Pokročilé techniky v PostgreSQL

Advanced PostgreSQL techniques

Tomáš Kubát

Bakalářská práce
2009

Univerzita Tomáše Bati ve Zlíně
Fakulta aplikované informatiky

Univerzita Tomáše Bati ve Zlíně
Fakulta aplikované informatiky
Ústav aplikované informatiky
akademický rok: 2008/2009

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

(PROJEKTU, UMĚLECKÉHO DÍLA, UMĚLECKÉHO VÝKONU)

Jméno a příjmení:	**Tomáš KUBÁT**
Studijní program:	**B 3902 Inženýrská informatika**
Studijní obor:	**Informační technologie**
Téma práce:	**Pokročilé techniky v PostgreSQL**

Zásady pro vypracování:

1. **Rešerše pokročilých metod: stored procedury, kurzory, triggery, transakce, zámky, datový typ pole, administrace oprávnění v databázovém systému PostgreSQL.**
2. **Vytváření vlastních funkcí a datových typů v databázovém systému PostgreSQL za použití jazyka C.**
3. **Využití programování na straně databázového serveru PostgreSQL v jazyku PL/pgSQL (PL/pgPSM).**
4. **Vytvoření ukázkové aplikace s důrazem na návrh databáze a přenesení složitých požadavků na stranu databázového serveru PostgreSQL.**

Rozsah práce:

Rozsah příloh:

Forma zpracování bakalářské práce: **tištěná/elektronická**

Seznam odborné literatury:

1. MOMJIAN, Bruce. PostgreSQL Praktický průvodce. 1. vyd. Brno: Computer Press, 2003. ISBN 80-7226-954-2.
2. WELLING, Luke - THOMSON, Laura. PHP a MYSQL rozvoj webových aplikací. 1. vyd. Praha: SoftPress, 2002. ISBN 80-86497-20-8.
3. SCHLOSSNAGLE, George. Pokročilé programování v PHP5. 1. vyd. Brno: Zoner software, 2004. ISBN 80-86815-14-5.
4. LIBERTY, Jesse. Naučte se C++ za 21 dní. 2. vyd. Brno: Computer Press, 2007. ISBN 978-80-251-1583-1.
5. PostgreSQL 8.3 Documentation [dokumentace online]. The PostgreSQL Global Development Group, 1996-- . [cit. 2009-01-22]. Dostupné z URL http://www.postgresql.org/docs/8.3/.
6. PostgreSQL [seriál online]. Praha: Pavel Stěhule, 2008-- . [cit. 2009-01-22]. Dostupné z URL http://www.pgsql.cz.
7. PostgreSQL [seriál online]. Praha: Linuxsoft-Marek Olšavský, 2004. [cit. 2009-01-22]. Dostupné z URL http://www.linuxsoft.cz/article.php?id_article=304. ISSN 1801-3805.

Vedoucí bakalářské práce: **doc. Ing. Zdenka Prokopová, CSc.**
Ústav aplikované informatiky

Datum zadání bakalářské práce: **20. února 2009**

Termín odevzdání bakalářské práce: **1. června 2009**

Ve Zlíně dne 13. února 2009

UNIVERZITA TOMÁŠE BATI VE ZLÍNĚ Fakulta aplikované informatiky L.S.

prof. Ing. Vladimír Vašek, CSc.
děkan

doc. Ing. Ivan Zelinka, Ph.D.
ředitel ústavu

ABSTRAKT

Bakalářská práce má sloužit všem zájemcům, kteří již pracují s databázovými systémy a nevyužívají jejich potenciálu naplno. V dokumentu se čtenář dozví, jak použít řadu pokročilých metod, s kterými se doposud nemusel setkat. Bude schopen je v návrzích vlastních databází použít s využitím databázového systému PostgreSQL.

Klíčová slova: cluster, databáze, schéma, role, spouště, procedury, funkce

ABSTRACT

This bachelor thesis aims to serve for all those interested who have already worked with database systems and who do not achieve their real potential. In this thesis readers will learn about many ways how to use various methods which could appear new to them. They will be able to use these methods in their own drafts of database systems when using PostgreSQL database system.

Keywords: cluster, database, schema, roles, triggers, procedures, functions

Na tomto místě bych rád poděkoval vedoucí bakalářské práce, doc. Ing. Zdence Prokopové, CSc. za ochotu, cenné rady a pomoc, kterou mi s maximální mírou flexibility věnovala při řešení této práce.

Motto:

„Dobře nakonfigurovaná a dobře provozovaná databáze se Vám odvděčí perfektním, stabilním a spolehlivým výkonem.“

Pavel Stěhule

Prohlašuji, že

- beru na vědomí, že odevzdáním bakalářské práce souhlasím se zveřejněním své práce podle zákona č. 111/1998 Sb. o vysokých školách a o změně a doplnění dalších zákonů (zákon o vysokých školách), ve znění pozdějších právních předpisů, bez ohledu na výsledek obhajoby;
- beru na vědomí, že bakalářská práce bude uložena v elektronické podobě v univerzitním informačním systému dostupná k prezenčnímu nahlédnutí, že jeden výtisk bakalářské práce bude uložen v příruční knihovně Fakulty aplikované informatiky Univerzity Tomáše Bati ve Zlíně a jeden výtisk bude uložen u vedoucího práce;
- byl/a jsem seznámen/a s tím, že na moji bakalářskou práci se plně vztahuje zákon č. 121/2000 Sb. o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon) ve znění pozdějších právních předpisů, zejm. § 35 odst. 3;
- beru na vědomí, že podle § 60 odst. 1 autorského zákona má UTB ve Zlíně právo na uzavření licenční smlouvy o užití školního díla v rozsahu § 12 odst. 4 autorského zákona;
- beru na vědomí, že podle § 60 odst. 2 a 3 autorského zákona mohu užít své dílo – bakalářskou práci nebo poskytnout licenci k jejímu využití jen s předchozím písemným souhlasem Univerzity Tomáše Bati ve Zlíně, která je oprávněna v takovém případě ode mne požadovat přiměřený příspěvek na úhradu nákladů, které byly Univerzitou Tomáše Bati ve Zlíně na vytvoření díla vynaloženy (až do jejich skutečné výše);
- beru na vědomí, že pokud bylo k vypracování bakalářské práce využito softwaru poskytnutého Univerzitou Tomáše Bati ve Zlíně nebo jinými subjekty pouze ke studijním a výzkumným účelům (tedy pouze k nekomerčnímu využití), nelze výsledky bakalářské práce využít ke komerčním účelům;
- beru na vědomí, že pokud je výstupem bakalářské práce jakýkoliv softwarový produkt, považují se za součást práce rovněž i zdrojové kódy, popř. soubory, ze kterých se projekt skládá. Neodevzdání této součásti může být důvodem k neobhájení práce.

Prohlašuji,

že jsem na bakalářské práci pracoval samostatně a použitou literaturu jsem citoval. V případě publikace výsledků budu uveden jako spoluautor.

Ve Zlíně …….

podpis diplomanta

OBSAH

ÚVOD

Řada programátorů vyvíjející databázové aplikace si ve svých počátcích vystačí se základní funkčností databázového systému (DBS) a zřídka kdy využijí funkcionalitu nabízenou databázovým systémem, který používají.

Často také s rostoucími nároky na databázové aplikace přestává programátorem aktuálně používaný DBS dostačovat a to jak z pohledu své funkcionality, z pohledu výkonu, tak z hlediska požadované spolehlivosti. V takovém případě musí programátor přistoupit buď ke kompromisu se stávajícím řešením, anebo migrovat na jiný systém, který lépe pokryje jeho potřeby. V této práci byl jako nový databázový systém vybrán databázový server PostgreSQL (PgSQL), jež je hojně využíván v podnikovém prostředí a to zejména díky těmto vlastnostem:

- je poskytován pod licencí open-source
- velmi vysoká spolehlivost a stabilita
- vysoký výkon při zpracování několika konkurujících úloh (při nízkém zatížení a zpracování jednoduchých úloh v databázovém serveru PgSQL může být tento pomalejší než řešení založené například na DBS MySQL [2])
- možnost rozšiřitelnosti o novou funkcionalitu
- optimalizace výkonu
- disponuje téměř stejnou funkcionalitou jako plně profesionální komerční řešení

Tato práce tedy předpokládá základní znalosti dotazovacího jazyku SQL (Structured Query Language), předchozí zkušenost s jiným databázovým systémem a pochopení základní problematiky relačních databázových systémů (RDBS). Metody používané v PgSQL jsou blízké například komerčnímu řešení databázového serveru Oracle, znalost práce v PgSQL lze tedy vnímat i jako další logický mezikrok k cestě k robustnějšímu řešení.

Práce nemá simulovat kompletní popis a možnosti jednotlivých popisovaných technik. Poukazuje na metody, které mohou zefektivnit práci s moderními DBS, v této práci pak konkrétně PgSQL.

I. TEORETICKÁ ČÁST

1 ZÁKLADNÍ POJMY

1.1 Databázový systém

DBS (databázový systém) tvoří DB (databáze) a SŘBD (systém řízení báze dat).

1.2 Databáze

Základní jmenný uživatelský prostor v rámci databázového clusteru. Cluster databázového serveru může obsahovat neomezené množství databází. Databáze obsahují neomezené množství schémat.

1.3 Systém řízení báze dat

Je aplikace, která je pověřena zpracováním, ukládáním a získáváním údajů (dat) v DB.

1.4 Cluster

Clusterem se rozumí v DBS PgSQL každá běžící instance DBS PgSQL. Clustery mohou poté naslouchat na rozdílných IP adresách, portech, socketech. Běžící clustery se navzájem neovlivňují, SŘBD obsluhují různé DB a DB jsou mezi sebou nezávislé, čehož lze využít k aplikaci důraznější bezpečnostní politiky. Rozdíl v režii při provozu jediného clusteru a více souběžných clusterů je minimální.

1.5 Schéma

Schéma obsahuje vlastní databázové objekty (tabulky, pohledy, spouště atd.). Jednotlivé objekty v různých schématech se neovlivňují, lze tak například mít tabulku se shodným názvem ve více různých schématech. V analogii s programovacími jazyky se jedná v podstatě o jmenné prostory, kontejnery. Uživatel databáze má díky schématům lepší možnost logické organizace objektů a může tak zabránit konfliktům v pojmenování, protože k jednotlivým objektům v rámci databáze se přistupuje pomocí notace *schema.objekt*.

1.6 Role

Role může být chápána jako standardní uživatelský účet v DBS. Role obsahuje vlastnost *login*/*nologin*, která ji zásadním způsobem ovlivňuje. V případě nastavení vlastnosti *login* je možné se prostřednictvím role přihlásit do databáze a je možné uvažovat o roli jako o uživatelském účtu. V opačném případě a nastavení *nologin* není možné se s takovou rolí připojit. Díky vlastnosti, že role mohou sdružovat (zanořovat) další role jsou pak tyto role bez hesla chápány jako skupiny. S výhodou lze využít tedy vlastnosti, že skupina (role) obsahuje jinou skupinu (roli) a nijak není limitován počet těchto zanoření.

2 OPRÁVNĚNÍ

Oprávnění je definováno vždy pro celý cluster, nikoliv pro jednotlivé databáze obsažené v clusteru. Systémové tabulky obsahující informace o oprávnění jsou viditelné napříč celým clusterem. Jedna role tak může mít definován přístup k více databázím. Naopak nelze v rámci clusteru vytvořit dvě role s identickým uživatelským jménem, byť by každá role měla mít vyhrazený přístup do jiné databáze.

Oprávnění lze definovat na úrovní následujících objektů: databáze, schéma, tabulka, sekvence, funkce, jazyk, tablespace.

Je-li role vlastníkem (owner) objektu, není nutné explicitně definovat pro tuto roli oprávnění vůči tomuto objektu, jelikož vlastník automaticky získává veškerá oprávnění související s objektem. Vlastník objektu má možnost se těchto automaticky získaných oprávnění zříct.

2.1 Oprávnění obdržená při vytváření role

Role získává některá oprávnění již při svém vytváření. Tato oprávnění se volí pomocí příznaků uvedených za základní syntaxí příkazu pro vytváření role, CREATE ROLE "jméno-role" [PRÁVA]. Pokud PRÁVA nejsou zadána, databázový server má předvoleny výchozí hodnoty, které použije. PRÁVA mohou být:

- SUPERUSER | NOSUPERUSER

  Tzv. superuživatel je automaticky oprávněn ke všem operacím v daném clusteru, naopak uživatel bez tohoto příznaku smí provádět pouze operace, které mu byly povoleny. V rámci clusteru je z bezpečnostních důvodů doporučen maximálně jeden superuživatel. Výchozí nastavení: NOSUPERUSER

- CREATEDB | NOCREATEDB

  Role je/není oprávněná k vytváření nových databází. Výchozí nastavení: NOCREATEDB

- CREATEROLE | NOCREATEROLE

  Vytvářená role bude/nebude moci vytvářet další role. V případě, že role bude oprávněná vytvářet nové role, tyto nemusí být podrolemi nadřazené role. Výchozí nastavení: NOCREATEROLE

- INHERIT | NOINHERIT

  Je-li vytvářená role podrolí nadřazené role, příznak udává, zda má tato zanořená role zdědit/nezdědit oprávnění od nadřazené role. Výchozí nastavení: INHERIT

- LOGIN | NOLOGIN

  Role získá/nezíská oprávnění se připojit k databázovému clusteru. Výchozí nastavení: NOLOGIN

- PASSWORD heslo

  Nastaví přihlašovací heslo vytvářené roli. Nabývá smyslu v případě, že je nastaven příznak LOGIN.

- IN ROLE jméno-role

  Pomocí příznaku lze vytvářet vnořené (zanořené) role již pod existující roli. Ty poté mohou dědit (INHERIT) od nadřazené role.

- a další, viz manuálové stránky [5].

2.2 Oprávnění globální skupiny PUBLIC

Každá role po jejím vytvoření automaticky spadá do souhrnné globální skupiny, ke které lze přistupovat pomocí klíčového slova PUBLIC. Skupina PUBLIC má v závislosti na typu objektu definovanou základní úroveň oprávnění:

- CONNECT

  Role se může připojit k libovolné databázi a jsou pro ni viditelné objekty v databázi obsažené, ale s objekty nemůže nijak více pracovat, například provádět SELECT dotazy nad objekty typu tabulka, vlastní data tak nejsou viditelná. Je však viditelná struktura databáze, tedy jaká schémata databáze obsahuje, názvy a struktura jednotlivých tabulek, názvy funkcí atd.

- CREATE

  Role může vytvářet nová schémata pod aktuální databází. Umožňuje roli ve schématech vytvářet další nové objekty (tabulky, pohledy atd.).

- TEMP

  Role může vytvářet dočasné tabulky.

- EXECUTE

  Role může spouštět existující funkce.

- USAGE

  Role může používat nainstalované programovací jazyky v databázi.

V produkčním prostředí je doporučeno, aby po vytvoření role byla tato automaticky vytvořená oprávnění odebrána a manuálně vyspecifikována později. Je velkým potenciálním rizikem, aby klienti, kteří sdílí své databáze v rámci jednoho databázového clusteru, mohli vidět například strukturu svých objektů.

3 SPECIÁLNÍ DATOVÉ TYPY

3.1 Datový typ SERIAL a BIGSERIAL

DBS PgSQL disponuje mechanismem umožňujícím pracovat s tzv. sekvencemi. Sekvence jsou de facto celočíselné čítače, které mohou být použity k lineárnímu i nelineárnímu číslování požadovaných sloupců. S výhodou lze sekvence využít pro celočíselné primární klíče. Při vytváření tabulky a uvedení datového typu SERIAL/BIGSERIAL je automaticky vytvořena nová sekvence. Rozsahem je datový typ SERIAL shodný s typem INTEGER, tedy 4 byty. Typ BIGSERIAL s rozsahem 8 bytů. Použití typu SERIAL/BIGSERIAL je tedy nejsnazší cestou, jak zajistit automatickou inkrementaci požadovaných celočíselných sloupců (v jiných databázových systémech se tato automatická inkrementace zajišťuje přidáním parametru AUTO_INCREMENT za požadovaný sloupec). V případě odstranění tabulky je sekvence automaticky odstraněna a to i bez použití klíčového slova CASCADE v příkazu DROP.

```
CREATE TABLE "public"."pocitadlo" (
 "primarni_klic" SERIAL PRIMARY KEY,
 "jmeno"         VARCHAR(24)
 );


INSERT INTO "public"."pocitadlo" ("jmeno") VALUES ('Vlastimil');
INSERT INTO "public"."pocitadlo" ("jmeno") VALUES ('Barbora');


SELECT * FROM "public"."pocitadlo";
primarni_klic  |   jmeno
---------------+-----------
             1 | Vlastimil
             2 | Barbora
```

3.2 Datový typ DOMAIN

```
CREATE DOMAIN name [ AS ] data_type
 [ DEFAULT expression ]
 [ constraint [ ... ] ]
```

constraint je:

```
[ CONSTRAINT constraint_name ]
{ NOT NULL | NULL | CHECK (expression) }
```

DOMAIN, česky doména, není sama o sobě datovým typem. Jedná se o mechanismus, díky kterému je možné vytvořit si vlastní datový typ, který:

- si uživatel pojmenuje podle svých potřeb a poté k němu pomocí zvoleného názvu přistupuje,

- je založen na některém ze základních datových typů (integer, char, float ...),
- může zvolený základní datový typ rozšířit o požadovaná omezení,
- si může zajistit vynucení nenulové hodnoty NOT NULL,
- může obsahovat výchozí hodnotu v případě, že hodnota nebude při vkládání specifikována.

Doména nalezne uplatnění v případech, kdy jsou v různých tabulkách obsaženy stejné typy informací, které je nutné kontrolovat na žádaná omezení. Výhodou poté je, že případná aktualizace domény se projeví všude tam, kde je použita. Na aplikační úrovni již záleží na programátorovi, zda-li bude i v aplikaci obsažena kontrola na omezení vstupních dat anebo zda bude tato kontrola zajištěna pouze použitím domény na straně DBS.

```
CREATE DOMAIN "public"."kontrola_psc" AS VARCHAR(6)
 NOT NULL
 DEFAULT '76001'
 CHECK(
  VALUE ~ '^\\d{5}$' OR VALUE ~ '^\\d{3} \\d{2}$'
);


CREATE TABLE "public"."psc" (
 "id"     SERIAL  PRIMARY KEY,
 "psc"    kontrola_psc,
 "nazev"  VARCHAR(24)
);


-- následující vložení proběhne v pořádku
INSERT INTO "public"."psc" ("psc", "nazev") VALUES ('47001',  'Česká Lípa');
-- následující vložení proběhne v pořádku
INSERT INTO "public"."psc" ("psc", "nazev") VALUES ('110 01', 'Praha');
-- následující vložení vyvolá vyjímky, vložení NULL hodnoty do psc
INSERT INTO "public"."psc" ("psc", "nazev") VALUES (NULL,     'Litomyšl');
-- následující vložení bude v pořádku, jako psc se použije výchozí hodnota
INSERT INTO "public"."psc" ("nazev")        VALUES ('Zlín');
```

3.3 Datový typ ARRAY

Array, česky pole, je n-tice hodnot zvoleného datového typu a to jak jednorozměrná (vektory) tak mnohorozměrná (matice). V PgSQL není nutné předem deklarovat velikost pole, databázový server se sám stará o alokaci a realokaci systémových prostředků. V případě, že je velikost pole přesto uvedena, může být tato překročena, aniž by systém vyvolal chybu. Uvádění velikosti pole má tedy spíše informativní charakter pro návrháře databáze.

```
CREATE TABLE "public"."prumery" (
 rok     INTEGER,
```

```
 prumer  INTEGER[12]
);
```

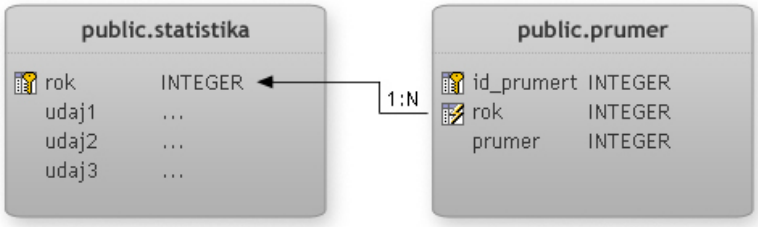
Při vkládání dat do pole jsou hodnoty uzavřeny do složených závorek a hodnoty odděleny čárkou. Celé složené závorky musí být uvedeny v rámci jednoduchých uvozovek.

```
INSERT INTO "public"."prumery" ("rok", "prumer")
 VALUES (2006, '{4,5,6,4,5,6,5,4,8,10,13,8}');       -- 12 hodnot, v pořádku

INSERT INTO "public"."prumery" ("rok", "prumer")
 VALUES (2007, '{4,5,6,5,4,8,10,13,8}');             -- 9 hodnot, v pořádku

INSERT INTO "public"."prumery" ("rok", "prumer")
 VALUES (2008, '{12,15,4,5,6,4,5,6,5,4,8,10,13,8}'); -- 14 hodnot, v pořádku
```

Pracovat lze s celými poli, jejich jednotlivými hodnotami na jednotlivých pozicích (indexech), kdy pozice jsou číslovány od jedničky. DBS disponuje řadou notací a funkcí pro práci s poli.

```
-- načtení celého pole 'prumer'
SELECT rok, prumer FROM prumery;

-- načtení pouze hodnoty na pozici prvního indexu z pole 'prumer'
SELECT rok, prumer[1] FROM prumery;

-- načtení pouze hodnoty na pozici posledního indexu z pole 'prumer'
SELECT rok, prumer[array_upper(prumer,1)] FROM prumery;

-- alespoň jedna hodnota v poli 'prumer' musí byt větší než 10
SELECT rok FROM prumery WHERE 10 < SOME (prumer);

-- všechny hodnoty poli 'prumer' musí byt větší než 3
SELECT rok, prumer FROM prumery WHERE 3 < ALL (prumer);
```

Možnost datového typu pole je příjemnou vlastností DBS PgSQL. Při návrhu databáze se musí autor rozhodnout, zda použít pole anebo využít nahrazení ukládání dat další databázovou tabulkou, která bude ve vztahu N:1 k nadřazené tabulce.



Obr. 1. Relační náhrada 1:N namísto použití datového typu ARRAY.

Důležitým faktorem při tomto rozhodování může být fakt, zdali bude nutné provádět indexaci dat obsažených v poli, kterou nelze explicitně zajistit. Nebudou-li data indexována, je bez vlivu na výkon možné využít datový typ pole.

3.4 Kompozitní (složený/strukturovaný) datový typ

Nejlépe podstatu tohoto datového typu vystihuje označení složený. Složený datový typ umožní programátorovi definovat navenek jeden složený datový typ, kterému přiřadí pojmenování, kdy tento může v sobě zahrnovat strukturu několika jiných základních datových typů přístupných pod přiřazenými asociativními klíči.

```
-- vytvoření vlastního složeného datového typu
CREATE TYPE "public"."kompl_cislo" AS (
 realna     float,
 imaginarni float
);

-- použití vytvořeného datového typu v nové tabulce
CREATE TABLE "public"."prubeh_funkce" (
 cas     INTEGER,
 hodnota kompl_cislo
);
```

Plnění datového typu probíhá vypsáním všech hodnot do kulatých závorek (na rozdíl od složených závorek použitých u datového typu array) a oddělením jednotlivých hodnot čárkou.

```
INSERT INTO "public"."prubeh_funkce" ("cas", "hodnota")
 VALUES (5,  (2.34, 1.01));

INSERT INTO "public"."prubeh_funkce" ("cas", "hodnota")
 VALUES (10, (2.48, 1.09));

INSERT INTO "public"."prubeh_funkce" ("cas", "hodnota")
 VALUES (15, (3.02, 1.13));
```

Pracovat lze následně s celým datovým typem nebo s jednotlivými složkami. Pro přístup k jednotlivým složkám složeného datového typu je použita notace přes tečku. Pozor, u načítání dat konkrétní složky je nutné název typu uvést do kulatých závorek, jinak server interpretuje přístup ne jako ke složce datového typu, ale jako *schema.tabulka*.

```
-- vypsat pouze reálnou složku z vlastního dat. Typu
SELECT (hodnota).realna FROM "public"."prubeh_funkce";

-- aktualizace jedné složky v podmínce na jiné složce vlastního dat. typu
UPDATE "public"."prubeh_funkce"
 SET hodnota.imaginarni=1.15
 WHERE (hodnota).realna=3.02;
```

Na rozdíl od datového typu array, je u složeného datového typu možné indexovat data. Vytvářet lze i složené indexy z více složek.

```
-- vytvoření indexu pouze ze jedné složky vlastního dat. typu
CREATE INDEX "idx-prubeh_funkce-realna" ON
 "public"."prubeh_funkce"(((hodnota).realna));

-- vytvoření indexu z kombinace několika složek vlastního dat. typu
CREATE INDEX "idx-prubeh_funkce-realna-a-imaginarni" ON
 "public"."prubeh_funkce"(((hodnota).realna),((hodnota).imaginarni));
```

Složený datový typ lze využít nejen pro ukládání strukturovaných dat, ale také jako návratový typ vlastních funkcí, o kterých pojednává následující kapitola.

4 FUNKCE A ULOŽENÉ PROCEDURY

Funkce a uložené procedury fungují velmi podobně a je mezi nimi jen malý rozdíl. Vždy se jedná o uživatelské funkce naprogramované v některém z dostupných programovacích jazyků, který je do DBS zaveden (programovací jazyk musí zavést uživatel s právy SUPERUSER). Jedná se tedy o programování na straně serveru, při volání funkce je celé tělo funkce vykonáno na straně serveru a ke klientovi jsou přenášena pouze výstupní data specifikovaná v těle funkce. Návratová hodnota je hlavním rozdílem mezi funkcí a uloženou procedurou. Funkce po vykonání svého obsahu vrací hodnotu. Uložená procedura vykoná svůj obsah a již nevrací žádnou hodnotu. Formálně DBS PgSQL umožňuje vytvářet pouze funkce, ty však mohou sloužit jako uložené procedury.

Vždy by měl být nalezen kompromis mezi tím, které programové jednotky začlenit do aplikační vrstvy na stranu klienta, a které umístit na stranu serveru. Pozitiva, jež hovoří pro přenesení řešení některých úloh na stranu serveru:

- u složitých úloh, které je nutné řešit průběžným zpracováváním a vyhodnocováním mezivýsledků odpadá režie nutná pro komunikaci mezi klientem (aplikací) a serverem (DBS),
- v důsledku rychlého zpracování úlohy na straně DBS může tento dříve uvolnit alokované zdroje, které tak může využít jiná úloha a značně tím roste výkonnost DBS,
- vykonáváním uložených procedur a funkcí na straně serveru se lze efektivně bránit proti napadení a zneužití aplikace útočníkem pomocí tzv. sql-injectingu (podvrhnutí prováděného dotazu).

Ve funkcích a procedurách není možné explicitně používat transakce (viz. Transakční zpracování, kapitola 7). Funkce jsou zpracovány jako celek v implicitně (vytvoří ji automaticky SŘBD) vytvořené transakci. Pokud proběhne funkce v pořádku, automaticky je transakce potvrzena. V opačném případě funkce/procedura vyhodí výjimku, kterou může programátor zachytit a zpracovat, a zároveň je transakce odvolána.

Funkce a procedury lze programovat v DBS PgSQL v řadě programovacích jazyků.

PL/PgPSM – [6] procedurální programovací jazyk vycházející ze standardu SQL/PSM (přijatého v roce 1998). Do DBS PgSQL implementovaný až v roce 2007. Obsahuje

bohatý repertoár řídících konstrukcí. Implementuje model zachycení a zpracování chyb. Datové typy, funkce přebírá z hostitelského DBS. Nepodporuje I/O operace. Předností jazyka je přenositelnost mezi DBS, které podporují tento jazyk (standard SQL/PSM ponechává určitou volnost v implementaci jazyku, přenesení procedury na jiný DBS může tedy vyžadovat malé zásahy v závislosti na implementaci).

PL/PgSQL – procedurální programovací jazyk vyvinutý pouze pro DBS PgSQL ještě předtím, než byl přijat standard SQL/PSM. Ideově je velmi podobný procedurálnímu programování použitého u firmy Oracle. Procedury naprogramované v PL/PgSQL jsou nepřenositelné na jiné DBS (pouze na DBS firmy Oracle lze s minimálními úpravami procedury snadno přenést). Datové typy, funkce přebírá z hostitelského DBS. Umožňuje definovat vlastní proměnné uvnitř těla procedury. Obsahuje rovněž řadu konstrukcí pro řízení běhu aplikace, zachycení a zpracování chybových stavů, dynamické dotazování a řadu dalších vlastností. Nepodporuje I/O operace. Funkce lze přetěžovat (mít více funkcí stejného názvu s rozdílnými argumenty). Často využívanou vlastností je možnost iterování skrze provedený SQL dotaz, kdy při iterování jsou implicitně použity kurzory (viz. Kurzory, kapitola 6).

C, **PL/Perl**, **PL/Python** a **PL/Tcl**, **PL/Java**, **PL/PHP**, **PL/Ruby**, **PL/Scheme**, **PL/sh** – [7] programování v těchto jazycích se řídí pravidly a zákonitostmi daného jazyka. Z pohledu začlenění vytvořených funkcí do DBS PgSQL je při spuštění těchto funkcí volán vnější interpret, z tohoto pohledu tedy pomalejší řešení než řešení uvedená výše. Na straně druhé je možné řešit řadu úloh v těchto jazycích rychleji než přímo v DBS nebo v klientské aplikaci. Argumentem pro rozšíření funkčnosti o funkce napsané v těchto jazycích je možnost I/O operací. Obtížnější je ladění těchto funkcí. [4]

5 SPOUŠTĚ

```
CREATE TRIGGER name { BEFORE | AFTER } { event [ OR ... ] }
 ON table [ FOR [ EACH ] { ROW | STATEMENT } ]
 EXECUTE PROCEDURE funcname ( arguments )
```

Mechanismus spouští umožňuje automatické vyvolání uložené procedury (viz. Funkce a uložené procedury, kapitola 4) v závislosti na určité události provedené nad specifikovanou tabulkou. Spoušť musí být pojmenována. Událostmi, které mohou spoušť aktivovat, jsou:

- INSERT
  vložení nových dat
- UPDATE
  aktualizace existujících dat
- DELETE
  odstranění existujících dat

Dále je možné specifikovat, zda se spoušť aktivuje:

- BEFORE
  před vykonáním zvolené události
- AFTER
  po vykonání zvolené události

Událost může ovlivnit více řádků specifikované tabulky, je možné se rozhodnout, jakým způsobem se spoušť aktivuje:

- FOR EACH ROW

  je-li ovlivněno více řádků, spoušť se spustí tolikrát, kolik bylo ovlivněno řádků, vždy pro každý ovlivněný řádek
- FOR EACH STATEMENT

  výchozí způsob v případě, že není uvedeno, aktivuje spoušť pouze jednou i v případě že je ovlivněno více řádků

6 KURZORY

Kurzor je možné si představit jako analogii s deskriptorem souboru. Před použitím je nutné kurzor aktivovat (otevřít) a po použití deaktivovat (zavřít). K tomu slouží SQL příkazy OPEN a CLOSE. Data z tabulky zpřístupněné kurzorem jsou získávána příkazem FETCH.

Kurzory je vhodné používat v případech, kdy je nutné pracovat s velkým množstvím dat. Bez použití kurzorů jsou zpracovávaná data uložena v paměti, představují tak velkou režii, a následně jsou s daty prováděny požadované operace, kdy tyto operace znamenají také určitou režii. Systémové prostředky DBS nejsou neomezené a takový přístup přináší řadu komplikací, zejména degradaci jeho výkonu. Přístup ke stejnému problému s využitím kurzoru spočívá v tom, že kurzor pracuje pouze nad řádkem dat, na který ukazuje, odtud analogie s deskriptorem souboru. V paměti zabere místo pouze pro právě zpracovávaný řádek.

Kurzory je nevhodné používat s účelem dosažení vyšší rychlosti (naopak, zpravidla je zpracování dat pomocí kurzoru pomalejší), ale za účelem zajištění realizace objemného zpracování dat.

V DBS PgSQL lze vytvářet dva typy kurzorů. Tzv. forward kurzory je možné procházet pouze směrem vpřed a není možné se vracet. Naproti tomu v tzv. scrollable kurzorech je možné se vracet.

Kurzory se dále rozdělují na volné kurzory a vázané kurzory. Volné kurzory nemají omezený životní cyklus, jsou přístupné pomocí svého jména jak mimo funkci, tak i uvnitř funkcí, do kterých mohou být předány. Kurzor je zrušen až s příkazem CLOSE. Vázaný kurzor je možné použít pouze uvnitř funkce/uložené procedury, kdy je kurzor vytvořen v návěští DECLARE. Po vykonání funkce je kurzor automaticky zrušen.

Následující dva příklady demonstrují rozdíl mezi klasickým zpracováním a zpracováním pomocí kurzoru. Existuje tabulka public.test_table čítající 1.000.000 záznamů. A je nutné načíst data, eventuelně je nějak upravit a zpracovat a uložit do jiné tabulky.

```
CREATE OR REPLACE FUNCTION "public"."copy_test_table"() RETURNS void AS $body$
DECLARE
 radek RECORD;

BEGIN
 DROP TABLE IF EXISTS "public"."test_copy_table";
 CREATE TABLE "public"."test_copy_table" (
  "id"         BIGSERIAL NOT NULL,
  "md5"        CHAR(32)                          NOT NULL,
  "timestamp"  TIMESTAMP(0) without time zone  NOT NULL,
  "boolean"    BOOLEAN DEFAULT TRUE             NOT NULL,
```

```
 CONSTRAINT "idx-test_copy_table-id" PRIMARY KEY("id")
) WITH OIDS;

/* Jeden z možných klasických přístupů, načtení dat a procházení recordsetu
   ve smyčce, případná modifikace dat a vytvoření dynamického dotazu
   a jeho provedení */
/*
FOR radek IN SELECT * FROM "public"."test_table" LOOP
 EXECUTE 'INSERT INTO "public"."test_copy_table"
  ("id", "md5", "timestamp", "boolean")
  VALUES (' || quote_literal(radek.id) || ', '
  || quote_literal(radek.md5) || ', '
  || quote_literal(radek.timestamp) || ', '
  || quote_literal(radek.boolean) || ')';
END LOOP;

*/

/* Další z klasických přístupů, možné použít v případě, kdy není zdrojová
   data nutné upravovat, a jsou kopírována do nové tabulky tak jak jsou */
/*
INSERT INTO "public"."test_copy_table" SELECT * FROM "public"."test_table";

*/

/* Řešení pomocí kurzoru, nejdříve je otevřen kurzor pro konkrétní dotaz */
OPEN kurzor FOR SELECT * FROM "public"."test_table";
/* Je načten řádek na první pozici, kam aktuálně kurzor ukazuje */
FETCH kurzor INTO radek;
/* Vyhodnocení podmínky, zda na řádku, kam kurzor ukazuje jsou data */
WHILE FOUND LOOP
 /* Opět zpracování, případná úprava dat, příprava dynamického dotazu
    a jeho vykonání */
 EXECUTE 'INSERT INTO "public"."test_copy_table"
  ("id", "md5", "timestamp", "boolean")
  VALUES (' || quote_literal(radek.id) || ', '
  || quote_literal(radek.md5) || ', '
  || quote_literal(radek.timestamp) || ', '
  || quote_literal(radek.boolean) || ')';
 /* Po zpracování aktuálního řádku je posunuta pozice kurzoru */
 FETCH kurzor INTO radek;
END LOOP;
CLOSE kurzor;

END;
$body$ LANGUAGE 'plpgsql';
```

7 TRANSAKČNÍ ZPRACOVÁNÍ

Transakční zpracování je nástroj, který má zajistit programátorovi databáze a aplikace správnou integritu dat tzn., že během ukládání (přenosu) dat nedojde k jejich neočekávanému změnění. Takovou neočekávanou změnou dat může být např. případ, kdy při aktualizaci tisíce řádků tabulky je při aktualizaci stého prvního řádku vyhozena chyba, prvních sto záznamů je aktualizováno a následujících devět set nikoliv.

Zároveň transakční zpracování řeší problematiku konkurujících si úloh. Takovou situaci představuje např. moment, kdy se v jednom okamžiku pomocí jednoho připojení vkládají nová data a v současně běžícím druhém připojení je zaslán na DBS požadavek na získání těchto dat.

V neposlední řadě transakční zpracování představuje ochranu před hardwarovou chybou, například nedostatek místa na disku.

Pod *transakcí* je možné si představit příkaz nebo skupinu příkazů, které převedou databázi (resp. data) z jednoho konzistentního stavu do druhého. Pokud bude prováděna aktualizace dat velké tabulky, tak konzistentní stav je před zahájením příkazu a po dokončení příkazu. V mezičase je obsah tabulky nekonzistentní [6].

Při konkurujících si úlohách mohou nastat následující situace:

- **Dirty read** (špinavé čtení) – příkaz SELECT načte data, která nejsou ještě potvrzena v jiné transakci. Zásadní problém vzniká u nepotvrzené transakce, protože data získaná dotazem SELECT se mohou velmi lišit,
- **Nonrepeatable read** (neopakovatelné čtení) – dva po sobě jdoucí dotazy SELECT mohou vrátit různé výsledky,
- **Phantom read** (výskyt fantomů) – při něm může mít uživatel pocit, že v DBS působí nadpřirozené síly.

Standard SQL předepisuje čtyři následující úrovně izolací transakcí:

- **Read uncommitted,**
- **Read committed,**
- **Repeatable read,**
- **Serializable,**

jejichž chování shrnuje následující tabulka.

Tabulka 1: vlastnosti jednotlivých úrovní izolace při transakčním zpracování podle standardu SQL

	Dirty read	*Nonrepeatable read*	*Phantom read*
Read uncommitted	umožněno	umožněno	umožněno
Read committed	nemožné	umožněno	umožněno
Repeatable read	nemožné	nemožné	umožněno
Serializable	nemožné	nemožné	nemožné

DBS PgSQL ale implementuje pouhé dvě úrovně a pro další dvě úrovně má definovány pouze aliasy pro kompatibilní zápis SQL příkazů s jinými DBS. DBS PgSQL tedy nedodržuje u izolací transakcí standard SQL a u dvou aliasových izolací je nutné mít na paměti, že se tyto izolace nechovají v souladu se standardem:

- SERIALIZABLE – nejbezpečnější forma transakčního zpracování. Server nepovolí paralelně přes sebe běžící transakce, ale řadí jednu za druhou. V tomto módu je nutné mít aplikaci připravenu na opakování transakce (bloku příkazů), když bude serverem změna zamítnuta. Velký vliv na výkon DBS, který při velkém zatížení rapidně klesá, jelikož nemohou běžet souběžné transakce,
- READ COMMITTED – výchozí mód PgSQL, kdy dotaz čte pouze data z ukončených/potvrzených transakce, izolace tedy řeší problém se špinavým čtením,
- REPEATABLE READ – alias pro SERIALIZABLE,
- READ UNCOMMITTED – alias pro READ COMMITTED.

V SQL standardu pro transakce existují tři základní příkazy: `BEGIN` – začátek transakce, `ROLLBACK` - odvolání transakce, a `COMMIT` - potvrzení transakce. V DBS PgSQL je možné zvolit úroveň izolace rozšířeným příkazem `BEGIN TRANSACTION [ISOLATION LEVEL transaction_mode]`.

Na následujícím příkladu je uvedena modelová situace, kdy první klient vytváří statistický report. V podobný okamžik se ale druhý klient rozhodne zanést do tabulky nová data bez použití transakčního zpracování

```
-- stav před aktualizací dat:
SELECT rok, prumer[2] FROM prumery;
```

```
 rok  | prumer
------+--------
 2006 |      5
 2007 |      5
 2008 |     15
 2009 |      3
```

```
-- bez transakčního zpracování
-- Klient1; t=0s
UPDATE prumery
 SET prumer[2]=15 WHERE rok=2009;
```

```
-- bez transakčního zpracování
-- Klient2; t=0s
UPDATE prumery
 SET prumer[2]=22 WHERE rok=2009;
```

```
-- stav po aktualizací dat:
SELECT rok, prumer[2] FROM prumery;
```

```
 rok  | prumer
------+--------
 2006 |      5
 2007 |      5
 2008 |     15
 2009 |     22
```

```
-- stejně tak jak je ve výstupu u roku 2009 hodnota 22 by tam mohla být
-- hodnota 15, aniž by to programátor bez transakčního zpracování mohl ovlivnit
-- a jeden z klientů by byl zmaten, proč je v DB jiná hodnota, než očekával
```

```
-- Klient1; t=0s
BEGIN TRANSACTION
 ISOLATION LEVEL SERIALIZABLE;

UPDATE prumery
 SET prumer[2]=15 WHERE rok=2009;

COMMIT;
```

```
-- Klient2; t=0s
BEGIN TRANSACTION
 ISOLATION LEVEL SERIALIZABLE;

UPDATE prumery
 SET prumer[2]=22 WHERE rok=2009;

> ERROR: could not serialize access
due to concurrent update

COMMIT;

> ROLLBACK
```

```
-- stav po aktualizací dat:
SELECT rok, prumer[2] FROM prumery;
```

```
 rok  | prumer
------+--------
 2006 |      5
 2007 |      5
 2008 |     15
 2009 |     15
```

```
-- konkurující uživatel je informován o souběhu a je upozorněn na aktualizaci
-- dat jiným uživatelem
```

8 ZÁMKY

Aby DBS mohl realizovat transakční zpracování, implementuje tzv. zámky. Pomocí implicitních zámků (zámek je automaticky vytvořen v rámci transakce v závislosti na požadovaném modelu izolace a aktuální operaci) řeší problematiku blokování objektů. Ve většině případů postačuje ponechat zamykání objektů automaticky na DBS, mohou ale nastat situace, kdy je žádané zpřísnit pravidla pro uzamykání. Proto je možné zámků využít i explicitně. Programátorovi DB je umožněno zamykat řádky tabulky, celé tabulky, schémata.

Stejně jako transakční zpracování je i pro zámky definováno několik módů, které umožňují/zakazují určité operace. Každý mód má přesně vymezeno, jaké jiné operace blokuje. Těchto módů zámků je mnoho (plný seznam viz [5]), proto je zde uvedeno, pro vytvoření představy, jak se se zámkem pracuje, pouze použití jednoho z mnoha módů.

Mód zámku *Access exclusive* je nejrestriktivnější mód, který nedovolí jakékoliv jiné souběžné operace na uzamčené tabulce, a to vč. dotazu SELECT. V modelovém příkladu nastala situace, kdy je nutné naprosto vyprázdnit tabulku a naplnit ji novými daty. Je kladen důraz na to, aby během okamžiku, kdy bude tato operace probíhat, nemohl být vrácen například prázdný výsledek po požadavku SELECT souběžně pracujícímu klientovi.

```
BEGIN;

-- zamknutí celé tabulky až do ukončení (potvrzení/zamítnutí) transakce
LOCK TABLE prubeh_funkce IN ACCESS EXCLUSIVE MODE;

-- pokud se nyní jiný klient pokusí o jakoukoliv operaci na této tabulce,
-- DBS bude čekat na uvolnění zámku

-- provedení požadovaných operací
DELETE FROM prubeh_funkce;
INSERT INTO "public"."prubeh_funkce" ("cas", "hodnota")
 VALUES (5,  (3.34, 1.01));
INSERT INTO "public"."prubeh_funkce" ("cas", "hodnota")
 VALUES (10, (3.48, 1.09));
INSERT INTO "public"."prubeh_funkce" ("cas", "hodnota")
 VALUES (15, (3.02, 1.13));

-- v okamžiku potvrzení transakce se zámek zruší a tabulka se uvolní
COMMIT;
```

9 OPTIMALIZACE VÝKONU

9.1 Indexy a příkaz REINDEX

Základním prostředkem pro udržení optimálního výkonu DBS je používání indexů. Nejinak tomu je u DBS PgSQL a práce s indexy se zde nijak neliší od jiných DBS.

U tabulek s velkou četností aktualizace dochází k fragmentaci indexů, které nemusí být optimálně uspořádány. V takovém případě je vhodné používat příkaz REINDEX. Příkaz lze použít na několik objektů, z pohledu programátora DB je zajímavá možnost reindexovat konkrétní index, nebo veškeré indexy na dané tabulce. REINDEX by měl být prováděn po příkazu VACUUM, je-li tento také prováděn.

```
-- reindexace konkrétního indexu
REINDEX INDEX "public"."idx-prubeh_funkce-realna"
-- reindexace všech definovaných indexů konkrétní tabulky
REINDEX TABLE  "public"."prubeh_funkce";
```

9.2 Příkaz VACUUM a VACUUM ALL

Když DBS PgSQL aktualizuje řádek, uchovává původní kopii řádku v souboru tabulky a zapíše nový řádek. Původní řádek, který označí za neplatný, mohou stále využívat běžící transakce, které zobrazují databázi v předchozím konzistentním stavu. Podobně i odstraněné řádky se označí jako neplatné, ale fyzicky se neodstraní z tabulky [1].

Provedením příkazu VACUUM lze vyčistit tabulky od těchto neplatných záznamů. U intenzivně se měnících tabulek by měl být příkaz VACUUM prováděn alespoň jednou denně. Již při překročení hranice 100% neplatných záznamů rapidně klesá výkon operací nad tabulkou s tolika neplatnými záznamy [6].

Příkaz VACUUM lze volat bez parametru, poté se snaží DBS optimalizovat všechny tabulky v aktuální databázi. DBS aktualizuje pouze tabulky, jejichž majitelem je role, pod kterou byl VACUUM spuštěn. Dále je možné spustit VACUUM pouze pro konkrétní tabulku, `VACUUM jmeno_tabulky`. Příkaz VACUUM neuzamyká tabulku během optimalizace. Neplatná data jsou uvolněna, ale nejsou fyzicky přeuspořádána.

`VACUUM FULL` spustí optimalizaci pro všechny tabulky obsažené ve všech databázích a je důležitý spíše z pohledu správce DBS. Opět je nutné mít oprávnění pro práci s tabulkami, a jelikož se optimalizuje napříč všemi DB, příkaz spouští zpravidla role s oprávněním SUPERUSER. VACUUM FULL fyzicky přeuspořádává záznamy v datovém souboru. Po

odstranění neplatných záznamů se v závislosti na počtu odstraněných záznamů zmenší i velikost datového souboru. Po dobu optimalizace je tabulka zamknuta.

9.3 Příkaz ANALYZE

Pro každý SQL příkaz existuje tzv. prováděcí plán. Je to posloupnost nízkoúrovňových příkazů, jejichž provedení zajistí korektní naplnění výsledné množiny. Ke každému netriviálnímu SQL příkazu existují desítky až tisíce různě efektivních prováděcích plánů. Výběr jednoho konkrétního, který se bude provádět, je úkol tzv. optimalizátoru. Optimalizátor, použitý v PgSQL, vychází ze statistik. Ke každému sloupci se udržuje histogram hodnot a orientační počet řádků. Pak se relativně snadno určí efekt podmínky. Pokud ale statistiky neodpovídají reálným datům, pak optimalizátor 100% nevybere optimální prováděcí plán [6].

O aktualizaci statistik se postará příkaz `ANALYZE`. Ten by se měl spouštět při změně 20% řádků (samozřejmě, že také vždy po každém importu).

II. PRAKTICKÁ ČÁST

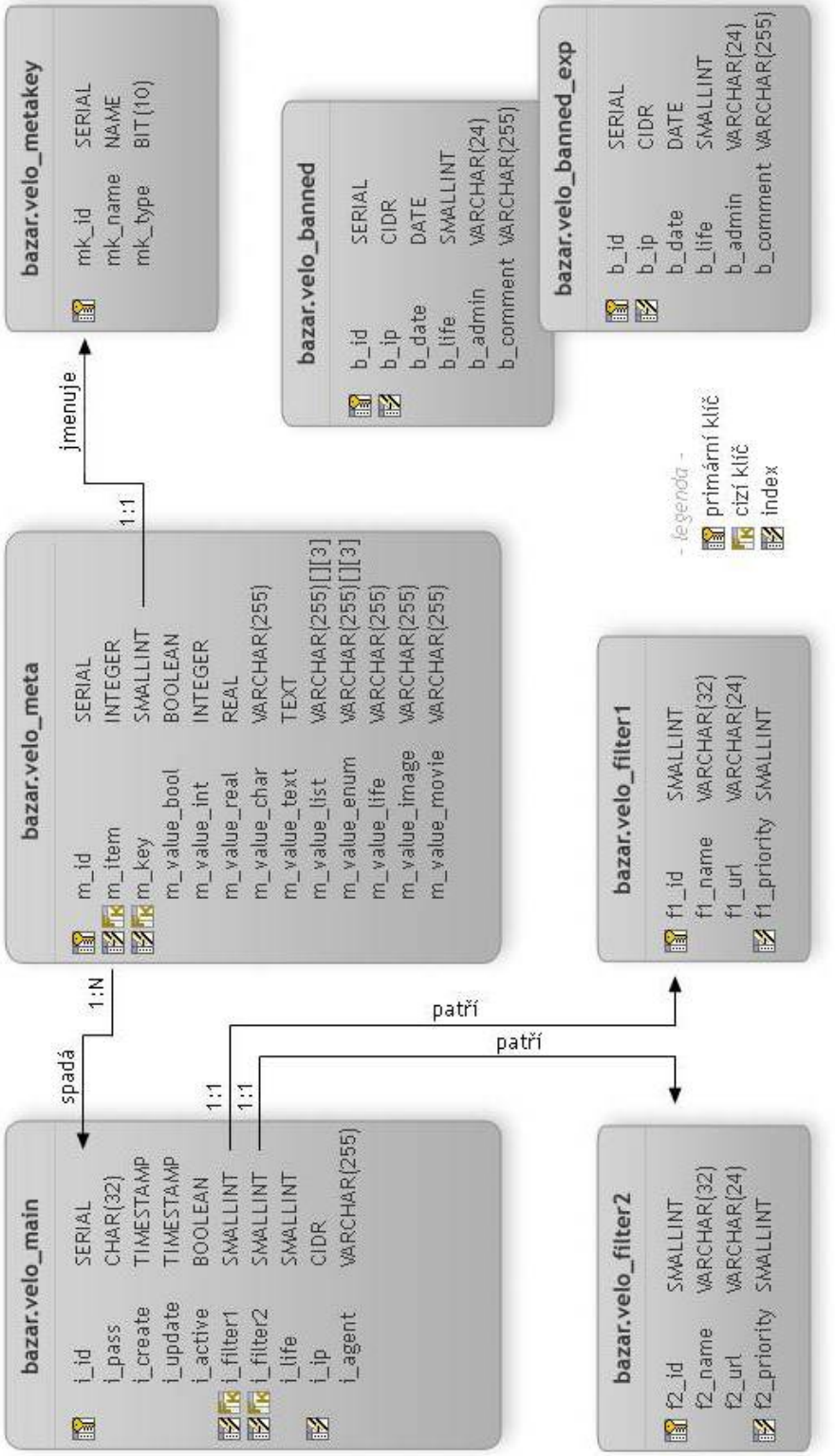
10 PŘÍPADOVÁ STUDIE

Vlastnosti DBS PgSQL popsané v teoretické části, budou demonstrovány na konkrétním uceleném příkladu z reálného tržního prostředí.

Vydavatelství (bude používáno jednotné pojmenovávání perexem *vpress-cz*) vydává několik periodik, časopisů (pojmenování perexy *velo*, *53x11*, ...), které mají vlastní webové prezentace. Tyto prezentace obsahují některé individuální sekce a některé společné. V této praktické části bude naznačen jeden z mnoha možných návrhů a řešení jedné konkrétní společné sekce Bazar (perex *bazar*), která má splňovat následující požadavky.

- Návrh databáze musí být učiněn s ohledem na možnost škálovatelnosti uživatelských práv, aby v některých případech bylo umožněno, že jednotlivé Bazary mohou mezi sebou komunikovat (na stránce vydavatelství se totiž mohou agregovat inzeráty ze všech jednotlivých Bazarů). Zároveň ale návrh musí umožnit oddělení uživatelských práv v určitých případech, aby nedošlo k velkému riziku při neoprávněném získání přístupu k databázi útočníkem (například při údržbě jednoho Bazaru musí být k dispozici role bez oprávnění udělat zásahy v jiném Bazaru).
- Struktura databáze pro Bazar musí být navržena tak, aby bylo možné v budoucnu kdykoliv snadno přidávat nové položky k jednotlivým inzerátům (například rozšíření z dvou stávajících obrázků na čtyři obrázky atd.).
- Vysoká návštěvnost Bazaru klade velký důraz na rychlost zpracování a získávání informací.
- Možnost blokovat inzerenty vkládající nevhodné inzeráty.
- Programátorovi aplikační vrstvy poskytnout již hotové funkce uložené na straně databázového serveru, které bude moci použít, aniž by do hloubky znal strukturu databáze.

11 DATOVÝ MODEL (ERA DIAGRAM)



Obr. 2. Grafické znázornění navrhnutého datového modelu databáze.

12 OPRÁVNĚNÍ

Vychází se z předpokladu, že administrátor databázového serveru založil jednu databázi s názvem *vpress-cz* a roli pojmenovanou *vpress-cz*. Následující kód byl tedy vytvořen administrátorem, který má oprávnění s příznakem SUPERUSER, které umožňuje vytváření nových databází, rolí atd. SQL příkaz na prvním řádku vytvoří vlastní databázi *vpress-cz*. Druhý příkaz vytvoří roli *vpress-cz*, jíž umožní přihlásit se s přiděleným heslem a s právy vytváření dalších rolí. Třetí příkaz znemožní stávajícím i v budoucnosti vytvořeným rolím jakýkoliv přístup k databázi *vpress-cz*. Čtvrtý příkaz zajistí možnost připojit se k databázi *vpress-cz* pouze uživateli *vpress-cz*, tento bude moci zároveň díky příznaku CREATE vytvářet pod databázi nová schémata.

```
CREATE DATABASE "vpress-cz";

CREATE ROLE     "vpress-cz"
 LOGIN NOSUPERUSER NOCREATEDB CREATEROLE ENCRYPTED PASSWORD 'LrL4cA7cbWl2';

REVOKE ALL ON DATABASE "vpress-cz" FROM PUBLIC CASCADE;

GRANT CONNECT, CREATE, TEMP ON DATABASE "vpress-cz" TO "vpress-cz";
```

Struktura požadovaných rolí:

- vpress-cz
  (admin. role klienta umožňující vytváření nových objektů s vlastností login)
  - vpress-cz-bazar (s vlastností nologin)
    - vpress-cz-bazar-velo (s vlastností nologin)
      - vpress-cz-bazar-velo-read (s vlastností login)
      - vpress-cz-bazar-velo-write (s vlastností login)
    - vpress-cz-bazar-53x11 (analogicky s výše uvedenou položkou na stejné úrovni)
      - vpress-cz-bazar-53x11-read
      - vpress-cz-bazar-53x11-write
    - *další časopis …*

Následně již pod rolí klienta *vpress-cz* jsou vytvořeny vnořené role, konkr. pro větev *velo*.

```
CREATE ROLE "vpress-cz-bazar"
 NOLOGIN NOINHERIT NOSUPERUSER NOCREATEDB NOCREATEROLE IN ROLE "vpress-cz";

CREATE ROLE "vpress-cz-bazar-velo"
 NOLOGIN INHERIT   NOSUPERUSER NOCREATEDB NOCREATEROLE IN ROLE "vpress-cz-bazar";

CREATE ROLE "vpress-cz-bazar-velo-read"
 LOGIN   INHERIT   NOSUPERUSER NOCREATEDB NOCREATEROLE
 ENCRYPTED PASSWORD 'O4hnfim127yr' IN ROLE "vpress-cz-bazar-velo";

CREATE ROLE "vpress-cz-bazar-velo-write"
 LOGIN   INHERIT   NOSUPERUSER NOCREATEDB NOCREATEROLE
 ENCRYPTED PASSWORD 'Ci6e0MSGxAhU' IN ROLE "vpress-cz-bazar-velo";
```

Záměrně role *vpress-cz-bazar* nezdědila oprávnění od nadřazené role, nemůže se tedy ani připojit k databázi. Umožněním připojení této roli se automaticky tato změna v oprávnění projeví i v podřazených rolích, která od této role již dědí oprávnění.

```
GRANT CONNECT ON DATABASE "vpress-cz" TO "vpress-cz-bazar";
GRANT USAGE ON SCHEMA "bazar" TO "vpress-cz-bazar";
```

Záměr proč role *vpress-cz-bazar* nezdědila oprávnění od nejvyšší role *vpress-cz* je následující. Veškeré nové objekty v databázi (schémata, tabulky atd.) bude vytvářet administrátorská role klienta *vpress-cz*. Administrátorská role *vpress-cz* nikdy nebude použita v aplikační vrstvě a nebude tak vystavena možnosti prozrazení například díky chybě v aplikaci využívající databázi. Role *vpress-cz* se tedy bude automaticky stávat vlastníkem těchto objektů a získá k nim veškerá oprávnění. Těch se teoreticky může zříct, ale to je nežádoucí, jelikož by si role sama sobě zablokovala přístup k objektům. Proto role přímo odvozené od role *vpress-cz* nesmí dědit oprávnění. Byla-li by vyzrazena role používaná v aplikačním prostředí, např. *vpress-cz-bazar-velo-read*, nebude možné s touto rolí napáchat tak rozsáhlé škody.

Následně je tedy nutné přiřadit vytvořeným rolím jednotlivá oprávnění. Je žádané, aby role dědící od *vpress-cz-bazar-velo* mohli číst ze všech použitých tabulek.

```
GRANT SELECT ON "bazar"."velo_main"       TO "vpress-cz-bazar-velo";
GRANT SELECT ON "bazar"."velo_meta"       TO "vpress-cz-bazar-velo";
GRANT SELECT ON "bazar"."velo_metakey"    TO "vpress-cz-bazar-velo";
GRANT SELECT ON "bazar"."velo_filter1"    TO "vpress-cz-bazar-velo";
GRANT SELECT ON "bazar"."velo_filter2"    TO "vpress-cz-bazar-velo";
GRANT SELECT ON "bazar"."velo_banned"     TO "vpress-cz-bazar-velo";
GRANT SELECT ON "bazar"."velo_banned_exp" TO "vpress-cz-bazar-velo";
```

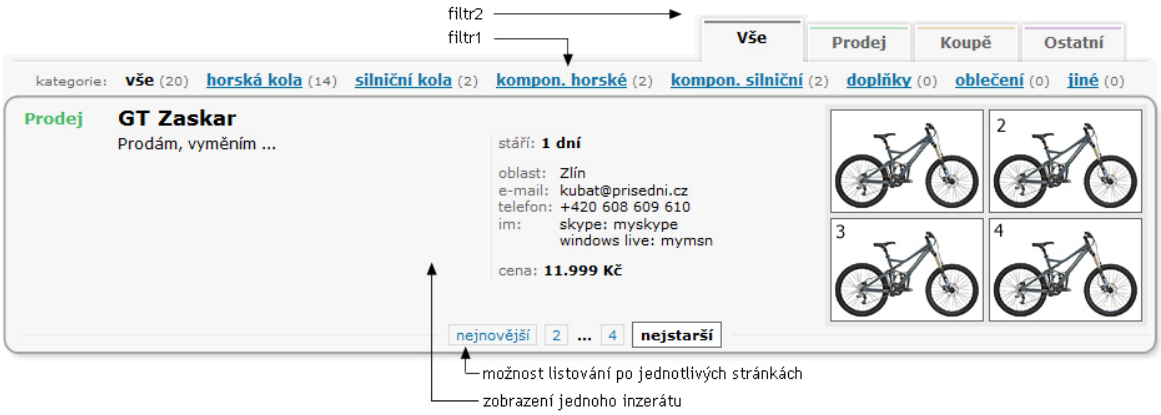
Role dědící od *vpress-cz-bazar-velo-write* musí mít možnost zapisovat data do tabulek, do kterých se ukládají informace o nových inzerátech. Není třeba uvádět oprávnění SELECT, protože i bez jeho uvedení bude moci role *vpress-cz-bazar-velo-write* číst, protože jej dědí od nadřazené role.

```
GRANT INSERT, UPDATE
 ON "bazar"."velo_main" TO "vpress-cz-bazar-velo-write";

GRANT INSERT, UPDATE
 ON "bazar"."velo_meta" TO "vpress-cz-bazar-velo-write";
```

Kromě toho role *vpress-cz-bazar-velo-write* musí mít možnost zapisovat/mazat data do dvou tabulek souvisejících s blokací uživatelů, jelikož na tyto tabulky budou dále vytvořeny spouště, které manipulují právě s těmito tabulkami, proto role, které mohou aktivovat tyto spouště musí mít dostatečná oprávnění pro vykonání uložené procedury.

```
GRANT INSERT, UPDATE, DELETE ON "bazar"."velo_banned" TO "vpress-cz-bazar-velo-write";

GRANT INSERT, UPDATE, DELETE ON "bazar"."velo_banned_exp" TO "vpress-cz-bazar-velo-write";
```

13 ZOBRAZENÍ DAT



Obr. 3. Výstup aplikace zobrazující data.

Cílem této části je poskytnout programátorovi aplikace dvě funkce, které mu budou stačit pro prezentaci dat, tak jak je uvedeno na obrázku Obr. 3. Z pohledu DBS bude využit kompozitní datový typ a funkce. Kdy jedna z funkcí bude schopna vracet sadu dat, druhá funkce samostatnou hodnotu.

První funkce *bazar.velo_list()* bude očekávat tři parametry. Jako výsledek vrátí iterovatelný recordset, v kterém budou vyfiltrovaná data v závislosti na předaných vstupních parametrech. Jednotlivé řádky recordsetu budou odpovídat připravenému kompozitnímu datovému typu. Vlastní kompozitní datový typ obsahuje veškeré sloupce, které jsou nutné pro zobrazení inzerátu. Naopak neobsahují sloupce, které jsou pro zobrazování inzerátu postradatelné (například šifrovaný text uživatelského hesla). Funkce *bazar.velo_list()* je volána s následujícími parametry:

- aktivní první filtr dat `filter1`
  řetězec typu VARCHAR, pro programátora velmi výhodné, jelikož se předpokládá používání následujícího schématu URI:
  *http://SITE_URL/bazar/**filter1**/filter2/?page=5.*
  NULL hodnota nebo prázdný řetězec představuje nepoužitý filtr.
- aktivní druhý filtr dat `filter2`
  řetězec typu VARCHAR,
  *http://SITE_URL/bazar/filter1/**filter2**/?page=5.*
  NULL hodnota nebo prázdný řetězec představuje nepoužitý filtr.

- pořadí stránky výpisu `page`

  Pořadí stránky, která má být zobrazena

  *http://SITE_URL/bazar/filter1/filter2/?**page=5***.

  NULL hodnota nebo prázdný řetězec představuje neznámou hodnotu, automaticky se nastaví na první stránku.

Nejdříve je nutné vytvořit nový kompozitní datový typ *bazar.velo_list*, který bude použít jako návratový typ funkce *bazar.velo_list()*.

```
CREATE TYPE "bazar"."velo_list" AS (
 "i_id"             INTEGER,
 "i_create"         TIMESTAMP(0) without time zone,
 "i_update"         TIMESTAMP(0) without time zone,
 "i_active"         BOOLEAN,
 "i_filter1_id"     SMALLINT,
 "i_filter2_id"     SMALLINT,
 "i_filter1_name"   VARCHAR(32),
 "i_filter2_name"   VARCHAR(32),
 "i_filter1_url"    VARCHAR(24),
 "i_filter2_url"    VARCHAR(24),
 "i_email"          VARCHAR(255),
 "i_title"          VARCHAR(255),
 "i_text"           TEXT,
 "i_place"          VARCHAR(255),
 "i_telefon"        VARCHAR(255),
 "i_im_icq"         INTEGER,
 "i_im_skype"       VARCHAR(255),
 "i_im_msn"         VARCHAR(255),
 "i_price_value"    INTEGER,
 "i_price_currency" VARCHAR(24),
 "i_img1"           VARCHAR(255),
 "i_img2"           VARCHAR(255),
 "i_img3"           VARCHAR(255),
 "i_img4"           VARCHAR(255),
 "i_age"            VARCHAR(255)
);
```

Funkce *bazar.velo_list()* nebude vracet jednu hodnotu základního typu (INT, CHAR …), ale iterovatelný recordset. Získaná data je možné procházet, řadit, vybírat jednotlivé sloupce, seskupovat data apod. stejně jako je tomu umožněno při jakýchkoliv SELECT dotazech. Nejdříve následující dotazy demonstrují možné použití funkce *bazar.velo_list()*:

```
SELECT * FROM "bazar"."velo_list"(NULL, NULL, NULL);
SELECT * FROM "bazar"."velo_list"('horska-kola', NULL, 2);
SELECT * FROM "bazar"."velo_list"('horska-kola', 'koupe', 2) ORDER BY i_id DESC;
SELECT count(*) as pocet FROM "bazar"."velo_list"('horska-kola', 'koupe', 2);
SELECT i_create FROM "bazar"."velo_list"('horska-kola', 'koupe', 2);
```

Důležité části jsou okomentovány přímo v těle funkce.

```
CREATE OR REPLACE FUNCTION "bazar"."velo_list"(
 filter1        varchar,
 filter2        varchar,
 page           integer
) RETURNS SETOF "bazar"."velo_list" AS $body$
/* V hlavičce funkce jsou uvedeny vstupní parametry, je možné si je vhodně
pojmenovat a určit jejich datový typ (funkce lze přetěžovat). Je nutné
specifikovat také návratový typ. Tím je v tomto případě recordset (klíčové slovo
SETOF) jehož řádek bude odpovídat kompozitnímu datovému typu velo_bazar_list. */

/* V sekci DECLARE jsou nadefinovány lokální proměnné a konstanty použité v bloku
funkce */
DECLARE
 ITEMS_ON_PAGE CONSTANT integer := 4;
 return_var       "bazar"."velo_list";
 return_var_temp  "bazar"."velo_list";
 record_data      RECORD;
 temp_counter     INTEGER := 0;

/* Začátek těla funkce */
BEGIN

/* Speciální konstrukce „FOR target IN query LOOP ... ENDLOOP“ umožňuje provést
„query“, který ve smyčce prochází po jednotlivých řádcích. K řádku je možné
přistupovat pomocí proměnné uvedené na pozici „target“. Tato konstrukce používá
implicitní kurzory. */
FOR record_data IN
 /* Začátek dotazu, který vrátí požadovaná data. V jeho těle jsou volány další
 uživatelské funkce bazar.velo_where_filter1() a bazar.velo_where_filter2() */
 SELECT
  fil1.*,
  fil2.*,
  maindata.*,
  meta.*,
  metakey.*,
  extract(day from age(NOW(),maindata.i_create)) as age_day,
  extract(hour from age(NOW(),maindata.i_create)) as age_hour,
  extract(minute from age(NOW(),maindata.i_create)) as age_minute
 FROM "bazar"."velo_meta"    as meta
 JOIN "bazar"."velo_metakey" as metakey  ON meta.m_key=metakey.mk_id
 JOIN "bazar"."velo_main"    as maindata ON meta.m_item=maindata.i_id
 JOIN "bazar"."velo_filter1" as fil1     ON maindata.i_filter1=fil1.f1_id
 JOIN "bazar"."velo_filter2" as fil2     ON maindata.i_filter2=fil2.f2_id
 WHERE
 maindata.i_id IN (
  SELECT i_id FROM "bazar"."velo_main" as submaindata WHERE
   submaindata.i_filter1 IN (SELECT f1_id FROM
   "bazar"."velo_where_filter1"(filter1)) AND
   submaindata.i_filter2 IN (SELECT f2_id FROM
   "bazar"."velo_where_filter2"(filter2)) AND
   extract(day from age(NOW(),submaindata.i_create)) <= submaindata.i_life
   ORDER BY  submaindata.i_id DESC
   LIMIT ITEMS_ON_PAGE OFFSET (ITEMS_ON_PAGE*(page-1))
 )
 ORDER BY
  maindata.i_id DESC,
  meta.m_key ASC
 /* Konec dotazu, začátek iterační smyčky */
 LOOP

  IF temp_counter=0 THEN
   temp_counter := record_data.i_id;
  END IF;

  -- inicializace
  IF temp_counter<>record_data.i_id THEN
   return_var_temp     := return_var;
   return_var.i_email  := NULL;
   -- a všechny další položky, vynecháno ...
  END IF;
```

```
  -- plnění základních dat
  return_var.i_id     := record_data.i_id;
  return_var.i_create := record_data.i_create;
  -- a obdobně další položky, vynecháno ...

  -- zobrazení slovního stáří inzerátu
  IF extract(day from age(NOW(),record_data.i_create))>0 THEN
   return_var.i_age := record_data.age_day||' dní';
  ELSIF extract(hour from age(NOW(),record_data.i_create))>0 THEN
   return_var.i_age := record_data.age_hour||' hodin';
  ELSIF extract(minute from age(NOW(),record_data.i_create))>0 THEN
   return_var.i_age := record_data.age_minute||' minut';
  ELSE
   return_var.i_age := 'méně než 1 minuta';
  END IF;

  -- plnění metadat
  IF record_data.mk_name='email' THEN
   return_var.i_email := record_data.m_value_char;
  ELSIF record_data.mk_name='email-show' THEN
   IF record_data.m_value_bool=FALSE THEN
    return_var.i_email := NULL;
   END IF;
  -- a další položky, vynecháno ...
  ELSIF record_data.mk_name='title' THEN
   return_var.i_title := record_data.m_value_char;
  END IF;

  IF temp_counter=record_data.i_id THEN
   CONTINUE;
  ELSE
   temp_counter := record_data.i_id;
  END IF;

  /* Přímo ve smyčce se vracejí pomocí klíčového slova NEXT jednotlivé řádky */
  RETURN NEXT return_var_temp;
 END LOOP;

 IF temp_counter>0 THEN
  RETURN NEXT return_var;
 END IF;

 RETURN;
END;
$body$ LANGUAGE 'plpgsql';
```

Druhá funkce *bazar.velo_list_pagecount()* očekává pouze dva parametry, ty jsou shodné s výše uvedenou funkcí. Funkce vrátí celé číslo představující počet stránek nutných k prolistování všech inzerátů.

- aktivní první filtr dat `filter1`
- aktivní druhý filtr dat `filter2`

Použití funkce je následující:

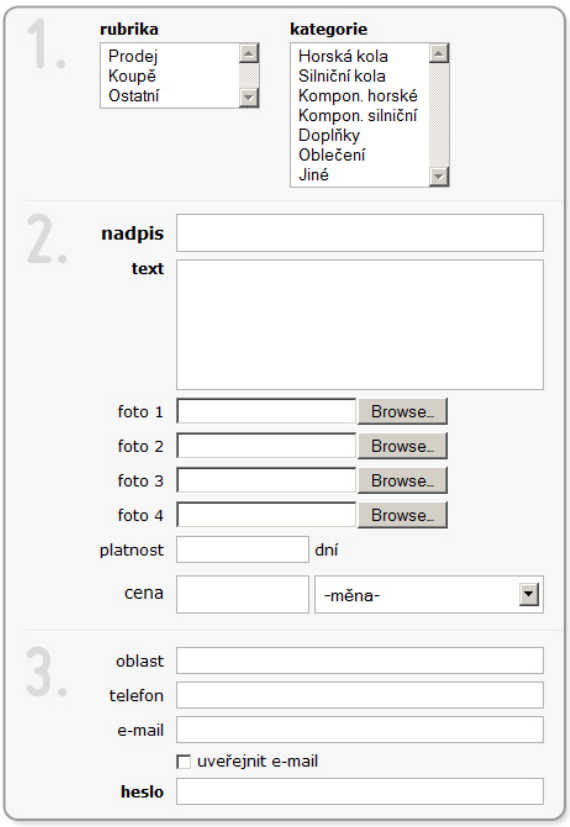
```
SELECT * FROM "bazar"."velo_list_pagecount"('', '');
SELECT * FROM "bazar"."velo_list_pagecount"('silnicni-kola', NULL);
SELECT * FROM "bazar"."velo_list_pagecount"('horska-kola', 'koupe');
```

Vytvoření funkce:

```
CREATE OR REPLACE FUNCTION "bazar"."velo_list_pagecount"(
 filter1        varchar,
 filter2        varchar
) RETURNS INTEGER AS $body$
```

```
DECLARE
 ITEMS_ON_PAGE CONSTANT integer := 4;
 list_pagecount REAL;

BEGIN

SELECT
 INTO list_pagecount
  count(*)
 FROM "bazar"."velo_main"    as maindata
 JOIN "bazar"."velo_filter1" as fil1     ON maindata.i_filter1=fil1.f1_id
 JOIN "bazar"."velo_filter2" as fil2     ON maindata.i_filter2=fil2.f2_id
 WHERE
 maindata.i_id IN (
  SELECT i_id FROM "bazar"."velo_main" as submaindata WHERE
  submaindata.i_filter1 IN (SELECT f1_id FROM
   "bazar"."velo_where_filter1"(filter1)) AND
  submaindata.i_filter2 IN (SELECT f2_id FROM
   "bazar"."velo_where_filter2"(filter2))

 );

 RETURN ceil(list_pagecount/ITEMS_ON_PAGE);
END;
$body$ LANGUAGE 'plpgsql';
```

14 VLOŽENÍ A AKTUALIZACE DAT



Obr. 4. Aplikační formulář pro získání nových dat.

Tabulky nesoucí data vlastních inzerátů jsou často měněna, je vhodné znát skutečnou četnost těchto změn a na základě četnosti zvolit a správně načasovat periodu mezi opakovaným spouštěním příkazů VACUUM, REINDEX a ANALYZE u těchto tabulek. Na úrovni DBS k tomu budou využity spouště, uložené procedury a sekvence.

Princip celého mechanismu bude velmi jednoduchý. Spouště umístěné na jednotlivé tabulky budou volat proceduru, která bude vždy aktualizovat vytvořenou sekvenci. Programátor databáze může sledovat a nulovat toto počítadlo podle svých potřeb a na základě získaných údajů správně naplánovat optimalizace.

Příkaz VACUUM nelze spouštět uvnitř uložených procedur! Příkaz REINDEX a ANALYZE ano, ale jak bylo uvedeno v teoretické části, uživatel který příkaz spustí, musí být majitelem tabulky. Jelikož veškeré operace v aplikaci jsou spouštěny s rolemi, které záměrně nejsou vlastníky objektů, nelze tedy uvažovat ani o příkazech REINDEX a ANALYZE jako o spustitelných uvnitř procedur.

Veškeré optimalizační procesy je vhodné spouštět například pomocí aplikace CRON, která na UNIXových systémech dokáže periodicky spouštět zadané úlohy.

Vytvoření sekvence pro počítání provedených změn:

```
CREATE SEQUENCE "bazar"."seq-velo_optimize"
  INCREMENT BY 1
  NO MAXVALUE
  NO MINVALUE
  CYCLE
  CACHE 1;
```

Vytvořené společné procedury pro inkrementaci sekvence:

```
CREATE OR REPLACE FUNCTION "bazar"."velo_optimize_trigger"()
RETURNS trigger AS $body$
BEGIN

 PERFORM nextval('seq-velo_optimize');
 RETURN NEW;

END;
$body$ LANGUAGE 'plpgsql';
```

Vytvoření spouští na požadovaných tabulkách:

```
CREATE TRIGGER "velo_optimize_main"
 AFTER INSERT OR UPDATE OR DELETE
 ON "bazar"."velo_main" FOR EACH ROW
 EXECUTE PROCEDURE velo_optimize_trigger();

CREATE TRIGGER "velo_optimize_meta"
 AFTER INSERT OR UPDATE OR DELETE
 ON "bazar"."velo_meta" FOR EACH ROW
 EXECUTE PROCEDURE velo_optimize_trigger();
```

Práce se sekvencí z pohledu programátora databáze:

```
-- zobrazí aktuální počet změn od posledního vyresetování sekvence
SELECT currval('seq-velo_optimize');

-- vyresetuje sekvenci na jedničku
SELECT setval('seq-velo_optimize', 1);
```

15 MOŽNOST BLOKACE

Systém vypořádání se s uživateli opakovaně vkládajícími nevhodné inzeráty může být vyřešen mnoha způsoby. Popsané řešení využije z pohledu DBS mechanismus spouští, funkcí, uložených procedur, dynamických dotazů, kurzorů a zámků.

Z pohledu programátora aplikace bude řešení v podstatě banální, programátor aplikace musí pouze naprogramovat jednoduché rozhraní pro správu tabulky s blokovanými IP adresami, která bude spočívat v INSERT/UPDATE/DELETE příkazech na tabulku *bazar.velo_banned*. Vlastní blokaci inzerátu už bude řešit server DBS zcela transparentně.

První ze dvou spouští pojmenovaná *bazar.velo_check_banned_single* bude aktivována po události INSERT nebo UPDATE na tabulce *bazar.velo_main*. Spoušť tedy spustí uloženou proceduru *bazar.velo_check_banned_trigger_single()* v podstatě při jakékoliv interakci uživatele s hlavní tabulkou inzerátů. Vlastní uložená procedura spustí připravenou nadřazenou společnou proceduru *bazar.velo_check_banned_function(cidr)*, které v jednom parametru předá aktuální IP adresu inzerenta, který inzerát vkládá/aktualizuje.

Druhá ze spouští pojmenovaná *bazar.velo_check_banned_all* je navázána opět na událost INSERT nebo UPDATE na tabulce *bazar.velo_banned*. Vkládat/aktualizovat tuto tabulku bude oprávněn pouze správce sekce Bazar. Při každé modifikaci tabulky tedy spoušť aktivuje uloženou proceduru *bazar.velo_check_banned_trigger_all*, která ve svém těle nedělá nic jiného, než zavolá nadřazenou společnou proceduru *velo_check_banned_function(cidr)*, kde jako parametr je předána hodnota NULL.

Společná procedura *velo_check_banned_function()* načte všechny blokované IP adresy z tabulky *bazar.velo_banned* (v případě, že je v parametru funkci předána konkrétní IP adresa, omezí se výběr pomocí podmínky na tuto adresu). Následně procedura provede nastavení příznaku o publikování inzerátu v hlavní tabulce *bazar.velo_main* na *false*, čímž inzerát učiní neaktivním. Při zadání blokované IP adresy je zadán i počet dní, po které má být IP adresa blokována. Procedura kontroluje, zda již počet dní vypršel, pokud ano, automaticky funkce maže záznam o blokaci a přesouvá jej do záložní tabulky s exspirovanými blokacemi.

Vytvoření první procedury a spouště, která jí bude automaticky spouštět:

```
CREATE OR REPLACE FUNCTION "bazar"."velo_check_banned_trigger_single"()
RETURNS trigger AS $body$
BEGIN
 /* V případě, že je žádané provést dotaz a výstup zahodit, slouží
  k tomu příkaz PERFORM. V tomto případě PERFORM zavolá společnou
  proceduru bazar.velo_check_banned_function */
 PERFORM "bazar"."velo_check_banned_function"(NEW.i_ip::CIDR);
 RETURN NEW;
END;
$body$ LANGUAGE 'plpgsql';

CREATE TRIGGER "velo_check_banned_single"
 AFTER INSERT OR UPDATE
 ON "bazar"."velo_main" FOR EACH ROW
 EXECUTE PROCEDURE velo_check_banned_trigger_single();
```

Vytvoření druhé procedury a spouště, která jí bude automaticky spouštět:

```
CREATE OR REPLACE FUNCTION "bazar"."velo_check_banned_trigger_all"()
RETURNS trigger AS $body$
BEGIN
 /* zavolání společné procedury bazar.velo_check_banned_function */
 PERFORM "bazar"."velo_check_banned_function"(NULL::CIDR);
 RETURN NEW;
END;
$body$ LANGUAGE 'plpgsql';

CREATE TRIGGER "velo_check_banned_all"
 AFTER INSERT OR UPDATE
 ON "bazar"."velo_banned" FOR EACH ROW
 EXECUTE PROCEDURE velo_check_banned_trigger_all();
```

Vytvoření společné procedury, kterou volají obě výše uvedené procedury:

```
CREATE OR REPLACE FUNCTION "bazar"."velo_check_banned_function"(
 cidr_blok        cidr
) RETURNS void AS $body$

DECLARE
 -- banned_recordset RECORD;
 temp_var            INTEGER;
 dotaz               TEXT;
 banned_cursor       refcursor;
 banned_row          "bazar"."velo_banned"%ROWTYPE;
 item_row            "bazar"."velo_main"%ROWTYPE;

BEGIN

 /* v proměnné „dotaz“ je vytvářen dynamický dotaz */
 dotaz = 'SELECT * FROM "bazar"."velo_banned" ';
 IF cidr_blok IS NOT NULL THEN
  dotaz = dotaz || 'WHERE b_ip = ' || quote_literal(cidr_blok);
 END IF;

 /* Naznačené řešení pomoci klasické smyčky */
 /*
 FOR banned_recordset IN EXECUTE dotaz LOOP
   RAISE NOTICE 'Testovani adresy: %', banned_recordset.b_ip;
 END LOOP;
 */

/* Řešení pomocí explicitního kurzoru, nejdříve je otevřen kurzor pro dynamicky
připravený dotaz */
OPEN banned_cursor FOR EXECUTE dotaz;
/* Je načten první řádek výsledku dotazu, na který kurzor ukazuje */
FETCH banned_cursor INTO banned_row;
/* Testování, zda na nalezené pozici kurzoru byla nalezena data */
WHILE FOUND LOOP
  /* vykonávání potřebných úkolů */
  temp_var := extract(day from age(NOW(),banned_row.b_date));
  IF temp_var>banned_row.b_life THEN
```

```
  EXECUTE 'INSERT INTO "bazar"."velo_banned_exp"
   SELECT * FROM "bazar"."velo_banned"
   WHERE b_id = ' || quote_literal(banned_row.b_id);
   EXECUTE 'DELETE FROM "bazar"."velo_banned"
    WHERE b_id = ' || quote_literal(banned_row.b_id);

   /* Posunutí kurzoru na následující pozici */
   FETCH banned_cursor INTO banned_row;

   /* Příkaz CONTINUE slouží stejně jako v jakýchkoliv jiných progr. Jazycích
   k přeskočení provádění následujících instrukcí a pokračování další iterací */
   CONTINUE;
  END IF;

  EXECUTE 'UPDATE "bazar"."velo_main"
   SET i_active=false WHERE i_ip=' || quote_literal(banned_row.b_ip) || '
   AND i_active=true';

  /* Posunutí kurzoru na následující pozici */
  FETCH banned_cursor INTO banned_row;
 END LOOP;

 /* Zavření kurzoru */
 CLOSE banned_cursor;
 /* Formální RETURN */
 RETURN;
END;
$body$ LANGUAGE 'plpgsql';
```

Otestování funkčnosti:

```
/* Vložení nového inzerátu do databáze.
Je aktivována spoušť „velo_check_banned_single“, která spustí proceduru
„velo_check_banned_trigger_single()“ a tato procedura spustí jinou proceduru
„velo_check_banned_trigger_all('10.8.0.23')“, které v parametru předá IP adresu
„10.8.0.23“. Je-li tato IP adresa obsažena v tabulce blokovaných IP adres, u
inzerátu je rovnou nastaven příznak false, tedy nezobrazovat. */
INSERT INTO "bazar"."velo_main"
 (i_id, i_pass, i_create, i_filter1, i_filter2, i_ip)
 VALUES
 (26, MD5('aaa'), NOW(), 2, 2, '10.8.0.23');


/* Přidání nově zablokované IP adresy do tabulky blokovaných adres.
Je aktivována spoušť „velo_check_banned_all“, která spustí proceduru
„velo_check_banned_trigger_all()“ a tato procedura spustí jinou proceduru
„velo_check_banned_function()“. Výsledkem je skutečně nastavení příznaku false u
všech inzerátu v tabulce bazara.velo_main, u kterých se vyskytovala IP adresa
10.8.0.6. Mimo to byly zaarchivovány expirované blokace. */
INSERT INTO "bazar"."velo_banned"
  VALUES (4, '10.8.0.6', NOW(), 4, 'admin', 'porušení');
```

ZÁVĚR

Smyslem bakalářské práce bylo poukázat na pokročilé techniky programování v DBS PgSQL. Práce tedy nejdříve v teoretické úrovni a následně i v praktické rovině ukázala na techniky, které nemusejí být programátorovi DBS známé nebo zcela zřejmé. Bude-li o těchto možnostech programátor DBS vědět, případně si je osvojí, bude schopen navrhovat takové DB, které se vypořádají s řadou nároků a problémů, které jsou na DB kladeny v reálném produkčním prostředí.

Práce si nekladla za cíl přinést kompletní dokumentaci jednotlivých probíraných technik, ale měla uvést a objasnit možné pokročilé techniky, aby na základě pochopení těchto pokročilých metod a možností mohl čtenář dále rozvíjet své znalosti v této oblasti DBS.

CONCLUSION

The purpose of this bachelor thesis was to point out the advanced methods of programming in PostgreSQL database system. In the theoretical part there is an explanation of general methods which need not be clear or known to a database system programmer and these are further referred to in the practical part. If a programmer is aware of these database system techniques he will be able to create database systems capable of meeting many requirements which are essential in this area.

This bachelor thesis did not aim to give complete documentation of the concrete explained methods but to introduce and clarify all possible technologies so readers could extend their knowledge of database systems.

SEZNAM POUŽITÉ LITERATURY

[1] MOMJIAN, Bruce. PostgreSQL Praktický průvodce. 1. vyd. Brno: Computer Press, 2003. ISBN 80-7226-954-2.

[2] WELLING, Luke - THOMSON, Laura. PHP a MYSQL rozvoj webových aplikací. 1. vyd. Praha: SoftPress, 2002. ISBN 80-86497-20-8.

[3] SCHLOSSNAGLE, George. Pokročilé programování v PHP5. 1. vyd. Brno: Zoner software, 2004. ISBN 80-86815-14-5.

[4] LIBERTY, Jesse. Naučte se C++ za 21 dní. 2. vyd. Brno: Computer Press, 2007. ISBN 978-80-251-1583-1.

[5] PostgreSQL 8.3 Documentation [dokumentace online]. The PostgreSQL Global Development Group, 1996– . [cit. 2009-01-22]. Dostupné z URL http://www.postgresql.org/docs/8.3/.

[6] PostgreSQL [seriál online]. Praha: Pavel Stěhule, 2008– . [cit. 2009-01-22]. Dostupné z URL http://www.pgsql.cz.

[7] PostgreSQL [seriál online]. Praha: Linuxsoft-Marek Olšavský, 2004. [cit. 2009-01-22]. Dostupné z URL http://www.linuxsoft.cz/article.php?id_article=304. ISSN 1801-3805.

SEZNAM POUŽITÝCH SYMBOLŮ A ZKRATEK

DBS Databázový systém.

DB Databáze.

SŘBD Systém řízení báze dat.

PgSQL PostgreSQL

SEZNAM OBRÁZKŮ

SEZNAM TABULEK

SEZNAM PŘÍLOH

PŘÍLOHA P I: VZOROVÁ DATA

```
-- bazar.velo_main
-- ***************
i_id|              i_pass              |      i_create       | i_update | i_active | i_filter1 | i_filter2 | i_life |     i_ip     | i_agent
   ------+----------------------------------+---------------------+----------+----------+-----------+-----------+--------+--------------+----
 01 | 47bce5c74f589f4867dbd57e9ca9f808 | 2009-05-20 22:41:45 |   NULL   | t        |         2 |         2 |     14 | 10.8.0.1/32  | NULL
 02 | 08f8e0260c64418510cefb2b06eee5cd | 2009-05-20 23:02:54 |   NULL   | f        |         2 |         3 |     14 | 10.8.0.2/32  | NULL
 03 | 9df62e693988eb4e1e1444ece0578579 | 2009-05-20 23:47:52 |   NULL   | t        |         2 |         2 |     14 | 10.8.0.3/32  | NULL
 04 | 77963b7a931377ad4ab5ad6a9cd718aa | 2009-05-21 01:02:01 |   NULL   | t        |         3 |         2 |     14 | 10.8.0.4/32  | NULL
 05 | d2f2297d6e829cd3493aa7de4416a18f | 2009-05-21 03:58:12 |   NULL   | t        |         4 |         2 |     14 | 10.8.0.5/32  | Firefox
 06 | 343d9040a671c45832ee5381860e2996 | 2009-05-21 06:01:58 |   NULL   | t        |         5 |         2 |     14 | 10.8.0.6/32  | MSIE
 07 | 343d9040a671c45832ee5381860e2996 | 2009-05-21 06:03:53 |   NULL   | t        |         5 |         2 |     14 | 10.8.0.7/32  | Opera
 08 | ba248c985ace94863880921d8900c53f | 2009-05-21 06:33:23 |   NULL   | t        |         4 |         3 |     14 | 10.8.0.8/32  | NULL
 09 | a3aca2964e72000eea4c56cb341002a4 | 2009-05-21 06:58:45 |   NULL   | t        |         3 |         3 |     14 | 10.8.0.9/32  | NULL
 10 | 36347412c7d30ae6fde3742bbc4f21b9 | 2009-05-21 07:12:18 |   NULL   | t        |         2 |         4 |     14 | 10.8.0.10/32 | NULL


-- bazar.velo_banned
-- *****************
 b_id |    b_ip     |   b_date   | b_life |    b_admin    |     b_comment
------+-------------+------------+--------+---------------+--------------------
    1 | 10.8.0.2/32 | 2009-05-22 |      2 | Kubát         | Porušil ustanovení
    2 | 10.8.0.5/32 | 2009-05-22 |      5 | Administrátor | Porušil kodex
    3 | 10.8.0.9/32 | 2009-05-22 |     10 | Administrátor | Porušil kodex


-- bazar.velo_metakey
-- ******************
mk_id |    mk_name     |  mk_type
------+----------------+------------
     1 | email          | 0001000000
     2 | title          | 0001000000
     3 | text           | 0000100000
     4 | place          | 0001000000
     5 | telefon        | 0001000000
     6 | img1           | 0000000010
     7 | img2           | 0000000010
     8 | img3           | 0000000010
     9 | img4           | 0000000010
    10 | price-value    | 0100000000
    11 | price-currency | 0001000000
    12 | email-show     | 1000000000
    13 | im-icq         | 0100000000
    14 | im-skype       | 0001000000
    15 | im-msn         | 0001000000
```

```
-- bazar.velo_meta
-- ***************
```

m_id	m_item	m_key	m_value_bool	m_value_int	m_value_char	m_value_text	m_value_image
225	1	1			kubat@prisedni.cz		
226	1	2			GT Zaskar		
227	1	3				Prodám, vyměním ...	
228	1	4			Zlín		
229	1	5			+420 608 609 610		
230	1	6					muj-zaskar.jpg
231	1	7					zaskar-v-lese.jpg
232	1	8					venku.jpg
233	1	9					skrabnuty-lak.jpg
234	1	10		11999			
235	1	11			Kč		
236	1	12	t				
237	1	13			66123456		
238	1	14			myskype		
239	1	15			mymsn		

```
-- bazar.velo_filter1
-- ******************
```

f1_id	f1_name	f1_url	f1_priority
1	Vše	all	98
2	Horská kola	horska-kola	88
3	Silniční kola	silnicni-kola	78
4	Kompon. horské	komponenty-horske	68
5	Kompon. silniční	komponenty-silnicni	58
6	Doplňky	doplnky	48
7	Oblečení	obleceni	38
8	Jiné	jine	28

```
-- bazar.velo_filter2
-- ******************
```

f2_id	f2_name	f2_url	f2_priority
1	Vše	all	98
2	Prodej	prodej	88
3	Koupě	koupe	78
4	Ostatní	ostatni	68