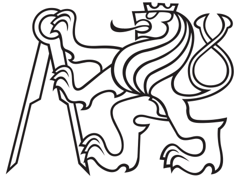
Bakalárska práca



České
vysoké
učení technické
v Praze

F3

Fakulta elektrotechnická
Katedra počítačov

Sémantický objektový dotazovací jazyk

Martin Krupa

Školiteľ: Ing. Martin Ledvinka
Študijný program: Softwarové inženýrství a technologie
Január 2020

Poďakovanie

Chcem sa poďakovať môjmu školiteľovi pánovi Ing. Martinovi Ledvinkovi za pomoc, usmernenie, cenné rady a čas, ktoré mi bol ochotný venovať pri vypracovaní mojej práce. Vďaka patrí aj mojim rodičom a sestre za neustálu podporu.

Prehlásenie

Prehlasujem, že som predloženú prácu vypracoval samostatne, a že som uviedol všetkú použitú literatúru.

V Prahe, 6. januára 2020

／
Abstrakt

Ako sa sémantický web stáva bežnejším, enterprise aplikácie viac využívajú sémantické dáta. Technológie určené na vývoj týchto aplikácií sa tejto zmene prispôsobujú ťažšie. Vrstva v enterprise aplikáciách, určená pre ukladanie a správu dát v databáze, bola pôvodne navrhnutá tak, aby pracovala s relačnými databázami. Táto práca sa zaoberá úpravou spôsobu, akým táto vrstva pracuje so sémantickými dátami.

Výsledkom práce je návrh objektového dotazovacieho jazyka pre sémantické dáta. Prototyp prekladača sa úspešne integroval do knižnice JOPA. Úspešná integrácia bola overená pomocou prekladu testovacích dotazov.

Ako riešenie problému bol navrhnutý a vytvorený prekladač. Jeho účelom je spracovať dotaz napísaný v objektovom dotazovacom jazyku a následne vytvoriť jeho ekvivalent v dotazovacom jazyku pre sémantické dáta.

Kľúčové slová: Sémantický Web, Java, JOPA, JPA, JPQL, RDF, SPARQL

Školiteľ: Ing. Martin Ledvinka
Katedra počítačů,
Karlovo nám. 13,
Praha 2

Abstract

As the Semantic Web is becoming more common, enterprise applications start to use Semantic data as well. However, technologies for developing such applications adapt harder to this change. Mostly because data access software libraries were mostly designed to work with relational databases. The goal of this work is to extend the current possibilities of using Semantic Web technologies when accessing data.

The result is a design of an object query language for Semantic data. This language has been implemented by creating a translator to the standard query language for the Semantic Web. A prototype of the translator has been successfully integrated into JOPA – a data access library for the Semantic Web. Successful integration was validated by testing queries.

Keywords: Semantic Web, Java, JOPA, JPA, JPQL, RDF, SPARQL

Obsah

1 Úvod 1

2 Náhľad použitých technológií 3

2.1 Sémantický Web 3

2.2 RDF 4

2.2.1 RDF graf a trojice 4

2.3 SPARQL Protocol and RDF Query Language 5

2.3.1 Formy dotazu 6

2.3.2 Spôsob filtrovania 6

2.4 Java Persistence Query Language 7

2.4.1 Selekcia a filtrovanie 8

2.4.2 Query Parametre 9

2.4.3 Join 9

2.5 ANTLR 10

2.5.1 Abstract Syntax Tree 10

2.5.2 Gramatika 11

2.5.3 Parse Tree Listener 12

3 Tvorba návrhu 13

3.1 Analýza existujúcich riešení 13

3.1.1 Využitie JPA Criteria Query nad RDF 13

3.1.2 Empire 14

3.1.3 JOPA 14

3.1.4 Prekladač 15

3.2 Analýza rozdielov medzi SPARQL a JPQL 16

3.2.1 Rozdiely v konštruktoch 16

3.3 Návrh riešenia 19

3.3.1 Gramatika 19

3.3.2 Vzorový dotaz 20

3.3.3 Návrh implementačnej časti . 23

3.3.4 Návrh integrácie do JOPA . . 25

4 Implementácia prototypu 27

4.1 Tvorba gramatiky 27

4.2 Implementácia návrhu 27

5 Testovanie prototypu **31**

5.1 Unit Testy 31

5.2 Integračné Testy 32

6 Záver **35**

6.1 Dosiahnuté ciele projektu 35

6.2 Budúci vývoj 36

7 Inštalácia **37**

A Literatúra **39**

B Zoznam skratiek **41**

C Obsah CD **43**

D Zadanie práce **45**

Obrázky

2.1 RDF graf [Sau18] 5

2.2 Abstract Syntax Tree [PAR12] . 11

2.3 Náhľad parsovania [PAR12] 11

2.4 Ukážka fungovania ParseTreeWalkeru [PAR12] 12

3.1 Zapojenie prekladača HQL2SPARQL [HBL09] 15

3.2 Vzorový dotaz a jeho rozdelenie podľa kluazúl 19

3.3 Šablóna SelectStatementTemplate 20

3.4 Šablóna SelectFromClause 21

3.5 Šablóna WhereClause 22

3.6 Šablóna GroupByClause spolu s OrderByClause 23

3.7 Ilustrácia procesu prekladu dotazu 24

3.8 Diagram ilustrujúci intergáciu prekladača do knižnice JOPA 25

4.1 Class Diagram rozhrania prekladača 28

Úvod

V súčasnosti je WWW preplnený rôznymi formami informácii. Preto vznikol súbor štandardov a technológii nazývaný sémanticky web. Jeho účelom je umožniť strojové spracovanie webu a zjednotiť spôsob reprezentácie informácií. Pre webové aplikácie to znamená zjednodušenie zdieľania a prístupu k informáciám medzi sebou. Informácie sa popisujú pomocou štruktúry RDF. Enterprise aplikácie sa tiež postupne prispôsobujú sémantickému webu a ako spôsob ukladania dát využívajú pravé RDF formu. Tejto zmene sa však horšie prispôsobujú technológie, ktoré umožňujú vývoj aplikácií. Persistenčné vrstvy pre enterprise aplikácie boli pôvodne navrhnuté, aby podporovali predovšetkým využitie relačných databáz. Dôsledkom je obtiažnejší vývoj aplikácií pre sémanticky web. Katedra počítačov úspešne vyvinula knižnicu JOPA, ktorá tvorí základ pre aplikácie využívajúce RDF, avšak chýba jej objektový dotazovací jazyk pre sémantické dáta. Preto bude mojou úlohou navrhnúť sémanticky objektový dotazovací jazyk, ktorý bude vychádzať z JPQL a následne prototyp návrhu zaintegrovať do knižnice JOPA.

V nasledujúcej kapitole predstavím technológie, ktoré úzko súvisia s mojou úlohou. V tretej kapitole popíšem analýzu existujúcich riešení spolu s návrhom riešenia úlohy. Obsahom štvrtej kapitoly bude popis implementácie prototypu na základe vytvoreného návrhu a zaintegrovanie tohto prototypu do knižnice JOPA. Návrh a implementáciu prototypu zakončím kapitolou, v ktorej sa budem venovať popisu, akým spôsobom som prototyp otestoval v rámci integrácie do knižnice JOPA.

Ciele, ktoré chcem v tejto práci dosiahnuť sú:

- Analyzovať existujúce prototypy objektového dotazovacieho jazyka pre sémantické dáta, zhodnotiť ich silné a slabé stránky a vhodnosť ich integrácie do knižnice JOPA.
- Navrhnúť podmnožinu objektového dotazovacieho jazyka pre sémantické dáta, ktorá sa bude prekladať do dotazovacieho jazyku SPARQL. Podmnožina musí podporovať nasledujúce konštrukty: selekcia, join, filtrovanie a vybrané agregačné funkcie. Inšpiráciou navrhovaného jazyka bude JPQL.
- Vytvoriť prototyp implementácie navrhnutého dotazovacieho jazyka a zaintegrovať tento prototyp do knižnice JOPA.
- Overiť správnosť implementácie môjho vytvoreného jazyka porovnaním výsledkov typických dotazov napísaných v jazyku SPARQL.

Kapitola 2

Náhľad použitých technológií

Táto kapitola by mala poskytnúť istý základný popis hlavných technológií, ktoré úzko súvisia s návrhom a implementáciou môjho riešenia.

Moje riešenie je určené pre enterprise aplikácie, ktoré ukladajú dáta vo forme RDF. Používajú databázy, ktoré označujeme ako triple store a ich dotazovacím jazykom je SPARQL. Tieto technológie úzko súvisia s pojmom sémantický web. Práve sémantický web využíva formu RDF pre popis dát.

Pri návrhu a implementácii prototypu riešenia vychádzam z objektového dotazovacieho jazyka JPQL. V súvislosti s realizáciou prototypu je dôležitý aj program ANTLR. Ten bude nápomocný pri preklade dotazov z jazyka JPQL do jazyka SPARQL.

2.1 Sémantický Web

Sémantický web je rozšírenie už existujúceho WWW o metadáta zverejnených informácií. Pomocou technológií a štandardov sú k aktuálnemu obsahu webu pridané dodatočné informácie tak, aby poskytovali jednotný spôsob jeho reprezentácie. Výsledkom by mala byť jednoduchšia správa, zdieľanie a znovupoužiteľnosť dát medzi aplikáciami. Najdôležitejšie technológie sú RDF a SPARQL [Sau18].

2.2 RDF

Resource Description Framework alebo RDF je štruktúra, ktorej úlohou je reprezentovať informácie na webe [RC14]. Podoba tejto štruktúry je formovaná tak, aby bola rovnako čitateľná pre stroj, ako aj pre človeka. Účelom RDF je mať jednoduchý dátový model, s ktorým bude ľahké pracovať.

Pomocou RDF sa snažíme popísať veci, alebo objekty, ktoré nazývame zdroje. Zdrojom môže byť čokoľvek. Napríklad fyzické veci, dokumenty, abstraktné návrhy, čísla, reťazce a mnoho ďalších [RC14].

Zdroje delíme na dve skupiny: referentov, ktorý sú reprezentovaní ako IRI (Internationalized Resource Identifier) a literály, ktoré reprezentujú hodnotu. Pri literáloch musíme definovať ich dátový typ spolu s rozsahom hodnoty. Dátový typ píšeme za hodnotu a začína značkou ^^. Literál typu langString môže obsahovať jazykové tagy, ktoré rovnako píšeme za hodnotu a značíme ako @ spolu s jazykovou skratkou. Listing 2.1 zobrazuje použitie literálov spolu s dátovými typmi a jazykovým tagom.

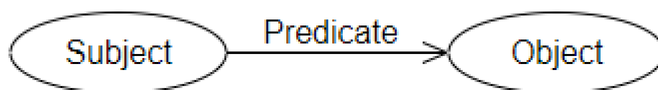
Listing 2.1: Ukážka používania literálov [HS13]

```
PREFIX dt: <http://example.org/datatype#>
PREFIX ns: <http://example.org/ns#>
PREFIX xsd: <http://example.org/XMLSchema#>

?x ns:p "cat"@en .
?y ns:p "42"^^xsd:integer .
?z ns:p "abc"^^dt:specialDatatype .
```

2.2.1 RDF graf a trojice

Základom podoby RDF je zoskupenie, ktoré voláme trojica [RC14]. Každá trojica obsahuje subjekt, predikát a objekt. V trojici predikát určuje vzťah medzi subjektom a objektom. Subjekt spolu s objektom zaraďujeme medzi zdroje.

Trojice označujeme ako RDF graf a môžeme ich zobraziť pomocou orientovaného grafu (obrázok 2.1), kde budú dva uzly spojené hranou [RC14]. V



Obrázok 2.1: RDF graf [Sau18]

tomto grafe je jeden uzol subjekt, druhý objekt a hrana predstavuje predikát. Kolekciu viacerých RDF grafov označujeme ako RDF dataset.

2.3 SPARQL Protocol and RDF Query Language

SPARQL je sémantický dotazovací jazyk pre grafové databázy, ktoré ukladajú svoje dáta vo forme RDF. Takéto databázy zaraďujeme do skupiny triple store databázy. Pomocou SPARQL dokážeme vytvárať dotazy, ktoré nám umožňujú získať a pracovať s dátami, ktoré sú prirodzene uložené vo forme RDF, alebo sú do RDF formy prevedené pomocou nejakého prostredníka [HS13].

V dotazovacom jazyku SPARQL môžeme využívať RDF referencie ako sú IRI. V rámci SPARQL sú tieto referencie absolútne [HS13].

Dotazy písane v SPARQL najčastejšie obsahujú trojice. Jediným rozdielom medzi trojicami v SPARQL a RDF je, že v SPARQL môže byť každá časť trojice premennou [HS13]. Ukážku SPARQL dotazu obsahujúceho trojice s prefixom i bez prefixu môžeme vidieť v Listingu 2.2.

Listing 2.2: Ukážka trojice v dotaze SPARQL

```
PREFIX ex: <http://www.example.org/>

SELECT ?name ?age
WHERE {
   ?x ex:name ?name .
   ?x <http://www.example.org/age> ?age .
}
```

V dotaze sa môžu vyskytovať aj literály. Tie nahrádzajú konkrétne hodnoty a chovajú sa rovnako, ako v RDF grafe. Podľa rovnakých pravidiel, ako pri RDF im definujeme dátový typ, prípadne im pripisujeme jazykový tag.

2.3.1 Formy dotazu

SPARQL môže mať rôzne formy dotazu [HS13]. Všetko záleží od toho, čo chceme získať ako výsledok dotazu. Môžeme použiť bežný dotaz typu SELECT, ktorý sa chová rovnako ako v iných dotazovacích jazykoch. V SPARQL je možné zvoliť aj iné formy dotazu. CONSTRUCT query nám vráti dáta vo forme RDF grafu. Pomocou typu ASK zisťujeme, či existuje výsledok dotazu.

2.3.2 Spôsob filtrovania

V prípade že chceme z databázy vybrať určitú množinu dát, využijeme funkciu WHERE. Do bloku tejto funkcie dávame trojicu, alebo sadu trojíc. Obsah tohto bloku zodpovedá RDF forme dát, ktoré chcem získať [HS13].

Často sa môže stať, že chceme na tieto dáta uplatniť dodatočné podmienky, alebo obmedzenia, ktoré už nie sú súčasťou trojice. V takom prípade môžeme využiť jednu, prípadne viac variant filtrovania dát.

Základnou variantou je funkcia FILTER. Do tejto funkcie dopĺňame dodatočné podmienky, ktoré už nie sú súčasťou trojice, ale chceme podľa nich špecifikovať hľadané dáta. Pokiaľ chceme obmedziť dáta pomocou operátorov <, >, <=, >=, tak je to práve obsahom funkcie FILTER. Vo FILTRI môžeme zároveň využiť niektorú z množstva funkcií. Napríklad funkcia REGEX nahrádza operátor LIKE. Pokiaľ funkcia filter obsahuje aspoň dve podmienky, oddeľujeme ich pomocou logických operátorov, ako sú "&&" a "||". V Listingu 2.3 je zobrazená ukážku použitia funkcie FILTER.

Listing 2.3: Ukážka funkcie FILTER

```
PREFIX ex: <http://www.example.org/>
SELECT ?x WHERE {
   ?x a ex:person .
   ?x ex:age ?age .
   FILTER ( ?age > 30)
}
```

Ďalšou variantou je funkcia FILTER NOT EXISTS. Jeho použitie je ukázané v Listingu 2.4. Táto funkcia nahrádza unárny operátor NOT a obsahuje trojicu, či sadu trojíc, ktoré zodpovedajú nepožadovaným dátam vo forme

RDF. Môže obsahovať vlastnú funkciu FILTER, kde doplníme dodatočné podmienky pre tieto dáta.

Listing 2.4: Ukážka funkcie FILTER NOT EXISTS

```
PREFIX ex: <http://www.example.org/>
SELECT ?x WHERE {
   ?x a ex:person .
   FILTER NOT EXISTS {
      ?x ex:age ?age .
      FILTER (?age <= 30)
   }
}
```

Funkcia FILTER EXISTS testuje a vracia informáciu, či zvolenú trojicu je možné nájsť medzi dátami [HS13]. Použitie FILTER EXISTS môžeme vidieť v Listingu 2.5.

Listing 2.5: Ukážka funkcie FILTER EXISTS

```
PREFIX ex: <http://www.example.org/>
SELECT ?x WHERE {
   ?x a ex:person .
   FILTER EXISTS { ?x ex:name ?name .}
}
```

2.4 Java Persistence Query Language

Java Persistence API. JPA je framework pre persistenčnú vrstvu v jazyku Java [MM09]. Jej úlohou je správa dát, ktoré chceme uložiť do databázy. Poskytuje objektovo relačné mapovanie spolu s definíciami pre dotazovací jazyk JPQL a criteria API [JCP09].

Criteria API predstavuje alternatívny spôsob tvorby dotazov, využívajúci programovací jazyk Java namiesto JPQL [MM09]. Listing 2.6 zobrazuje príklad použitia criteria API.

Listing 2.6: Ukážka criteria API

```
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery<Person> c = cb.createQuery(Person.class);
Root<Person> person = c.from(Person.class);
c.select(person);
List<Person> persons = em.createQuery(c).getResultList();
```

JPQL je primárnym dotazovacím jazykom, ktorý podporuje Java Persistence API. Pracuje s logickým modelom entít [MM09]. Je veľmi podobný ako dotazovací jazyk SQL. JPQL je rozšírený o funkcie ako:

- jednotné, alebo viacnásobné hodnoty výsledkov
- prirodzenejší join, inner aj outer
- mapovanie výsledkov dotazu na entity a nepersistentné triedy

Nasledujúce sekcie popisujú vlastnosti JPQL, ktoré sú relevantné k téme mojej práce.

2.4.1 Selekcia a filtrovanie

Selekcia využíva klauzulu SelectFrom (Listing 2.7), podobne ako SQL. Rozdielom je, že v JPQL neselektujeme z konkrétnej tabuľky, ale z konkrétnej triedy entity. Teda tam, kde by sa mal nachádzať názov tabuľky, máme názov entity spolu s jej aliasom [MM09].

Listing 2.7: SelectFrom klauzule

```
SELECT b FROM Book b
```

Selektovať môžeme aj entity, ktoré nie sú obsiahnuté v klauzule FROM (Listing 2.7). Stačí, ak sú asociované s entitou, ktorá sa v klauzule FROM nachádza [MM09]. Tu treba dať pozor, aby sme touto formou nežiadali dva rôzne typy návratových hodnôt.

Na príklade v Listingu 2.8 môžeme vidieť, že návratová hodnota bude entita autor i keď sa nenachádza v klauzule FROM.

Listing 2.8: Selekcia asociovanej entity

```
SELECT b.author FROM Book b
```

Filtrovanie využíva klauzulu WHERE, za ktorú sa píše zoznam podmienok. Porovnávacie aj logické operátory sú rovnaké ako v SQL [MM09]. Môže obsahovať aj agregačné funkcie, či parametre.

2.4.2 Query Parametre

JPQL podporuje dva typy parametrov [MM09]. Jeden typ je pozičný, zapisuje sa ako otáznik spolu s číslom. pri jeho náhrade sa určuje pomocou čísla. Druhý typ je menný parameter. Zapisuje sa pomocou dvojbodky a názvu. Pri jeho náhrade za hodnotu sa určuje pomocou názvu.

Názornú ukážka vymenovaných druhov parametrov je vidieť v Listingu 2.9. V prvom dotaze sa nachádzajú pozičné parametre, v druhom dotaze menné parametre.

Listing 2.9: Ukážka pozičných a menných parametrov

```
SELECT b FROM Book b WHERE b.title = ?1

SELECT b FROM Book b WHERE b.title = :title
```

2.4.3 Join

Pri spájaní v dotaze sa kombinujú výsledky z viacerých entít podľa zlučovacích podmienok. Spojenie entít spolu s podmienkami môžeme zapísať použitím operátoru JOIN v klauzule FROM, alebo pomocou podmienok v klauzule WHERE (Listing 2.10) [MM09]. Operátor JOIN môžeme použiť v súvislosti

s jedným asociovaným objektom rovnako ako s kolekciou. Ak to situácia vyžaduje, je možné využiť aj viacnásobný JOIN.

Listing 2.10: Ukážka zápisu joinu dvoch entít

```
SELECT b.title FROM Author a, Book b WHERE a = b.author

SELECT b.title FROM Author a JOIN a.books b
```

Inner Join. Inner join medzi dvoma entitami vráti ako výsledok objekty oboch typov entít, pokiaľ spĺňajú všetky zlučovacie podmienky [MM09]. Pre zapísanie inner join je možné využiť obidva spôsoby znázornené v Listingu 2.10.

Outer Join. Outer join medzi dvoma entitami vráti výsledok, v ktorom je vyžadovaná iba jedna entita. Druhá entita sa nachádza vo výsledku len ak existuje spojenie s prvou [MM09]. Pri takomto type joinu píšeme operátor v tvare LEFT JOIN (Listing 2.11).

Listing 2.11: Outer Join

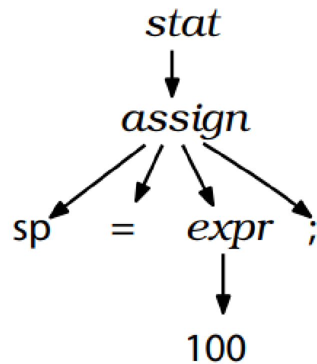
```
SELECT e, d FROM Employee e LEFT JOIN e.department d
```

2.5 ANTLR

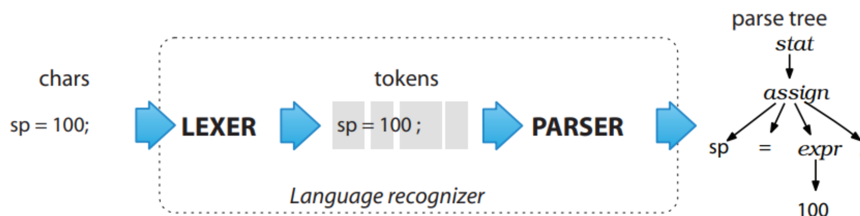
ANTLR, alebo Another Tool for Language Recognition, je generátor nástroja pre syntaktickú analýzu, ktorý je písaný v jazyku Java. Využívame ho na čítanie, spracovanie a preklad štruktúrovaného textu.

2.5.1 Abstract Syntax Tree

AST (obrázok 2.2) je spôsob reprezentácie akéhokoľvek štruktúrovaného textu do podoby štruktúry stromu.



Obrázok 2.2: Abstract Syntax Tree [PAR12]



Obrázok 2.3: Náhľad parsovania [PAR12]

Práve parser dokáže previesť predaný text na AST [Tom17]. Za pomoci pravidiel rozparsuje text do podoby AST, kde listy sú konkrétne časti textu a uzly predstavujú nami definované pravidlá.

2.5.2 Gramatika

Gramatika je súbor pravidiel, kde každé pravidlo rozdelí text na istú frázu [PAR12]. ANTLR je program, ktorý na základe gramatiky ďalej vygeneruje programy ako je Lexer a Parser.

Pomocou lexeru sa text rozdelí na tokeny, ktoré sa následne predajú parseru. Ten vyhodnotí, pod ktoré pravidlá tieto tokeny zaradiť a podľa toho zostaví AST (obrázok 2.3).

Obrázok 2.4: Ukážka fungovania ParseTreeWalkeru [PAR12]

2.5.3 Parse Tree Listener

Vďaka tomu, že je ANTLR písaný v jazyku Java, je možné vygenerované súbory pretvoriť do Java aplikácie a následne s ňou pracovať. Medzi vygenerovanými súbormi je aj rozhranie, ktoré poskytuje možnosť pracovať s textom počas priechodu cez celý AST.

Týmto rozhraním je ParseTreeListener, ktoré sa správa ako visitor [GHJV95] a reaguje na udalosti pri prechode AST. Proces prechodu zabezpečuje rozhranie ParseTreeWalker [PAR12]. Listener reaguje na pohyb walkeru po uzloch stromu. Vždy pri vstupe, alebo výstupe z nejakého uzlu (pravidla) sa spustí funkcia listeneru, v ktorej už môžeme pracovať s kontextom navštíveného uzlu.

Na obrázku 2.4 je znázornené, ako bude ParseTreeWalker postupovať cez AST.

Pomocou kontextu máme k dispozícii celý podstrom daného uzlu. Preto máme v tejto funkcii možnosť pracovať so všetkým pravidlami a textom, ktoré spadajú do podstromu navštíveného uzlu (pravidla).

Kapitola 3

Tvorba návrhu

3.1 Analýza existujúcich riešení

.

3.1.1 Využitie JPA Criteria Query nad RDF

V súvislosti s JPA sa objavuje problém pre backend, ktorý využíva RDF ako formu ukladania dát. Zo strany JPA nie je poskytnutá žiadna podpora pre takýto typ backendu [SL17].

Objektové mapovanie by mohlo byť využiteľné. Upravením do podoby nejakého objektového RDF mapovania by sme mohli dosiahnuť dotazovanie nad objektami, ktoré by boli mapované ako RDF trojice [SL17]. S využitím Java modelu by sa nejednalo o statické prepisovanie dotazov a vďaka tomu by nebolo závislé len na jednom backende.

Úpravou dôležitých JPA komponent a criteria API sa dá dosiahnuť RDF mapovanie [SL17]. Java triedy by sme mohli označiť tak, aby reprezentovali objekty aj v RDF databáze. Pridali by sa identifikátory, ktoré by odpovedali IRI.

EntityManager a Criteria Builder. EntityManager je základná komponenta pre správu entít. Zabezpečuje základné funkcie Create, Read, Update, Delete, ktoré označujeme ako CRUD a vytváranie dotazov. Poskytuje funkciu getCriteriaBuilder, ktorá vráti objekt criteriaBuilderu. Ten slúži na vytváranie criteria queries, selekcií a výrazov [SL17].

Architektúra využitia JPA nad RDF. Architektúra takejto implementácie zahrňuje doplnkové anotácie do Java triedy, pomocou ktorých prebieha objektové RDF mapovanie [SL17]. Toto mapovanie nie je jednoznačné a môže existovať viac spôsobov, ako to dosiahnuť pre skupinu tried.

Okrem anotácií môžu pribudnúť aj ďalšie rozhrania. Ich úlohou by bolo pomáhať pri identifikácii a prevedení dát na RDF. Naviac môžu obsahovať implementáciu pre správu entít a tvorbu dotazov, aby bolo možné pracovať s Java entitami ako sémantickými dátami

3.1.2 Empire

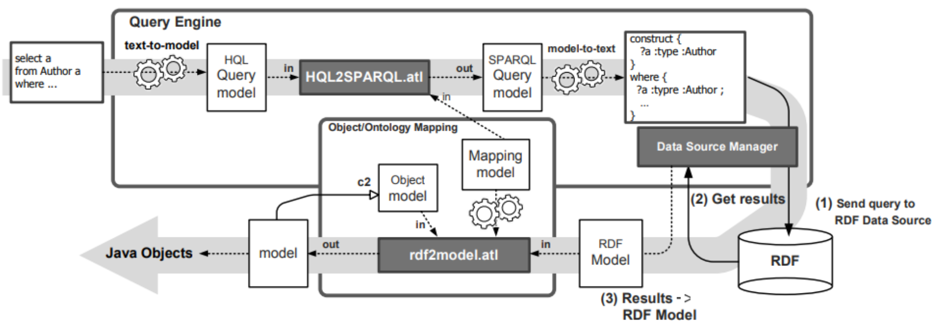
Ukážkou implementácie JPA nad RDF je framework, ktorý sa nazýva Empire. Využíva JPA implementáciu a doplnené anotácie tak, aby bolo možné pracovať s dátami vo forme RDF [Gro10]. Pomocou týchto anotácií, ktoré obsahujú IRI, popisujú entity a ich vlastnosti. Spolu s tým entity implementujú rozhranie, ktoré entite umožňuje mať RDF identifikátor.

Podobné riešenie implementuje objektovo ontologická mapovacia knižnica JOPA. Avšak na rozdiel od Empire, JOPA využíva OWL implementáciu.

3.1.3 JOPA

Java OWL Persistence API [LK15],[1] teda JOPA, je knižnica, ktorá implementuje podobné riešenie ako úprava JPA. Pomocou nej môžeme pracovať s dátami ako s Java objektami. Entitám a ich vlastnostiam sa pridávajú anotácie, ktoré obsahujú identifikačné IRI, namiesto bežných JPA anotácií `@Table` a `@Column`. Knižnici JOPA chýba objektový dotazovací jazyk pre sémantické dáta.

[1] `https://github.com/kbss-cvut/jopa`, navštívené 6.1.2020.



Obrázok 3.1: Zapojenie prekladača HQL2SPARQL [HBL09]

3.1.4 Prekladač

Riešením nedostatku knižnice JOPA by mohol byť prostredník medzi JPQL a SPARQL. Prekladač by nijak nezasahoval do persistančnej vrstvy, mapovania objektov, ani tvorby dotazov bežným spôsobom s výstupom dotazu v jazyku SPARQL. Tento prekladač by si zobral vytvorený JPQL dotaz ako vstup a na výstup by odoslal odpovedajúci SPARQL dotaz. Takto vytvorené rozhranie by sme mohli integrovať do projektu JOPA, bez väčších zásahov do jej architektúry.

Realizované riešenie prekladača medzi HQL a SPARQL. Podobné riešenie už bolo navrhnuté, spolu s vrstvou, ktorá nahrádza JPA a prekladačom dotazovacieho jazyka do SPARQL [HBL09]. Podobne ako JOPA má toto riešenie vlastné objektové OWL mapovanie modelu. Dotazy sa generujú v jazyku HQL a využíva sa k tomu mapovanie entít. Dotaz sa po vytvorení nevykoná okamžite. Predá sa prekladaču, ktorý ho prevedie na SPARQL a až potom sa spracuje.

Na obrázku 3.1 je zobrazené zapojenie prekladača. Vygenerovaný HQL dotaz sa predá ako vstup pre `HQL2SPARQL.alt`. Výstupom je potom SPARQL dotaz. Po vykonaní SPARQL dotazu je návratová hodnota RDF graf. Pomocou `rdf2model.alt` sa získaný RDF graf prevedie na Java objekt.

Jednou z nevýhod je, že ako vstup je potrebný HQL dotaz. JOPA potrebuje prekladať JPQL a teda nie je možné toto riešenie plne využiť. Aj keď by JOPA mohla využívať HQL, stále by integrácia HQL2SPARQL mohla predstavovať problém. Spôsob mapovania môže byť odlišný od toho, ktoré využíva JOPA.

Myšlienka prekladača, ktorý v JOPA prevedie JPQL na SPARQL, je však

zaujímavá a ako riešenie prijateľná. Rozhranie, ktoré spravuje a pracuje so SPARQL dotazmi už je v knižnici JOPA implementované.

3.2 Analýza rozdielov medzi SPARQL a JPQL

Pred návrhom implementácie je nutné vedieť, v čom spočívajú rozdiely medzi týmito dotazovacími jazykmi. Bude sa to hodiť pre zlepšenie efektivity návrhu. Zo zadania máme istú predstavu, s akými konštruktami budeme pri rekonštrukcii dotazu pracovať.

Niektoré konštrukty sa v jazykoch SPARQL a JPQL od seba veľmi nelíšia. Napríklad GROUP BY a ORDER BY. Iné sa v jazyku SPARQL píšu komplikovanejšie ako v JPQL. Napríklad WHERE spolu so svojimi operátormi NOT, AND, OR a LIKE.

3.2.1 Rozdiely v konštruktoch

WHERE. Jedným z rozdielov je, že SPARQL dotaz môže obsahovať WHERE, aj keď by sa v odpovedajúcom dotaze v JPQL nenachádzal.

Ak WHERE v JPQL obsahuje operátor =, v SPARQL dotaze funkciu prirovnania zabezpečuje samotná trojica. V opačnom prípade sa do bloku konštruktoru FILTER píše podmienka. FILTER sa v SPARQL komplikuje hlavne pri práci s operátormi ako sú OR, NOT.

Nasledujúce ukážky Listing 3.1, Listing 3.2, Listing 3.3, Listing 3.4 ukazujú rozdiely použitia dotazu s funkciou WHERE jej operátormi v jazyku JPQL a SPARQL.

AND. Práca s operátorom AND je v SPARQL intuitívna. AND má pri čítaní dotazu vždy prednosť pred OR. V SPARQL majú všetky trojice medzi sebou pomyselný AND a preto ho, na rozdiel od JPQL, do funkcie WHERE nepíšeme. Listing 3.1 obsahuje ukážku rozdielu medzi JPQL a SPARQL pre operátor AND.

Listing 3.1: Ukážka rozdielov medzi JPQL a SPARQL s operátorom AND

```
SELECT p FROM Person p WHERE p.age > :age AND p.height > :height

PREFIX ex: <http://www.example.org/>
SELECT ?x WHERE {
   ?x a ex:Person .
   ?x ex:age ?pAge .
   ?x ex:height ?pHeight .
   FILTER (?pAge > ?age && ?pHeight > ?height)
}
```

LIKE. Pokiaľ sa v JPQL vyskytne operátor LIKE, v SPARQL ho do bloku konštruktoru zapíšeme v tvare “regex(parameter, queryParameter/Hodnota)“. Listing 3.2 obsahuje ukážku rozdielu medzi JPQL a SPARQL pre operátor LIKE.

Listing 3.2: Ukážka rozdielov medzi JPQL a SPARQL s operátorom LIKE

```
SELECT p FROM Person p WHERE p.username LIKE :username

PREFIX ex: <http://www.example.org/>
SELECT ?x WHERE {
   ?x a ex:Person .
   ?x ex:username ?pUsername .
   FILTER (regex(?pUsername, ?username))
}
```

NOT. Je to unárny operátor s jednoduchým zápisom v JPQL. V SPARQL preň musíme mať vlastnú funkciu FILTER NOT EXISTS. Do bloku funkcie zapisujeme trojice. Môže tiež obsahovať vlastný FILTER. Listing 3.3 obsahuje ukážku rozdielu medzi JPQL a SPARQL pre operátor NOT.

Listing 3.3: Ukážka rozdielov medzi JPQL a SPARQL s operátorom NOT

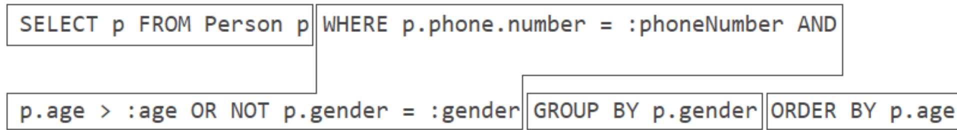
```
SELECT p FROM Person p WHERE p.phone.number = :phoneNumber AND NOT p.age >
  :age

PREFIX ex: <http://www.example.org/>
SELECT ?x WHERE {
   ?x a ex:Person .
   ?x ex:phone ?phone .
   ?phone ex:number ?phoneNumber .
   FILTER NOT EXISTS (
      ?x ex:age ?pAge .
      FILTER (?pAge > ?age)
   )
}
```

OR. V SPARQL nahrádza operátor OR slovo UNION. Nachádza sa medzi dvoma blokmi "{ }". Pokiaľ chceme prepísať OR z JPQL do SPARQL, tak do prvého bloku zapisujeme všetko od začiatku, prípadne od predošlého OR, či od začiatku zátvoriek. Do druhého bloku zapisujeme všetko, čo je za ním, až kým neprídeme na koniec podmienok, alebo ďalší OR, či koniec zátvoriek. Listing 3.4 obsahuje ukážku rozdielu medzi JPQL a SPARQL pre operátor OR.

Listing 3.4: Ukážka rozdielov medzi JPQL a SPARQL s operátorom OR

```
SELECT p FROM Person p WHERE p.phone.number = :phoneNumber OR
                  p.age > :age OR NOT p.gender = :gender

PREFIX ex: <http://www.example.org/>
SELECT ?x WHERE {
   ?x a ex:Person .
   {
      ?x ex:phone ?phone .
      ?phone ex:number ?phoneNumber .
   } UNION {
      ?x ex:age ?pAge .
      FILTER (?pAge > ?age)
   } UNION {
      FILTER NOT EXISTS (
         ?x ex:gender ?gender .
      )
   }
}
```

```
SELECT p FROM Person p WHERE p.phone.number = :phoneNumber AND

p.age > :age OR NOT p.gender = :gender GROUP BY p.gender ORDER BY p.age
```



Obrázok 3.2: Vzorový dotaz a jeho rozdelenie podľa kluazúl

3.3 Návrh riešenia

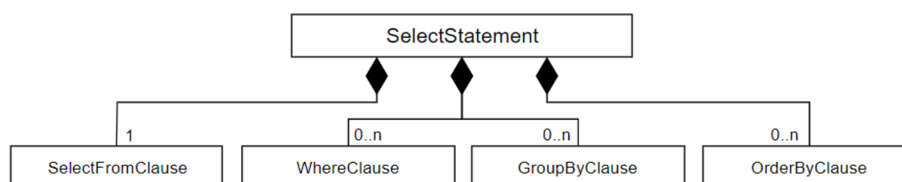
Pri analýze už existujúcich riešení som našiel niekoľko možností, ako by som mohol k problému pristupovať.

Využiť samotnú persistence vrstvu JPA obohatenú o vlastnú implementáciu. Tým by sa mala docieliť schopnosť systému pracovať s dátami uloženými vo forme RDF rovnako, ako s relačnými. Išlo by však o tvorbu novej knižnice, ktorá by fungovala veľmi podobne ako projekt katedry počítačov JOPA. Takéto riešenie by nebolo vyhovujúce zo strany efektivity a bolo by zbytočne náročné a vlastne aj duplicitné.

Na druhej strane je lepšie využiť spomínanú, už existujúcu knižnicu JOPA. Zaintegrovať do nej rozhranie, pomocou ktorého by sme mohli prekladať vygenerované JPQL dotazy do jazyka SPARQL. Rozhranie, ktoré spracuje a vykonáva SPARQL dotazy už v knižnici existuje, preto nebude doplnenie takéhoto rozhrania veľmi zložité. Docielime tým, že nahradíme aktuálny spôsob generovania SPARQL dotazov.

3.3.1 Gramatika

Pri návrhu gramatiky je podstatné zanalyzovať, čo bude vlastne vstup pre prekladač, ktorý budem implementovať. Je potrebné sa zamerať na to, čo môže vstup obsahovať spolu s tým, v akom tvare bude tento obsah. Musím brať do úvahy aj skutočnosť, že neimplementujem všetky možné tvary a konštruktory jazyku JPQL, či SPARQL. Hlavne sa zameriavam na selekciu, filtrovanie a elementárny join.



Obrázok 3.3: Šablóna SelectStatementTemplate

Napríklad implementácia SPARQL konštruktoru CONSTRUCT nie je zahrnutá ani do návrhu, ani do prototypu môjho riešenia. Preto by gramatika mala byť navrhnutá tak, aby ju bolo možné upraviť, či rozšíriť bez zbytočných komplikácií.

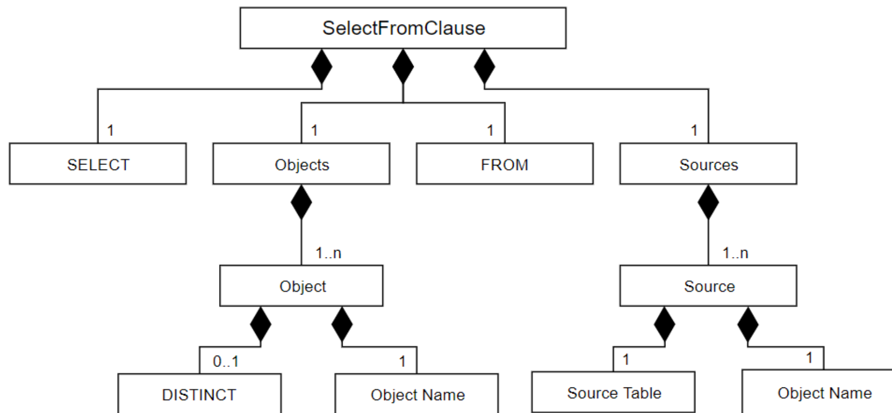
3.3.2 Vzorový dotaz

Pre vytvorenie najpresnejšieho návrhu je nutné mať predstavu, čo bude vstupom. Preto som si vytvoril dotaz, ktorý by mal byť vzorom vstupu. Jeho obsahom budú všetky potrebné konštruktory, tak aby som podľa neho mohol urobiť šablónu vzorového vstupného dotazu. Šablónu si následne rozdelím na menšie šablóny, ktoré nakoniec využijem pri tvorbe gramatiky.

Vzorový dotaz (3.2) obsahuje všetky potrebné konštrukty, aby som ho mohol rozdeliť na menšie časti a vytvoriť tak z neho šablónu.

Po rozdelení je vidieť, že dotaz obsahuje 4 hlavné časti, ktoré môžeme zovšeobecniť a podľa nich vytvoriť vzorovú šablónu pre SelectStatement (3.3), ktorá bude obsahovať:

- SelectFromClause
- WhereClause
- GroupByClause
- OrderByClause



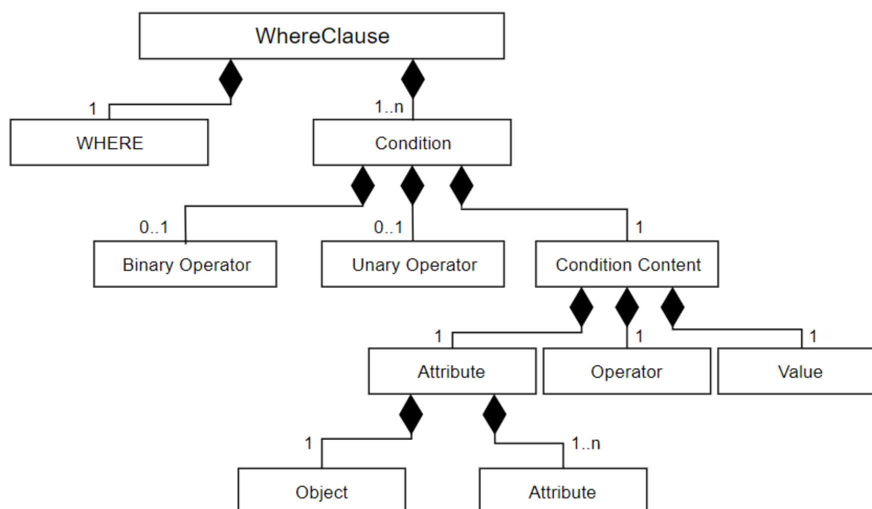
Obrázok 3.4: Šablóna SelectFromClause

SelectFromClause. Klauzula SelectFrom (3.4) je jediná vo výraze, ktorá je povinná a existuje v ňom práve jeden krát. Skladá sa z kľúčových slov SELECT a FROM. Obsahuje označenie cieľových dát, ktoré chceme získať. Toto označenie je v podobe celého objektu, alebo vybraných vlastností. Je zakončená definovaním, odkiaľ chceme cieľové dáta získať.

Pri označení cieľových dát sa môže vyskytovať slovo DISTINCT, čo znamená, že medzi vybranými dátami nebudú duplicity. Celú klauzulu si môžeme teda rozdeliť na 4 časti. SELECT, množina objektov, FROM, množina definícií zdrojov objektov.

Obrázok 3.4 je grafické zobrazenie SelectFromClause šablóny .

WhereClause. Klauzula bude na začiatku obsahovať kľúčové slovo WHERE, za ním sa bude nachádzať 1..n podmienok filtrovania dát. Podmienka sa bude skladať z vlastnosti, operátora a hodnoty. Každá podmienka môže mať pred sebou unárny operátor NOT. Pokiaľ sa v klauzule nachádzajú aspoň dve podmienky, oddeľujeme ich binárnym operátorom OR, alebo AND. Vlastnosť môže obsahovať viac reťazených fragmentov. Pokiaľ je ich viac ako 1, tak filtrujeme pomocou asociovaného objektu a teda sa jedná o elementárny JOIN (Listing 3.5).



Obrázok 3.5: Šablóna WhereClause

Listing 3.5: Filtrovanie za pomoci vlastnosti asociovaného objektu

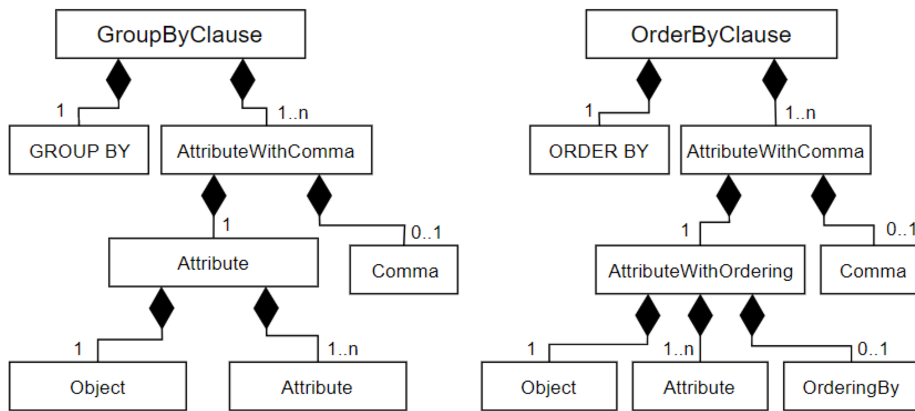
```
SELECT p FROM Person p WHERE p.phone.number = :phoneNumber

PREFIX ex: <http://www.example.org/>
SELECT ?x WHERE {
   ?x a ex:Person .
   ?x ex:phone ?phone .
   ?phone ex:number ?phoneNumber .
}
```

Obrázok 3.5 zobrazuje prvky, ktoré WhereClause obsahuje.

GroupByClause. GroupBy klauzula začína slovným spojením GROUP BY. Obsahuje minimálne jednu vlastnosť, podľa ktorej chceme získané dáta rozdeliť do skupiny. Vlastnosti sú od seba oddeľované čiarkou. Vlastnosť, podľa ktorej chceme zoskupovať výsledky, musí byť súčasťou SPARQL dotazu.

Šablónu GroupByClause znázorňuje obrázok 3.6.

OrderByClause. Posledná klauzula v dotaze je klauzula pre radenie výsledkov. Nachádza sa v nej slovné spojenie ORDER BY a zoznam vlastností, podľa ktorých budeme zoraďovať. Mala by byť zapísaná aspoň jedna vlastnosť. Ak sa ich v klauzule nachádza viac, sú oddelené čiarkou. Každá vlastnosť



Obrázok 3.6: Šablóna GroupByClause spolu s OrderByClause

môže byť zakončená slovom, ktoré určuje spôsob radenia. ASC pre vzostupné, alebo DESC pre zostupné radenie. Pokiaľ slovo nie je uvedené, automaticky je radenie podľa ASC. Vlastnosť, podľa ktorej chceme zoraďovať výsledky, musí byť súčasťou SPARQL dotazu.

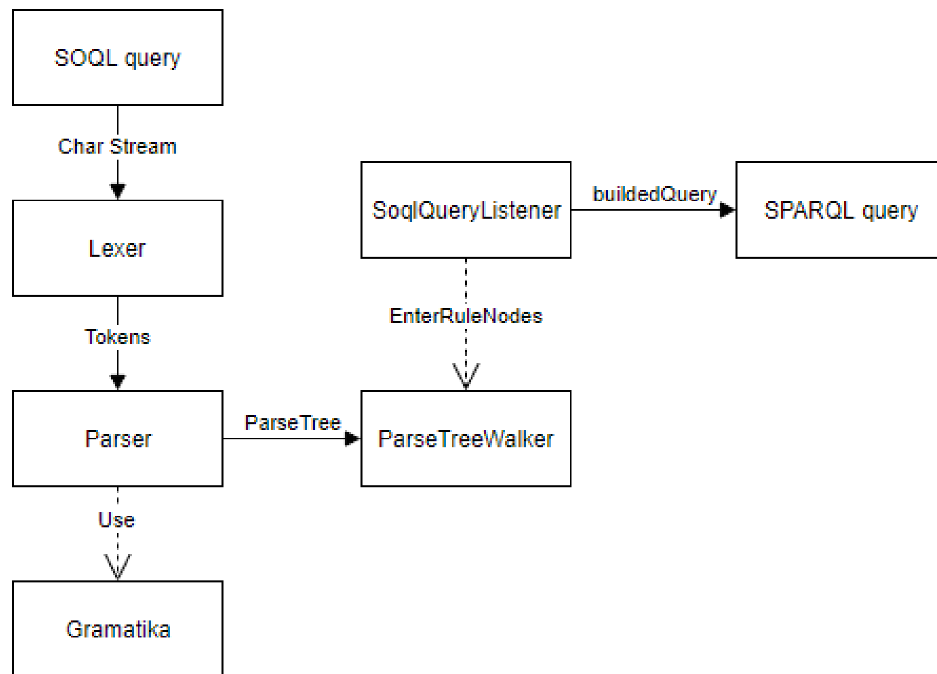
Šablónu OrderByClause môžeme vidieť na obrázku 3.6.

Po rozdelení dotazu je vidieť, čo môžeme považovať za pravidlo v gramatike. Všeobecnejšie pravidlá budeme postupne deliť na detailnejšie. Ani v gramatike sa veľmi nechceme opakovať, preto je dobré písať pravidlá tak, aby pokryli čo najväčšiu časť dotazu. Okrem toho je tiež vidieť, že niektoré štruktúry dotazu sa budú opakovať, napríklad atribúty. V gramatike ich môžeme preto definovať pod jedným pravidlom, ktoré využijeme viac krát.

3.3.3 Návrh implementačnej časti

Celá moja implementácia bude obsiahnutá v balíčku Semantic Object Query Language, skrátene SOQL.

Ďalším krokom je navrhnúť, akým spôsobom využijem vytvorenú gramatiku s pravidlami. Využije sa na to nástroj ANTLR, ktorý podľa navrhnutej gramatiky vygeneruje triedy Lexer, Parser. Lexer rozdelí vstupný reťazec na tokeny, podľa ktorých parser vytvorí AST. Tieto triedy už majú implementovaný kód pre spracovanie štruktúrovaného textu podľa gramatiky, teda v našom prípade SOQL dotazu. Spolu s týmito triedami sa generuje aj rozhranie pre Listener, ktoré rozširuje rozhranie ParseTreeListener. Ilustráciu procesu



Obrázok 3.7: Ilustrácia procesu prekladu dotazu

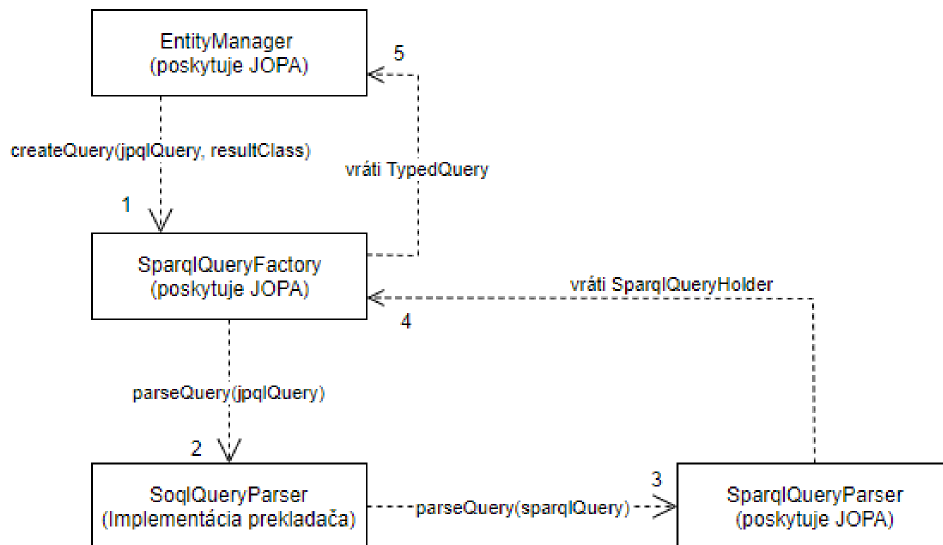
prekladu môžeme vidieť na obrázku 3.7.

Súčasťou SOQL bude trieda SoqlQueryListner, ktorá bude mať hlavnú úlohu pri rekonštrukcii SPARQL dotazu. Okrem toho bude obsahovať dalšie triedy, ktoré budú reprezentovať parametre a vlastnosti spracovaného SOQL dotazu.

SoqlQueryListener. Trieda SoqlQueryListener je implementáciou vygenerovaného rozhrania pre Listener. Obsahuje metódy typu void, ktoré reagujú na vstup a výstup z pravidla objektom triedy ParseTreeWalker pri priechode AST.

Po analýze rozdielov medzi SPARQL a SOQL som zistil, že by nebolo veľmi dobré riešenie tvoriť SPARQL dotaz počas návštev pravidiel pri priechode AST. Je lepšie si vyzbierať potrebné dáta a samotnú rekonštrukciu dotazu v jazyku SPARQL vykonať až na konci. Výsledný dotaz sa bude dať získať pomocou getteru.

Mapovanie vlastností a objektov zo SOQL do SPARQL. Na rozdiel od SOQL, ktoré pracuje s názvami entít a ich vlastností pri mapovaní objektov



Obrázok 3.8: Diagram ilustrujúci intergáciu prekladača do knižnice JOPA

v databáze, SPARQL k tomu potrebuje ich IRI.

Všetky tieto informácie o entitách a ich vlastnostiach si drží objekt metamodel. Referencia na tento objekt je predaná triede SoqlQueryListener, aby sa dalo s metamodelom pracovať v dobe prekladu.

Každá entita má tieto IRI zapísané pomocou anotácií. Entitu označujeme anotáciou @OWLClass(iri). Vlastnosti popisujeme pomocou anotácií @OWL-DataProperty(iri). Asociované objekty zase pomocou anotácie @OWLObjectProperty(iri).

Aby bolo možné vytvoriť rekonštrukciu funkčného SPARQL dotazu, odpovedajúceho jeho variante v SOQL, potrebujeme získať IRI. Tieto IRI si vyhľadáme v spomínanom objekte metamodel pomocou ich názvu. Spravíme tak pre každú entitu, asociovaný objekt, či vlastnosť, ktorú nájdeme v SOQL dotaze.

3.3.4 Návrh integrácie do JOPA

Vďaka tomu, že implementácia prekladača nezasahuje do už implementovaného kódu knižnice, je integrácia pomerne jednoduchá. Využijeme k tomu už existujúce rozhrania knižnice SparqlQueryFactory a SparqlQueryParser.

Vstupom bude SOQL dotaz, ktorý získam od funkcie, ktorú implementuje SparqlQueryFactory. Dotaz sa pomocou mojej implementácie preloží na SPARQL. Ten následne predám SparqlQueryParseru, ktorý dotaz spracuje. Konečným výstupom, ktorý vrátim do SparqlQueryFactory, bude spracovaný SPARQL dotaz, získaný od SparqlQueryParseru. Tento proces je zobrazený na obrázku 3.8.

Kapitola 4

Implementácia prototypu

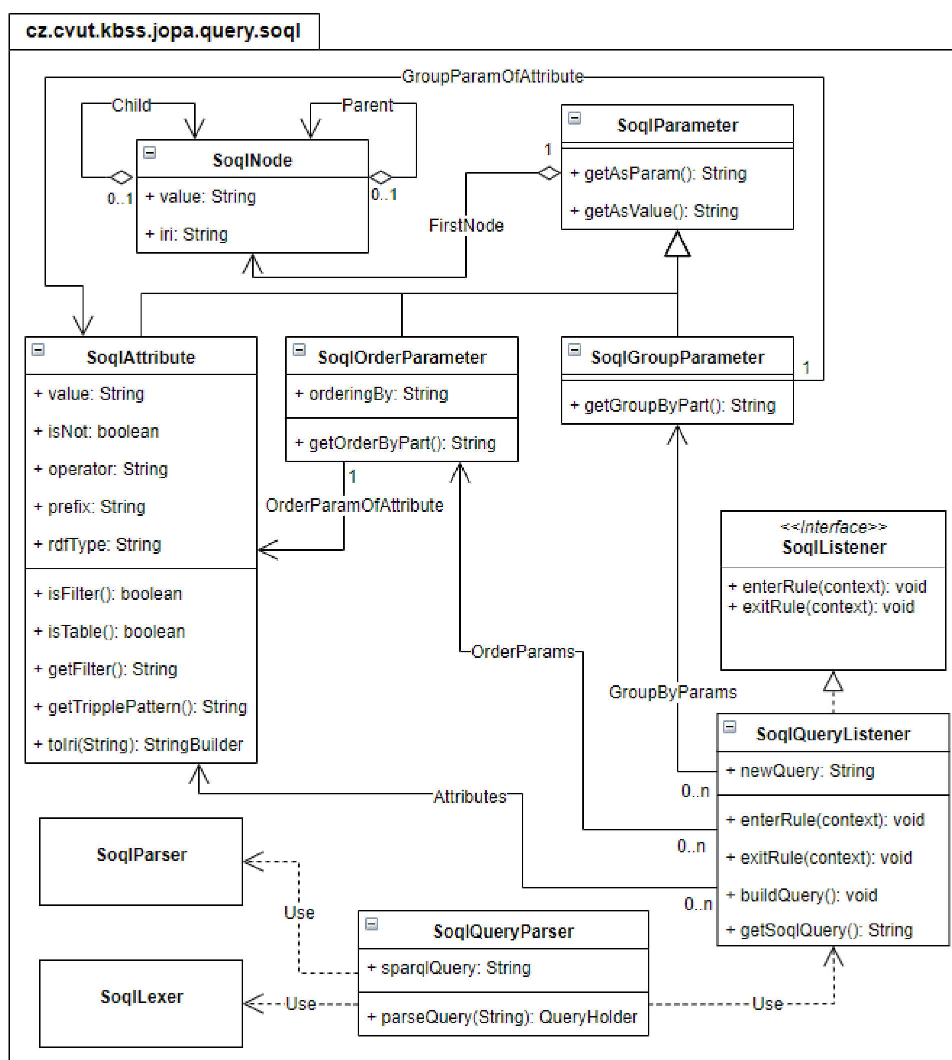
4.1 Tvorba gramatiky

Samotná tvorba prototypu začína vytvorením gramatiky pre program ANTLR, ktorý následne vygeneruje potrebné triedy pre parsovanie vstupného dotazu. Najprv bolo potrebné vstupný dotaz zanalyzovať a vytvoriť, čo najpodrobnejšie, šablóny AST. Vďaka tomu bolo možné efektívne vytvoriť gramatiku. Z analýzy bolo možné vidieť kde a ako sa niektoré prvky dotazu opakujú. Tým sa predišlo duplicitným pravidlám, ktoré by gramatiku a následnú implementáciu komplikovali.

4.2 Implementácia návrhu

Po vytvorení gramatiky boli vygenerované potrebné triedy a rozhrania programom ANTLR. Triedy Lexer a Parser nebolo nutné ďalej modifikovať a stačilo ich len použiť. Bolo však vhodné vyhnúť sa použitiu vygenerovanej triedy Listeneru, nakoľko bude obsahovať vlastnú implementáciu. Pri každej úprave gramatiky by sa táto trieda vygenerovala znova a hrozila by strata vlastnej implementácie. Preto som si vytvoril vlastnú triedu, ktorú som si nazval SoqlQueryListener.

Prototyp obsahuje triedy SoqlNode, SoqlParameter, SoqlAttribute, So-



Obrázok 4.1: Class Diagram rozhrania prekladača

qlOrderParameter, SoqlGroupParameter. Objekty týchto tried reprezentujú prvky dotazu s potrebnými informáciami. Sú implementované tak, aby sa sami starali o to, kde a čo vypíšu v rekonštruovanom dotaze.

SoqlNode. Trieda, ktorá reprezentuje fragment atribútu alebo parametra v dotaze. Drží si pointer na predošlý a nasledujúci objekt SoqlNode. Vo forme Stringu má zapísaný názov fragmentu spolu s identifikátorom IRI, ktorý identifikuje vzťah fragmentu k vlastníkovi. Akýkoľvek atribút v dotaze sa zapisuje ako reťaz objektov SoqlNode.

SoqlParameter. Základná trieda, ktorá obsahuje spoločné funkcie a vlastnosti všetkých atribútov a parametrov v dotaze. Trieda si drží pointer na prvý fragment v podobe objektu SoqlNode. Obsahuje tiež funkciu, ktorá všetky zreťazené SoqlNode prevedie na String v tvare "?node1Node2...Noden".

SoqlAttribute. Trieda, ktorá rozširuje SoqlParameter. Objekty tejto triedy reprezentujú akýkoľvek atribút v SelectClause a WhereClause. Je rozšírený o informácie o porovnávacom operátore a hodnoty, s ktorou porovnáva. Obsahuje aj informáciu, ktorá určuje, či bol pred ním unárny operátor. Spolu s tým obsahuje aj funkcie, ktoré vrátia atribút v tvare trojice. Dokáže vrátiť reťazec pre FILTER blok aj zistiť, či je nutná jeho prítomnosť v tomto bloku.

SoqlOrderParameter. Rozširuje SoqlParameter. Je určený pre atribúty v OrderByClause. Drží si pointer na odpovedajúci objekt SoqlAttribute. Má zapísanú informáciu, akým spôsobom bude zoraďovať. Vracia korektný tvar daného parametru pre ORDER BY konštruktor.

SoqlGroupParameter. Rozširuje SoqlParameter. Je určený pre atribúty v GroupByClause. Má za úlohu držať si pointer na odpovedajúci objekt SoqlAttribute. Vracia korektný tvar daného parametru pre GROUP BY konštruktor.

SoqlQueryListener. Trieda SoqlQueryListener tieto objekty zhromažďuje počas návštev metód pravidiel. Medzi objektami je intuitívne operátor AND. Pokiaľ sa našiel pri niektorom objekte operátor OR, uložil sa jeho ukazateľ. Výsledný zoznam sa podľa týchto ukazateľov rozdelí, aby sa každý objekt nachádzal v správnom bloku pri rekonštrukcii.

SoqlQueryParser. Úlohou triedy SoqlQueryParser je preložiť vstupný SOQL dotaz na SPARQL. Metóda parseQuery, ktorú trieda implementuje, obsahuje celý proces prekladu. Aby sa dodržali zavedené normy pre SPARQL dotaz v JOPA, metóda na konci vytvára objekt SparqlQueryParser a predáva mu zrekonštruovaný dotaz. Zároveň si zrekonštruovaný dotaz drží v inštančnej premennej sparqlQuery. Návratovou hodnotou je požadovaný objekt SparqlQueryHolder, ktorý sa získa od objektu SparqlQueryParser. Implementovaná metóda parseQuery sa využíva v triede SparqlQueryFactory, ktorá získaný objekt SparqlQueryHolder odošle na vykonanie do databázy.

Kapitola 5

Testovanie prototypu

Môj návrh riešenia spolu s jeho prototypom, ktorým sa v tejto práci zaoberám, je v podstate modul. Preto pre účely otestovania návrhu aj samotného prototypu je potrebná knižnica a keďže môj návrh a prototyp bol primárne zameraný na rozšírenie funkcionality knižnice JOPA, prostredie tejto knižnice využijem aj na testovanie.

Môj prototyp obsahuje vlastne len jeden prípad použitia a to preklad medzi SOQL a SPARQL. Nie je ani samostatná aplikácia, preto testovanie priechodu nepripadá do úvahy. Jediný spôsob, ako otestovať moju implementáciu je pomocou Unit Testov a Integračných Testov.

5.1 Unit Testy

Hlavnou úlohou Unit Testov bude otestovať štruktúru a tvar preloženého dotazu. Testy budú fungovať bez pripojenia do databázy a preložené dotazy nebudú v rámci testov nijak vykonané, ani sa nebude očakávať výsledok v podobe dát. V testoch sa bude očakávať, že sa preložené dotazy budú zhodovať s poskytnutým vzorom. Pri Unit testovaní nebude prekladač obsahovať referenciu na objekt metamodelu. Z toho dôvodu nebudú preložené dotazy obsahovať vlastné reálne IRI, ale len jednu defaultnú IRI.

Pomocou týchto testov je potrebné otestovať čo najviac rozličných tvarov,

ktoré môže obsahovať dotaz:

- trojice pre požadovaný objekt, jeho atribúty, k nemu asociované objekty a prípadne ich atribúty, či ďalšie asociované objekty
- správne tvary a rôzne variácie unárnych a logických operátorov NOT, AND a OR v klauzule WHERE
- správne umiestnenie konštruktu FILTER a jeho obsah, vrátane viacnásobného filtrovania, či funkcie regex, ktorá nahrádza operátor LIKE
- korektnosť obsahu klauzúl ORDER BY a GROUP BY v prípade, že sa vlastnosť nachádza v klauzule WHERE, aj v prípade, že sa v nej nenachádza a je nutné, aby trojica vlastnosti bola obsiahnutá v tele dotazu
- korektný tvar funkcie COUNT

Celkový počet Unit testov je 48. Ich cieľom bolo pokryť čo najväčší rozsah vymenovaných variácií tvaru vstupného dotazu.

5.2 Integračné Testy

Po otestovaní preložených dotazov, ich štruktúr a tvarov, je potrebné otestovať aj ich reálne použitie pri získavaní dát. To je úlohou Integračných Testov. V tomto prípade už bude zabezpečené pripojenie do databázy určenej pre testy. Bude nastavené prostredie s testovacími entitami a prekladaču bude poskytnutá referencia na objekt metamodelu. Vďaka tomu sa pri preklade použije reálne IRI namiesto defaultnej. Tvar dotazu tu už nie je nutné testovať, preto sa testy zameriavajú hlavne na správny výsledok vykonania dotazu. Parametre sú nahradené URI objektov, alebo literálmi.

Testuje sa zoznam získaných objektov pomocou dotazu. Kontroluje sa typ objektov v liste, počet získaných objektov, prípadne či je list prázdny. V prípade ORDER BY dotazu sa kontroluje aj poradie objektov.

Ukážku integračných testov môžeme vidieť v listingu 5.1 a 5.2.

Listing 5.1: Ukážka integračného testu pre získanie všetkých objektov typu OWLClassA

```
public void testFindByTransitiveAttributeValue() {
    final OWLClassD expected = Generators.getRandomItem(QueryTestEnvironment.getData(
        OWLClassD.class));
    final OWLClassD result = getEntityManager()
        .createQuery("SELECT d FROM OWLClassD d WHERE
            d.owlClassA.stringAttribute = :stringAtt", OWLClassD.class)
        .setParameter("stringAtt", expected.getOwlClassA().getStringAttribute(), "en")
            .getSingleResult();
    assertEquals(expected.getUri(), result.getUri());
    assertEquals(expected.getOwlClassA().getUri(), result.getOwlClassA().getUri());
}
```

Listing 5.2: Ukážka integračného testu pre získanie všetkých objektov typu OWLClassT zoradených podľa vlastnosti intAttribute

```
public void testOrderBy() {
    final List<OWLClassT> expected = QueryTestEnvironment.getData(OWLClassT.class);
    expected.sort(Comparator.comparing(OWLClassT::getIntAttribute));
    final List<OWLClassT> result = getEntityManager()
        .createQuery("SELECT t FROM OWLClassT t ORDER BY t.intAttribute", OWLClassT.
            class).getResultList();
    assertEquals(expected.size(), result.size());
    for (OWLClassT t : result) {
        assertTrue(expected.stream().anyMatch(tt -> tt.getUri().equals(t.getUri())));
    }
}
```

Kapitola 6

Záver

6.1 Dosiahnuté ciele projektu

- Analyzovať existujúce prototypy objektového dotazovacieho jazyka pre sémantické dáta, zhodnotiť ich silné a slabé stránky a vhodnosť ich integrácie do knižnice JOPA.

  Celá analýza je popísaná v sekcii 3.1. Nie je veľa riešení, ktoré by sa zaoberali stanoveným problémom. Niektoré sú súčasťou implementácie knižníc podobných JOPE. Našlo sa aj jedno konkrétne riešenie, ktoré bolo vytvorené ako prekladač medzi jazykom HQL a SPARQL, avšak integrovať ho do knižnice JOPA by nebolo vhodné. Toto riešenie sa stalo inšpiráciou pre návrh a realizáciu prototypu vlastného prekladača.

- Navrhnúť podmnožinu objektového dotazovacieho jazyka pre sémantické dáta, ktorá sa bude prekladať do dotazovacieho jazyku SPARQL. Podmnožina musí podporovať nasledujúce konštrukty: selekcia, join, filtrovanie a vybrané agregačné funkcie. Inspiráciou tohto jazyku bude JPQL.

  Návrh môjho riešenia je modul, ktorý prekladá dotazy písané v jazyku JPQL do jazyka SPARQL. Preklad podporuje stanovené konštrukty. Medzi vybrané agregačné funkcie patrí count.

- Vytvoriť prototyp implementácie navrhnutého dotazovacieho jazyka a zaintegrovať tento prototyp do knižnice JOPA.

  Implementovaný prototyp je balíček SOQL (Semantic Object Query Language), ktorý je popísaný v sekcii 4.2. Tento balíček je zaintegrovaný

do knižnice JOPA, čo je ilustrované na obrázku 3.8.

- Overiť správnosť implementácie môjho vytvoreného jazyka porovnaním výsledkov typických dotazov napísaných v jazyku SPARQL.

  Správnosť implementácie bola overená pomocou metód Unit Testov a Integračných testov. Úlohou Unit testov bolo overiť správnosť tvaru preloženého dotazu. Integračné testy otestovali správnosť integrácie do knižnice JOPA. Obe metódy sú popísané v kapitole 5.

Navrhnutý prekladač vychádza z jazyka JPQL, preto je možné ho označiť za podmnožinu objektového dotazovacieho jazyka pre sémantické dáta. Prekladač podporuje stanovené konštrukty. Zároveň boli splnené ciele projektu, preto ho môžeme pokladať za úspešný.

6.2 Budúci vývoj

Implementovaný prototyp je len základ sémantického objektového dotazovacieho jazyka. V balíčku implementácie sa nachádza zoznam funkcií, ktorých implementácia by mohla byť predpokladaná, avšak nie je zahrnutá do riešenia. Jedným z cieľov budúceho vývoja by malo byť doplnenie týchto funkcií a oprava prípadných bugov.

Moje riešenie vo finálnej verzii podporuje len konkrétne konštrukty stanovené v zadaní práce. Ďalším z cieľov budúceho vývoja bude rozšíriť preklad aj na ďalšie konštrukty, funkcie a aspekty, ktoré jazyk SPARQL poskytuje.

Kapitola 7

Inštalácia

Prototyp je súčasťou knižnice JOPA. Táto knižnica sa nachádza v CD prílohe mojej práce, alebo je možné ju stiahnuť z GitHubu na adrese `https://github.com/kbss-cvut/jopa`, prípadne `https://github.com/krupama3/jopa`. Pri použití GitHubu je možné naklonovať repozitár.

Ako súčasť knižnice, je možné otestovať funkcie prototypu pomocou testov spomínaných v kapitole 5. Odporúčané prostredie pre spustenie testov je IntelliJ IDEA.

Unit testy sa nachádzajú v `/jopa/jopa-impl/src/main/java/cz.cvut.kbss.jopa/query/soql/SoqlQueryParserTest.java`.

Integračné testy sa nachádzajú v `/jopa/jopa-integration-tests/src/main/java/cz.cvut.kbss.jopa.test/query/runner/SoqlRunner.java`.

Integračné testy je možné spustiť pomocou dvoch tried.

- Triple Store Apache Jena: `/jopa/jopa-integration-tests-jena/src/test/java/cz.cvut.kbss.jopa.test/query.jena/SoqlTest.java`
- Triple Store Sesame: `/jopa/jopa-integration-tests-sesame/src/test/java/cz.cvut.kbss.jopa.test/query.sesame/SoqlTest.java`

Literatúra

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, *Design Patterns: Elements of Reusable Object-oriented Software*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[Gro10] Michael Grove, *Empire: RDF & SPARQL Meet JPA*, Dataversity (2010), `http://www.dataversity.net/empire-rdf-sparql-meet-jpa/`, accessed 2019-12-29.

[HBL09] Guillaume Hillairet, Frédéric Bertrand, and Jean Yves Lafaye, *Rewriting queries by means of model transformations from sparql to oql and vice-versa*, Theory and Practice of Model Transformations (Berlin, Heidelberg) (Richard F. Paige, ed.), Springer Berlin Heidelberg, 2009, pp. 116–131.

[HS13] Steve Harris and Andy Seaborne, *SPARQL 1.1 Query Language*, W3C recommendation, W3C, 2013, `https://www.w3.org/TR/sparql11-query/`, accessed 2019-12-16.

[JCP09] JCP, *JSR 317: Java^{TM} Persistence API, Version 2.0*, Tech. report, Java Community Process, 2009.

[LK15] Martin Ledvinka. and Petr Křemen., *Jopa: Accessing ontologies in an object-oriented way*, Proceedings of the 17th International Conference on Enterprise Information Systems - Volume 1: ICEIS,, INSTICC, SciTePress, 2015, pp. 212–221.

[MM09] M.Keith and M.Schnicariol, *Pro jpa 2: Mastering the java persistence api*, New York: Apress, 2009.

[PAR12] T. PARR, *The definitive antlr 4 reference.*, The Pragmatic Bookshelf, 2012.

[RC14] Markus Lanthaler Richard Cyganiak, David Wood, *RDF 1.1 Concepts and Abstract Syntax*, W3C recommendation, W3C, 2014, `https://www.w3.org/TR/rdf11-concepts/`, accessed 2019-12-16.

[Sau18] Cruce Saunders, *What Is the Semantic Web?*, `https://simplea.com/Articles/what-is-the-semantic-web`, accessed 2019-12-29.

[SL17] Claus Stadler and Jens Lehmann, *Jpa criteria queries over rdf data*, Joint Proceedings of the 2nd RDF Stream Processing (RSP 2017) and the Querying the Web of Data (QuWeDa 2017) Workshops co-located with 14th ESWC 2017 (ESWC 2017), Portoroz, Slovenia, May 28th - to - 29th, 2017. (Aachen) (Jean-Paul Calbimonte, Minh Dao-Tran, Daniele Dell'Aglio, Danh Le Phuoc, Muhammad Saleem, Ricardo Usbeck, Ruben Verborgh, and Axel-Cyrille Ngonga Ngomo, eds.), CEUR-WS.org, 2017, pp. 55–62.

[Tom17] Gabriele Tomassetti, *The ANTLR Mega Tutorial*, `https://tomassetti.me/antlr-mega-tutorial/`, accessed 2019-12-29.

Dodatok B

Zoznam skratiek

ANTLR Another Tool for Language Recognition. 10

AST Abstract Syntax Tree. 10

HQL Hibernate Query Language. 15

IRI Internationalized Resource Identifier. 4

JOPA Java OWL Persistence API. 1

JPA Java Persistence API. 7

JPQL Java Persistence Query Language. 1

OWL Ontology Web Language. 14

RDF Resource Description Framework. 1

SOQL Semantic Object Query Language. 23

SPARQL SPARQL Protocol and RDF Query Language. 2

URI Uniform Resource Identifier. 32

WWW World Wide Web. 1

Dodatok C

Obsah CD

```
/
├── BP ........................................... Dokument BP
└── jopa ......................................... Projekt JOPA
    ├── jopa-impl
    │   ├── main/../query/soql .................. Zložka implementácie
    │   └── test/../query/soql .................. Zložka s unit testami
    ├── jopa-integration-tests
    │   └── main/../query/runner/SoqlRunner.java ... Integračné testy
    ├── jopa-integration-tests-jena
    │   └── test/../query.jena/SoqlTest.java ....... Integračné testy
    └── jopa-integration-tests-sesame
        └── test/../query.sesame/SoqlTest.java ..... Integračné testy
```

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

ČVUT
ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Krupa** Jméno: **Martin** Osobní číslo: **434972**

Fakulta/ústav: **Fakulta elektrotechnická**

Zadávající katedra/ústav: **Katedra počítačů**

Studijní program: **Softwarové inženýrství a technologie**

II. ÚDAJE K BAKALÁŘSKÉ PRÁCI

Název bakalářské práce:

Sémantický objektový dotazovací jazyk

Název bakalářské práce anglicky:

Semantic Object Query Language

Pokyny pro vypracování:

1. Analyzujte existující prototypy objektového dotazovacího jazyka pro sémantická data, zhodnoťte jejich silné a slabé stránky a vhodnost pro integraci do objektově-ontologické mapovací knihovny JOPA.
2. Navrhněte podmnožinu objektového dotazovacího jazyka pro sémantická data, která se bude překládat do dotazovacího jazyka SPARQL.
Tato podmnožina musí podporovat alespoň následující konstrukty: selekce, join, filtrování, vybrané agregační funkce. Inspirujte se jazyky pro relační databáze - JPQL či HQL.
3. Vytvořte prototyp implementace navrženého dotazovacího jazyka a zaintegrujte tento prototyp do knihovny JOPA.
4. Ověřte správnost implementace porovnáním výsledků typických dotazů napsaných v jazyce SPARQL a ve vámi vytvořeném jazyce.

Seznam doporučené literatury:

[1] M. Keith and M. Schincariol, Pro JPA 2: Mastering the Java™ Persistence API, Apress, 2009
[2] S. Harris and A. Seaborne, SPARQL 1.1 Query Language, W3C recommendation, W3C, 2013
[3] R. Cyganiak, D. Wood, and M. Lanthaler, RDF 1.1 Concepts andAbstract Syntax, W3C recommendation, W3C, 2014

Jméno a pracoviště vedoucí(ho) bakalářské práce:

Ing. Martin Ledvinka, skupina znalostních softwarových systémů FEL

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) bakalářské práce:

Datum zadání bakalářské práce: **02.10.2019** Termín odevzdání bakalářské práce: **07.01.2020**

Platnost zadání bakalářské práce: **30.09.2021**

Ing. Martin Ledvinka
podpis vedoucí(ho) práce

podpis vedoucí(ho) ústavu/katedry

prof. Mgr. Petr Páta, Ph.D.
podpis děkana(ky)

III. PŘEVZETÍ ZADÁNÍ

Student bere na vědomí, že je povinen vypracovat bakalářskou práci samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací. Seznam použité literatury, jiných pramenů a jmen konzultantů je třeba uvést v bakalářské práci.

.

Datum převzetí zadání Podpis studenta

© ČVUT v Praze, Design: ČVUT v Praze, VIC