **ISO/IEC 9075-9 Management of External Data (SQL/MED)**
  Implementováno v rámci jádra pgSQL.
- **ISO/IEC 9075-10 Object Language Bindings (SQL/OLB)**
  V současné době pgSQL tuto specifikaci neimplementuje.
- **ISO/IEC 9075-11 Information and Definition Schemas (SQL/Schemata)**
  Implementováno v rámci jádra pgSQL.
- **ISO/IEC 9075-13 Routines and Types using the Java Language (SQL/JRT)**
  Tato specifikace je řešena prostřednictvím pluginu PL/Java.
- **ISO/IEC 9075-14 XML-related specifications (SQL/XML)**
  Implementováno v rámci jádra pgSQL.

2.2 Jak funguje rozšířitelnost PostgreSQL?

Databázový systém PostgreSQL je rozšířitelný, jelikož je jeho provoz řízen katalogy. Ve standardních relačních databázových systémech jsou uchovány systémové informace o databázích, tabulkách, sloupích atd. v takzvaných systémových katalozích (některé systémy mají tyto informace pojmenovány jako datové slovníky). Tyto katalogy se jeví uživateli jako standardní tabulky, ale jak již bylo zmíněno, databázové systémy je používají k interním informacím o fungování či ke konfiguraci databáze.

Zásadní rozdíl mezi PostgreSQL a jinými standardními relačními databázovými systémy je však v tom, že PostgreSQL obsahuje ve svých katalozích mnohem víc informací. Katalogy tak nevlastní pouze informace o svých tabulkách, sloupcích, ale rovněž informace o datových typech, funkcích, přístupových metodách a tak dále. Tyto tabulky (katalogy) mohou být navíc upraveny uživatelem. A jelikož PostgreSQL pracuje na základě těchto katalogových informací, znamená to, že PostgreSQL je uživatelsky rozšířitelný. Pro srovnání lze tradiční databázové systémy rozšířit pouze změnou pevných systémových procedur ve zdrojovém kódu či vložením externích modulů, jenž jsou speciálně napsané přímo tvůrcem daného SŘBD. [5]

Server PostgreSQL má navíc možnost zahrnutí vlastního kódu uživatele pomocí dynamického načítání. To znamená, že uživatel může určit objektový soubor kódu (například knihovnu), který implementuje nový datový typ či funkci, a PostgreSQL následně v případě potřeby provede načtení. Kód napsaný v SQL je tak mnohem jednodušší k zakomponování do serveru. Tato možnost přidání kódu „za běhu“ zajišťuje databázi jedinečnou vhodnost pro rychlé prototypování nových aplikací a uložných struktur.

2.3 Systém typů PostgreSQL

2.3.1 Základní typy (Base Types)

Základní typy jsou ty (jako například int4), které jsou implementovany pod vrstvou jazyka SQL. Většinou se jedná o implementaci nízkoúrovňového jazyka jako je kupříkladu jazyk C. Tyto typy mají velmi často shodu v podobě abstraktního datového typu.

PostgreSQL dokáže s takovými typy pracovat pouze ve funkcích vytvořených uživatelem a rozumí chování těchto typů jen do takové míry, jakou popíše uživatel. Základní typy lze pak ještě dělit na typy skalární (scalar types) a typy polí (array types). Pro každý sklarání typ je vytvořen příslušný typ pole, který může obsahovat variabilně velká pole daného skalarního typu.

2.3.2 Kompozitní typy (Composite Types)

Kompozitní či řádkové typy jsou vytvořeny vždy, když uživatel vytvoří tabulku. Je také možné vytvořit vlastní kompozitní typ pomocí SQL příkazu CREATE TYPE, který nebude příslušet žádné tabulce. Kompozitní typ je ve své podstatě list typů s příslušnými identifikátory.

Hodnotou kompozitního typu je řádek či záznam hodnot sloupců. Uživatel má možnost přistupovat k hodnotám těchto „kompozitů“ pomocí SQL dotazů.

2.3.3 Domény (Domains)

Doména vychází z určitého základní typu (base type) a často je i zaměnitelná s příslušným základním typem. Nicméně doména může obsahovat omezení. Tato omezení určují podmnožinu výchozího base typu, a ta poté reprezentuje validní hodnoty.

2.3.4 Pseudo-typy (Pseudo-Types)

Pro zvláštní případy PostgreSQL obsahuje pár pseudo-typů. Pseudo-typy nemohou být sloupci v tabulce či atributy kompozitních typů. Mohou však být použity jako argumenty či návratové typy funkcí. Toto zajišťuje mechanismus uvnitř systému typů k identifikaci speciální tříd funkcí.

2.3.5 Polymorfní typy (Polymorphic Types)

Psedo-typy *anyelement*, *anyarray*, *anynonarray*, *anyenum*, a *anyrange* se jako celek nazývají polymofrní typy. Jakákoliv vytvořená funkce využívající jeden z těchto typů je nazývana polymofrní funkcí.

Polymofrní funkce může pracovat s mnoha různými datovými typy. Specifický datový typ či více typů je vyhodnoceno dle datových typů předaných při jednotlivých voláních. Polymofrní argumenty a výsledky jsou navzájem mezi sebou svazány. Vyhodnocení v určitý datový typ proběhne po dokončení parsování příslušného dotazu volajícího polymorfní funkci.

2.4 Uživatelská rozšíření pgSQL

1. **Uživatelsky vytvořené funkce**

   PostgreSQL nám nabízí 4 základní varianty funkcí.

   - Funkce dotazovacího jazyka
     (funkce napsané v SQL)
   - Funkce procedurálního jazyka
     (funkce napsané v jazycích typu PL/pgSQL, PL/Python, ...)
   - Interní funkce
   - Funkce jazyka C

   Každý druh funkce může brát jako argumenty (parametry) základní typy 2.3.1, kompozitní typy 2.3.2 či kombinaci těchto typů. Zároveň mohou funkce vracet základní, kompozitní typ či pole těchto typů. Některé typy funkcí dokážou také přijímat či vracet určité typy pseudo-typů (například polymofrní typy). Způsobů vytvaření a fungování SQL funkcí se využívá i v jiných typech funkcí.

2. **Uživatelsky vytvořené agregáty**
3. **Uživatelsky vytvořené datové typy**
4. **Uživatelsky vytvořené operátory**
5. **Třídy operátorů pro indexy**
6. **Balíčky souvisejících objektů**

2.5 Nadstandardní vlastnosti PostgreSQL

PostgreSQL kromě těchto uživatelských rozšíření 2.4 nabízí spoustu dalších vlastností, které vyčnívají z SQL standardu. Výčet těch nejzajímavějších vlastností je uveden níže.

- **Pokročilé fulltextové vyhledávání**
- **Různé typy indexů** (B-strom, Hash, GiST, SP-GiST, GIN)
- **Částečné (partial) indexy**
- **Indexy nad výrazy** (Indexes on expressions)
- **Pokročilé definice triggerů** (v procedurálním jazyku, v jazyce C)
- **Triggery událostí**

3 Obtížně nahraditelné vlastnosti PostgreSQL

3.1 Funkce dotazovacího jazyka

Funkce dotazovacího jazyka, což jsou funkce napsané v jazyku SQL, slouží k vykonání libovolného seznamu standardních SQL příkazů. V této části si navíc ukážeme jak lze vytvořit jednoduché SQL funkce a na co si dát při jejich implementaci pozor.

3.1.1 Návrat (return) SQL funkcí

Návratem dotazovacích funkcí je výsledek posledního SQL příkazu v seznamu. Není-li určeno jinak, vrací se vždy první řádek výsledku posledního příkazu. Je také důležité myslet na to, že „první řádek“ víceřádkového výsledku není možné přesně definovat, pokud se nepoužije příkaz ORDER BY. Jestliže poslední SQL příkaz v seznamu funkce nevrací žádné záznamy, funkce vrací *null* hodnotu.

Pokud chceme zajistit to, aby funkce vracela SET (čili více řádků), musíme jako návratový typ určit SETOF (nějaký typ). Jako návratový typ lze také použít ekvivalentní sloupec tabulky TABLE(sloupec). Je-li tak učiněno, funkce vrátí všechny řádky posledního příkazu funkce.

3.1.2 Struktura SQL funkce

Tělo funkce tvoří seznam SQL příkazů, které musí být odděleny středníkem. Středník za posledním příkazem je dobrovolný. V případě, že není nastaven návratový typ RETURN jako *void* (funkce nevrací žádnou hodnotu), musí být posledním příkazem ve funkci příkaz SELECT nebo INSERT, DELETE či UPDATE. V případě posledních tří však musí příkaz obsahovat návratovou RETURN klauzuli.

Jak už bylo řečeno, jakákoliv kolekce SQL příkazů může společně tvořit funkci. Kromě SELECT dotazů, může kolekce obsahovat INSERT, DELETE, UPDATE a samozřejmě i další SQL příkazy. Co ve funkci nelze použít, jsou příkazy určené k řízení transakcí, jsou jimi COMMIT, SAVEPOINT či některé další (například čistící nástroj VACCUM). Nicméně konečným příkazem musí být příkaz SELECT nebo příkaz mající návratovou klauzuli, která navrací libovolný typ, jenž bude určený jako navrátový typ funkce. Za předpokladu, že chceme definovat funkci, která nemá žádnou užitečnou návratovou hodnotu, použijeme jako návratový typ klíčové slovo *void*.

Syntaxe příkazu CREATE FUNCTION vyžaduje, aby by bylo tělo funkce napsáno jako jeden řetězec. Často je pro řetězec těla funkce nejvhodnější použít uvození znakem dolaru ($). Pokud se rozhodneme pro řetězec použít jednoduché uvozovky, musíme v těle funkce zdvojit použití jednoduchých uvozovek (') a zpětných lomítek (\).[5]

3.1.2.1 Ukázka jednoduché validní funkce Na zdrojovém kódu 1 můžeme vidět, jak taková jednoduchá SQL funkce může vypadat. Jedná se o funkci, která z existující tabulky *Student* vymaže všechny záznamy odpovídající podmínkám WHERE. Funkci poté

můžeme jednoduše zavolat pomocí příkazu SELECT. Pokud by navíc funkce vracela nějakou hodnotu, příkaz by nám místo hodnoty *null* vypsal řádek výstupu.

```
CREATE FUNCTION deleteBadStudents() RETURNS void AS $$
DELETE FROM Student WHERE points < 51;
DELETE FROM Student WHERE inActive = true;
$$ LANGUAGE SQL;

SELECT deleteBadStudents();
```

Výpis 1: Jednoduchá ukázka validní SQL funkce

3.1.2.2 Použití příkazu nad systémovými katalogy Celá struktura těla SQL funkce je nejprve parsována a až následně spuštěna. Ačkoli SQL funkce mohou obsahovat příkazy upravující systémové katalogy jako je například CREATE TABLE, ALTER TABLE či další, výsledek těchto funkcí se neprojeví za běhu funkce, nýbrž až po ukončení běhu funkce. Ukázka funkce 2 tedy nebude funkční.

```
CREATE FUNCTION createAndInsertStudent() RETURNS INTEGER AS $$
CREATE TABLE Student (int id);
INSERT Into Student (1);
SELECT id FROM Student;
$$ LANGUAGE SQL;
```

Výpis 2: Ukázka nefunkční SQL funkce pracující se systémovými katalogy

Tento postup vytvoření a naplnění tabulky nebude v rámci jedné funkce fungovat, jelikož tabulka nebude při vykonávání druhého příkazu INSERT ještě existovat. Pro řešení podobných situací je doporučeno použít jeden z procedurální jazyků SQL, které PostgreSQL nabízí. [5]

3.1.2.3 Argumenty SQL funkcí Argumenty SQL funkcí mohou mít referenci v těle funkce jak v podobě jmen tak i čísel.

K použití jména, je třeba zadeklarovat argument s tím, že bude mít jméno. Následně lze použít toto jméno jako identifikátor v těle funkce. Je-li jméno argumentu v těle funkce stejné jako některý ze sloupců v aktuálním SQL příkazu, název sloupce bude mít přednost. Abychom toto obešli, můžeme použít jméno argumentu společně se jménem funkce *(nazevFunkce.nazevArgumentu)*. Pokud by ale i tento kvantifikátor selhal v případě, že bude jméno tabulky stejné jako jméno funkce, opět vyhrává kvantifikátor sloupce tabulky. Těmto problémům se můžeme vyhnout zvolením vhodného aliasu pro tabulku.

Možnost jmen pro referenci argumentů funkcí byla přidána do PostgreSQL ve verzi 9.2. Funkce serverů běžících na starších verzích systému tak musí implementovat dolarovou ($) notaci. [5]

3.1.3 SQL funkce pracující s Base Typy

Jednoduchá základní funkce 3, která pracuje s Base Typem 2.3.1, nemá žádné argumenty a jako navratový typ definuje Base Typ INTEGER (více ve zdrojovém kódu). Všimněte si, že uvnitř funkce definujeme alias sloupce pro výsledek funkce *(integerOut)*. Tento alias však nebude dostupný mimo funkci.

```
CREATE FUNCTION returnInteger() RETURNS integer AS $$
SELECT id FROM Student WHERE id = 1 AS integerOut;
$$ LANGUAGE SQL;
```

Výpis 3: SQL funkce vracející Base Type INTEGER

3.1.3.1 Base Type argumenty Co se týče SQL funkcí s argumenty, tak je velmi jednoduché tyto argumenty do funkce přidat. Stačí v závorkách za kvantifikátorem funkce pojmenovat argument a následně určit jeho typ. Jednoduchou SQL funkci s Base Type argumenty můžete vidět na kódu 4. Funkce bere jako argument Base Type INTEGER. Následně jej vrátí vynásobený dvěma.

```
CREATE FUNCTION multiplyBy2(argument integer) RETURNS integer AS $$
SELECT a * 2 AS integerOut;
$$ LANGUAGE SQL;

SELECT multiplyBy2(4);
```

Výpis 4: SQL funkce s Base Type argumentem

3.1.3.2 Praktický příklad využití funkce s Base Typy Více praktický příklad SQL funkce nad Base Typy pak může být následující: 5. Funkce zde bere jako parametry výsledek (body) posledního testu a také studentův identifikátor. Studentovi tedy chceme přičíst tyto body k jeho již získaným bodům, které už jsou v tabulce *Student* ve sloupci *point* typu INTEGER uloženy. Student je identifikován pomocí sloupce *id* typu INTEGER. Funkce aktualizuje záznam příslušného studenta (přičte body za poslední test) a následně sečtené body vrátí.

```
CREATE FUNCTION addScore(testScore integer, id integer) RETURNS integer AS $$
UPDATE Student
SET points = points + score
WHERE id = addScore.id;
SELECT points FROM Student WHERE id = addScore.studentIdentification;
$$ LANGUAGE SQL;

SELECT addScore(24, 4875);
```

Výpis 5: Praktické použití SQL funkce s Base Type

3.1.4 SQL funkce pracující s Composite Typy

Při práci s Composite Typy 2.3.2 musíme dávat pozor nejen na identifikátor argumentů, ale také na to s jakým atributem (políčkem) argumentu chceme pracovat. Předpokládejme například, že máme tabulku *Student* a ta obsahuje 3 sloupce *id, name, points*. Pokud tedy předáme nějaké funkci řádek tabulky, jedná se tak o kompozitní typ a při manipulaci s tímto řádkem je nutné uvnitř funkce vždy specifikovat, s kterým sloupcem (atributem) budeme v jednotlivých příkazech pracovat. Jak lze s takovým typem ve funkcích pracovat, si ukážeme na jednoduchém příkladu 6. Funkce bere jako argument kompozitní typ reprezentující řádek tabulky *Student*. Pro přístup k jednotlivým sloupcům pak je potřeba před identifikátor sloupce přidat identifikátor argumentu funkce, kterým v tomto případě je právě kompizitní typ typu Student.

```
CREATE TABLE Student (
id  integer,
name text,
points  integer
);

INSERT INTO Student VALUES (12, 'Jack␣Johnson', 51);

CREATE FUNCTION studentsWish(Student) RETURNS integer AS $$
SELECT $1.points * 2;
$$ LANGUAGE SQL;

SELECT name, studentsWish(Student.*) FROM Student WHERE Student.name='Jack␣Johnson';
```

Výpis 6: Příklad použití Composite Type v SQL funkci

Všimněte si při volaní funkce použití znaku (*), který nám zaručuje vložení celého řádku (kompozitního typu) jako argument funkce.

Řádek tabulky může být použit jako argument funkce i bez nutnosti použití zmíněného znaku. Tento postup je ale zastaralý. [5] PostgreSQL nám také umožňuje upravit hodnoty řádku, který předáváme jako argument funkci. K tomuto úkonu slouží klíčové slovo ROW. (více v kódu 7)

```
SELECT name, studentsWish(ROW(id, name, points*2)) FROM Student
WHERE Student.name='Jack␣Johnson';
```

Výpis 7: Upravení kompozitního typu pomocí ROW

3.1.4.1 Kompozitní typ jako výstup funkce Samozřejmě není žádný problém, aby SQL funkce kompozitní typ vracela. (kód 8) Je však nutné dodržet pořadí atributů kompozitního typu přesně tak, aby odpovídaly pořadí sloupců příslušné tabulky. Je také nutné převádět konstanty na příslušné typy odpovídající datovým typům sloupců tabulky. Funkce taktéž může vytvářet kompozitní typ pomocí ROW, které bylo zmíněno v kódu 7. Příklad použití ROW pro vytvoření kompozitního typu ve funkci můžete vidět v druhé části kódu 8. Kód dále obsahuje volání funkce a metody přístupů k jednotlivým atributům navráceného kompozitního typu.

```
CREATE FUNCTION returnStudent() RETURNS Student AS $$
SELECT 12 AS id,
 text  'Jack␣Johnson' AS name,
51 AS points;
$$ LANGUAGE SQL;

CREATE OR REPLACE FUNCTION returnStudent() RETURNS Student AS $$
SELECT ROW(12, 'Jack␣Johnson', 51)::Student;
$$ LANGUAGE SQL;

SELECT returnStudent();

SELECT (returnStudent()).name;
SELECT name(returnStudent());
```

Výpis 8: Funkce vracející kompozitní typ

Ekvivalence mezi funkcionální notací a atributovou notací umožňuje použití funkcí s kompozitními typy k emulaci „vypočtených atributů“. [5] Můžeme tak například zavolat funkci *studentsWish(Student)* (viz. kód 6) i způsobem *Student.studentsWish()*. Další použití funkce, která vrací kompozitní typ, pak může být předání výsledku jiné funkci, jenž stejný kompozitní typ přijímá jako argument (*SELECT studentsWish(returnStudent());*).

3.1.5 SQL funkce s výstupními parametry

SQL funkce v PostgreSQL nám nabízí i další způsob výstupních hodnot a to pomocí výstupních parametrů. Výstupní parametry SQL funkce se definují naprosto stejně jako klasické parametry s tím, že se před identifikátor parametru přidá navíc klíčové slovo OUT. Výstupní parametr se při volání funkce nezádává. Pokud neurčíme jméno (identifikátor) OUT parametru, systém vygeneruje své vlastní jméno parametru. U vstupních parametrů můžeme uvést klíčové slovo IN. Není to však nutné, pokud totiž neurčíme jinak, jedná se ve výchozím stavu o parametry vstupní.

V ukázce jednoduché SQL funkce 9, která pouze sčítá dvě čísla, pak můžete vidět, jak se s výstupním parametrem *sum* pracuje.

```
CREATE FUNCTION sumOfTwoNumbers(number1 integer, number2 integer, OUT sum integer)
AS $$
SELECT number1 + number2;
$$ LANGUAGE SQL;

SELECT sumOfTwoNumbers(1,2);

DROP FUNCTION sumOfTwoNumbers (int, int);
DROP FUNCTION sumOfTwoNumbers (number1 int, number2 int, OUT sum);
```

Výpis 9: Ukázka výstupního parametru SQL funkce

Návratovým typem se následně stává kompozitní typ 2.3.2, jehož atributy jsou jednotlivé výstupní parametry. S výstupem takovéto funkce se bude tedy pracovat naprosto stejně, jako jsme si již vysvětlili v části 3.1.4.

Volání DROP FUNCTION také nemusí uvádět typy výstupních parametrů. Není problémem mít více výstupních parametrů, musí se však u nich určit identifikátor.

Parametry mohou být kromě IN a OUT označeny dalšími dvěma klíčovými slovy a to INOUT či VARIADIC. INOUT parametr slouží zároveň jako vstupní tak i výstupní parametr. Jedním parametrem se tak určí vstupní parametr i výstupní datový typ. A v případě, že se vloží před parametr klíčové slovo VARIADIC, jedná se o vstupní parametry. Parametrům VARIADIC se podrobněji věnuje kapitola 3.1.6.

3.1.6 SQL funkce s proměnným počtem parametrů

SQL funkce mohou být definovány i tak, že budou přijímat libovolný počet parametrů s předpokladem, že parametry jsou stejného datového typu. Tyto libovolné parametry budou funkci předány jako pole. Deklarace se provadí pomocí klíčového slova VARIADIC. VARIADIC parametr musí být v deklaraci parametrů funkce vždy jako poslední. Datovým typem musí být pole (například *integer[] v kódu 10*). Funkce bere jako parametr *integer*, který určí pozici pole, jejíž hodnotu funkce vrátí. Druhým parametrem je následně pole typu *integer*. Pokud se podíváme na volání funkce, první *integer* parametr určí index pole, jehož hodnotu následně dostaneme a další čtyři *integer* hodnoty vytvoří pole, se kterým bude funkce pracovat.

```
CREATE FUNCTION returnFieldOfArray(position integer, VARIADIC array integer[])
RETURNS integer AS $$
SELECT $1[0];
$$ LANGUAGE SQL;

SELECT returnFieldOfArray(1, 5, 6, 7, 8);
SELECT returnFieldOfArray(1, VARIADIC ARRAY[5, 6, 7, 8]);
SELECT returnFieldOfArray(1, VARIADIC array := ARRAY[5, 6, 7, 8]);
```

Výpis 10: SQL funkce s variabilním počtem paramterů

Funkci s parametrem VARIADIC lze předat i již předem vytvořené pole (viz. druhé volání v kódu 10). Tato metoda je obzvlášť vhodná, pokud chceme uvnitř funkce předat

pole jako parametr nějaké další funkci, která bere jako parametr pole stejného typu. Výhoda spočívá v tom, že funkce, které toto pole předáme, nemusí brát parametr uvozený klíčovým slovem VARIADIC ale pouze klasické pole. V třetím způsobu volání pak pole i pojmenujeme. Je zásadní uvést klíčové slovo VARIADIC, pokud pole vytváříme při volání funkce.

Jestliže bychom nechtěli předat funkci žádný parametr, nemůžeme slovo VARIADIC vynechat. VARIADIC parametr je totiž v základu pořád jenom parametr jako každý jiný. Pokud bychom vynechali klasický IN parametr, volání funkce bez parametru by také nefungovalo. Nechceme-li funkci žádnou hodnotu předat, musíme při volání předat prázdné pole (takto: *VARIADIC ARRAY[]::integer[]*.)

3.1.7 SQL funkce s výchozími hodnotami argumentů

Funkce mohou být deklarovány s výchozími hodnotami některých nebo všech vstupních parametrů. Výchozí hodnoty jsou parametrům nastaveny vždy, když se vynechají při volání funkce. Je třeba si ale dát pozor na to, že parametry s DEFAULT hodnotou mohou být vynechány jen z konce posloupnosti paramterů funkce. Místo klíčového slova DEFAULT můžeme taktéž použít znak (=). Všechny parametry za parametrem s výchozí hodnotou musí také obsahovat výchozí hodnoty. Ačkoli není toto omezení při použití zápisu s pojmenovanými argumenty takto striktní, je doporučeno deklarovat parametry se správnou posloupností. [5] Jednoduchý příklad i s možnostmi vynechání argumentů při volání si můžete prohlédnout na kódu 11.

```
CREATE FUNCTION sumWithDefaults
(number1 integer, number2 integer DEFAULT 2, number3 integer DEFAULT 3)
RETURNS integer AS $$
SELECT number1 + number2 + number3;
$$ LANGUAGE SQL;

SELECT sumWithDefaults(1, 10, 15);
SELECT sumWithDefaults(1, 10);
SELECT sumWithDefaults(1);
```

Výpis 11: SQL funkce s výchozími hodnotami paramterů

Výsledkem prvního volání bude 26. U druhého volání získáme hodnotu 14, parametru number3 se přiřadí vychozí hodnota 3. A u třetího volání pak získáme hodnotu 6. Je tomu tak proto, že jsme vynechali druhý a třetí argument při volání funkce. Byly jim tedy ve funkci přiřazeny výchozí hodnoty 2 a 3.

Pokud bychom se snažili zavolat funkci takto *sumWithDefault()*, čili bez jakéhokoliv parametru, vyskočí chyba z důvodu, že je nutné určit hodnotu prvního parametru *number1*. Za předpokladu, že by měly všechny tři argumenty výchozí hodnoty, bylo by takovéto volání funkce samozřejmě provedeno bez problémů.

3.1.8 SQL funkce jako tabulkové zdroje

Všechny SQL funkce mohou být použity v klauzuli FROM v rámci dotazu. Tento fakt je ale zejména užitečný pro funkce vracející kompozitní typy 2.3.2. Pokud je návratový typ

funkce nadefinován na base typ 2.3.1, funkce vrací tabulku s jedním sloupcem. Jestliže ale nadefinujeme funkci jako návrat kompozitní typ, funkce navrátí tabulku se sloupcem pro každý atribut daného kompozitního typu. Praktický příklad použití funkce jako tabulkového zdroje můžeme vidět na kódu 12. Se sloupci, pocházejícími z výsledků takovýchto funkcí, můžeme dále manipulovat naprosto stejně jako by to bylo u sloupců standardní tabulky. Výsledkem takovýchto funkcí však bude pouze jeden řádek. Funkcím navrácejícím více záznamů (řádků) se budeme věnovat v následující části 3.1.9.

```
CREATE TABLE Student (id integer, name text, points integer);

INSERT INTO foo VALUES (1, 'Jack Johnson', 88);
INSERT INTO foo VALUES (2, 'James Sawyer', 90);

CREATE FUNCTION returnStudent(id int) RETURNS Student AS $$
SELECT * FROM Student WHERE id = returnStudent.id;
$$ LANGUAGE SQL;

SELECT * FROM returnStudent(1);
```

Výpis 12: SQL funkce jako tabulkový zdroj

3.1.9 SQL funkce navrácející SET (více záznamů)

Je-li návratový typ funkce nastaven na RETURNS SETOF *nějaký typ*, poslední dotaz funkce nenávrátí pouze jeden řádek nýbrž všechny řádky, které danému výsledku dotazu odpovídají. Tuto vlastnost tak lze využít v obdobném případě jako v ukázce 12. Jedinou změnou tak bude návratový typ, který bude nyní nastaven na SETOF *Student*. Pro demonstraci nám poslouží příklad 13. Zde nám funkce vrací všechny záznamy, které odpovídají dotazu uvnitř funkce (a ne jen jeden). Využití SETOF je tedy opět velice vhodné jako tabulkový zdroj dotazů SELECT.

```
CREATE FUNCTION studentsWithMorePointsThan(points int) RETURNS SETOF Student AS $$
SELECT * FROM Student WHERE id > returnStudent.points;
$$ LANGUAGE SQL;

SELECT * FROM studentsWithMorePointsThan(80);
```

Výpis 13: SQL funkce vracející SET

3.1.9.1 Návrat SETu skrze výstupní parametry

Je navíc také možné vracet SET (více záznamů) skrze výstupní (OUT) parametry. Funkce se implementuje téměř stejným způsobem, jako tomu bylo v části věnující se výstupním parametrům 3.1.5. Hlavní a jediný rozdíl spočívá v určení návratového typu jako SETOF *nějaký typ* v případě jednoho výstupního parametru či SETOF *record* v případě dvou a více OUT parametrů.

3.1.9.2 Dodatky k funkcím navrácejícím SET

V aktuální verzi pgSQL 9.4.1 lze volat funkce, které vrací SET, i jen pomocí dotazu SELECT bez nutnosti určení tabulkového zdroje.

Volání funkce 13 pak může vypadat i takto *SELECT studentsWithMorePointsThan(80);*. Je ale důležité myslet na to, že tato metoda volání je zastaralá a může být v některé z budoucích verzi pgSQL odstraněna. [5]

Pokud je posledním dotazem funkce INSERT, UPDATE nebo DELETE s návratovou klauzulí, bude takový dotaz vždy proveden kompletně celý a to i tehdy, není-li funkce deklarovaná tak, aby vracela SET, či voláním funkce nezískáváme všechny navracející se řádky. Veškeré záznamy „navíc", které vytváří návratová klauzule RETURNING, budou „tiše" zahozeny, ale úpravy provedené nad tabulkami budou i přesto provedeny a to před návratem hodnot z funkce. [5]

3.1.10 SQL Funce navracející tabulku

Existuje další způsob deklarace funkce vracející SET 3.1.9. Tím je použití RETURNING klauzule s datovým typem TABLE (*sloupce*). Tento způsob deklarace je ekvivaletní způsobu navrátu SETu skrze výstupní parametry 3.1.9.1. Použití tohoto zápisu je specifikováno v novějších verzích SQL standardu, to znamená, že by měl tento zápis být přenositelnější. [5] Použití OUT a INOUT parametrů není v případě funkce navracející tabulku povoleno. Jak taková jednoduchá funkce vracející tabulku může vypadat, lze pozorovat na příkladu 14.

```
CREATE FUNCTION returnAllStudents(id int)
RETURNS TABLE(id int, name text, points int) AS $$
SELECT * FROM Student;
$$ LANGUAGE SQL;
```

Výpis 14: SQL funkce vracející TABLE

3.1.11 Polymorfní SQL funkce

Polymorfní funkce, jak označení napovídá, pracuje s polymorfními typy 2.3.5. SQL funkce mohou polymorfní typy *anyelement, anyarray, anynonarray, anyenum* a *anyrange* přijímat jako parametry či je vracet. Více o tom, jak vlastně polymorfní typy fungují, se dočtete v příslušné části první kapitoly 2.3.5.

3.1.11.1 Co u polymorfních funkcí lze a co nelze

- Polymorfní typy mohou být výstupními (OUT) parametry.
- Polymorfismus může být využit ve funkcích s VARIADIC parametry.
- Je povoleno funkcím mít polymorfní argumenty a zároveň standardní „pevný" návratový typ.
- Není povolena konverze base typu na polymorfní typ. (Nelze tedy například při návratu převádět Base Type na polymorfní typ.)

- Funkce s polymorfním návratovým typem musí obsahovat alespoň jeden polymorfní argument.
- Při volání polymorfní funkce vracející *anyarray* je třeba prvky textových typů pole převést na příslušný datový typ.

3.1.12 SQL funkce s ražením (collation)

Pokud má SQL funkce jeden nebo více seřaditelných parametrů, je řazení provedeno pro každé volání funkce. Vše závisí na řazení určeném argumentům funkce. Je-li řazení úspěšně rozpoznáno, jsou všechny seřaditelné parametry brány tak, jako by měli řazení nastaveno implicitně. [5]

Vestavěné seřaditelné datové typy jsou *text*, *varchar* a *char*. Uživatelsky vytvořené Base Typy 2.3.1 mohou také být seřaditelné, stejně jako domény 2.3.3 vytvořené nad seřaditelnými typy. [7] Více o tom, jak se přesně s řazením (collaction) v pgSQL pracuje, je možno dohledat v oficiální dokumentaci pgSQL.

3.1.13 Funkce dotazovacího jazyka a SQL standard

SQL standard definuje možnost vytváření SQL funkcí (a procedur) pomocí příkazů CREATE FUNCTION (CREATE PROCEDURE). Funkcionalita je však oproti PostgreSQL velmi omezená a to především z důvodu nemožnosti použití specifických datových typů pgSQL.

3.2 Funkce procedurálního jazyka

PostgreSQL umožňuje psát uživatelské funkce i v jiných jazycích než jsou SQL a C. Tyto „ostatní“ jazyky se nazývají procedurální jazyky (PL). Procedurální jazyky nejsou věstavěnou součástí pgSQL serveru. Jsou nabízeny v podobě přídavných modulů.

3.2.1 Propojení jádra pgSQL s procedurálními jazyky

Z důvodu že nejsou tyto jazyky součástí serveru, jsou zdrojové kódy funkcí psaných v procedurálním jazyku předány speciálnímu handleru, který zná syntaxi a detaily příslušného jazyka. Handler může buď „dělat všechnu práci“, to znamená parsování, analýzu syntaxe, vykonání, atd. sám nebo může sloužit jako prostředník mezi pgSQL a existující implementací programovacího jazyka. Sám handler je pak funkcí jazyka C kompilovanou do sdíleného objektu a je tak volána na požádání jako jakákoliv jiná C funkce. [8]

3.2.2 Nabízené procedurální jazyky a jejich vlastnosti

Ve standardní distribuci pgSQL jsou v současné době k dispozici čtyři hlavní procedurální jazyky. Tyto čtyři jazyky jsou standardně předinstalovany a připraveny k použití. Jedná se však stále o externí moduly. Je tedy možné tyto jazyky administrátorem kdykoliv odebrat. Funkce vytvořené v PL/pgSQL mohou být použity všude tam, kde lze

použít „klasické“ SQL funkce 3.1. Deklarace parametrů a návratových typů je také totožná s SQL funkcemi. Výhoda použití však spočívá v dodatečných a více komplexních strukturách, které toto procedurální rozšíření v rámci vytváření funkcí nabízí.

3.2.2.1 PL/pgSQL Struktura jazyka PL/SQL je tvořena bloky. Proto musí definici těla funkce tvořit blok. Deklarace každého bloku a každého příkazu uvnitř bloku musí být ukončena středníkem. Strukturu bloku ukazuje výpis 15. Jak je možno vidět z kódu, před samotným blokem se nachazí klíčové slovo DECLARE. Toto klíčové slovo uvozuje deklaraci proměnných, se kterými se bude v rámci bloku manipulovat.

Častou chybou bývá vložení středníku ihned za příkaz BEGIN. Jedná se o nesprávný zápis, který vyústí v chybu syntaxe. [9]

```
DECLARE deklarace
BEGIN
prikazy
END;
```

Výpis 15: Blok procedurálního jazyka PL/pgSQL

Všechna klíčová slova jsou Case-Sensitive (rozlišují se velká a malá písmena). Identifikátory jsou pak implicitně převáděny, pokud nejsou uvozeny v dvojitých (") uvozovkách, na malá písmena stejně, jak je tomu v „klasických“ SQL příkazech. Práce s komentáři je v kódu PL/pgSQL totožná jako v SQL. Jak vypadá jednoduchá kompletní deklarace funkce napsané v jazyce PL/pgSQL, lze pozorovat na zdrojovém kódu 16. Vidíme zde deklaraci proměnné i její naplnění a následně také její předání klauzuli RETURN. Jak již bylo zmíněno, deklarace probíhá totožně jako u standardních SQL funkcí s tím rozdílem, že musíme určit jako používaný jazyk *plpgsql*.

```
CREATE FUNCTION doublePoints(points real) RETURNS real AS $$
DECLARE
doubledPoints integer;
BEGIN
doublePoints := points * 2;
RETURN doublePoints;
END;
$$ LANGUAGE plpgsql;
```

Výpis 16: Ukázka deklarace jednoduché PL/pgSQL funkce

PL/pgSQL však nabízí mnohé struktury, výrazy a vlastnosti známé ze standardních programovacích jazyků. Lze tak využít například některé z těchto vlastností:

- Proměnné *(DECLARE popisek text;)*
- Podmínky *(IF THEN, ELSIF, ELSE)*
- Vyjímky *(TRY, CATCH, EXCEPTION, RAISE)*
- Cykly *(WHILE, LOOP, CONTINUE, EXIT WHEN, FOR, FOREACH)*
- Cursory (např.: cyklické procházení výsledků dotazů)

Procedurální jazyk PL/pgSQL je v mnoha aspektech podobný jazyku PL/SQL, který je výchozím procedurálním jazykem v databázích Oracle. Také jedná o blokově struktorovaný, imperativní jazyk a všechny proměnné musí být předem deklarovány. Přiřazování, cykly a podmínky jsou taktéž velice podobné. [9] Při dodržení pokynů definujících hlavní rozdíly mezi těmito dvěma jazyky by tedy přenos kódů mezi Oracle a PostgreSQL databázemi neměl být problém. Více informací o přenositelnosti a rozdílech je možno dohledat v oficiální dokumentaci PostgreSQL.

3.2.2.2 Další „zabudované“ procedurální jazyky

Jak již bylo zmíněno v úvodu části věnující se nabízeným procedurálním jazykům 3.2.2, standardní distribuce pgSQL má zabudované kromě PL/pgSQL další tři hlavní procedurální jazyky. Konkrétně se jedná o tyto tři:

- PL/Tcl
- PL/Python
- PL/Perl

Funkce implementované v těchto jazycích tak mohou využít specifických struktur, výrazů a funkcí, které každý z těchto jazyků nabízí. PostgreSQL tak nabízí možnost psát serverovou logiku přímo nad instancemi databází jazykem, jenž je vývojáři známý, jelikož v něm například implementuje aplikaci, která právě databázi PostgreSQL využívá.

3.2.2.3 „Nezabudované“ procedurální jazyky

Existuje množství dalších procedurálních jazyků pracujících s pgSQL, nejedná se již však o součást hlavní distribuce databáze. Je tedy potřeba modul se žádaným procedurálním jazykem doinstalovat. Výčet těch nejpoužívanějších si můžete prohlédnout v následujícím seznamu:

- PL/Java
- PL/PHP
- PL/Py
- PL/R
- PL/Ruby
- PL/Scheme
- PL/sh

3.2.3 Funkce procedurálního jazyka a SQL standard

Samotný procedurální jazyk SQL standard definuje jako SQL/PSM. Prakticky se jedná o jednodušší obdobu procedurálního jazyka Oracle PL/SQL či PL/pgSQL.

3.3 Interní funkce

Interními funkcemi jsou funkce jazyka C, které jsou staticky propojeny se serverem PostgreSQL. „Tělo“ takovéto funkce definuje jméno příslušné funkce jazyka C. Toto jméno nemusí být totožné se jménem SQL funkce, která se bude v rámci SQL dotazů v databázi používat. Z důvodů zpětné kompatibilty je prázdné tělo funkce přijato ve smyslu, že jméno jazyka C bude totožné se jménem vytvářené SQL funkce. [5]

Standardně jsou všechny interní funkce serveru deklarované během incializace databázového clusteru. Uživatel však může vytvořit své alias jména pro interní funkce (viz. předchozí odstavec). Interní funkce s vlastním aliasem se deklarují opět pomocí SQL příkazu CREATE FUNCTION s tím rozdílem, že se jako použitý jazyk určí klíčové slovo *internal*. Interní funkce by taktéž měly být deklarované jako STRICT, jak je možno vidět na ukázce 17. V praxi si tedy můžeme například srozumitelněji pojmenovávat různé matematické funkce, které C nabízí.

```
CREATE FUNCTION squareRootOfTwo(double precision) RETURNS double precision
AS 'dsqrt' LANGUAGE internal STRICT;

SELECT dsqrt(4);
SELECT squareRootOfTwo(4);
```

Výpis 17: Ukázka deklarace interní funkce pgSQL

Ne všechny „předefinované“ funkce jsou v tomto smyslu „interními“. Některé předefinované funkce jsou napsány v SQL. [5]

3.3.1 Interní funkce a SQL standard

Z důvodu obecnosti SQL standardu nelze se specifikací interních funkcí počítat. Standard totiž nedefinuje implementaci jádra databázového systému prostřednictvím určitého jazyka, kterým je v případě PostgreSQL nízkoúrovňové C.

3.4 Funkce jazyka C

Uživatelsky vytvořené funkce mohou být v rámci rozšířitelnosti PostgreSQL napsány v jazyce C nebo v jazyce, který může být uzpůsoben tak, aby byl kompatibilní s C (například C++). Takovéto funkce se kompilují do dynamicky přístupných objektů, které se často nazývají sdílené knihovny (shared libraries). A následně jsou přístupné serveru na požádání. Tato vlastnost dynamického přístupu je hlavním rozdílem funkcí jazyka C a interních funkcí 3.3. Konvence kódu jsou ale v podstatě pro oba typy funkcí stejné. Proto jsou standardní interní funkce knihovny bohatým zdrojem příkladů kódů pro C funkce definované uživatelem. [5]

V současné době jsou pro volání C funkcí (v rámci zdrojového C kódu) používány dvě rozdílné konvence. Novější konvence „verze 1“ je reprezentována voláním makra PG_FUNCTION_INFO_V1().

Vynechání tohoto makra pak indikuje starý způsob („verze 0“) volání funkce. V každém případě je však nutno specifikovat jazyk vytváření funkce (CREATE FUNCTION) jako LANGUAGE C.

Použití způsobu („verze 0“) je však nyní již zastaralé z důvodu problémů s přenositelností a nedostatku novějších funkcionalit. Stále jsou však funkce využívající „staršího způsobu“ podporované pro zachování kompatibility. [5]

3.4.1 Dynamický přístup

Při prvním volání uživatelsky definované funkce určité sdílené knihovny v rámci sezení (session) je tato knihovna nahrána do paměti, čímž se zajistí možnost tuto funkci spustit. Definice CREATE FUNCTION pro patřičnou uživatelsky vytvořenou C funkci tak musí nést název knihovny a název specifikující funkci v rámci této knihovny (souboru). V případě, že není specifikován název C funkce explicitně, je předpokládáno, že je totožný s názvem SQL funkce.

3.4.1.1 Metody načtení sdílených knihoven a jejich úskalí Pro lokalizaci sdílené knihovny v příkazu CREATE FUNCTION se využívá následujícího algoritmu:

1. Pokud je jménem absolutní cesta, určený soubor je nahrán.
2. V případě, že jméno začíná řetězcem *$libdir*, je tento řetězec nahrazen jménem adresáře balíčkové knihovny PostgreSQL. (jméno je určeno při sestavě)
3. Není-li ve jméně obsažena cesta k adresaři, soubor je hledán v rámci cesty specifikované konfigurační proměnnou *dynamic_library_path*.
4. Pokud není soubor nalezen v určeném adresáři či ve výchozím (při nespecifikování absolutní cesty), server nejpravděpodobněji skončí s chybou.

V případě, že sekvence algoritmu nebudou úspěšné, systém přidá ke jménu sdílené knihovny specifickou příponu závislou (nejčastěji *.so*) na platformě, nad kterou server běží. Následně je nově vzniklá cesta vyzkoušena znova. Pokud se ani po tomto pokusu nepodaří knihovnu nahrát, připojení této knihovny definitivně selže.

Je doporučeno lokalizovat sdílené knihovny buď pomocí relativního *$libdir* nebo skrz cestu nastavenou v konfiguraci (*dynamic_library_path*). Toto usnadňuje upgrade při umístění nové instalace v jiném adresáři. Aktuální adresář, na který odkazuje *$libdir*, pak můžeme zjistit pomocí příkazu *pg_config--pkglibdir*.

Uživatelské ID, pod kterým uživatel PostgreSQL pracuje, musí mít práva přístupu k adresáři, ze kterého máme v úmyslu knihovny nahrávat. Vytvoření souboru nebo adresáře, ke kterému nemá PostgreSQL přístup, bývá častou chybou. [5] V každém případě je však jméno specifikované v rámci příkazu CREATE FUNCTION zaznámeno v systémovém katalogu pro případ opětovného přístupu ke knihovně. Toto slouží k optimalizaci běhu serveru.

PostgreSQL nezajistí kompilaci C funkce automaticky. Sdílená knihovna musí být před použitím deklarace v příkazu CREATE FUNCTION nejprve zkompilovaná.

3.4.1.2 Zajištění kompatibility serveru a rozšíření Pro zajištění kompatibility sdílené knihovny se serverem PostgreSQL se kontroluje, zda soubor knihovny obsahuje „magic blok“ odpovídajícího obsahu. Tato vlastnost zajistí serveru detekci zřejmých problémů spojených s kompatibilitou, pokud je například kód kompilován pro jinou „významnou“ verzi PostgreSQL. Tento blok je vyžadován od verze 8.2. Pro správnou funkčnost magic bloku je nutné v jednom ze zdrojových souborů napsat kód 18 a zároveň přidat hlavičku *fmgr.h*.

Uvození #*ifdef* může být vynecháno, pokud není potřeba kompilovat kód pro starší verze než PostgreSQL 8.2. Po prvním použití je sdílená knihovna uchována v paměti. Budoucí volaní funkcí sdílené knihovny tak v rámci jedné session nezpůsobí zbytečné přetěžování serveru. Pokud chceme vynutit opětovné nahrání knihovny (například v případě rekompilace), je třeba začít novou session.

```
#ifdef PG_MODULE_MAGIC
PG_MODULE_MAGIC;
#endif
```

Výpis 18: Magic blok vyžadovaný v rámci C rozšíření pgSQL

Sdílená knihovna může také obsahovat volitelné incializační a ukončovaní funkce. Jestliže soubor bude obsahovat funkce *_PG_init*, bude tato funkce zavolána neprodleně po načtení. Funkce nepřijímá žádné parametry a měla by vracet *void*. Pokud navíc soubor obsahuje funkci *_PG_fini*, bude ukončovací funkce spuštěna při odpojení knihovny. Stejně jako u inicializační funkce, ukončovací funkce taktéž nepřijímá žádné parametry a měla by vracet *void*. Je však nutné myslet na to, že ukončovací funkce *_PG_fini* bude spuštěna pouze při odpojení knihovny, ne při ukončení procesu.

V soušasné době, je odpojování deaktivováno a nikdy tak nenastane. Toto se to však může v budoucnu změnit. [5]

3.4.2 Base Typy ve funkcích jazyka C

Interně jsou v PostgreSQL základní typy 2.3.1 velkými binárními datovými objekty (blob) paměti. Následující část tedy popisuje, jak pgSQL reprezentuje tyto základní typy v rámci jazyka C a jak je nutno s nimi při vytváření C funkcí pracovat. Base typy mohou být předávány jedním s následujících tří interních formátů.

3.4.2.1 Předání hodnotou s pevnou délkou Typy předáváné hodnotou mohou mít délku pouze 1, 2 nebo 4 bytů (8 bytů je také přípustné, pokud se *sizeof(Datum)* rovná 8). Je důležité si při definování typů dávat pozor na jejich velikost z hlediska různých architektur. Typ *long* je například velmi problematický, jelikož může mít na některých strojích délku 4 byty a na jiných 8. Definice typu *integer* z hlediska typu SQL pak může v rámci kódu C vypadat například takto: *typedef int integer;*.

3.4.2.2 Předání referencí s pevnou délkou Referencí mohou být předány typy s jakoukoliv délkou. Pro vstup či výstup z funkcí PostgreSQL mohou být použity pouze

ukazatele (pointery) na tyto typy. Chceme-li vracet hodnotu tohoto typu, musíme nejprve alokovat správné množství paměti pomocí příkazu *palloc*, následně ji naplnit a vrátit na tuto část paměti pointer. (V případě, že chceme vrátit stejnou hodnotu, jakou je hodnota některého vstupního parametru a parametr je navíc stejného typu, můžeme přeskočit alokaci a vrátit pointer přímo na hodnotu vstupu.) Na kódu 19 pak můžeme vidět definici PostgreSQL typu *Point* jako strukturu v jazyce C, tento „typ“ má pevnou délku a můžeme jej předávat referencí.

```
typedef struct
{
    double x, y;
} Point;
```

Výpis 19: Definice typu *Point* v rámci PostgreSQL C kódu [5]

3.4.2.3 Předání referencí s variabilní délkou Všechny typy s variabilní délkou musejí být předány jako reference. Všechny tyto typy musejí také začínat atributem s přesnou délkou - 4 byty, který bude nastaven pomocí SET_VARSIZE;. Tento atribut nesmí být nikdy naplněn přímo. Data uložena v rámci typu s variabilní délkou se musí nacházet ihned za délkovým atributem. Délkový atribut pak obsahuje délku celé struktury, to znamená, že v délce musí být obsažena i velikost samotného délkového atributu. Nikdy nemodifikujte obsah vstupní hodnoty předané referencí. Pokud tak učiníte, je možné, že dojde k požkození dat na disku, jelikož pointer, se kterým pracujete, může ukazovat přímo na buffer v disku. Opět si definici tohoto typu můžeme prohlédnout na zdrojovém kódu 20 jazyka C.

```
typedef struct {
int32 length;
char data[1];
} text;
```

Výpis 20: Definice typu *text* v rámci PostgreSQL C kódu [5]

Jak jste si určitě všimli, atribut *data* není dostatečně velký na to, aby struktura udržela všechny možnosti řetězců. Jelikož je nemožné v jazyce C deklarovat strukturu s varibilní velikostí, spoleháme na poznatek, že kompilér C nebude kontrolovat rozsah indexů polí. Alokujeme tedy potřebné možství místa a následně přistoupíme k poli jakoby bylo deklarovaná správná délka. (Jedná se o známý „trik“, o kterém se můžeme dočíst v mnohých knihách o C.) [5] Při manipulaci s proměnnými typy pak musíme důkladně alokovat správné množství paměti a správně nastavit atribut s velikostí. Ukázku alokace struktury text na délku 100 lze vidět na zdrojovém kódu 21. Jak již zbylo zmíněno, atribut držící délku musí být nastaven příkazem SET_VARSIZE;. Následně můžeme vstupní data překopírovat do alokované struktury.

```
char inputData[100]; /* vstupni data */
text *textString = (text *) palloc(sizeof(int32) + 100);
SET_VARSIZE(textString, sizeof(int32) + 100);
memcpy(textString->data, buffer, 100);
```

Výpis 21: Alokace typu *text* v rámci PostgreSQL C kódu

3.4.2.4 Ekvivalence datových typů SQL a jejich implementace v C

Tabulka 3.4.2.4 znázorňuje datové typy jazyka C, které je nutné při vytváření C funkcí použít pro manipulaci s korespondujícími vestavěnými SQL typy. Poslední sloupec pak říká, jakou knihovnu je do příslušeného C kódu nutné přidat.

Typ jazyka C	PostgreSQL typ	Knihovna
abstime	AbsoluteTime	utils/nabstime.h
boolean	bool	postgres.h
box	BOX*	utils/geo_decls.h
bytea	bytea*	postgres.h
"char"	char	(compiler built-in)
character	BpChar*	postgres.h
cid	CommandId	postgres.h
date	DateADT	utils/date.h
smallint (int2)	int16	postgres.h
int2vector	int2vector*	postgres.h
integer (int4)	int32	postgres.h
real (float4)	float4*	postgres.h
double precision	(float8) float8*	postgres.h
interval	Interval*	datatype/timestamp.h
lseg	LSEG*	utils/geo_decls.h
name	Name	postgres.h
oid	Oid	postgres.h
oidvector	oidvector*	postgres.h
path	PATH*	utils/geo_decls.h
point	POINT*	utils/geo_decls.h
regproc	regproc	postgres.h
reltime	RelativeTime	utils/nabstime.h
text	text*	postgres.h
tid	ItemPointer	storage/itemptr.h
time	TimeADT	utils/date.h
time with time zone	TimeTzADT	utils/date.h
timestamp	Timestamp*	datatype/timestamp.h
tinterval	TimeInterval	utils/nabstime.h
varchar	VarChar*	postgres.h
xid	TransactionId	postgres.h

3.4.3 C funkce psané dle konvence verze 0

V případě psaní funkce dle verze 0, jsou argumenty a výsledky C funkce deklarovány stejným způsobem jako v normálním „čistém“ C kódu. Je však stále nutné dát si pozor na ekvivaletntní C typy pro každý datový typ SQL. 3.4.2.4. Psaní funkcí dle nulové konvence je zastaralé. Nicméně pro vstup do problematiky psání C rozšíření PostgreSQL je zpočátku jednodušší využít právě tuto starší metodu. [5] Příklady implementace dvou jednoduchých funkcí dle konvence verze 0 lze demonstrovat na ukázkách 22 a 23. První ukázka demonstruje velmi jednoduchou funkci, která vrací dvojnásobek vstupního *integer* parametru. Jak je možno vidět, kód není nutné nijak zvlášť oproti standardnímu C kódu upravovat. V druhé ukázce pak můžeme vidět již komplexnější funkci, která z původních dvou *text* parametrů vrátí jeden spojený *text* řetězec. Je zde potřebná alokace struktury *text*, jelikož se jedná o typ předávaný referencí s proměnnou délkou 3.4.2.3. Jako první se nesmí opomenout přiložit příslušené hlavičky. Hlavičku *postgres.h* je nutné přidat vždy. Následně bychom v aktuálních verzích PostgreSQL také neměli zapomenout na magic blok, o kterém bylo psáno v části 3.4.1.2.

```
#include "postgres.h"

#ifdef PG_MODULE_MAGIC
PG_MODULE_MAGIC;
#endif

int multiplyByTwo(int parameter)
{
    int result = parameter * 2;
    return result;
}
```

Výpis 22: Ukázka uživatelské C funkce dle konvence verze 0 (dvojnásobek)

V případě ukázky spojení řetězců bylo potřeba použití specifických funkcí a konstanty, které hlavička *postgres.h* nabízí. Konstanta *VARHDRSZ* je v ekvivalentem *sizeof(int32)*. Funkce *VARSIZE(řetězec)* pak získá délku řetězce a funkce *VARDATA* vrací pole znaků reprezentující řetězec v rámci struktury *text*; Po vytvoření nové proměnné typu text*, její alokaci a následném kopírování dvou vstupních řetězců, vracíme výsledný spojený řetězec.

```
#include "postgres.h"
#include <string.h>

#ifdef PG_MODULE_MAGIC
PG_MODULE_MAGIC;
#endif

text* joinTexts(text* firstPart, text* lastPart)
{
    int32 resultSize = VARSIZE(firstPart) + VARSIZE(lastPart) - VARHDRSZ;
    text* result = (text*) palloc(new_text_size);

    SET_VARSIZE(result, resultSize);
```

```
    memcpy(VARDATA(new_text), VARDATA(firstPart), VARSIZE(firstPart) − VARHDRSZ);
    memcpy(VARDATA(new_text) + (VARSIZE(firstPart) − VARHDRSZ), VARDATA(lastPart),
        VARSIZE(lastPart) − VARHDRSZ);
    return result ;
}
```

Výpis 23: Ukázka uživatelské C funkce dle konvence verze 0 (spojení řetězců)

3.4.3.1 Deklarace C funkcí v SQL Deklarace funkcí 22 a 23 bude v rámci dotazů SQL poté vypadat následovně 24. Řetězcem *XYZ* za klíčovým slovem AS (u první funkce) se určí adresář, kde se nachází soubor sdílené knihovny. Vhodnějším určením cesty je však definování pouze názvu knihovny (druhá funkce) a přidání cesty k adresáři mezi vyhledávané cesty 3.4.1.1. V každém případě však můžeme vynechat přípony sdílených knihoven (například .dll, .so, .sl či jiné). Klíčové slovo STRICT, které se nachází u určení jazyka funkce, znamená, že systém automaticky ošetří vstupní hodnoty *null*. Využitím tohoto způsobu tedy nemusíme ošetřovat případy „nullových" hodnot v rámci kódu C. Bez deklarování jazyka jako STRICT je nutné ošetřit *null* hodnoty explicitně.

```
CREATE FUNCTION multiplyByTwo(integer) RETURNS integer
AS 'XYZ/functions', 'multiplyByTwo'
LANGUAGE C STRICT;

CREATE FUNCTION joinTexts(text, text) RETURNS text
AS 'functions', 'joinTexts'
LANGUAGE C STRICT;
```

Výpis 24: Deklarace C Funkcí v SQL

3.4.4 C funkce psané dle konvence verze 1

Ačkoli je psaní funkcí dle starší konvence 3.4.3 jednodušší, nejsou tyto funkce dobře přenositelné. Na některých architekturách se při použití této starší metody nachází problémy v předávání datových typů, jenž jsou menší než *int*. Navíc zde neexistuje jednoduchý způsob vracení „nullové" hodnoty či jednoduché ošetření *null* parametrů funkce, která není deklarovaná jako STRICT. Následující část se věnuje konvenci verze 1, která všechny tyto problémy řeší. [5]

Při implementaci (dle konvence verze 1) je vždy nutné deklarovat volané C funkce tímto způsobem: *Datum nazevFunkce(PG_FUNCTION_ARGS)*. Následně je také nutné definovat makro *PG_FUNCTION_INFO_V1(nazevFunkce)*, které se musí nacházet společně s implementací funkce ve stejném zdrojovém souboru. Ve verzi 1 lze získat jednotlivé argumenty pomocí funkcí *PG_GETARG_datovyTyp(indexArgumentu)* příslušných pro každý jednotlivý base typ 2.3.1. Parametrem se pomocí čísla určuje pořadí chtěného argumentu. (Indexy argumentů začínají nulou.) Podobným způsobem pak funguje prostřednictvím funkcí *PG_RETURN_datovyTyp(navratovaHodnota* i návrat funkce.

Příklady C funkcí uvedené v části věnující se starší konvenci (verze 0) 3.4.3 budou pak v rámci novější konvence vypadat následovně (25 a 26). Deklarace funkcí v rámci SQL 24 zůstane nezměněna.

```c
#include "postgres.h"
#include "fmgr.h"

#ifdef PG_MODULE_MAGIC
PG_MODULE_MAGIC;
#endif

Datum multiplyByTwo(PG_FUNCTION_ARGS)
{
int32 parameter = PG_GETARG_INT32(0);
int32 result = parameter * 2;
return result;
}
PG_FUNCTION_INFO_V1(multiplyByTwo);
```

Výpis 25: Ukázka uživatelské C funkce dle konvence verze 1 (dvojnásobek)

```c
#include "postgres.h"
#include "fmgr.h"
#include <string.h>

#ifdef PG_MODULE_MAGIC
PG_MODULE_MAGIC;
#endif

Datum joinTexts(PG_FUNCTION_ARGS)
{
text* firstPart = PG_GETARG_TEXT_P(0);
text* lastPart = PG_GETARG_TEXT_P(1);

int32 resultSize = VARSIZE(firstPart) + VARSIZE(lastPart) - VARHDRSZ;
text* result = (text*) palloc(new_text_size);

SET_VARSIZE(result, resultSize);
memcpy(VARDATA(new_text), VARDATA(firstPart), VARSIZE(firstPart) - VARHDRSZ);
memcpy(VARDATA(new_text) + (VARSIZE(firstPart) - VARHDRSZ), VARDATA(lastPart), VARSIZE(
    lastPart) - VARHDRSZ);
PG_RETURN_TEXT_P(result);
}
PG_FUNCTION_INFO_V1(joinTexts);
```

Výpis 26: Ukázka uživatelské C funkce dle konvence verze 1 (spojení řetězců)

Na první pohled se může zdát, že konvence verze 1 je zbytečná, nepřináší mnoho výhod a navíc dělá kód složitějším. Toto tvrzení však není pravdivé. Výčet hlavních výhod konvence verze 1 můžeme pozorovat níže.

- GETARG funkce nám umožňují lepší zpracování variabilního počtu argumentů funkce.
- Nová verze nabízí funkci *PG_ARGISNULL(n)* pro testování *null* hodnot argumentů. (V případě „nestriktních“ funkcí.)

- Novější verze taktéž nabízí funkci *PG_RETURN_NULL()* pro vrácení *null* hodnoty.
- Nová verze nabízí funkce *PG_GETARG_datovyTyp_COPY()*, které vrací kopie hodnot argumentů, je tedy možné bez problémů tyto kopie modifikovat. (Standardní funkce může v určitých případech vracet pointery, se kterými není vhodné z hlediska konzistence manipulovat.)
- Funkce *PG_GETARG_datovyTyp_SLICE()* pak nabízí možnost vyběrů částí argumentů (vhodné zejména u větších hodnot).
- Funkce definovavné dle konvence verze 1 mohou navracet SETy.
- Větší přenositelnost funkcí.

3.4.5 Základní pravidla implementací C funkcí pro pgSQL

- Kompilace a linkování uživatelského kódu tak, aby bylo možné jej dynamicky do PostgreSQL nahrát, vyžaduje speciální značky.
- Zdrojový kód určený pro novější verze PostgreSQL musí obsahovat magic blok 3.4.1.2.
- Pro alokaci paměti by se mělo využít příkazů *palloc* a *pfree* místo ekvivalentích příkazů *malloc* a *free* standardní C knihovny *stdlib.h*. Pamět alokovaná pomocí *palloc* bude automaticky vyčistěna při konci každé transakce, čímž se zamezí úniku paměti.
- Identifikátory uvnitř sdílených knihoven nesmí být konfliktní s identikátory definovanými v rámci PostgreSQL serveru. Při výskytu chyby je tak třeba přejmenovat definované funkce či proměnné.
- Většina interních PostgreSQL datových typů je deklarovaná v knihovně *postgres.h*, zatímco funkce typu PG_FUNCTION_ARGS a jiné v hlavičce *fmgr.h*. Je tedy potřeba přiložit alespoň tyto dvě knihovny. Z důvodu přenositelnosti, je nejlepší přiložit knihovnu *postgres.h* jako první před všemi ostatními systémovými či uživatelskými hlavičkovými soubory.

Ačkoli by mohlo být možné do pgSQL nahrát funkce psané v jiných jazycích než je C, je to často velmi obtížné či neproveditelné, jelikož ostatní jazyky jako C++, FORTRAN či Pascal často nedodržují stejné konvence volání jako C. To znamená, že nepředávají argumenty a nevrací hodnoty mezi funkcemi stejným způsobem. [5]

3.4.6 Funkce C s argumenty kompozitního typu

Kompozitní typy 2.3.2 nemají fixní podobu jako struktury v C. Instance kompozitního typu může obsahovat atributy s hodnotami *null*. Navíc mohou kompozitní typy, které tvoří část hiearchie dědičnosti, mít jiné atributy než jiné části stejné hiearchie. Z tohoto

důvodu PostgreSQL poskytuje v C rozhraní pro přístup k kompozitním typům. Předpokládejme dotaz 27 nad tabulkou s naplněnými daty definovanou v rámci SQL kódu 12.

```
SELECT name, isSuccessful(Student) )FROM Student WHERE name = "Jack␣Johnson";
```

Výpis 27: SQL funkce jako tabulkový zdroj

Dotaz volá funkci *isSuccessful(Student)*, kterému předává kompozitní typ v podobě tabulky *Student*. Implementaci funkce *isSuccessfull()* je pak možno pozorovat v rámci kódu 28. Funkce nejprve načte do pomocné struktury typu *HeapTupleHeader* celý kompozitní typ a následně z něj pomocí funkce *GetAttributeByName* získá požadovaný atribut, který uloží do pomocné proměnné *points*. Na základě hodnoty této proměnné rozhodne, jestli vratit *true* či *false*.

```
#include "postgres.h"
#include "executor/executor.h"  /* pro funkci GetAttributeByName() */

#ifdef PG_MODULE_MAGIC
PG_MODULE_MAGIC;
#endif

Datum isSuccessful(PG_FUNCTION_ARGS)
{
    HeapTupleHeader tuple = PG_GETARG_HEAPTUPLEHEADER(0);
    bool isNull;
    bool successful;
    Datum points;

    points = GetAttributeByName(tuple, "points", &isNull);

    if (isNull)
    {
        PG_RETURN_NULL();
    }

    if (DatumGetBool(points) > 50)
    {
            PG_RETURN_BOOL(true);
    }
    else
    {
            PG_RETURN_BOOL(false);
    }
}
PG_FUNCTION_INFO_V1(isSuccessful);
```

Výpis 28: Ukázka uživatelské C funkce s argumentem kompozitního typu

3.4.7 C funkce navracející kompozitní typy a SET

Pokud chceme zajistit to, aby funkce napsaná v jazyce C navracela kompozitní typ (řádek) či SET (více řádků) musí se využít dostupného API v podobně hlavičového souboru *funcapi.h*. V obou těchto případech je nutné použít specifické funkce, které API poskytuje a pomocí nich vytvářet struktury (a následně je naplnit). Více o použití tohoto API se lze dočíst v oficiální dokumentaci pgSQL.

3.4.8 Polymorfní C funkce

Funkce jazyka C mohou být deklarovány tak, aby přijímaly či vracely polymorfní typy *anyelement, anyarray, anynoarray anyenum* či *anyrange*. Více se o těchto typech můžete dočíst v části 2.3.5. Pokud funkce příjímá či vrací polymorfní typy, autor funkce nemůže předem vědět, s jakým datovým typem bude funkce operovat. Existují však metody, jak v rámci C funkce dodržující konvenci verze 1 3.4.4, tyto problémy vyřešit. Je ale potřeba využít hlavičkového souboru *fmgr.h*.

3.4.8.1 Funkce řešící polymorfismus Funkce, které tato hlavička poskytuje, umožňují „odkrývat“ datové typy parametrů a také určit, jaký datový typ se má vracet. Metoda tohoto odkrývání spočívá v použití funkcí *get_fn_expr_rettype(FmgrInfo *flinfo)* a *get_fn_expr_argtype(FmgrInfo *flinfo, int argnum)*. Výsledek těchto funkcí vrací hodnotu datového typu *OID* či *InvalidOid* v případě nedostupnosti informace. Struktura *flinfo* je přístupná skrze *fcinfo->flinfo*. Parametr *argnum* sloužící jako index parametru je pak opět indexován od nuly.

Existuje zde také funkce *get_fn_expr_variadic*, která poskytuje zjištění, zda byly VARIADIC parametry 3.1.6 sloučeny do pole. Toto je primárně užitečné pro „any“ funkce s VARIADIC parametry, jelikož tato slučování se provádějí vždy, když funkce (s použitím VARIADIC) příjímají obyčejné pole (array). [5]

3.4.8.2 Polymorfní "any" funkce Funkce jazyka C mohou využít i jiné „varianty“ polymorfismu, mohou totiž být deklarovány s argumenty typu *"any"*. Jméno takového datového typu musí být uvozeno dvojitými závorkami ("), jelikož se navíc jedná i o klíčové slovo jazyka SQL. Tato „varianta“ polymorfismu funguje podobným způsobem jako typ *anyelement* s tím rozdílem, že neomezuje jednotlivé *"any"* argumenty na stejný typ a ani neurčuje návratový typ funkce. C funkce může mít na rozdíl od SQL funkcí 3.1 svůj poslední parametr deklarován jako VARIADIC *"any"*. V takovém případě pokryje jeden či více argumentů a jako plus nemusí být stejného typu. Argumenty totiž nebudou tvořit pole (array) jako u standardních VARIADIC funkcí. Z tohoto důvodu budou argumenty předány funkci odděleně. Pomocí makra *PG_NARGS()* a výše popsaných funkcí je pak ale nutné v rámci těla funkce získat aktuální počet parametrů a jejich typy. Uživatelé by také měli využít klíčového slova VARIADIC při volání těchto funkcí a to s předpokladem, že funkce bude pracovat s elementy pole jako s oddělenými argumenty. Sama funkce pak musí využít vlastnosti *get_fn_expr_variadic* k detekci použití argumentu označeného klíčovým slovem VARIADIC.

3.4.9 Využití C++ z hlediska rozšířitelnosti pgSQL

I přesto, že je jádro pgSQL napsáno v jazyce C, je možné vytvářet rozšíření v C++ za předpokladu dodržení následujících zásad. Stručně řečeno je potřebné vyhnout se přetečení paměti, vyjímkám a taktéž nedovolit C funkcím pgSQL přímo přistupovat ke kódu jazyka C++.

1. Všechny funkce oslovené jádrem pgSQL musí obsahovat C rozhraní. Tyto funkce následně však již mohou volat jiné C++ funkce. Je tak například vyžadováno propojení pomocí *extern* pro funkce přístupné jádru pgSQL. Toto je tak nutné i pro funkce, jenž jsou mezi jádrem a C++ kódem předávány pomocí pointeru. [5]

2. Dealokace (čistění) pamětí musí probíhat pomocích příslušných metod. Většina paměti jádra je alokována pomocí *palloc()*, je tedy nutné použít *pfree()* k její dealokaci. Použití *delete* jazyka C++ tak v takovýchto případech selže.

3. Je taktéž důležité obalit C++ kód tak, aby se případné vyhozené vyjímky nikdy nedostaly až do čistého kódu C. Je nutné obalit celý C++ kód i v případě, že se explicitně žádné vyjímky nekonají a to z důvodu implicitních vyjímek typu nedostatku paměti.

3.4.10 Funkce jazyka C a SQL standard

Standard definuje možnost volání funkcí jazyka Java a to v časti 14 nazvané SQL Routines and Types Using the Java Programming Language (SQL/JRT). Je tedy možné využívat funkcionalit a knihoven jazyka Java. Co se však funkcí jazyka C týče, ty standardem specifikovány nejsou.

3.5 Přetěžování funkcí pgSQL

Funkce mohou mít v pgSQL stejný identifikátor (jméno) právě, pokud mají rozdílné vstupní parametry. Jinými slovy mohou být funkce přetěžovány. Server se při volání takovýchto funkcí rozhodne jakou funkci zavolat a to na základě datových typů a počtu poskytnutých argumentů. Přetěžování tak může například sloužit k simulaci funkcí s proměnným počtem argumentů (do nějakého konečného počtu).

Funkce, která přijímá jeden argument kompozitního typu 2.3.2, by neměla mít stejné jméno jako jméno onoho atributu. Může totiž dojít k nejednoznačnosti mezi funkcí pracující nad kompozitním typem a atributem kompozitního typu. Volání funkce *parametr(cislo)* je pak tedy ekvivaletní zápisu přístupu k atributu kompozitního typu *cislo.parametr*. V takovém případě bude vždy použit atribut. Lze však dosáhnout i opačného efektu a to kvantifikací schématu, kterému funkce náleží *schema.funkce(cislo)*. Nicméně je doporučeno vyhnout se těmto komplikacím volbou nekonfliktních jmen.

Další problém může nastat mezi funkcemi s proměnným počtem argumentů 3.1.6 a těmi s pevným počtem parametrů. Je tak například možné vytvořit funkce *funkce(integer)* a *funkce(VARIADIC integer[])*, problém ale nastane v případě, kdy zavoláme funkci a předáme ji pouze jeden parametr. (V našem případě by takové volání vypadalo například

takto: *funkce(1)*.) Jestliže se nachází obě funkce ve stejném schématu databáze, tak se zavolá ta s pevným počtem parametrů. V opačném případě je použita ta funkce, která se nachází jako první z hlediska vyhledávané cesty.

Pro přetěžování funkci jazyka C 3.4 pak existuje další pravidlo. Jména C funkcí (přímo ve zdrojovém kódu C) musí být v rámci přetěžovaných funkcí jiná než jména všech ostatních C funkcí, ať už se jedná o funkce interní či dynamicky načtené. Pokud se toto pravidlo nedodrží, je chování nepředvídatelné. Může se vyskytnout chyba běhu *(run-time linker error)*, v lepším případě se pak zavolá jedna z patřičných interních funkcí 3.3. Alternativní použití klauzule AS pro SQL příkaz CREATE FUNCTION pak odděluje název SQL funkce od názvu C funkce (ve zdrojovém kódu). Z pohledu SQL tak jsou funkce přetěžovány s tím, že jim v rámci zdrojového kódu C přiřadíme rozdílně identifikované funkce. Jednoduchou ukázku takového přetížění (s použitím AS klauzule) můžeme pozorovat na kódu 29.

```
CREATE FUNCTION cFunction(int) RETURNS int AS 'pgSQLExtension', '
    functionWithOneParameter' LANGUAGE C;
CREATE FUNCTION cFunction(int, text) RETURNS int AS 'pgSQLExtension', '
    functionWithTwoParameters' LANGUAGE C;
```

Výpis 29: Přetěžování funkcí C

3.6 Uživatelsky vytvořené datové typy

Jak již bylo zmíněno v úvodu 2.4, pgSQL nabízí uživatelům v rámci rozšiřitelnosti možnost vytvářet vlastní datové typy. Při vytváření nových základních typů 2.3.1 je nutné implementovat strukturu a funkce, které definují operace nového typu, a to v nízkoúrovňovém jazyku. Většinou se tak jedná o jazyk C.

Uživatelsky definovaný typ musí vždy obsahovat vstupní a výstupní funkce. Tyto funkce tak určují, jak reprezentovat řetězec vstupu a výstupu a také jak bude typ organizován z hlediska paměti. Vstupní funkce přijímá znakový řezězec ukončený hodnotou *null* (null-terminated) a vrací interní (paměťovou) reprezentaci typu. Výstupní funkce pak naopak příjímá interní reprezentaci typu a vrací znakový (null-terminated) řetězec. V případě, že ale chceme s typem pracovat více než ho jen ukládat, musíme pro tyto operace implementovat dodatečné funkce.

3.6.1 Definování vlastního typu v jazyce C

Můžeme například definovat typ *Student*, který bude obsahovat číslo identifikace studenta a zároveň jeho bodový zisk. Nejprve je nutné vytvořit strukturu v C 30, která tento typ bude představovat (v paměti).

```
typedef struct Student
{
    int id; int points;
} Student;
```

Výpis 30: C struktura uživatelského datového typu

3.6.1.1 Vstupní funkce uživatelského typu Řetězec vstupu potom bude mít tuto formu: *(id, points)*. Implementace vstupních a výstupních funkcí není velmi obtížná, obzvlášť pak pro náš ukázkový typ. Bývá nutné pro vstupní řetězec implementovat kompletní a funkční parser. (V našem případě to ale není potřeba). Zdrojový kód vstupní funkce pro typ *Student* může vypadat následovně 31. Funkce nejprve získa celý vstupní řetězec. V případě nesprávného řetězce pak vyhodí interní pgSQL chybu, která uživateli oznámí neplatnost vstupního řetězce. V opačném případě pak vrátí ukazatel na vytvořenou a naplněnou strukturu *Student*.

```
Datum studentInput(PG_FUNCTION_ARGS)
{
    char *inputString = PG_GETARG_CSTRING(0);
    int id;
    int points;
    Student *result;

    if (sscanf(inputString, "(%i, %i)", &id, &points) != 2)
    {
        ereport(ERROR, (errcode(ERRCODE_INVALID_TEXT_REPRESENTATION), errmsg("Invalid
            input syntax for Student type: \"%s\"", inputString)));
    }

    result = (Student *) palloc(sizeof(Studentplex));
    result->id = id;
    result->points = points;
    PG_RETURN_POINTER(result);
}
PG_FUNCTION_INFO_V1(studentInput);
```

Výpis 31: Vstupní funkce uživatelského typu (Student)

3.6.1.2 Výstupní funkce uživatelského typu Následná implementace výstupní funkce je pak o něco jednodušší. Výstupní funkce našeho typu *Student* lze pozorovat na zdrojovém kódu 32. Vstupní a výstupní funkce v podstatě tvoří vzájemnou inverzi sebe samých. Při implementaci, kdy se bude strukturou pracovat jiným způsobem, mohou nastat problémy s nekonzistencí. Časté problémy se vyskytují hlavně u typů, které zahrnují desetinná čísla. [5]

```
Datum studentOutput(PG_FUNCTION_ARGS)
{
    Student *student = (Student *) PG_GETARG_POINTER(0);
    char *result;

    result = psprintf("(%i,%i)", student->id, student->points);
    PG_RETURN_CSTRING(result);
}
PG_FUNCTION_INFO_V1(studentOutput);
```

Výpis 32: Výstupní funkce uživatelského typu (Student)

3.6.2 Definování C funkce na úrovní SQL

Po napsání a zkompilování vstupních a výstupních funkcí do sdílené knihovny 3.4.1.1, je v rámci SQL nejprve nutné vytvořit jednoduchý uživatelský typ (takto *CREATE TYPE Student;*). Druhým krokem je registrace vstupních a výstupních funkcí. To se provádí stejným postupem, kterému se již věnovala část o funkcích jazyka C 3.4.3.1. Demonstrace vstupní a výstupní funkce vlastního typu *Student* lze vidět na kódu 33.

```
CREATE FUNCTION studentInput(cstring)
RETURNS Student
AS 'studentType'
LANGUAGE C IMMUTABLE STRICT;

CREATE FUNCTION studentOutput(Student)
RETURNS cstring
AS 'studentType'
LANGUAGE C IMMUTABLE STRICT;
```

Výpis 33: Definice vstupní a výstupní funkce uživatelského typu (Student)

Posledním nutným krokem registrace nového uživatelské typu je úplná typová definice 34. Definice obsahuje registraci vstupní a výstupní funkce společně s velikostí struktury, která je v našem případě pevná a to v podobně dvou 4-bytových integer hodnot.

```
CREATE TYPE Student (
 internallength  = 8,
input = studentInput,
output = studentOutput);
```

Výpis 34: Úplná typová definice uživatelského typu

3.6.3 Dodatek k uživatelským typům

- PostgreSQL poskytuje při definici nového base typu automatickou podporu pro příslušné pole (arrays).
- Jakmile nový datový typ existuje, můžeme mu poskytnout užitečnou funkcionalitu v podobě přídavných funkcí.

3.6.4 Uživatelsky vytvořené typy a SQL standard

Vytváření uživatelských typů (CREATE TYPE) je definováno standardem již od roku 1999 (SQL:1999). Nejedná se však o vytváření typů prostřednictvím struktur a funkcí jazyka C. Dle specifikace lze využít pouze jazyky SQL či Java (SQL/JRT).

3.7 Balíčky souvísejících objektů

Praktické rozšíření PostgreSQL typicky obsahuje několik SQL objektů (datový typ 3.6, funkce 3.1, operátory, třídy operátorů pro indexy). Je tak vhodné „zabalit“ všechny tyto objekty do jednoho společného balíčku pro jednodušší správu databáze. Z hlediska pgSQL se takovému balíčku říká rozšíření (extension). K vytvoření rozšíření je potřeba alespoň jeden SQL script obsahující SQL příkazy vytvářející potřebné objekty a také kontrolní soubor, který obsahuje základní konfiguraci rozšíření. Pokud rozšíření navíc zahrnuje kód jazyka C, je také potřeba přiložit soubor příslušné sdílené knihovny 3.4.1.1, jenž tento kompilovaný C kód obsahuje. Následně je možné pomocí příkazu CREATE EXTENSION objekty rozšíření do databáze nahrát.

Výhodou využití rozšíření (extension) oproti spustění klasického SQL skriptu obsahujícího potřebné objekty je to, že PostgreSQL „pochopí“ vzájemnou vazbu mezi objekty rozšíření. Je tak možno všechny objekty rozšíření (čili celé rozšíření) vymazat pomocí jediného příkazu DROP EXTENSION. [6] PostgreSQL navíc ani pokus o vymazání jednotlivých objektů tvořící rozšíření nepovolí. Dovolí tak pouze smazání rozšíření jako celku. Ačkoli je dovoleno upravovat členské objekty rozšíření (např. CREATE OR REPLACE FUNCTION u funkcí), je nutné si uvědomit, že se smazání takto upravených objektů při smazání celého rozšíření neprojeví. Je tak vhodné upravit i přiložený skript rozšíření.

Autor rozšíření také může využít příkazu ALTER EXTENSION UPDATE, který slouží k modifikaci či aktualizaci již instalovaného rozšíření databáze. Autor musí dodat modifikační skripty, které kupříkladu přidávájí nové funkce a upravují implementaci již existujících funkcí.

Objekty představující celé databáze, jejich role, tabulkové prostory nemohou být členy rozšíření, jelikož rozsah rozšíření nepřesahuje hranice jedné databáze. I přesto, že tabulka může být členem rozšíření, jejich podružné objekty (jako například indexy) jimi být nemohou. Přesné vymezení druhů objektů, které mohou být členy rozšíření, je možno dohledat v popisu příkazu ALTER EXTENSION v oficiální dokumentaci PostgreSQL.

3.7.1 Soubory rozšíření pgSQL

Příkaz CREATE EXTENSION je závislý na kontrolním souboru, který musí být pojmenován totožně jako název rozšíření a zároveň také musí mít příponu *.control*. Umístěn probíhá v rámci instalačního adresáře *SHAREDIR/rozšíření*. Tento adresář, jak již bylo v úvodu této podkapitoly zmíněno, musí také obsahovat alespoň jeden soubor SQL příkazů (skript). Název souboru obsahujucí skript pak nese název například *extension--2.0.sql*. Dvojitá pomlčka (--) v tomto názvu souboru odděluje samotné označení rozšíření a jeho verzi. V případě potřeby lze ale kontrolním souborem *.control* určit i jiný adresář pro umístění SQL skriptů.

3.7.1.1 Struktura kontrolního souboru rozšíření

Formát kontrolního souboru rozšíření pgSQL je stejný jako většina konfiguračních souborů databáze. V podstatě se jedná o slovník typu parametr-hodnota a to ve formě *jménoParametru = hodnota(y)*. Každý parametr se svou hodnotou je pak oddělen novým řádkem. Jsou povoleny prázdné řádky

a komentáře pomocí znaku (#). Výčet parametrů kontrolního souboru předvádí následující seznam:

- **directory** *(string)* => Adresář obsahující SQL skripty. Při neuvedení absolutní cesty, je uvedená cesta brána relativně vůči adresáři SHAREDIR. Výchozí hodnota je ekvivaletní relativní cestě *'extension'*.
- **default_version** *(string)* => Výchozí verze rozšíření, která bude při příkazu CREATE EXTENSION instalována. (Pokud není určeno jinak.)
- **comment** *(string)* => libovolný komentář k rozšíření
- **encoding** *(string)* => Znakové kodování používáné v rámci souborů se skripty. Tento parametr by měl být specifikován, pokud SQL skript obsahuje znaky neobsažené ve standardu ASCII. V případě neurčení, bude použito kodování databáze.
- **module_pathname** *(string)* => Hodnota tohoto parametru bude nahrazovat každý výskyt konstanty *MODULE_PATHNAME* v rámci souborů se skripty. Pokud tento parametr nenastavíme, nebude se konat žádná taková substituce. Typicky se tento parametr nastavuje s hodnotou *$libdir/shared_library_name*. Následně se pak při vytváření C funkcí příkazem CREATE FUNCTION využívá zmíněné konstanty *MODULE_PATHNAME* k tomu, aby se nemuselo uvádět jméno sdílených knihoven „natvrdo“.
- **requires** *(string)* => Seznam jmen rozšíření, na kterých toto rozšíření závisí. (Příklad *requires = 'extension1, extension2'*. Tato rozšíření však musí být instalována jako první.
- **superuser** *(boolean)* => Pokud je tento parameter *true*, znamená to, že mohou rozšíření vytvářet a aktualizovat pouze administrátoři (superusers). V opačném případě (*false*) stačí uživatelům vlastnit příslušná práva (vykonávání příkazu instalace či aktualizace rozšíření).
- **relocatable** *(boolean)* => Tento parametr určuje přemístitelnost rozšíření. To znamená možnost přenést obsažené objekty (členy) do jiného schématu. Výchozím nastavením je *false*.
- **schema** *(string)* => Tento parametr může být nastaven pouze pro nepřemístitelné rozšíření. Parametr vynucuje nahrání rozšíření do určeného schématu.

Lze přiložit i sekundární kontrolní soubory ve formátu *extension--verze.control*. Tyto kontrolní soubory se ale musí nacházet ve stejném adresáři jako SQL skripty. Struktura těchto sekundárních kontrolních souborů je totožná jako u těch primárních. Parametry sekundárního souboru navíc při instalaci či aktualizaci rozšíření přepíšou hodnoty souboru primárního. Nicméně, parametry *directory* a *default_version* nelze v rámci sekundárního souboru nastavit. [6]

3.7.1.2 Zásady SQL skriptu rozšíření

Skriptovací SQL soubory rozšíření mohou obsahovat jakékoli SQL příkazy až na ty, které slouží k ovládání transakcí (BEGIN, COMMIT, ...) a zároveň nemohou obsahovat příkazy, které nemohou být vykonány v rámci bloku transakce (například VACUUM) a to z důvodu, že celé skriptovací soubory jsou vykonávány jako transakce.

I když mohou soubory SQL skriptů obsahovat jakékoli znaky povolené určeným kodováním, kontrolní soubory by měly obsahovat pouze znaky, jenž jsou obsazené ve standardu ASCII. PostgreSQL totiž nemá možnost zjistit kódování kontrolních souborů. V praxi je toto problémem jen při použití znaků neodpovídajících ASCII v parametru *comment* 3.7.1.1. V tomto případě je tak doporučeno nepoužívat komentář jako parametr kontrolního souboru, ale vytvořit komentář pomocí příkazu COMMENT ON EXTENSION uvnitř souboru obsahující SQL skript. [5]

3.7.2 Ukázka jednoduchého SQL rozšíření

Na následujícím příkladu si ukážeme, jak lze jednoduché rozšíření (využívajícího pouze SQL) vytvořit. Rozšíření zahrnuje vytvoření jednoduchého typu *Student* a dvou SQL funkcí 3.1. Za předpokladu existence souborů *example--1.0.sql* 3.7.2.1 a *example.control* 3.7.2.2, můžeme rozšíření do databáze nahrát pomocí příkazu *CREATE EXTENSION example;*.

3.7.2.1 Implementace SQL skriptu

Zdrojový kód 35 ukazuje možný SQL skript, který by tvořil obsah potřebného skriptovacího SQL souboru (např. *example--1.0.sql*). Na implementaci funkcí v této ukázce nezáleží, slouží zde jen pro demonstraci.

```
CREATE TYPE TypeExample AS (int id, int points);

CREATE FUNCTION functionOne() RETURNS void AS $$
-- functionOne implementation
$$ LANGUAGE SQL;

CREATE FUNCTION functionTwo() RETURNS INTEGER AS $$
-- functionTwo implementation
$$ LANGUAGE SQL;
```

Výpis 35: Ukázka SQL skriptu rozšíření pgSQL

3.7.2.2 Implementace kontrolního souboru

Obsahem jednoduchého ukázkové kontrolního souboru (*example.control*) pak může být následující konfigurace 36.

```
comment = 'PostgreSQL extension Test'
encoding = 'UTF-8'
default_version = '1.0'
superuser = true
relocatable = true
```

Výpis 36: Ukázka kontrolního souboru rozšíření pgSQL

3.7.3 Balíčky souvisejících objektů a SQL standard

Ačkoli standard umožňuje vytvářet funkce (například prostřednictvím jazyka Java), specifikace nedefinuje žádnou možnost vytváření balíčkování těchto funkcí či jiných „modulů“ jako jeden celek (extension).

3.8 Různé typy indexů

PostgreSQL poskytuje několik typů indexů. Přesně se jedná o typy B-strom, Hash, GiST, SP-GiST a GIN. Každý typ indexu používá specifický algoritmus hodící se k zrychlení různých typů dotazů. Výchozím typem příkazu CREATE INDEX je typ B-strom a to pro jeho vhodnost použití v širokém množství případů. [10]

3.8.1 Využití různých typů indexů

3.8.1.1 Indexy typu B-strom Použití B-stromů je zejména vhodné pro porovnávání a rozsahové dotazy nad seřaditelnými daty. B-strom indexy lze sice použít i pro získání seřazených dat, toto využití však není vždy rychlejší než jednoduchý sekvenční průchod. [10]

B-strom indexy umí pracovat s operátory (<, <=, =, >= a >) či s jejich SQL ekvivalenty, kterými jsou operátory BETWEEN, IN, IS NULL či IS NOT NULL.

3.8.1.2 Hash indexy Hashovací indexy slouží jen pro jednoduché zjištění rovnosti pomocí operátoru (=). Deklarace hashovacího indexu se provádí tímto způsobem: *CREATE INDEX nazev ON tabulka USING hash (sloupec);*.

3.8.1.3 GiST a SP-GiST indexy Index typu GiST v podstatě není jeden druh indexu, jelikož tvoří infrastrukturu, uvnitř které může být implementováno více indexovacích postupů. Z tohoto důvodu závisí použití konkrétních operátorů na zvoleném indexovacím postupu (třídě operátorů). Standardní distribuce PostgreSQL tak například zahrnuje GiST třídů operátorů pro několik vícerozměrových geometrických typů.

Indexy SP-GiST oproti GiST indexům zamezují implementaci širokého rozsahu různých nevyvážených datových struktur jako jsou k-d stromy, radix stromy či quad stromy (quadtrees).

3.8.1.4 GIN indexy GIN indexy mohou obsahovat více hodnot než jen jeden klíč. Jedná se tak například o pole (arrays). Podobně jako GiST a SP-GiST mohou GIN indexy implementovat různé uživatelsky definované strategie a využít tak příslušné operátory.

3.8.2 Indexy a SQL standard

Klíčové slovo INDEX není dodnes specifikované žádným ze standardů SQL. Jedná se tak o nestandardní funkcionalitu, kterou PostgreSQL ale i velké množství konkurečních DBMS využívá k urychlení zpracování dotazů.

3.9 Extra užitečné vlastnosti indexů

3.9.1 Indexy nad výrazy

Sloupec indexu nemusí být vždy sloupcem příslušné tabulky, může jej tvořit funkce či výraz pracující s jedním či více sloupci dané tabulky. Takovéto použití indexu je vhodné pro získaní rychlého přístupu k tabulkám založených na výsledcích výpočtů. [10] Indexované výrazy jsou relativně náročné z hlediska požadových prostředků a z důvodu nutnosti výpočtu pro každý nově vložený či aktualizovaný řádek. Nicméně výrazy těchto indexů nejsou přepočítávány během vyhledávání, jelikož výsledky těchto výpočtu jsou již uložené v rámci indexu. Z toho výplývá, že využití indexů nad výrazy je nejlepší v případě, kdy je získávání dat důležitější než jejich vkládání či aktualizování.

Velmi často se k porovnávání textové hodnoty sloupce využívá funkce *lower(sloupec)*. Příklad takového typického dotazu a příslušného indexu sloužícího ke zrychlení těchto dotazů demonstruje ukázka 37.

```
SELECT * FROM Student WHERE lower(name) = 'jack johnson';
CREATE INDEX StudentLowerNameIdx ON Student (lower(name));
```

Výpis 37: Ukázka indexu nad výrazem

3.9.2 Částečné (partial) indexy

Částečný index je index pracující s částí (subsetem) tabulky. Takovýto subset je definován takzvaným „predikátem“ částečného indexu. Částečný index tak zahrnuje pouze vybrané řádky tabulky, které určuje právě zmíněný predikát. Jedná se sice o velmi specializovanou vlastnost, existují však případy, kdy je jejich využití efektivní.

Jedním z hlavních důvodů použití částečného indexu je vyhnutí se indexaci běžných hodnot. Dotaz hledající běžnou hodnotu (takovou, která tvoří více než několik jednotek procent všech řádků tabulky) nevyužije index, protože neexistuje žádný důvod uchovávat tyto řádky v rámci indexů. Částečný index tak bude mnohem menší, než by byl ten úplný. Díky tomu dojde k nárustu rychlosti dotazů, jenž využívají tyto partial indexy. Zároveň dochází i ke zrychlení operací UPDATE, protože nebudou prováděny tak časté aktualizace indexu. [10]

Vytváření částečných indexů je velmi jednoduché. Využívá se podmínky WHERE, která vymezuje řádky, jenž chceme indexovat. Následující SQL skript 38 prezentuje vytvoření částečného indexu nad tabulkou *Student* 6. Index vymezuje rozsah hodnot sloupce *points*.

```
CREATE INDEX StudentPointsIdx ON Student (point)
WHERE (points > 30 AND
 client_ip  < 90);
```

Výpis 38: Ukázka částečného (partial) indexu

3.10 Pokročilé možností triggerů

Trigger specifikuje funkci, kterou databáze automaticky vykoná vždy, když se provede určená operace. Triggery mohou být navázány na operace tabulek či pohledů (views). Funkce triggerů se implementují prostřednictvím procedurálních jazyků či v jazyce C.

Spuštění triggerů tabulek lze definovat pro příkazy INSERT, UPDATE nebo DELETE a to buď před nebo po jejich vykonání. UPDATE trigger pak může být spuštěn i v případech úpravy jednotlivých sloupců (klauzule SET příkazu UPDATE). U triggerů pohledů, mohou být triggery navíc definovány tak, aby nahradily operace INSERT, UPDATE či DELETE a to díky speciálnímu druhu triggerů vyznačujícím se příkazem INSTEAD OF. Všechny potřebné úkony jsou pak přenechány funkci určeného triggeru. Triggery mohou být definovány i pro příkaz TRUNCATE (rychlé vyčištění tabulek).

Funkce triggeru musí být definována před samotným vytvořením triggeru. Funkce se deklaruje bez argumentů a návratovým typem je *trigger*. Vstupní data jsou funkci místo standardních argumentu předána pomocí speciální struktury *TriggerData*. Jedna trigger funkce může být navíc použita v rámci většího množství triggerů. Samotný trigger se pak vytváří pomocí příkazu CREATE TRIGGER definujícího trigger funkci, tabulku (pohled, sloupec) a určení spuštění před nebo po události.

Ukázku vytvoření jednoduchého triggeru pak může představovat následujcí SQL kód 39.

```
CREATE TRIGGER studentUpdate
BEFORE UPDATE ON Student
FOR EACH ROW
EXECUTE PROCEDURE studentUpdated();
```

Výpis 39: Ukázka jednoduchého triggeru

3.10.1 Triggery řádku a příkazu

PostgreSQL umožňuje definovat triggery pro řádek (per-row) a pro příkaz (per-statement). Občas se lze také setkat s označením row-level a statement-level. Rozdíl triggeru pro řádek a triggeru pro příkaz se při vytváření v SQL určí pomocí specifikace FOR EACH ROW nebo FOR EACH STATEMENT.

3.10.1.1 Triggery řádku (per-row) Z hlediska per-row triggeru je funkce zavolána pro každý řádek, který je ovlivněn příkazem spouštějící daný trigger. V druhém případě (per-statement) je trigger spuštěn pouze jednou, nehledě na to, kolik řádků bylo daným příkazem ovlivněno. Pohledové (view) triggery implementované jako INSTEAD OF pro operace INSERT, UPDATE či DELETE lze definovat pouze jako per-row.

3.10.1.2 Triggery příkazu (per-statement) Per-statement trigger je spuštěn pouze tehdy, když je příslušný příkaz vykonán. Nehledí se přitom na to, kolik řádku je tímto příkazem ovlivněno. Trigger bude tedy spuštěn i v případě, že daný příkaz neovlivní žádný řádek. Triggery příkazu TRUNCATE mohou být definovány pouze jako trigger

per-statement stejně jako triggery pohledů, které se spouští před nebo po operaci nad daným pohledem. Funkce těchto triggerů by vždy měly vracet *NULL*.

3.10.2 Nastavení BEFORE, AFTER, INSTEAD OF

Jak již bylo v úvodu této podkapitoly zmíněno, spuštění triggerů lze nastavit jako BEFORE (před), AFTER (po) či INSTEAD OF (místo). Per-statement triggery 3.10.1.2 nastavené jako BEFORE se spustí před jakoukoliv manipulací daného příkazu, zatímco AFTER triggery spustí funkci až na samém konci příkazu. Per-row BEFORE triggery 3.10.1.1 spustí funkci řádků dříve než je operace řádku provedena, zatímco per-row AFTER trigger provede spuštění na konci příkazu (avšak dříve než jakýkoliv per-statement AFTER trigger). Tyto typy triggerů mohou být definovány pouze pro tabulky či pohledy (triggery per-row INSTEAD OF pak pouze v rámci pohledů). [11]

3.10.3 Triggery a SQL standard

Triggery jsou standardizovány již delší dobu. V nejnovější verzi specifikace standardu SQL:2011 byla navíc přidána možnost INSTEAD OF (pro view triggery). Nicméně definice triggerů v jazyce C či definice jako per-row či per-statement není dle standardu možná.

3.11 Pokročilé fulltextové vyhledávání

Fulltextové vyhledávání (či prostě textové vyhledávání) poskytuje možnost identifikovat dokumenty přirozeného jazyka, které vyhovují dotazu, a také možnost řadit je podle relevance. Nejčastěji se provádí hledání všech dokumentů obsahujících požadavky daného dotazu a jejich návrat v pořadí dle podobnosti požadavku. Podoba dotazu je velmi flexibilní a závisí na specifické aplikaci. Nejjednodušší typ vyhledávání předpokládá dotaz v podobě několika klíčových slov. Následně jsou tato klíčová slova (ale navíc i slova jim podobná) identifikována v rámci určených dokumentů a výsledkem pak jsou dokumenty s největší frekvencí výskytu těchto slov.

3.11.1 Existující textové operátory databáze

Textové operátory hledání již existují v databázích roky. PostgreSQL vlastní operátory textových datových typů jako ~, ~*, LIKE či ILIKE, nicméně tyto operátory prostrádají mnoho základních vlastností, které jsou vyžadované v moderních informačních systémech. [12] Neexistuje tak žádná jazyková podpora, která by zajistila rozpoznání odvozených slov ze slov klíčových. Neexistuje také žádný systém pro seřazení výsledků dle relevance. V případě stovky a více dokumentů to tedy představuje významný problém. A z důvodu chybějící indexace je navíc tento postup velice pomalý a to především proto, že je nutné při každém volání procházet všechny dokumenty znova.

3.11.2 Indexace fulltextového vyhledávání

Fulltextová indexace přináší předzpracování dokumentů. Díky ní lze zajistit rychlé výsledky vyhledávání.

Prvním krokem zpracování vstupního dokumentu je parsování do tokenů. Parser tak zpracuje jednotlivé elementy do takzvaných tokenových tříd (slova, čísla, e-mailové adresy). Definice jednotlivých tokenových tříd závisí na konkrétní aplikaci. Pro většinu případů je však doporučeno využít předefinované třídy. Parser a definici těchto tříd však lze uživatelsky modifikovat.

V dalším kroku jsou tokeny tvořící slova z hlediska příslušného jazyka (například angličtiny) normalizovány. Normalizace zahrnuje registraci příbuzných a odvozených slov. Prakticky se jedná o přidání a odebrání různých předpon a přípon slov či ignorace velkých a malých písmen. Ve výsledku je tak hledané klíčové slovo reprezentováno velkou četností odvozenin, což zajistí hlubší hledání z hlediska jednoduchého „porozumění“ dotazu. Výsledkem normalizace vstupního tokenu je pak takzvaný lexeme. Posledním krokem je uložení těchto lexemů v polích (arrays). Dále se ukládají také dodatečné informace sloužící k určení přesnější relevence výsledků dotazů.

3.11.3 Dokument pgSQL

Obsah typického dokumentu může tvořit například dopis, zpráva, článek nebo e-mail. Jádro textového vyhledávání provede zparsování a uloží asociace klíčových slov. Později jsou tyto asociace využity pro hledání.

Při hledání v databázi pgSQL je dokument standardně tvořen textovým polem řádku tabulky či případnou kombinací více textových polí. Jinými slovy, dokument může být vytvořen z více různých „částí“. Není tedy nutné ukládat dokumenty jako celek. Konstrukce jednoduchého dokumentu ze sloupců tabulky se provádí způsobem 40. Jak lze vidět, stačí provést jednoduché spojení textových řetězců (sloupců) a označit výběr jako dokument (AS *document*). PostgreSQL umožňuje dokumenty ukládat i jako jednoduché textové soubory v rámci souborového systému. Pro získávání těchto souborů ze zdroje, jenž se nachází mimo databázi, je však třeba práv administrátora či využít specializovaných funkcí. Je tak tedy vhodnější uchovávat tyto soubory uvnitř PostgreSQL.

```
SELECT name || '␣' ||  surname || '␣'  ||  description  AS document
FROM User
WHERE id = 123;
```

Výpis 40: Ukázka dokumentu pgSQL

Pro použití fulltextového vyhledávání musí být každý dokument uložen v rámci formátu *tsvector*. Vyhledávání a určení relevance je vykonáváno čistě nad typem *tsvector* reprezentujícím dokument. Originální text je získán pouze tehdy, kdy je potřebné dokument uživateli zobrazit. [12]

3.11.4 Ukázka jednoduchého fulltextového vyhledávání

Fulltextové vyhledávání se v PostgreSQL provádí pomocí operátoru (@@), který vrací *true* v případě, že dokument (*tsvector*) odpovídá dotazu, jenž reprezentuje datový typ *tsquery*. Následující výčet ukazuje s jakými typy (a jejich případným pořadím) dokáže operátor (@@) pracovat. Praktické využití všech těchto možností demonstruje SQL kód 41.

- *tsvector* @@ *tsquery*
- *tsquery* @@ *tsvector*
- *text* @@ *tsquery*
- *text* @@ *text*

Všechny SELECT příkazy prakticky představují jeden a ten samý dotaz vracející *true*. Povšimněte si, že v rámci dotazu (*tsquery*) je implementován operátor AND (&). V dotazech je možné použít jakýkoliv z operátoru AND, OR či NOT. Třetí dotaz ukazuje možnost záměny pořadí dotazu a dokumentu. Dále je dobré si všimnout, že máme možnost vytvářet datové typy dvěma způsoby a to pomocí funkcí *to_tsvector* a *to_tsquery* či převedením *::tsvector* a *::tsquery*. Jak ale ukazuje poslední dotaz SQL kódu, převedení není vždy nutné, jelikož PostgreSQL umí převádět příslušné hodnoty na požadované typy implicitně.

```
SELECT 'Hello my name is Jack Johnson.'::tsvector @@ 'name & Jack'::tsquery;

SELECT to_tsvector('Hello my name is Jack Johnson.') @@ to_tsquery('name & Jack');

SELECT 'name & Jack'::tsquery @@ 'Hello my name is Jack Johnson.'::tsvector;

SELECT 'Hello my name is Jack Johnson.' @@ to_tsquery('name & Jack');

SELECT 'Hello my name is Jack Johnson.' @@ 'name & Jack';
```

Výpis 41: Ukázka použití fulltextového vyhledávání

3.11.5 Fulltextové vyhledávání a SQL standard

Fulltextového vyhledávání není SQL standardem specifikováno.

4 Implementace C rozšíření PostgreSQL

Jako implementaci vlastního rozšíření jsem zvolil funkci jazyka C. C funkce databáze, kterým se důkladně věnuje podkapitola 3.4, totiž přináší úplně odlišný pohled na programování databázové logiky. Je tak možné využít obrovského množství vestavěných i externích knihoven a implementovat tak komplexní úlohy, které by bylo velmi obtížné či zcela nemožné řešit prostřednictvím SQL funkcí 3.1 či funkcí procedurálního jazyka 3.2.

4.1 Popis vlastního C rozšíření

Vlastní funkce provádí validaci vstupního XML řetězce či XML souboru, jehož obsah reprezentuje schéma databáze. Struktura XML tak představuje jednotlivé tabulky, jejich sloupce s příslušnými datovými typy a vazby mezi tabulkami, to znamená cizí a primární klíče. Samotná validace XML vstupu (schéma databáze) probíhá vůči XSD schématu, které je zabudované uvnitř C funkce. XSD schéma má přesně definovanou strukturu, která určuje podobu XML elementů a jejich atributů, jenž reprezentují tabulky, sloupce, jejich datové typy a klíče.

4.2 Jednotlivé kroky implementace

Tato podkapitola chronologicky popisuje průběh vytváření uživatelského C rozšíření PostgreSQL.

4.2.1 Předpis struktury XML (XSD)

Prvním krokem byla tvorba XSD schématu určujícího strukturu vstupních XML souborů představujících schémata databází. XSD určuje strukturu XML elementů z hlediska datových typů sloupců tabulek. Možnosti použitelných datových typů odpovídá seznamu typů, které aktuální verze PostgreSQL nabízí. Sloupce tabulek s neodpovídajícími datovými typy tak neprojdou validací. V případě datových typů s parametry je také nutno uvést hodnotu těchto parametrů, jedná se tak například o *scale* či o *precision* u typu *decimal*. Implementaci XML schématu (XSD soubor) lze prozkoumat v rámci přílohy práce (soubor *xmlschema.xsd*).

4.2.2 Vstupní XML soubor

Po implementaci předpisu (XSD) určujícího strukturu XML souboru, který reprezentuje tabulky databáze, jsem pro testování jeden takový vyhovující XML soubor vytvořil. Jak tento validní XML soubor vypadá, lze pozorovat v přiloženém souboru *db1.xml*. Vytvoření jedné tabulky můžeme vidět na následující krátké ukázce 42. Zde je vidět, že struktura tabulek v rámci XML je velmi snadno pochopitelná. Tabulka obsahuje dva sloupce *name* a *surname* typu *text* a primární klíč složený z těchto sloupců.

```
<table name="table2">
    <columns>
        <column name="name">
            <text/>
        </column>
        <column name="surname">
            <text/>
        </column>
    </columns>
    <primaryKey>
        <key name="PK_table2" clustered="true">
            <column name="name"/>
            <column name="surname"/>
        </key>
    </primaryKey>
    <uniqueConstraints>
    </uniqueConstraints>
    <relationships>
    </relationships>
</table>
```

Výpis 42: Ukázka validní tabulky v rámci vstupního XML souboru

4.2.3 Validace XML vůči XSD

Po přípravě vstupních dat byla na řadě implemetace algoritmu v jazyce C. Ten slouží k validaci XML souboru z hlediska předpisu XSD. Pro tento úkol jsem využil existujících knihoven *libxml2, libiconv* a *zlib*. Tyto knihovny totiž pro práci s XML a XSD poskytují velmi užitečné struktury a funkce.

Celou validaci prakticky zajišťuje funkce *int validateDocAgainstSchema(char xmlFileName[], char *xmlFromMemory)* vyskutující se v řešení (VS), které je taktéž doloženo v příloze. Prvně bylo v rámci kódu této funkce potřeba vytvořit strukturu *xmlDocPtr doc* pro vstupní XML soubor nebo řetězec, strukturu *xmlSchemaPtr schema* k uchování XSD schématu a také XSD parser *xmlSchemaParserCtxtPtr schemaParser*.

Zdrojový kód 43 ukazuje hlavní část validační funkce. První 4 řádky ukázky nejprve provedou kontrolu správného formátu XSD schématu, které je binárně „zadrátováno“ v rámci hlavičky *xsd.h* řetězecem *char[] xsd* obsahující binární reprezentaci výše zmíněného XSD souboru. V případě výskytu chyby je tato informace předána do standardního error streamu *stderr*. Pro převedení textového XSD souboru na binární řetězec jsem využil UNIXové utility xxd.

Následující rozhodovací blok určí metodu naplnění struktury *xmlDocPtr doc*. Funkci *int validateDocAgainstSchema(char xmlFileName[], char *xmlFromMemory)* lze totiž předat XML vstup jak v podobě kompletního XML souboru tak i v podobě textového řetězce.

Za tímto rozhodovacím blokem se nachází další blok, který testuje prázdnost (či špatné naplnění) struktury *xmlDocPtr doc*. V neúspěšném případě funkce vrátí hodnotu -2 indikující neúspěch funkce z hlediska parsování. Chyba je opět zaznamenána do streamu *stderr*. V opačném případě funkce pokračuje validační částí. Funkce tak může v této fázi

skončit třemi různými způsoby - úspěšnou validací (1), neúspěšnou validací (0) či interní validační chybou (-1). Výsledky jsou v rámci streamu *stderr* opět záznamenány. Kompletní řešení implementace i s komentáři je součástí přílohy práce.

```
...

schemaParser = xmlSchemaNewMemParserCtxt((char*)xsd, xsd_len);

xmlSchemaSetParserErrors(schemaParser,
    (xmlSchemaValidityErrorFunc)fprintf, (xmlSchemaValidityWarningFunc)fprintf, stderr);

schema = xmlSchemaParse(schemaParser);
xmlSchemaFreeParserCtxt(schemaParser);

if (xmlFromMemory == NULL){
    doc = xmlReadFile(xmlFileName, NULL, 0);
}
else
{
    doc = xmlReadMemory(xmlFromMemory, length(xmlFromMemory), NULL, NULL, 0);
}

if (doc == NULL)
{
     fprintf (stderr, "Could not parse.\n");
    freeTheResource(schema);
    return -2;
}
else
{
    xmlSchemaValidCtxtPtr ctxt;
    int ret;
    ctxt = xmlSchemaNewValidCtxt(schema);

    xmlSchemaSetValidErrors(ctxt, (xmlSchemaValidityErrorFunc)fprintf,
        (xmlSchemaValidityWarningFunc)fprintf, stderr);

    ret = xmlSchemaValidateDoc(ctxt, doc);
    xmlSchemaFreeValidCtxt(ctxt);
    xmlFreeDoc(doc);

     ...
}
```

Výpis 43: Ukázka implementace validace XML

4.2.4 Funkce napojení na PostgreSQL

Další stěžejní částí byla úprava nastavení vývojového prostředí (v mém případě: VS) a to tak, aby bylo výslednou sdílenou knihovnu možno kompilovat a aby byla kompatibilní s databází PostgreSQL.

Nastavení prostředí zahrnovalo následující kroky:

- Nastavení výstupu programu => dynamická knihovna (.dll)
- Změna direktivy preprocesoru (WIN32 => WINDLL)
- Deaktivace C++ vyjímek.
- Nastavení kompilace kódu jako C (ne jako C++).
- Nastavení závislostí PostgreSQL. (hledání hlaviček v podadresářích instalace databáze PostgreSQL)

Na zdrojovém kódu 44 můžeme vidět hlavní funkci *validateSchemaFile()*, která se při zavolání příslušného SQL dotazu spustí. Tato C funkce psaná dle konvence verze 1 3.4.4 přijímá jako první parametr *text* řetězec určující cestu XML souboru. Druhý *boolean* parametr pak určuje spuštění přesměrování chybového logu do interního datového uložiště databáze PostgreSQL. Následně se (ať už s nebo bez přesměrování *stderr* streamu) volá výše popsaná funkce 4.2.3, která provádí XML validaci. Číselný výsledek indikující úspěšnost či neúspěšnost validace je poté databázi vrácen pomocí návratové funkce PG_RETURN_INT32.

```
PGDLLEXPORT
Datum
validateSchemaFile(PG_FUNCTION_ARGS)
{
    bool redirectOutput = PG_GETARG_BOOL(1);

    int32  validation ;

    char* schemafilePath = text_to_cstring(PG_GETARG_TEXT_P(0));

    if  (redirectOutput == true)
    {
        redirectOutputAndErrors(VALIDATIONLOGPATH);

        validation  = validateDocAgainstSchema(schemafilePath, NULL);

        revertRedirectOfOutputAndErrors();
    }
    else
    {
        validation  = validateDocAgainstSchema(schemafilePath, NULL);
    }

    PG_RETURN_INT32(validation);
}
```

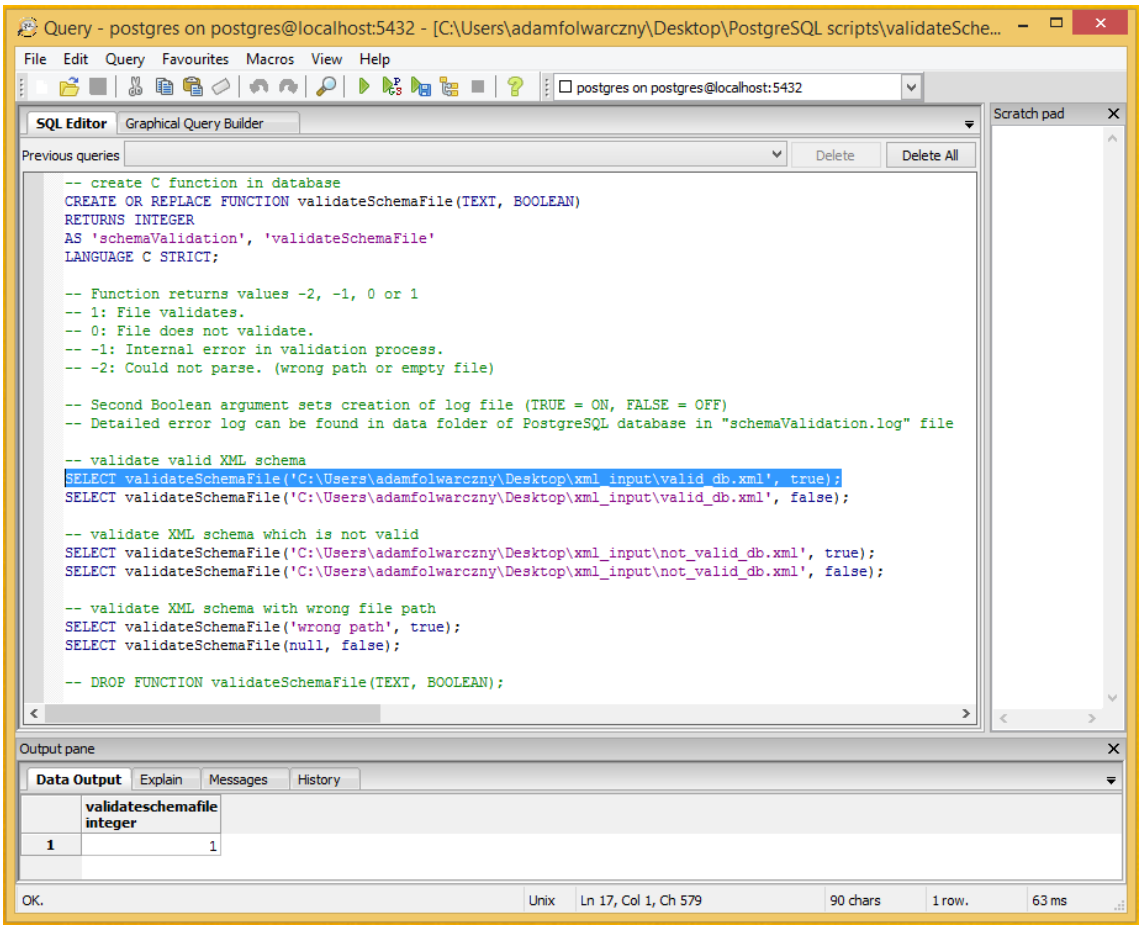
Výpis 44: Implementace hlavní části vlastní PostgreSQL C funkce

Přiložené řešení obsahuje i druhou funkci *validateSchemaString*, která jako vstup bere XML řetězec místo adresy souboru. Implementace je však téměř totožná. Jediný rozdíl spočívá v předání XML řetězce funkci *validateDocAgainstSchema* jako druhý parametr s tím, že za první parametr, jenž určuje cestu k souboru XML, se dosadí hodnota *NULL*.

4.2.5 Vytvoření a volání funkcí pomocí SQL dotazů

Po zavedení vlastní sdílené knihovny 3.4.1.1 s C funkcemi do databáze, jsem provedl registraci prostřednictvím příslušných SQL dotazů (v mém případě skrze správce pgAdmin III). Deklaracím C funkcí z hlediska SQL dotazů se již práce věnovala v části 3.4.3.1.

Na obrázku 1 lze pozorovat deklaraci, volání a funkční výsledek jednoho z nich přímo ve správci pgAdmin III pracujícím nad databází, do které bylo C rozšíření instalováno. Jak je z obrázku vidět, implementovaná PostgreSQL C funkce vrací anonymní řádek s hodnotou indikující úspěšnost či neúspěšnost porovnání XML schématu.



Obrázek 1: Zavolání a výsledek vlastní PostgreSQL C funkce v prostředí pgAdmin III

4.2.6 Dodatek k přiložené implementaci

V kořenovém adresáři přiloženého řešení se nachází textový soubor *guide.txt* obsahující popis struktury složek, přiložených souborů, knihoven a také pokyny pro možnost vyzkoušení kompilace výsledné sdílené knihovny v rámci řešení VS. Samotnou funkčnost rozšíření, tedy vytvoření a spuštění validace je možno provést skrze správce pgAdmin III nad přiloženou přenositelnou (portable) pgSQL databází, ve které je již mé řešené uživatelské rozšření nainstalované.

5 Srovnání rozšířitelnosti PostgreSQL s konkurencí

Mezi nejpopulárnější open-source SŘBD patří mimo PostgreSQL především databázové systémy MySQL či SQlite. MySQL jako nejpoužívanější databázový open-source systém však z hlediska uživatelské rozšířitelnosti rozhodně nenabízí tolik možností jako právě PostgreSQL. V případě server-less DBMS SQlite je pak oficiálně podporovaná uživatelská rozšířitelnost téměr nulová.

5.1 Rozšířitelnost MySQL

I když není databázový systém MySQL rozšířitelný jako PostgreSQL, nabízí i tak pár zajímavých možností rozšíření. Patří mezi ně například možnost psát serverovou logiku pomocí uživatelsky vytvořených SQL procedur, funkcí či UDF implenetovaných v jazyce C/C++. MySQL nenabízí žádnou alternativu či obdobu procedurální jazyků nachazejících se v PostgreSQL. Zajímavou možností rozšíření MySQL serveru je pak také nabídka rozhraní v podobě Plugin API.

5.1.1 Uložené funkce a procedury jazyka MySQL

Možnosti funkcí a procedur MySQL odpovídají rozsahem možnostem SQL funkcí PostgreSQL 3.1. PostgreSQL sice nenabízí definování procedur, procedury jsou však prakticky funkce nemající návratovou hodnotu. Ekvivalencemi procedur v rámci PostgreSQL jsou tedy funkce vracející *void*. MySQL navíc, jak již bylo zmíněno, neumožňuje v těchto funkcích využít vlastnosti, které poskytují procedurální jazyky pgSQL 3.2.

5.1.2 MySQL Plugin API

MySQL poskytuje API pro doplňky, které umožňuje vytvářet serverové komponenty. Doplňky mohou být nahrány jak při startu serveru, tak i za jejich běhu. API je obecné a nelimituje možnosti, které doplňky mohou a nemohou provádět. Mezi podporované možnosti API tak kupříkladu patří storage-engines, fulltextové parsery a mnoho dalších. V praxi může uživatelský fulltextový parser nahradit ten vestavěný a pracovat tak s textem přesně podle uživatelské implementace. Modul rozhraní je obecnější než starší rozhraní uživatelem definovaných funkcí (UDF) 5.1.3. [13]

5.1.3 Uživatelské funkce definované pomocí rozhraní UDF

MySQL umožnuje uživateli přidat funkce skrze rozhraní UDF (user-defined function). Tyto uživatelsky vytvořitelné funkce se kompilují jako objektové soubory a následně jsou dynamicky přidány či odebrány ze serveru pomocí příkazů CREATE FUNCTION a DROP FUNCTION. [13] Zmíněná funkcionalita je tedy velmi podobná funkcím jazyka C v pgSQL 3.4. Takto vytvořené funkce lze následně vyvolat stejným způsobem jako již vestavěné funkce (například *ABS()* či *SQRT()*).

5.2 Rozšířitelnost SQlite

Z důvodu server-less povahy DBMS SQlite je rozšířitelnost tohoto systému značně omezená. Jedná se tak především o embedded databázi, která nebyla přímo vyvinuta jako typická klient/server databáze. Databáze tak bývá nejčastěji součástí lokálního aplikačního filesystému. Ve své podstatě se však ale stále jedná o typickou relační SQL databázi. Nicméně i systém SQlite lze použít a je v poslední době často používán jako datové uložiště serverově orientovaných aplikací.

Vývojáři hlásí, že použití SQlite v podobě klient/server SQL databaze nabízí větší rychlost než využití konkurenčních řešení. [14]

5.2.1 SQLite C/C++ Interface

SQlite i přes svou jednoduchost obsahuje rozhraní pro implementaci uživatelských rozšíření skrze jazyk C a C++.

Databázový systém zahrnuje 4 základní metody rozšíření:

- Rozhraní **sqlite3_create_collation()** se využívá k vytváření řadicích sekvencí (collaction sequences) sloužících k seřazení textů.
- Rozhraní **sqlite3_create_function()** nabízí metody vytváření a úprav vlastních SQL funkcí či agregátů.
- Rozhraní **sqlite3_create_module()** umožňuje uživateli vytvářet takzvané virtuální tabulky (tabulky spárované a viditelné pouze pro určitá připojení)
- Rozhraní **sqlite3_vfs_register()** slouží k manipulaci nejspodnější virtuální vrstvy systému SQlite. VFS zajišťuje komunikaci mezi SQlite a příslušným operačním systémem. Toto rozhraní umožňuje vytvářet a spravovat tyto virtuální VFS vrstvy, čímž lze následně zajistit přenositelnost databáze mezi nejrůznějšími operačními systémy.

6 Závěr

Bakalářská práce měla za úkol především prozkoumat a zdokumentovat nadstardní vlastnosti a možnosti rozšíření relačního databázového systému PostgreSQL. Dalšími cíli pak bylo uskutečnit implementaci jednoho praktického rozšíření a také srovnat možnosti rozšíření s dvěma jinými a populárními DBMS.

V rámci úvodní části jsem vytýčil možnosti uživatelských rozšíření PostgreSQL, fungovaní datových typů databáze a také popsal strukturu SQL standardu, kterému se databáze PostgreSQL snaží vyhovět.

V dokumentační části jsem se pak snažil vybrat a následně do detailu popsat obtížně nahraditelné vlastnosti PostgreSQL z hlediska standardu SQL. Pro dokumentaci jsem zvolil funkce dotazovacího jazyka SQL, funkce procedurálních jazyků, interní funkce, uživatelské datové typy, balíčky souvisejících objektů tvořící rozšíření (extensions), nadstandardní typy indexů, jedinečné vlastnosti indexů, pokročilé definice triggerů, fulltextové vyhledávání ale především funkce jazyka C. A to z toho důvodu, že ne každý databázový systém (obzvlášť pak open-source) takové či podobné možnosti nabízí.

Cílem navazující části byla implementace zvoleného rozšíření v podobě C funkce. Ta demonstrovala možnosti využití externích knihoven či práci se soubory. Vlastní C doplněk provádí validaci možných XML schémat reprezentujících databázi a to podle přesně definovaného předpisu ve formě XML schématu (XSD). Doplněk tak může sloužit jako základní „stavební kámen“ pro dodatečné dokončení doplňku, kdy by mohla být databáze dle validního schématu vytvořena. Ačkoli implementace nízkoúrovňového C rozšíření skrývá mnohá úskalí hlavně z hlediska správné a kompatibilní kompilace výstupní sdílené knihovny, podařilo se jednoduchý a funkční doplněk vytvořit.

Po srovnání se dvěma nejpoužívanějšími konkurečními open-source DBMS (MySQL a SQlite) jsem nakonec došel k závěru, že PostgreSQL je vůči této konkurenci z hlediska uživatelské benevolence a rozšířitelnosti jednoznačně nejdále.

Adam Folwarczny

7 Reference

[1] Momjian, Bruce, *PostgreSQL: Introduction and Concepts*, Computer Press. 2003.

[2] Molinaro, Anthony, *SQL Cookbook*, O'Reilly Media. 2005.

[3] *ISO/IEC 9075-1:2011*, ISO Standards. 2011.

[4] *PostgreSQL 9.4.1 Documentation: SQL Conformance*,
http://www.postgresql.org/docs/9.4/interactive/features.html

[5] *PostgreSQL 9.4.1 Documentation: Extending SQL*,
http://www.postgresql.org/docs/9.4/static/extend.html

[6] *PostgreSQL 9.4.1 Documentation: Packaging Related Objects into an Extension*,
http://www.postgresql.org/docs/9.4/static/extend-extensions.html

[7] *PostgreSQL 9.4.1 Documentation: Collation*,
http://www.postgresql.org/docs/9.4/interactive/collation.html

[8] *PostgreSQL 9.4.1 Documentation: Procedural Languages*,
http://www.postgresql.org/docs/9.4/interactive/xplang.html

[9] *PostgreSQL 9.4.1 Documentation: PL/pgSQL - SQL Procedural Language*,
http://www.postgresql.org/docs/9.4/interactive/plpgsql.html

[10] *PostgreSQL 9.4.1 Documentation: Indexes*,
http://www.postgresql.org/docs/9.4/static/indexes.html

[11] *PostgreSQL 9.4.1 Documentation: Triggers*,
http://www.postgresql.org/docs/9.4/static/triggers.html

[12] *PostgreSQL 9.4.1 Documentation: Full Text Search*,
http://www.postgresql.org/docs/9.4/static/textsearch.html

[13] *MySQL 5.6 Reference Manual: Extending MySQL*,
http://dev.mysql.com/doc/refman/5.6/en/extending-mysql.html

[14] *SQlite Documentation: Categorical Index Of SQLite Documents*,
http://www.sqlite.org/docs.html

8 Použitý software

Tato sekce obsahuje výčet softwaru, který byl při tvorbě této práce použit.

- SŘBD PostgreSQL 9.4.1 (x86)
- SŘBD PostgreSQL - Portable 9.4.1 (x86)
- pgAdmin III 1.2.0 (x86)
- Microsoft Visual Studio 2013 Ultimate with Update 4 (x86)
- Microsoft Windows Professional 8.1 (x86)
- Apple OS X 10.10 Yosemite
- Apple Xcode 6.1
- xxd (UNIX Utility)
- MacTeX 2014
- TeXstudio 2.9.4

9 Přílohy

Přiložené CD

- **/lib/** - potřebné C/C++ knihovny.
- **/PostgreSQL scripts/** - SQL skripty k vytvoření a otestování C funkce
- **/PostgreSQLPortable-9.4/** - přenositelná 32-bitová databáze PostgreSQL 9.4.1
- **/schemaValidation/** - řešení rozšíření v prostředí Microsoft Visual Studio 2013
- **/xml_input/** - vstupní testovací XML data (+ XSD schéma)
- **guide.txt** - soubor obsahující popis pro testování