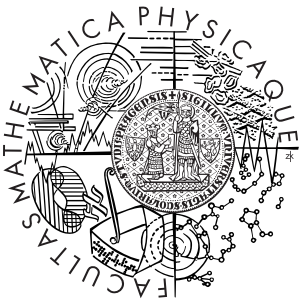
Univerzita Karlova v Praze
Matematicko-fyzikální fakulta

DIPLOMOVÁ PRÁCE



Bc. Miroslav Čermák

Překladač SPARQL pro Bobox

Katedra softwarového inženýrství

Vedoucí diplomové práce: RNDr. Jiří Dokulil
Studijní program: Informatika, softwarové systémy

2010

Na tomto mieste by som chcel poďakovať RNDr. Jiřímu Dokulilovi za vedenie diplomovej práce, za cenné rady a pripomienky hlavne pri návrhu aplikácie a písaní diplomovej práce.

Prohlašuji, že jsem svou diplomovou práci napsal samostatně a výhradně s použitím citovaných pramenů. Souhlasím se zapůjčováním práce a jejím zveřejňováním.

V Praze dne Bc. Miroslav Čermák

Obsah

Název práce: Prekladač SPARQL pro Bobox
Autor: Bc. Miroslav Čermák
Katedra (ústav): Katedra softwarového inženýrství
Vedoucí diplomové práce: RNDr. Jiří Dokulil
e-mail vedoucího: Jiri.Dokulil@mff.cuni.cz

Abstrakt: Cílem práce je navrhnout a implementovat překladač jazyka SPARQL pro systém Bobox. Kromě lexikální a syntaktické analýzy odpovídající W3C standardu jazyka SPARQL je součástí práce i sémantická analýza a optimalizace dotazů, která pro dotaz navrhne vhodný model výpočtu pro Bobox v závislosti na konkrétním fyzickém schématu databáze.
Klíčová slova: SPARQL, RDF, optimalizace dotazů, Bobox

Title: SPARQL compiler for Bobox
Author: Bc. Miroslav Čermák
Department: Department of Software Engineering
Supervisor: RNDr. Jiří Dokulil
Supervisor's e-mail address: Jiri.Dokulil@mff.cuni.cz

Abstract: The goal of the work is to design and implement a SPARQL compiler for the Bobox system. In addition to lexical and syntactic analysis corresponding to W3C standard for SPARQL language, it performs semantic analysis and optimization of queries. Compiler will constuct an appropriate model for execution in Bobox, that depends on the physical database schema.

Keywords: SPARQL, RDF, query optimalization, Bobox

Kapitola 1

Úvod

RDF je formát na reprezentáciu dát v Sémantickom Webe. V súčasnej dobe najpopulárnejší jazyk pre tvorbu požiadaviek na prácu s RDF dátovými kolekciami je SPARQL. Začiatkom roka 2008 dosiahol SPARQL status odporučenia W3C. To dodalo nový impulz aktuálnemu výskumu v oblasti reprezentácie RDF dát a ich následného spracovania pomocou jazyka SPARQL. Taktiež prebieha intenzívny výskum v oblasti paralelného a distribuovaného spracovania dát. Do tejto oblasti môžeme zaradiť aj systém Bobox, zameraný primárne na spracovanie požiadaviek v paralelnom prostredí. Táto práca ma za úlohu skĺbiť tieto dve technológie a vytvoriť prekladač jazyka SPARQL, generujúci plán spracovania pre systém Bobox.

Tento prekladač vykoná lexikálnu, syntaktickú analýzu odpovedajúcu W3C štandardu jazyka SPARQL. Vykoná taktiež sémantickú analýzu s ohľadom na dostupné typy dát v požiadavke a transformuje požiadavku do štruktúry vhodnej pre optimalizáciu, ktorá tvorí kľúčovú oblasť tejto diplomovej práce. Optimalizácia je kritickou časťou prekladu požiadavky v snahe dosiahnuť minimálny čas na jej odpoveď s ohľadom na snahu minimalizovania ceny samotného vykonávania. Ide o netriviálnu úlohu, vzhľadom na veľkosť množiny možných riešení, nutnosti definovať heuristík na jej redukciu a nepresnosti odhadu ceny jednotlivých riešení, ktorú ovplyvňuje množstvo faktorov. Taktiež neštandardná algebra jazyka SPARQL neumožňuje triviálne využiť postupy, ktoré sú navrhnuté pre jazyky požiadaviek založené na štandardnej relačnej algebre. Samotná optimalizácia je založená na využití dvoch prístupov:

- *Logická optimalizácia*, ktorá aplikovaním prepisovacích pravidiel zjednodušuje a zefektívňuje požiadavku.
- *Fyzická optimalizácia* je založená na vlastnostiach operácie *Join*: *komutativita* a *asociativita* . Za pomoci dynamického programovania sa

snaží skonštruovať strom operácií, usporiadaním operácií *Join* a voľbou vhodných fyzických operátorov reprezentujúcich jednotlivé operácie. Výsledný strom bude optimálny vzhľadom k použitým heuristikám a ohodnocovaciemu modelu.

Ako fyzickú schému sme zvolili schému vertikálnej tabuľky (tabuľky trojíc). Táto schéma je reprezentovaná jedinou tabuľkou o troch stĺpcoch, zodpovedajúcim trom častiam RDF záznamu. Schéma je zvolená aj napriek existencii rýchlejších, pretože na rozdiel od nich umožňuje bez problémov uložiť dáta ľubovoľného *RDF grafu* bez nutnosti jej zmien.

Nad fyzickou databázou budú vytvorené a udržiavané sumárne štatistiky. Tieto poskytujú podklady pre ohodnocovací modul a odhady veľkosti dát. Práca sa obmedzuje primárne na odhady operácií *Join* a *Scan*. Ostatné operácie sú ponechané na následné pokračovanie vývoja.

1.1 Štruktúra práce

Kapitola 1 obsahuje motiváciu a popis zadania diplomovej práce. Kapitola 2 nás oboznámi s technológiami RDF, SPARQL a systémom Bobox. Kapitola 3 je obecný úvod do problematiky optimalizácií, hlavne relačných jazykov na tvorbu požiadaviek. Zaoberá sa stručným prehľadom vývoja a možností optimalizácie všeobecne. Kapitola 4 popisuje základný popis návrhu nášho prekladača. Sú tu definované obmedzenia a požiadavky kladené na prekladač, oboznámenie sa s použitými dátovými štruktúrami a štruktúrou prekladača. Kapitola 5 je zameraná na použité metódy optimalizácie. Obsahuje popis statických optimalizácií a algoritmus dynamickej optimalizácie. Na konci popisuje sumárne štatistiky a metódy odhadu veľkosti dát, využívané pri dynamickej optimalizácii. Kapitola 6 popisuje testovaciu sadu dát a testy použité pri experimentovaní s výsledným prekladačom. Zaujímavejším prípadom sa tu venujeme podrobnejšie. Kapitola 7 zhrňuje výsledky práce a navrhuje možné smery ďalšieho pokračovania práce.

Kapitola 2

Úvod do problematiky

V tejto kapitole sú vysvetlené základné pojmy a technológie, ktoré stoja za ideou tejto práce. Predovšetkým sú to základné definície zo štandardov *RDF* a *SPARQL* a oboznámenie sa so systémom *Bobox*.

2.1 RDF

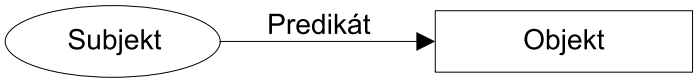
Resource Description Framework (RDF) je štandardom W3C [2], navrhnutým na reprezentáciu a spracovanie metadát. V súčasnosti sa RDF používa ako obecná metda k popisu, alebo modelovaniu informácií obsiahnutých v rôznych zdrojoch. Primárnym cieľom je umožniť dáta strojovo čítať a spracovávať a nie ich prezentovať užívateľovi. Za týmto účelom je prispôsobený aj ich formát, popsujúci informácie pomocou tvrdení. Tieto tvrdenia sa sú v *RDF* reprezentované pomocou *trojíc* (*subjekt predikát objekt*). Napríklad tvrdenie „Článok1 bol vydaný v roku 2010.“ popisuje *subjekt* „Článok1“. Konkrétne definuje jeho vlastnosť (*predikát*) „bol vydaný v roku“, priradením hodnoty (*objekt*) „2010“.

Uveďme si potrebné formálne definície a pojmy týkajúce sa RDF. Majme navzájom disjunktné nekonečné množiny U, B a L. Kde U je možina *URI* odkazov, B je množina *anonymných vrcholov* (*blank nodes*) a L je množina *literálov*. Trojica $(s, p, o) = (U \cup B) \times (U) \times (U \cup B \cup L)$ sa nazýva *RDF trojica*. V tejto trojici reprezentujú s, p, o postupne *subjekt*, *predikát* a *objekt*.

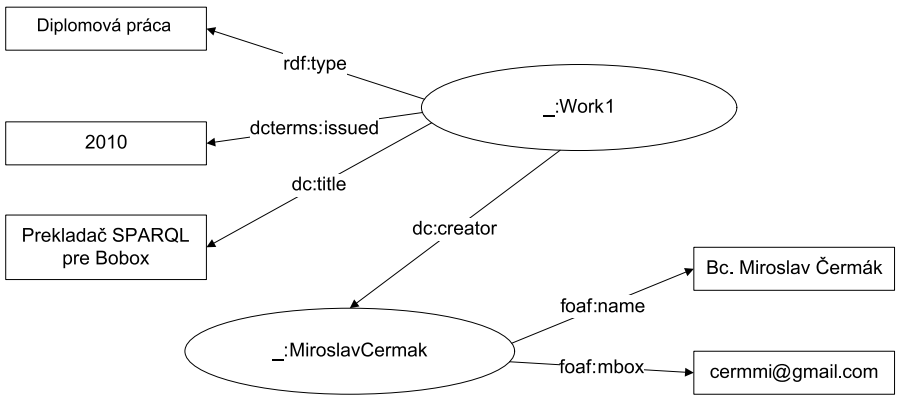
Definícia 1 *RDF graf [2] je množina RDF trojíc. V kontexte tejto práce sa odkazuje na RDF graf ako na sadu RDF dát, alebo sadu dát.*

Graficky môžeme tvrdenie reprezentovať ako dva vrcholy *subjekt* a *objekt*, spojené orientovanou hranou reprezentujúcou *predikát*, vyjadrujúcou vzťah medzi vrcholmi (obrázok 2.1). Možnosť použiť subjekt jednej *RDF trojice*

ako objekt inej, umožňuje vytvárať komplexnejšie vzťahy v rámci *RDF grafu*. Tento môžeme graficky reprezentovať ako orientovaný graf, viď príklad na obrázku 2.2.



Obr. 2.1: Grafická reprezentácia vzťahu subjekt-predikát-objekt.



Obr. 2.2: Príklad RDF grafu.

2.2 SPARQL

SPARQL [15] je jazyk na vytváranie požiadaviek nad RDF grafom. Bol štandardizovaný World Wide Web Konzorciom (W3C) a od 15. januára 2008 je oficiálnym W3C odporučením.

Jeho idea je založená na „matchovaní“ grafov. SPARQL požiadavka má formát $Q\colon H \leftarrow B$, kde B označuje *telo* požiadavky, ktoré je tvorené komplexným *RDF pattern expression* a H je hlavička, ktorá definuje ako sa má vyzerať odpoveď. *RDF trojice* v tele požiadavky môžu obsahovať premenné, nad ktorými môžu byť ďalej definované rôzne obmedzenia.

Spracovanie požiadavky prebieha v dvoch krokoch. V prvom kroku sa aplikuje telo požiadavky B na *RDF sadu dát* D, čím sa získa množina premenných naviazaných na hodnoty v grafe. V druhom kroku je táto množina pomocou hlavičky H transformovaná na požadovanú konečnú podobu odopvede požiadavky. Oproti iným jazykom, ako napríklad SQL, SPARQL neumožňuje vytvárať požiadavky na zmenu dát databáze, typu Insert, Update a Delete.

V súčasnosti sú podporované štyri druhy požiadaviek:

- **SELECT** vracia množinu n-tic premenných, ktoré vyhovujú šablóne požiadavky.
- **ASK** vracia hodnotu `yes` pokiaľ výsledok obsahuje aspoň jeden riadok, inak vracia hodnotu `no`.
- **DESCRIBE** vracia RDF graf popisujúci výsledné dáta.
- **CONSTRUCT** vracia RDF graf vytvorený nahradením premenných v zadanej šablone trojíc za výsledné hodnoty.

Následujúci príklad zobrazuje jednoduchú SPARQL požiadavku, ktorá vráti zoznam všetkých položiek a ich cien, ktoré majú definované vlastnosti *ns:price* a *rdf:type* a zároveň platí, že cena > 10 a typ položky je "A", alebo "B".

```
PREFIX ns: <http://example.org/ns#>
PREFIX rdf: <http://xmlns.com/foaf/0.1/>
SELECT ?item ?price
WHERE {
       ?item ns:price ?price .
       FILTER ( ?z > 10 ).
       {?item rdf:type "A"} UNION {?item rdf:type "B"}
}
```

2.2.1 Syntax SPARQL

Špecifikácia SPQRQL definuje ako základný grafový vzor trojicu (*triple pattern*). Trojice sú kombinované pomocou operátorov `OPTIONAL`, `UNION`, `GRAPH`, `FILTER` a operátora zreťazenia "`.`". Zoskupovanie grafových vzorov je možné pomocou operátorov { }. Výsledný *SPARQL graph pattern* môžeme definovať rekurzívnym spôsobom následovne:

1. *Trojica* $(s, p, o) = (I \cup V \cup L) \times (I \cup V) \times (I \cup V \cup L)$, kde I sú *IRI odkazy [12]* , V premenné a L literály, je grafový vzor (*triple pattern*).
2. Ak sú P_1 a P_2 grafové vzory, potom aj $(P_1 \,.\, P_2)$, $(P_1$ `OPTIONAL` $P_2)$, $(P_1$ `UNION` $P_2)$ sú grafové vzory (*conjunction graph pattern*, *alternative graph pattern*, *union graph pattern*).
3. Ak je P grafový vzor a F je SPARQL výraz, potom aj `FILTER (F, P)` je grafový vzor (*filter graph pattern*).

4. Ak sú P_1, ..., P_N grafové vzory potom aj { P_1. ....P_N } je grafový vzor *group graph pattern*.

5. Ak je GP *group graph pattern* a N je premenná alebo IRI, potom aj `GRAPH N GP` je grafový vzor *pattern on named graph*.

SPARQL výrazy sú tvorené prvkami množiny $I \cup L \cup V$, konštantami, logickými spojkami (`!`, `&&`, `||`), symbolmi (ne)rovnosti (`<`, `<=`, `>`, `>=`, `=`), zabudovanými operátormi `bound`, `isBlank` a ďalšími tak, ako sú definované v odporučení *W3C* [15].

Pre grafový vzor P si definujeme $var(P)$ ako množinu premenných, ktoré sa v danom vzore vyskytujú.

2.2.2 Sémantika SPARQL

Na úvod si uvedieme použitú terminológiu a definujeme základné operácie SPARQL algebry.

- *Mapovanie* μ z V do T, kde T je množina RDF termov definovaná ako $I \cup B \cup L$, je parciálna funkcia $\mu : V \to T$.
- *Doména* mapovania μ dom(μ) je definovaná ako podmnožina V, kde je funkcia μ definovaná.
- Mapovania μ_1 a μ_2 sú *kompatibilné*, ak pre všetky premenné $?X \in dom(\mu_1) \cap dom(\mu_2)$ platí $\mu_1(?X) = \mu_2(?X)$. Z toho taktiež vyplýva, že mapovania s disjunktnými doménami sú stále kompatibilné a prázdne mapovanie μ_0 ($dom(\mu_0) = 0$) je kompatibilné s každým mapovaním.
- Množinu mapovaní značíme Ω.

Majme množiny mapovaní Ω_1 a Ω_2 a výraz $expr$. Potom môžeme definovať operácie Filter, Join, Union a Diff následovne:

- $Filter(expr, \Omega) = \{\mu \mid \mu \in \Omega$ a $expr(\mu)$ nadobúda hodnotu $true\}$
- $Join(\Omega_1, \Omega_2) = \{\mu_1 \cup \mu_2 \mid \mu_1 \in \Omega_1$ a $\mu_2 \in \Omega_2$ a $\mu_,\mu_2$ sú kompatibilné$\}$
- $Union(\Omega_1, \Omega_2) = \{\mu \mid \mu \in \Omega_1$ alebo $\mu \in \Omega_2\}$
- $Diff(\Omega_1, \Omega_2, expr) = \{\mu \in \Omega_1 \mid \forall \mu' \in \Omega_2, \mu$ a μ' nie sú kompatibilné$\}$

Použitím takto definovaných operácií môžeme definovať operáciu *LeftJoin*, ktorá reprezentuje *optional graph pattern*:

- $LeftJoin(\Omega_1, \Omega_2, expr) = Join(\Omega_1, \Omega_2) \cup Diff(\Omega_1, \Omega_2, expr)$

Kompletná SPARQL algebra je definovaná v odporučení W3C [15].

2.2.3 Špecifiká SPARQL

Definícia SPQRQL algebry obsahuje niekoľko špecifických vlastností, ktoré ju odlišujú od klasickej relačnej algebry. Najdôležitejšie z nich si priblížime v následujúcich odstavcoch.

Neviazané premenné a reprezentácia chýbajúcej informácie. Na rozdiel od relačnej algebry, ktorá k reprezentácii chýbajúcich informácií používa špeciálnu hodnotu *NULL*, algebra SPARQL jednoducho necháva takéto premenné neviazané. SPARQL model nerozlišuje medzi *OPTIONAL* premennou, ktorá môže byť v niektorých výsledkoch neviazaná, a premennou, ktorá sa v požiadavke vôbec nevyskytuje. Relácie neobsahujú hlavičky a pracujú s aktuálnymi viazanými premennými.

Operácia Join a chýbajúce informácie. Chovanie operácií *Join* v algebre jazyka SPARQL sa líši od chovania ich ekvivalentov v regulárnej relačnej algebre, najmä vzhľadom k prístupu k chýbajúcej informácii. V tradičnej relačnej algebre sú odmietnuté n-tice dát, ktoré obsahujú *NULL* hodnotu v ľubovoľnom stĺpci zapojenom v operácii *Join*.

SPARQL algebra naproti tomu odmieta všetky n-tice, v ktorých sú v konflikte premenné viazané na oboch stranách. Pokiaľ je premenná neviazaná, považuje sa za kompatibilnú s ľubovoľnou hodnotou (a aj neviazanou premennou) na opačnej strane. Vďaka tomu je napríklad možné vytvoriť aditívnu požiadavku, ktorá doplňuje chýbajúce informácie medzivýsledkov.

Ukážme si takýto prípad na následujúcej požiadavke:

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
SELECT ?person, ?contact
WHERE {
      ?person foaf:mbox ?contact
      OPTIONAL(?person foaf:phone ?contact)
}
```

subject	predicate	object
_:a	foaf:mbox	a@email
_:a	foaf:phone	987654321
_:b	foaf:phone	123456789

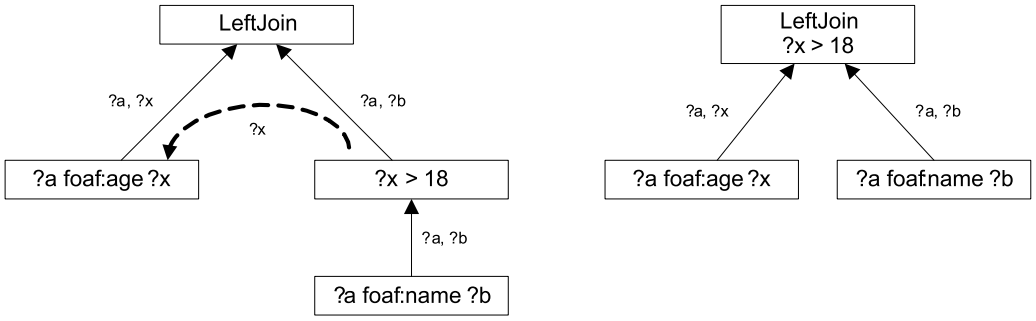
Tabuľka 2.1: Ukážkové dáta k príkladu aditívnej Join operácie.

?person	?contact
_:a	a@email
_:b	123456789

Tabuľka 2.2: Výsledok príkladu aditívnej Join operácie.

Do premennej `?contact` sa načíta emailová adresa. Pokiaľ táto nie je definovaná, ale je definované telefónne číslo, potom sa dosadí telefónne čislo.

Tabuľka 2.1 ukazuje vzorové dáta, kde osoba `_:b` nemá zadaný email, a osoba `_:a` má kompletné údaje. Tabuľka 2.2 ukazuje výsledok požiadavky, kde vidíme, že pre osobu `_:b` bol ako kontakt doplnený telefón, a pre osobu `_:a` ostal prvý dostupný údaj - email.



Obr. 2.3: (a) Rozšírenie pôsobnosti operácie Filter v `optional` vetve stromu operácií. (b) Ekvivalentná reprezentácia presunutím podmienky pod operáciu *LeftJoin*.

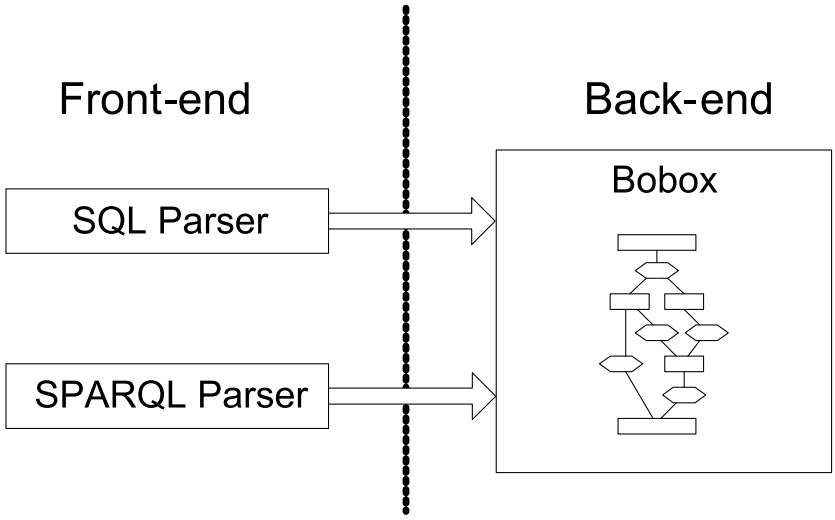
Pôsobnosť Filtov a grafový vzor `OPTIONAL` Ďalším špeciálnym prípadom je reprezentácia operácie *Filter* v kombinácii s grafovým vzorom `OPTIONAL`. Výraz v podmienke operácie *Filter* môže obsahovať ľubovoľné premenné, bez ohľadu na to, či sú, alebo kde sú definované. Filter má definovanú pôsobnosť v rámci vzoru *group graph pattern*, v ktorom sa nachádza. Táto pôsobnosť neplatí, ak sa *Filter* nachádza v *optional* podstrome operácie *LeftJoin* (obr. 2.3a), kedže z definície spracovania operácie *LeftJoin* vyplýva, že tieto filtre definujú jej internú podmienku na výsledné dáta. Takéto operácie *Filter* je teda nutné považovať za súčasť operácie *Filter* (obr. 2.3(b)).

2.3 Bobox

Systém Bobox je databázovo orientovaný framework, vyvíjaný na MFF UK, určený na paralelné a v budúcnosti aj distribuované výpočty [1]. Základnou myšlienkou je rozdelenie úlohy na jednotlivé elemetárne úlohy, podobne ako Intel Threading Building Blocks [16]. Tieto relatívne jednoduché úlohy sú usporiadné do nelineárneho reťazu spracovania úlohy. Spracovanie úloh je riadené „tokom dát“ a vykonávané paralelne. Jednotlivé komponenty výpočtu medzi sebou komunikujú pomocou správ. O synchronizáciu a plánovanie úloh sa stará samotný systém. Týmto je paralelizácia jednotlivým dielčím komponentom skrytá, čo zjednodušuje ich vývoj.

2.3.1 Architektúra

Bobox je zamýšľaný predovšetkým ako exekučné prostredie pre rôzne jazyky požiadaviek. Architektúru Boboxu je možné rozdeliť na dva celky tak, ako je to zobrazené na obrázku 2.4. *Front-end*, ktorý je závislý na jazyku požiadaviek a jeho úlohou je preloženie požiadavky do *modelu*, ktorý definuje spôsob spracovania. *Back-end* má na starosti samotné spracovanie a je spoločný pre rôzne *front-endy*.



Obr. 2.4: Rozdelenie architektúry systému Boboxu na *front-end* a *back-end*.

Model, ktorý je výstupom *front-endu*, je prevedený do *pipeline*, ktorá je tvorená dvoma základnými elementami:

- *Krabička* (*box*) je základná výpočtová jednotka. Môže mať 0 až n vstupov a 0 až m výstupov.
- *Via* predstavuje spojenie v *pipeline* a spája jeden výstup jednej *krabičky* s jednym, alebo niekoľkými vstupmi na iných *krabičkách*.

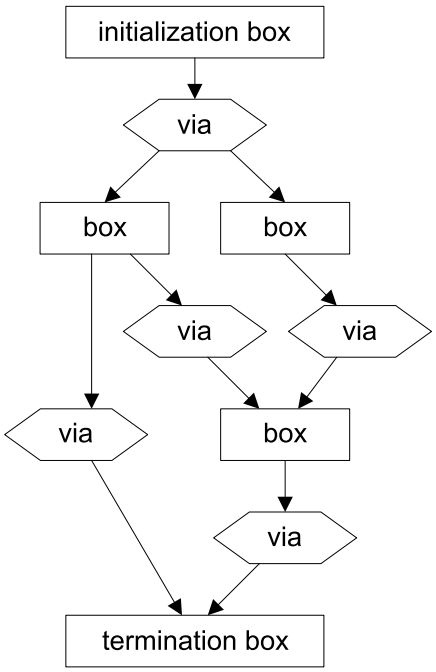
Dáta a informácie sú medzi jednotlivými *krabičkami* prenášané pozdĺž *via* po častiach, zabalené v štruktúre nazývanej *obálka* (*envelope*). Obálka predstavuje najmenšiu prenášanú jednotku dát.

Samotný výpočet (obr. 2.5 je naštartovaný pomocou inicializačnej krabičky (*initialization box*), ktorej úlohou je odoslanie jedinej správy na jej jediný výstup. Príchodom dát na vstup neaktívnej krabičky, je táto pridaná do skupiny úloh čakajúcich na spracovanie.

Úlohy sú spracovávané nad obmezeným počtom vlákien, ktorých počet a vytváranie je riadené plánovačom. Limitovaním počtu vlákien sa zabraňuje vytvoreniu ich veľmi veľkého množstva, čo by viedlo k zníženiu celkového výkonu systému.

V prípade, ak je krabička naplánovaná a spustená, môže v závislosti od dát na jej vstupoch nastať jedna zo situácií: Ak sú dostupné dostatočné dáta na vstupe, krabička ich bude spracovávať a generovať výstup. Pokiaľ nie sú dostupné všetky potrebné dáta, mala by svoj výpočet ukončiť. V prípade, že krabička stihla spracovať iba časť vstupu v pridelenom čase, je znovu naplánovaná. V opačnom prípade a taktiež aj v prípade nekompletných vstupov, už nedochádza k jej opätovnému naplánovaniu. Takáto krabička sa dostane medzi plánované krabičky až príchodom nových dát na jej ľubovoľný vstup.

Rýchlosť výpočtu jednotlivých vetiev nemusí byť rovnomerná. V takomto prípade sa môžu na vstupoch niektorých krabičiek hromadiť veľké množstvá dát od rýchlejších vetiev. Tieto dáta nemôžu byť spracované dovtedy, než pomalšie vetvy dodajú potrebné dáta. Na odstránenie tohto problému a vyrovnávanie rýchlosti spracovania jednotlivých vetiev výpočtu slúžia vyrovnávacie pamäte na vstupoch krabičiek. Pokiaľ dôjde k ich zaplneniu, dôjde k pozastaveniu plánovania krabičiek, ktoré tieto vstupy plnia. Takto uvoľnená výpočtová kapacita sa použije na urýchlenie pomalších vetiev.



Obr. 2.5: Ukážka modelu spracovania v Boboxe pomocou *pipeline*.

Kapitola 3

Optimalizácia požiadaviek v relačných jazykoch

V súčasnosti populárne relačné databázové systémy, ktorých počiatok vývoja siaha do 70-tych rokov minulého storočia, využívajú k prístupu k dátam deklaratíve relačné jazyky ako napríklad SQL. Spracovanie požiadavky zahŕňa postupnosť troch krokov: *dekompozícia*, *optimalizácia* a *vykonanie*. Počas *dekompozície* dochádza použitím logickej schémy k transformácii z relačnej požiadavky, do algebraickej požiadavky. Súčasťou tejto transofrmácie je aj lexikálna, syntaktická a sémantická analýza. *Optimalizácia* má za úlohu vytvorenie efektívnho plánu vykonania fyzických operácií požiadavky s prihliadnutím na množinu *search space*. *Vykonanie* požiadavky znamená vykonanie fyzických operácií v poradí, ako sú definované vo vytvorenom pláne a návrat výsledných dát.

Optimalizácia je kritickým krokom spracovania požiadavky [4] a je vykonávana optimalizátorom. Hlavné požiadavky, ktoré sú na ňu kladené sú:

- Minimalizácia času odpovede na požiadavku.
- Zvýšenie priepustnosti DBMS systému pri minimalizácii ceny za vykonanie požiadavky.

Pre dôležitosť a komplexnosť problému optimalizácie požiadavky [3], je databázovou komunitou vyvíjane obrovské úsile na vývoj prístupov, metód a techník optimalizácie pre rôzne DBMS.

Optimalizáciu môžeme definovať následovne:

Definícia 2 *Majme požiadavku q, plán vykonania operácií E a ohodnocujúcu funkciu cost(q) asociovanú s plánom $p \in E$. Potom optimalizácia je hľadanie takého plánu p vykonania požiadavky q, pre ktorý je jeho cena cost(q) minimálna.*

Optimalizátor môžeme rozdeliť do troch hlavných častí:

- *Priestor hľadania* (*search space*) predstavuje množinu plánov, ktorá je prehľadávaná. Táto je podmnožinou množiny všetkých ekvivalentných plánov požiadavky *plan space*, obmedzenej použitími heuristikami.
- *Prehľadávacia stratégia* (*Search strategy*) má za úlohu nájdenie optimálneho plánu z daného *priestoru hľadania*.
- *Ohodnocovací model* (*Cost model*) slúži k ohodnoteniu jedotlivých plánov v množine *search space*, aby ich bolo možné porovnávať. Kvalita optimalizácie je silno závislá na kvalite *ohodnocovacieho modelu*.

3.1 Prístupy optimalizácie

Pozrime sa na dva základné prístupy k optimalizácii z hľadiska času, kedy sa vykonáva: *statický prístup* a *dynamický prístup*.

3.1.1 Statický prístup

Statický prístup k optimalizácii je založený na postupnom generováni plánu výpočtu, na základe ktorého sú následne vykonané fyzické operácie. Tento plán je nemenný až do ukončenia vykonávania požiadavky. Tento prístup je používaný mnohými DBMS už viac ako dvadsať rokov. Jeho slabým miestom sú predpoklady na nemennosť dostupných zdrojov počas spracovania požiadavky a spoliehanie sa na presnosť použitého ohodnocovacieho modelu.

Avšak tieto predpoklady často nebývajú zaručené. Stav dostupných prostriedkov počas optimalizácie nemusí odpovedať dostupným prostriedkom počas vykonávania požiadavky. Taktiež odhady veľkosti dát zapojených do operácií nebývajú úplne presné, čo ovplyvňuje určovanie výslednej ceny. Nepresnosť odhadov plynie z nedostupnosti všetkých potrebných informácií, ich neplatnosti, alebo z použitia sumárných štatistík, ktoré sú použité namiesto reálnych dát z dôvodu efektívnejšieho spracovania. Vzniknuté chyby sú postupne propagované v strome operácií a navyše sa ukazuje, že ich rast je exponenciálny vzhľadom k množstvu zapojených operácií [9]. Z týchto dôvodov môžu byť nájdené plány neoptimálne. K zmierneniu nepresnosti odhadov sa môže napríklad pristúpiť k rozšíreniu a zkvalitneniu množiny dostupných informácií a štatistík.

3.1.2 Dynamický prístup

Dynamický prístup je založený na vylepšovaní neoptimálnych plánov, ktoré sa deje počas vykonávania požiadavky. Myšlienka dynamickej úpravy plánu počas spracovania je založená na snahe vyžiť aktuálne informácie o dostupných systémových zdrojoch, využiť medzivýsledky k spresneniu ohodnotenia a redukovaní niektorých nereálnych predpokladov *statického prístupu* (napríklad neobmedzená pamäť). Tento prístup robí tieto optimalizátory robustné vzhľadom k odhadu ceny a zmenám dostupných systémových prostriedkov.

3.2 Logické optimalizácie

Logické optimalizácie patria pod *statický prístup*. Pozostávajú z aplikovania prepisovacích pravidiel na strom operácií a ich cieľom je redukcia spracovávaných dát.

Medzi typické prepisovacie pravidlá patria napríklad:

- *Merging Views* - Ak požiadavka obsahuje pohľad (*view*), ten sa za určitých podmienok môže rozbaliť v rámci požiadavky a zúčastniť sa ďalších optimalizácií (napr. usporiadania operácií *Join*).

- *Merging nested subqueries* - Odstraňovanie vnorených podpožiadaviek, ktorého cieľom je sploštenie požiadavky jej prepísaním tak, aby pokiaľ je to možné, neobsahovala vnorené podpožiadavky.

- *Filter push down/pull up* - Propagácia operácie filter sa vykonáva za účelom zmenšovanie priebežných výsledkov. Aplikácia obmedzení definovaných filtrom čo najnižšie v strome operácií zredukuje ďalšie spracovávanie nevyhovujúcich dát.

- *Distinct push down/pull up* - Propagácia operácie distinct sa snaží o včasné odstraňovanie duplicít. Počas spracovania požiadavky môže dochádzať k nárastu veľkosti spracovávaných dát tvorených duplicitami. Ich skorým odfiltrovaním dôjde k redukcií priebežných výsledkov a eliminácií operácií potrebných k spracovaniu duplicít.

- *Priebežné projekcie* - Cieľom priebežných projekcií je eliminácia stĺpcov priebežných výsledkov, ktoré sa už nezúčastnia ďalšieho výpočtu a ani výslednej projekcie. Týmto dochádza k redukcií veľkosti dát medzivýsledkov a zníženiu pamätovej náročnosti.

- *Eliminácia redundantných operácií Join* za účelom zjednodušenia operácie SELECT.

3.3 Fyzické optimalizácie

Fyzické optimalizácie patria pod *statický prístup* a sú zamerané na splnenie dvoch cieľov:

- Určenie vhodných fyzických operátorov reprezentujúcich operáciu, pričom sa berie do úvahy: veľkosť dát vstupujúcich do relácií, fyzická štruktúra, vlastnosti a dostupnosť dát.
- Generovanie usporiadania operácií *Join* s ohľadom na *ohodnocovací model*.

V tejto kapitole si definujeme a odhadneme veľkosť priestoru hľadania (*search space*) a pozrieme sa na metódy jeho prehľadávania (*enumeration methods*).

3.3.1 Priestor hľadania

V relačných databázových systémoch, môžeme plán vykonania požiadavky reprezentovať pomocou stromu, kde listy sú reprezentované základnými reláciami a vrcholy predstavujú operácie. Podľa tvaru stromov operácií rozoznávame tri základné typy (obr. 5.3): *left-deep*, *right-deep* a *bushy* stromy. Množstvo plánov tvoriacich priestor hľadania exponenciálne narastá s počtom operácii Join [11]. Takýto priestor sa stáva ťažšie spracovateľný, preto sa môže priestor hľadania obmedziť vzhľadom na použitú prehľadávaciu stratégiu a spôsob výpočtu. Typickým príkladom je snaha o obmedzenie *bushy* stromov. *Left-deep*, resp. *right-deep* stromy sú preferované vďaka umožneniu zreťazeného spracovania, keď priebežné výsledky operácii Join nie sú ukladané, ale priamo vstupujú do výpočtu nasledujúcich operácií.

3.3.2 Prehľadávacie stratégie

Prehľadávacie stratégie (*search strategies*) definujú spôsob generovania optimálneho plánu z definovaného priestoru plánov. Všeobecne sa rozoznávajú dve triedy:

- *enumeratívne stratégie* [10, 8]
- *randomizované stratégie* [8]

Enumeratívne stratégie

Enumeratívne stratégie sú založené na postupnom generovaní výsledného plánu. Využívajú k tomu princíp dynamického programovania. Začínajú s množinou všetkých relácií a postupne budujú strom operácií. V každom kroku využívajú informácie vyrátané v predchádzajúcom kroku. V poslednom kroku je zvolený najlacnejší strom, ktorý predstavuje hľadaný plán. Pri hľadaní sa prechádza celým priestorom hľadania, ale jeho exponenciálna zložitosť vedie k používaniu efektívnejších algoritmov [10]. Tieto uvedením heuristík, ako napríklad prehľadávanie do hĺbky, umožňujú priebežne eliminovať neoptimálne vetvy výpočtu.

Randomizované stratégie

Randomizované stratégie sa snažia riešiť problém enumeratívnych stratégií s veľkosťou priestoru hľadania pri veľkých požiadavkách. Sú založené na transformovaní neoptimálneho plánu. Z transformácií môžeme spomenúť napríklad [6, 9] Swap, ktorá vymieňa dve susediace relácie v zozname, alebo 3Cycle, ktorá cyklicky rotuje tri susediace relácie v zozname relácií.

Medzi randomizované algoritmy patria napríklad Simulate Annealing, alebo Iterative Improvement [6]. Tieto reprezentujú priestor hľadania ako graf, kde je každý plán reprezentovaný ako vrchol a hrana medzi vrcholmi reprezentuje transformáciu medzi plánmi reprezentujúcimi vrcholy. Algoritmy začínajú s neoptimálnymi plánmi a pomocou náhodných prechádzok v grafe sa snažia nájsť globálne minimum, reprezentujúce optimálny plán.

Kapitola 4

Prekladač SPARQL pre Bobox

Táto kapitola sa zaoberá implementáciou prekladača SPARQL pre Bobox a použitými štuktúrami a postupmi. Na úvod sú uvedené predpoklady a obmedzenia, na ktorých je práca založená. Následuje popis použitých štruktúr a popis implementácie optimalizátora, podrobnejší rozbor použitých metód zjednodušovania a optimalizácie požiadavky.

4.1 Predpoklady a obmedzenia

Úplné pokrytie všetkých možností jazyka SPARQL z pohľadu sémantickej analýzy a optimalizácie je rozsahovo nad možnosti tohto typu práce, preto bola definovaná skupina predpokladov a obmedzení.

Optimalizácia požiadavky a jej rozdelenie na dielčie úlohy je vykonávané s ohľadom na spracovanie pomocou systému Bobox. Výsledný plán vykonania úlohy je rozdelený na rozumne malé a jednoduché části, ktoré zodpovedajú jednotlivým *krabičkám* Boboxu.

4.1.1 Zdroj dát

Potreba štatistík, ktoré slúžia k odhadom ohodnotenia jednotlivých variant výpočtu a kontroly prístupu k dátam, predpokladá prácu nad lokálnou databázou. Dôsledkom toho je, že sa v tejto verzii práce neberú do úvahy zdroje grafov zadané pomocou príkazov `FROM` a `FROM NAMED`. V dôsledku toho dochádza aj k obmedzeniu príkazu `GRAPH`.

4.1.2 Vyhodnocovanie výrazov

Neznalosť dátového typu premenných v dobe prekladu znamená, že prekladač sa nesnaží o úplné vyhodnocovanie a kontrolu SPARQL výrazov. K tomu

dochádza až v priebehu vykonávania požiadavky. Neznalosť dátového typu v dobe sémantickej analýzy je spôsobená jeho závislosťou od typu aktuálne spracovávanej položky dát. Navyše jedna premenná môže nadobúdať rôzne dátové typy pre rôzne *trojice* v sade dát.

4.2 Fyzická schéma databáze

Efektívna a škálovatelná správa *RDF dát* je veľkou výzvou a objektom aktivného výskumu. Vzniklo niekoľko návrhov systémov spoliehajúcich sa na využitie relačnej infraštruktúry, pretože RDBMS (*Relation Database Management Systems*) systémy ukázali, že dokážu úspešne hosťovať rôzne typy dát, aj keď pre ne neboli pôvodne určené. Tieto návrhy systémov môžeme rozdeliť do následujúcich tried:

- **Tabuľka trojíc**(*Triples table*). Každá *RDF trojica* je uložená priamo v trojstĺpcovej tabuľke so stĺpcami subjekt, predikát a objekt.
- **Tabuľky vlastností** (*Property tablee*). Skupiny *RDF* vlastností (*predikátov*) tvoriacich stĺpce n-árnych tabuliek so spoločným *subjektom* ako kľúčom.
- **Binárne tabuľky** (*Binary tables*).textitRDF trojice sú rozložené na dvojstĺpcové tabuľky reprezentujúce jednotlivé vlastnosti so stĺpcami subjekt a objekt.

Výkon týchto spôsobov uloženia v porovnaní s tradičnou relačnou schémou porovnávali napríklad Sakr a MahmoudiNasab [13]. *Binárne tabuľky* a *tabuľky trojíc* sú výkonovo približne na rovnakej úrovni, avšak oproti *tabuľkám vlastností* a klasickej *relačnej schéme* v niektorých prípadoch zaostávajú. *Tabuľky trojíc* si uchovávajú svoju schopnoť jednoducho ukladať ľubovoľné trojice. Vzhľadom na ich veľkosť sa ich výkon snaží napraviť intenzívnejšie použitie indexov. *Binárne tabuľky* sú založené na myšlienke rýchleho prístupu k vlastnostiam záznamov. Výkon týchto tabuliek je degradovaný nárastom počtu vlastností v dátach a používaním spojení *subjek-objekt* a *objekt-subjekt*. *Tabuľky vlastností* sa snažia pokryť medzeru vo výkone medzi predchádzajúcimi dvoma spôsobmi a klasickou *relačnou schémou*. Ich veľkým problémom je potreba poznať schému dát, ktoré budú v nich uložené a problémy pri ukladaní dát nezodpovedajúcich tejto schéme.

Po zvážení výhod a nevýhod bola ako podporovaná fyzická schéma zvolená reprezentácia pomocou *tabuľky trojíc*. Táto, hoci výkonovo nepatrí k najlepším, je veľmi univerzálná, pretože umožňuje ukládať *RDF trojice* z ľubovoľnej schémy, bez nutnosti poznať jej štruktúru.

Subject	Predicate	Object
Id1	publishedIn	ITAT 2009
Id1	hasTitle	The Bobox Project...
Id1	authoredBy	Id1
Id2	hasName	Jiri
Id2	affiliatedBy	CUNI
Id2	roomNo	209
Id1	year	2009
Id1	editedBy	Id3
Id3	hasName	Jakub
Id3	affiliatedBy	CUNI

Obr. 4.1: Relačná reprezentácia *Tabuľky trojíc*

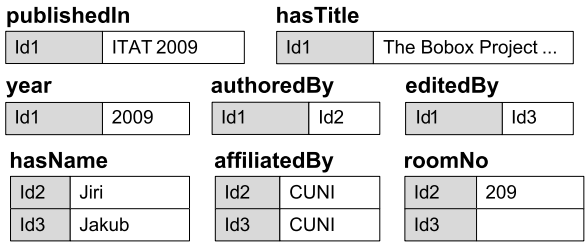
Publication

ID	publishedIn	hasTitle	year	authoredBy	editedBy
Id1	ITAT 2009	The Bobox Project...	2009	Id2	Id3

Person

ID	hasName	affiliatedBy	roomNo
Id2	Jiri	CUNI	209
Id3	Jakub	CUNI	

Obr. 4.2: Relačná reprezentácia *Tabuľkiek vlastností*



Obr. 4.3: Relačná reprezentácia *Binárnych tabuliek*

4.3 Štruktúra prekladača

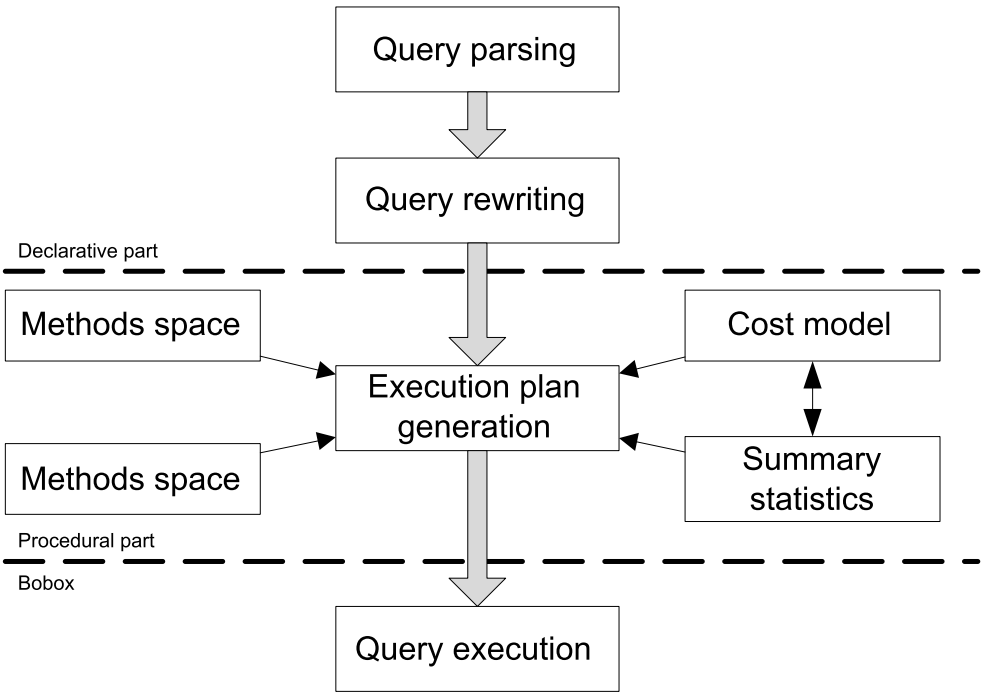
Spracovanie požiadavky prebieha v niekoľkých krokoch, ktoré sú vykonávané jednotlivými časťami aplikácie. Priebeh, tak ako je zobrazený na obrázku 4.4, pozostáva z nasledujúcich krokov:

- *Parsovanie požiadavky (Query parsing)* - v tomto kroku prebieha lexikálna, syntaktická a sémantická analýza.
- *Prepisovanie požiadavky (Query rewriting)* - aplikujú sa logické prepisovacie pravidlá. Účelom je požiadavku zjednodušiť odstránením redundantných, alebo nesplnitelných častí, vykonať logické optimalizácie

na úrovní `group graph patternov` a zjednotenie zápisu, napríklad pre umožnenie jej cachovania v budúcnosti.

- *Generovaniu plánu vykonávania (Execution plan generation)* - hľadanie optimálneho plánu vykonania fyzických operátorov (*methods space*) reprezentujúcich operácie, vzhľadom k definovanému priestoru hľadania (*search space*). Optimálny plán má minimálnu cenu, ktorá je určená ohodnocovacím modelom (*cost model*). Táto je ovplyvnená typom operácie, veľkosťou a typom dát, ktoré má spracovávať. K odhadom veľkosti a typu dát slúžia sumárne štatistiky (*summary statistics*).

- *Vykonávanie požiadavky (Query execution)* - samotné spracovanie požiadavky Boboxom.



Obr. 4.4: Štruktúra prekladača.

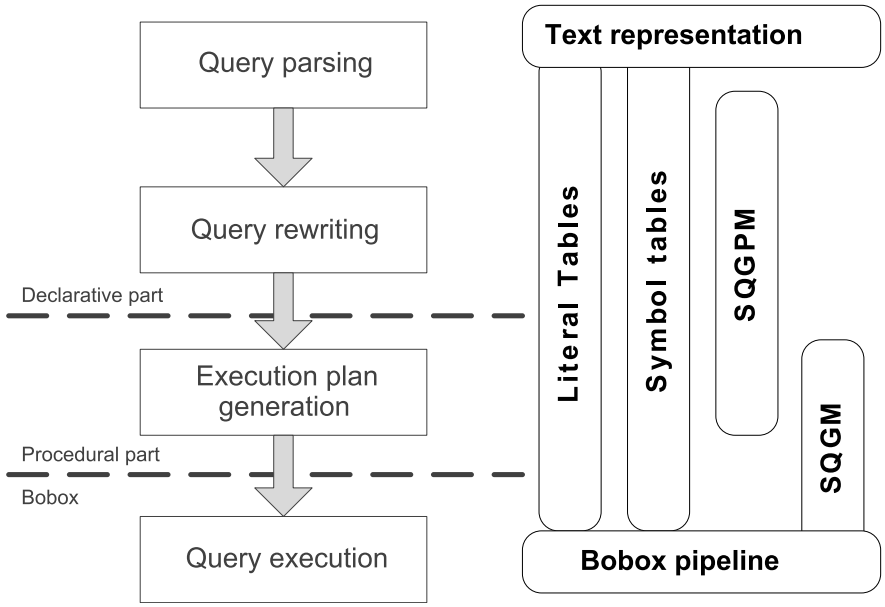
Parsovanie požiadavky je neoddeliteľná súčasť prekladača, ktorá sa stará o kontrolu zadanej požiadavky a vytvorenie jej reprezentácie pre ďalšie strojové spracovanie. Kroky *prepisovanie požiadavky* a *generovanie plánu vykonavania* sú zamerané na optimalizáciu za účelom efektívnejšieho využívania dostupných výpočtových kapacít. Posledný krok vykonávania požiadavky už nie je úlohou prekladača, ale *databázového systému*, v tomto prípade Boboxu.

4.4 Parsovanie požiadavky (Query parsing)

Lexikálna a syntaktická analýza je vykonávaná podľa gramatiky definovanej v odporučení SPARQL od W3C [15] vo významovo nezmenenej podobe. Definícia je prepísaná do definícií použitých voľne dostupných nástrojov *Flex* a *Bison*.

4.5 Reprezentácia dát

V priehehu spracovania požiadavky sa použije niekoľko jej reprezentácií. Na obrázku 4.5 sú znázornené použité reprezentácie vo vzťahu k jednotlivým krokom prekladača. Následujúce podkapitoly približujú tieto jednotlivé reprezentácie.



Obr. 4.5: Reprezentácie požiadavky vo vzťahu k jednotlivým krokom prekladača.

4.5.1 Tabuľky literálov a symbolov

Tabuľky literálov slúžia k uloženiu hodnôt jednotlivých literálov tvoriacich požiadavku. Tabuľky sú plnené pri lexikálnej analýze a odkazy do tabuľky literálov slúžia na ich reprezentáciu v nasledujúcich krokoch. Opakovaný výskyt rovnakej hodnoty je reprezentovaný jediným záznamom v tabuľke literálov. Týmto dochádza k zjednodušeniu častých operácií rovnosti, hlavne

nad veľkými textovými literálmi a zníženiu pamäťovej náročnosti. K dispozícii sú tabuľky pre dátové typy *int*, *double*, *decimal*, *string* a tabuľka pre reťazcové identifikátory.

Tabuľka symbolov slúži na ukládanie definícií symbolov definovaných počas syntaktickej a sémantickej analýzy. Symboly reprezentujú numerické, logické a textové hodnoty, funkcie, *IRI* odkazy a premenné.

4.5.2 SPARQL Query Graph Model

Pirahesh a kol. navrhli grafový model požiadavky (*Query Graph Model*[14]), určený na reprezentáciu SQL požiadaviek. Hartig a Reese [5] tento model upravili do *SPARQL Query Graph Model (SQGM)* tak, aby reprezentoval SPARQL požiadavky.

Tento model sme zvolili k reprezentácii postupu výpočtu, pretože pri vhodnej definícii vlastných operátorov (tabuľka 4.1) reprezentuje strom fyzických operátorov a je triviálne transformovatelný na *pipeline* Boboxu. Vzhľadom k obmedzeniu na prácu s lokálnou databázou (4.1.1) a efektívnejšiemu prístupu k dátam s využitím indexov, táto implementácia nevyužíva osobitné operátory zamerané výhradne na načítanie zdrojových grafov. K dátam pristupuje iba *Graph Pattern Operátor*, ktorý aplikovaním zadaného vzoru hľadá zodpovedajúce *trojice* dát a viaže premenné.

Nasledujú definície základných pojmov *operátor*, *dátový tok* a *ekvivalencia modelov*.

Definícia 3 *Operátor vykonáva operáciu nad svojími vstupnými dátami a generuje dáta na svoj výstup.*

Hrana symbolizuje *dátový tok* medzi dvoma operátormi. Hrany sú orientované a smerujú ku konzumerovi dát.

Definícia 4 *Dátový tok spája dva operátory a prenáša dáta poskytované jedným z nich a spotrebúvané druhým.*

Definícia 5 *Query Graph Model(SQGM) Reprezentuje SPARQL požiadavku. Je to trojica (OP, DF, r), kde*

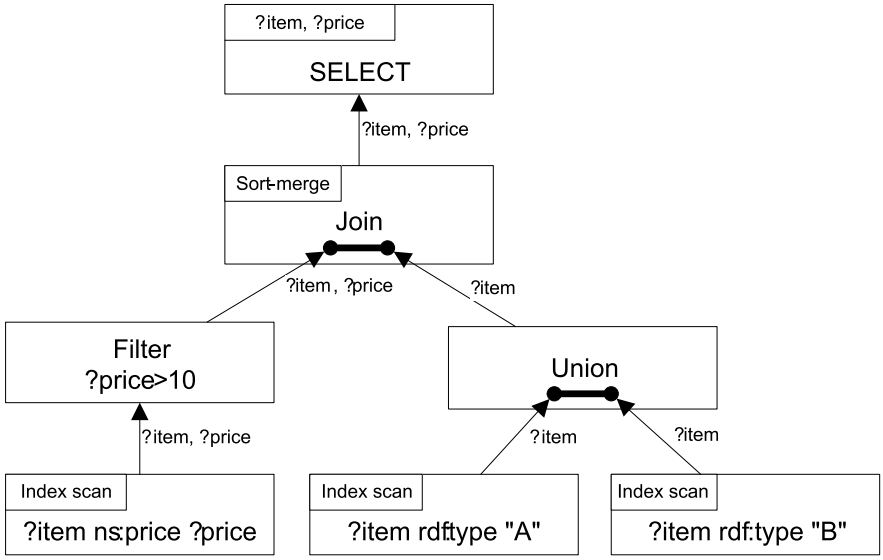
- *OP značí množinu všetkých operátorov potrebných k modelovaniu požiadavky.*
- *DF značí množinu všetkých potrebných dátových tokov k modelovaniu požiadavky.*
- *r je operátor zodpovedný za generovanie výsledku požiadavky ($r \in OP$)*

Definícia 6 *Dva SQGM modely požiadaviek q a q' sú sémanticky ekvivalentné, ak rovnosť*

$$Result_D(q) = Result_D(q')$$

pre ľubovoľnú sadu RDF dát D, kde $Result_D(q)$ *označuje výslednú množinu spracovania požiadavky q nad D.*

SQGM môžeme graficky interpretovať ako orientovaný graf, presnejšie orientovaný strom, s vrcholmi a hranami reprezentujúcimi operátory a dátové toky (obrázok 4.6). Operátory sú znázornené ako krabičky obsahujúce hlavičku, telo a prípadné anotácie. Dátové toky sú znázornené pomocou šípky v smere toku a prípadnej anotácie.



Obr. 4.6: Grafická reprezentácia SQGM pre SPARQLpožiadavku z úvodu kapitoly 2.2

4.5.3 SPARQL Query Graph Pattern Model

SPARQL Query Graph Pattern Model (SQGPM) model reprezentuje požiadavku na úrovni grafových vzorov, ktorých zoznam je v tabuľke 4.2. Požiadavku teda definuje na abstraktnejšej úrovni než SQGM model, ktorý reprezentuje konkrétne usporiadanie konkrétnych operácií. Model je budovaný počas syntaktickej analýzy. V priebehu kroku prepisovania (*query rewriting)* dochádza k jeho úpravám a pre *optimalizátor* slúži ako zdroj dát pri budovaní výsledného SQGM stromu.

Operátor	**Význam a vlastnosti**
Scan Operator	Operátor reprezentujúci jednu z *trojíc* vyskytujúcich sa v požiadavke. *typ*: definuje očakávaný prístup k dátam (*Full table scan, indexy, hashovacie tabuľky).* *index*: určuje stĺpce definujúce index, ktorý sa má použiť k prístupu k dátam.
Join Operator	Spojí dve množiny viazaných premenných. *typ*: definuje spôsob spojenia množin. (*Nested loops, Merge join, ...).*
Union Operator	Vytvorí zjednotenie množín viazaných premenných.
Empty Group Operator	Vytvorí množinu s jediným výsledkom, ktorý nemá viazanú žiadnu premennú.
Graph Operator	Pristupuje k množine pomenovaných *RDF grafov.* *var*: Premenná viazaná k *IRI* adrese vybraného *RDF grafu*, alebo *IRI* adresa označujúca *RDF graf*, na ktorý sa má obmedziť matchovanie.
Filter Operator	Aplikuje obmedzenia definované výrazom na množinu viazaných premenných. *expr* výraz definujúci obmedzenia.
Solution Modifier Operator	Aplikuje modifikátory na množinu viazaných premených. *limit*: definuje maximálnu veľkosť množiny. *offset*: určuje polohu začiatku výslednej množiny. *order_by*: definuje usporiadanie množiny. *distinct*: definuje, či majú byť ponechané duplicitné hodnoty.
Select Operator	Vykoná finálnu projekciu dát, podľa definície nastavenej v hlavičke.
Describe Operator	Vracia jeden *RDF graf* popisujúci výsledné dáta.
Construct Operator	Vracia *RDF graf* vytvorený podľa zadanej šablony dosadením viazaných premenných.
Ask Operator	Vracia "yes" pokiaľ výsledok nie je prázdny. V opačnom prípade vracia "no"

Tabuľka 4.1: Zoznam operátorov definovaných v SQGM

Triple Pattern	Vzor zodpovedá jednej trojici *subjekt*, *predikát*, *objekt*.
Group Graph Pattern	Skupina vzorov.
Optional Graph Pattern	Skupina vzorov, rozšírená o ďašiu skupinu vzorov, ktoré môžu rozšíriť existujúci výsledok.
Alternative Graph Pattern	Zlúčenie dvoch alternatívnych skupín vzorov.
Patterns On Named Graphs	Skupina vzorov, ktoré sú „matchované“ oproti pomenovaným grafom.
Filter Graph Pattern	Definícia obmezení pre vzory z aktuálnej skupiny.

Tabuľka 4.2: Zoznam grafových vzorov reprezentovaných v SQGPM.

Telo požiadavky je reprezentované stromovou štruktúrou, kde jednotlivé vrcholy stromu sú reprezentované skupinovými grafovými vzormi (*group graph patterns*). Vrchol obsahuje zoznam ďalších grafových vzorov, ktoré sú v ňom obsiahnuté. Ich poradie nie je pevné dané, avšak tvoria množinu vrcholov SQGM podstromu spojených operáciou *Join - join stromu*. Pokiaľ tieto vnorené vzory obsahujú skupinové vzory, tieto sú na ne viazané ako ďalšie vrcholy stromu. Obrázok 4.7 ukazuje grafickú reprezentáciu SQGPM stromu požiadavky.

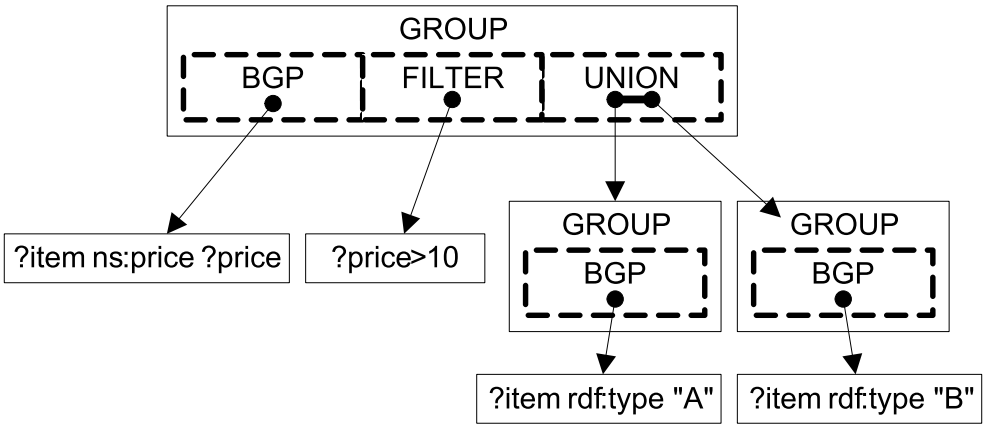
Modifikátory požiadavky sú uložené v osobitných tabuľkách. Tento model zachováva hierarchiu skupín grafových vzorov a usporiadanie medzi operáciami (`OPTIONAL`), ktoré sú nekomutatívne a neasociatívne. Definovaním týchto pevných (skupiny grafových vzorov) a voľných(zoznam grafových vzorov vo skupine) častí sa uľahčuje následná optimalizácia a budovanie výsledného stromu operácií, rozdelením budovania výsledného stromu operácií na budovanie jednotlivých *join-stromov*.

SQGPM model môžeme formálne definovať ako:

Definícia 7 *SQGPM je množina (NG, GP, r) kde*
-NG je množina všetkých vrcholov,
-GP je množina všetkých grafových vzorov,
-r je group graph pattern predstavujúci graph pattern požiadavky.

Množinu prvkov $p \in GP$, ktoré patria vrcholu v ($p \in g$) označujeme List(v), kde $v \in NG$.

Definícia 8 *Nech g je group graph pattern požiadavky q. Potom existuje práve jeden SQGPM vrchol $v \in GGP$, ktorý reprezentuje g.*



Obr. 4.7: SQGPM graf požiadavky z úvodu kapitoly 2.2

Definícia 9 *Nech $v \in GGP$ je SQGPM vrchol a g je group grap pattern, ktorý reprezentuje. Potom pre každý grafový vzor (graph pattern) $p \in GP$, ktorý je priamym potomkom g platí $p \in v$*

Definícia 10 *Dva SQGPM modely požiadaviek q a q' sú sémanticky ekvivalentné, ak sú všetky SQGM modely z nich odvodené navzájom ekvivalentné pre ľubovoľnú sadu RDF dát.*

4.5.4 Návrh operácií pre Bobox

Zoznam navrhnutých *krabčiek* pre Bobox, ktorý pokrýva požiadavky práce, je uvedený v tabuľke 4.3.

Krabička	**Význam a vlastnosti**
Full Table Scan	Hľadá v dátach trojice, ktoré vyhovujú predpisu a viaže zodpovedajúce premenné. K dátam fyzicky pristupuje ich postupným prechádzaním.
Index Scan	Hľadá v dátach trojice, ktoré vyhovujú predpisu a viaže zodpovedajúce premenné. K dátam fyzicky pristupuje pomocou idexov.
Hash Table Scan	Hľadá v dátach trojice, ktoré vyhovujú predpisu a viaže zodpovedajúce premenné. K dátam fyzicky pristupuje pomocou hashovacích tabuliek.
NestedLoopsJoin	Aplikuje operáciu *Join*, resp. *LeftJoin* metódou vnorených cyklov.
MergeSortJoin	Aplikuje operáciu *Join*, resp. *LeftJoin* spájaním usporiadaných vstupov. Ak vstupy nie sú vhodne usporiadané, potom ich najprv usporiada.
Filter	Aplikuje operáciu *Filter* na dáta na vstupe.
Graph	Ak je ako názov grafu premenná, potom pre ňu nastaví *IRI* adresu grafu, z ktorého pochádzajú dáta z vnorenej skupiny operátorov.
Union	Vykoná operáciu *Union* na vstupných dátach.
Modify	Aplikuje nastavené modifikácie dát: usporiadanie, zahodenie duplicitných záznamov, obmedzenie veľkosti dát, alebo zahodenie prvých m záznamov.
Empty group	Vygeneruje množinu o veľkosti 1, ktorá nemá viazané žiadne premenné.
Select	Vykoná projekciu na množine vstupných dát.
Ask	Ak je vstupná množina dát neprázdna, potom vráti hodnotu "yes".
Describe	Vytvorí *RDF graf*, ktorý popisuje výslednú množinu.
Construct	Vytvorí *RDF graf* použitím zadanej šablóny.

Tabuľka 4.3: Zoznam navrhnutých krabičiek pre systém Bobox

Kapitola 5

Optimalizácia požiadavky

Motivácia stojaca za optimalizáciou požiadavky, je efektívnejšie využívanie dostupných zdrojov a rýchlejšie spracovanie požiadaviek. Rozdiely medzi výkonom jednotlivých možností spracovania jednej požiadavky môžu byť priepastné. Spravidla platí, čím je požiadavka zložitejšia, tým rozdiel narastá. Zvolené spôsoby optimalizácie môžeme rozdeliť na *logické* a *fyzické*.

Logické optimalizácie sa vykonávajú aplikovaním sady prepisovacích pravidiel na SQGPM model. Ich úlohou je zjednodušovanie požiadavky a zmenšovanie medzivýsledkov. *Fyzické optimalizácie* využívajú vlastnosti operácie *Join* a dynamického programovania pri hľadaní ich najvhodnejšieho usporiadania. Aby bolo možné jednotlivé usporiadania porovnávať, sú k dispozícii *ohodnocovacie funkcie*, *štatistiky* a *funkcie na odhady veľkosti dát*.

5.1 Prepisovanie požiadavky

Od spracovávaných požiadaviek sa neočakáva, že budú optimálne zadané. Požiadavky môžu obsahovať redundantné časti (napríklad duplicitné zátvorky, grafové vzory, ktoré nemajú možnosť ovplyvniť výsledok), rôzne rozdelený zápis `Filtov`, nesplitelné podmienky a podobne. Úlohou kroku prepisovania požiadavky je odhaliť takéto prípady a vysporiadať sa s nimi.

Naša implementácia vykonáva následujúce operácie, ktoré si následne podrobnejšie popíšeme:

- Zlučovanie vnorených *Group graph patternov*.
- Odstraňovanie duplicít.
- Zmenšovanie medzivýsledkov.
- Zjednocovanie tvaru požiadavky.

Požiadavka je v tomto kroku spracovania reprezentovaná pomocou SQGPM modelu, zameraného na reprezentáciu grafových vzorov a vzťahov medzi skupinami grafových vzorov. To určuje aj zameranie pravidiel prepisovania. Výsledkom tejto *logickej optimalizácie* je ekvivalntný SQGPM model, ktorý je možné efektívnejšie vyhodnotiť, resp. je možné pre neho nájsť počas nasledujúcej *fyzickej optimalizácie* výhodnejší plán spracovania .

5.1.1 Zlučovanie vnorených *Group graph pattern*ov

Úlohou zlučovania vnorených skupín je nájdenie takých *group graph patternov*, ktoré môžu byť zlúčené s *group graph patternom*, v ktorom sa nachádzajú, pri zachovaní ekvivalentnosti ich SQGPM modelov. Príklad 1 ukazuje výhody zlučovania.

Príklad 1 *Majme požiadavku:*

SELECT $*$
WHERE { $?x$ $?\mathbf{a}$ $?b$. { $?x$ $?y$ $?z$. { $?\mathbf{a}$ $?b$ $?c$ } } }

Takéto zátvorkovanie trojíc necháva ďalšiemu spracovaniu požiadavky jedinú možnosť usporiadania operácií:

Join(?x ?a ?b, Join(?x ?y ?z, ?a ?b ?c))

Pri tomto usporiadaní generuje vnorená operácia Join kartézsky súčin, ktorý je následne spracovaný druhou operáciou Join, čo vedie k dlhému spracovaniu požiadavky. Táto skupina trojíc môže byť ekvivalente zapísaná ako:

SELECT $*$
WHERE { $?x$ $?\mathbf{a}$ $?b$. $?x$ $?y$ $?z$. $?\mathbf{a}$ $?b$ $?c$ }

Tento zápis umožňuje väčšiu voľnosť pri voľbe usporiadania operácií Join. Umožňuje napríklad nájsť výhodnejšie usporiadanie operácií:

Join(?x ?y ?z, Join(?x ?a ?b, ?a ?b ?c))

Takéto usporiadanie nevytvára kartézsky súčin a pracuje s menšími výslednými množinami a teda je predpoklad, že je výrazne efektívnejšie.

Avšak nie každý *group graph pattern*(GGP) je zlúčiteľný s rodičovským GGP. Problém nastáva, pokiaľ GGP môže obsahovať neviazané premenné a obsahuje `FILTER`, ktorý na nich definuje obmedzenia. Príčinou problému je to, že u neviazaných premenných môže dôjsť pre nejaký riadok k ich zmene - naviazaniu. Pri zmene rozsahu platnosti operácie *Filter* zlúčením GGP, môže táto následne daný riadok vyhodnotiť rozdielne, čo má za následok zmenu výslednej množiny dát. U viazaných premenných k zmene hodnoty dôjsť nemôže, preto sú vzhľadom k operácii *Filter* bezpečné. Príklad 2 demonštruje prípad, kedy nie je možné aplikovať zlúčenie GGP.

Príklad 2 *Majme požiadavku:*

```
SELECT *
WHERE {
        ?s rdf:type ?t .
        {P . FILTER(bound(?t)) }
}
```

kde P reprezentuje skupinu grafových vzorov, ktorých výsledná množina obsahuje premennú `?s` *a premennú* `?t`*, ktorá môže byť neviazaná. Potom pôvodná reprezentácia odmietne všetky n-tice s neviazanou premennou* `?t` *pred spojením s trojicou v rodičovskom GGP. Naproti tomu, ak by sa tento vnorený GGP spojil s rodičovským, potom sa Filter vykoná až po vykonaní všetkých operácii Join, pri ktorých môžu niektoré n-tice s neviazanou premennou* `?t` *túto naviazať. Takéto n-tice už Filter neodstráni a sú teda navyše.*

Definícia 11 *Majme dva vrcholy SQGPM v, v' pre ktoré platí, že $v' \in v$. Ak pre všetky grafové vzory p, $p \in v'$ platí: $p \neq$* `FILTER` *alebo p neobsahuje neviazanú premennú, potom $List(v) = (List(v)/v') \cup List(v')$*

5.1.2 Odstraňovanie duplicitných grafových vzorov

Účelom tejto operácie je teda odstránenie takýchto duplicitných grafových vzorov. Problém duplicitných vzorov si predvedieme na nasledujúcom príklade:

Príklad 3 *Majme požiadavku:*

```
SELECT DISTINCT *
WHERE {
        ?obj rdf:type ?t .
        ?obj rdf:type ?t
}
```

Táto požiadavka obsahuje dve rovnaké trojice `?obj rdf:type ?t` *a nepovoľuje duplicitné výsledky. Vykonanie druhej trojice a jej spojenie s prvou v tomto prípade nevytvorí žiadné nové výsledky. Pretože spojením identických množín, existuje pre každý prvok množiny minimálne jeho identita v druhej. Keďže sú prítomné iba viazané premenné, neexistuje kombinácia riadkov, ktorá by vytvorila nový unikátny riadok. Veľkosť výslednej množiny je teda minimálne rovnaká ako veľkosť vstupných množín operácie, prípadne navýšená o možné duplicity. Aplikovaním obmedzenia na unikátne dáta, odstránime duplicity a dostávame rovnosť zlúčenia identických množín s pôvodnými množinami.*

Podmienky pre uskutočnenie tejto operácie sú

- Duplicitný vzor nesmie obsahovať neviazané premenné.
- Požiadavka musí byť typu `DISTINCT`, alebo `REDUCED`.

Požiadavka na vlastnosť `DISTINCT`, alebo `REDUCED` je z dôvodu vysporiadania sa s prípadnými duplicitami. Prítomnosť neviazaných premenných nezaručuje identitu výsledkov, ako to ilustruje následujúci príklad.

Príklad 4 *Majme požiadavku, ktorej duplicitné grafové vzory generujú nasledujúce identické množiny* {{`?x=1`}, {`?y=2`}}. *Aplikovaním operácie Join dostaneme množinu* {{`?x=1`}, {`?x=1, ?y=2`}, {`?x=1, ?y=2`}, {`?y=2`}}, *ktorá obsahuje nové unikátne riadky a teda nie je ekvivalentná s pôvodnou množinou ani po aplikácií vlastnosti* `DISTINCT`, *alebo* `REDUCED`.

5.1.3 Zmenšovanie medzivýsledkov

Jednou z ďalších skupín operácii k zníženiu pamäťovej náročnosti a urýchleniu vykonávania požiadavky, sú optimalizácie zamerané na zmenšovanie medzivýsledkov operácii. Neustále však platí podmienka, že výsledná množina dát musí ostať nezmenená. Do tejto skupiny optimalizácii sa radí:

- Propagácia operácie `Filter`.
- Priebežná projekcia premených.
- Propagácia operácií `Distinct`/`Reduced`.

Propagácia operácie Filter

Propagácia operácie *Filter*, označuje jej presun čo najbližšie k operátorom, ktoré poskytujú všetky premenné definované v rámci výrazu tvoriaceho danú operáciu *Filter*. Včasným filtrovaním medzivýsledkov sa zmenšuje objem dát. To má za následok menšiu réžiu pri prenášaní dát medzi operátormi, urýchlenie následných operácii a spomalenie prípadného rastu dát pri následných operáciách `Join` a `LeftJoin`.

Ak je logická podmienka zadaného filtru v konjunktívnej forme, potom je filter rozdelený podľa operácie `AND` na sadu menších filtrov. Rozdelením sa dosiahne zmenšenie domény filtru a je pravdepodobnejšie jeho nižšie umiestnenie v strome. Filtre však nie je možné propagovať ľubovoľne. Prítomnosť neviazaných hodnôt limituje možnosti propagácie operácie filter. Problém si môžeme ilustrovať na následujúcom príklade.

Príklad 5 *Majme požiadavku:*

```
SELECT DISTINCT *
WHERE { A . FILTER(bound(?y)) . {B} }
```

kde:

- `A, B` *sú skupiny operácií, ktorých výsledky sú nasledujúce:*
  `A={{?x=1}, {?x=2, ?y=2}}` *a*
  `B={{?x=1,?y=1}, {?x=2}}`*.*
- `FILTER(bound(?y))` *je filter nad premennou* `?y`*, ktorá môže byť neviazaná.*

Potom je výsledkom množina `{{?x=1, ?y=1}, {?x=2, ?y=2}}`*, ale propagovaním filtru do vnorenej skupiny sa zmení výsledok na prázdny. Je to dôsledkom vlastnosti neviazaných premenných zmeniť svoju hodnotu z neviazanej, na viazanú.*

Definujme si *bezpečné* a *nebezpečné* premenné a podmienky propagácie operácie *Filter*.

Definícia 12 *Bezpečná premenná je premenná, ktorá je pre všetky možné n-tice viazaná. Nebezpečná premenná je premenná, pre ktorú môže existovať n-tica, kde je neviazaná.*

Pokiaľ obsahuje `Filter` nebezpečné premenné, potom je v strome operácií zaradený za poslednú operáciu reprezentujúcu *group graph pattern* v ktorom sa nachádza. Teda s *Filtrom* obsahujúcim nebezpečné premenné sa nemanipuluje a je spracovaý podľa definície [15] po vykonaní všetkých operácii z daného *group graph patternu*.

Ak neobsahuje *Filter* nebezpečné premenné a nepatrí do *group graph patternu* `optional` vetvy *LeftJoin*, potom v strome operácií:

- môže byť presunutý pred nasledujúcu operáciu (bližšie ku koreňu),
- môže byť presunutý pred predchádzajúcu operáciu (bližšie k listom), pokiaľ sa nejedná o `optional` vetvu *LeftJoin* operácie a sú dostupné všetky zodpovedajúce premenné.

Filtre z `optional` vetvy operácie *LeftJoin* sú vylúčené, pretože podľa definície jazyka SPARQL definujú podmienku operácie *LeftJoin*. Naopak ani do tejto `optional` vetvy nie sú zaradené ďalšie, pretože táto vetva je nepovinná a má za úlohu rozšíriť výsledok iba pokiaľ je to možné. Jej zmenšením by nedošlo k odstráneniu výsledných n-tic, ktoré sú kompatibilné s odfiltrovanými n-ticami, ale dôjde k ich zmene, čo je proti požiadavke ekvivalencie.

Aby nedochádzalo k spracovávaniu viacerých takto propagovaných operácií *Filter* za sebou, sú takéto skupinky nakoniec spojené do jednej operácie *Filter* za použitia logickej operácie `AND`.

Propagácia operácií *Distinct* a *Reduced*

Ak má požiadavka definovanú vlastnosť `DISTINCT`, alebo `REDUCED`, znamená to, že vo výsledku neočakáva duplicitné riadky, respektíve ich počet môže byť zredukovaný. Úlohou tejto optimalizácie je spropagovať operácie redukujúce duplicity tak, aby došlo k zmenšeniu medzivýsledkov operácií. Zároveň sa berie ohľad na to, aby ich cena neprevýšila cenu ušetrenú.

Pri štandardnom spôsobe spracovania, sa operácie redukujúce duplicity vykonávajú po finálnej projekcii dát. Ak výsledok obsahuje veľké množstvo dát, je táto operácia pomalá. Prepisovacie pravidlá sa snažia túto operáciu vložiť taktiež na miesta nižšie v strome operácií.

Redukujúca operácia je pridaná pod operáciu *Join*, pokiaľ súčet veľkostí jej vstupov neprevyšuje očakávanú veľkosť výstupu. Pokiaľ dochádza k priebežnej projekcii premenných výslednej množiny, nie je možné nadradenú operáciu *Distinct*, alebo *Reduced* odstrániť, pretože hrozí vzik nových duplicít (Napríklad výsledná možina bez duplicít {{`?x=1, ?y=1`},{`?x=1, ?y=2`}, bude po aplikacií projekcie `?x` obsahovť duplicitné riadky {{`?x=1`},{`?x=1`}}.

Odstraňovanie duplicít sa taktiež pridáva napríklad k operácií *OrderBy*, ktorá je pridaná na usporiadanie vstupu operátora *Sort merge Join*, pretože na kompletne usporiadaných dátach je odstraňovanie triviálne a lacné.

Priebežná projekcia premenných

Postupným spracovaním požiadavky sa môžu hromadiť premenné, ktoré nebudú vstupovať do operácií v ďalšom spracovaní, ani sa nebudú vyskytovať vo výsledku. Predávanie ich dát v medzivýsledkoch znamená zvýšené pamäťové požiadavky a neprináša žiadny úžitok. Priebežná projekcia zabezpečuje, že takéto premenné sa nebudú propagovať za poslednú operáciu, v ktorej sa aktívne zúčastňujú.

V prípade, ak je definovaná finálna projekcia všetkých premenných (`*` v požiadavkách typu `SELECT` a `DESCRIBE`), priebežná projekcia odstraňuje iba premenné reprezentujúce *anonymné vrcholy* (*blank nodes*), ktoré sú automaticky generované pri prepise skrátených foriem zápisu požiadavky do ich neskrátenej formy.

5.1.4 Ďalšie zvažované metódy

Počas tvorby tejto diplomovej práce, sme zvažovali rôzne postupy a možnosti optimalizácie. Niektoré z nich boli zahrnuté do výslednej implementácie a niektoré boli nakoniec z rôznych dôvodov zavrhnuté. Následuje popis niekoľkých myšlienok a postupov, ktoré považujeme za vhodné spomenúť.

Znovupoužitie grafových vzorov

Jednou z možností optimalizácie, ktorá je navrhovaná v niekoľkých testoch SP^{2}Bench je znovupoužitie grafových vzorov. Túto metódu sme zamietli ako problematickú, vzhľadom k možnosti zablokovania behu výpčtu v Boboxe, spôsobeného jeho vlastnosťou na vyrovnávanie rýchlosti výpočtu jednotlivých vetiev.

Princíp metódy spočíva v nájdení disjunktných skupín grafových vzorov, ktoré až na odlišné pomenovanie premenných, produkujú ekvivalentné výsledné množiny n-tic. Pre tieto skupiny sa zvolí jedna, ktorá ich reprezentuje a mapovania premenných na pôvodné názvy. V systéme Bobox by to znamenalo, že krabička reprezentujúca vrchol tejto zvolenej skupiny, bude posielať vhodne premenované dáta na všetky vstupy týchto duplicitných grafových vzorov.

Zásadným problémom sa ukázalo vyrovnávanie rýchlosti výpočtu Boboxom v jednotlivých vetvách. Predstavme si že, krabička predstavujúca zlúčený grafový vzor, dodáva dáta do dvoch vetiev výpočtu. Je pravdepodobné, že rýchlejšia vetva (napr. pretože je kratšia) zaplní v strome operácií vyrovnávacie pamäte po spoločný vrchol s druhou - pomalou vetvou a zablokuje sa jej výpočet. Týmto sa však zablokuje aj zlúčená krabička, ktorá je súčasťou oboch vetiev. Zastavením prísunu dát do pomalšej vetvy, sa nikdy neuspokojí krabička spájajúca obe vetvy a dochádza k zaseknutiu výpočtu. Tento jav závisí od aktuálneho stavu systému v čase výpočtu. Možným riešením by bolo použitie medzikrabičiek, predstavujúcich veľké buffre, avšak toto riešenie by bolo potenciálne veľmi pamäťovo náročné.

Algebraické prepisovacie pravidlá

V prípade tradičných relačných jazykov, je skupina optimalizácií založená na aplikovaní relačných prepisovacích pravidiel. Avšak neštandardná algebra (2.2.3) jazyka SPARQL tvorí prispôsobenie prepisovacích pravidiel relačnej algebry netriviálne [17].

MergeJoined Operátor

Jedným zo zvažovaných operátorov pre systém Bobox bol aj *MergeJoined Operátor*. Operátor je založený na hromadnom spracovaní skupiny grafových vzorov *trojíc* spojených operáciou Join. Výhodou by bolo lepšie využitie dostupných indexov. Operátor bol zamietnutý z dôvodu požiadavky na funkčne minimalistické operátory, vzhľadom k svojej komplikovanosti a duplikovaní operácií *Scan* a *Join*.

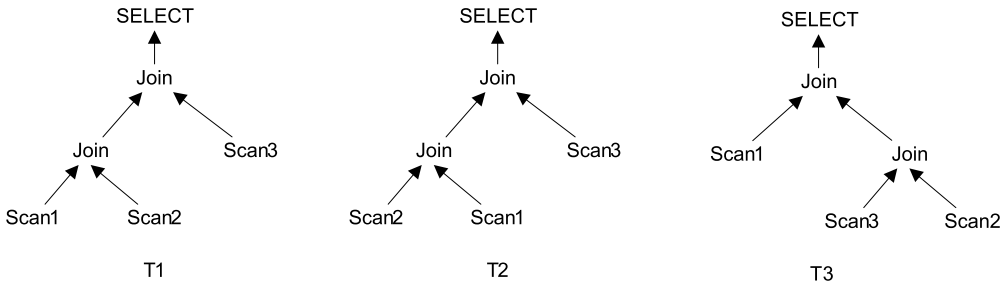
5.2 Generovanie plánu spracovania

Najdôležitejšou súčasťou prekladača je generovanie stromu fyzických operácií reprezentujúceho požiadavku. Algebraická forma požiadavky môže byť transformovaná do mnoho iných, logicky ekvivalentných algebraických reprezentácií. Každej takejto reprezentácii odpovedá viacero stromov fyzických operácií (napríklad *Join* môže byť reprezentovaný viacerými fyzickými operáciami ako nested-loops, merge join atď.). Strom fyzických operácií nazveme *plán*. Na výslednom pláne závisí pamäťová náročnosť a rýchlosť vykonania požiadavky. Vykonanie najlepšieho plánu môže trvať zlomok sekundy, oproti tomu najhoršie plány môžu trvať napríklad desiatky hodín. Úloha vyhnúť sa výberu nevhodných plánov a nájsť optimálny vzhľadom na použité heuristiky leží na pleciach *optimalizátora*.

Optimalizátor prehľadáva množinu všetkých ekvivalentných usporiadaní operácií. Jednotlivé plány sú následne ohodnotené za pomoci ohodnocovacích metód a je vybraný plán s najmenšou cenou. Výsledný strom usporiadania operácií je reprezentovaný pomocou nášho SQGM modelu.

5.2.1 Search space

Ako už bolo naznačené v úvode kapitoly, každá požiadavka má mnoho ekvivalentných stromov usporiadaní operácií. Ich množina tvorí *search space*. Obrázok 5.1 ukazuje možné reprezentácie požiadavky.



Obr. 5.1: Príklady ekvivalentných stromov operácií jednej požiadavky.

Pre zložitejšiu požiadavku môže byť počet stromov, ktoré ju reprezentujú enormný. Pretože stratégie hľadania optimálneho plánu pracujú nad touto množinou možných stromov, ktorú prechádzajú, stáva sa jej veľkosť problémom. Na zmiernenie tohoto problému sa definujú rôzne podmienky - heuristiky, ktoré túto množinu redukujú. Pritom sa však snažia zachovať optimálné riešenie, alebo aspoň riešenia, ktoré sa mu približujú.

Podmienka 1 *Projekcie sa vykonávajú priebežne a negenerujú osobitné operácie. Vykonávajú sa priamo na výsledkoch operácií.*

Pokiaľ operátor podporuje definovanie obmedzení na premenné, potom sa operácia Filter vykonáva priebežne ako súčasť tejto operácie a negeneruje sa ako samostatná operácia.

Tieto podmienky slúžia na elimináciu tých operácií v plánoch, ktoré môžu byť efektívne vykonané v rámci iných naväzujúcich operácií. Neredukujú zásadne množinu všetkých stromov operácii, ale definujú jej pravidlá, ktoré znižujú jej výslednú cenu.

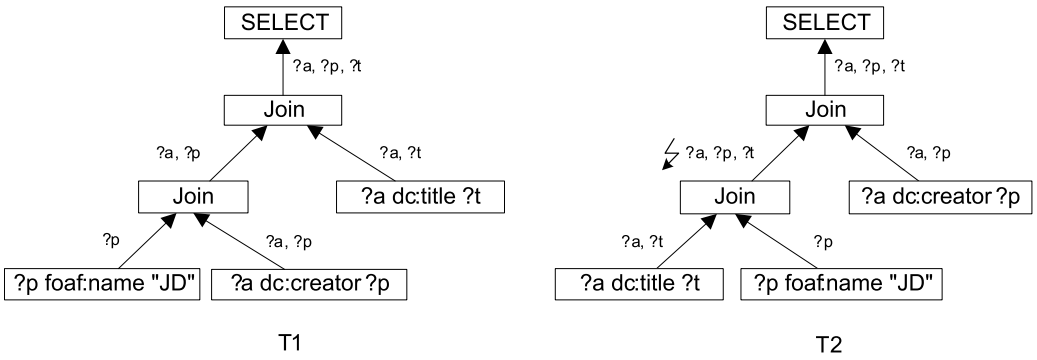
Majme množinu relácií tvoriacich požiadavku, potom množina všetkých stromov operácií je tvorená využitím algebraických vlastností operácie *Join*: *komutativita* ($R_1 \bowtie R_2 \equiv R_2 \bowtie R_1$) a *asociativita* ($(R_1 \bowtie R_2) \bowtie R_3 \equiv R_1 \bowtie (R_2 \bowtie R_3)$). Veľkosť množiny všetkých usporiadaní operácií *Join*, generovanej spomenutými vlastnosťami je pre N relácií $\Omega(N!)$. Táto množina je teda veľmi veľká, a preto sú definované obmedzenia na jej zmenšenie. Nasledujúci text je zameraný na stromy operácií *Join*, definujúce ich usporiadanie. Pretože SQGM strom obsahuje aj iné operácie, takýto strom si môžeme predstaviť ako niekoľko stromov operácií *Join* spojených pomocou ostatných operácií.

Podmienka 2 *Operácie Join nikdy nevytvárajú kartézsky súčin dát, pokiaľ si to priamo nevyžaduje požiadavka.*

Uvažujme príklad :

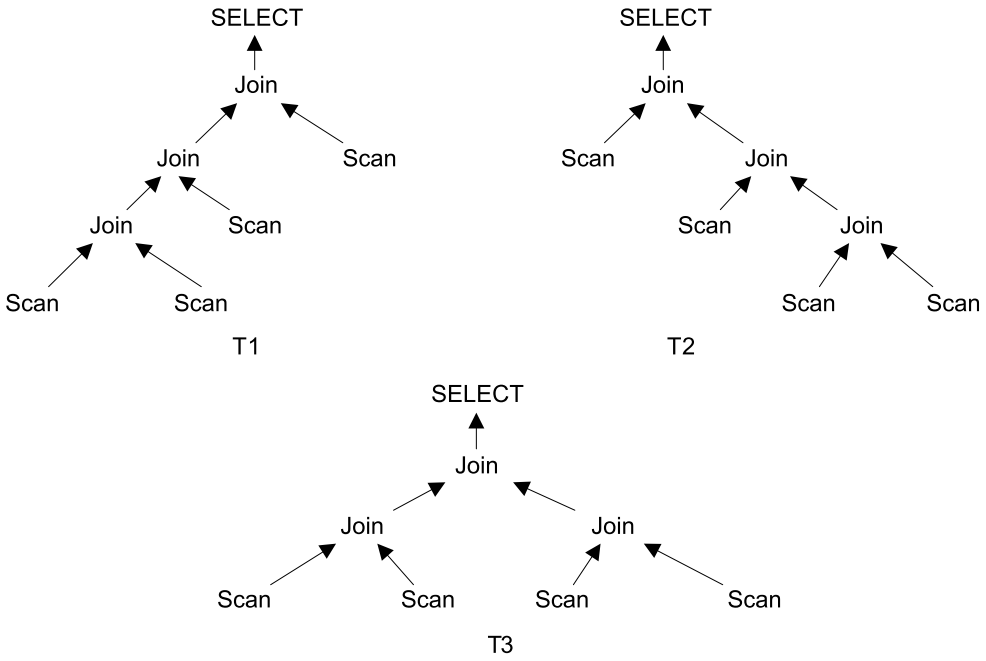
```
PREFIX foaf:    <http://xmlns.com/foaf/0.1/>
PREFIX dc:      <http://purl.org/dc/elements/1.1/>

SELECT *
WHERE {
      ?p foaf:name "JD".        #P1
      ?a dc:creator ?p.         #P2
      ?a dc:title ?t            #P3
}
```

Obrázok 5.2 zobrazuje možné usporiadania *trojíc P1*, *P2*, *P3*, ktoré zodpovedajú požiadavke. Strom T1 spĺňa Podmienku 2, pretože každá operácia *Join* má k dispozícii spoločné premenné pre obe vstupné množiny. To neplatí pre strom T2, ktorý vo vnorenej operácii Join nad *P1* a *P2* vytvára množinu obsahujúcu kartézsky súčin vstupných množín. Táto potom vstupuje do ďalšej operácie *Join*, ktorá je týmto výrazne pomalšia. Podmienka 2 má teda za úlohu eleminovať neoptimálne stromy, znevýhodnené generovaním zbytočne veľkých medzivýsledkov.



Obr. 5.2: Ukážka usporiadania stromu operácií, vzhľadom na generovanie kartézskeho súčinu operáciou *Join*.

Podmienka 3 *Vnútorný operand (inner operand) každej operácie Join je vždy databázová relácia a nie medzivýsledok.*



Obr. 5.3: Ukážka *left-deep* (T1), *right-deep* (T2) a *bushy* (T3) stromov.

Obrázok 5.3 ukazuje tri základné tvary stromov usporiadania operácií *Join*. Strom T1 spĺňa Podmienku 3 a nazýva sa *left-deep tree*, T2 sa nazýva *right-deep tree)* a T3 sa označuje ako *rozvetvený (bushy tree)*. Stromy T2 a T3 nespĺňajú Podmienku 3, pretože obsahujú operáciu *Join* ako vnútorný operand.

Podmienka 3 je výrazne heuristickejšia a môže v niektorých prípadoch vylúčiť aj optimálne usporiadanie. Avšak ukazuje sa, že najlepšie usporiadanie pomocou *left-deep* stromu sa blíži usporiadaniu optimálnemu [8]. Usporiadanie taktiež umožňuje priebežné spracovanie pomocou *pipeline*, čo zvýhodňuje cenu usporiadania. Hlavným dôvodom pre toto usporiadanie je jeho redukcia množiny všetkých možných stromov operácie join na $O(2^N)$ pre N relácií.

5.2.2 Optimalizátor

Fyzická optimalizácia požiadavky je reprezentovaná modulom optimalizátor. Použitím dynamického programovania nad priestorom plánov (*plan space*) buduje „optimálný“ strom operácií smerom od listov ku koreňu. Výsledný strom nemusí byť vďaka použitiu heuristík na zmenšenie prehľadávaného priestoru optimálny, ale mal by sa tomuto plánu približovať. Optimalita plánov sa posudzuje podľa ich ceny, ktorá je im priradená ohodnocovacím modelom.

Ohodnocovací model priradzuje cenu kompletným aj čiastočným plánom a taktiež odhaduje očakávanú veľkosť dát, ktoré budú vstupovať do jednotlivých operácií. Tieto odhady ovplyvňuje niekoľko faktorov:

- Množina štatistík nad jednotlivými stĺpcami databázy (*subject, predicate, object*) a vzťahmi medzi nimi.
- Funkcie odhadujúce selektivitu predikátov a odhadujúce veľkosť výstupných dát jednotlivých operácií.
- Funkcie určujúce cenu jednotlivých operácií uvažovaním ceny CPU, I/O a komunikácie.

Od optimalizátora sa očakáva, aby jeho *search space* mal malú cenu, dokázal presne určovať cenu plánov a dokázal efektívne vyhľadať optimálny plán.

Podstata prístupu dynamického programovania je založená na predpoklade, že model ohodnocovania spĺňuje princíp optimality. Ten predpokladá, že pokiaľ máme získať optimálny plán požiadavky q, pozostávajúcej z n operácií *Join*, potom postačuje nájsť optimálne usporiadanie pre podpožiadavky q' požiadavky q o veľkosti n-1 operácií *Join* a toto rozšíriť o zvyšnú operáciu *Join*. Z toho vyplýva, že čiastočné plány, ktoré nie sú optimálne, sa nezúčastňujú v ďalšom výpočte optimálneho plánu.

Dôležitou vlastnosťou optimalizátora je to, že berie do úvahy aj usporiadania (*interesting orders*) medzivýsledkov. Čiastočné plány vedúce k uspo-

riadaným dátam môžu byť lokálne nevýhodné, ale nakoniec môžu viesť k plánom s menšou cenou. Predstavme si požiadavku, ktorá predstavuje spojenie troch *trojíc* P_1, P_2, P_3 pomocou operácie *Join*. Nech všetky trojice obsahujú spoločnú premennú `?s`, ktorá je zotriedená po P_3. Ďalej nech sa najprv vykoná operácia $Join(P_1, P_2)$ s ohodnoteniami x, resp. y pre fyzické operácie *nested-loops*, resp. *sort merge* a platí $x < y$. Potom plány nad $\{P_1, P_2\}$ s ohodnotením x sa nikdy nepoužijú, aj keď ich výsledok bude zotriedený na `?s`, čo môže viesť k výraznemu zefektívneniu spojenia s P_3. Odmietnutím týchto čiastočných plánov môže dôjsť k výberu neoptimálneho globálneho plánu. Z tohto dôvodu sa pre každé usporiadanie medzivýsledkov sa osobitne posudzuje jeho optimalita.

Ako zdroj dát slúži SQGPM model. Keďže algoritmus pracuje na princípe usporiadávania operácií *Join*, aplikuje sa na množinu grafových vzorov spracovávaného vrcholu SQGPM stromu. Ak nejaký grafový vzor patriaci aktuálne spracovanému vrcholu viaže ďalšie vrcholy, algoritmus je na ne rekurzívne aplikovaný. Hľadanie "optimálneho"[1] stromu operátorov, reprezentovaného pomocou SQGM prebieha v nasledujúcich krokoch:

Krok 1 Pre všetky trojice *Basic Graph Pattern* sa vytvoria operátory pre všetky možné prístupy k dátam (napríklad použitím metód *full table scan, index scan, hash table scan*) tvoriace *prístupové plány*. Každý operátor sa zaradí do priehradky, podľa ním generovaného usporiadania a reprezentujú čiastočné plány. Tieto čiastočné plány sa ohodnotia použitím vhodných ohodnocovacích funkcií. Pre operátory negenerujúce žiadne usporiadanie je vytvorená osobitná priehradka. Pre ostatné grafové vzory sa rekurzívne aplikuje tento algoritmus. Každý plán z výsledku rekurzie sa zaradí do zodpovedajúcej priehradky aktuálnej skupiny a je taktiež považovaný za *prístupový plán*. V každej priehradke sa nájde plán s najmenšou cenou, ktorý bude reprezentovať dané usporiadanie. Najlacnejší plán z priehradky plánov bez usporiadania sa vezme do úvahy, len ak je jeho cena menšia než ceny vo všetkých priehradkách usporiadaní.

Krok 2 Pre každú dvojicu relácií požiadavky spojenú operáciou *Join* vytvor čiastočné plány použitím odpovedajúcich *prístupových plánov* a aplikovaním dostupných operátorov operácie *Join* ako napríklad *nested loops*, alebo *merge sort*. Tieto nové plány sú roztriedené ako v predchadzajúcom kroku.

[1] Vzhľadom k použitiu heuristík 5.2.1 na zmenšenie prehľadavaného priestoru, nemusí byť nájdený plán optimálny, ale očakáva sa, že sa mu bude približovať.

Krok i Pre každý čiastočný plán tvorený i-1 reláciami z predchádzajúceho kroku sa vytvoria nové čiastočné plány jeho spojením s *čiastočnými prístupovými plánmi* nepoužitých relácií a aplikovaním dostupných operátorov operácie *Join* ako napríklad *nested loops*,alebo *merge sort*. Tieto nové plány sú roztriedené ako v predchadzajúcom kroku.

Krok n Posledný krok vygeneroval zoznam plánov odpovedajúcich požiadavke (unikátnu sadu N spojení relácií). Najlacnejší plán predstavuje hľadaný plán. V prípade, ak sa jedná o vnorený podstrom, sa ako výsledok použijú najlacnejšie plány z predchádzajúceho kroku. Týmto je ponechaná možnosť v nadradenej skupine vzorov zvážiť taktiež možné usporiadania generované týmto podstromom.

5.3 Súhrnné štatistiky

Uchovávanie sumárnych dát pre štatistiky je bežne používaná metóda v databázových systémoch. Slúži napríklad k odhadom selektivity, alebo veľkosti dát vstupujúcich do operácií použitých k ohodnocovaniu plánov výpočtu. Vhodný výber a podrobnosť štatistík môže výrazne ovplyvniť výber správneho plánu výpočtu požiadavky. Naproti tomu, s kvalitou štatistík, rastú aj požiadavky na ich spracovanie, uloženie a dostupnosť.

V tejto práci sme sa snažili zvoliť takú skupinu súhrnných štatistík [19], ktoré pomôžu zlepšiť kvalitu odhadov, avšak nebudú znamenať prehnané nároky na ich údržbu a používanie.

5.3.1 Štatistiky pre *triple pattern*

Každá časť trojice *subjekt*, *predikát* a *objekt* - je v požiadavke tvorená hodnotou, alebo premennou. Časti s definovanou hodnotou nazveme *viazané* a časti s premennou nazveme *neviazané*. Táto trojica definuje *mapovanie*. *Neviazané* časti nemajú vplyv na štatistiky. U *viazaných* nás však zaujíma každá časť, pretože každá z nich požaduje rozdielné štatistiky. Štatistiky sú zamerané na odhad *kardinality mapovania*, čiže počtu *RDF trojíc*, ktoré zodpovedajú mapovaniu.

Pri *viazanom subjekte* je *kardinalita mapovania* aproximovaná priemernou hodnotou *RDF trojíc*, ktoré zodpovedajú jednému *subjektu*. K výpočtu sa teda použije celkový počet *RDF trojíc* a počet unikátnych *subjektov*.

Veľkosť pre *viazaný predikát* je počet *RDF trojíc*, ktoré zodpovedajú tomuto predikátu. Vytvára sa presná štatistika pre každú unikátnu hodnotu *predikátu* ako počet *RDF trojíc* mapujúcich sa na daný *predikát*. Presná

štatistika je zvolená pre malý očakávaný počet unikátnych hodnôt *predikátu* a jeho častý viazaný výskyt.

Aproximácia veľkosti *viazaného objektu* je vykonávaná za pomoci *equal-height histogramu* [7]. Použitie histogramu vyžaduje očakávaná rôznorodosť dát *objektu*. *Equal-height* varianta bola zvolená, pretože predpokladáme jej lepšiu adaptabilitu pri extrémoch tvorených napríklad objektmi viazanými na *predikát* `rdf:type`. Obdobné *equal-height* histogramy sú generované aj pre viazanú dvojicu *predikát-objekt*.

5.3.2 Štatistiky pre Join

Pre operácie *Join* sú dostupné obecné informácie o selektivite vzájomných kombinácií jednotlivých části *RDF trojice*: *subjekt*, *predikát* a *objekt*. Selektivita je vypočítaná z dát pomocou predpisu

$$sel(c_1, c_2) = \frac{J_{c_1 c_2}}{T^2}$$

kde $c_1, c_2 \in \{subjekt, predikt, objekt\}$, $sel(c_1, c_2)$ označuje selektivitu pre danú väzbu, $J_{c_1 c_2}$ označuje veľkosť výsledku operácie *Join* nad množinou všetkých *RDF trojíc* v dátovej sade s väzbou c_1, c_2 a T^2 je druhá mocnina veľkosti počtu všetkých *RDF trojíc* v datovej sade.

Ďalšia skupina presnejších sumárnych štatistík za zameriava na väzby medzi jednotlivými *predikátmi*. Obsahuje selektivity medzi *subjektmi a objektmi* pre jednotlivé kombinácie *predikátov*. Tieto selektivity sú definované ako

$$sel_{p_1 p_2}(c_1, c_2) = \frac{J^{p_1 p_2}_{c_1 c_2}}{T_{p_1} \times T_{p_2}}$$

kde $c_1, c_2 \in \{subjekt, objekt\}$. T_{p_1}, resp. T_{p_2} označujú množinou *RDF trojíc* s *predikátom* c_1, resp. c_1. $sel_{p_1 p_2}(c_1, c_2)$ označuje selektivitu väzby $c_1 - c_2$ medzi množinami T_{p_1} a T_{p_2}. $J^{p_1 p_2}_{c_1 c_2}$ označuje počet výsledkov operácie Join nad množinami T_{p_1} a T_{p_2}.

5.4 Odhad selektivity

Odhad selektivity je úlohou *štatistického modulu*. Ten je založený nad súhrnnými štatistikami z kapitoly 5.3. Ponúka rozhranie na zistenie odhadu selektivity a veľkosti dát pre vzory *triple pattern* a ich spojenie pomocou operácie *Join* .

Selektivita nadobúda hodnoty z intervalu [0, 1] a vyjadruje úspešnosť výberu zo vstupných dát. Grafový vzor je selektívny, ak sa jeho hodnota selektivity približuje hornej hranici intervalu (t.j. 1). Príkladom takéhoto

vzoru môže byť napríklad trojica `?s ?p ?o`. Tieto grafové vzory majú veľkú pravdepodobnosť zodpovedať trojiciam v zadanej *RDF dátovej sade*. Opakom je neselektívny grafový vzor, ktorého hodnota sa približuje spodnej hranici intervalu (t.j. 0). Príkladom môže byť trojica `ex:s ex:p ex:o`, ktorá má malú pravdepodobnosť, že bude zodpovedať nejakým trojiciam v *RDF dátovej sade*.

5.4.1 Odhad selektivity vzoru *triple pattern*

Selektivita vzoru *triple pattern* je definovaná ako

$$sel(t) = sel(s) \times sel(p) \times sel(o)$$

kde $sel(t)$ predstavuje selektivitu *triple patternu* t, $sel(s)$ predstavuje selektivitu *subjektu*, $sel(p)$ predstavuje selektivitu *predikátu* a $sel(o)$ predstavuje selektivitu *objektu*. Tento vzorec iba aproximuje selektivitu $sel(t)$, pretože predpokladá, že $sel(s)$, $sel(p)$ a $sel(o)$ sú nezávislé, avšak zvyčajne tento predpoklad neplatí. Selektivita neviazaných časti vzoru *triple pattern* túto hodnotu neovplyvňuje, pretože ich selektivita je rovná hodnote 1. Premenné vyberajú každý záznam z *RDF dát*

Selektivita viazaného *subjektu*, je odhadnutá ako $sel(s) = \frac{\frac{T}{R}}{T} = \frac{1}{T}$, kde R je počet unikátných subjektov v *dátach* a T je počet všetkých *RDF trojíc*.

Selektivita viazaného *predikátu*, je odhadnutá ako $sel(p) = \frac{T_p}{T}$, kde T označuje počet všetkých *RDF trojíc* a T_p je počet všetkých *RDF trojíc* ktoré majú hodnotu predikátu p. Oboje hodnoty T_P, aj T sú dostupné v sumárnych štatistikách.

Selektivita viazaného *objektu*, je odhadnutá ako

$$sel(o) = \begin{cases} sel(p,o) & \text{ak je p viazaná} \\ sel_H(o) & \text{inak} \end{cases}$$

kde dvojica (p, o) predstavuje interval histogramu pre predikát p, do ktorého patrí hodnota o. $sel(p, o)$ vyjadruje frekvenciu intervalu (p, o) normalizovanú počtom trojíc predikátu p. $sel_H(0)$ vyjadruje frekvenciu intervalu histogramu nad *objektami*, do ktorého patrí hodnota o, normalizovanú počtom všetkých *RDF trojíc*.

5.4.2 Odhad selektivity operácie *Join*

Selektivita operácie *Join* je definovaná ako

$$sel(j) = \prod_i sel(v_i)$$

kde $sel(v_i)$ značí selektivitu i-tej premennej, zúčastňujúcej sa operáce *Join*. Táto selektivita je určená použitím sumárnych štatistík definovaných v časti 5.3.2. V prípade, ak je daná premenná v oboch vstupných množinách špecifikovaná aspoň jedným *predikátom*, potom sú použité špecifickejšie štatistiky. Pokiaľ je premenná definována viac ako jedným predikátom na niektorom vstupe, odhad sa chová optimisticky a vyberá sa maximálna selektivita spomedzi selektivít kombinácií predikátov z oboch vstupov. Obecnejšie štatistiky sa používajú v prípade, ak premenné nie sú upresnené žiadnym predikátom.

5.5 Odhad ceny plánu

Cena plánu slúži k vyjadreniu očakávanej náročnosti vykonania plánu. Závisí od použitej fyzickej operácie a množstva dát, ktoré musí spracovť a prípadne aj od typu spracovávaných dát (zotriedenosť, vytváranie kartézskych súčinou, unikátnosť dát, dostupnosť indexov). Cena pozostáva z dvoch zložiek, cena výpočtu odvodená od množstva vykonaných operácií a cena za komunikáciu medzi operáciami.

Vzhľadom k tomu, že počas tvorby tejto práce nie je dostupná konkrétna implementácia fyzických operácií a databázovej vrstvy, je cena operácií vyrátavaná pomocou obecných výpočtov podľa daného typu operátora. Napríklad pre operátor *Full Table Scan* je počet operácií stanovený ako počet *RDF trojíc* v databáze. Pre operátor *Index Scan* sa predpokladá použitie B+ stromov a cena zodpovedá cene nájdeniu prvého prvku pomocou indexu a očakávanej veľkosti načítaných záznamov.

Vzhľadom na dôležitosť presnejších odhadov, je potrebné výpočet ceny aktualizovať s dostupnosťou fyzických implementácií jednotlivých operácií a implementácie databázy.

Kapitola 6

Experimenty

Počas vývoja prekladača sme vykonávali rôzne experimenty na testovacích dátach a sade testovacích požiadaviek na overovanie riešenia . Ich zdrojom bol generátor testovacích dát a sada požiadaviek SP^{2}Bench[18]. Vzhľadom k nepripravenosti systému Bobox k nasadeniu prekladača, boli výkony testov posudzované iba na základe definovaných abstraktných cien plánov týchto požiadaviek a výsledná štruktúra operácií bola vizuálne hodnotená s ohľadom na implementované optimalizácie.

Sumárne štatistiky použité pri testovaní boli vygenerované nad množinou dát o veľkosti približne 50 000 *RDF trojíc*. Táto veľkosť už poskytuje rozumnú vzorku pre sumárne štatistiky a vzhľadom k reálnemu nevykonávaniu požiadavky by zväčšenie testovacej množiny nemalo viditeľný prínos.

6.1 O SP^{2}Bench

SP^{2}Bench[18] je voľne dostupný testovací framework, špecificky zameraný na jazyk SPQRQL a je situovaný v prostredí DBLP. Pozostáva z generátora dát na vytvorenie ľubovoľne veľkej množiny DBLP-špecifických dát vo formáte *RDF* a sady testovacích požiadaviek. Generované dokumenty reflektujú kľúčové charakteristiky a distribúcie vyskytujúce sa v originálnej DBLP sade dát pri zachovní určitej miery náhodnoti generovaných dát. Taktiež pokrýva rôzne RDF konštrukty, ako napríklad anonymné vrcholy (*blank nodes*), alebo kontainery. Testovacie požiadavky sú navrhnuté tak, aby na rozumných príkladoch testovali rôzne operátory, prístupy k dátam a rôzne optimalizačné stratégie.

6.2 Experimenty

Na následujúcej sade testovacích požiadaviek z SP^{2}Bench prebiehali experimenty. Niektoré z testov sú zamerané na špecifické optimalizácie, ktoré nie sú pokryté týmto optimalizátorom, alebo na dátové prístupy, slúžili aspoň k overeniu funkčnosti jednotlivých operátorov a aplikovatelných optimalizácií. Testy, ktoré obsahujú zaujímavé konštrukcie z pohľadu tejto aplikácie, sú podrobnejšie popísané. Výsledné modely sú ilustrované zvoleným SQGM grafom. Čísla v jednotlivých krabičkách ukazujú odhady veľkosti dát, vygenerovaných danou operáciou. Očakávané veľkosti výsledkov jednotlivých požiadaviek sú pre porovnanie uvedené v tabuľke 6.1.

T1	T2	T3a	T3b	T3c	T4	T5ab	T6	T7	T8	T9	T10	T11
1	965	3647	25	0	104746	1085	1769	2	264	4	307	10

Tabuľka 6.1: Očakávané veľkosti výsledkov jednotlivých testovacích požiadaviek.

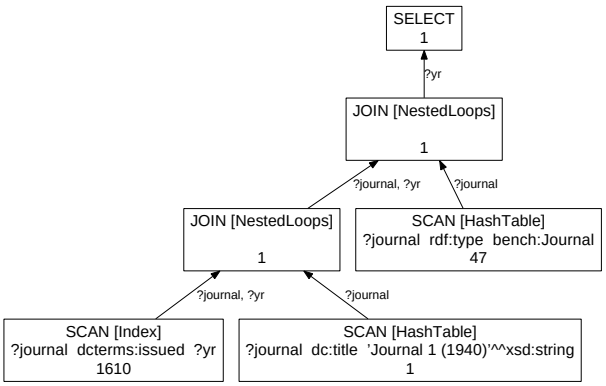
Pre prehľadnosť je v zadaní jednotlivých testovaných požiadaviek vynechaná deklarácia prefixov. Všetky použité prefixy, sú definované pomocou následujúcich predpisov:

```
PREFIX bench:    <http://localhost/vocabulary/bench/>
PREFIX dc:       <http://purl.org/dc/elements/1.1/>
PREFIX dcterms:  <http://purl.org/dc/terms/>
PREFIX foaf:     <http://xmlns.com/foaf/0.1/>
PREFIX person:   <http://localhost/persons/>
PREFIX rdf:      <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs:     <http://www.w3.org/2000/01/rdf-schema#>
PREFIX swrc:     <http://swrc.ontoware.org/ontology#>
PREFIX xsd:      <http://www.w3.org/2001/XMLSchema#>
```

6.2.1 Testovacie požiadavky

T1 *Vráti rok vydania publikácie* 'Journal 1 (1940)'.

```
SELECT ?yr
WHERE {
  ?journal rdf:type bench:Journal .
  ?journal dc:title "Journal 1 (1940)"^^xsd:string .
  ?journal dcterms:issued ?yr
}
```



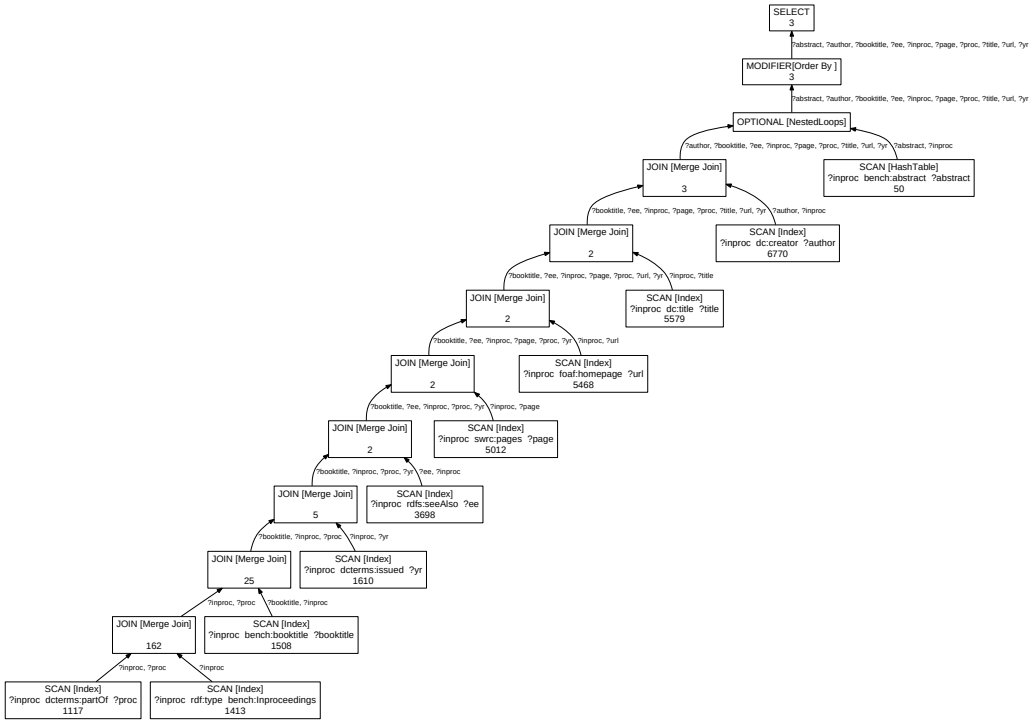
Obr. 6.1: Výsledný SQGM graf požiadavky T1.

Jedná sa o jednoduchú požiadavku, ktorá vracia presne jeden výsledok. Pre vyhľadanie názvu prekladač vybral indexovaný prístup, čo je efektívne hlavne pri prístupu k veľkým dátam.

T2 *Vyberie všetky príspevky s vlastnosťami* `dc:creator`, `bench:booktitle`, `dc:title`, `swrc:pages`, `dcterms:partOf`, `rdfs:seeAlso`, `foaf:homepage`, `dcterms:issued` *a voliteľne* `bench:abstract` *vrátane týchto vlastností.*

```
SELECT ?inproc ?author ?booktitle ?title
       ?proc ?ee ?page ?url ?yr ?abstract
WHERE {
  ?inproc rdf:type bench:Inproceedings .
  ?inproc dc:creator ?author .
  ?inproc bench:booktitle ?booktitle .
  ?inproc dc:title ?title .
  ?inproc dcterms:partOf ?proc .
  ?inproc rdfs:seeAlso ?ee .
  ?inproc swrc:pages ?page .
  ?inproc foaf:homepage ?url .
  ?inproc dcterms:issued ?yr
  OPTIONAL {
    ?inproc bench:abstract ?abstract
  }
}
ORDER BY ?yr
```

Táto požiadavka je zameraná na *bushy graph pattern*. Avšak v našom prípade sa prejavuje heuristika na obmedzenie prehľadávaného priestoru hľadania (*search space*), obmedzujúca tvar výsledného stromu na *left-deep* strom. Výrazná výška výsledného stromu taktiež akumuluje chyby odhadov, čo má v tomto prípade za následok podcenenie výsledného odhadu veľkosti dát.



Obr. 6.2: Výsledný SQGM graf požiadavky T2.

T3a *Vyberie všetky články s vlastnosťou* `swrc:pages`.

```
SELECT ?article
WHERE {
  ?article rdf:type bench:Article .
  ?article ?property ?value
  FILTER (?property=swrc:pages) }
```

T3b *Vyberie všetky články s vlastnosťou* `swrc:month`.

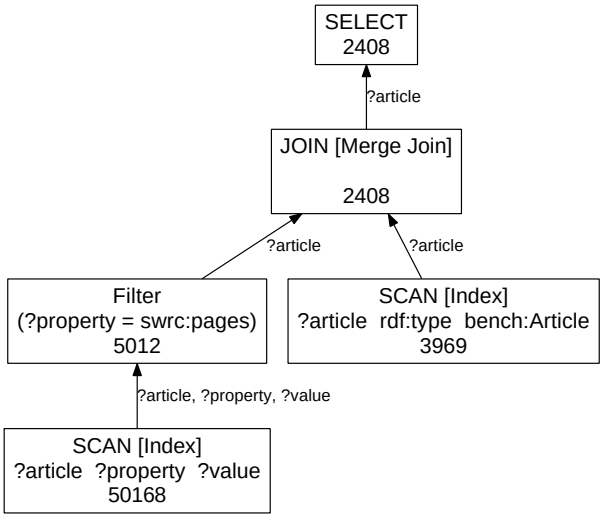
```
SELECT ?article
WHERE {
  ?article rdf:type bench:Article .
  ?article ?property ?value
  FILTER (?property=swrc:month) }
```

T3c *Vyberie všetky články s vlastnosťou* `swrc:isbn`.

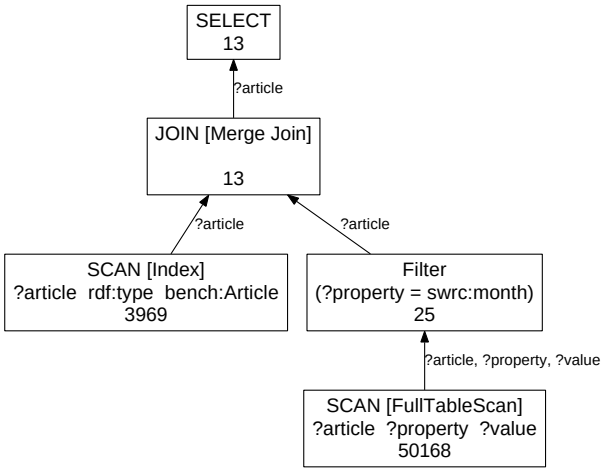
```
SELECT ?article
WHERE {
  ?article rdf:type bench:Article .
  ?article ?property ?value
  FILTER (?property=swrc:isbn) }
```

Trojica požiadaviek T3abc je zameraná na testovanie výrazov operácie *Filter* s rôznou selektivitou. Vzhľadom k charakteristike testovacích dát sa očakáva, že filter vráti zhruba 92.61% v prípade požiadavky T3a, 0.65% v prípade T3b a 0%v prípade požiadavky T3c, pretože články nikdy nemajú definovaný predikát `swrc:isbn`.

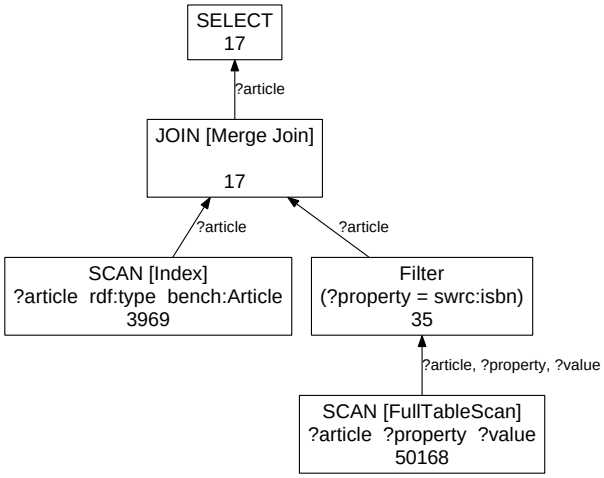
Vzhľadom k tom, že náš optimalizátor nepracuje so schémou dát, neodhalí v prípade T3c nulovú selektivitu, ale použije obecnú selektivitu daného predikátu. V prípadoch T3a a T3b je táto selektivita dostačujúca.



Obr. 6.3: Výsledný SQGM graf požiadavky T3a.



Obr. 6.4: Výsledný SQGM graf požiadavky T3b.



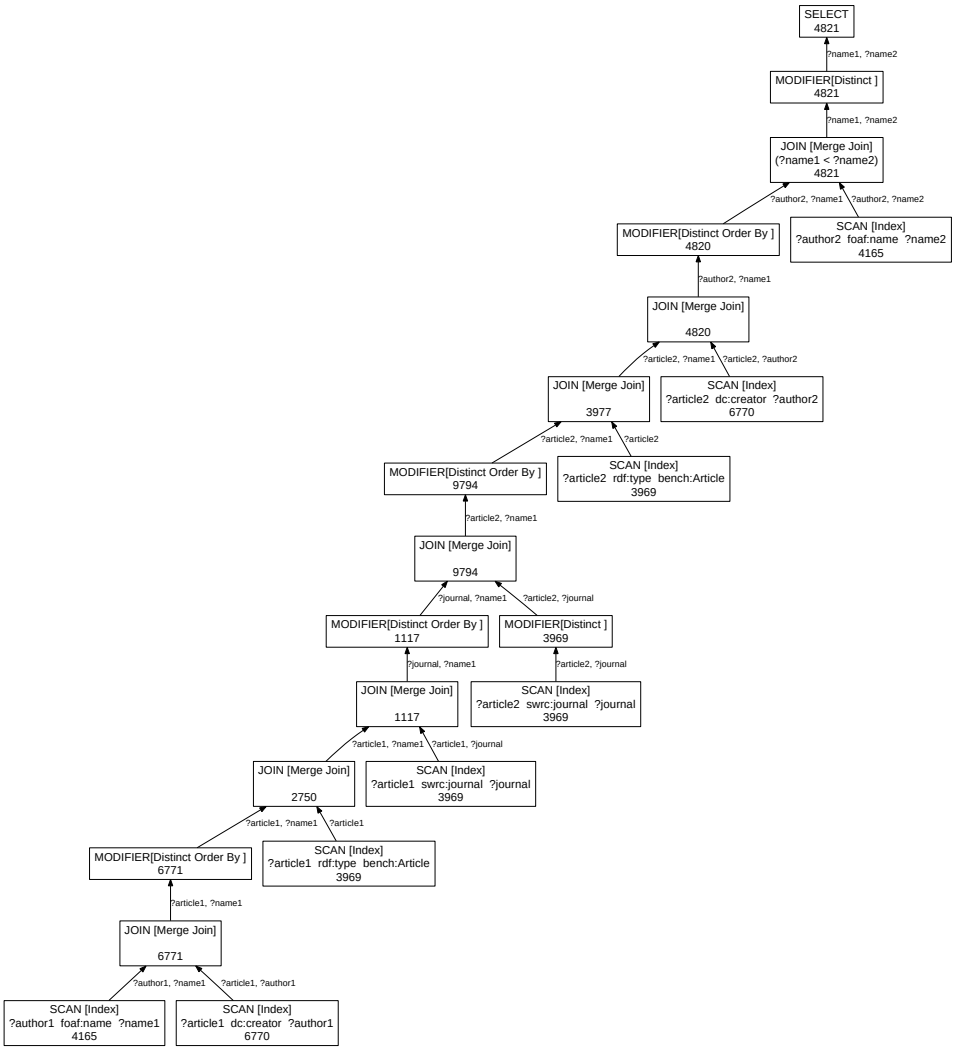
Obr. 6.5: Výsledný SQGM graf požiadavky T3c.

T4 *Vyberie všetky rôzne dvojice mien autorov, ktorý publikovali v rovnakom žurnále.*

```
SELECT DISTINCT ?name1 ?name2
WHERE {
  ?article1 rdf:type bench:Article .
  ?article2 rdf:type bench:Article .
  ?article1 dc:creator ?author1 .
  ?author1 foaf:name ?name1 .
  ?article2 dc:creator ?author2 .
  ?author2 foaf:name ?name2 .
  ?article1 swrc:journal ?journal .
  ?article2 swrc:journal ?journal
  FILTER (?name1<?name2)
}
```

Požiadavka T4 obsahuje dlhý grafový reťazec, ktorý produkuje veľké medzivýsledky a obmedzenie vo forme filtra na jeho konci. SP^{2}Bench očakáva zahrnutie tohto filtra do výpočtu reťazca. Avšak nami definované heuristiky, ktoré nepreferjú kartézske súčiny a definujúce tvar stromu na *left-deep*, vynútia riešenie toho obmedzenia až na konci reťazca. Odhady sú tu taktiež ovplyvnené ako v prípade T2 a nepresnosti sú zvýraznené výškou výsledného stromu.

Na tomto príklade je taktiež možné vidieť, ako sa prekladač snaží propagovať vlastnosť `DISTINCT` jej pripojením k operácií *OrderBy*, pretože je jej vykonanie na zotriedených dátach takmer zadarmo. Priebežné projekcie premenných pomáhajú znižovať pamäťovú náročnosť pri spracovaní medzivýsledkov.



Obr. 6.6: Výsledný SQGM graf požiadavky T4.

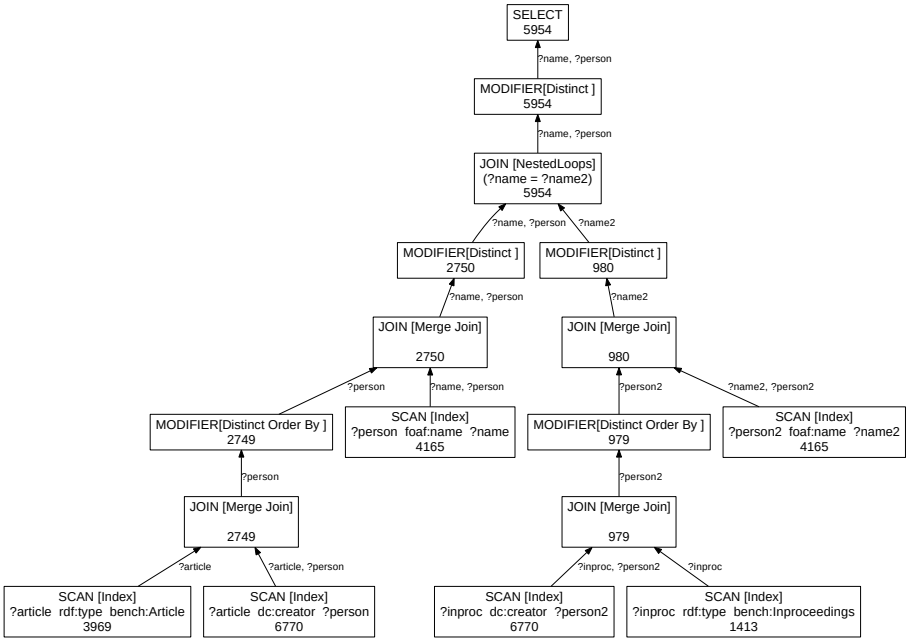
.

T5a *Vráti mena všetkých osôb, ktoré sa vyskytujú ako autori aspoň jedného príspevku a aspoň jedného článku.*

```
SELECT DISTINCT ?person ?name
WHERE {
  ?article rdf:type bench:Article .
  ?article dc:creator ?person .
  ?inproc rdf:type bench:Inproceedings .
  ?inproc dc:creator ?person2 .
  ?person foaf:name ?name .
  ?person2 foaf:name ?name2
  FILTER (?name=?name2)
}
```

T5b *Vráti mena všetkých osôb, ktoré sa vyskytujú ako autori aspoň jedného príspevku a aspoň jedného článku.* (Rovnako ako (T5a)).

```
SELECT DISTINCT ?person ?name
WHERE {
  ?article rdf:type bench:Article .
  ?article dc:creator ?person .
  ?inproc rdf:type bench:Inproceedings .
  ?inproc dc:creator ?person .
  ?person foaf:name ?name
}
```

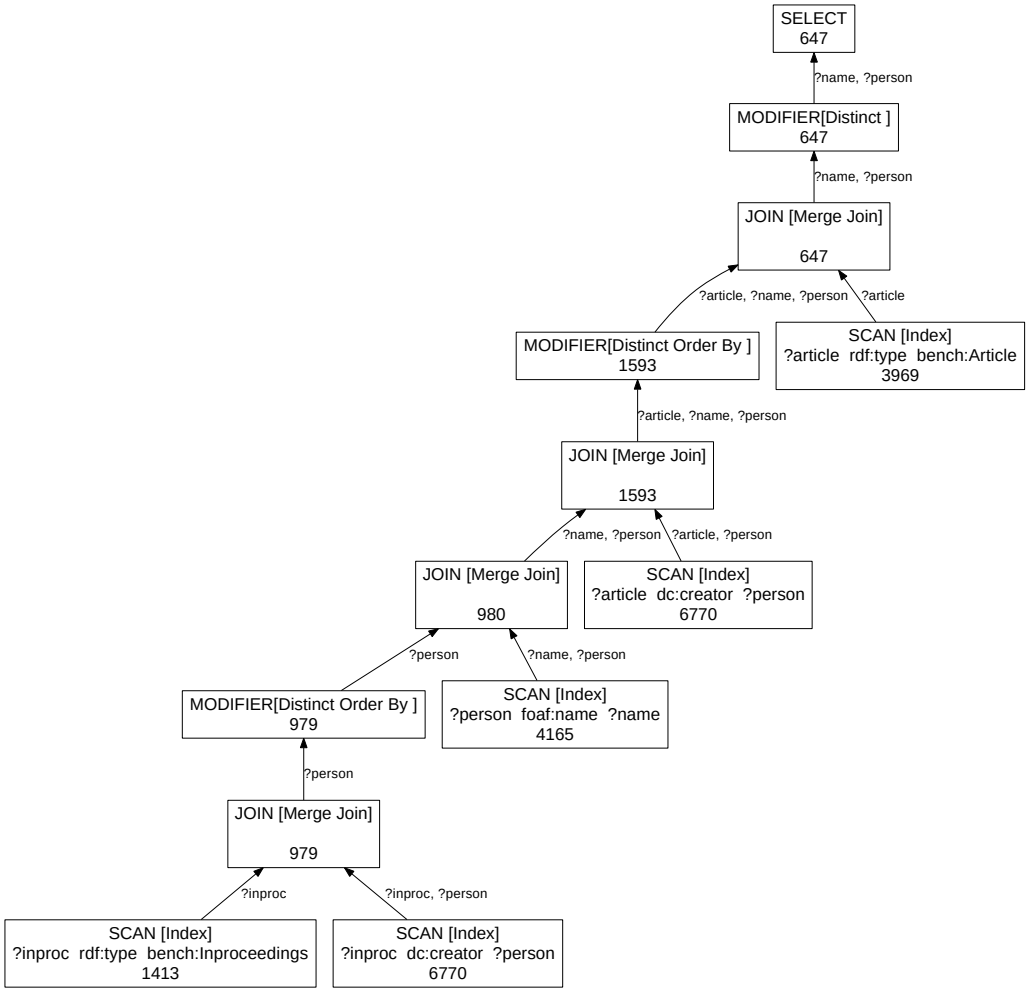


Obr. 6.7: Výsledný SQGM graf požiadavky T5a.

Požiadavky T5a a T5b testujú rôzne varianty spájania dvoch reťazcov. T5a implementuje implicitné spojenie na mene autora definovaním podmienky operácie *Filter*. T5b spája autorom použitím premennej `?name`.

Na výsledku T5a môžeme vidieť uprednostnenie heuristiky zakazujúcej kartézske súčiny pred tvarom stromu. Výsledný strom nie je *left-deep*, pretože v okamihu, keď algoritmus nenašiel vhodnú reláciu na doplnenie bez kartézskych súčinov, spracoval zvyšné relácie ako samostatný podstrom. Tento sa následne pripojil ako pravý podstrom operácie *Join*.

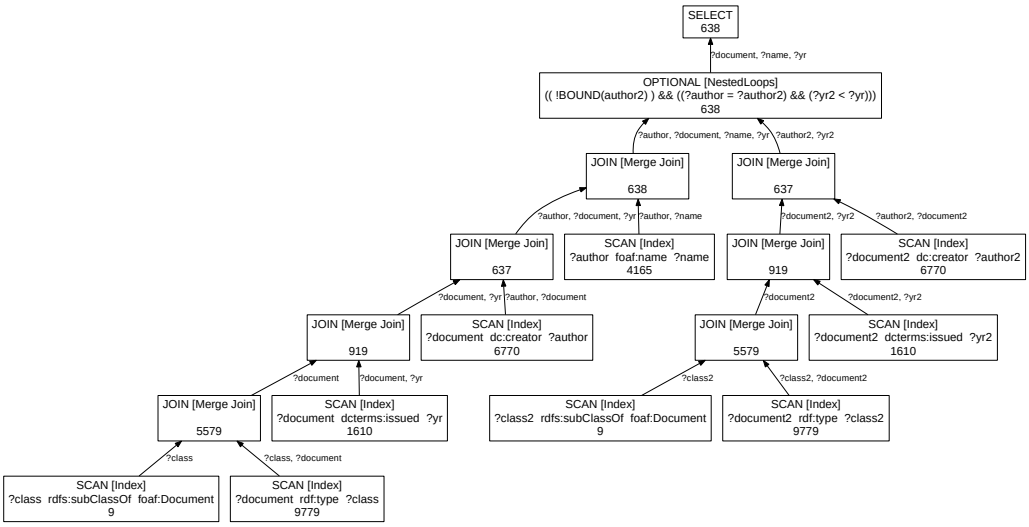
V oboch variantach vidieť snahu o propagáciu vlastnosti `DISTINCT`, jej pripojením k operácii *OrderBy*. Táto je opakovaná, pretože priebežná projekcia môže potenciálne vytvárať nové duplicity.



Obr. 6.8: Výsledný SQGM graf požiadavky T5b.

T6 *Pre každý rok, vráti množinu všetkých publikácií autorov, ktorí nepublikovali v predchádzajúcich rokoch.*

```
SELECT ?yr ?name ?document
WHERE {
  ?class rdfs:subClassOf foaf:Document .
  ?document rdf:type ?class .
  ?document dcterms:issued ?yr .
  ?document dc:creator ?author .
  ?author foaf:name ?name
  OPTIONAL {
    ?class2 rdfs:subClassOf foaf:Document .
    ?document2 rdf:type ?class2 .
    ?document2 dcterms:issued ?yr2 .
    ?document2 dc:creator ?author2
    FILTER (?author=?author2 && ?yr2<?yr)
  } FILTER (!bound(?author2))
}
```

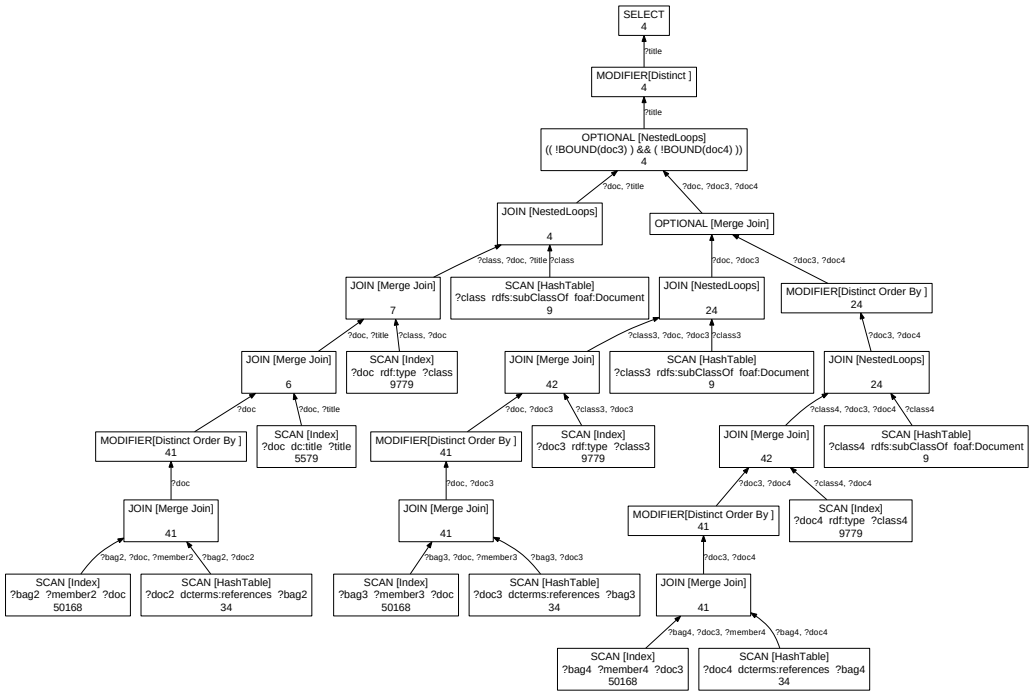


Obr. 6.9: Výsledný SQGM graf požiadavky T6.

Požiadavka T6 vyjadruje negáciu vyjadrenú pomocou `OPTIONAL`, `FILTER` a `BOUND`. Na výsledku môžeme vidieť, ako `OPTIONAL` nútene rozvetvuje výsledný strom.

T7 *Vráti názvy všetkých článkov, ktoré boli aspoň raz citované článkami, ktoré boli taktiež niekedy citované.*

```
SELECT DISTINCT ?title
WHERE {
  ?class rdfs:subClassOf foaf:Document .
  ?doc rdf:type ?class .
  ?doc dc:title ?title .
  ?bag2 ?member2 ?doc .
  ?doc2 dcterms:references ?bag2
  OPTIONAL {
    ?class3 rdfs:subClassOf foaf:Document .
    ?doc3 rdf:type ?class3 .
    ?doc3 dcterms:references ?bag3 .
    ?bag3 ?member3 ?doc
    OPTIONAL {
      ?class4 rdfs:subClassOf foaf:Document .
      ?doc4 rdf:type ?class4 .
      ?doc4 dcterms:references ?bag4 .
      ?bag4 ?member4 ?doc3
    } FILTER (!bound(?doc4))
  } FILTER (!bound(?doc3))
}
```



Obr. 6.10: Výsledný SQGM graf požiadavky T7.

Požiadavka T7 implementuje dvojitú negáciu. Očakáva sa iba niekľko výsledkov, z dôvodu nekompletného citačného systému. V SP^{2}Bench navrhujú znovupoužitie grafových vzorov. Avšak to je v kontexte tejto práce problematické, ako to je zdôvodnené v časti 5.1.4.

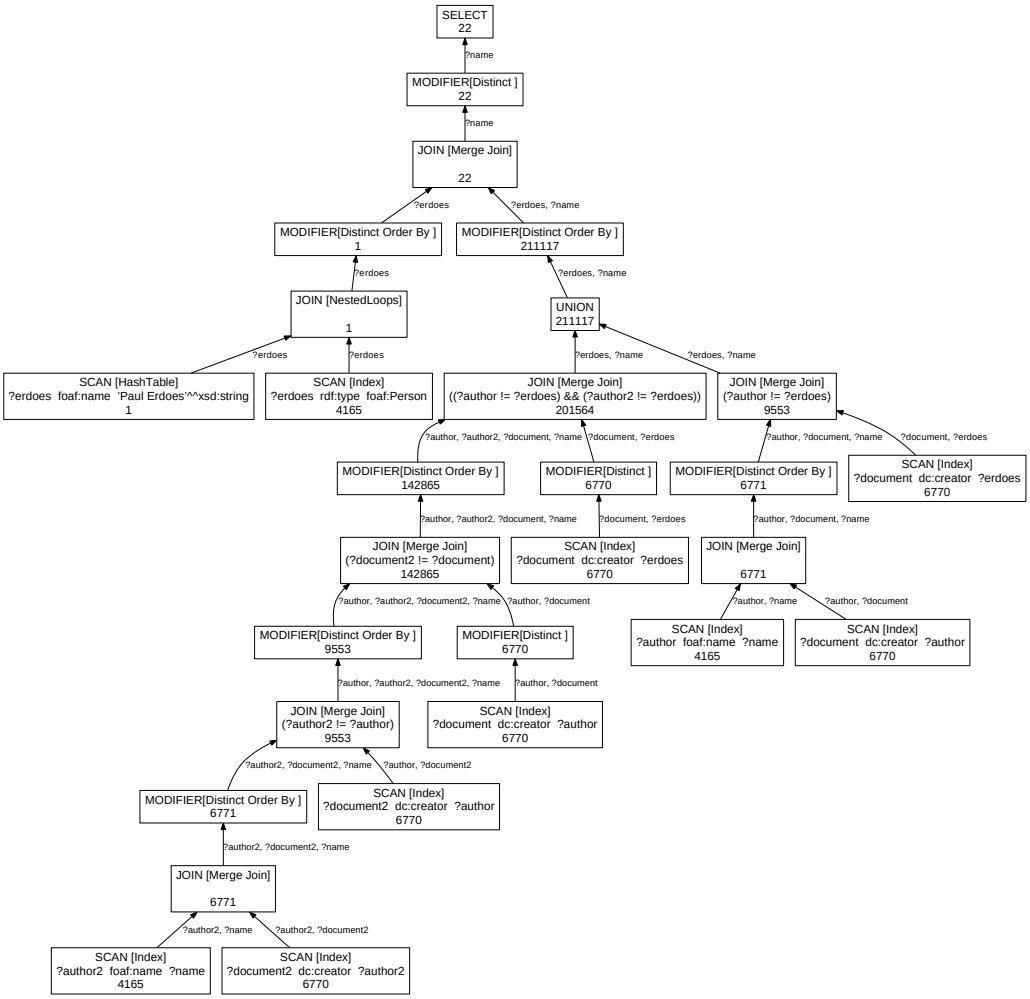
T8 *Vráti zoznam autorov, ktorý publikovali s* 'Paul Erdoes'*, alebo s autorom, ktorý publikoval s* 'Paul Erdoes'.

```
SELECT DISTINCT ?name
WHERE {
  ?erdoes rdf:type foaf:Person .
  ?erdoes foaf:name "Paul Erdoes"^^xsd:string .
  {
    ?document dc:creator ?erdoes .
    ?document dc:creator ?author .
    ?document2 dc:creator ?author .
    ?document2 dc:creator ?author2 .
    ?author2 foaf:name ?name
    FILTER (?author!=?erdoes &&
            ?document2!=?document &&
            ?author2!=?erdoes &&
            ?author2!=?author)
  } UNION {
    ?document dc:creator ?erdoes.
    ?document dc:creator ?author.
    ?author foaf:name ?name
    FILTER (?author!=?erdoes)
  }
}
```

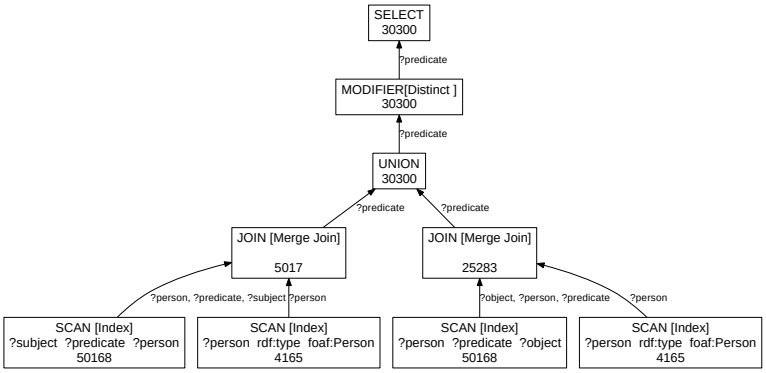
T8 je opäť zameraný na znovuvyužitie grafových vzorov ktoré tu nebolo možné. Na výsledku môžeme naproti tomu vidieť prácu s výrazmi operácie *Filter*. Konkrétne došlo k rozdeleniu jednje operácie *Filter* s veľkým výrazom, ktorý je v konjunktívnej forme na menšie. Tieto bolo následne možné propagovať nižšie v stome operácií.

T9 *Vráti vstupné a výstupné vlastnosti osôb.*

```
SELECT DISTINCT ?predicate
WHERE {
  {
    ?person rdf:type foaf:Person .
    ?subject ?predicate ?person
  } UNION {
    ?person rdf:type foaf:Person .
    ?person ?predicate ?object
  }
}
```



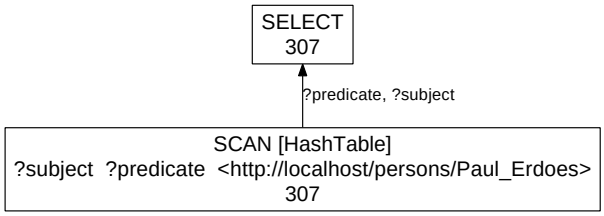
Obr. 6.11: Výsledný SQGM graf požiadavky T8.



Obr. 6.12: Výsledný SQGM graf požiadavky T9.

T10 *Vráti všetky subjekty, ktoré sú v nejakom vzťahu s* 'Paul Erdoes'.

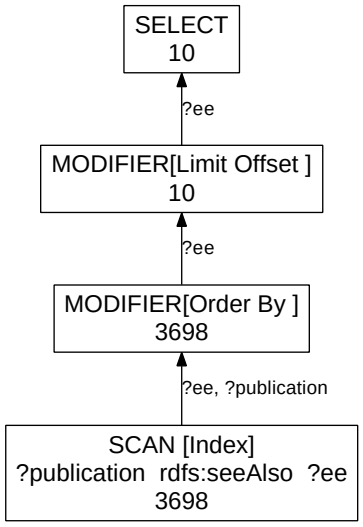
```
SELECT ?subject ?predicate
WHERE {
  ?subject ?predicate person:Paul_Erdoes
}
```



Obr. 6.13: Výsledný SQGM graf požiadavky T10.

T11 *Vráti maximálne 10 URL adries elektronických vydaní, začínajúc od 51. publikácie v lexikografickom poradí.*

```
SELECT ?ee
WHERE {
  ?publication rdfs:seeAlso ?ee
}
ORDER BY ?ee
LIMIT 10
OFFSET 50
```



Obr. 6.14: Výsledný SQGM graf požiadavky T11.

Kapitola 7

Záver

Výsledkom tejto diplomovej práce je prekladač jazyka Bobox, ktorý vykoná lexikálnu, syntaktickú a sémantickú analýzu a preloží zadanú požiadavku na model. Tento je možné použiť systémom Bobox na vytvorenie *pipeline*. Hlavný dôraz je kladený na optimalizáciu, ktorá zvládne väčšinu bežných požiadaviek použitím *logických* optimalizácií a *fyzickej* optimalizácie.

Logické optimálizácie aplikujú sadu logických prepisovacie pravidlá, ktorými zjednodušuje požiadavku odstraňovaním redundantných operácií a znižovaním úrovne zanorenia operácií. Množstvo spracovaných dát redukuje zmenšovaním medzivýsledkov, k čomu využíva propagáciu operácie *Filter* a propagáciu vlastností *DISTICT* a *REDUCED*. Vzhľad požiadavky nakoniec zjednocuje, aby bolo možné v budúcnosti naviazať cachovanie požiadaviek.

Fyzická optimalizácia buduje strom fyzických oprácií za pomoci dynamického programovania. Na redukciu prehľadávaného priestoru sú definované heuristiky, ako obmedzenie tvaru výsledného stromu na *left-deep* strom a nevytváranie kartézskych súčinov, pokiaľ si to požiadavka priamo nevynucuje. Generovanie stromu operácií sa vykonáva prechodom do hĺbky, čím je unožnené eliminovať vetvy s vyššiou cenou, ako priebežne nejnižšia cena.

Použité sumárne štatistiky sú zamerané primárne na určenie veľkosti dát operácie načítania dát a základné informácie o selektivite jednotlivých stĺpcov pri operácii *Join*. K odhadu ceny slúži obecný ohodnocovací model, založený na ohodnotení ceny načítania, prenosu a spracovania dát. Cena spracovania dát je obecným odhadom náročnosti fyzických operácií, ktoré avšak momentálne nie sú dostupné.

S výsledným prekladačom sme experimentovali na testoch definovaných v testovacom prostredí SP^{2}Bench. Výsledné modely splnili naše očakávania, vzhľadom na použité metódy optimalizácie, heuristiky a možnosti iba hrubých odhadov ceny. Odhady veľkostí výsledných dát testovaných požiadaviek vo väčšine prípadov boli rozumné, vzhľadom na použité sumárne štatistiky a dostupné informácie pre rozhodovanie.

7.1 Ďalšie smery vývoja

Vzhľadom na rozsiahlosť riešenej problematiky a aktívny výskum v oblasti reprezentácie RDF dát a optimalizácie prístupu k nim, je možné prácu vylepšiť a rozvinúť viacerými smermi.

Vylepšenie je možné napríklad v oblasti odhadov veľkosti dát a štatistík. Upresnenie a rozšírenie dostupných štatistík pre operácie *Join* a *Filter* za účelom zlepšenia ich odhadov veľkosti dát. Zisťovaním schémy dát je možné lepšie pokryť vzťahy medzi reláciami vstupujúcimi do operácie *Join*. Na základe týchto podrobnejších informácií bude možné doplniť odhady redukcie operácií *filter* a *Distinct*.

Po implementácii *back-endu* a databáze je vhodné upresniť cenový model, ktorý bude presnejšie vystihovať aktuálnu implemenáciu. Pre poskytovanie štatistík vytvoriť modul, ktorý ich bude poskytovať podľa reálnych dát.

Optimalizátor môže byť rozšírený o možnosť použitia randomizovaných algoritmov, využívajúcich náhodné prechádzky v grafe. Tieto algoritmy by sa použili pri spájaní väčšieho množstva relácií (viac ako 10 [8], keď sa začina výraznejšie prejavovať nevýhoda enumeratívneho algoritmu). Ukazuje, že randomizované algoritmy dokážu nájsť rozumné plány veľmi rýchlo [8].

Literatúra

[1] David Bednárek, Jiří Dokulil, Jakub Yaghob, and Filip Zavoral. The bobox project - a parallel native repository for semi-structured data and the semantic web. In *ITAT 2009 - IX. Informačné technológie - aplikácie a teória*, pages 44–59. PONT Slovakia, 2009.

[2] Jeremy J. Carroll and Graham Klyne. Resource description framework (RDF): Concepts and abstract syntax. W3C recommendation, W3C, February 2004. http://www.w3.org/TR/2004/REC-rdf-concepts-20040210/.

[3] Surajit Chaudhuri. An overview of query optimization in relational systems. In *In PODS*, pages 34–43, 1998.

[4] Abdelkader Hameurlain and Franck Morvan. Evolution of query optimization methods. pages 211–242, 2009.

[5] Olaf Hartig and Ralf Heese. The SPARQL query graph model for query optimization. In *ESWC '07: Proceedings of the 4th European conference on The Semantic Web*, pages 564–578, Berlin, Heidelberg, 2007. Springer-Verlag.

[6] Y. E. Ioannidis and Younkyung Kang. Randomized algorithms for optimizing large join queries. *SIGMOD Rec.*, 19(2):312–321, 1990.

[7] Yannis Ioannidis. The history of histograms (abridged). In *VLDB '2003: Proceedings of the 29th international conference on Very large data bases*, pages 19–30. VLDB Endowment, 2003.

[8] Yannis E. Ioannidis. Query optimization. volume 28, pages 121–123, New York, NY, USA, 1996. ACM.

[9] Yannis E. Ioannidis and Stavros Christodoulakis. On the propagation of errors in the size of join results. In *SIGMOD '91: Proceedings of the 1991 ACM SIGMOD international conference on Management of data*, pages 268–277, New York, NY, USA, 1991. ACM.

[10] Rosana S. G. Lanzelotte and Patrick Valduriez. Extending the search strategy in a query optimizer. In *VLDB '91: Proceedings of the 17th International Conference on Very Large Data Bases*, pages 363–373, San Francisco, CA, USA, 1991. Morgan Kaufmann Publishers Inc.

[11] Rosana S. G. Lanzelotte, Mohamed Zaďt, and Allen Van Gelder. Measuring the effectiveness of optimization. search strategies. In Eric Simon, editor, *BDA*, pages 162–. INRIA, 1992.

[12] W3C M. Duerst, M. Suignard. Internationalized resource identifiers (IRIs). Request for comments, IETF, January 2005. http://www.ietf.org/rfc/rfc3987.txt.

[13] Hooran MahmoudiNasab and Sherif Sakr. An experimental evaluation of relational RDF storage and querying techniques. In *WWW '10: Proceedings of the 19th International World Wide Web Conference*, 2010.

[14] Hamid Pirahesh, Joseph M. Hellerstein, and Waqar Hasan. Extensible/rule based query rewrite optimization in starburst. In *In SIGMOD*, pages 39–48, 1992.

[15] Eric Prud'hommeaux and Andy Seaborne. SPARQL query language for RDF. W3C recommendation, W3C, January 2008. http://www.w3.org/TR/2008/REC-rdf-sparql-query-20080115/.

[16] James Reinders. *Intel threading building blocks*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2007.

[17] Michael Schmidt, Michael Meier 0002, and Georg Lausen. Foundations of SPARQL query optimization. *CoRR*, abs/0812.3788, 2008.

[18] Michael Schmidt, Thomas Hornung, Georg Lausen, and Christoph Pinkel. SP^{2}Bench: A SPARQL performance benchmark. In *ICDE '09: Proceedings of the 2009 IEEE International Conference on Data Engineering*, pages 222–233, Washington, DC, USA, 2009. IEEE Computer Society.

[19] Markus Stocker, Andy Seaborne, Abraham Bernstein, Christoph Kiefer, and Dave Reynolds. SPARQL basic graph pattern optimization using selectivity estimation. In *WWW '08: Proceeding of the 17th international conference on World Wide Web*, pages 595–604, New York, NY, USA, 2008. ACM.