Západočeská univerzita v Plzni
Fakulta aplikovaných věd
Katedra informatiky a výpočetní techniky

Diplomová práce

Překlad logického programu do uložených objektů SŘBD PostgreSQL

Plzeň 2015 Kateřina Fuksová

Prohlášení

Prohlašuji, že jsem diplomovou práci vypracovala samostatně a výhradně s použitím citovaných pramenů.

V Plzni dne 3. května 2015

Kateřina Fuksová

Abstract

Title: Translation of logic program into stored objects in PostgreSQL DBMS

Anotation: This thesis focuses on the translation of logic program into stored objects in PostgreSQL DBMS. In the first part, it clarifies the basic concepts of logic programming, the PL/pgSQL language and the theory of graphs. It also describes in detail the algorithm of transformation of logic program into stored objects in PostgreSQL DBMS and the PrologToPostgreSQL program, which deals with the transformation. A great degree of attention is devoted to creating of the main function respecting the relations between the stored functions and calling them in the most effective order. In the future, this program will become the part of experimental deductive database system. In the last part, the results of the testing are mentioned.

Keywords: logic program, stored database objects, PostgreSQL, deductive database system

Abstrakt

Název: Překlad logického programu do uloženýćh objektů SŘBD PostgreSQL

Anotace: Tato diplomová práce se zabývá překladem logického programu do uložených objektů SŘBD PostgreSQL. V první části jsou objasněné základní pojmy týkající se logického programování, programovacího jazyka PL/pgSQL a teorie grafů. Dále je podrobně popsán algoritmus transformace logického programu do uložených objektů SŘBD PostgreSQL a realizace programu PrologToPostgreSQL, který tuto transformaci zajišťuje. Velká pozornost je věnována vytvoření hlavní funkce, která respektuje vztahy mezi uloženými funkcemi a volá je v co nejvýhodnějším pořadí. Tento vytvořený program se v budoucnu stane součástí experimentálního deduktivního databázového systému. V poslední části jsou zmíněny výsledky testování aplikace.

Klíčová slova: logický program, uložené databázové objekty, PostgreSQL, deduktivní databázový systém

Obsah

Seznam obrázků

1 Úvod

Deduktivní databázový systém kombinuje výhody logických programovacích jazyků a objektově-relačních databázových systémů. Logické programovací jazyky mají velkou vyjadřovací schopnost, databáze zase umožňují rychlé zpracování velkého množství dat.

Deduktivní databázový systém umožňuje ukládání faktů a pravidel. Díky definici pravidel je možné z uložených faktů odvozovat i informace, které nejsou explicitně uvedeny. Fakta jsou obdobou záznamů v objektově-relačních databázích, pravidla jsou obdobou relačního pohledu. Pravidla ale mohou být i rekurzivní, takové relace v objektově-relačních databázích definovat nelze [23].

Cílem této diplomové práce je vytvořit aplikaci, která bude umožňovat transformaci logického programu do uložených objektů SŘBD PostgreSQL. Fakta budou uložená jako záznamy v tabulkách a pravidla se transformují do uložených funkcí v jazyce PL/pgSQL. Aplikace bude součástí systému, který bude deduktivní databázový systém implementovat.

Diplomová práce je členěna do kapitol. Následující kapitola shrnuje základní poznatky o logickém programování, kapitola 3 se zabývá SŘBD PostgreSQL a programovacím jazykem PL/pgSQL. Kapitola 4 poskytuje úvod do teorie grafů. V 5. kapitole se seznámíme s algoritmem převodu logických pravidel do jazyka PL/pgSQL. Poté navazuje 6. kapitola, která popisuje implementaci aplikace PrologToPostgeSQL, která tuto transformaci zajišťuje. Závěrečná kapitola shrnuje výsledky testování aplikace.

K práci je přiložena uživatelská příručka, UML diagram tříd vytvořené aplikace a obsah přiloženého DVD.

2 Základy logického programování

Logické programování je programovací technika, která se zásadně liší od klasického, procedurálního programování. Programátor, který vytváří program v procedurálním programovacím jazyku, píše pomocí příkazů postup výpočtu. Tyto příkazy se po spuštění programu postupně provádějí. Musí se tedy vyjádřit nejen, co se má vypočítat, ale i jak k tomu dospět.

Naproti tomu vytvořit logický program znamená zformulovat množinu faktů a pravidel. Tyto informace se ukládají do databáze. Cíl výpočtu se formuluje pomocí tzv. cílové klauzule (neboli dotazu). Účelem výpočtu je zjištění splnitelnosti cíle a zobrazení hodnot proměnných, pro které je cíl splnitelný. Logické programování obvykle probíhá v interaktivním režimu. Zadání dotazu spouští výpočet programu, po zobrazení výsledku se čeká na reakci uživatele.

2.1 Symboly logického programovacího jazyka

Tato část pojednává obecně o logickém programovacím jazyce. Pro lepší přehlednost v dalších kapitolách zde však bude použita syntaxe logického programovacího jazyka Prolog. Tento programovací jazyk je case-sensitive, záleží tedy na velikosti písmen.

2.1.1 Konstanta

Konstantu logického programu tvoří číslice nebo řetězec znaků. Řetězec je buď ohraničený apostrofy nebo začíná malým písmenem a obsahuje pouze číslice, písmena a podtržítko.

Např.: `3, deset, logicky_program, 'logicky program', 'Velikonoce'`

2.1.2 Proměnná

Proměnná je řetězec, který může obsahovat alfanumerické znaky a podtržítko. Aby se proměnné odlišily od konstant, budeme proměnné označovat řetězci, které začínají velkým písmenem nebo podtržítkem.

Proměnné se vyskytují buď v cílovém dotazu jako hledané hodnoty nebo v pravidlech jako účastníci vztahu.

Např.: `X, Jmeno, _cinitel`

2.1.3 Term

Term je definován následovně: Konstanty a proměnné jsou termy. Pokud je t term, potom (t) je také term. Jsou-li t_1 a t_2 termy, potom $t_1 + t_2$, $t_1 - t_2$, $t_1 * t_2$ a t_1/t_2 jsou také termy [5].

Např.: `3, sobota, Jmeno, X + 1, Y / (Z-4)`

2.1.4 Atom

Atom (neboli výskyt predikátu) se zapisuje se ve tvaru:

`predikatovy_symbol(`t_1`, `t_2`, ..., `t_n`)`

Predikátový symbol je řetězec, který začíná malým písmenem a obsahuje pouze číslice, písmena a podtržítko. Argumenty t_1 až t_n jsou termy.

Např.: `rodic(karel, X), potomek(A, B)`

2.1.5 Pravidlo

Pravidlo má tvar:

`predikatovy_symbol(`t_1`, `t_2`, ..., `t_n`) :- `p_1`, `p_2`, ..., `p_m`.`

Skládá se z hlavy a těla a je ukončené tečkou. Hlavu pravidla tvoří atom, tělo pravidla obsahuje konjunkci atomů. Zároveň platí implikace `tělo` $\rightarrow$ `hlava`. Pokud pro dané hodnoty proměnných nabývají všechny výskyty predikátů v těle pravidla logické hodnoty `true`, je pro tyto hodnoty pravdivý i výskyt predikátu v hlavě pravidla.

Např.: Vezměme si za příklad rodinné vztahy. Pravidlem definujeme vztah prarodič:

```
prarodic(X, Y) :- rodic (X,Z), rodic (Z, Y).
```

Pravidlo lze přečíst takto: X je prarodičem Y, jestliže X je rodičem Z a zároveň Z je rodičem Y. Mějme definované následující vztahy: Karel je rodič Jany a Jana je rodič Laury. Pokud si v pravidle `prarodic` za proměnnou `X` dosadíme konstantu `karel`, za `Y laura` a za `Z jana`, budou v těle pravidla pravdivé oba výskyty predikátu `rodic` a tím pádem budeme mít definovaný vztah `prarodic(karel, laura)`.

Zde jsme si uvedli zjednodušený tvar pravidel – omezili jsme se na formule ve tvaru Hornových klauzulí. Některé logické programovací jazyky dovolují použití libovolné formule v těle pravidla. V tom případě se v těle pravidla mohou vyskytovat i závorky a jiné logické operátory než konjunkce (konkrétně negaci a disjunkci).

2.1.6 Fakt

Fakt je tvrzení, které vždy nabývá logické hodnoty `true`. Zapisuje ve tvaru:

`predikatovy_symbol(`t_1`, `t_2`, ..., `t_n`).`

Jedná se tedy o pravidlo s prázdným tělem, kde termy t_1 až t_n jsou konstanty. Použití proměnné v hlavě pravidla s prázdným tělem není dovoleno.

Např: V předchozím příkladě jsme následující vztahy považovali za definované: Karel je rodič Jany a Jana je rodič Laury. Programově je ale definujeme právě pomocí faktů:

```
rodic(karel,jana).
rodic(jana,laura).
```

2.1.7 Predikát

Predikát soubor pravidel se stejným predikátovým symbolem a stejným počtem termů v hlavě pravidla, tedy stejnou aritou. Predikát je buď pravdivý nebo nepravdivý, nabývá tedy funkční hodnoty `true` nebo `false` [4].

Např.: Zůstaneme u rodinných vztahů jako v předchozích příkladech. Vezměme predikát `rodic(Rodic, Dite)`. Tento predikát bude definovaný dvěma fakty:

```
rodic(karel,jana).
rodic(jana,laura).
```

Predikát `rodic(Rodic, Dite)` bude nabývat funkční hodnoty `true` pouze pro dvojice proměnných `(karel, jana)` a `(jana, laura)` . Pro ostatní hodnoty proměnných bude predikát nabývat funkční hodnoty `false`.

2.1.8 Cílová klauzule

Cílová klauzule (neboli dotaz) má tvar:

`?-` C_1`,` C_2`, ...` C_n`.`

C_1 až C_n, jsou dílčí cíle, jejichž konjunkce tvoří globální cíl. Mají tvar atomu. Zadání cílové klauzule spouští výpočet programu. Cílem výpočtu je zjistit:

- zda je cíl splnitelný - odpověď ano/ne
- pro jaké hodnoty je cíl splnitelný – n je počet různých proměnných, které se vyskytují v dotazu. Odpovědí je množina n-tic. Každá n-tice je množina hodnot, pro kterou byla cílová klauzule splněna.

Např.: Mějme logický program definovaný dvěma fakty a jedním pravidlem:

```
rodic(karel,jana).
rodic(jana,laura).
prarodic(X,Y) :- rodic (X,Z), rodic (Z,Y).
```

Příklady cílových klauzulí:

```
?- rodic(jana,laura).
```

Ptáme se, zda je Jana rodičem Laury. Dostaneme kladnou odpověď: `yes`.

```
?- rodic(X,jana).
```

Ptáme se tedy, kdo je rodičem Jany. Odpověď bude `X=karel`.

```
?-prarodic(jana,Y).
```

Ptáme se, čí je Jana prarodič. Jana nemá žádné vnouče, odpověď tedy bude `no`.

2.2 Rekurzivní konstrukce pravidel

Vezměme si předchozí příklad s definicí rodinných vztahů. Pokud bychom chtěli definovat vztah `potomek(Kdo, Ci)`, šlo by v našem případě definovat dvě pravidla:

```
potomek(X,Y) :- rodic(Y,X).
potomek(X,Y) :- prarodic(Y,X).
```

Tedy X je potomkem Y, jestliže Y je rodičem nebo prarodičem X. Kdybychom ale měli definovanou rozsáhlejší rodinnou linii, řekněme třeba 10 generací, museli bychom už definovat takových pravidel mnoho a navíc by nebyla univerzální – s narozením další generace by relace potomek nebyla definována správně. V tomto případě je tedy mnohem praktičtější a správné definovat rekurzivní vztah:

```
potomek(X,Y) :- rodic(Y,X).
potomek(X,Y) :- rodic(Y,Z), potomek(X,Z).
```

Tedy X je potomkem Y, jestliže je Y rodičem X nebo jestliže existuje Z, jehož rodičem je Y a potomkem X. V těle pravidla potomek jsme tedy využili opět predikát potomek.

Díky tomu je naše definice pravidla potomek univerzální a bude fungovat správně pro jakkoliv velký rodokmen. U rekurzivních pravidel je důležité zajistit, aby se výpočet programu nezacyklil. Toho docílíme definicí ukončovací podmínky – musíme stanovit nějaký pevný bod, kde se rekurze zastaví. Tuto ukončovací podmínku v našem případě představuje první pravidlo: `potomek(X,Y) :- rodic(Y,X).` Dále je také důležité, aby tato ukončovací podmínka byla definována dříve než rekurzivní pravidlo a aby uvnitř tohoto pravidla byl rekurzivní predikát co nejvíce vpravo [1, 3].

2.3 Logický program

Logický program je množina faktů a pravidel. Na vlastnosti relací se můžeme dotazovat pomocí cílové klauzule. Ta není součástí programu, ale je příkazem k jeho spuštění [2]. Celý proces logického programování probíhá následovně. Programátor nejprve napíše logický program (množinu pravidel a faktů) a uloží ho do textového souboru. Poté spustí běhové prostředí, takzvaný interpret. Pomocí příkazu `consult` načte do paměti uložený logický program. Pokud načtení programu proběhne správně, přepne se interpret do dotazovacího režimu a programátor může zadat cílový dotaz. Zadáním dotazu se spustí výpočet programu. Výpočet vychází z lo-

gického odvozování. Strategie získávání výsledků závisí na konkrétní implementaci logického programovacího jazyka.

Poté se zobrazí výsledky výpočtu a čeká se na interakci uživatele. Pokud je programátor s výsledkem spokojen, stiskne klávesu `enter` a je zpět v dotazovacím režimu. Pokud ho ale zajímá další řešení cílové klauzule, může stiskem středníku spustit znovu výpočet, který se pokusí nalézt jiné řešení. Pokud žádné další řešení neexistuje, dostaneme negativní odpověď.

2.4 Výpočet programu

Jak již bylo řečeno, výpočet se spouští zadáním cílové klauzule ve tvaru:

```
?- C1, C2, ... Cn.
```

Na proměnné, které se v cílové klauzuli objevují, se váže existenční kvantifikátor – ptáme se, zda existují takové hodnoty proměnných, pro které je splněna konjunkce dílčích cílů. Během výpočtu jsou generovány další cíle pomocí logického odvozování. To zahrnuje unifikaci (porovnávání cílů s hlavami příkazů), rezoluci (nahrazení testovaného cíle konjunkcí podmínek pro jeho splnění) a backtracking (hledání alternativ pomocí zpětného sledování) [2].

2.4.1 Unifikace

Substituce je zobrazení z množiny proměnných do množiny termů. Substituci popisujeme výčtem dvojic, které si odpovídají.

$$\delta = \{X_1 \leftarrow t_1, \ldots, X_n \leftarrow t_n\}$$

Tento zápis značí substituci, která proměnné X_1 přiřazuje term t_1, ... a proměnné X_n přiřazuje term t_n . Aplikace této substituce na formuli A znamená, že ve formuli A nahradíme každý výskyt proměnné X_1 termem t_1, ... a každý výskyt proměnné X_n termem t_n. Výslednou formuli budeme označovat $A\delta$. Říkáme, že substituce δ unifikuje formule A a C, jestliže jejím provedením získáme dvě shodné formule $A\delta$ a $C\delta$ [2].

Např: Mějme formule

```
rodic(karel,X).
rodic(Y,Z).
```

Tyto dvě formule unifikuje substituce $\delta = \{Y \leftarrow karel, X \leftarrow Z\}$. Po jejím provedení získáme z obou různých formulí shodnou formuli rodic(karel, Z). Podobně je unifikuje i substituce $\delta = \{Y \leftarrow karel, Z \leftarrow X\}$, jejím provedením získáme shodnou formuli `rodic(karel, X)`. Unifikace dvou formulí tedy znamená nalezení takové substituce termů za proměnné, aby výsledné formule byly shodné. Možných substitucí je samozřejmě více, například substituce $\delta = \{Y \leftarrow karel, X \leftarrow laura, Z \leftarrow laura\}$ formule také unifikuje. Výsledná formule vypadá takto: rodic(karel, laura). Tato substituce je ale méně obecná než předchozí dvě. Ve výpočtu logického programu se vždy používá co nejobecnější unifikační substituce [2].

2.4.2 Rezoluce

Rezoluce je metoda odvozování klauzulí v rámci predikátové logiky 1. řádu. Má vlastnost, že z libovolné sporné množiny klauzulí umožní odvodit spor. Spor v terminologii logického programování odpovídá prázdné klauzuli. Základním krokem je vytvoření rezolventy, tedy nové klauzule, kterou lze odvodit z výchozích klauzulí. Metoda odvození sporu pomocí rezoluce spočívá v systematickém prohledávání všech možných rezolvent. Všechny možné rezolventy generované při hledání sporu se nazývají strom řešení úlohy. Konkrétní implementace logického programovacího jazyka může využít libovolnou úplnou strategii prohledávání stromu řešení.

2.4.3 Backtracking

V některém kroku výpočtu se může stát, že lze pokračovat více možnými směry. Každá implementace logického programovacího jazyka si podle svého kritéria určí, kterou větví bude pokračovat. Pokud vybraná větev zavede výpočet k nesplnitelnému cíli, je třeba vyzkoušet ostatní možnosti. Pomocí backtrackingu se tedy vrátí zpět do posledního místa, ve kterém bylo na výběr více možností postupu a zvolí doposud neprobádanou větev výpočtu. Backtracking se také využívá, pokud programátor v interaktivním režimu žádá další řešení úlohy.

2.5 Prolog

Prolog je logický programovací jazyk (z anglického PROgramming in LOGic). Vznikl ve Francii v na začátku 70. let 20.století. Prolog byl od počátku využíván pro zpracování přirozeného jazyka (francouzštiny) a pro různé výpočty v oblasti umělé inteligence. Dále je využíván v databázových systémech a expertních systémech a jako podpora specializovaných činností (např. při projektování v CAD systémech) [2, 6].

Dále zde budou popsána některá specifika Prologu, která je nutno zmínit pro pochopení dalších částí této práce.

2.5.1 Anonymní proměnná

Proměnné, které pro nás v daném výpočtu nemají žádný význam, můžeme zapsat jako anonymní proměnné. Ty se značí podtržítkem, tedy znakem '_'. Tyto proměnné Prolog ve svých výsledcích nezobrazuje. Anonymní proměnnou je možné použít pouze v těle pravidla, v hlavě je nepřípustná.

Definujme si predikát muzikant, který bude definovat osoby, které hrají na nějaký hudební nástroj:

```
muzikant(X) :- hraje(X,Y).
```

V tomto pravidle nás zajímá pouze proměnná X, hodnota proměnné Y může být libovolná. Můžeme ji tedy nahradit anonymní proměnnou:

```
muzikant(X) :- hraje(X,_).
```

Pokud se v pravidle vyskytuje více anonymních proměnných, jsou považovány za různé proměnné.

2.5.2 Disjunkce a negace

Některé implementace Prologu dovolují použít v těle pravidel nebo v dotazu použití formulí obsahujících disjunkci a negaci. Disjunkce se v prologu značí středníkem, tedy znakem ';', negace se značí klíčovým slovem not [3].

Např.: Vezměme pravidla definující predikát potomek, která už jsme dříve použili.

```
potomek(X,Y) :- rodic(Y,X).
potomek(X,Y) :- rodic(Y,Z), potomek(X,Z).
```

Pokud implementace prologu dovoluje použít disjunkci v těle pravidla, můžeme tento zápis zkrátit. Díky disjunkci můžeme vytvořit jedno pravidlo, které je ekvivalentní předchozím dvěma pravidlům.

```
potomek(X,Y) :- rodic(Y,X); (rodic(Y,Z), potomek(X,Z)).
```

Např. Vezměme predikát `hraje(Kdo, NaCo)`, který už jsme dříve použili. Kdybychom chtěli definovat množinu osob, které nehrají na žádný hudební nástroj, vypadalo by to asi takhle:

```
nemuzikant(X) :- osoba(X), not(hraje (X,_)).
```

2.5.3 Operátory

Prolog je programovací jazyk, který obvykle nezpracovává numerické úlohy. Přesto ale zavádí pojem operátorů. Kdybychom například potřebovali sečíst dvě čísla, mohli bychom využít funktor `+`, který sčítání zajišťuje. Zápis by vypadal takto: `+(A,B)`. To je ale v rozporu se zvyklostmi matematického zápisu, proto Prolog dovoluje deklarovat funktory i jako operátory. Poté je možný standardní zápis součtu dvou čísel `A+B`.

Takto jsou v Prologu definovány základní relační operátory: `=`, `\=`, `>`, `<` (rovnost, nerovnost, větší, menší) atd. a aritmetické operátory: `+`, `-`, `*`, `mod` (součet, rozdíl, součin, dělení modulo). Významným operátorem je také operátor přiřazení, značený klíčovým slovem `is` . Přiřazení hodnoty do proměnné na levé straně přiřazovacího operátoru proběhne až poté, co jsou známé hodnoty všech proměnných výrazu na pravé straně operátoru.

Např.: `?- X=2, Y is X-1.`

Odpověď bude `X=2, Y=1.`

3 Programovací jazyk PL/pgSQL

PL/pgSQL je procedurální programovací jazyk určený pro programování uložených funkcí systému PostgreSQL.

3.1 PostgreSQL

PostgreSQL je objektově-relační databázový systém. PostgreSQL se vyvinul z univerzitního výzkumu, v současné době patří mezi nevyspělejší Open Source databázový software. Svými vlastnostmi, výkonem i spolehlivostí konkuruje komerčním softwarům, navíc je k dispozici zcela zdarma [8].

3.2 Uložené funkce

Uložená funkce je kód, který je uložen a prováděn na straně SQL serveru. Všechny klientské aplikace pak tuto funkci mohou jednoduše vyvolat. Pokud je potřeba funkci modifikovat, učiní se tak pouze na jednom místě a všechny klientské aplikace automaticky začnou používat novou verzi. Při složitějších výpočtech klient zasílá dotazy na databázový server, přijímá výsledky, zpracovává je a zasílá další dotazy na server. Uložené funkce umožňují přesunutí celého výpočtu na server. Tím odpadá meziprocesová komunikace i režie sítě a celý výpočet se tak značně urychlí. Uložené funkce ale nelze použít, pokud je během výpočtu vyžadována interakce s uživatelem.

Uložené funkce lze vytvářet v několika jazycích. V základní distribuci jsou to jazyky: SQL, C, PL/pgSQL, PL/Tcl, PL/Perl a PL/Python. Lze použít i mnoho dalších jazyků, které nejsou součástí distribuce PostgreSQL a jsou vyvíjeny externě (např. PL/Java, PL/Php atd.). Pro programování uložených funkcí se nejčastěji používá jazyk PL/pgSQL [7].

3.3 PL/pgSQL

Jazyk PL/pgSQL je procedurální rozšíření jazyku SQL. Datové typy sdílí s databázovým systémem, může používat většinu SQL příkazů. Navíc obsahuje konstrukce

pro větvení programu (IF, CASE), smyčky (WHILE, FOR, LOOP), obsluhu výjimek a další konstrukce. Velmi využívané je použití kurzorů, tedy iterace nad množinou záznamů, které se získají příkazem SELECT. Tento jazyk není case-sensitive, můžeme tedy pro názvy datových typů, proměnných i klíčových slov libovolně používat velká a malá písmena, při kompilaci se vše převede na malá písmena (to samozřejmě neplatí pro řetězcové konstanty) [9]. Pro lepší přehlednost je ale doporučeno psát klíčová slova velkými písmeny a vše ostatní malými.

S každou verzí PostgreSQL se zvětšují možnosti uložených funkcí, příkaz pro vytvoření funkce má již hodně složitou strukturu. Zde si uvedeme jen jednoduchou variantu funkce, která neodráží všechny možnosti, ale ukazuje základní použití jazyka PostreSQL. Zjednodušená struktura je zde:

```
CREATE [OR REPLACE] FUNCTION
     nazev (argument1 typ1, argument2 typ2, ..)
     RETURNS navratovyTyp AS $body$
DECLARE
     <deklarační část - deklarování proměnných>
BEGIN
     <tělo funkce - vlastní příkazy>
END;
$body$ LANGUAGE plpgsql;
```

V hlavičce funkce tedy určíme název funkce, její argumenty včetně datových typů a návratovou hodnotu funkce. V deklarační části můžeme deklarovat proměnné a to ve tvaru: název typ středník. Tělo funkce obsahuje blok příkazů oddělených středníkem. Provádění funkce je vždy ukončeno příkazem RETURN.

Zde je příklad jednoduché funkce, která vrátí absolutní hodnotu součtu dvou čísel:

```
CREATE OR REPLACE FUNCTION
     abssum(a int, b int)
     RETURNS int AS $body$
DECLARE
     sum int;
BEGIN
     sum = a+b;
     IF (sum >= 0) THEN RETURN sum;
     ELSE RETURN -sum;
     END IF;
END;
$body$ LANGUAGE plpgsql;
```

3.4 Kurzory

Uvnitř metod je možné vybírat data z tabulek pomocí příkazu SELECT. Ten vrátí množinu záznamů, které je obvykle potřeba projít a zpracovat. Postupné procházení vybraných záznamů umožňuje tzv. kurzor. Kurzor je svázaný s dotazem, jehož výsledky prochází. Vytvoříme ho v deklarační části funkce.

```
nazev_kurzoru CURSOR FOR <dotaz>
```

Pokud například chceme vytvořit kurzor pro procházení všech záznamů z tabulky kniha, které mají hodnotu atributu `pocet_stran` menší nebo rovnou padesáti, deklarujeme ho následovně:

```
cur1 CURSOR FOR
     SELECT *
     FROM kniha
     WHERE pocet_stran <= 50;
```

V těle funkce pak vytvoříme smyčku, která záznamy projde a umožní jejich zpracování. Na začátku smyčky se vždy aktuální záznam uloží do proměnné, která má vždy stejnou strukturu, jako je struktura záznamu vybraného kurzorem [7]. Přes tuto proměnnou poté přistupujeme k jednotlivým polím záznamu. Syntaxe iterace nad kurzorem je:

```
FOR nazev_promenne IN nazev_kurzoru LOOP
     <příkazy>
END LOOP;
```

Předpokládejme, že tabulka kniha má atribut `oblibena` datového typu boolean. Do knihovny chodí líní čtenáři, kteří mají rádi krátké knihy. Proto všem knihám, jejichž počet stran nepřesáhne 50, nastavíme atribut `oblibena` na true. Počet knih, které se nově stanou oblíbenými, budeme čítat v proměnné `nove_oblibene`.

```
nove_oblibene = 0;
FOR data IN cur1 LOOP
     IF(data.oblibena == false) THEN
          UPDATE kniha
          SET oblibena = true
          WHERE id = data.id;
          nove_oblibene := nove_oblibene + 1;
     END IF;
END LOOP;
```

Nyní vytvoříme funkci `nastav_kratke_na_oblibene()`, která bude využívat výše uvedený kurzor. Vybere všechny knihy, které mají padesát nebo méně stránek. Všechny vybrané knihy projde a neoblíbené změní na oblíbené. Vrátí počet nově oblíbených knih.

```
CREATE OR REPLACE FUNCTION
     nastav_kratke_na_oblibene()
     RETURNS INTEGER AS $body$
DECLARE
     nove_oblibene INTEGER;
     cur1 CURSOR FOR
          SELECT * FROM kniha
          WHERE pocet_stran <= 50;
BEGIN
     nove_oblibene = 0;
     FOR data IN cur1 LOOP
          IF( data.oblibena = false) THEN
               UPDATE kniha
               SET oblibena = true
               WHERE id = data.id;
               nove_oblibene := nove_oblibene + 1;
          END IF;
     END LOOP;
     RETURN nove_oblibene;
END;
$body$ LANGUAGE plpgsql;
```

4 Úvod do teorie grafů

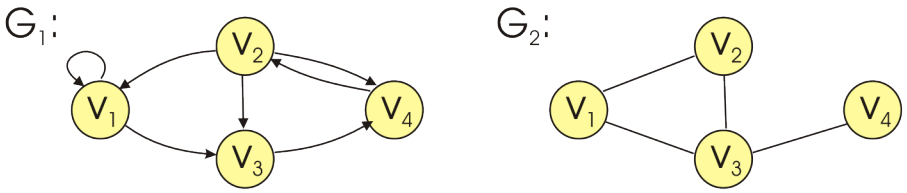
V této kapitole budou uvedeny základní pojmy z teorie grafů a dále různé způsoby prohledávání grafů. V poslední části kapitoly se budeme zabývat hamilotonovskými kružnicemi.

4.1 Základní pojmy

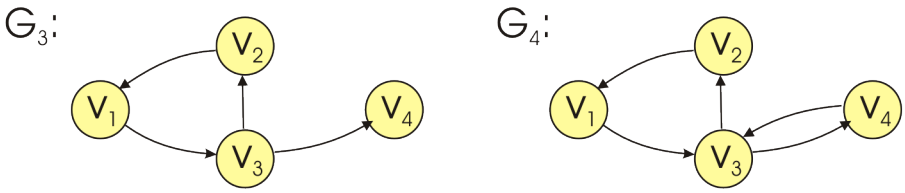
Graf je uspořádaná dvojice G = (V,E), kde V je množina vrcholů a E je množina hran. Graf obvykle znázorňuje vztahy mezi nějakými objekty. Objektům se tedy přiřadí vrcholy a vztahy mezi objekty se znázorní hranami mezi vrcholy. Hrana vždy spojuje dva vrcholy a může být orientovaná nebo neorientovaná. U orientovaných hran se rozlišuje počáteční a koncový vrchol. U neorientovaných hran nezáleží na pořadí vrcholů. Hrana, která spojuje vrchol se sebou samým, se nazývá smyčka.

Orientovaný graf je takový graf, který obsahuje pouze orientované hrany. Neorientovaným grafem nazveme graf, jehož všechny hrany jsou neorientované. Příklad orientovaného grafu G_1 se smyčkou u vrcholu v_1 a neorientovaného grafu G_2 je na obrázku 4.1. Existují i tzv. smíšené grafy, které mají oba druhy hran, v praxi se ale nepoužívají [11, 12, 15].

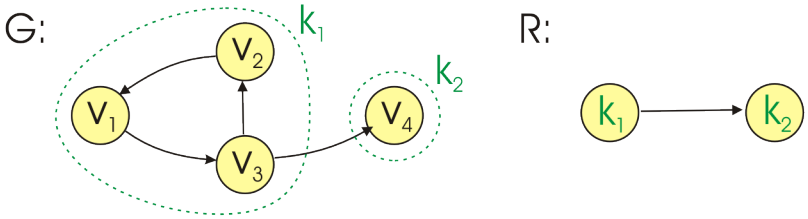
Cesta je taková posloupnost vrcholů, pro kterou platí, že z každého vrcholu vede hrana do jeho následníka a žádný vnitřní vrchol se v ní nevyskytuje vícekrát. Pokud je počáteční a koncový vrchol stejný, jedná se o uzavřenou cestu neboli kružnici [12].



Obrázek 4.1: Orientovaný a neorientovaný graf.



Obrázek 4.2: Silně a slabě souvislý graf



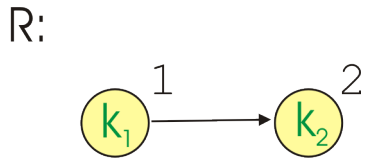
Obrázek 4.3: Graf a redukovaný graf

4.2 Souvislost orientovaných grafů

Orientovaný graf je silně souvislý, pokud se lze z každého vrcholu dostat orientovanou cestou do jakéhokoliv jiného vrcholu. Graf je slabě souvislý, pokud se lze z každého vrcholu neorientovaného grafu, který vznikne symetrizací daného orientovaného grafu, dostat po cestách do jakéhokoliv jiného vrcholu [12]. Na obrázku 4.2 vidíme dva orientované grafy G_3 a G_4. Graf G_3 není silně souvislý, ale je slabě souvislý. Graf G_4 je silně souvislý, takže je zároveň i slabě souvislý.

Maximální silně souvislý podgraf grafu se nazývá kvazikomponenta. V grafu můžeme vyhledat všechny kvazikomponenty a sestrojit tzv. redukovaný graf. Každé kvazikomponentě v původním grafu odpovídá jeden vrchol v redukovaném grafu. Pokud v původním grafu vede hrana z kvazikomponenty k_1 do kvazikomponenty k_2, pak také v redukovaném grafu vede hrana z vrcholu k_1 do vrcholu k_2 [10, 11]. Na obrázku 4.3 je zobrazen původní graf G a jeho redukovaný graf R.

Redukovaný graf je vždy acyklický. Každý acyklický graf lze acyklicky očíslovat. Acyklické číslování je takové očíslování vrcholů grafu (1,2, ... n), že pro každou hranu, která vede z vrcholu v_i do v_j, platí nerovnost $i < j$ [11]. Na obrázku 4.4 je redukovaný graf z předchozího příkladu acyklicky očíslován.



Obrázek 4.4: Acyklicky očíslovaný redukovaný graf

4.3 Prohledávání grafů

Prohledávání grafů je systematický postup k prozkoumání jejich struktury. Prohledáváním postupně projdeme všechny vrcholy. Zde budou uvedeny dva způsoby průchodu grafem – prohledávání do šířky a prohledávání do hloubky.

4.3.1 Prohledávání do šířky

Prohledávání do šířky začíná v jednom vrcholu, dále projde všechny jeho sousedy, pak sousedy jeho sousedů a tak se pokračuje, dokud nenavštíví všechny vrcholy. V každém kroku se tedy projdou vrcholy, které mají stejnou vzdálenost od počátečního vrcholu.

Každý vrchol si uchovává svůj stav – neobjevený, otevřený nebo zavřený. Algoritmus využívá frontu pro uchovávání otevřených vrcholů. Na začátku jsou všechny vrcholy neobjevené. Postup je jednoduchý, začneme z libovolného vrcholu, ten označíme jako otevřený a uložíme ho do fronty. Následující postup se opakuje, dokud není fronta prázdná: Vyjmeme z fronty vrchol. Najdeme všechny vrcholy, do kterých z aktuálního vrcholu vede hrana. Ty z nich, které jsou neobjevené, uložíme do fronty a označíme jako otevřené. Aktuální vrchol označíme jako zavřený.

Pokud jsou všechny vrcholy zavřené, algoritmus končí. V opačném případě vybereme libovolný neobjevený vrchol, ze kterého začneme prohledávat a celý postup opakujeme.

Asymptotická složitost tohoto algoritmu je O(V+H), kde V je počet vrcholů a H počet hran grafu [18].

4.3.2 Prohledávání do hloubky

Tento algoritmus je založen na backtrackingu. Začne se v jednom vrcholu, pokračuje jeho nenavštíveným sousedem. Pokud takový soused neexistuje, vrátí se k předchozímu vrcholu. Postup je velmi podobný jako u prohledávání do šířky, pro uchovávání otevřených vrcholů se ale tentokrát používá zásobník.

Zvolíme si libovolný vrchol, ze kterého začneme prohledávat, označíme ho jako otevřený a vložíme ho do zásobníku. Potom se opakuje následující postup, dokud není zásobník prázdný: Vyjmeme vrchol ze zásobníku, najdeme všechny jeho neobjevené sousedy, označíme je jako otevřené a vložíme je do zásobníku. Zavřeme aktuální vrchol. Pokud graf obsahuje nějaké neobjevené vrcholy, vybereme z nich jeden, ze kterého graf začneme znovu prohledávat.

Jestliže jsou všechny vrcholy zavřené, prohledávání je ukončeno. Asymptotická složitost je stejná jako u prohledávání do šířky, tedy O(V+H).

4.3.3 Tarjanův algoritmus pro hledání kvazikomponent

Tarjanův algoritmus umožňuje vyhledat v grafu všechny kvazikomponenty. Je založen na prohledávání do hloubky, má také stejnou asymptotickou složitost. Stejně jako při prohledávání do hloubky se indexují vrcholy podle pořadí nalezení. Navíc si ještě každý vrchol uchovává nejnižší číslo vrcholu, do kterého se lze dostat po orientované cestě. Toto číslo se zjišťuje při zpětném průchodu uzlu. Všechny vrcholy, které na konci algoritmu mají toto číslo stejné, patří do jedné kvazikomponenty [11, 16].

Algoritmus používá zásobník. Do zásobníku se přidávají vrcholy při jejich prvním navštívení a odebírají se až po uzavření celé kvazikomponenty. Vrcholy z různých kvazikomponent se mezi sebou nikdy nepromíchají, všechny vrcholy jedné kvazikomponenty se tak ze zásobníku odeberou najednou [11].

Zde bude popsán Tarjanův algoritmus v jazyce Java, inspirovaný pseudokódem z webové stránky `Algoritmy.net` [16]. V popisu je pro zkrácení slovo kvazikomponenta nahrazeno slovem komponenta.

Nejprve si definujeme globální proměnné:

```
boolean[][] matrix;
int indexes[];
int lowLinks[];
```

```
List<List<Integer>> components;
Stack<Integer> s;
int i = 0;
```

Dvourozměrné pole `matrix[][]` reprezentuje matici sousednosti grafu. Do pole `indexes[]` se budou ukládat indexy uzlů podle pořadí nalezení. V poli `lowLinks[]` bude vždy na indexu x uložen nejnižší index uzlu, do kterého vede cesta z uzlu x. `Components` obsahuje seznam komponent. Každá komponenta je uložena jako seznam integerů a bude obsahovat indexy vrcholů, které do dané komponenty patří. Proměnná `s` reprezentuje zásobník. Proměnná `i` reprezentuje index a je inicializovaná na nulu.

```
private List<List<Integer>> tarjan(boolean[][] matrix) {
     this.matrix = matrix;
     indexes = new int[matrix.length];
     lowLinks = new int[matrix.length];

     for (int i = 0; i < indexes.length; i++) {
          indexes[i] = -1;
          lowLinks[i] = -1;
     }

     components = new ArrayList<List<Integer>>();
     s = new Stack<Integer>();
     for (int v = 0; v < matrix.length; v++) {
          if (indexes[v] == -1) {
               strongConnect(v);
          }
     }
     return components;
}
```

Dále vytvoříme metodu `tarjan()`. Jako parametr jí předáme matici sousednosti `matrix[][]`. Uložíme ji do globální proměnné. Vytvoříme si pole `indexes[]` a `lowLinks[]` o velikosti odpovídající počtu řádků matice sousednosti. V cyklu poté inicializujeme všechny prvky těchto polí na hodnotu -1. Dále vytvoříme seznam komponent `components` a zásobník `s`. Poté v cyklu projdeme každý vrchol grafu. Pokud vrchol ještě nebyl navštívený (hodnota pole `indexes[]` na daném indexu je -1), zavoláme metodu `strongConnect()` a předáme jí jako parametr index vrcholu v. Poté vrátíme objekt `components` - seznam komponent grafu.

```
private void strongConnect(int v) {
    indexes[v] = i;
    lowLinks[v] = i;
    index++;
    s.push(v);

    for (int i = 0; i < matrix.length; i++) {
        if (matrix[v][i]) {
            int w = i;
            if (indexes[w] == -1) {
                strongConnect(w);
                lowLinks[v]
                    = Math.min(lowLinks[v], lowLinks[w]);
            } else if (s.contains(w)) {
                lowLinks[v]
                    = Math.min(lowLinks[v], indexes[w]);
            }
        }
    }

    if (lowLinks[v] == indexes[v]) {
        List<Integer> actList
            = new ArrayList<Integer>();
        int w;
        do {
            w = s.pop();
            actList.add(w);
        } while (w != v);
        components.add(actList);
    }
}
```

V metodě `strongConnect()` nastavíme vrcholu v index a lowlink na hodnotu i. Hodnotu i tedy uložíme do polí `indexes[]` a `lowLinks[]` na index v. Poté index i inkrementujeme o jedničku a vrchol v vložíme do zásobníku. V cyklu projdeme všechny vrcholy - aktuální zpracovávaný vrchol označíme w. Vrcholy, do kterých vede hrana z vrcholu v a které ještě nebyly navštívené, prohledáme (zavoláme metodu `strongConnect()` s jejich indexem).

Poté vrcholu v aktualizujeme hodnotu lowlink - přiřadíme mu menší z hodnoty lowlink vrcholu v a lowlink vrcholu w. Pokud narazíme na vrchol, který byl navští-

vený, ale nebyl uzavřený, také aktualizujeme vrcholu v hodnotu lowlink - přiřadíme mu menší z hodnoty lowlink vrcholu v a indexu vrcholu w.

Pokud je vrchol v kořenem komponenty, vytvoříme novou komponentu a přidáváme do ní prvky z vrcholu zásobníku, posledním vloženým vrcholem bude vrchol v, poté přidávání skončí. Vytvořenou komponentu vložíme do seznamu komponent.

Na konci algoritmu je v proměnné `components` uložený seznam kvazikomponent grafu.

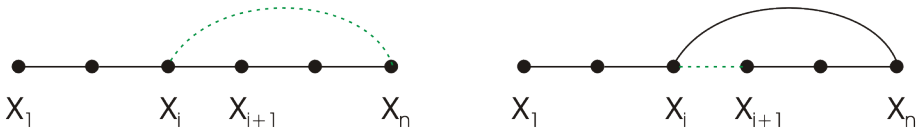
4.4 Hamiltonovské cesty a kružnice

Hamiltonovská cesta je cesta, která prochází všemi vrcholy grafu právě jednou. Obdobně je definována hamiltonovská kružnice – je to kružnice, která prochází všemi vrcholy grafu právě jednou [11]. Graf, který obsahuje hamiltonovskou kružnici nazýváme hamiltonovský graf.

Je známo velké množství nutných podmínek, které graf musí splňovat, aby byl graf hamiltonovský. Například musí být souvislý a každý vrchol musí být nejméně stupně dva [13]. Dále také existují některé postačující podmínky, které zajišťují hamiltonovskost neorientovaného grafu. Pokud je graf úplný (tj. vede hrana mezi každou dvojicí vrcholů), pak je vždy hamiltonovský. Některé další podmínky ukazují, že graf nemusí být úplný, postačuje, když je dostatečně hustý. Diracova podmínka říká: jestliže je počet vrcholů v větší než tři a zároveň má každý vrchol stupeň nejméně $\frac{v}{2}$, pak je graf hamiltonovský [13]. Ještě obecnější je Oreho podmínka: jestliže pro všechny dvojice nesousedních vrcholů platí, že součet jejich stupňů je větší nebo rovný počtu vrcholů grafu, pak je graf hamiltonovský [19].

Nalezení hamiltonovské cesty či kružnice je velmi složité. Hledat můžeme pomocí backtrackingu, tedy upraveného prohledávání do hloubky. Projdeme všechny možné cesty v grafu a budeme zkoumat, jestli některá z nich není hamiltonovskou cestou. Pokud bychom nalezli hamiltonovskou cestu a z koncového do počátečního vrcholu by vedla hrana, získali bychom tak hamiltonovskou kružnici. To je ale velmi časově náročné – backtracking má exponenciální asymptotickou složitost vzhledem k počtu vrcholů grafu. Lze ho tedy použít pouze pro malé grafy [11, 19].

Existují některé heuristické postupy pro nalezení Hamiltonovských cest a kružnic. Tyto algoritmy ale nemusí řešení nalézt, přestože existuje. Nejznámější je Pósova heuristika. Tento algoritmus se snaží prodlužovat cestu postupným přidáváním hran, dokud je to možné. Pokud z koncového vrcholu neexistuje hrana do nějakého volného



Obrázek 4.5: Pósova heuristika

vrcholu, postupujeme následovně: cestu pozměníme přidáním hrany, která vede do vnitřního vrcholu cesty a zároveň zrušením hrany z tohoto vrcholu do vrcholu následujícího. Tímto způsobem otočíme směr části cesty a získáme jiný koncový uzel, z něhož opět zkoušíme cestu prodlužovat. Právě kvůli otočení směru části cesty algoritmus funguje pouze pro neorientované grafy. Na obrázku 4.5 je znázorněna popisovaná úprava cesty [19].

Problém nalezení hamiltonovské kružnice je NP-těžký [11]. Pro nalezení hamiltonovské kružnice v orientovaném grafu existují heuristiky založené například na hledání menších kružnic a jejich následné spojování do větší kružnice. Žádná heuristika ale nezaručuje nalezení hamiltonovské kružnice.

5 Transformace logického programu do PL/pgSQL

V této kapitole bude popsán algoritmus transformace logických pravidel do databázového jazyka PL/pgSQL a vytvoření hlavní funkce, která bude spouštět vyhodnocování dotazu.

Předpokládejme, že v databázi již pro každý predikát existuje tabulka, která mu svým názvem odpovídá. Počet atributů tabulky odpovídá počtu argumentů predikátu. Atributy tabulek mají v názvu znak 'a' a pořadové číslo atributu. Pokud má tedy tabulka čtyři atributy, jejich názvy jsou a1, a2, a3 a a4. Výjimku v tomto označování tvoří magická pravidla, těmi se budeme zabývat v kapitole 5.1.2. Atributy tabulky mají určený datový typ podle charakteru argumentů v logickém pravidle. Vyskytují se v nich pouze dva datové typy, a to číslo (`numeric`) a řetězec neomezené délky (`varchar`). Datovými typy se zde ale dále nebudeme zabývat, protože v algoritmu transformace nehrají žádnou roli. Budeme předpokládat, že atributy tabulky jsou vždy právě takového datového typu, jakou proměnnou budeme potřebovat na danou pozici vložit.

Zároveň předpokládejme, že některé tabulky jsou již naplněny daty. Tato data odpovídají faktům v logickém programu. Pokud se v logickém programu vyskytoval fakt `xml(2,'books',1,1)`, budeme předpokládat, že v databázi existuje tabulka s názvem `xml`, která má čtyři atributy a1, a2, a3, a4 a obsahuje záznam (`2,'books',1,1`).

5.1 Transformace logických pravidel

Ze všech logických pravidel stejného názvu se vygeneruje jedna uložená funkce v PL/pgSQL. Jak již bylo zmíněno, algoritmus se liší pro magická pravidla. Zde bude popsán základní algoritmus a dále jeho modifikace pro magická pravidla.

5.1.1 Generování funkcí – originální algoritmus

Ze všech pravidel se stejným identifikátorem se vytvoří jedna uložená funkce se stejným názvem. Funkce bude vždy obsahovat tolik kurzorů, kolik výskytů pravi-

del tohoto názvu bylo v logickém programu. Každá funkce bude generovat data do tabulky, která bude mít stejný název jako tato funkce a bude vracet počet vygenerovaných záznamů.

Např:

```
intersection(Line1, Line2, Element, 1) :-
xml(Line1, Element, Order, 1),
xml(Line2, Element, Order, 1),
Line1 < Line2.

intersection(Line1, Line2, Element, Level1) :-
xml(Line1, Element, Order, Level1),
xml(Line2, Element, Order, Level1),
level_dec(Level2, Level1),
intersection(Line1, Line2, _, Level2).
```

V příkladu máme predikát s názvem `intersection`, vytvoříme tedy uloženou funkci s tímto názvem. Funkce bude vracet počet záznamů vložených do tabulky. V deklarační části funkce inicializujeme proměnnou `rows`, která bude tyto řádky počítat:

```
CREATE OR REPLACE FUNCTION
     intersection()
     RETURNS INTEGER AS $body$
DECLARE
     rows INTEGER;
```

Nyní ve funkci vytvoříme pro každé vstupní pravidlo jeden kurzor. Kurzory pojmenujeme postupně `cur1`, `cur2`, `cur3` atd. V našem příkladu máme dvě pravidla, funkce tedy bude obsahovat kurzory `cur1` a `cur2`. Začneme s prvním kurzorem `cur1`, který odpovídá prvnímu pravidlu:

```
cur1 CURSOR FOR
```

Za klíčovým slovem `FOR` následuje SQL dotaz. Nejprve se budeme zabývat částí dotazu FROM. Za každý fakt, vyskytující se v těle pravidla, vložíme do této části referenci na tabulku, klíčové slovo `AS` a její alias. Název tabulky odpovídá názvu faktu, alias k němu přidává ještě pořadové číslo (tj. po kolikáté se tato tabulka ve FROM části vyskytuje). V našem příkladě máme v těle pravidla pouze dva fakty, oba s názvem `xml`, proto FROM klauzule bude vypadat následovně:

```
FROM xml AS xml1,
     xml AS xml2
```

Ostatní bloky v těle pravidla (ty, které nejsou fakta - v našem případě blok s aritmetickým výrazem `Line1 < Line2`) prozatím vynecháme, budeme se jimi zabývat až v klauzuli WHERE. Nyní se vrátíme k příkazu SELECT.

Každému faktu v těle pravidla v databázi odpovídá tabulka se sloupci a1, a2, a3 atd. podle počtu argumentů. Argumenty tohoto faktu si představme jako alias k danému sloupci (podle pořadí). V našem příkladě tedy fakt `xml(Line1, Element, Order, 1)` odpovídá tabulce jménem `xml1` se sloupci `a1` alias `Line1`, `a2` alias `Element`, `a3` alias `Order`. Poslední argument je konstanta a odpovídá sloupci `a4`. To samé provedeme se všemi fakty v těle pravidla:

xml1			xml2	
a1	Line1		a1	Line2
a2	Element		a2	Element
a3	Order		a3	Order
a4	1		a4	1

V příkazu SELECT chceme vybrat všechny sloupečky ze všech tabulek, jejichž alias je uveden jako argument v hlavičce pravidla. Všechny názvy aliasů převedeme na malá písmena. Například první argument v hlavičce našeho pravidla je `Line1`. Tento alias jsme přiřadili sloupečku `a1` v tabulce `xml1`, do příkazu SELECT tedy uvedeme `xml1.a1 AS line1`. Jestliže se nám stejný alias vyskytuje ve více tabulkách, můžeme vybrat libovolný z nich. Nakonec se stejně budou rovnat, jak uvidíme ve WHERE klauzuli. Pokud je mezi argumenty konstanta, uvedeme v seznamu pro výběr opět tuto konstantu. V našem případě to bude vypadat následovně:

```
SELECT
        xml1.a1 AS line1,
        xml2.a1 AS line2,
        xml1.a2 AS element,
        1
```

Nyní doplníme klauzuli WHERE. Všechny sloupce, jejichž aliasy se vyskytují ve více tabulkách, musíme ve WHERE klauzuli položit do rovnosti. V našem příkladu se alias Element1 vyskytuje v tabulce `xml1` u sloupce `a2` a v tabulce `xml2` u sloupce `a2`. Ve WHERE klauzuli tedy přibude rovnost `xml1.a2 = xml2.a2` .

Pokud u nějakého atributu tabulky máme uvedenu konstantu, položíme tuto konstantu rovnou hodnotě daného atributu. Například v tabulce `xml1` máme u sloupce `a4` uvedenou konstantu `1`. Přidáme tedy podmínku `xml1.a4 = 1` .

Nyní se budeme zabývat i bloky v pravidlech, které nejsou fakta. V našem příkladu máme aritmetický výraz `Line1 < Line2`. `Line1` odpovídá sloupci `a1` v tabulce `xml1` a `Line2` odpovídá sloupci `a1` v tabulce `xml2`. Přidáme tedy podmínku `xml1.a1 < xml2.a1`.

Celá klauzule WHERE bude tedy vypadat následovně:

```
WHERE
        xml1.a2 = xml2.a2 AND
        xml1.a4 = 1 AND
        xml2.a4 = 1 AND
        xml1.a3 = xml2.a3 AND
        xml1.a1 < xml2.a1
```

Blok v pravidlech může také obsahovat přiřazení (klíčové slovo `IS`) nebo negaci (klíčové slovo `NOT`). Těmito speciálními případy se budeme zabývat ve zvláštní kapitole.

V tabulce chceme mít pouze unikátní data. Abychom se vyhnuli vkládání duplicitních dat, využijeme množinový operátor mínus (`EXCEPT`) a od záznamů získaných předchozím dotazem odečteme záznamy, které se již v tabulce nacházejí:

```
EXCEPT
        SELECT *
        FROM intersection;
```

Tím máme hotovou deklaraci prvního kurzoru. Druhý kurzor vytvoříme podle stejného algoritmu. Jediné, co by nás mohlo zmást je podtržítko místo názvu třetího argumentu pravidla. Podtržítko značí tzv. anonymní proměnnou. Ta se používá v případě, kdy nám na hodnotě dané proměnné nezáleží. Zde je tedy deklarace druhého kurzoru `cur2`:

```
cur2 CURSOR FOR
     SELECT
          xml1.a1 AS line1,
          xml2.a1 AS line2,
          xml1.a2 AS element,
          xml1.a4 AS level1
     FROM
```

```
        xml AS xml1,
        xml AS xml2,
        level_dec AS level_dec1,
        intersection AS intersection1
    WHERE
        xml1.a1 = intersection1.a1 AND
        xml1.a2 = xml2.a2 AND
        xml1.a3 = xml2.a3 AND
        xml1.a4 = xml2.a4 AND
        xml1.a4 = level_dec1.a2 AND
        xml2.a1 = intersection1.a2 AND
        level_dec1.a1 = intersection1.a4
    EXCEPT
        SELECT *
        FROM intersection;
```

Zde končí deklarační část funkce, nyní vytvoříme tělo funkce a inicializujeme proměnnou `rows` na hodnotu nula.

```
BEGIN rows := 0;
```

Nyní zpracujeme data určená kurzory. Ve smyčce projdeme výsledky dotazu, který je s každým kurzorem svázán. Každý záznam se nejprve uloží do proměnné `data` a poté všechny hodnoty uložíme do tabulky odpovídající názvu funkce. Struktura proměnné data odpovídá struktuře záznamu získaného dotazem spjatým s kurzorem. V našem příkladu v kurzoru `cur1` vybíráme dotazem `SELECT` hodnoty: `line1`, `line2`, `element` a `1`. Hodnoty vkládané do tabulky z proměnné data budou tedy: `data.line1`, `data.line2`, `data.element` a konstanta `1`.

Po každém vložení dat do tabulky inkrementujeme proměnnou `rows` o jedničku. Celý cyklus odpovídající prvnímu kurzoru bude vypadat následovně:

```
FOR data IN cur1 LOOP
    INSERT INTO intersection
    VALUES ( data.line1, data.line2, data.element, 1);
    rows := rows + 1;
END LOOP;
```

Obdobně vytvoříme i cyklus odpovídají druhému kurzoru `cur2`:

```
FOR data IN cur2 LOOP
    INSERT INTO intersection
    VALUES ( data.line1, data.line2,
```

```
                data.element, data.level1);
     rows := rows + 1;
END LOOP;
```

Nakonec vrátíme počet vygenerovaných řádek a funkci ukončíme.

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

Celá PL/pgSQL funkce vygenerovaná z logického pravidla v příkladu tedy bude vypadat takto:

```
CREATE OR REPLACE FUNCTION
     intersection()
     RETURNS INTEGER AS $body$
DECLARE
     rows INTEGER;

cur1 CURSOR FOR
     SELECT
          xml1.a1 AS line1,
          xml2.a1 AS line2,
          xml1.a2 AS element,
          1
     FROM xml AS xml1,
          xml AS xml2
     WHERE
          xml1.a2 = xml2.a2 AND
          xml1.a4 = 1 AND
          xml2.a4 = 1 AND
          xml1.a3 = xml2.a3 AND
          xml1.a1 < xml2.a1
     EXCEPT
          SELECT *
          FROM intersection;

cur2 CURSOR FOR
     SELECT
          xml1.a1 AS line1,
          xml2.a1 AS line2,
          xml1.a2 AS element,
```

```
          xml1.a4 AS level1
     FROM
          xml AS xml1,
          xml AS xml2,
          level_dec AS level_dec1,
          intersection AS intersection1
     WHERE
          xml1.a1 = intersection1.a1 AND
          xml1.a2 = xml2.a2 AND
          xml1.a3 = xml2.a3 AND
          xml1.a4 = xml2.a4 AND
          xml1.a4 = level_dec1.a2 AND
          xml2.a1 = intersection1.a2 AND
          level_dec1.a1 = intersection1.a4
     EXCEPT
          SELECT *
          FROM intersection;

BEGIN rows := 0;
     FOR data IN cur1 LOOP
          INSERT INTO intersection
          VALUES ( data.line1, data.line2,
               data.element, 1);
          rows := rows + 1;
     END LOOP;

     FOR data IN cur2 LOOP
          INSERT INTO intersection
          VALUES ( data.line1, data.line2,
               data.element, data.level1);
          rows := rows + 1;
     END LOOP;

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

Negace a vyčíslení

Tělo pravidel může kromě faktů a aritmetických výrazů obsahovat i negace (klíčové slovo `NOT`) a vyčíslení (klíčové slovo `IS`). Zde uvedeme příklad pravidla, které tyto klauzule obsahuje a postup, jak je převést do jazyka PL/pgSQL.

```
level_inc(Level2, Level1) :-
     level(Level1),
     xml(Line, Element, _, Level1),
     Level2 is Level1 + 1,
     not (xml(Line, _, _, Level1)).
```

Postup při vytváření funkce je shodný jako v minulém případě. Vytvoříme funkci se stejným názvem, jako je název predikátu a deklarujeme potřebné proměnné.

```
CREATE OR REPLACE FUNCTION
     level_inc()
     RETURNS INTEGER AS $body$
DECLARE
     rows INTEGER;
     cur1 CURSOR FOR
```

V příkazu SELECT vybíráme všechny sloupečky ze všech tabulek, jejichž alias je uveden jako argument v hlavičce pravidla. Argument `Level1` odpovídá atributu `a1` v tabulce `level1`. Argument `Level2` je v logickém pravidle vyčíslen jako `Level1 + 1`. Proto v příkazu SELECT přiřadíme alias `level2` výrazu `level1.a1 + 1`.

```
SELECT
     level1.a1 + 1 AS level2,
     level1.a1 AS level1
FROM
     level AS level1,
     xml AS xml1
```

Ve WHERE klauzuli kromě klasického porovnávání musíme zohlednit i negaci, tedy v našem příkladu `not (xml(Line, _, _, Level1))`. Negaci do databázové funkce promítneme pomocí klíčových slov `NOT EXISTS`, za kterými bude následovat výběr z databáze. Vybírat budeme celý záznam z tabulky, která svým názvem odpovídá názvu faktu uvnitř negace. V našem případě tedy z tabulky `xml`. V tabulce `xml` odpovídá argument `Line` sloupci `a1` a argument `Level1` sloupci `a4`. Ve WHERE klauzuli tohoto vnořeného výběru musíme opět porovnávat. Argument `Line` se již dříve vyskytovat v tabulce `xml1` jako sloupec `a1`, argument `Level1` se vyskytoval

v tabulce `Level` jako sloupec `a1`. Celá WHERE klauzule tedy bude vypadat následovně:

```
WHERE
     level1.a1 = xml1.a4 AND
     NOT EXISTS (
          SELECT *
          FROM xml
          WHERE
               xml.a1 = xml1.a1 AND
               xml.a4 = level1.a1)
```

Další pokračování funkce se generuje stejně, jako v předchozím příkladu:

```
EXCEPT
     SELECT *
     FROM level_inc;
BEGIN
     rows := 0;
     FOR data IN cur1 LOOP
          INSERT INTO level_inc
          VALUES ( data.level2, data.level1);
          rows := rows + 1;
     END LOOP;
     RETURN rows;
END;
$body$ LANGUAGE plpgsql;
```

5.1.2 Generování funkcí z magických pravidel

Výše je popsána transformace klasického vstupního logického programu. Dále mu budeme říkat „originální program“. Kromě originálního programu může být na vstupu i takzvaný magický program. Ten vznikne optimalizací originálního logického programu Metodou magických množin. Tuto metodu popisuje Martin Zíma ve své disertační práci [5].

Magický program využívá toho, že se v argumentech cílového dotazu nacházejí konstanty. Tyto konstanty se promítnou i do pravidel programu. Výsledek cílového dotazu je ekvivalentní s výsledkem v originálním programu. Díky optimalizaci je ale výsledek získán rychleji.

Metoda magických množin má několik fází. První z nich se nazývá zdobení programu. Predikát, který má za svým názvem podtržítko a několik dalších znaků, se nazývá ozdobený. Tyto znaky jsou buď „b“ nebo „f“ a budeme jim říkat řetězec ozdobení. Řetězec ozdobení má tolik znaků, kolik má predikát argumentů. Tyto znaky určují, které argumenty jsou volné (f jako free) a které jsou vázané (b jako bound).

Z ozdobených predikátů se poté generují magické predikáty. Jejich název začíná znakem m a podtržítkem, zbytek názvu je stejný jako u ozdobeného predikátu. Příkladem názvu magického pravidla je `m_intersection_bbfb.` Magický predikát přebírá od ozdobeného predikátu pouze vázané argumenty, všechny volné argumenty se vypustí. Magický predikát má tedy tolik argumentů, kolik je znaků 'b' v jeho řetězci ozdobení. Z tohoto důvodu dochází při zpracování magických pravidel k malé úpravě algoritmu.

V originální verzi algoritmu jsou se sloupce tabulky číslují podle pořadí – má-li predikát v těle pravidla čtyři argumenty, pak má odpovídající tabulka čtyři sloupečky `a1`, `a2`, `a3` a `a4`, které argumentům odpovídají. Při zpracování magického pravidla např. `m_intersection_bbfb(Line1, Line2, Level2)` ale záleží na řetězci ozdobení. Ten je v tomto případě bbfb a označuje, že původní třetí argument byl volný a byl tedy vypuštěn. Argumenty u tohoto magického pravidla byly původně na pozicích 1, 2 a 4. To znamená, že tabulka obsahuje sloupečky `a1`, `a2` a `a4`, které podle pořadí odpovídají argumentům.

5.2 Generování hlavní funkce

Hlavní funkce volá v cyklu všechny vygenerované funkce tak dlouho, dokud se budou do tabulek zapisovat nějaká data. Pokud již žádná volaná funkce data negeneruje, hlavní funkce se ukončí.

Pro každý soubor pravidel v logickém programu se vytváří tři hlavní funkce: `main_abc()`, `main_zyx()` a `main_clever()`. Každá volá funkce v jiném pořadí. První dvě volají funkce v abecedním pořadí, chytrá hlavní funkce `main_clever()` zohledňuje vzájemné závislosti mezi funkcemi. Předpokládá se, že výsledek i počet vygenerovaných řádek bude ve všech případech stejný a také to, že chytrá hlavní funkce `main_clever()` bude o něco rychlejší. Jestli je tento předpoklad správný, se dozvíme v kapitole 7.

Nejdříve se budeme zabývat hlavními funkcemi `main_abc()` a `main_zyx()`, které volají funkce v abecedním pořadí. Vytvoříme funkci s daným názvem a deklarujeme si dvě proměnné. Do proměnná `rows` budeme ukládat celkový počet vygenerovaných

řádek. Proměnná `loop_rows` bude uchovávat počet vygenerovaných řádek v aktuálním cyklu.

```
CREATE OR REPLACE FUNCTION
     main_abc()
     RETURNS INTEGER AS $body$
DECLARE
     rows INTEGER;
     loop_rows INTEGER;
```

Poté vytvoříme tělo funkce. Proměnnou `rows` inicializujeme na nulu. Poté zahájíme cyklus a uvnitř cyklu inicializujeme na nulu proměnnou `loop_rows`. Dále budeme volat jednotlivé funkce. Pořadí volání funkcí bude v hlavní funkci `main_abc()` abecední. V hlavní funkci `main_zyx()` budou funkce volány v invertovaném abecedním pořadí, tedy abecedně pozpátku.

Každá funkce vrací počet vygenerovaných řádek, tuto hodnotu budeme přičítat do proměnné `loop_rows`. Cyklus se ukončí, když na konci cyklu bude tato proměnná nulová. V opačném případě se k proměnné `rows` přičte hodnota proměnné `loop_rows`. Po skončení smyčky funkce vrátí celkový počet vygenerovaných řádek, tedy proměnnou `rows`, a ukončí se.

Tělo funkce `main_abc()` bude vypadat následovně:

```
BEGIN
     rows := 0;
     LOOP loop_rows = 0;
          loop_rows := loop_rows + child_fffb();
          loop_rows := loop_rows + intersection_bbfb();
          loop_rows := loop_rows + intersection_fbff();
          loop_rows := loop_rows + level_dec_fb();
          loop_rows := loop_rows + level_inc_fb();
          loop_rows := loop_rows + m_intersection_bbfb();
          loop_rows := loop_rows + m_intersection_fbff();
          loop_rows := loop_rows + m_level_dec_fb();
          loop_rows := loop_rows + m_level_inc_fb();
          loop_rows := loop_rows + m_self_bb();
          loop_rows := loop_rows + m_self_fb();
          loop_rows := loop_rows + self_bb();
          loop_rows := loop_rows + self_fb();
          EXIT WHEN loop_rows = 0;
          rows := rows + loop_rows;
     END LOOP;
```

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

Funkce `main_zyx()` by se lišila pouze v pořadí řádek uvnitř smyčky – funkce by byly volány v opačném pořadí.

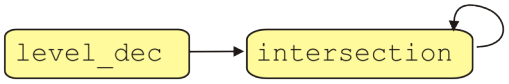
5.2.1 Generování chytré hlavní funkce

Chytrá hlavní funkce `main_clever()` bude zohledňovat vzájemné vztahy funkcí. Jako příklad vezměme nám již známou funkci `intersection()` (viz výše). Konkrétně nás bude zajímat, jaká data funkce vybírá, připomeňme si tedy klauzule FROM v deklaraci kurzorů.

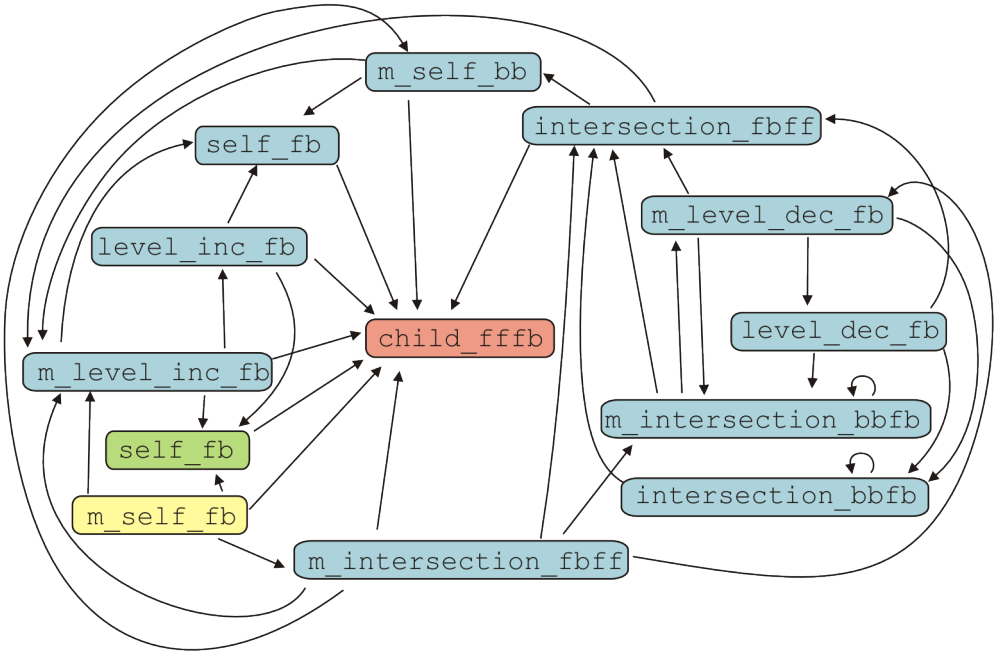
```
  cur1:
FROM
    xml AS xml1,
    xml AS xml2

  cur2:
 FROM
    intersection AS intersection1,
    level_dec AS level_dec1,
    xml AS xml1,
    xml AS xml2
```

Funkce `intersection()` tedy vybírá některá data z tabulek `xml`, `intersection` a `level_dec` a ukládá je do tabulky `intersection`. Žádná tabulka nikdy nebude obsahovat duplicitní data (viz popis množinového operátoru EXCEPT). Z tohoto důvodu je jasné, že pokud se nebudou měnit data v tabulkách `xml`, `intersection` a `level_dec`, nebude se měnit ani tabulka `intersection`. Dá se tedy říci, že funkce `intersection()` je závislá na tabulkách `xml`, `intersection` a `level_dec`.

Existují dvě možnosti, jak se data do tabulky dostanou. První možností je naplnění tabulek na začátku, tj. před spuštěním hlavních funkcí na základě logických faktů. Tímto způsobem se plní tabulky `xml`, `data` a `atribut`. Tato data se už v průběhu programu nemění a funkce s tímto názvem neexistují. Druhou možností je, že v průběhu programu vygeneruje data do tabulky stejnojmenná funkce. V prů-



Obrázek 5.1: Graf závislosti funkcí

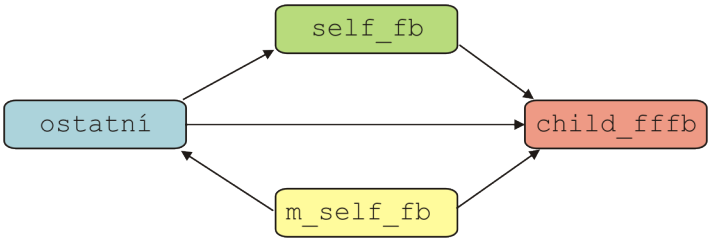


Obrázek 5.2: Graf závislosti funkcí programu `child_fffb.sql`

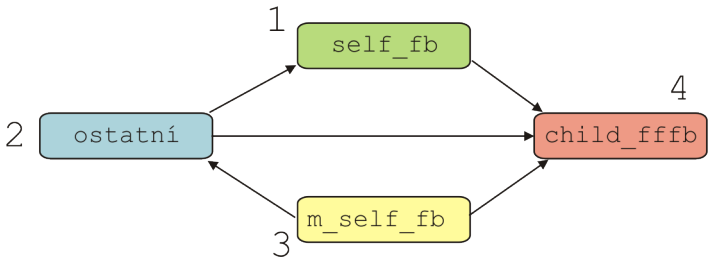
běhu programu tedy může tabulku `level_dec` změnit pouze funkce `level_dec()` a tabulku `intersection` funkce `intersection()`.

Funkce `intersection()` je tedy závislá na funkci `level_dec()`. Zároveň je závislá sama na sobě, takové funkci říkáme rekurzivní funkce. Závislosti mezi funkcemi můžeme zaznamenat pomocí orientovaného grafu. Na obrázku 5.1 jsou vyjádřeny popisované závislosti.

Na obrázku 5.2 je graf závislostí funkcí, které byly vygenerovány ze zdrojového souboru `child_fffb.pro`. Vstupní program v prologu a vygenerovaný SQL skript jsou k dispozici na přiloženém DVD ve složce `tests\child\magic`. Tento graf je souvislý. Není ale silně souvislý, lze ho tedy rozdělit na kvazikomponenty. Každá kvazikomponenta je na obrázku 5.2 označena jednou barvou. Redukovaný graf kvazikomponent je na obrázku 5.3.



Obrázek 5.3: Graf kvazikomponent



Obrázek 5.4: Acyklicky očíslovaný graf kvazikomponent

Graf má čtyři kvazikomponenty. Tři z nich obsahují pouze jeden vrchol, čtvrtá obsahuje všechny zbylé vrcholy. Graf kvazikomponent je ze své podstaty acyklický, lze jej tedy acyklicky očíslovat. Acyklicky očíslovaný graf kvazikomponent je na obrázku 5.4. Acyklicky očíslovaný graf kvazikomponent nám ukazuje, jak jsou jednotlivé skupiny funkcí na sobě závislé. To snadno aplikujeme v naší chytré hlavní funkci.

Pro každou kvazikomponentu vytvoříme vlastní cyklus. Uvnitř cyklu zavoláme všechny funkce, které obsahuje daná kvazikomponenta. Tyto cykly za sebe poskládáme v pořadí acyklického číslování. Pokud nějaká kvazikomponenta obsahuje pouze jednu funkci, která není rekurzivní (není závislá sama na sobě), nebudeme tuto funkci vkládat do cyklu, ale zavoláme ji pouze jednou. Těmito úpravami si ušetříme zbytečné volání funkcí, které už negenerují žádná data.

Pořadí volání funkcí uvnitř kvazikomponent

Každá kvazikomponenta je silně souvislý graf. Abychom eliminovali zbytečné volání funkcí, které aktuálně negenerují žádná data, bylo by potřeba funkce volat co nejvíce „popořadě“. Ideální by bylo najít v každé kvazikomponentě Hamiltonovskou kružnici a funkce volat popořadě tak, jak na kružnici leží. Problém je, že hledání Hamiltonovské kružnice je algoritmicky velmi náročné (je to NP-úplný problém), navíc ji

kvazikomponenta vůbec nemusí obsahovat. Bylo tedy nutné najít nějaké univerzální a algoritmicky jednodušší řešení.

Vhodným algoritmem se jeví prohledávání do hloubky – respektuje závislosti funkcí v grafu a je velmi rychlý. Nejprve zavoláme funkce s přímou rekurzí. Zbytek kvazikomponenty prohledáme do hloubky a funkce seřadíme podle pořadí při prohledávání.

```
CREATE OR REPLACE FUNCTION
     main_clever()
     RETURNS INTEGER AS $body$
DECLARE
     rows INTEGER;
     loop_rows INTEGER;
BEGIN
     rows := 0;
     rows := rows + m_self_fb();
     LOOP
          loop_rows = 0;
          loop_rows := loop_rows + m_intersection_bbfb();
          loop_rows := loop_rows + intersection_bbfb();
          loop_rows := loop_rows + m_self_bb();
          loop_rows := loop_rows + m_level_inc_fb();
          loop_rows := loop_rows + level_inc_fb();
          loop_rows := loop_rows + self_fb();
          loop_rows := loop_rows + m_intersection_fbff();
          loop_rows := loop_rows + m_level_dec_fb();
          loop_rows := loop_rows + level_dec_fb();
          loop_rows := loop_rows + intersection_fbff();
          EXIT WHEN loop_rows = 0;
          rows := rows + loop_rows;
     END LOOP;
     rows := rows + self_bb();
     rows := rows + child_fffb();
     RETURN rows;
END;
$body$ LANGUAGE plpgsql;
```

6 Realizace systému

V této kapitole bude popsána realizace aplikace PrologToPostgreSQL, který provádí transformaci logických pravidlech do uložených objektů PL/pgSQL.

6.1 Zadání

Úkolem praktické části této diplomové práce je vytvořit program, který bude přepisovat logická pravidla do uložených funkcí databáze a generovaná data pravidel do odpovídajících tabulek (dle algoritmu popsaného dříve). Pro implementaci programu byl zadavatelem určen programovací jazyk Java.

Program bude součástí většího celku, není tedy nutné vytvářet grafické uživatelské rozhraní. Bude se tedy jednat o konzolovou aplikaci, kterou bude možné spustit s parametry z příkazové řádky nebo využít API[1] pro volání metod z jiného programu.

Vstupem programu bude textový soubor s logickými pravidly. Výstupem bude soubor s uloženými funkcemi v PL/pgSQL nebo se vygenerované funkce přímo uloží do databáze.

6.2 Architektura

V realizaci systému byla použita dvouvrstvá architektura. Datová vrstva vytváří vstupní a výstupní proudy, readery a writery. Aplikační vrstva tvoří jádro implementace, zajišťuje zpracování vstupních souborů, transformaci logických pravidel do PL/PgSQL, vytvoření hlavních funkcí a předání výsledného textu do výstupního proudu.

Aplikace je členěna do balíků podle toho, jak třídy tvoří logické celky. V balíku `cz.zcu.dip.exceptions` jsou obsaženy uživatelsky definované vyjímky. Balík `cz.zcu.dip.io` tvoří datovou vrstvu aplikace, která se stará o vstupní a výstupní proudy. Ostatní balíky tvoří aplikační vrstvu. Balík `cz.zcu.dip.preparation` se stará o přípravu na samotnou transformaci – o parsování argumentů příkazové řádky a vymazání komentářů ze vstupního souboru. V balíku `cz.zcu.dip.translator` už

[1] API - Application Programming Interface, česky programové rozhraní aplikace

probíhá samotná transformace. Metoda `runTransformation()` ze třídy `PrologToPostgreSql` je jakýmsi vstupním bodem transformace, pokud se tedy transformace bude spouštět z jiného programu, doporučuji využít právě tuto metodu. Generování hlavních funkcí probíhá v balíku `cz.zcu.dip.translator.mainFunc`.

6.3 Knihovna Java Prolog Parser

Vstupní soubor obsahuje pravidla v logickém programovacím jazyce. Je tedy nutné provést syntaktickou analýzu a převést vstupní text do datových struktur. Pro usnadnění tohoto procesu byla použita knihovna Java Prolog Parser[2].

Knihovna byla do verze 1.3.1 šířena pod licencí GNU Lesser General Public License. Tato licence aplikuje na program copyleftové restrikce, tedy jakákoliv modifikace programu musí být šířena opět pod touto licencí. Pokud ale program pouze používá knihovnu, nejedná se o odvozené dílo a licence se na něj nevztahuje [20]. Starší verze knihovny lze stáhnout včetně zdrojových kódů a dokumentace na domovských stránkách knihovny.

Od verze 1.3.2 , která byla vydána v únoru 2014, je již knihovna šířena pod licencí Apache 2.0. Jedná se tedy o svobodný software. Po uživateli se požaduje zachování autorství, ale neklade se omezení na licenci, pod kterou se bude odvozené dílo šířit [21]. Tuto verzi knihovny již nelze stáhnout na domovských stránkách knihovny. Podle slov autora knihovny Google zakázal publikování binárních souborů v Google Code, proto lze nyní knihovnu získat v centrálním Maven repozitáři[3]. Lze zde získat také zdrojové soubory a dokumentaci. V aplikaci byla použita tato nejnovější verze knihovny Java Prolog Parser.

Knihovna umožňuje parsovat program psaný v Prologu v Edinburgském stylu. Na základě načtených údajů je vytvořena struktura objektů, které reprezentují původní program. V této struktuře se lze snadno pohybovat, lze ji analyzovat a číst potřebné údaje.

Edinburgský styl umožňuje používat pouze jednořádkové komentáře (řádka začíná znakem `%`). Vstupní soubory ale mohou obsahovat i blokové komentáře (ve tvaru `/* komentář */`). V průběhu transformace logických pravidel do PL/pgSQL se komentáře ignorují, nemají žádný vliv na podobu výsledného programu. Nabízí

[2]Domovské stránky této knihovny jsou na adrese `http://code.google.com/p/java-prolog-parser` .

[3]Konkrétně na adrese `http://search.maven.org/remotecontent?filepath=com/igormaznitsa/prologparser/1.3.2/prologparser-1.3.2.jar`

se tedy snadné řešení – nejprve odstraníme komentáře ze vstupního souboru, poté použijeme knihovnu Java Prolog Parser.

6.4 Popis programu PrologToPostgreSQL

V této kapitole bude podrobně popsán celý průběh programu, budou zde popsány důležité třídy a metody aplikace.

6.4.1 Argumenty příkazové řádky

Program neobsahuje žádné uživatelské rozhraní. Spouští se z příkazové řádky spolu s několika argumenty (viz uživatelská příručka). Hlavní třída `ArgParser` obsahuje jedinou metodu `main()`. Ta zajišťuje parsování argumentů příkazové řádky. Uloží si tedy do příslušných proměnných jméno vstupního souboru, jméno výstupního souboru a nastaví logické proměnné, které určují, zda se má vstupní program zpracovávat jako magický a zda se má generovat chytrá hlavní funkce. Poté s těmito parametry pouze zavolá metodu `runTransformation()` ze třídy `PrologToPostgreSql`.

6.4.2 Transformace

Metoda `runTransformation()` ze třídy `PrologToPostgreSql` zajišťuje celý proces transformace logického programu do PL/pgSQL. Postupně volá metody z ostatních tříd, které vykonávají dílčí úlohy. Pokud je transformace úspěšná, vytvoří se výstupní soubor s daným jménem a uloží se do něj funkce v PL/pgSQL.

Metoda má osm parametrů. Protože tato metoda se pravděpodobně bude volat z dalších aplikací, budou zde parametry podrobně popsány.

String inputFilePath Absolutní nebo relativní cesta ke vstupnímu souboru.

String outputFilePath Cesta k výstupnímu souboru. Pokud má hodnotu null, nebude se výstupní soubor generovat.

boolean magicON Pokud má hodnotu true, bude se vstupní program zpracovávat jako magický. Při hodnotě false se bude zpracovávat originálním způsobem.

boolean cleverMainON Pokud má tato proměnná hodnotu true, bude kromě abecedních hlavních funkcí vygenerována i chytrá hlavní funkce.

boolean dataON Pokud má hodnotu true, budou se ze vstupního souboru kromě logických pravidel zpracovávat i logická fakta.

String url URL výstupní databáze. Pokud má hodnotu null, nebude se výstup do databáze zapisovat.

String user Uživatel databáze.

String password Heslo daného uživatele do databáze.

Dále bude podrobněji popsán celý průběh transformace.

6.4.3 Odstranění blokových komentářů

Nejprve je nutné zbavit se blokových komentářů ze vstupního souboru, protože takovéto komentáře neumí zpracovávat použitá knihovna Java Prolog Parser. K tomu se použije metoda `getFileWithoutComments()` z třídy `CommentRemover`.

Nejprve se vytvoří dočasný soubor, do kterého se bude kopírovat text ze vstupního souboru kromě blokových komentářů. Je nutné, aby jméno dočasného souboru bylo unikátní, abychom případně nepřepsali jiný již existující soubor. Jméno souboru je tedy tvořeno takto:

```
jmenoVstupnihoSouboru_aktualniCas.temp
```

Při vytvoření souboru se navíc testuje, zda již takový soubor neexistuje. Pokud by náhodou existoval, vygeneruje se nové jméno souboru.

Poté se otevře vstupní soubor pro čtení a dočasný soubor pro zápis. Poté se čte vstupní soubor znak po znaku (to probíhá v metodě `stripFile()`) a hledá se sekvence znaků `/*` , která znamená začátek blokového komentáře. Dokud tato sekvence není nalezena, všechny přečtené znaky se zapisují do dočasného souboru. Po nalezení začátku komentáře se přestanou znaky zapisovat, pouze se dále čtou ze vstupního souboru a hledá se sekvence znaků `*/`, která znamená konec komentáře. Po přečtení celého vstupního souboru se uzavřou vstupní i výstupní proudy a vrátí se vytvořený dočasný soubor.

6.4.4 Načtení struktury vstupního souboru

Od této chvíle budeme jako vstupní soubor označovat dočasný soubor vytvořený v předchozím kroku. Jedná se o kopii původního souboru, která ale neobsahuje blokové komentáře.

Struktura vstupního logického programu se načte do datových struktur, se kterými se bude v dalších krocích dobře pracovat. Připravíme si mapu (tabulku) `rulesMap` a seznam `dataList`. Mapa `rulesMap` bude mít klíče datového typu String, které označují predikátový symbol (identifikátor) pravidla a hodnoty budou objekty třídy `RulesWithEqualId`. Třída `RulesWithEqualId` obsahuje seznam pravidel (objekty třídy Rule) se stejným predikátovým symbolem a několik metod pro jejich zpracování. Seznam `dataList` bude obsahovat objekty třídy Data.

Když je připravena tato struktura, otevře se vstupní soubor pro čtení. V cyklu se začnou načítat logická fakta a pravidla s pomocí knihovny Java Prolog Parser. Každý fakt nebo pravidlo se načte do objektu třídy `PrologStructure` (knihovní třída).

Pokud se jedná o fakt, vytvoříme objekt třídy `Data`, v konstruktoru mu předáme načtenou strukturu. Do proměnné `predicatName` si uložíme název predikátu, do seznamu řetězců `valuesList` si v cyklu načteme všechny argumenty predikátu. Celý objekt uložíme do seznamu `dataList`.

Pokud se jedná o pravidlo, zjistíme identifikátor této struktury, tedy predikátový symbol pravidla. Pokud takový identifikátor ještě neexistuje v mapě `rulesMap`, přidáme ho jako klíč a jako hodnotu vytvoříme nový objekt třídy `RulesWithEqualId`. Vytvoříme nové pravidlo, tedy nový objekt třídy Rule a v konstruktoru mu předáme strukturu (objekt třídy `PrologStructure`). V mapě podle identifikátoru najdeme správný objekt typu `RulesWithEqualId` a pomocí metody `addRule()` do něj přidáme vytvořené pravidlo. Uvnitř této metody se pravidlo přidá do seznamu `rules` a pokud se jedná o pravidlo s přímou rekurzí, nastaví se proměnná `recursive` na true. Tímto způsobem se zpracují všechna pravidla ze vstupního souboru.

6.4.5 Parsování pravidel

Nad každým přidaným pravidlem zavoláme metodu `parseStructure()` z třídy Rule. Tato metoda projde strukturu pravidla, která mu byla předána v konstruktoru a do vlastních datových struktur si uloží důležité informace.

Vytvoří se mapa alias. V této mapě bude potřeba mít prvky seřazené v tom pořadí, ve kterém byly vkládány, proto se bude jednat o `LinkedHashMap`. Klíče budou datového typu String a hodnoty budou objekty třídy `List<String>` . Klíče tvoří názvy logických proměnných, které poté v uložených funkcích PL/pgSQL tvoří aliasy k atributům tabulky. Hodnotu ke každému klíči tvoří seznam řetězců, které v uložené funkci budou mít daný alias. Do mapy přidáme jako klíče všechny argumenty pravidla (tj. termy v hlavě pravidla).

Vytvoříme další důležité objekty, do kterých se budou ukládat informace o pravidlech. Vytvoříme dva seznamy objektů typu `PrologStructure` , do kterých se budou ukládat struktury, které se budou zpracovávat až později. První seznam s názvem `operatorsList` bude uchovávat struktury, které představují aritmetické výrazy. Do seznamu `negationsList` se budou ukládat predikáty z těla pravidla, které jsou v negaci. Dále vytvoříme mapu `facts`, která bude mít klíče typu String a hodnoty typu integer. Jako klíče se budou ukládat všechny predikátové symboly z těla pravidla a jako hodnota se bude ukládat počet, kolikrát se v těle pravidla vyskytly.

Poté v cyklu projdeme všechny bloky v těle pravidla a pro každý z nich zavoláme metodu `processTerm()`. Zpracování bloku se liší podle toho, zda se jedná o predikát nebo o aritmetický výraz.

Predikát

Načteme predikátový symbol. Pokud se jedná o negaci, přidáme strukturu do seznamu `negationsList` a přejdeme k dalšímu bloku. Pokud se o negaci nejedná, uložíme predikátový symbol jako klíč do mapy `facts` a započteme jeho výskyt inkrementací hodnoty odpovídají tomuto klíči.

Pokud se jedná o magické pravidlo a zároveň je aktivní magické zpracovávání pravidel, vytvoříme pole `magicArr`. Jedná se o pole integerů, jehož velikost udává počet vázaných argumentů a hodnoty určují na jakých indexech se vázané argumenty nachází. Pokud tedy bude řetězec ozdobení fffb, `magicArr` bude mít tvar `[4]` – jediný vázaný argument se nachází na čtvrté pozici. Pro řetězec ozdobení bbfb bude `magicArr [1,2,4]` – vázané argumenty jsou na první, druhé a čtvrté pozici.

Nyní vytvoříme alias jména tabulky. Jméno tabulky odpovídá predikátovému symbolu. Alias jména tabulky tvoří predikátový symbol spolu s indexem, který určuje, po kolikáté se tento predikátový symbol v těle pravidla vyskytl (lze vyčíst z mapy `facts`). Pokud tedy máme například predikátový symbol `xml` a v těle pravidla se vyskytuje již podruhé, bude alias jména tabulky `xml2`.

Poté projdeme všechny argumenty predikátu a určíme atributy tabulky, které budou odpovídat jednotlivým argumentům. V tomto kroku se odlišuje magické zpracování pravidel od klasického zpracování. Nejdříve popíšeme klasické zpracování. Jména atributů tabulky se skládají z písmene 'a' a z indexu, který určuje pořadí argumentu. Vezměme například predikát `self_fb(Line2, Element2)`. Argumentu `Line2` odpovídá atribut `a1` a argumentu `Element2` odpovídá atribut `a2`. Pokud se predikát `self_fb` objevuje v těle pravidla poprvé, alias tabulky je `self_fb1`. Do mapy alias přidáme dva řetězce: do seznamu odpovídajícímu klíči `Line2` přidáme řetězec `self_fb1.a1` a do seznamu odpovídajícímu klíči `Element2` přidáme řetězec `self_fb1.a2`.

Nyní se zaměříme na magické zpracování pravidel. To se bude lišit pouze v názvech atributů pro jednotlivé argumenty, přidávání do mapy `alias` bude probíhat stejně jako při klasickém zpracování. Při magickém zpracování pravidel využijeme již vytvořené pole `magicArg`. Například pro predikát `m_intersection_bbfb(Line1, Line2, Level2)` bude `magicArr` vypadat takto: `[1, 2, 4]`. Atributy tedy budou mít názvy `a1`, `a2` a `a4`, které v tomto pořadí odpovídají argumentům predikátu.

Aritmetický výraz

Zpracování se liší pro aritmetický výraz s přiřazovacím operátorem (`is`) a pro aritmetické výrazy s ostatními operátory. Výrazy s přiřazovacím operátorem jsou zpracovány ihned, ostatní termy se pouze přidají do seznamu `operatorsList` a budou zpracovány později. Nyní přejdeme ke zpracování výrazu s přiřazovacím operátorem. Nejprve se budeme zabývat pravou stranou aritmetického výrazu, tedy částí za klíčovým slovem `is`. V této části zaměníme všechny proměnné za konkrétní atribut určité tabulky. Vezměme si za příklad aritmetický výraz:

```
Level2 is Level1 + 1
```

Proměnnou `Level1` musíme zaměnit za určitý atribut tabulky. Podíváme se do mapy alias, klíčem bude `Level1` a ze získaného seznamu vybereme libovolný řetězec. Ten může vypadat třeba takto: `m_level_inc_fb1.a2`. Upravená pravá strana výrazu je tedy `m_level_inc_fb1.a2 + 1`. Tento řetězec uložíme do mapy alias, jako klíč k vybrání správného seznamu použijeme proměnnou v levé části výrazu. V našem případě tedy bude klíč Level2.

Tímto postupem jsme naplnili datové struktury, které budou nezbytné pro generovaní uložených funkcí – mapy `alias` a `facts` a seznamy `operatorsList` a `negationsList` .

6.4.6 Vytvoření writeru

Nejprve si vytvoříme writer, který bude ukládat výsledky transformace. V balíku `cz.zcu.dip.io` se nachází abstraktní třída `Writer`. Má dvě abstraktní metody `write()` a `close()`. Metoda `write()` má jeden parametr typu `String` a zajišťuje zapsání předané uložené funkce. Metoda `close()` zajišťuje korektní ukončení výstupu. Od třídy `Writer` jsou zděděné třídy `FileWriter`, `JdbcWriter` a `JdbcAndFileWriter`.

Třída `FileWriter` zajišťuje zápis vygenerovaných uložených funkcí do textového souboru. V konstruktoru požaduje jako parametr jméno souboru, pro který si vytvoří výstupní proud. V metodě `write()` jednoduše zapíše předaný text do výstupního proudu. Metoda `close()` výstupní proud uzavře.

Třída `JdbcWriter` slouží pro ukládání PL/pgSQL funkcí přímo do databáze. Konstruktoru je nutné předat tři parametry: url databáze, jméno uživatele a heslo. Konstruktor vytvoří spojení s databází. Metoda `write()` spustí předaný dotaz v databázi. V metodě `close()` se ukončí spojení s databází.

Třída `JdbcAndFileWriter` se používá, pokud si uživatel přeje vygenerované funkce uložit do souboru a zároveň je rovnou předat databázi. Třída pouze využívá dvou výše popsaných tříd. V konstruktoru třídě předáme čtyři parametry: jméno výstupního souboru, url databáze, jméno uživatele a heslo. V konstruktoru se vytvoří objekt třídy `FileWriter` a objekt třídy `JdbcWriter`. V metodě `write()` a `close()` se poté pouze zavolají tyto funkce nad oběma objekty.

Podle toho, jaké byly metodě `runTransformation()` zadány parametry, vytvoříme příslušný typ writeru. Pokud je zadáno jméno výstupního souboru, vytvoříme objekt třídy `FileWriter`. Pokud byly předány údaje potřebné k databázovému spojení, vytvoříme objekt třídy `JdbcWriter`. Pokud byly předány všechny tyto údaje, vytvoříme třídu `JdbcAndFileWriter`. Díky dědičnosti poté bude možné ke všem typům writerů přistupovat jednotně, jako ke třídě `Writer`.

6.4.7 Generování databázových záznamů

V seznamu `dataList` máme uložené objekty třídy `Data`. V cyklu všechny objekty projdeme a zavoláme nad nimi metodu `getInsert()`. Každý objekt v sobě má uložený název predikátu `predicatName` a seznam argumentů predikátu `valuesList`. V metodě `getInsert()` se vytvoří řetězec ve tvaru:

`INSERT INTO predicatName (a1,a2, ... an) VALUES (val1,val2, ... valn).`

`PredicateName` označuje název tabulky, `a1` až `an` jsou názvy atributů tabulky. Jejich počet odpovídá počtu hodnot v seznamu `valuesList`. Pro magické predikáty jsou pravidla pro označování sloupců tabulky odlišná, implementace generování správného označení je popsána v kapitole 6.4.5. Hodnoty $\texttt{val}_1$ až $\texttt{val}_n$ jsou hodnoty seznamu `valuesList`.

Řetězce získané zavoláním metody `getInsert()` pomocí writeru zapíšeme na výstup.

6.4.8 Generování uložených funkcí

Pro každý seznam logických pravidel se stejným predikátovým symbolem bude vygenerována jedna uložená funkce v PL/pgSQL. Seznamy pravidel se stejným identifikátorem jsou uloženy jako hodnoty v mapě `rulesMap`. V cyklu tedy projdeme všechny tyto objekty typu `RulesWithEqualId`, zavoláme nad nimi metodu `getFunctionStr()` a výsledný řetězec uložíme do výstupního souboru. Nyní podrobněji probereme, co se uvnitř této metody děje.

Tvorba uložené funkce je vcelku přímočará a je popsána v kapitole 5.1.1. Napíšeme hlavičku, jméno funkce je shodné s predikátovým symbolem pravidla. Pro každé pravidlo s tímto predikátovým symbolem vytvoříme jeden kurzor. SQL dotaz spojený s každým kurzorem získáme metodou `getSelect()`, kterou nad pravidlem zavoláme. Tvorba tohoto dotazu bude popsána dále. Poté v těle funkce pro každý kurzor vytvoříme rutinu, která zajistí uložení výsledků dotazu spjatým s kurzorem do tabulek. Napíšeme ukončení funkce a vrátíme vygenerovaný řetězec. Každý takto vygenerovaný řetězec se pomocí vytvořeného writeru zapíše na výstup.

Generování SQL dotazů pro jednotlivé kurzory

SQL dotazy pro jednotlivé kurzory se generují uvnitř metody `getSelect()` ve třídě Rule. Napíšeme klíčové slovo `SELECT`, dále projdeme prvních n klíčů mapy `alias`, kde n je počet argumentů pravidla (jedná se o `LinkedHashMap`, je tedy zaručeno stálé pořadí prvků). Každý klíč vypíšeme spolu s klíčovým slovem `AS` a libovolným řetězcem ze seznamu, který odpovídá danému klíči. Jednotlivé bloky oddělíme čárkou. Pokud je klíčem číslice, vypíšeme pouze tuto číslici.

Celý blok může vypadat například takto:

```
SELECT
    m_intersection_bbfb1.a1 AS line1,
    m_intersection_bbfb1.a2 AS line2,
    xml1.a2 AS element,
    1
```

Dále následuje klíčové slovo `FROM`. Poté projdeme mapu `facts` a vypíšeme vždy klíč (tj. predikátový symbol), klíčové slovo `AS` a predikátový symbol spolu s indexem. Indexy začínají jedničkou a zvětšují se o jedničku až do hodnoty uvedené v hodnotě mapy pro daný klíč. Jednotlivé záznamy od sebe oddělujeme čárkou.

Např. pro dvojici hodnot v mapě `[xml, 2]` vygenerujeme dva záznamy

```
xml AS xml1,
xml AS xml2
```

Poté napíšeme klíčové slovo `WHERE`. Projdeme celou mapu `alias`. Pokud je klíčem číslo, dáme do rovnosti vždy toto číslo s každým řetězcem v seznamu, který klíči odpovídá. Jednotlivé záznamy oddělujeme klíčovým slovem `AND`. Například pro dvojici hodnot v mapě [1, [m_intersection_bbfb1.a4, xml1.a4, xml2.a4]] vygenerujeme tři záznamy:

```
m_intersection_bbfb1.a4 = 1 AND
xml1.a4 = 1 AND
xml2.a4 = 1
```

Pokud je klíčem řetězec, dáme do rovnosti všechny řetězce v seznamu, který klíči odpovídá. Není nutné dávat do rovnosti každé dvě dvojice, stačí položit do rovnosti první řetězec se všemi ostatními. Například pro záznam v mapě [line2, [self_fb1.a1, m_intersection_fbff1.a2, intersection_fbff1.a2, xml1.a1]] vytvoříme tři záznamy:

```
self_fb1.a1 = m_intersection_fbff1.a2 AND
self_fb1.a1 = intersection_fbff1.a2 AND
self_fb1.a1 = xml1.a1
```

Nyní přichází na řadu zpracování aritmetických výrazů a predikátů, které byly v negaci. Začneme se zpracováním seznamu `operatorsList` , tedy zpracováním aritmetických výrazů. Všechny proměnné ve výrazech musíme zaměnit za určitý atribut tabulky. V mapě `alias` jako klíč zadáme název dané proměnné a z odpovídajícího seznamu vybereme libovolný řetězec. Takto upravený aritmetický výraz přidáme do klauzule `WHERE`.

Například mějme aritmetický výraz `Line1 < Line2` a v mapě alias mějme záznamy:

```
[line1, [m_intersection_bbfb1.a1, xml1.a1]]
[line2, [m_intersection_bbfb1.a2, xml2.a1]]
```

Upravený aritmetický výraz bude vypadat takto:

```
m_intersection_bbfb1.a1 < m_intersection_bbfb1.a2
```

Tento výraz přidáme do klauzule `WHERE`.

Jako poslední přichází na řadu zpracování seznamu `negationsList` , tedy predikátů, které byli v negaci. Negace se promítá do `WHERE` klauzule pomocí klíčových slov `NOT EXIST`, za kterým následuje výběr z tabulky. Vybírá se celý záznam z tabulky, jejíž jméno je shodné s predikátovým symbolem v negaci. Ve `WHERE` klauzuli vnořeného dotazu musíme dát do rovnosti atributy aktuální tabulky s atributy, které odpovídají stejné proměnné (mají stejný `alias`).

Například pro predikát v negaci `not (xml(Line, _, _, Level2))` argument `Line` odpovídá atributu `xml.a1` a argument `Level2` atributu `xml.a4`. Tyto atributy musíme dát do rovnosti s atributy, kterým tyto proměnné odpovídaly již dříve. Požadované informace opět vyčteme z mapy `alias`.

Mějme v mapě záznamy

```
[line, [m_self_bb1.a1, xml1.a1]]
[level2, [level_inc_fb1.a1]]
```

Potom dáme do rovnosti atribut `xml.a1` s atributem `m_self_bb1.a1` (odpovídají proměnné line) a dále atribut `xml.a4` s atributem `level_inc_fb1.a1` (odpovídají proměnné `level2`). Celý blok odpovídající predikátu v negaci `not(xml(Line, _, _, Level2))` bude vypadat takto:

```
NOT EXISTS (
     SELECT *
     FROM xml
     WHERE
          xml.a1 = level_inc_fb1.a1 AND
          xml.a4 = level_inc_fb1.a1)
```

Na konec SQL dotazu přidáme množinový operátor `EXCEPT`. Ten z výsledků předchozího dotazu odečte záznamy, které už se v tabulce nacházejí:

```
EXCEPT
     SELECT *
     FROM nazev_funkce;
```

6.4.9 Generování hlavních funkcí

Vytvoříme si generátory hlavních funkcí. Třídy, které se zabývají generováním hlavních funkcí se nachází v balíku `cz.zcu.dip.translator.mainFunc.` Abstraktní třída `MainFunctGenerator` obsahuje jednu abstraktní metodu `getMainString()`, která vrací kód hlavní funkce. Dále obsahuje několik metod, které vrací části kódu hlavní funkce. Od třídy `MainFunctGenerator` jsou zděděné dvě třídy `AlphabetMainFunctGen` a `CleverMainFunctGen`. Třída `AlphabetMainFunctGen` zajišťuje generování funkcí v abecedním pořadí, třída `CleverMainFunctGen` zohledňuje vztahy mezi funkcemi a snaží se je volat v co nejvýhodnějším pořadí.

Vytvoříme si seznam `mainGenList` datového typu `MainFunctGenerator`. Do seznamu vždy vložíme dvě instance třídy `AlphabetMainFunctGen`, jedné z nich nastavíme proměnnou `reverse` na `true`. Pokud je zapnuto generování chytré hlavní funkce, vytvoříme i instanci třídy `CleverMainFunctGen`.

Poté projdeme celý seznam generátorů hlavních funkcí `mainGenList`, nad každým z nich zavoláme metodu `getMainString()` a získaný řetězec pomocí writeru zapíšeme na výstup.

Generování abecedních hlavních funkcí

Třída `AlphabetMainFunctGen` zajišťuje generování funkcí v abecedním pořadí. V konstruktoru se ji předává mapa `rulesMap` a logická proměnná `reverse`. Pokud má proměnná `reverse` hodnotu `false`, jméno funkce se nastaví na hodnotu `main_abc` a pravidla se budou řadit podle abecedy. Pokud má proměnná `reverse` hodnotu `true`, jméno funkce je `main_zyx` a pravidla se budou řadit podle abecedy od konce (reverzní abecední pořadí).

Postup generování hlavní funkce je velmi přímočarý, text začátku i konce funkce je jasně daný. Zbývá pouze vygenerovat řádky uvnitř smyčky. Seznam všech vygenerovaných funkcí je v mapě `rulesMap`, ze které stačí vybrat všechny klíče. Tyto klíče

seřadíme podle abecedy (v případě funkce `main_zyx()` v reverzním pořadí) a pro každý název funkce vygenerujeme jednu řádku ve tvaru:

```
loop_rows := loop_rows + nazev_funkce();
```

Generování chytré hlavní funkce

Generování chytré hlavní funkce zohledňuje vztahy mezi funkcemi a je algoritmicky náročné. Zajišťuje ho třída `CleverMainGenerator` , které se v konstruktoru předá mapa `rulesMap`. Text chytré hlavní funkce vrací metoda `getCleverMainString()`. Nejprve si připravíme matici sousednosti `matrix`, která bude reprezentovat vztahy mezi jednotlivými funkcemi. Matice bude binární čtvercová a bude mít velikost odpovídající počtu klíčů v mapě `rulesMap`. Řádky i sloupce odpovídají vždy jedné funkci v pořadí, v jakém jsou uložené klíče v mapě `rulesMap`. `RulesMap` je typu `LinkedHashMap`, takže je zaručeno stálé pořadí prvků. Logická hodnota `true` na pozici [i, j] v matici znamená, že uvnitř funkce na pozici i se pracuje s výsledky funkce na pozici j, tedy funkce i je závislá na funkci j.

Nyní vyplníme jednotlivé řádky matice pomocí metody `getUsageArr()` ze třídy `RulesWithEqualId`. V této metodě se vytvoří binární pole prvků o velikosti odpovídající počtu klíčů v mapě `rulesMap` a předvyplní se logickou hodnotou `false`. Poté se nad všemi pravidly ze seznamu `rules` zavolá metoda `setUsageArr()`, které se předá vytvořené pole a seřazený seznam funkcí. Tato metoda vyhledá v pravidle všechny použité tabulky a postará se o zapsání logické hodnoty `true` na příslušné pozice. Takto vyplněné pole se poté použije jako řádka matice `matrix`.

Když je vyplněna matice sousednosti, můžeme provést Tarjanův algoritmus pro nalezení kvazikomponent v grafu. Tento algoritmus je implementován ve třídě `TarjanStrongConnectedRecognizer` a je podrobně popsán v kapitole 4.3.3. Metoda `getComponentsList()` vrátí seznam kvazikomponent `components`, který obsahuje seznamy indexů náležících do jedné kvazikomponenty. Seznam `components` je typu `List<List<Integer>>` .

Kvazikomponenty musíme seřadit podle acyklického číslování. K tomu slouží metoda `acyclicSortComponents()` třídy `CleverMainGenerator` . Nejprve si vytvoříme pole integerů `memberOfComp` . Jeho velikost bude odpovídat počtu řádek matice sousednosti a bude určovat, která funkce patří do které kvazikomponenty. Tedy hodnota h na pozici p bude znamenat, že funkce p patří do komponenty h. Projdeme seznam komponent a pole vyplníme. Poté vytvoříme matici sousednosti `compMatrix` pro jednotlivé kvazikomponenty. Bude se jednat o čtvercovou binární matici o velikosti odpovídající počtu kvazikomponent. Nejprve inicializujeme všechny hodnoty

matice na logickou hodnotu `false`. Poté projdeme celou původní matici sousednosti `matrix`, u všech logických hodnot `true` zjistíme (z pole `memberOfComp`), jakým komponentám indexy náleží a zapíšeme `true` na příslušnou pozici matice `compMatrix`.

Po vyplnění celé matice `compMatrix` může začít algoritmus acyklického číslování. Algoritmus je založen na hledání prázdných řádek v matici sousednosti. Pokud jsou všechny hodnoty v řádce v matici sousednosti `false`, znamená to, že se jedná o vstupní vrchol grafu. Po nalezení vstupního vrcholu tento vrchol očíslujeme a dále ho ignorujeme. Indexy, které se týkají již očíslovaných vrcholů přeskakujeme a hledáme další prázdný řádek v matici. Takto postupujeme dokud neočíslujeme všechny vrcholy. Kvazikomponenty seřadíme dle acyklického pořadí.

V této chvíli zbývá ještě seřadit vrcholy uvnitř jednotlivých kvazikomponent. K tomu slouží metoda `sortNodesInComponents()` třídy `CleverMainGenerator`. V každé kvazikomponentě se na začátek dají vrcholy s přímou rekurzí (u vrcholu je smyčka) a zbytek kvazikomponenty se prohledá do hloubky a vrcholy se seřadí podle pořadí při prohledávání.

Nyní jsou v seznamu `components` acyklicky seřazeny jednotlivé kvazikomponenty a vrcholy uvnitř kvazikomponent jsou seřazeny podle prohledávání do hloubky. Můžeme tedy začít generovat text chytré hlavní funkce. Hlavička a deklarační část funkce jsou pevně dané. Uvnitř těla funkce budeme postupně volat všechny funkce z jednotlivých kvazikomponent. Pokud kvazikomponenta obsahuje pouze jeden vrchol, jednoduše ji zavoláme a započítáme počet vygenerovaných řádek:

```
rows := rows + nazev_funkce();
```

Pokud kvazikomponenta obsahuje více vrcholů, vytvoříme pro ni smyčku a jednotlivé funkce budeme volat uvnitř této smyčky:

```
LOOP
     loop_rows = 0;

     loop_rows := loop_rows + nazev_funkce_1();
     loop_rows := loop_rows + nazev_funkce_2();
     loop_rows := loop_rows + nazev_funkce_3();

     EXIT WHEN loop_rows = 0;
     rows := rows + loop_rows;
END LOOP;
```

Takto vygenerujeme bloky kódu pro všechny kvazikomponenty a funkci ukončíme.

6.4.10 Závěrečná rutina

Vraťme se zpět do metody `runTransformation()` ze třídy `PrologToPostgreSql`. Kód všech vygenerovaných funkcí získáme jako návratovou hodnotu volaných metod. Tento String zapíšeme do výstupního souboru. Poté ukončíme všechny vstupní i výstupní proudy. Dále smažeme dočasný soubor, který obsahoval kopii zdrojového souboru bez komentářů. Tím program skončí.

7 Testování aplikace

V této kapitole se zaměříme na testování správnosti vygenerovaných funkcí. Vyjasníme si vztah mezi programem v logickém a databázovém prostředí a dále porovnáme výkonnostní aspekty hlavních funkcí.

7.1 Program potomek

V této kapitole ukážeme jednoduchý příklad programu v Prologu včetně několika dotazů. Poté si stejný příklad převedeme do databázového prostředí a ukážeme si, jak používat vygenerované funkce a jak se dotazovat. Výsledky dotazů v databázovém prostředí porovnáme s výsledky obdobných dotazů v logickém prostředí.

Vezměme si jednoduchý program v Prologu[1]:

```
rodic(karel, jana).
rodic(jana, laura).
potomek(X,Y) :- rodic(Y,X).
potomek(X,Y) :- rodic(Y,Z), potomek(X,Z).
```

Fakta definují dvě instance relace rodič, pravidla rekurzivně definují relaci potomek. Pokud bychom v Prologu chtěli vypsat všechny dvojice, pro které tato relace platí, položili bychom dotaz:

```
?- potomek(X,Y).
```

Zde je odpověď Prologu, tedy všechny možné výsledky:

```
X = jana, Y = karel ;
X = laura, Y = jana ;
X = laura, Y = karel ;
false.
```

Vznikly celkově tři instance relace potomek.

[1] Všechny soubory k tomuto příkladu jsou k dispozici v přiloženém DVD ve složce `tests\potomek`

Pokud bychom chtěli vědět, kdo je potomkem Jany, položili bychom dotaz:

```
?- potomek(X,jana).
```

Odpověď je:

```
X = laura ;
false.
```

Jediným potomkem Jany je Laura.

Nyní si ukážeme, jak bude vypadat tento program převedený do databázového prostředí a jakým způsobem budeme zjišťovat odpovědi na dotazy.

Předpokladem je mít vytvořené tabulky, které svým názvem odpovídají predikátům a počet atributů tabulky odpovídá počtu argumentů predikátu. Datový typ atributů tabulky odpovídá datovému typu argumentů predikátu. O konvenci pojmenovávání atributů tabulky již byla řeč v kapitole 5. V našem případě musíme mít vytvořenou tabulku `rodic`, která bude obsahovat dva sloupce řetězcového typu s názvy `a1` a `a2`. Dále musíme mít vytvořenou tabulku `potomek`, která také bude obsahovat dva sloupce `a1` a `a2` řetězcového typu. Tabulky vytvoříme tímto SQL skriptem:

```
CREATE TABLE rodic(
        a1 character varying,
        a2 character varying);
CREATE TABLE potomek(
        a1 character varying,
        a2 character varying);
```

Když máme tento předpoklad splněný, můžeme pomocí vytvořeného programu PrologToPostgreSQL vygenerovat data a funkce. Překlad programu vypadá takto:

```
INSERT INTO rodic (a1, a2) VALUES ('karel', 'jana');
INSERT INTO rodic (a1, a2) VALUES ('jana', 'laura');
CREATE OR REPLACE FUNCTION potomek()
        RETURNS INTEGER AS $body$
DECLARE
        rows INTEGER;
cur1 CURSOR FOR
        SELECT
                rodic1.a2 AS x,
                rodic1.a1 AS y
```

```
        FROM
                rodic AS rodic1
        EXCEPT
                SELECT *
                FROM potomek;

cur2 CURSOR FOR
        SELECT
                potomek1.a1 AS x,
                rodic1.a1 AS y
        FROM
                rodic AS rodic1,
                potomek AS potomek1
        WHERE
                rodic1.a2 = potomek1.a2
        EXCEPT
                SELECT *
                FROM potomek;

BEGIN
        rows := 0;
FOR data IN cur1
        LOOP
                INSERT INTO potomek(a1, a2)
                        VALUES (data.x, data.y);
                rows := rows + 1;
        END LOOP;
FOR data IN cur2
        LOOP
                INSERT INTO potomek(a1, a2)
                        VALUES (data.x, data.y);
                rows := rows + 1;
        END LOOP;
RETURN rows;
END; $body$
LANGUAGE plpgsql;
```

Spuštěním tohoto kódu se nám do tabulky rodic uloží dva záznamy:

rodic	
a1	a2
karel	jana
jana	laura

A vytvoří se funkce potomek().

Dále program vygeneruje hlavní funkce (zde je uvedena pouze jedna z nich):

```
CREATE OR REPLACE FUNCTION main_abc()
        RETURNS INTEGER AS $body$
DECLARE
        rows INTEGER;
        loop_rows INTEGER;
BEGIN
        rows := 0;
        LOOP
                loop_rows = 0;
                loop_rows := loop_rows + potomek();
                EXIT WHEN loop_rows = 0;
                rows := rows + loop_rows;
        END LOOP;
        RETURN rows;
END; $body$
LANGUAGE plpgsql;
```

Po spuštění tohoto kódu se v databázi vytvoří funkce main_abc();

Nyní již můžeme spustit kód hlavní funkce. Toho docílíme dotazem:

```
select main_abc();
```

Funkce vrátí hodnotu 3, to znamená, že byly vygenerovány celkem 3 záznamy v databázi.

Nyní se podívejme do tabulky potomek:

potomek	
a1	a2
jana	karel
laura	jana
laura	karel

V tabulce vidíme všechny dvojice, které odpovídají relaci potomek. Pouhý pohled do tabulky nám dá odpověď, na kterou jsme se v Prologu ptali dotazem:

```
?- potomek(X,Y).
```

Nyní si ukážeme, jak v databázovém prostředí můžeme zjistit, kdo je potomkem Jany. V Prologu jsme pro tento účel zadali dotaz:

```
?- potomek(X,jana).
```

V databázi vykonáme SQL dotaz nad tabulkou potomek:

```
SELECT a1
FROM potomek
WHERE
        a2 = 'jana';
```

Dostaneme odpověď: laura.

Na tomto jednoduchém příkladu jsme si ukázali, jakým způsobem se používají funkce vygenerované programem PrologToPostgreSQL a jak dostaneme v databázovém prostředí odpovědi na dotazy. Zároveň jsme si ukázali, že v tomto příkladu se odpovědi v logickém i databázovém prostředí shodují.

7.1.1 Rekurze

V kapitole 2.2 jsme si řekli, že pokud konstruujeme rekurzivní pravidla, musíme zajistit, aby se výpočet nezacyklil. Toho se docílí definicí ukončovací podmínky. Tato podmínka musí být definována dříve než rekurzivní pravidlo a také je nutné, aby

uvnitř tohoto pravidla byl rekurzivní predikát co nejvíce vpravo. V případě jednoduchých pravidel je relativně snadné to zajistit. Pokud máme ale velké množství pravidel, které jsou na sobě závislé a graf závislosti tvoří cyklus, může už být velmi obtížné určit takové pořadí pravidel, aby výpočet proběhl korektně.

Po převodu logického programu do databázového prostředí už toto nehrozí. Vezměme si například tento program:

```
rodic(karel, jana).
rodic(jana, laura).
potomek(X,Y) :- potomek(X,Z), rodic(Y,Z).
potomek(X,Y) :- rodic(Y,X).
```

Jedná se o podobný program, se kterým jsme již pracovali v kapitole 7.1 . Rozdíl je v tom, že jsou pravidla prohozená a v těle prvního pravidla jsou prohozené predikáty. První pravidlo tedy obsahuje levou rekurzi a navíc je toto rekurzivní pravidlo v programu dříve, než ukončovací podmínka. Zkusíme interpretu Prologu předložit tento program a zadat dotaz:

```
?- potomek(X,jana).
```

Tento dotaz není vyhodnocen, interpret vypíše chybovou hlášku „ERROR: Out of local stack“, která oznamuje přetečení zásobníku.

Pokud tento program převedeme do databázového prostředí a položíme obdobný dotaz, dostaneme očekávanou odpověď: laura.

V tomto jednoduchém příkladu by bylo snadné dodržet pravidla pro psaní rekurzivních pravidel a předejít tak zacyklení interpretu Prologu. Ale u složitých vztahů, zvláště pokud obsahují nepřímou rekurzi, je velmi těžké zajistit korektní fungování programu.

7.2 Program child

V této kapitole si rozebereme rozsáhlejší příklad, ve kterém si ukážeme vztah mezi originálním a magickým programem.

Vstupní program v Prologu (v originální i magické verzi) i odpovídající uložené funkce v PL/pgSQL byli dodány zadavatelem. Uložené funkce byly psány ručně a sloužili zároveň jako specifikace, co aplikace musí umět. Po vytvoření aplikace PrologToPostgeSQL bylo možné zkontrolovat, jestli uložené funkce vygenerované

aplikací odpovídají funkcím vytvořeným ručně. Pro porovnání byl použit program KDiff3[2]. Funkce si odpovídají, liší se pouze v několika drobnostech, například v klauzuli WHERE jsou uvedené jiné proměnné, které ale odpovídají stejnému aliasu, takže význam zůstává stejný. Liší se také pořadí tabulek v klauzuli FROM a pro větší přehlednost je ve vygenerovaných funkcích mezi názvem proměnné a jejím aliasem klíčové slovo AS.

Jak bylo řečeno v úvodu, aplikace PrologToPostrgeSQL je vytvářena jako součást deduktivního databázového systému. Navazuje na aplikaci, která transformuje data ve formátu xml do Prologu[3]. Vstupní data tohoto programu jsou fakta Prologu získaná právě transformací xml souboru do Prologu. Prologovská pravidla definují vztahy mezi jednotlivými predikáty a umožňují vyhledávat vnořené elementy v původním xml souboru. Zde je původní xml soubor[4]:

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <books>
3       <book year="2003">
4            <author>Author1</author>
5            <author>Author2</author>
6            <author>Author3</author>
7            <title>Title1</title>
8            <review>Review1</review>
9       </book>
10      <book year="2002">
11           <author>Author2</author>
12           <author>Author3</author>
13           <author>Author4</author>
14           <title>Title2</title>
15           <review>Review2</review>
16      </book>
17      <book year="2002">
18           <author>Author4</author>
19           <author>Author5</author>
20           <author>Author1</author>
21           <title>Title3</title>
22           <review>Review3</review>
23      </book>
24 </books>
```

[2]Domovské stránky programu KDiff3: `http://kdiff3.sourceforge.net/`

[3]Více o aplikaci a o algoritmu převodu xml do Prologu v bakalářské práci [22].

[4]Soubor books.xml je k dispozici také na přiloženém DVD ve složce `tests\child`

7.2.1 Originální program

Všechny soubory, které se vztahují k tomuto programu, jsou k dispozici na přiloženém DVD ve složce `tests\child\original`. Do interpretu Prologu načteme programy `data.pro` a `child.pro`. Zadáme dotaz:

```
?- child(A, B, C, 'book').
```

Hledáme tedy elementy s názvem `book`. Proměnná B určuje číslo řádku, na kterém element začíná. Proměnná C určuje název nadřazeného elementu a proměnná A říká, na kterém řádku nadřazený element začíná. Dostaneme odpověď:

```
A = 2, B = 3, C = books ;
A = 2, B = 10, C = books ;
A = 2, B = 17, C = books ;
false.
```

Tato odpověď nám říká, že existují tři elementy s názvem book. Všechny mají nadřazený element books, který začíná na řádce 2. Elementy začínají na řádkách 3, 10 a 17.

Nyní tento příklad pustíme v databázovém prostředí. Pomocí skriptu `tables.sql` si v databázi vytvoříme potřebné tabulky. Skript `data.sql` vloží do tabulek vstupní data a skript `child.sql` v databázi vytvoří potřebné funkce. Poté spustíme hlavní funkci `main_abc()` zadáním dotazu:

```
select main_abc();
```

Funkce vrátí hodnotu 258. To znamená, že v databázi bylo vytvořeno 258 záznamů. Pro zjištění, kde se v původním xml dokumentu nacházejí elementy s názvem book, zadáme dotaz:

```
SELECT a1, a2, a3
FROM child
WHERE a4 ='book';
```

Dostaneme odpověď:

child		
a1	a2	a3
2	10	books
2	17	books
2	3	books

Získali jsme stejné výsledky, jako v interpretu Prologu.

Vyprázdníme všechny tabulky, které odpovídají jménům funkcí a spustíme funkci `main_zyx()`. Počet vygenerovaných řádků i výsledky našeho dotazu jsou stejné jako u funkce `main_abc()`. To samé platí i pro funkci `main_clever()`. Počet vygenerovaných záznamů i výsledky dotazu zůstávají stejné.

Ještě je nutné zmínit, že originální program v databázovém prostředí po spuštění hlavní funkce dává odpovědi na všechny dotazy, které by bylo možné položit i v interpretu Prologu. Pokud chceme položit jakýkoliv jiný dotaz, nemusíme znovu generovat žádná data, pouze se dotážeme nad existujícími tabulkami. Pokud bychom se například rozhodli zjistit, jaké elementy jsou vnořené v elementu `book` definovaného na řádku 10, stačilo by zadat dotaz:

```
SELECT a2, a4
FROM child
WHERE
     a1 = 10   AND
     a3 = 'book';
```

Dostaneme odpověď:

child	
a2	a4
12	author
13	author
14	title
15	review
11	author

Element book, který začíná na řádce 10 má celkem pět vnořených elementů. Na řádkách 11, 12 a 13 se nacházejí elementy s názvem `author`, na řádce 14 element `title` a na řádce 15 element `review`.

7.2.2 Magický program

Magický program je optimalizace originálního programu pro jeden konkrétní dotaz. Tento program je optimalizován pro dotaz `?- child_fffb(A, B, C, 'konstanta')`. Všechny soubory, které se k tomuto programu vztahují, jsou k dispozici na přiloženém DVD ve složce `tests\child\magic`.

Do interpretu Prologu zavedeme programy `child_fffb.pro` a `data.pro`. Poté zadáme dotaz:

```
?- child_fffb(A, B, C, 'book').
```

Dotazujeme se na stejnou věc, jako u originálního programu. Zajímá nás, na jaké řádce se nacházejí elementy s názvem `book`, jak se jmenuje jejich nadřazený element a na jaké řádce se nachází. Získáme výsledky:

```
A = 2, B = 3, C = books ;
A = 2, B = 3, C = books ;
A = 2, B = 3, C = books ;
A = 2, B = 3, C = books ;
A = 2, B = 3, C = books ;
A = 2, B = 3, C = books ;
```

Stejný výsledek vypíše interpret ještě několikrát, poté chvíli nereaguje a nakonec vypíše hlášku „Out of local stack“. Na obrázku 5.2 jsou zobrazeny závislosti predikátů tohoto programu. Graf obsahuje cyklus. Interpret Prologu vzhledem k nepřímé rekurzi není schopný dotaz správně vyhodnotit.

Zkusíme příklad převést do databázového prostředí. Pomocí skriptů `tables.sql` vytvoříme potřebné tabulky, skriptem `data.sql` je naplníme vstupními daty a spuštěním skriptu `child_fffb.sql` vytvoříme uložené funkce. Konstantu, kterou v dotazu chceme použít zadáme do tabulky `m_child_fffb` na pozici `a4`. V našem případě vložíme konstantu `book`. Poté spustíme výpočet zadáním dotazu:

```
SELECT main_abc();
```

Funkce vrátí číslo 70. V databázi bylo vytvořeno 70 záznamů. To je oproti originální verzi programu o poznání méně. Podíváme se do tabulky child_fffb:

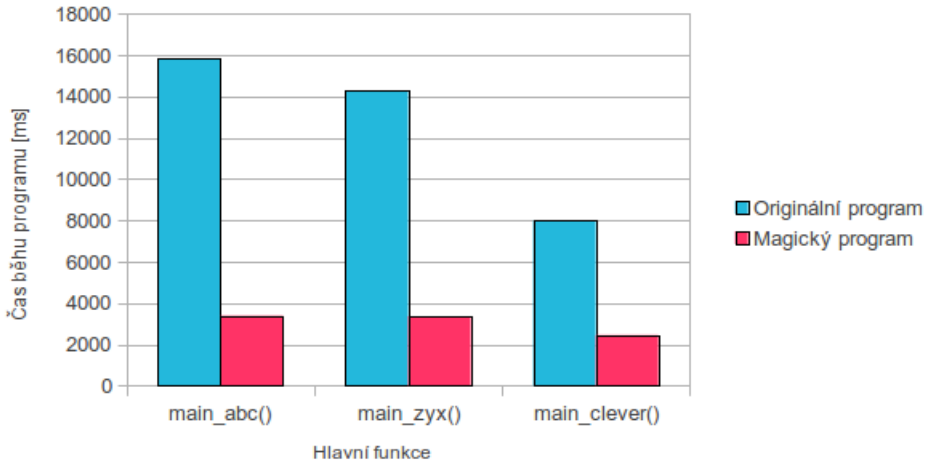
child_fffb		
a1	a2	a3
2	17	books
2	10	books
2	3	books

Tabulka obsahuje pouze data, která jsou odpovědí na dotaz, pro který byl tento magický program vytvořen. Odpověď na dotaz se shoduje s odpovědí originálního programu. Nepřímá rekurze v závislostech mezi funkcemi nebyla překážkou pro správné vyhodnocení dotazu.

Pokud místo hlavní funkce `main_abc()` spustíme hlavní funkci `main_zyx()` nebo `main_clever()`, dostaneme stejný počet vygenerovaných záznamů a vygenerovaná data se budou shodovat.

7.3 Porovnání hlavních funkcí

Ke každému programu se vygenerují tři hlavní funkce: `main_abc()`, `main_zyx()` a `main_clever()`. První dvě hlavní funkce ve svém těle volají ostatní funkce v abecedním pořadí, chytrá hlavní funkce zohledňuje jejich vzájemné vztahy. Při testování



Obrázek 7.1: Graf závislosti času výpočtu programu na zvolené hlavní funkci.

se potvrdila domněnka, že na pořadí volání funkcí nezáleží a že tedy všechny hlavní funkce generují stejná data.

V této kapitole zkusíme zjistit, zda chytrá hlavní funkce pozitivně ovlivní dobu výpočtu. V předchozích příkladech jsme pracovali s tak malými daty, že doba výpočtu byla zanedbatelná a tudíž neporovnatelná.

Vezměme si opět program `child`. Tentokrát však bude zpracovávat větší vstupní soubor[5], aby časy výpočtu byly větší.

Na obrázku 7.1 vidíme graf závislosti času výpočtu programu na zvolené hlavní funkci. Na ose x jsou tři varianty hlavní funkce, na ose y je čas výpočtu programu v milisekundách. Modrou barvou jsou znázorněny časy výpočtu originálního programu, růžovou barvou časy výpočtu magického programu.

Z grafu vidíme, že optimalizace v podobě magického programu je opravdu účinná. Čas výpočtu odpovědi na stejný dotaz se oproti originálnímu programu zkrátil přibližně na čtvrtinu. Zároveň vidíme, že chytrá hlavní funkce v případě originálního programu zkrátila čas výpočtu přibližně o polovinu. V případě magického programu není urychlení tak zřetelné, doba výpočtu se zkrátila přibližně o 30 procent.

Výpočet programu byl spouštěn na studentském serveru `students.kiv.zcu.cz`. Časy výpočtu jsou průměrem z deseti měření. Výsledky jsou relevantní pro konkrétní

[5]Vstupní xml soubor i vygenerovaná data v Prologu a SQL jsou k dispozici v přiloženém DVD ve složce `tests\child\main_function`

program a vstupní soubor. Pro jiné programy, které mají jiné vztahy mezi funkcemi, se výsledky mohou lišit.

8 Závěr

Hlavním tématem této diplomové práce je popis a realizace algoritmu, který převádí logický program do uložených objektů SŘBD PostgreSQL. Tomu předchází teoretická část, ve které jsou objasněny základy logického programování, práce s procedurálním jazykem PL/pgSQL a úvod do teorie grafů.

Vytvořená aplikace PrologToPostgreSQL umožňuje převod logických faktů do záznamů databázových tabulek. Logická pravidla jsou transformována do uložených funkcí v jazyce PL/pgSQL. Při každé transformaci je vytvořena hlavní funkce, která postupně spouští vytvořené funkce a umožňuje tak výpočet odpovědí na dotazy. Aplikace bude součástí rozsáhlého systému tvořícího experimentální deduktivní databázový systém. Z toho důvodu se jedná pouze o konzolovou aplikaci, předpokládá se, že se bude používat spíše jako knihovna a komunikace bude probíhat přes API.

Při vývoji aplikace byla zvláštní pozornost věnována vytvoření chytré hlavní funkce, která respektuje vztahy mezi jednotlivými funkcemi a snaží se je volat v co nevýhodnějším pořadí. Při testování se ukázalo, že použití této hlavní funkce výrazně zrychluje výpočet programu.

Literatura

[1] BIELIKOVÁ, Mária a Pavol NAVRÁT. *Funkcionálne a logické programovanie*. 1. vyd. Bratislava: Vydavateľstvo STU, 2000, 281 s. ISBN 80-227-1459-3.

[2] JIRKU, Petr. *Programování v jazyku PROLOG*. 1. vyd. Praha: SNTL, 1991, 251 s. ISBN 80-030-0609-0.

[3] BABKA, Otakar, Jaroslav JANČÍK a Zbyněk ZAHRADNÍK. *Programovací jazyk Prolog: Učební text kurzu*. Plzeň: Dům techniky ČSVTS Plzeň, 1988, 209 s.

[4] RUBÁČEK, Filip. Prolog - *Programování v Prologu* [online]. 1999 - 2005 [cit. 2015-02-05]. Dostupné z: http://iris.uhk.cz/logpro/index.html

[5] ZÍMA, Martin. *Experimentální deduktivní databázový systém s neurčitostí*. Plzeň, 2002. Disertační práce. Západočeská univerzita v Plzni. Vedoucí práce Karel Ježek.

[6] SEBESTA, Robert W. *Concepts of programming languages*. 7th ed. Boston: Pearson/Addison-Wesley, 2006, xvii, 724 s. ISBN 03-213-3025-0.

[7] PostgreSQL 9.3.5 Documentation. THE POSTGRESQL GLOBAL DEVELOPMENT GROUP. *PostgreSQL* [online]. © 1996-2014 [cit. 2014-12-01]. Dostupné z: `http://www.postgresql.org/docs/9.3/static/index.html`

[8] MOMJIAN, Bruce. *PostgreSQL: praktický průvodce*. 1. vyd. Brno: Computer Press, 2003, xiii, 402 s. ISBN 80-722-6954-2.

[9] PL/pgSQL. *PostgreSQL* [online]. 3. 8. 2013 [cit. 2014-12-02]. Dostupné z:`http://postgres.cz/wiki/PL/pgSQL`

[10] SEDLÁČEK, Jiří. *Úvod do teorie grafů*. 3. vyd. Praha: Academia, 1981, 272 s.

[11] DEMEL, Jiří. *Grafy a jejich aplikace*. Vyd. 1. Praha: Academia, 2002, 257 s. ISBN 80-200-0990-6.

[12] ŽAMBOCHOVÁ, Marta. *Teorie grafů v příkladech*. Vyd. 1. Ústí nad Labem: Univerzita Jana Evangelisty Purkyně, 2007, 97 s. ISBN 978-807-0449-622.

[13] BUCKLEY, Fred a Marty LEWINTER. *A friendly introduction to graph theory*. Upper Saddle River, New Jersey 07458: Prentice Hall, 2003, xiv, 365 p. ISBN 01-306-6949-0.

[14] GIBBONS, Alan. *Algorithmic graph theory*. Cambridge: Cambridge University Press, 1985, ix, 259 s. ISBN 05-212-8881-9.

[15] VEČERKA, Arnošt. KATEDRA INFORMATIKY, Přírodovědecká fakulta Univerzita Palackého. *d*. Olomouc, 2007, 106 s.

[16] MIČKA, Pavel. Tarjanův algoritmus. *Algoritmy.net* [online]. 2008 - 2014 [cit. 2014-11-13]. Dostupné z: `http://www.algoritmy.net/article/1515/Tarjanuv-algoritmus`

[17] MIČKA, Pavel. Prohledávání do hloubky. *Algoritmy.net* [online]. 2008 - 2014 [cit. 2014-11-18]. Dostupné z: `http://www.algoritmy.net/article/1378/Prohledavani-do-hloubky`

[18] MIČKA, Pavel. Prohledávání do šířky. *Algoritmy.net* [online]. 2008 - 2014 [cit. 2014-11-18]. Dostupné z: `http://www.algoritmy.net/article/1399/Prohledavani-do-sirky`

[19] RYJÁČEK, Zdeněk. ZÁPADOČESKÁ UNIVERZITA V PLZNI. *Teorie grafů a diskrétní optimalizace 1: Pracovní texty přednášek*. 2011, 73 s. Dostupné z: `http://www.cam.zcu.cz/~ryjacek/students/ps/TGD1.pdf`

[20] GNU Lesser General Public License. *GNU Operating System* [online]. © 2007 [cit. 2015-03-26]. Dostupné z: `https://www.gnu.org/licenses/lgpl.html`

[21] Apache License, Version 2.0. *The Apache Software Foundation* [online]. 2004 [cit. 2015-03-26]. Dostupné z: `https://www.apache.org/licenses/LICENSE-2.0.html`

[22] NOVÁ, Kateřina. *Nativní vícevláknová aplikace pro zpracování a transformaci XML dokumentů*. Plzeň, 2012. Dostupné z: `https://portal.zcu.cz/StagPortletsJSR168/KvalifPraceDownloadServlet?typ=1&adipidno=49926`. Bakalářská práce. Západočeská univerzita v Plzni.

[23] JEŽEK, Karel. Deduktivní databáze a jejich implementace v relačním prostředí. In: CHLAPEK, Dušan. *Datakon 2002: sborník databázové konference: Brno, 19.-22.10.2002.* 1. vyd. Brno: Masarykova univerzita, 2002, s. 16. ISBN 80-210-2958-7. Dostupné z: `http://www.kiv.zcu.cz/~jezek_ka/vyuka/DB2%202005/DEDUC/datakon02.pdf`

A Uživatelská příručka

Pro spuštění aplikace je nutné mít nainstalovanou Javu verze 1.6 nebo vyšší. Ve stejné složce jako spustitelný soubor `PrologToPostgeSQL.jar` je nutné mít složku s názvem `PrologToPostgreSQL_lib` a uvnitř této složky musí být uloženy dvě potřebné knihovny: `postgresql-9.3-1101.jdbc41.jar` a `prologparser-1.3.2.jar`.

Program `PrologToPostgeSQL` se spouští z příkazové řádky v následujícím tvaru:

```
java -jar PrologToPostgreSQL.jar inputFile
        [-out outputFile] [-db url user password]
        [-clever] [-magic] [-data]
```

První argument je povinný a určuje vstupní soubor. Je možné zadat absolutní nebo relativní cestu ke vstupnímu souboru. Přepínače `-out` a `-db` určují výstup programu. Je nutné definovat alespoň jeden typ výstupu. Za přepínačem `-out` následuje cesta k výstupnímu souboru (absolutní nebo relativní). Za přepínačem `-db` následují další tři parametry - url, user a password, které definují přístup do databáze. Přepínač `-clever` způsobí, že se kromě abecedních hlavních funkcí vygeneruje i chytrá hlavní funkce. Pokud bude zadán argument `-magic`, bude se vstupní soubor zpracovávat jako magický. Dojde tedy k malé úpravě algoritmu převodu logického programu do uložených funkcí Pl/pgSQL. Přepínač `-data` způsobí, že se ze vstupního souboru kromě logických pravidel zpracovávat i logická fakta. Pokud není zadán žádný argument, je vypsána nápověda.

Během zpracování je vypsáno jméno vytvořeného dočasného souboru, který obsahuje kopii vstupního souboru s výjimkou blokových komentářů. Pokud transformace proběhne v pořádku, je vypsána hláška „Finished.“ a je vypsáno jméno výstupního souboru nebo url výstupní databáze. Pokud se povede smazat dočasný soubor, vypíše se hláška „Temp file deleted.“. Pokud se během zpracování vyskytne nějaká chyba, je vypsána chybová hláška.

Použití PrologToPostgreSQL jako knihovny

Vytvoříme si Java projekt, připojíme soubor `PrologToPostgreSQL.jar` jako knihovnu. Pokud to SDK neudělá za nás, připojíme ještě knihovny `postgresql-9.3-1101.jdbc41.jar` a `prologparser-1.3.2.jar`. Poté vytvoříme objekt třídy `PrologToPostgreSql`, můžeme si ho přetypovat na rozhraní `PrologToPostgreSqlAPI`:

```
PrologToPostgreSqlAPI api = new PrologToPostgreSql();
```

Poté nad vytvořeným objektem zavoláme metodu `runTransformation()`. Metoda má osm parametrů:

String inputFilePath Absolutní nebo relativní cesta ke vstupnímu souboru.

String outputFilePath Cesta k výstupnímu souboru. Pokud má hodnotu `null`, nebude se výstupní soubor generovat.

boolean magicON Pokud má hodnotu `true`, bude se vstupní program zpracovávat jako magický. Při hodnotě `false` se bude zpracovávat originálním způsobem.

boolean cleverMainON Pokud má tato proměnná hodnotu `true`, bude kromě abecedních hlavních funkcí vygenerována i chytrá hlavní funkce.

boolean dataON Pokud má hodnotu `true`, budou se ze vstupního souboru kromě logických pravidel zpracovávat i logická fakta.

String url URL výstupní databáze. Pokud má hodnotu `null`, nebude se výstup do databáze zapisovat.

String user Uživatel databáze.

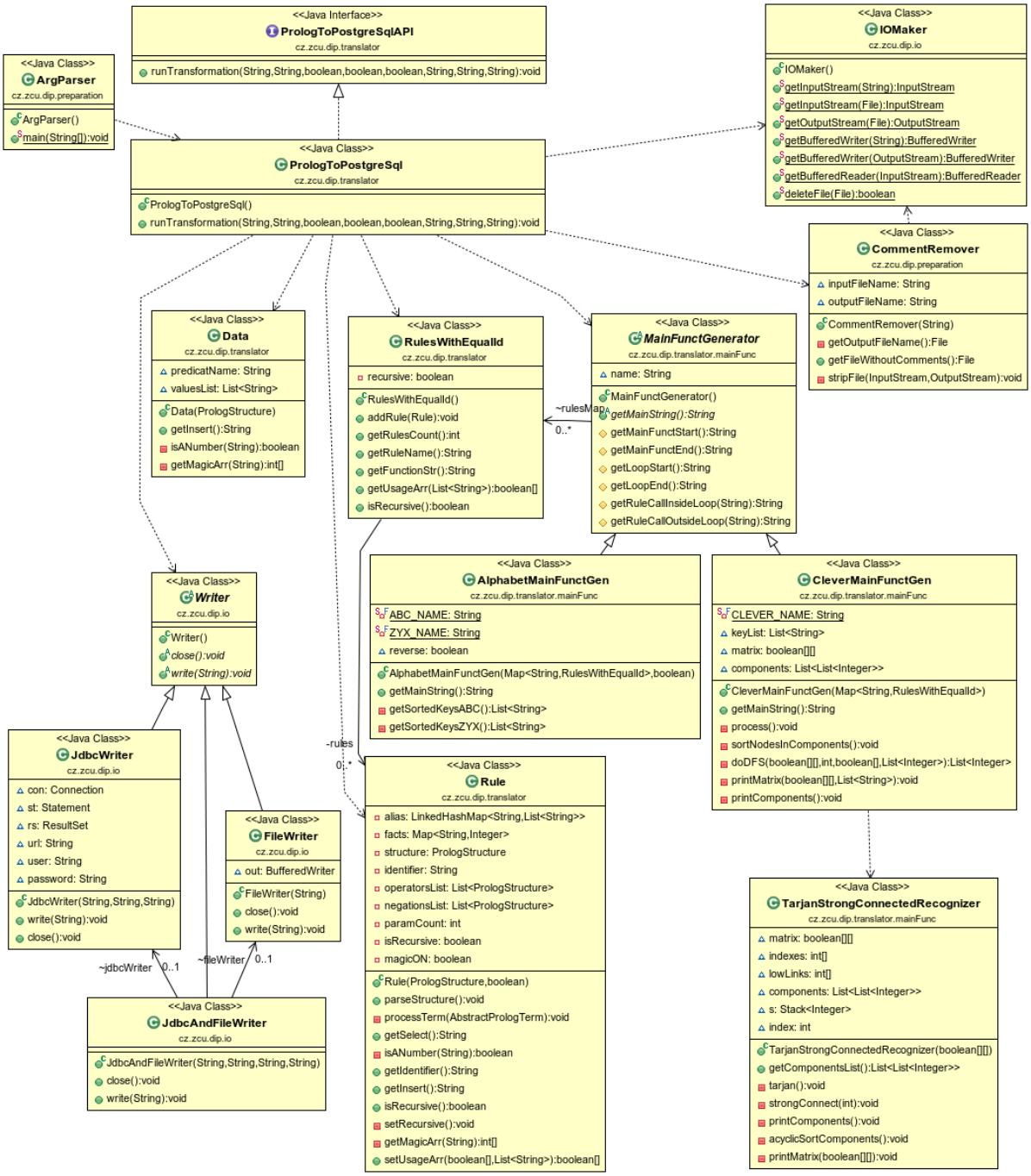
String password Heslo daného uživatele do databáze.

Pokud například chceme spustit transformaci pro vstupní soubor `in.pro`, výstupní soubor `out.sql` a chceme generovat chytrou hlavní funkci, zavoláme metodu `runTransformation()` s těmito parametry:

```
api.runTransformation("in.pro", "out.sql", false,
                true, false, null, null, null);
```

Metoda `runTransformation()` vrací hodnotu `true`, pokud transformace proběhla v pořádku. Pokud se vyskytne nějaká chyba a transformace neproběhne správně, vrací hodnotu `false`.

B UML diagram tříd



C Obsah DVD

Přílohou této diplomové práce je DVD, které má následující strukturu:

- **readme.txt** – Textový soubor, obsahující popis jednotlivých adresářů uložených na médiu.
- **doc** – Obsahuje elektronickou verzi této diplomové práce.
- **PrologToPostgreSQL** – Obsahuje všechny součásti programu, jako jsou spustitelný soubor a zdrojové kódy.
  - **bin** – Obsahuje spustitelnou verzi programu a potřebné knihovny.
  - **doc** – Vygenerovaná programátorská dokumentace.
  - **src** – Obsahuje zdrojové kódy programu a UML diagram tříd.
  - **lib** – Obsahuje knihovny, které program používá.
- **tests** – Obsahuje všechny logické programy i SQL skripty, které byly použity při testování
  - **potomek** – Obsahuje soubory vztahující se k programu potomek.
  - **child** – Obsahuje soubory vztahující se k programu child.
    - **books.xml** – Vstupní xml soubor programu.
    - **original** – Složka se soubory pro originální verzi programu.
    - **magic** – Složka se soubory pro magickou verzi programu
    - **main_functions** – Větší data pro porovnání výkonu hlavních funkcí.