Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Alen Roguljić

NAPREDNI GRADNIKI POIZVEDOVALNEGA JEZIKA SPARQL

diplomsko delo

univerzitetni študijski program prve stopnje
računalništvo in informatika

Mentor: doc. dr. Dejan Lavbič

Ljubljana, 2014

Besedilo je oblikovano z urejevalnikom besedil LaTeX.

Univerza *v Ljubljani*
Fakulteta *za računalništvo in informatiko*

Tržaška 25
1001 Ljubljana, Slovenija
telefon: 01 476 84 11
01 476 83 87
faks: 01 426 46 47
01 476 87 11
www.fri.uni-lj.si
e-mail: dekanat@fri.uni-lj.si

Št. naloge: 00121 / 2013
Datum: 11.4.2013

Univerza v Ljubljani, Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Kandidat: **ALEN ROGULJIĆ**

Naslov: **NAPREDNI GRADNIKI POIZVEDOVALNEGA JEZIKA SPARQL**
ADVANCED ASPECTS OF SPARQL QUERY LANGUAGE

Vrsta naloge: Diplomsko delo univerzitetnega študija prve stopnje

Tematika naloge:

RDF je jezik za zajem usmerjenih in označenih podatkov v obliki grafa, ki je namenjen predvsem predstavitvi podatkov na svetovnem spletu. Za podporo iskanju po teh podatkih je na voljo poizvedovalni jezik SPARQL, ki ga je tako možno uporabljati tudi za izvajanje poizvedb nad različnimi in heterogenimi podatkovnimi viri, kjer so podatki posredno ali neposredno shranjeni v RDF obliki. SPARQL podpira poizvedovanje po zahtevanih in opcijskih vzorcih grafov kot tudi njihovo konjunkcijo in disjunkcijo. Prav tako ima zelo dobro podpro za testiranje vrednosti in omejitev poizvedb glede na vir RDF grafa. V okviru diplomske naloge naj študent na izbrani množici virov prikaže napredne gradnike poizvedovalnega jezika SPARQL v referenci z ostalimi poizvedovalnimi jeziki (npr. SQL, XQuery ipd.). Pri primerjavi naj bo poudarek na porazdeljenih poizvedbah, ki so ključna prednost poizvedovalnega jezika SPARQL. Kritični primerjavi naprednih konceptov naj sledi tudi praktična implementacija demonstracije poizvedovalnega vmesnika po porazdeljenih podatkih v obliki prototipne informacijske rešitve.

Mentor:

doc. dr. Dejan Lavbič

Fakulteta za računalništvo in informatiko
2
UNIVERZA V LJUBLJANI

Dekan:

prof. dr. Nikolaj Zimic

Izjava o avtorstvu diplomskega dela

Spodaj podpisani Alen Roguljić, z vpisno številko **63100296**, sem avtor diplomskega dela z naslovom:

Napredni gradniki poizvedovalnega jezika SPARQL

S svojim podpisom zagotavljam, da:

- sem diplomsko delo izdelal samostojno pod mentorstvom doc. dr. Dejana Lavbiča,
- so elektronska oblika diplomskega dela, naslov (slov., angl.), povzetek (slov., angl.) ter ključne besede (slov., angl.) identični s tiskano obliko diplomskega dela,
- soglašam z javno objavo elektronske oblike diplomskega dela v zbirki "Dela FRI".

V Ljubljani, dne 14.02.2014 Podpis avtorja:

Zahvaljujem se mentorju doc. dr. Dejanu Lavbiču za vso pomoč in nasvete pri izdelavi diplomske naloge. Posebna zahvala pa gre moji družini, ki me je skozi obdobje študija spodbujala in podpirala.

Kazalo

Seznam uporabljenih kratic in simbolov

ANSI (American National Standards Institute) – ameriški državni inštitut, ki skrbi za razvoj standardov za izdelke, storitve, sisteme, procese in osebje
ARQ – SPARQL procesor za Jena ogrodje, ki omogoča izvajanje poizvedb poizvedovalnega jezika SPARQL
AWT (Abstract Window Toolkit) – prvotno grafično okolje za izgradnjo Java grafičnih aplikacij
Basic Auth – vrsta avtentikacije z uporabniškim imenom in geslom
BASE (Basically Available, Soft state, Eventually consistent) – model, ki se uporablja pri NoSQL podatkovnih bazah
BNode – prazno vozlišče
C++ - splošno namenski računalniški programski jezik
DCL (Data Control Language) – skupina SQL ukazov za dodajanje in odvzemanje pravic uporabnikom baze
DDL (Data Definition Language) – skupina SQL ukazov za ustvarjanje in spreminjanje strukture baze oz. sheme
Digest Auth – vrsta avtentikacije, ki geslo zakodira s hash funkcijo
DML (Data Manipulation Language) – skupina SQL ukazov za upravljanje s podatki
FLWOR (For-Let-Where-Order by-Return) – pomemben izraz za poizvedovanje v jeziku XQuery
GUI (Graphical user interface) – uporabniški vmesnik za komunikacijo med

uporabnikom in računalnikom
HTML (Hyper Text Markup Language) – označevalni jezik za izdelavo spletnih strani
IDE (Integrated development environment) – okolje kot je Eclipse, ki omogoča pisanje in razvoj programske opreme
ISO (International Organization for Standardization) – mednarodno združenje organizacij za standardizacijo na vseh področjih, razen elektrotehnike in elektronike
Java – objektno usmerjeni ter prenosljiv jezik
JavaScript – objektno skriptni programski jezik
JSON (JavaScript Object Notation) – sintaksa za shranjevanje in izmenjavo podatkov
N-triples – oblika zapisa RDF trojic
Notation 3 – oblika zapisa RDF trojic
NoSQL (Not only SQL) – vrsta podatkovnih baz, ki se uporablja za delo z velikimi količinami podatkov
OAuth – odprto kodni standard za avtentikacijo z uporabniškim imenom in geslom
P2P (Peer-to-peer) – sistem omrežja za izmenjavo datotek, kjer vozlišča (računalniki) pošiljajo vire vsem ostalim vozliščem brez centraliziranega sistema
PHP (PHP Hypertext Preprocessor) – programski jezik za razvoj dinamičnih spletnih vsebin
RDBMS (Relational database management system) – sistem za upravljanje z relacijskimi bazami
RDF (Resource Description Framework) – osnovni format zapisa podatkov na semantičnem spletu
RDFa (Resource Description Framework in Attributes) – omogoča dodajanje semantike podatkom v HTML ter XHTML dokumentih
RDF shema – je dodatek RDF-u, ki definira vire z razredi, vrednostmi ter lastnostmi

RDQL (RDF Data Query Language) – povpraševalni jezik, ki omogoča poizvedovanje po podatkih shranjevanih v RDF formatu
RDF/XML – sintaksa s katero predstavimo RDF grafe
RQL (Resource Query Language) – eden prvih deklarativnih jezikov za poizvedovanje
XML (Extensible Markup Language) – razširljiv označevalni jezik za shranjevanje strukturiranih podatkov
SPARQL (Simple Protocol and RDF Query Language) – poizvedovalni jezik za delo s podatki v obliki RDF
SQL (Structured Query Language) – strukturirani povpraševalni jezik za delo z relacijskimi podatkovnimi bazami
Swing – primarno grafično okolje za Javo
TCL (Transaction Control Language) – je podmnožica ukazov SQL in se uporablja za kontroliranje transakcij v relacijskih bazah
Triplestore – vrsta baze za shranjevanje in pridobivanje trojic
Turtle – format zapisa RDF trojic
UnQL (Unstructured Data Query Language) – standardiziran poizvedovalni jezik za NoSQL podatkovne baze
URI (Uniform resource identifier) – niz, ki identificira ime spletnega vira
URL (Uniform resource locator) – niz, ki enolično določa spletni naslov
URN (Uniform resource name) – služi za predstavitev imena vira, ne pa tudi poti, kako pridemo do tega vira
W3C (World Wide Web Consortium) – mednarodni inštitut, kjer razvijajo standarde za svetovni splet
XQuery – poizvedovalni jezik po XML podatkih

Povzetek

Na spletu je veliko podatkov, ki so večinoma razumljivi samo ljudem. Kaj pa podatki, ki jih razume računalnik, ali podatki, ki bi omogočali uporabniku spleta, da dobi kar se da več podatkov skupaj na enem mestu. Že nekaj časa imamo povezave z ene spletne strani na drugo, vendar pri razvoju semantičnega spleta ne gre samo za povezavo spletnih strani med seboj, temveč je cilj omogočiti medsebojno povezavo vseh podatkov na spletu, s čimer bi dosegli, da bi podatke razumeli tudi računalniki, ki bi tako pomagali ljudem pri hitrejšem in bolj učinkovitem iskanju podatkov, do katerih želimo priti. Za vse to pa je potrebno podatke shraniti v obliko RDF, po kateri lahko poizvedujemo s poizvedovalnim jezikom, kot je SPARQL.

Zato smo v sklopu diplomske naloge razvili program, ki omogoča in prikazuje napredne gradnike za poizvedovanje po podatkih na oddaljenih SPARQL točkah, kot sta oddaljena točka, ki hrani podatke o novicah iz 24 ur, ter DBPedija, in poizvedovanje po lokalni RDF datoteki, v katero lahko podatke preko programa tudi vnesemo. Glavni namen je bil prikazati delovanje naprednih gradnikov poizvedovalnega jezika SPARQL, predvsem glede poizvedovanja, saj vemo, da se poizvedovanje oz. tako imenovani SELECT ukazi pri povpraševalnih jezikih uporabljajo v večji meri. Ključnega pomena je bilo prikazati napredne funkcionalnosti poizvedovalnega jezika SPARQL. Program temelji predvsem na problemski domeni novic iz 24ur ter DBPedije, vendar pa smo omogočili tudi ustvarjanje lastnih novic, katere se shranijo v lokalno RDF datoteko, in nato omogočili poizvedovanje tudi po teh podatkih iz lokalne datoteke. Cilj je bil tudi ta, da uporabniku prikažemo delovanje

porazdeljenih poizvedb, ki se uporabljajo za poizvedovanje po podatkih iz več oddaljenih točk naenkrat.

Ključne besede: SPARQL, RDF, semantični splet, Java, oddaljene SPARQL točke

Abstract

There are a lot of data on the web that is comprehensible only for humans. But what about the data understandable to computers which could help gather all the necessary data for the user, all in one place. The web site connections exist quite a while, but the main goal of sematic web is to enable the connections not only between web sites, but also between the data which could help computers understand and later help people with faster and more effective search. But for all that to be possible, it is necessary for us, to save the data into the RDF form, which allows us enquire with SPARQL query language.

This is why we have developed a program that shows and allows us to use advanced key words for query on distant SPARQL endpoints, such as endpoint which store data about news from 24ur website and DBPedia, and enquire on local RDF file, which also allows us to insert the data. The main purpose was to show SPARQL's use of advanced keywords mainly with equiries, since we know that equiries or so called SELECT queries are most commonly used in query languages. It was also vital to show the advanced functions of SPARQL. The program is mainly based on the news from 24ur domain and DBPedia. We have also enabled a function, that allows us to create our own news, that are saved into the local RDF file and are later allowed to use in equeries. We also wanted to show the user the operation of distributed queries that are used for enqueries from multiple distant endpoints at a time.

Keywords: SPARQL, RDF, semantic web, Java, SPARQL endpoints

Poglavje 1

Uvod

Ko začnemo z učenjem spletnih tehnologij, kot so na primer PHP, JavaEE, HTML ipd., nikoli ne pomislimo, kako so podatki na internetu med seboj povezani, kako ena spletna stran ve za drugo, kako ena spletna stran črpa podatke z druge spletne strani in kako podatke na spletu shraniti, da jih bodo razumeli tudi računalniki. Tukaj govorimo o semantičnem spletu. Podatki v semantičnem spletu so shranjeni v RDF formatu, ki je mehanizem za opisovanje podatkov na internetu in je ključnega pomena, da razumemo njegov način hranjenja le teh, poizvedovanje po njih, torej samo manipulacijo s podatki v tej obliki. Ob tem pa potrebujemo še nek poizvedovalni jezik, in to je SPARQL, katerega bom v diplomski nalogi na primeru podatkov iz 24ur ter DBPedije podrobno opisal in pokazal, kako se z njegovo pomočjo zelo enostavno poizveduje po podatkih.

Tako kot je SQL nepogrešljivi povpraševalni jezik pri relacijskih podatkovnih bazah, tako je SPARQL nepogrešljivi pripomoček za poizvedovanje po podatkih, shranjenih v RDF obliki [11]. Vendar pa SPARQL ne omogoča samo osnovnega pridobivanja in manipuliranja s podatki, temveč tudi napredne storitve, kot so porazdeljene poizvedbe - dostopi do oddaljenih SPARQL točk. To storitev bom v nadaljevanju še bolj podrobno opisal ter na primerih v programu, ki smo ga razvili, pokazal, kako je mogoče pridobiti podatke iz oddaljenih SPARQL točk.

SPARQL je dokaj mlad poizvedovalni jezik, vendar je do sedaj izdal že dve različici. Nova različica 1.1 (marec 2013), je prinesla kar nekaj novosti iz jezika SQL, ki so nepogrešljive pri resnem delu s podatki, shranjenimi v RDF obliki. Učenje poizvedovalnega jezika SPARQL je dokaj lahko opravilo, predvsem za tiste, ki že od prej poznajo povpraševalni jezik SQL, kajti razlika med njima ni tako zelo velika in le-ta uporabniku omogoča zelo širok spekter možnosti za manipulacijo s podatki.

Diplomsko nalogo sem razdelil v 4 ključna poglavja. V drugem poglavju bom naredil primerjavo med tremi večjimi poizvedovalnimi jeziki: SPARQL, SQL ter XQuery in opisal, kje in kdaj je primerno kateri jezik uporabiti. V tretjem poglavju bom predstavili RDF (angl. Resource Description Framework) podatkovni model, ki se uporablja za hranjenje podatkov, po katerih poizvedujemo s poizvedovalnim jezikom SPAQRL, zato je pomembno, da razumemo zgradbo ter koncept delovanja RDF-a. V tem poglavju bom tudi opisal arhitekturo podatkov na strani 24ur, ki je ena izmed arhitektur, preko katere bom skozi diplomsko nalogo tudi podajal primere. V tem poglavju pa bom predstavil še NoSQL podatkovne baze, ki so alternativa relacijskemu pristopu hranjenju podatkov. NoSQL je zanimiv predvsem zato, ker omogoča hranjenje in poizvedovanje po velikih količinah podatkov. Sledi četrto poglavje, v katerem bom podrobno opisal poizvedovalni jezik SPARQL, vse njegove napredne gradnike na primeru poizvedb, njihovo delovanje, kaj bi bilo mogoče izboljšati ter bolj podrobno predstavil pomembne ukaze, kot so povezovanje podatkov znotraj SPARQL poizvedbe – rezervirana beseda SERVICE, OPTIONAL, FILTER, GRAPH, UNION ter mnogi drugi. V nadaljevanju pa še dodatne gradnike, ki ponujajo dodatne napredne možnosti, kot je spreminjanje podatkov.

Za zaključek diplomske naloge pa sem predstavil na primeru programa, ki sem ga napisal v Javi kako "graph-based" strežniki omogočajo izvajanje SPARQL poizvedb. Le-ta je napisan na podlagi oddaljenih SPARQL točk, kot je [1], ki zajema podatke o novicah s spletne strani 24 ur [5] ter DBpedia [3],ki zajema podatke iz Wikipedije [6]. V primeru povezave podatkov iz

24ur ter podatkov iz DBpedie smo pokazali delovanje ukaza SERVICE, ker je le-ta ključen in pomemben, da lahko s poizvedovalnim jezikom SPARQL opravljamo poizvedbe po večjem številu oddaljenih grafov naenkrat. Poleg te pomembne funkcionalnosti smo prikazali še, kako deluje shranjevanje oz. ustvarjanje podatkovnega modela, v našem primeru model novic, ki je dokaj okrnjen, za razliko od tistega z oddaljene točke [1], ter poizvedovanje po podatkih, ki so shranjeni lokalno v RDF datoteki.

Poglavje 2

Primerjava SPARQL-a z ostalimi poizvedovalnimi jeziki

2.1 SPARQL

SPARQL je tako poizvedovalni jezik po podatkih, zapisanih v RDF obliki, kot tudi protokol za dostop do RDF podatkov. SPARQL je sorazmerno mlad poizvedovalni jezik, prav tako kot XQuery. Pojavil se je 15. januarja 2008, v verziji SPARQL 1.0, in uradno postal standard W3C [11].

Na SPARQL lahko gledamo kot na SQL, ki je v relacijskih podatkovnih bazah nepogrešljiv jezik, le da je ta nepogrešljiv poizvedovalni jezik v semantičnem spletu, ki pa se iz dneva v dan vse bolj razširja in uveljavlja. Dandanes se skoraj vsa vprašanja, ki si jih zastavimo, skrivajo na spletu. Namen semantičnega spleta pa je vse podatke, ki vsebujejo odgovor na posameznikova vprašanja, smiselno povezati skupaj, da bi lahko vsak uporabnik spleta čim hitreje in učinkoviteje uporabil znanje, pridobljeno iz teh podatkov. Vsi podatki so shranjeni v RDF obliki, ki jo bom v naslednjem poglavju bolj podrobno opisal. Za poizvedovanje po teh podatkih pa seveda potrebujemo poizvedovalni jezik SPARQL, ki nam omogoča ogromno možnosti za delo s podatki, več o tem pa v četrtem poglavju.

Glavna razlika med SPARQL-om, XQuery-jem ter SQL-om je v tem, da

SPARQL poizveduje po podatkih, ki so shranjeni v obliki grafa, XQuery po podatkih, shranjenih v XML obliki, SQL pa po podatkih, shranjenih v relacijskih bazah. Če pogledamo, kako obsežen je splet, in koliko imamo podatkov, po katerih lahko poizvedujemo, ugotovimo, da če bi bili vsi podatki na spletu med seboj povezani, je splet zelo velika zbirka podatkov. SPARQL omogoča uporabniku dostopati do tistih podatkov, s katerimi želi delati. V primeru, da želimo dostopati do vseh držav v Evropi, njihovih populacij ter številu brezposelnih oseb, ni potrebno, da za to zapravljamo veliko denarja za razvoj aplikacij, ki bi te podatke pridobile, temveč nam SPARQL omogoča, da napišemo samo eno poizvedbo, ki bo bodisi iz enega vira (Wikipedija npr.) ali več virov pridobila informacije o željenih podatkih.

2.2 SQL

SQL je strukturiran povpraševalni jezik za delo z relacijskimi podatkovnimi bazami in je v splošnem zelo podoben SPARQL-u, tako po načinu poizvedovanja kot po ukazih, ki se uporabljajo za poizvedovanje po podatkih.

SQL se je razvil iz jezika SEQUEL (angl. Structured English Query Language), ki je nastal leta 1974, vendar je šele kasneje postal jezik za specifikacijo konceptualne, notranje in zunanjih shem. Leta 1986 je postal standard za ANSI, leto kasneje pa še standard ISO [14]. SQL je med ostalimi poizvedovalnimi jeziki, ki jih bom v tem poglavju opisal, najstarejši.

Najpomembnejše ukaze za poizvedovanje s tem jezikom lahko razdelimo v dve ključni skupini: DDL, ki se uporablja za ustvarjanje in spreminjanje strukture baze oz. sheme, ter DML, ki se uporablja za upravljanje s podatki. DML tip podatkov je nepogrešljiv pri delu z bazo, saj so stavki SELECT, INSERT, UPDATE in DELETE najpogosteje uporabljeni stavki za delo z relacijskimi podatkovnimi bazami. SQL ima definirana še dva tipa stavkov, to sta DCL, ki se uporablja za dodeljevanje in odvzemanje pravic določenim uporabnikom nad podatkovno bazo, ter TCL, ki se uporablja za kontroliranje sprememb, ki jih naredijo DML stavki, tako imenovane transakcije. SQL pa

ne pozna samo teh tipov ukazov, temveč tudi ostale napredne elemente, kot so agregatne funkcije, ki vračajo rezultat glede na vrednosti iz določenega stolpca, ter skalarne funkcije, ki vračajo vrednosti glede na vhod enega parametra [7]. SQL ponuja še mnogo več kot le-to, praktično vse, kar potrebuje vsakdo, ki se z relacijskimi podatkovnimi bazami ukvarja profesionalno ali pa samo za zabavo.

```
-- PRIMER
SELECT NovicaId, ImeNovice
FROM Novice
WHERE Avtor LIKE '%le%' AND PovprGlasov > 4;
```

V zgornjem primeru imamo poizvedbo, ki vrne dva podatka: id novice ter ime novice iz tabele Novice, glede na pogoj znotraj WHERE ukaza, ki pravi, da mora avtor novice imeti ime, ki vsebuje črki "le", povprečje glasov neke novice pa mora biti večje od 4.

2.3 XQuery

XQuery ima zelo močno podporo pri poizvedovanju po podatkih, skriptah, tekstovnih datotekah ter raznih spletnih storitvah, kot je PayPal, ter socialnih omrežjih, kot sta Twitter in Facebook. Namenjen je prav za splet, prav zaradi tega razvijalci z uporabo XQuery-ja dosežejo enako funkcionalnost kot pri ostalih poizvedovalnih jezikih, kot je SQL, vendar 5x manj kode, kar pomeni, da je razvoj 5x hitrejši, potrebuje manj vzdrževanja in ponuja manj možnosti za napake, vendar pa je zaradi teh prednosti tudi zelo počasen.

XQuery se uporablja za poizvedovanje po podatkih, shranjenih v XML obliki, za pretvorbo XML v XHTML in kasneje za uporabo na spletu, za uporabo v spletnih storitvah, za spreminjanje in računanje novih podatkov, za združevanje podatkov iz več različnih dokumentov in še mnogo več. Namenjen je tako za strukturirane kot za nestrukturirane podatke, omogoča, da lahko podatke pretvorimo v katero koli strukturo in v tej obliki podatke vr-

nemo preko poizvedbe. Osnovna struktura za večino poizvedb je sestavljena iz FLWOR izraza. Ta izraz vsebuje ključne besede: for, let, where, order by ter return za poizvedovanje, vendar pa XQuery vsebuje še preko 100 ostalih funkcij za delo z nizi, datumi, matematičnimi izrazi ter mnogo več, kot je na primer pisanje svojih lastnih funkcij [49].

```
(: PRIMER :)
for $x in doc("novice.xml")/novice/novica
where $x/povpGlasov>4
order by $x/naslov
return $x/naslov
```

V zgornjem primeru vidimo primer FLWOR izraza, kjer v for bloku izberemo vse elemente 'novica', ki so podkoreni elementa 'novice' v spremenljivko x iz datoteke novice.xml. V where bloku imamo pogoj za vse novice, ki imajo podelement 'povpGlasov' in je njihova vrednost več kot 4. Nato sledi ukaz order by, kjer določimo, da želimo sortirati rezultate po elementu 'naslov'. Na koncu pa še napišemo, kaj želimo dobiti kot rezultat, in v tem primeru vrnemo elemente 'naslov'.

Poglavje 3

RDF

Semantični splet je zelo obširen pojem, na splošno pa lahko rečemo, da je velika baza podatkov, integrirana v arhitekturo, kjer ima vsak podatek svoj naslov in mesto, da lahko računalnik podatke učinkovito uporablja in razume. Zato pa je zanj potreben RDF standard, ki omogoča, da so podatki v semantičnem spletu organizirani.

Na spletu imamo ogromno podatkov, kot so imena avtorjev spletnih strani, knjig, datumi nastanka spletnih strani, števila prebivalcev neke države in ogromno ostalih informacij, ki jih želimo shraniti, jim določiti neke vrednosti in jih kasneje uporabljati. Za shranjevanje teh podatkov uporabljamo RDF, ki je bil prvotno narejen kot meta podatkovni model, kjer lahko shranjene podatke predstavimo v obliki grafa.

RDF se uporablja za shranjevanje spletnih virov ter meta podatkov o spletnih virih, sintaksa pa omogoča izmenjavo in uporabo podatkov, shranjenih na različnih lokacijah. Podatki so shranjeni v obliki stavkov oz. trojic, kjer množica stavkov predstavlja graf – predmet, predikat, objekt tako imenovane trojice. Predmet predstavlja vir podatka, predikat predstavlja vez med predmetom in objektom oz. ime lastnosti, objekt pa nosi vrednost lastnosti [8].

Podatki, shranjeni v RDF modelu, so lahko shranjeni v več možnih oblikah (formatih), kot so JSON, RDF/XML, Notation 3 (N3), TURTLE, za poizvedovanje pa se dandanes uporablja poizvedovalni jezik SPARQL, ki ga

bom v naslednjem poglavju podrobno opisal. Skozi čas se je za poizvedovanje po podatkih uporabljalo več jezikov, kot so RDQL, ki je prednik SPRAQL-a, RQL, ki je eden prvih deklarativnih jezikov za poizvedovanje, N3QL ter ostali [9].

Trojček - stavek		
Predmet	Predikat	Objekt
(identifikacija vira)	(ime lastnosti)	(vrednost lastnosti)
<urlnovice>	rdf:type	sioc:Post

Tabela 3.1: Enostaven trojček v RDF grafu

Prednosti RDF-a:

- boljše razumevanje podatkov
- interoperabilnost
- razumljiv tako človeku kot računalniku
- strukturirana zgradba XML datoteke
- omogoča učinkovito spreminjanje sheme

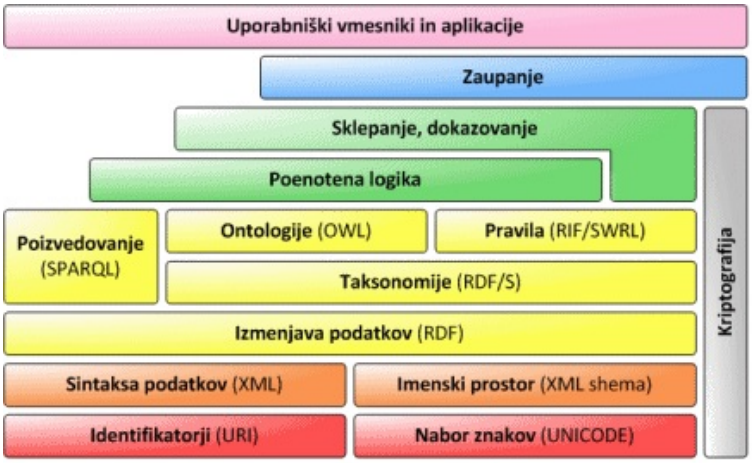
Slabosti RDF-a:

- omejitve, kjer te RDF omejuje, kako zgraditi XML datoteko
- zahteva poznavanje RDF sintakse
- poizvedovanje po podatkih v primerjavi z relacijskimi bazami je manj učinkovito

3.1 RDF Shema

RDF shema je semantični dodatek RDF-u, ki mu lahko rečemo tudi besednjak, in omogoča mehanizem za opisovanje skupin s podobnimi viri in relacijami med njimi, torej je zelo podoben objektno orientiranem programiranju, kajti bistvo RDF sheme je v tem, da lahko definiramo razrede ter lastnosti in jih med seboj povežemo. Lastnosti predstavljajo povezave oz. relacije med viri. Prednost RDF sheme je v tem, da nam omogoča, da z njo postavimo razredno hierarhijo, ki nam posledično omogoča lažje razumevanje podatkov ter bolj učinkovito iskanje po podatkih, pomembno pa je povedati tudi to, da se uporablja za definiranje semantike RDF podatkov oz. podajanje pomena podatkom [15].

Vsaka RDF shema je tudi RDF dokument, ki uporablja svoj besednjak oz. sintakso, in je standard, katerega se morajo držati vsi programerji. Posledično omogoča, da lahko ljudje, ki delajo na različnih podatkih ob različnem času, skupaj povezujejo podatke.



Slika 3.1: Arhitektura semantičnega spleta [50]

3.2 Načini za shranjevanje podatkov

Ljudje že od začetka razvoja računalništva hranimo vse mogoče podatke, od podatkov o prebivalcih določene države do podatkov o nebesnih telesih. In vse te podatke je potrebno shraniti, da jih lahko kasneje uporabimo, bodisi da so to zapisi glasbe na disku ali pa podatki, shranjeni v neki podatkovni bazi.

Ko govorimo o semantičnem spletu, želimo, da se lahko podatki med seboj povezujejo med različnimi domenami podatkov na spletu, zato potrebujemo pravi način za shranjevanje le teh. Podatki so zelo pomembni, saj nam lahko prinesejo veliko novega znanja in informacij. Podatki, ki se uporabljajo v semantičnem spletu, so namenjeni predvsem računalnikom, da lažje razumejo vsebino neke spletne strani in omogočijo uporabniku, da lažje pride do želene vsebine.

3.3 NoSQL podatkovne baze

NoSQL podatkovne baze se uporabljajo za velike količine podatkov in prav zaradi tega so jih tudi razvili, ker relacijske podatkovne baze niso bile sposobne tako učinkovito hraniti velikih količin podatkov. NoSQL ne pomeni, da ne smemo uporabljati SQL poizvedb, temveč pomeni, da ne omogoča samo SQL (Not Only SQL), temveč tudi ostale poizvedovalne jezike. NoSQL podatkovne baze imajo lastnosti modela BASE, kjer je razpoložljivost sistema zelo pomembna, podatkovna shema ni nujno potrebna, razvoj je hitrejši, skalabilnost se poveča, delovanje je pa precej hitrejše kot pri relacijskih podatkovnih bazah. Standardiziran poizvedovalni jezik za NoSQL baze je UnQL, ki je narejen na podlagi relacijske algebre. Težava pri tem poizvedovalnem jeziku je, da ga večina ponudnikov NoSQL podatkovnih baz ne podpira. V primeru da bi ga, bi bil tako kot SQL pri relacijskih bazah, ključnega pomena za poizvedovanje po NoSQL bazah različnih ponudnikov [48]. Glavni namen pri ustvarjanju NoSQL podatkovnih baz je bil zagotoviti skalabilnost (v primeru, da število podatkov konstantno narašča), hitrost ter visoko

stopnjo dostopnosti. NoSQL ima v primerjavi z relacijskimi podatkovnimi bazami manj funkcionalnosti, vendar ima prednost v tem, da ponuja večjo zmogljivost. Podatki pa so shranjeni v zbirkah, kar omogoča, da so lahko v strukturirani ali pa tudi v nestrukturirani obliki. Pri relacijskih bazah vemo, da so podatki lahko shranjeni samo v strukturirani obliki. Primeri nestrukturiranih podatkov so: avdio posnetki, komentarji na socialnih omrežjih, tekstovna sporočila in vsi podobni podatki, ki konstantno naraščajo in po možnosti tudi spreminjajo strukturo. Primeri NoSQL podatkovnih baz so Couchbase, MongoDB, AllegroGraph, Redis in mnogi drugi, kajti sedaj je na trgu okoli 150 NoSQL podatkovnih baz, katere lahko razdelimo v več skupin: ključ-vrednost, razširljivi zapis, dokumentno usmerjene ter nekatere ostale vrste [31].

Česar NoSQL podatkovne baze ne podpirajo, so JOIN operacije, ker prav zaradi teh relacijske podatkovne baze niso skalabilne. Ne podpirajo še kompleksnih transakcij ter omejitev pri poizvedbah, zato morata biti ti dve funkcionalnosti implementirani na aplikacijskem nivoju. NoSQL podatkovne baze je primerno uporabiti, kadar povezave med relacijami niso pomembne, ker lahko to implementiramo na aplikacijskem nivoju, kadar se ukvarjamo z naraščajočimi podatki (blogi, raznoraznimi logi, komentarji na socialnih omrežjih itd.), če so podatki nestrukturirani, če se struktura hitro spreminja (danes imamo 5 atributov, jutri 15) ipd. [10].

3.4 N-triples

N-triples format zapisa RDF trojic je zelo preprost način, ki ga je W3C ustvaril z namenom, da bi bil enostavnejši od Turtla ter Notation 3. Datoteka, v kateri so shranjeni podatki, ima končnico ".n", sintaksa pa je zelo preprosta, kar je vidno iz primera. Vsaka vrstica mora vsebovati trojček: predmet, predikat ter objekt, kjer mora biti predmet URI naslov, prazno vozlišče ali URI naslov zapisan v <>, predikat mora biti URI naslov, objekt pa je lahko bodisi prazno vozlišče, tekst zapisan v narekovajih ali URI naslov zapisan

med znakoma <>. Vsak trojček mora biti zapisan v svoji vrstici in ločen s piko na koncu vrstice [11].

Primer:

```
<www.test.local/novice/novica1_20392>
    <http://www.w3.org/2000/01/rdf-schema#label> "Novica1" .
<www.test.local/novice/novica2_34285>
    <http://www.w3.org/2000/01/rdf-schema#label> "Novica2" .
<www.test.local/novice/novica3_00125>
    <http://www.w3.org/2000/01/rdf-schema#label> "Novica3" .
```

3.5 RDF/XML

Je najstarejša RDF serializacija in obenem tudi oblika, v kateri so RDF podatki najpogosteje shranjeni. Datoteka te oblike ima končnico ".rdf". Slabost RDF/XML oblike je v tem, da če imamo strukturo podatkov zelo obsežno, nam XML ne omogoča lahkega razumevanja hierarhije podatkov, kjer je vsebina ene trojice shranjena znotraj <rdf:Description>, v podvozliščih pa so shranjene določene lastnosti in vrednosti. Prednost te oblike je v tem, da večina programskih jezikov podpira XML format ter da XML omogoča uporabo imenskih prostorov oz. predpon, ki nam olajšajo delo, da nam ni treba pisati dolgih URI naslovov [11].

Primer:

```
<rdf:RDF
    xmlns:news="http://opendata.lavbic.net/news/"
    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns:dct="http://purl.org/dc/terms/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
    xmlns:sioc="http://rdfs.org/sioc/ns#"
    xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#" >
```

```
  <rdf:Description
      rdf:about="http://www.testNovice.com/novica-aa10-cdd919f35cad">
    <dct:title>Naslov novice</dct:title>
    <sioc:UserName>Janez Novak</sioc:UserName>
    <dct:abstract>Povzetek vsebine novice ...</dct:abstract>
    <sioc:Content>Vsebina novice ... </sioc:Content>
    <dct:created>2013-12-28 21:13:33</dct:created>
    <rdfs:seeAlso>http://www.testNovice.com/novica1</rdfs:seeAlso>
  </rdf:Description>
</rdf:RDF>
```

3.6 N3 - Notation 3

N3 je projekt, ki si ga je zamislil Tim Berners-Lee, izumitelj interneta, ko je prišel na idejo, da bi lahko RDF/XML format spremenili tako, da bi bil format preprosto berljiv tudi za človeka ter da bi še vedno omogočal predpone, ki bi omogočale krajšanje dolgih URI naslov. Svojo zamisel je dokončal in dandanes N3 format poleg lažjega branja ter pisanja ponuja še mnogo več, kot je na primer možnost pisanja funkcij.

V primeru je predstavljena sintaksa, kjer vidimo, da podpičje pomeni, da se naslednji zapis predikata in objekta nanaša na predmet iz prejšnjega zapisa. Če pa je na koncu vrstice pika, to pomeni, da naslednji zapis predstavlja nov predmet, ki je po vsej verjetnosti predstavljen z drugimi predikati, če ne vsaj objekti [11].

Primer:

Shranjeni podatki predstavljajo tri dejstva:

- Novica 21 ima naslov Poizvedovalni jezik Sparql.
- Novica 21 je bila ustvarjena 2014-01-04.
- Komentar 3212 ima 23 glasov.

```
@prefix dct: <http://purl.org/dc/elements/1.1/> .
@prefix news: <http://opendata.lavbic.net/news/> .

<http://www.testnovice.local/novice/novica21>
     dct:title "Poizvedovalni jezik Sparql" ;
     dct:created "2014-01-04" .
<http://www.testnovice.local/komentarji/komentar3212>
     news:nVotings "23"
```

3.7 RDFa

RDFa kratica pomeni RDF(Resource Description Framework) + atributi, ki niso namenjeni ljudem, temveč računalnikom, in niso namenjeni, da spletnim stranem dodamo videz, temveč pomen podatkom, da lahko računalniki hitreje najdejo želeno vsebino posameznika.

Bistvo RDFa-ja je dodati semantiko vsebini spletnih strani oz. vsaki XML, HTML ali XHTML datoteki. RDFa je primerno uporabiti, če želimo, da bi imela naša spletna stran čim več obiskovalcev in da bi bila optimizirana, kar je dandanes zelo pomembno. Za čim večji obisk na spletnih strani je pri tem dobro uporabitit tudi RDFa. Drug dober odgovor je, da ljudje uporabljajo programe, da s spletnih strani pridobijo določene informacije, bodisi o voznih redih avtobusov ali informacije o delovnem času trgovin ipd. Za to je potrebnega veliko dela, od iskanja podatkov na spletni strani, pridobivanja in kopiranja le teh, šele nato jih lahko uporabimo. Če pa imamo na spletni strani uporabljen RDFa, pa je mogoče podatke enostavno pridobiti in jih uporabiti. Nova verzija RDFa 1.1 omogoča, da ni potrebno pisati predpone za nekatere znane besednjake, kot je FOAF [45].

```
<p xmlns:dc="http://purl.org/dc/elements/1.1/"
  about="http://www.test.local/diplosmka">
  V diplomski nalogi
```

```
  <cite property="dc:title">SPARQL</cite>, katere avtor je
  <span property="dc:creator">Alen Roguljic</span>
  so predstavljeni napredni gradniki poizvedovalnega jezika SPARQL.
  <span property="dc:date" content="2014-01-10">Januar 2014</span>.
</p>
```

3.8 Shranjevanje velikih količin podatkov

V semantičnem spletu je želja povezati skupaj čim večje število podatkov, kar pripelje do zelo velikega števila trojic, zato je potrebno podatke primerno shraniti. Za shranjevanje tako velikega števila podatkov niso primerni formati, kot sta N-triples ali RDF/XML, temveč SUPB-ji imenovani triplestore, ki so narejeni za shranjevanje in poizvedovanje tudi več bilijonov trojic. Narejeni so bili testi, ki so pokazali, da lahko AllegroGraph naloži več kot 1 trilijon trojic v sistem, za kar so potrebovali 338 ur. Malo manj zmožen je OpenLink Virtuoso, ki lahko obravnava okoli 15 bilijonov trojic in še mnogi drugi [46], kot je Sesame Native, ki je odprto kodno ogrodje za shranjevanje RDF podatkov in omogoča delo s približno 70 bilijonov trojicami [39].

Vendar pa to niso vsi načini, kako shraniti veliko količino trojic. Obstajajo načini, ki omogočajo, da shranimo RDF podatke tudi v relacijske SUPB-je, kot sta MySql ter Oracle, vendar pa takšen način ni dobrodošel, kajti triplestore način shranjevanja ima prav poseben standardiziran način za shranjevanje podatkov in standardiziran način za poizvedovanje po RDF podatkih [11].

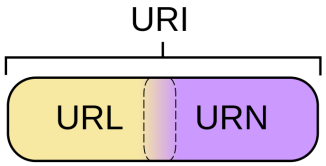
3.9 RDF graf

Graf je tehnični pojem za množico RDF trojic, kjer je graf predstavljen kot drevesna struktura brez hierarhije. Vsako vozlišče je lahko povezano s katerim koli vozliščem. V RDF grafu vozlišča predstavljajo predmet ali objekte (vire), predikati pa so povezave med vozlišči. Preprost graf, ki ga

predstavimo s trojicami je sestavljen iz predmeta, predikata oz. lastnosti predmeta ter objekta oz. vrednosti lastnosti [11].

3.10 URI naslov

URI (uniform resource identificator) se uporablja za poimenovanje imenskih prostorov, identifikacijo spletnih virov, kot so dokumenti, slike, storitve in ostali viri.



Slika 3.2: Zgradba URI naslova

URI naslove uporabljamo tudi, ko govorimo o pisanju RDF dokumentov, kjer uporabljamo predpone, ki nam omogočajo, da nam ni potrebno znova in znova pisati URI naslovov imenskega prostora. Vendar pa lahko namesto predpon pišemo tudi celotne URI naslove, ki jih pišemo med <>.

V primerjavi z relacijskimi podatkovnimi bazami so URI naslovi tisti, ki identificirajo RDF vire in predstavljajo naslove stolpcev v tabeli relacijske baze, razlika je v tem, da so ti naslovi po celotnem svetu enaki, kar omogoča, da lahko podatke med seboj povezujemo iz različnih virov po svetu, namesto da povezujemo podatke iz različnih tabel v relacijski podatkovni bazi [11].

Poznamo pa tudi IRI naslove, ki so v bistvu URI naslovi, ki pa lahko vsebujejo še dodatne znake cirilice ter kitajščine. Ko pa govorimo o poizvedovanju s SPARQL-om, pa le-ta uporablja IRI naslove, kajti ta zajema večje število znakov v imenskem prostoru kot URI ali URL. URI (URL+URN) je sestavljen iz URL-ja, ki služi kot spletni naslov, na katerem najdemo želeno vsebino, kot tudi naslov, preko katerega pridemo do želenega vira, ter URN-

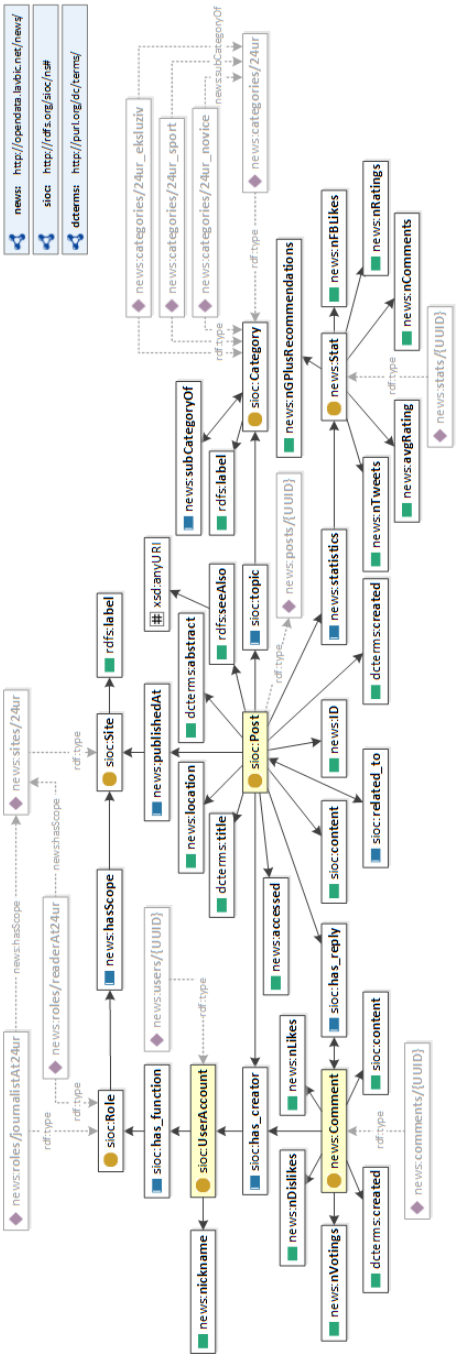
ja, ki služi za identifikacijo določenega vira na spletu. URN ne pove, kako priti do nekega vira, temveč predstavlja samo njegovo ime. Vendar pa URL in URN skupaj tvorita URI, s katerim identificirata posamezen vir na spletu. Včasih je težko ločiti URI od URL-ja, ker sta si tako podobna, vendar pa je med njima le neka bistvena razlika, po kateri ju ločimo. Ta je, da lahko URI uporabljamo tako za naslavljanje nekega spletnega mesta kot za imena datotek ali oboje, kar nam pride prav, ne samo pri brskanju po spletu, temveč tudi za izmenjavo datotek P2P(a peer to peer), medtem pa se lahko URL uporablja samo za naslavljanje lokacij spletnih virov [40].

3.11 Graf 24ur

Na sliki 3.3 je prikazan graf, nad katerim smo v sklopu diplomske naloge tudi izvajali poizvedbe jezika SPARQL in tako predstavili napredne gradnike le tega. Graf lahko razdelimo v štiri skupine. Prva pokriva novice, kjer imamo glavne podatke o novicah, nato so komentarji na novice, ki imajo svoje atribute, nato uporabnike, ki so lahko bodisi avtorji novic ali avtorji komentarjev, četrta glavna skupina pa je statistika novic, ki vsebuje atribute o številu glasov neke novice in podobno. Podatkovni model pa vsebuje še dve manjši podskupini, ki sta vloga nekega uporabnika (bodisi novinar ali bralec) ter kategorija, ki se deli na novice, šport ter ekskluziv, vsi trije pa so podkategorije 24ur. Graf se nahaja na oddaljeni točki [1], kjer se podatki konstantno zbirajo s spletne strani 24ur [5]. Za uporabo in dostop do te oddaljene točke lahko uporabimo spletni brskalnik in spletno stran [12], kjer lahko pišemo poizvedbe, ki se potem pošljejo na URL naslov [1], ki vsebuje parameter "query", njegova vrednost pa je niz poizvedbe. Pri dostopu do tega naslova je potrebno vnesti uporabniško ime ter geslo.

V sklopu diplomske naloge smo poizvedbe preverjali tudi preko dodatka za brskalnik Google Chrome – Postman [13], s katerim lahko izvajamo zahteve na poljubno spletno stran. V našem primeru smo uporabili še dva dodatna "headerja" za avtorizacijo, ker je dostop do oddaljene točke [1] zaščiten z uporabniškim imenom in geslom, ter enega za način prikaza podatkov v obliki JSON ali RDF/XML npr. Več v sliki 3.4.

Tretji način, kako lahko dostopamo do podatkov, pa je uporaba programskega jezika. V sklopu diplomske naloge smo razvili tudi program, ki omogoča poizvedovanje po podatkih iz oddaljenih SPARQL točk, ustvarjanje novih novic, iz katerih se zgradi RDF datoteka, ter poizvedovanje po njih. Program je napisan v programskem jeziku Java, več o tem pa v poglavju, ki zajema temo o programu, ki smo ga razvili.



Slika 3.3: Shema podatkov novic iz portala 24ur [5]

Slika 3.4: Primer uporabe programa Postman [13]

Poglavje 4

Napredni gradniki poizvedovalnega jezika SPARQL

Četrto poglavje zajema glavno temo diplomske naloge in opisuje večino naprednih funkcionalnosti poizvedovalnega jezika SPARQL. Namen tega poglavja je bralcu predstaviti bistvo in namen uporabe naprednih gradnikov poizvedovalnega jezika SPARQL ter narediti primerjavo z jezikom SQL.

4.1 Prazna vozlišča - Blank nodes

Če imamo nek podatek, kot je na primer naslov neke ustanove ali telefonska številka, ki ga želimo povezati z več kot enim predmetom v trojicah, je smiselno uporabiti prazno vozlišče oz. bnode. To je smiselno predvsem v primerih, kjer se podatki velikokrat ponavljajo in nam ni potrebno ponavljati kode. Torej prazno vozlišče služi kot neko vozlišče za grupiranje podatkov, ki nimajo imena niti trajne identitete, temveč le začasno, ki velja v času poizvedbe. V trojicah pa lahko zavzame mesto kot predmet ali objekt. Vlogo predmeta predstavlja, ko ga uporabimo za hranjenje podatkov, vlogo objekta pa predstavlja, ko ga uporabimo za povezovanje z nekim drugim predmetom.

S stališča RDF/XML je prazno vozlišče definirano kot rdf:Description element, ki ne vsebuje rdf:about elementa in mu lahko rečemo tudi anonimen vir, ki ne poda njegovega imena. Prazna vozlišča lahko tako kot URI naslovi in nizi predstavljajo RDF vozlišča [37].

Če uporabimo SPARQL poizvedbo, ki izpiše prazno vozlišče, bomo vsakič dobili drugačno vrednost kot tisto, ki je shranjena v trojici (npr.če imamo shranjeno _:a abc:name "Janez", nam bo poizvedba, kjer izpišemo predmet in objekt, vrnila neko drugo vrednost za predmet npr. _:x, ker uporabljamo prazna vozlišča, za objekt pa isto vrednost, kot je v trojici - "Janez".

Prazna vozlišča je smiselno uporabiti na nivoju enega dokumenta, na globalnem nivoju pa je to skorajda nesmiselno, kajti če nekdo spremeni podatke, bo druga oseba zelo težko oz. skoraj nemogoče razumela, kaj se je spremenilo.

Zaključimo lahko, da se je potrebno praznim vozliščem izogibati, če želimo imeti podatke shranjene kvalitetno, tako da jih bomo lahko kasneje tudi uspešno spreminjali in da bodo drugi razumeli, kaj in kako se je spremenilo. V realni uporabi se prazna vozlišča uporabljajo kot vozlišča, ki povezujejo različne sklope podatkov, vendar smisel povezanih podatkov na spletu ni v tem, da imamo anonimne vrednosti, temveč da imamo vrednosti, ki imajo nek bistven pomen za določen podatek.

Sintaksa 1:

_:poljubnoIme :predikat "test"

Turtle včasih uporablja oglate oklepaje [ ], ki predstavljajo prazno vozlišče.

Sintaksa 2:

[ ] :predikat "test"

Primer shranjenih podatkov s praznim vozliščem:

```
novica:7X8u22 dct:title ''Naslov novice'' ;
              dct:created ''2014-01-12 16:46:51'';
              news:location _:location .
_:location    vcard:postal-code ''1000'' ;
              vcard:locality ''Ljubljana'' ;
              vcard:country-name ''Slovenija'' .
```

Primerjava s SQL-om:

V naslednjem primeru gre za stavek, ki pove, da je Janez avtor novice, ki nima podanega naslova novice. Pri SQL lahko govorimo o vrednosti NULL ali praznem nizu, ker gre za niz. Pri SPARQL-u pa je to prazno vozlišče v tem primeru brez vrednosti.

```
"Janez avtor Null/blank"
```

Če pogledamo s stališča SQL-a, ne vemo, ali je Janez avtor neke novice, ker nima dodeljene vrednosti(Null). S stališča SPARQL-a pa vemo, da je Janez avtor nekega objekta oz. novice, vendar ime novice ignoriramo. Vidimo, da je razumevanje podatkov, kjer nimamo neke vrednosti podane, pri obeh jezikih precej različno.

4.2 Skupine vzorcev

Poizvedbe lahko naredimo bolj pregledne in uporabne s skupinami vzorcev(grupiranje), to pa ni vse, za kar lahko uporabimo grupiranje vzorcev. Predvsem uporabljamo skupine vzorcev za filtriranje in ločevanje pogojev v poizvedbi ter za povečanje učinkovitosti poizvedb.

V osnovi imenujemo trojice vzorci, ki so enostavna konjunkcija trojic, ki jih v sintaksi ločimo s piko. Te enostavne trojice so osnova za ostale,

bolj kompleksne vzorce, v katerih se uporabljajo besede, kot so FILTER, UNION in OPTIONAL. V praksi lahko naredimo enostavno poizvedbo iz dveh ali treh trojic, katere ločimo v skupine trojic tako, da je vsaka trojica obdana z zavitimi oklepaji. Skupine sestavljene iz več trojic skupaj znotraj enega bloka imenujemo skupina vzorcev, če pa imamo posebej vsako trojico znotraj bloka, pa le-to imenujemo osnovni grafični vzorec. [11]

Primer treh trojic znotraj ene skupine:

```
SELECT * WHERE
{
  { ?post a sioc:Post . ?post dct:title ?title . ?post news:ID ?id }
}
LIMIT 10
```

V prejšnjem primeru imamo poizvedbo, v kateri je ena skupina sestavljena iz treh trojic. Poizvedba vrne 10 zapisov, kjer kot rezultat dobimo podatke o viru novice, njenem naslovu ter vrednosti id zapisa.

Primer, kjer vsaka trojica predstavlja eno skupino:

```
SELECT * WHERE {
   { ?post a sioc:Post }
   { ?post dct:title ?title }
   { ?post news:statistics/news:nComments ?nCom }
   FILTER (?nCom > 10)
} LIMIT 10
```

V prejšnjem primeru imamo poizvedbo, v kateri imamo tri ločene skupine vzorcev, ter en pogoj, ki vrne samo tiste novice, ki imajo število komentarjev večje od 10. V tem primeru, kjer imamo 3 ločene vzorce, ni pomembno, na katero mesto postavimo FILTER ukaz, da dobimo rezultat za to poizvedbo. Je pa pomembno, da se zavedamo, da v primeru, če bi se FILTER ukaz na-

hajal znotraj prvega ali drugega vzorca, rezultata ne bi dobili, ker se ukaz FILTER nanaša le na tisto skupino vzorcev, v kateri se nahaja. Torej bi rezultat dobili samo še v primeru, če bi ukaz FILTER postavili znotraj tretjega vzorca, kjer podatke o številu komentarjev shranimo v spremenljivko ?nCom.

SPARQL ne omogoča besede AND, ki bi predstavljala konjunkcijo med trojicami, temveč za to uporablja zavite oklepaje. Pogosta je tudi uporaba pogojev v skupinah, ki omogoča, da lahko posamezno skupino vzorcev filtriramo. Če želimo uporabiti pogoj FILTER, ni pomembno, na katero mesto ga postavimo, če imamo več kot eno trojico. Poznamo pa tudi prazno skupino vzorcev, kjer med dvema zavitima oklepajema ne vpišemo nič. Takšen vzorec vrne bodisi prazen graf ali graf z eno rešitvijo, kjer nobena spremenljivka nima dodeljene vrednosti [16].

4.3 Property path

Property path se je pojavil šele v drugi verziji SPARQL-a in nam omogoča, da lahko med dvema vozliščema v grafu najdemo neko možno pot, ki je lahko poljubno dolga [11]. Če pa govorimo o poti, ki ima dolžino ena, pa govorimo o navadni trojici. Pot dolžine nič pa nam pove, da je vozlišče povezano samo s seboj [17]. Sama uporaba ukaza je preprosta, saj z ukazom delamo preko regularnih izrazov, ki jih uporabimo v predikatu trojice.

Poti ločimo na preproste in kompleksne. Preproste so tiste, ki vsebujejo operatorje /,∧,{n} in jim že vnaprej določimo dolžino poti. Kompleksni so pa tisti, ki vsebujejo operatorje *,?,+,n,m,{n},{n,},{,n} in nimajo določene dolžine poti, saj lahko s temi operatorji povemo le, kakšne naj bodo meje za dolžino poti [44].

Primer preproste poti (najdi povprečje glasov za novice, ki imajo statistiko za povprečje glasov):

```
SELECT * WHERE {
   ?post a sioc:Post .
   ?post news:statistics/news:avgRating ?rating
} LIMIT
```

Primer kompleksne poti (najdi vse vire in tipe, ki imajo eno ali več ponovitev):

```
SELECT * WHERE {
   ?resource rdf:type+ ?type
} LIMIT
```

Sintaksa	Opis
uri	URI oziroma ime predpone. Pot dolžine 1.
∧elt	Obratna pot (objekt proti predmetu).
(elt)	Skupina poti elt, oklepaji določajo prioriteto oz. prednost.
elt1 / elt2	Pot za elt1, kateri sledi elt2.
elt1 ∧ elt2	Krajšava za elt1 / ∧elt2, kjer obratni elt2 sledi elt1.
elt1 | elt2	Alternativna pot elt1 ali elt2 (obe možnosti se preverita).
elt*	Pot z 0 ali več pojavitvami elt.
elt+	Pot z 1 ali več pojavitvami elt.
elt?	Pot dolžine 0 ali 1 elt.
elt{n,m}	Pot med n in m pojavitev elt.
elt{n}	Točno n pojavitev elt oz. fiksna dolžina poti.
elt{n,}	n ali več pojavitev elt.
elt{,n}	Med 0 in n pojavitev elt.

Tabela 4.1: Sintaksa za property path

Ukaz Property path trojice v grafu obravnava kot ciklično usmerjen graf, zato v rezultatu včasih dobimo podvojene rezultate, ker se začetne točke v vzorcu ponovijo, ali pa iščemo rezultate spremenljivk, ki imajo isto vrednost. V resnici dobimo samo unikatne rezultate, ne glede na to, kateri regularni izraz uporabimo znotraj Property path ukaza [47].

Primer, kjer s FILTER ukazom določimo, da oseba z emailom janez@example ne more poznati samo sebe [44]:

```
SELECT * WHERE {
   ?x foaf:mbox mailto:janez@example.com .
   ?x foaf:knows/foaf:knows ?y .
   FILTER ( ?x != ?y )
   ?y foaf:name ?name
}
```

4.4 OPTIONAL

SPARQL vsebuje tudi napredne gradnike, ki nam omogočajo pisanje bolj kompleksnih poizvedb. V neki domeni podatkov imamo včasih shranjenih veliko podatkov, vendar pa vsi podatki nimajo vrednosti, kar pomeni, da če želimo narediti poizvedbo nad temi podatki in izpisati vse trojice, tudi tiste, ki nimajo nekaterih vrednosti, moramo to omogočiti in SPARQL nam ponuja rešitev z uporabo besede OPTIONAL. Brez uporabe te besede bi rezultat dobili le, če bi vse trojice v poizvedbi veljale.

Uporaba te besede je zelo pomembna, če želimo rezultat poizvedbe, ki vrne vrednost določene spremenljivke v primeru, da ta obstaja, v nasprotnem primeru pa izpiše ostale rezultate, vendar brez vrednosti določene spremenljivke v trojici. Torej če trojica znotraj OPTIONAL ne velja, se celotna rešitev veljavnih trojic ne bo zavrgla, vendar v rezultatu poizvedbe spre-

menljivka, ki je znotraj OPTIONAL stavka, ne bo imela vrednosti, ker ne obstaja oz. ni dodeljena. Znotraj ene poizvedbe imamo lahko več besed OPTIONAL, kar nam omogoča, da lahko izberemo več možnosti za določene spremenljivke, če želimo rezultat tudi brez vrednosti več kot ene spremenljivke. Obstaja tudi možnost, da si sami zakompliciramo poizvedbo z dodatnimi gradniki, kot je FILTER, ali gnezdenimi OPTIONAL stavki. Če v poizvedbi uporabimo besedo OPTIONAL za neko xy spremenljivko, poleg tega pa še besedo FILTER, kjer določimo pogoj, da mora ta xy spremenljivka imeti vrednost: FILTER(bound(?spremenljivka)), nam poizvedba ne bo vrnila želenega rezultata, če ta xy spremenljivka nima vrednosti, zato je potrebno biti v takšnih in podobnih primerih zelo pazljiv, katere ukaze povezujemo med seboj. Takšne malenkosti nas lahko stanejo napačnih rezultatov, zato je potrebno biti pozoren nanje. FILTER ukaz pa lahko uporabimo tudi znotraj ali zunaj ukaza OPTIONAL, kar nam omogoča, da dodamo pogoj na določeno spremenljivko v trojici in tako po želji omejimo rezultat poizvedbe [11].

Sintaksa

OPTIONAL { vzorec }

4.4.1 Uporaba ukaza FILTER v povezavi z ukazom OPTIONAL

Ločimo dve varianti uporabe ukaza FILTER v povezavi z ukazom OPTIONAL. Če uporabimo ukaz FILTER znotraj ukaza OPTIONAL, se ta pogoj v ukazu FILTER za določeno spremenljivko znotraj opcijskega dela nanaša samo na opcijski del, ki v primeru, da pogoj velja, izpiše vrednost te spremenljivke, v nasprotnem primeru pa se rezultat ne zavrže, kot se to zgodi v primeru, če uporabimo ukaz FILTER zunaj ukaza OPTIONAL, temveč samo ne izpiše vrednosti za določeno spremenljivko v vzorcu. Če pa uporabimo ukaz FILTER zunaj ukaza OPTIONAL, za spremenljivko znotraj

opcijskega dela, pa se ta ne upošteva kot opcijska rešitev. V tem primeru, če pogoj v FILTER za določeno spremenljivko ne velja, vendar ima le ta vrednost, se ne obnaša kot opcijski del in izpiše rezultat brez vrednosti te spremenljivke, temveč se rezultat vzorca zavrne in se ne izpiše [18]. Več o ukazu FILTER v naslednjem poglavju.

SPARQL nam omogoča, da lahko znotraj ukaza OPTIONAL vpišemo več trojic, ki nam lahko zakomplicirajo stvar tako, da če ne veljajo vse trojice, ne bomo dobili rezultata, ki smo ga želeli. Torej če želimo rezultat, kjer imamo več opcijskih spremenljivk, je smiselno oz. skoraj obvezno uporabiti ločene OPTIONAL ukaze znotraj SPARQL poizvedbe, ki posamezno obravnavajo vsako trojico. Tako bomo dobili želen rezulta bodisi imajo spremenljivke dodeljene vrednosti ali ne.

Primer 1:

```
SELECT ?country ?about ?long ?lat WHERE {
    ?country rdf:type yago:EuropeanCountries ;
             rdf:type dbpedia-owl:Country
    OPTIONAL {
        ?country rdfs:comment ?about1 .
        FILTER (lang(?about1) = "en")
    }
    OPTIONAL {
        ?country geo:lat ?lat ;
                 geo:long ?long .
        FILTER (?lat > 46)
    }
    BIND (str(?about1) AS ?about)
}
```

Primer 2:

```
SELECT ?country ?about ?lat ?long WHERE {
    ?country rdf:type yago:EuropeanCountries ;
             rdf:type dbpedia-owl:Country
    OPTIONAL {
        ?country rdfs:comment ?about1 .
        FILTER (lang(?about1) = "en")
    }
    OPTIONAL {
        ?country geo:lat ?lat ;
                 geo:long ?long .
    }
    BIND (str(?about1) AS ?about)
    FILTER (?lat > 46)
}
```

V prvem primeru imamo dva pogoja, kjer je vsak pogoj znotraj posameznega OPTIONAL bloka, kar pomeni, da tudi v primeru, če eden izmed pogojev ne velja, bomo dobili željeni rezultat, vendar v primeru, da eden izmed pogojev ne velja, se rezultat ne bo zavrgel, temveč bomo kot rezultat za spremeljivko ?about dobili prazno vrednost (če pogoj znotraj prvega OPTIONAL bloka ne velja), ali dve prazni vrednosti za spremenljivki ?lat ter ?long (če pogoj znotraj drugega OPTIONAL bloka ne velja).

V drugem primeru pa imamo isti pogoj, kjer mora biti geografska širina (spremenljivka ?lat) večja od 46, postavljen zunaj OPTIONAL ukaza, kar pomeni, da ta pogoj velja nad celotno skupino vzorcev, ter bomo rezultat dobili samo v primerih, kadar je ta pogoj izpolnjen.

Vrstni red ukazov OPTIONAL

Več OPTIONAL ukazov znotraj SPARQL poizvedbe je primerno uporabiti, če ena trojica ne velja. V tem primeru je smiselno uporabiti drugo trojico, da

dodelimo neki spremenljivki poljubno vrednost iz enega izmed OPTIONAL ukazov. Pri takšnem primeru uporabimo znotraj dveh ali več OPTIONAL ukazov isto spremenljivko. V naslednjem primeru smo uporabili dva OPTIONAL ukaza, kjer spremenljivki ?var1 dodelimo vrednost opisa glavnega mesta, če ta obstaja. Če pa ne obstaja, pa ji dodelimo vrednost opisa mesta. Torej v primeru, da procesor ne najde vrednosti za oznako glavnega mesta, bo spremenljivki ?var1 dodelil vrednost opisa glavnega mesta. SPARQL procesor preveri prvi OPTIONAL ukaz in če najde vrednost (oznaka glavnega mesta) le-to dodeli spremenljivki ?var1, v nasprotnem primeru pa spremenljivki ?var1 dodeli vrednost opisa glavnega mesta posamezne države. Če bi vrstni red ukazov OPTIONAL zamenjali, bi procesor najprej preveril, če ima neko glavno mesto opis, in v primeru, da bi ga imel, bi spremenljivka ?var na koncu poizvedbe imela vrednost opisa glavnega mesta. Vidimo, da je vrstni red pomemben. V poizvedbi imamo še ukaz FILTER, ki filtrira samo tista glavna mesta Evrope, ki imajo poleg oznake ali opisa še jezikovno značko "en". Na koncu poizvedbe pa imamo ukaz BIND, ki odstrani jezikovno značko spremenljivki ?var1 ter novo vrednost dodeli spremenljivki ?var.

Primer:

```
SELECT ?city ?var WHERE {
   ?city rdf:type <http://dbpedia.org/class/yago/CapitalsInEurope>.
   OPTIONAL { ?city rdfs:label ?var1. }
   OPTIONAL { ?city rdfs:comment ?var1. }
   FILTER (lang(?var1) = "en" )
   BIND (str(?var1) AS ?var)
}
```

Ukaz OPTIONAL lahko primerjamo z ukazom UNION, vendar je med njima vseeno dokaj velika razlika. OPTIONAL nam omogoča povečati možnost, da najdemo rezultat tudi v primeru, ko neka spremenljivka nima vrednosti,

UNION pa se uporablja predvsem za združevanje rezultatov dveh ali več množic [19].

Primerjava s SQL-om:

Ukaz OPTIONAL v SPARQL-u lahko primerjamo s SQL-ovim zunanjim stikom, ki ima možnost, da skupaj poveže dve ali več tabel, in enako kot SPARQL omogoča, da nekateri stolpci iz drugih tabel oz. podatki iz ene tabele nimajo vseh vrednosti določenih (bodisi imajo vrednost NULL ali pa prazen niz). Pri SQL-u je potrebno tabele med seboj povezati preko ključa, da dobimo podatke iz dveh ali več tabel, pri SPARQL-u pa z ukazom OPTIONAL določimo, za katere podatke želimo, da ne vplivajo na rezultat poizvedbe, če le-te nimajo vrednosti.

Sintaksa ukaza LEFT OUTER JOIN pri SQL-u:

```
SELECT stolpec1, ...
FROM tabela1
LEFT OUTER JOIN tabela2
ON tabela1.stolpec = tabela2.stolpec;
```

Primer:

```
SELECT n.Naslov, n.Vsebina, n.Povzetek, a.Ime, a.Priimek
FROM dbo.Novica n
LEFT OUTER JOIN dbo.Avtor a ON a.ID = n.Avtor
```

4.5 FILTER

Ukaz FILTER poveča možnosti za delo s SPARQL poizvedbami, kajti omogoča, da ga uporabimo za vse vrste pogojev, od pogojev, ki vključujejo števila, datume, razne funkcije, ki jih SPARQL vsebuje, do pogojev, kjer delamo z nizi,

jezikovnimi značkami in podobno. Obseg delovanja ukaza FILTER se nanaša na celotno skupino v kateri se pojavi. V primeru, da imamo v skupini več trojic ter ukaz FILTER, je za ta ukaz nepomemben vrstni red, na katerem se nahaja, tudi v primerih, kjer imamo več kot eno trojico. Znotraj skupine imamo lahko več kot en FILTER ukaz, vendar s tem ničesar ne pridobimo, kajti vsi FILTER ukazi so enakovredni, kot da bi vse pogoje zapisali znotraj enega FILTER ukaza, prednost je edino v tem, da je poizvedba lažje berljiva. Znotraj ukaza lahko uporabljamo logične izraze (&&, ||, !), matematične izraze (+, -, /, *) in operatorje za primerjavo (=, <, >, <=, >=, !=). Glede vrstnega reda rezultatov pa se moramo zavedati, da se ukaz FILTER vedno izvede na koncu [11].

V nadaljevanju bom opisal nekatere napredbe funkcije poizvedovalnega jezika SPARQL, ki omogočajo manipulacijo nad podatki znotraj ukaza FILTER.

4.5.1 Lang

V podatkih se zelo pogosto pojavlja zraven niza znak @ in za njim koda, ki predstavlja, za kateri jezik gre. Takšen način shranjevanja podatkov se uporablja, kadar želimo neko vrednost shraniti in pri tem določiti, kateri svetovni jezik smo uporabili.

Funkcija lang() znotraj ukaza FILTER nam omogoča, da filtriramo podatke glede na jezikovno značko določenega niza. V spodnjem primeru filtriramo glede na poljski jezik. Če bi želeli filtrirati po angleškem jeziku, bi uporabili jezikovno značko "en".

Primer:

```
#Sintaksa: FILTER (lang(?niz) = "jezikovnaZnacka")

SELECT ?city ?label WHERE {
   ?city rdf:type <http://dbpedia.org/class/yago/CapitalsInEurope> ;
         rdfs:label ?label
   FILTER (lang(?label) = "pl" )
}
```

V zgornjem primeru dobimo rezultat izpisan tako, da bo na koncu nekega niza jezikovna značka @pl. Pogosto tega ne želimo, zato se lahko tega dela znebimo z ukazom BIND(str(?label) AS ?izpisiNiz), tako da ga dodamo za ukaz FILTER (lang(?label) = "pl") in tako bomo dobili rezultat brez jezikovne značke. V spodnjem primeru filtriramo po angleškem jeziku, v SELECT stavku pa smo definirali izpis obeh spremenljivk, tako tiste z jezikovno značko kot brez.

```
SELECT ?city ?label ?label1 WHERE {
   ?city rdf:type <http://dbpedia.org/class/yago/CapitalsInEurope> ;
         rdfs:label ?label
   FILTER (lang(?label) = "en" )
   BIND (str(?label) AS ?label1)
}
```

Če imamo podatek shranjen v obliki @en-GB, rezultata z zgornjim primerom ne bomo dobili. Zato nam SPARQL omogoča funkcijo langMatches(), ki sprejme dva parametra, spremenljivko, katero obdamo z funkcijo lang(), ter drugi parameter, ki je niz in predstavlja jezik [11].

```
SELECT ?city ?label ?label1 WHERE {
   ?city rdf:type <http://dbpedia.org/class/yago/CapitalsInEurope> ;
```

SPARQL results:

city	label	label1
:Amsterdam	"Amsterdam"@en	"Amsterdam"
:Andorra_la_Vella	"Andorra la Vella"@en	"Andorra la Vella"
:Bratislava	"Bratislava"@en	"Bratislava"
:Bucharest	"Bucharest"@en	"Bucharest"
:City_of_San_Marino	"City of San Marino"@en	"City of San Marino"

Slika 4.1: Izpis podatkov z uporabo funkcije lang()

```
        rdfs:label ?label
    FILTER (langMatches(lang(?label), "en"))
    BIND (str(?label) AS ?label1)
}
```

V tem primeru bomo dobili rezultat v vseh različicah angleškega jezika. Vendar pa bo zraven še vedno značka, ki predstavlja, za kateri jezik gre, zato smo uporabili še funkcijo str(), ki odstrani jezikovno značko. Uporaba funkcije langMatches() je primerna za iskanje po podatkih, za katere nismo prepričani, kakšno jezikovno oznako imajo. Funkcija langMatches lahko sprejme kot drugi parameter znak '*', ki nam omogoča, da izpišemo vse vrednosti, ki imajo določeno jezikovno značko za nek jezik [11].

Primer:

FILTER (langMatches(lang(?niz), "*"))

4.5.2 Bound

Funkcija bound deluje tako, da vrne true (resnično vrednost), če je spremenljivki dodeljena neka vrednost, v nasprotnem primeru vrne false (neresnično vrednost). Pogoj lahko tudi negiramo z uporabo klicaja - !bound(?spremenljivka). V prvi verziji SPARQL-a 1.0 je bila negacija ukaza bound uporabljena v povezavi z ukazoma OPTIONAL ter FILTER, da smo

dosegli tako imenovano Negation by failure. Negation by failure se uporablja, da preverimo, ali neka trojica obstaja ali ne. V verziji SPARQL-a 1.1 sta bila dodana še ukaza NOT EXIST ter MINUS, ki tudi podpirata negacijo. Več o tem v naslednjem odstavku [11].

Primer:

```
SELECT * WHERE {
   ?city rdf:type yago:CapitalsInEurope ;
         dbpprop:populationTotal ?population
   FILTER(bound(?population))
}
ORDER BY DESC(?population)
```

4.5.3 NOT EXIST / MINUS

V novi verziji SPARQL-a 1.1 imamo za negacijo dva načina. Prvi način je uporaba ukaza NOT EXIST v povezavi z ukazom FILTER, ki deluje na podlagi filtriranja podatkov in vrne true (resnično vrednost), če v podatkih za določen vzorec ni rezultata. Nasprotni ukaz, ki ga lahko uporabimo, je ukaz EXIST, ki vrne true (resnično vrednost), če najdemo prisotnost vzorca v podatkih [11].

Sintaksa

FILTER (NOT EXIST { vzorec })

V naslednjem primeru izpišemo vsa tista glavna mesta Evrope, ki nimajo predikata dbpprop:populationTotal. Če bi želeli izpisati vsa tista mesta, ki pa imajo ta predikat, pa bi uporabili isto poizvedbe, brez besede NOT znotraj FILTER ukaza.

Primer:

```
SELECT * WHERE {
   ?city rdf:type yago:CapitalsInEurope
   FILTER(
       NOT EXISTS {
          ?city dbpprop:populationTotal ?population
       }
   )
}
```

V naslednjem primeru imamo uporabo ukaza EXIST, ki najde vsa tista glavna mesta Evrope, ki imajo predikat dbpprop:populationTotal, ter imajo več kot 5.000.000 prebivalcev.

Primer:

```
SELECT * WHERE {
   ?city rdf:type yago:CapitalsInEurope
   FILTER(
       EXISTS {
          ?city dbpprop:populationTotal ?population
          FILTER(?population > 5000000)
       }
   )
}
```

Drugi način uporabe negacije pa je ukaz MINUS in deluje tako, da vrne rešitev, ki na levi strani ukaza MINUS ni kompatibilna z rezultatom na desni strani ukaza [11].

Sintaksa

{vzorec1} MINUS {vzorec2}

Na splošno povedano ukaz MINUS odstrani tiste vrednosti z leve strani ukaza, ki imajo isto vrednost kot tiste, ki so veljavne oz. izvršljive na desni strani ukaza.

Primer:

```
SELECT * WHERE {
   ?city rdf:type yago:CapitalsInEurope ;
         rdfs:label ?label
   FILTER(lang(?label) = "en")
   MINUS
   {
      ?city dbpprop:populationEstimate ?population
   }
}
```

Zgornji primer najde vsa glavna mesta Evrope, ki imajo jezikovno značko ”en”, pri predikatu rdfs:label. Nato pa sledi ukaz MINUS, ki iz množice podatkov, ki jih dobimo z prvimi tremi vrsticami znotraj WHERE bloka, odstrani vse tiste, ki imajo predikat dbpprop:populationEstimate.

Primerjava s SQL:

Tudi SQL pozna ukaz MINUS, ki se izvede nad dvema SQL stavkoma. Ta ukaz vrne rezultat glede na prvi stavek in iz njega odstrani vrednosti, ki se pojavijo v rezultatu drugega stavka in so enake vrednostim iz prvega. Pomembno je povedati, da MINUS ukaz pri SQL-u ne vrača podvojenih vrednosti ter da morata oba SQL stavka imeti enako število stolpcev, ki jih vračamo, stolpci pa morajo uporabljati enake podatkovne tipe. Pri SQL-u je pomembno, katero bazo uporabljamo, kajti vse ne podpirajo ukaza MI-

NUS. Primer takšnega ponudnika relacijskih baz je tudi Miscrosoft MySQL Server, ki smo ga tudi mi uporabljali za pisanje SQL poizvedb. Microsoft MySQL Server ne pozna ukaza MINUS, temveč uporablja ukaz EXCEPT, ki ima isto vlogo kot MINUS, in tudi ta ne vrača podvojenih vrednosti. Tudi tukaj poznamo ukaz EXCEPT ALL, ki odstrani duplikate, vendar je le-ta implementiran samo pri IBM-ovem DB2 Universal serverju [14].

Primer:

```
SELECT n.Naslov
FROM dbo.Novica n
EXCEPT
SELECT n.Naslov
FROM dbo.Novica n
WHERE n.Naslov LIKE 'Naslov 1'
```

4.5.4 Regex

Regex se uporablja za iskanje nizov v vrednostih spremenljivke predmeta ali objekta, ki je tipa string (niz). S primerjavo dveh nizov lahko povečamo možnosti za iskanje po podatkih z regularnimi izrazi. Funkcija regex sprejme tri parametre, spremenljivko tipa string (niz), regularni izraz, katerega primerjamo s prvim parametrom, ter tretji parameter, ki je zastavica.

Regex lahko sprejme eno izmed štirih zastavic, ki je bodisi 's', 'm', 'i' ali 'x'. Zastavica 's' se uporablja za primerjavo nizov v pika-načinu, kjer pika predstavlja kateri koli znak v tekstu, ki ga primerjamo, razen znaka (#x0A), ki predstavlja novo vrstico. Zastavica 'm' se uporablja za delovanje v večvrstičnem načinu, kjer se znak $\wedge$ uporablja za začetek katere koli vrstice, $ pa konec katere koli vrstice. Vsaka vrstica predstavlja svoj niz in se lahko konča samo z znakom #x0A. Zastavica 'i' se uporablja za delovanje v načinu, ki ni odvisen samo od malih ali samo velikih črk (case-insensitive). Torej če damo primer izraza 'a', se bo le-ta ujemal tako z 'a' kot z 'A'. Primer izraza

[A-Z] se bo ujemal tako z [a-z] kot z [A-Z]. Zastavica 'x' pa se uporablja, da iz regularnega izraza umaknemo prazne znake (#x9, #xA, #xD, #x20). Če ne želimo uporabiti nobene od zgoraj navedenih zastavic, pa pustimo tretji parameter prazen [51].

Sintaksa

Regex(?x, "niz" [, "zastavice"])

V naslednjem primeru, z uporabo funkcije regex iščemo po nizu za vrednost objekta, ki se nanaša na predikat rdfs:label neke države in vsebuje niz, ki se konča z "nia". Zastavica "i" pa pove, da ni pomembno, ali niz vsebuje male ali velike črke.

Primer 1:

```
SELECT ?a ?countryName WHERE {
   ?a a <http://dbpedia.org/ontology/Country> ;
      rdfs:label ?label
   FILTER(lang(?label) = "en" && regex(?label, "nia$" ,"i"))
   BIND (str(?label) AS ?countryName)
}
```

V naslednjem primeru poizvedujemo iz množice glasbenih skupin, kjer želimo najti glasbeno skupino U2 z uporabo funkcije regex. Najprej zagotovimo, da je opis skupine predstavljen z jezikovno značko "en", ter z uporabo funkcije regex določimo, da mora niz, ki je shranjen v spremenljivki ?label, vsebovati niz "U2". V tem primeru nismo uporabili tretjega parametra, zato je ta primer brez zastavice, kar pomeni, da mora niz vsebovati točno takšen niz znakov, kot ga vpišemo znotraj drugega parametra, torej "U2". Če bi namesto velike črke 'U' napisali malo črko 'u', rezultata ne bi dobili, zato bi bilo v tem primeru potrebno kot tretji parameter vstaviti zastavico "i".

Primer 2:

```
SELECT ?subject ?bandName WHERE {
    ?subject rdf:type <http://dbpedia.org/ontology/Band> ;
             rdfs:label ?label
    FILTER (regex(str(?label), "U2") && lang(?label) = "en")
    BIND (str(?label) AS ?bandName)
}
```

Funkija regex omogoča še dodatne možnosti za delo z nizi, vendar pa moramo paziti, da kot prvi parameter podamo vrednost, ki je tipa xsd:string ali pa navaden niz brez jezikovne značke. Zato je vedno pametno uporabiti funkcijo str(), ki zagotovi, da je prvi parameter tipa xsd:string. V regex funkciji lahko uporabimo še dodatne znake, kot sta \d, ki predstavlja števko, ter \s, ki predstavlja prazen znak. Uporaba pike v regex funkciji pa omogoča, da določimo, koliko znakov le-ta predstavlja [11].

- .* predstavlja nič ali več znakov
- .+ predstavlja enga ali več znakov
- .? predstavlja nič ali en znak
- .{2} predstavlja točno dva znaka

Primer 3:

```
SELECT ?a ?countryName WHERE {
   ?a a <http://dbpedia.org/ontology/Country> ;
      rdfs:label ?label
   FILTER(lang(?label) = "en" && regex(?label, "^Sl.{2}e.?" ,"i"))
   BIND (str(?label) AS ?countryName)
}
```

V primeru 3 smo uporabili regex funkcijo, ki preveri, ali se niz v spremenljivki ?label začne na črki Sl, nato sledita dva poljubna znaka, za njima

sledi črka e ter nato še en ali noben poljuben znak.

SPARQL omogoča, da lahko regularne izraze izvajamo ne samo nad nizi, temveč tudi nad URI-naslovi. To omogočimo tako, da kot prvi parameter podamo spremenljivko predmeta, ki predstavlja URI naslov, in to pretvorimo v string (niz) z ukazom str(?spremenljivka). Ta način uporabimo tudi v primeru, če želimo zagotoviti, da je parameter res tipa string (niz) [11]. V primeru 2 bi dobili rezultat, tudi če bi kot prvi parameter funkcije regex podali spremenljivko ?subject, ki predstavlja URI naslov.

Primerjava s SQL-om:

Funkcijo regex znotraj ukaza FILTER lahko primerjamo z ukazom LIKE v SQL-u, ki nam omogoča, da lahko znotraj stavka WHERE iščemo po določenih vzorcih nizov v posameznem stolpcu znotraj tabele. Znotraj niza lahko vstavimo dva znaka: (_, %) ter s tem povečamo verjetnost, da najdemo poljuben niz. Podčrtaj(_) predstavlja natanko en poljuben znak, odstotek(%) pa predstavlja nič ali več poljubnih znakov. Ukaz LIKE pa omogoča tudi uporabo ostali regularnih izrazov, prav tako kot SPARQL [14].

Primer iskanja telefonskih številk, ki se začnejo z 051 ali 041, pri čemer morajo številke, ki se začnejo z 051, biti v pravem formatu (vsebovati vezaje, zadnja številka pa mora biti 1):

```
SELECT ime, priimek, telefonskaSt
FROM imenik
WHERE telefonskaSt LIKE '051-___-__1' OR telefonskaSt LIKE '041%';
```

Primer iskanja oseb, kjer se ime osebe začne na črko A, B ali D:

```
SELECT ime, priimek, telefonskaSt
FROM imenik
WHERE ime LIKE '[ABD]%';
```

4.5.5 NOT IN ter IN

Zelo uporabna ukaza znotraj ukaza FILTER sta IN ter NOT IN, ki nam omogočata, da neka spremenljivka obsega ali ne obsega vrednosti, ki jih podamo znotraj teh dveh ukazov. Ukaza se lahko uporabljata bodisi za števila, datume itd. ter delujeta na isti način kot ukaza IN ter NOT IN iz jezika SQL. Vendar pri SQL-u lahko ta dva ukaza uporabljamo še v primerih, kjer znotraj IN ali NOT IN napišemo novo poizvedbo ter s tem pridobimo podatke kot množico, s katero potem primerjamo poljuben stolpec.

Primer uporabe SPARQL poizvedbe:

```
SELECT * WHERE {
   ?s a sioc:Post ;
      dct:title ?title ;
      news:statistics/news:nRatings ?nRat
      FILTER(?nRat IN (10, 20))
}
ORDER BY DESC(?nRat)
```

Primer SQL poizvedbe:

```
SELECT ime, priimek, krajRojstva
FROM avtorji
WHERE emp_id IN (SELECT emp_id
FROM avtorji
```

```
WHERE status LIKE 'Urednik');
```

Primer uporabe ukaza FILTER, ki vsebuje pogoj tako nad števili, datumom kot nad regularnim izrazom za določen niz:

```
SELECT * WHERE {
    ?s a sioc:Post ;
       dct:title ?title ;
       dct:created ?created ;
       news:statistics/news:nRatings ?nRat
    FILTER(?created > xsd:dateTime("2013-02-01T00:00:00Z") &&
          ?nRat > 5 && regex(?title, "Janez", "i"))
}
ORDER BY DESC(?nRat)
```

4.5.6 Več možnosti za delo z ukazom FILTER

Ukaz FILTER se uporablja kot dodatni pogoj v poizvedbi in vrne kot rezultat vrednost true (resnično) ali false (neresnično). Torej ga lahko uporabljamo za razne načine, lahko ga uporabljamo v povezavi z nizi, kot smo prej opisali uporabo funkcije regex(), lahko ga uporabljamo za delo z datumi, števili, aritmetičnimi izrazi, lahko pa ga preprosto uporabljamo z vsemi funkcijami, ki jih ponuja SPARQL 1.1, kot so funkcije za delo z nizi, števili itd. V naslednjem primeru bomo prikazali delovanje le teh.

Primer:

```
SELECT DISTINCT ?longName ?longNameNoLangTag ?population ?abbr
      ?longNameLen ?abbrLCase ?abbrUCase ?longNameSubStr
      ?longNameStarts ?populationContains ?strConcat
WHERE {
   ?slo a ?o ;
        dbpedia-owl:longName ?longName ;
        dbpprop:populationEstimate ?population ;
        dbpprop:vehicleCode ?abbr
   BIND (str(?longName) AS ?longNameNoLangTag)
   FILTER (str(?slo) = "http://dbpedia.org/resource/Slovenia")
   FILTER (isUri(?slo))
   FILTER (isNumeric(?population))
   FILTER (isLiteral(?longName))
   FILTER (!(isNumeric(?abbr)))
   BIND (STRLEN(?longName) AS ?longNameLen)
   BIND (LCASE(str(?abbr )) AS ?abbrLCase)
   BIND (UCASE(str(?abbr )) AS ?abbrUCase)
   BIND (SUBSTR(str(?longName), 3, 3) AS ?longNameSubStr)
   BIND (STRSTARTS(str(?population), "205") AS ?longNameStarts)
   BIND (CONTAINS(str(?population), "2") AS ?populationContains)
   BIND (CONCAT("Janez", " Novak, ", 1000, " Ljubljana") AS
        ?strConcat)
}
```

V rezultatu (Tabela 4.3) imamo primer delovanja funkcije str(), ki odstrani jezikovno značko. Funkcija str() ima možnost, da pretvori vrednost spremenljivke v niz, če že le-ta ni v tej obliki. V rezultatu (Tabela 4.4) je prikazan način delovanja funkcij STRLEN, LCASE, UCASE ter SUBSTR. Izpostavili pa bi funkcijo STRLEN, ki vrne dolžino niza, vendar pri tem ne upošteva jezikovne značke, če ta obstaja. V rezultatu (Tabela 4.5) vidimo, da imata prvi dve spremenljivki vrednost 1, kar pomeni, da sta funkciji STR-

STARTS ter CONTAINS vrnili resnično vrednost (true ali 1). V tem primeru je pogoj znotraj funkcij veljal. V nasprotnem primeru pa bi kot rezultat dobili false ali 0. V tabeli 4.5 vidimo tudi primer uporabe ukaza CONCAT, ki lahko skupaj združi poljubno število nizov, katere ločimo z vejico.

str	Pretvori vrednost parametra v niz ter odstrani jezikovno značko.
isUri	Preveri, če je vrednost prvega parametra URI naslov.
isNumeric	Preveri, če je vrednost prvega parametra število.
isLiteral	Preveri, če je vrednost prvega parametra niz.
STRLEN	Vrne število znakov prvega parametra (jezikovne značke ne šteje).
LCASE	Pretvori vrednost prvega parametra v male črke.
UCASE	Pretvori vrednost prvega parametra v velike črke.
SUBSTR	Vrne podniz znakov iz prvega parametra.
STRSTARTS	Preveri, če se vrednost prvega parametra začne s poljubnim nizom znakov. Drugi parameter predstavlja začetek podniza, tretji parameter pa dolžino podniza.
CONTAINS	Preveri, če vrednost prvega parametra vsebuje določen znak ali niz znakov.

Tabela 4.2: Funkcije za delo z nizi

longName	longNameNoLangTag	population	abbr
"Republic of Slovenia"@en	Republica of Slovenia	2055496	"SLO"@en

Tabela 4.3: Prikaz rezultata poizvedbe - del 1

longNameLen	abbrLCase	abbrUCase	longNameSubStr
20	slo	SLO	pub

Tabela 4.4: Prikaz rezultata poizvedbe - del 2

longNameStarts	populationContains	strConcat
1	1	Janez Novak, 1000 Ljubljana

Tabela 4.5: Prikaz rezultata poizvedbe - del 3

4.6 UNION

UNION je binarni operator in vsi vzorci, ki so v sklopu ukaza UNION, se izvedejo ločeno. Rezultat pa dobimo v primeru, če se vsaj eden izmed podvzorcev izvede, torej da nek rezultat [20].

Za primerjavo lahko vzamemo dva navadna SELECT ukaza, kjer vsak izpiše podatke iz svoje množice. Razlikujeta se le v predikatu, ki predstavlja opis države. Ta dva ukaza lahko preprosto združimo z ukazom UNION, da vrneta isti rezultat tako, da njuno vsebino znotraj posamičnega WHERE bloka združimo z ukazom UNION.

```
#Primer 1
SELECT * WHERE {
   ?country rdf:type <http://dbpedia.org/ontology/Country> .
   ?country rdfs:label "Slovenia"@en
}
SELECT * WHERE {
   ?country rdf:type <http://dbpedia.org/ontology/Country> .
   ?country rdfs:label "Italy"@en
}

#Primer 2
SELECT * WHERE {
    {
       ?country1 rdf:type <http://dbpedia.org/ontology/Country> ;
                 rdfs:label "Slovenia"@en
    }
    UNION {
       ?country2 rdf:type <http://dbpedia.org/ontology/Country> ;
                 rdfs:label "Italy"@en
    }
}
```

Ukaz UNION nam omogoča, da lahko združimo več vzorcev znotraj enega WHERE stavka, tako da vzorce znotraj { } ločimo z besedo UNION. Vsak vzorec pa lahko predstavlja samo eno trojico ali pa bolj kompleksne vzorce, ki vsebujejo tudi ostale gradnike, kot so OPTIONAL, FILTER ali celo poizvedbe. Vzorci znotraj ukaza UNION delujejo enako kot vsi ostali vzorci. V primeru, da ena trojica znotraj vzorca ne drži, potem celoten vzorec za določen primer ne bo dal rezultata, temveč bodo v rezultatu prazne vrednosti za spremenljivke, ki se nahajajo znotraj tega vzorca [11].

Sintaksa:

{vzorec1} UNION {vzorec2}

Primer:

```
SELECT ?drzava1 ?populationSlovenia ?drzava2 ?populationAustria
   ?drzava3 ?populationItaly
WHERE {
   {
      ?drzava1 rdf:type <http://dbpedia.org/ontology/Country> ;
         rdfs:label ?label ;
         dbpprop:populationEstimate ?populationSlovenia
      FILTER(
         regex(?label, "Slovenia", "i") && lang(?label) = "en"
      )
   }
   UNION
   {
      ?drzava2 rdf:type <http://dbpedia.org/ontology/Country> ;
         rdfs:label ?label2 ;
         dbpprop:populationEstimate ?populationAustria
      FILTER(
         regex(?label2, "Austria", "i") && lang(?label2) = "en"
      )
```

```
    }
    UNION
    {
       ?drzava3 rdf:type <http://dbpedia.org/ontology/Country> ;
          rdfs:label ?label3 ;
          dbpprop:populationEstimate ?populationItaly
       FILTER(
          regex(?label3, "Italy", "i") && lang(?label3) = "en"
       )
    }
}
```

Ukaz UNION deluje kot disjunkcija med dvema vzorcema, ki vrne rezultat v primeru, če je vsaj eden izmed njiju resničen. Pri pisanju kompleksnih vzorcev ukaza UNION se lahko rešimo ponavljanja dela kode, ki bi se ponavljal v dveh ali več vzorcih, tako da ta del napišemo samo enkrat izven vseh vzorcev UNION in se s tem rešimo nepotrebne redundance [11].

Primerjava s SQL-om:

Če primerjamo SPARQL-ov ukaz UNION z SQL-ovim ukazom UNION, sta si na videz zelo podobna, vendar je pa razlika v delovanju in uporabi precejšnja. Kot smo videli pri SPARQL-u, lahko poizvedujemo katere koli podatke in jih združujemo, pri SQL-u pa je namen tega ukaza poizvedovanje po podatkih, kjer kombiniramo podatke iz dveh ali več tabel z uporabo SELECT stavkov, katere ločimo bodisi z UNION ali UNOIN ALL ukazom. Težava pri SQL-ovem ukazu je v tem, da morajo imena stolpcev biti enaka, ter morajo imeti isti tip (npr. varchar, int, datetime). Število stolpcev, ki jih izpisujemo, mora biti enako v vseh SELECT stavkih in vsi stolpci morajo imeti enako ime. Pri SPARQL-u ni pomembno, koliko spremenljivk pridobimo iz posameznega UNION bloka, to število je od posameznega UNION bloka poljubno. Prav tako ni pomembno, ali je neka spremenljivka tipa niz, druga pa tipa datum. SQL-ov ukaz ne vrača podvojenih vrednosti, vendar pa lahko to v SQL-u

omogočimo z ukazom UNION ALL [14].

Sintaksa SQL-ovega ukaza UNION:

```
SELECT [stolpec_i] FROM tabela1
UNION / UNION ALL
SELECT [stolpec_i] FROM tabela2
```

Primer:

```
SELECT a.ID AS "Stolpec ID", a.Ime + a.Priimek AS "Stolpec naziv"
FROM dbo.Avtor a
UNION ALL
SELECT n.ID AS "Stolpec ID", n.Naslov AS "Stolpec naziv"
FROM dbo.Novica n
```

4.7 BIND - Dodeljevanje vrednosti

Ukaz se je pojavil v novi verziji SPARQL 1.1 in je zelo pogost pri aritmetičnih izrazih, kjer rezultat nekega izraza shranimo v poljubno spremenljivko, katero lahko v poizvedbi uporabljamo bodisi v pogojih ali pa jo izpišemo kot rezultat. Uporaba ukaza ni omejena samo na aritmetične izraze, temveč ga lahko uporabimo za vse načine, kjer želimo dodeliti vrednost neki poljubni spremenljivki. Pogosto ta ukaz uporabljamo, če želimo iz poizvedbe izvleči še dodatne izračune, predvsem pri številčnih vrednostih. Uporabimo ga nekje v vzorcu znotraj WHERE bloka [11]. V primeru, da uporabljamo ukaz BIND za neko spremenljivko, ki se pojavi znotraj FILTER ukaza, ni pomembno, na katero mesto postavimo BIND ukaz, bodisi pred ali za FILTER ukaz, saj kot vemo, FILTER deluje nad celotno skupino vzorca, v katerem se pojavi, ter se izvede na koncu poizvedbe [22].

Sintaksa ukaza BIND:

BIND (izraz AS ?spremenljivka)

Primer:

```
SELECT ?s ?x ?x3
WHERE {
   ?s a sioc:Post .
   ?s news:statistics/news:avgRating ?x .
   FILTER (?x3 > 11)
   BIND (?x*3 AS ?x3)
}
ORDER BY DESC(?x3)
```

BIND je ukaz, ki ga uvrščamo v sklop ukazov za dodeljevanje vrednosti poljubnim spremenljivkam. V ta sklop spadata še dva načina. Prvi način je dodeljevanje vrednosti spremenljivkam znotraj SELECT stavka, kjer lahko neki spremenljivki dodelimo poljubno vrednost, ki lahko izhaja iz uporabljenih spremenljivk v poizvedbi, ali pa si izmislimo neke nove vrednosti. Znotraj SELECT bloka, kjer izpisujemo spremenljivke, lahko uporabimo več primerov dodeljevanja [23].

Sintaksa dodeljevanja v SELECT stavku:

SELECT ... (izraz AS ?spremenljivka) WHERE { }

Primer:

```
SELECT ?s ?x (?x*2 AS ?x2) (strlen(?content) AS ?contChar)
WHERE {
   ?s a sioc:Post .
   ?s news:statistics/news:avgRating ?x .
   ?s sioc:content ?content
}
```

V zgornjem primeru imamo način dodeljevanja vrednosti spremenljivkam znotraj SELECT stavka, kar pomeni, da pogoj znotraj WHERE bloka nad spremenljivko ?contChar nebi deloval. Zato je v primerih, kjer želimo nad neko spremenljivko, ki jo sami ustvarimo, zaželjeno uporabiti ukaz BIND, ter znotraj njega definirati to novo spremenljivko in nad njo izvesti pogoj.

Primer:

```
SELECT ?s ?x (?x*2 AS ?x2) ?contChar
WHERE {
   ?s a sioc:Post ;
      news:statistics/news:avgRating ?x ;
      sioc:content ?content
   BIND (strlen(?content) AS ?contChar)
   FILTER (?contChar > 1000)
}
ORDER BY DESC(?contChar)
```

Drugi način pa je podoben prvemu, vendar tukaj uporabimo izraz v sklopu agregarnih funkcij, kjer je rezultat posledica grupiranja podatkov. Verzija SPARQL 1.1 pozna sedem vrst agregarnih funkcij: SUM, AVG, MIN, MAX, COUNT, GROUP_CONCAT, SAMPLE, le-te pa se uporabljajo v povezavi z ukazom GROUP BY, ki grupira rezultate glede na spremenljivko, ki jo uporabimo za grupiranje. V povezavi z grupiranjem imamo ukaz HAVING,

ki deluje podobno kot ukaz FILTER, le da se ta uporablja samo za agregarne funkcije. Dodeljevanje vrednosti spremenljivkam, ki izhajajo iz agregarnih funkcij, pa uporabimo tako, da znotraj funkcije uporabimo spremenljivko, iz katere želimo dobiti rezultat ter ga dodeliti poljubni spremenljivki [24].

Sintaksa uporabe agregarnih funkcij znotraj poizvedbe:

```
SELECT ... (AGR_FUNK(?spremljivka1) AS ?rez)
WHERE { ... }
GROUP BY ?spremenljivka2
[HAVING (pogoj)]
```

Primer 1:

```
SELECT (SUM(?nVot) AS ?nVotes)
WHERE {
  ?comm rdf:type news:Comment .
  ?comm news:nVotings ?nVot
}
GROUP BY ?comm
HAVING (SUM(?nVot) > 10)
ORDER BY DESC(?nVotes)
```

Primer 2:

```
SELECT (SUM(?nVot) AS ?nVotings) ?post
WHERE {
  ?post a sioc:Post .
  ?post sioc:has_reply ?comm .
  ?comm news:nVotings ?nVot
}
GROUP BY ?post
```

```
HAVING (?nVotings > 25000)
ORDER BY DESC(?nVotings)
```

Primerjava s SQL-om:

SQL pozna podoben ukaz, ki se imenuje AS in se v primerjavi s SPARQL-om uporablja, da tabelam in stolpcem dodelimo poljubna imena, ki se hranijo samo za čas izvajanja poizvedbe. Začasna imena tabel se uporabljajo predvsem takrat, ko uporabljamo povezave med tabelami ter izpisujemo več podatkov iz različnih tabel, kjer so lahko tudi nekateri stolpci enako poimenovani. Začasna imena stolpcev pa se uporabljajo predvsem zaradi lepšega in preglednejšega izpisa nazivov le teh [25]. Učinkovitost obeh jezikov pri dodeljevanju vrednosti je dokaj enaka, vendar se SQL osredotoča predvsem na poimenovanje stolpcev ter tabel, katere povezujemo med seboj ipd., SPARQL pa se osredotoča predvsem na dodeljevanje vrednosti poljubnim spremenljivkam, s katerimi delamo (računamo, spreminjamo vrednosti, ipd.) v sklopu poizvedbe ali pri izpisu spremenljivk.

Sintaksa ukaza AS pri SQL-u:

```
SELECT me.ID AS "Dogodek ID", upo.id AS "Uporabnik ID"
FROM dbo.MailingEvent me, dbo.Uporabnik upo
WHERE me.UserID = upo.ID
```

4.8 SERVICE – Porazdeljene poizvedbe

Uporaba oddaljenega dostopa do nekega vira oz. storitve z uporabo besede SERVICE se je pojavila v novi verziji SPARQL 1.1 in nam omogoča, da lahko znotraj poizvedbe preko SPARQL protokola kličemo eno ali več poljubnih oddaljenih storitev, ki lahko sprejmejo in izvedejo SPARQL poizvedbe. Najpogosteje uporabljamo ta način za združevanje podatkov, kjer preko pro-

tokola pošljemo zahtevo za SPARQL poizvedbo neki oddaljeni točki, do katere želimo dostopati, in kot rezultat dobimo podatke v oblikah RDF/XML, JSON, XML, HTML, NTriples, CSV ipd. [11].

Dandanes poznamo že kar nekaj znanih oddaljenih točk, kot so DBpedia, BBC oddaljene točke, Linked Movie Database in ostala. Več na spletni strani: [26].

Pri poizvedovanju z ukazom SERVICE lahko pride do težav pri dostopu zaradi raznih okoliščin, na katere ne moremo vplivati, kot je izpad serverja, vzdrževalna dela ipd. Lahko pa vplivamo na to, da ne pišemo takšnih poizvedb, kjer bi želeli preiskati celotno bazo ali nek ogromen kos podatkov, kar bi vzelo ogromno časa in bi pri tem verjetno dobili opozorilo za napako, da je čas zahtevka potekel. Zato se pri pisanju poizvedb za oddaljen dostop osredotočimo predvsem na količino podatkov, ki jih trenutno potrebujemo. Celoten SERVICE blok lahko vstavimo v ukaz OPTIONAL, če želimo dobiti rezultat tudi v primerih, kjer določena spremenljivka nima vrednosti. Pogosto to prav pride, kadar poizvedujemo iz več kot ene oddaljene SPARQL točke in nismo prepričani, da imajo vsi željeni podatki nastavljene vrednosti. Pri uporabi ukaza SERVICE, znotraj začetnega SELECT ukaza definiramo katere spremenljivke oz. podatke bomo izpisali, te spremenljivke pa se morajo nato pojaviti v jedru poizvedbe (blok WHERE), torej bodisi znotraj ukaza SERVICE, ali pa izven, kar pomeni, da se nanašajo na lokalni graf, po katerem poizvedujemo. Spremenljivke, definirane znotraj ukaza SERVICE, so podatki, ki jih želimo pridobiti iz neke oddaljene RDF množice podatkov s porazdeljeno poizvedbo. Znotraj poizvedbe imamo lahko zelo preproste trojice, lahko pa pišemo tudi kompleksne poizvedbe, ki poleg osnovnih spremenljivk vračajo še posebne izračune. Lahko povezujemo podatke iz več kot enega oddaljenega vira z uporabo več SERVICE ukazov hkrati ipd [11].

Ukaz SERVICE se uporabi znotraj glavnega bloka poizvedbe in preko URI naslova dostopa do oddaljene točke, kjer se nahajajo podatki, do katerih želimo dostopati. Znotraj SERVICE bloka pa pišemo poljubne vzorce glede na želeni rezultat ter pri tem uporabljamo ukaze in funkcije, ki jih podpira

poizvedovalni jezik SPARQL 1.1.

Sintaksa:

```
SELECT ...
WHERE{
    SERVICE <uri> { ... }
}
```

Primer:

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX dbpedia-owl: <http://dbpedia.org/ontology/>
PREFIX dbpprop: <http://dbpedia.org/property/>
SELECT * WHERE {
   ?post a sioc:Post .
   ?post news:location ?x .
   FILTER (str(?x) = "Ljubljana")
   SERVICE <http://dbpedia.org/sparql> {
      ?city foaf:name "Ljubljana"@en .
      ?city dbpedia-owl:country ?country .
      ?city dbpprop:leaderName ?leaderName
   }
}
```

4.9 GRAPH

Pri kompleksnih poizvedbah oz. poizvedbah, kjer poizvedujemo po večjem številu grafov, uporabljamo ukaze FROM, FROM NAMED ter GRAPH. Ti trije ukazi so med seboj tesno povezani. Prvi način je uporaba ukaza FROM, kjer lahko navedemo poljubno število grafov, ti pa se združijo v privzeti graf.

Drugi način je uporaba imenovanih grafov, katere določimo z IRI naslovom ali imenom datoteke znotraj ukaza FROM NAMED, ter z ukazom GRAPH določimo, kateremu grafu mora vzorec znotraj ukaza ustrezati [27].

Sintaksa ukaza FROM:

```
SELECT ...
FROM <...>
WHERE { ... }
```

Sintaksa ukaza FROM NAMED ter GRAPH:

```
SELECT ...
FROM NAMED <...>
WHERE {
   GRAPH <...> { vzorec }
}
```

Število FROM ukazov, s katerimi definiramo grafe, je poljubno. Vsi definirani grafi se združijo v privzeti graf, po katerem lahko potem poizvedujemo. Znotraj ene poizvedbe lahko poizvedujemo bodisi po več imenovanih ali navadnih grafih, odvisno od tega, ali uporabimo FROM ali FROM NAMED ukaze. GRAPH je zelo močan ukaz, ki omogoča kompleksne poizvedbe, kot je medsebojno povezovanje podatkov iz različnih imenovanih grafov. Znotraj ukaza lahko pišemo poljubne poizvedbe z vsemi gradniki, ki jih SPARQL 1.1 podpira. Če imamo na nekem naslovu shranjenih več grafov, iz katerih želimo poizvedovati, lahko definiramo predpono in jo znotraj poizvedbe v posameznem ukazu GRAPH uporabimo glede na to, iz katerega grafa želimo pridobiti podatke. Zelo pogosto poizvedujemo naenkrat iz dveh ali več grafov, ki hranijo podobne podatke, vendar so zaradi nekih razlogov ločeni. Takrat lahko uporabimo način ukaza GRAPH s spremenljivko, kjer uporabimo spremenljivko, kateri nikjer ne določimo vrednosti, vendar le-ta hrani vrednost

IRI naslova, na katerem se nahaja graf. Če imamo več grafov na isti domeni, lahko namesto spremeljikve uporabimo predpono, ki jo definiramo z ukazom PREFIX, ter nato dostopamo do poljubnega grafa [29].

Sintaksa z uporabo spremenljivke:

```
SELECT ...
FROM NAMED <...>
FROM NAMED <...>
WHERE {
   GRAPH ?spremenljivka { vzorec }
}
```

Spremenljivka znotraj ukaza GRAPH omogoča dostop do vseh grafov v množici, kateri so definirani z ukazom FROM NAMED. Glede na to, v katerem grafu procesor trenutno poizveduje, spremenljivka hrani vrednost IRI naslova le tega, kar pride prav v primerih, kadar želimo poleg želenega podatka vedeti tudi naslov grafa, iz katerega smo dobili določen podatek. Druga pomembna stvar, ki nam jo omogoča spremenljivka v ukazu GRAPH, je, da lahko uporabimo referenco, tako da iz enega grafa naredimo povezavo na drugi graf. To lahko storimo bodisi tako, da imamo v podatkih shranjeno na mestu predmeta ali objekta IRI naslov ali ime grafa, v poizvedbi pa na to mesto postavimo spremenljivko, ki deluje kot referenca, in jo uporabimo v ukazu GRAPH [28].

Sintaksa uporabe reference na nek drug graf:

```
SELECT *
WHERE {
   ...
   GRAPH ?ref { ... }
}
```

V primerih, kjer predmet ali objekt v trojici hrani referenco na naslov grafa, dostop do le tega ne bi uspel, če spremenljivka na tem mestu ne bi imela vrednosti. Da se temu izognemo, je smiselno ukaz GRAPH uporabiti v povezavi z ukazom OPTIONAL.

Zaključimo lahko, da je ukaz GRAPH ter samo poimenovanje grafov v povezavi z ukazoma FROM ter FROM NAMED zelo močna funkcionalnost v SPARQL-u, ki omogoča tako oddaljen dostop kot lokalni dostop do grafov.

4.10 Dodatni gradniki za poizvedovanje

Do sedaj smo se osredotočili na ukaze, ki se uporabljajo znotraj ukaza SELECT, ki je v semantičnem svetu poizvedovanja najbolj pogosta oblika poizvedb, tako kot pri SQL-u. Vendar pa nam SPARQL ponuja mnogo več možnosti za obvladovanje podatkov, ki jih dobimo pri poizvedovanju. V tem poglavju se bomo osredotočili na ključni besedi ASK ter CONSTRUCT.

4.10.1 ASK

ASK je ukaz, ki se uporablja za povpraševanje po podatkih, kjer želimo preveriti, če nek vzorec, ki ga napišemo znotraj poizvedbe, obstaja ali ne. Kot rezultat nam procesor ne vrne podrobnosti o podatkih, temveč samo boolean vrednost (true - resnična ali false - resenična). Ukaz se uporablja predvsem v primerih, kjer preverjamo neka pravila v podatkih, na primer če želimo, da je začetno stanje števila prodaje pri vseh izdelkih postavljeno na 0 ali pa da je objekt tipa niz ipd. Takšne omejitve definiramo znotraj ukaza FILTER v vzorcu bloka WHERE [11].

Sintaksa

ASK {vzorec}
ali
ASK WHERE {vzorec}

Ukaz se lahko uporavlja v dveh načinih. Prvi način je ASK {}, kjer znotraj zavitih oklepajev pišemo preproste vzorce, brez naprednih gradnikov. Drugi način pa je z uporabo WHERE bloka, kjer lahko pišemo kompleksne poizvedbe, ki vsebujejo bodisi matematične izraze, ukaze kot so FILTER, OPTIONAL ter ostali nabor naprednih gradnikov poizvedovalnega jezika SPARQL.

Primer 1:

```
ASK {
   ?x dbpedia-owl:manufacturer <http://dbpedia.org/resource/Audi>
   FILTER(regex(?x, "A7"))
}
```

V primeru 1 najprej najdemo vsa vozila, ki jih je izdelalo podjetje Audi, v ukazu FILTER pa določimo, da želimo samo tista vozila, ki vsebujejo niz "A7" znotraj predmeta (spremenljivka ?x). V primeru, da procesor najde podatke, nam kot rezultat poizvedbe vrne resnično vrednost (true), v nasprotnem pa neresnično (false).

Primer 2:

```
ASK
WHERE {
   ?x a ?type .
   FILTER (?x = <http://dbpedia.org/resource/Slovenia>)
   ?x dbpprop:populationEstimate ?population .
   FILTER (?population > 2000000 && isUri(?x))
}
```

V primeru 2 pošljemo vprašanje na oddaljeno točko, ali obstajajo podatki za državo Slovenijo, kjer mora le-ta imeti zapis o populaciji s predikatom

dbpprop:populationEstimate, pri čemer mora biti ta vrednost večja kot 2 milijona, ter vrednost spremenljivke ?x mora biti URI naslov. Če procesor najde podatke s temi pogoji, vrne za rezultat resnično vrednost (true), v nasprotnem primeru pa neresničeno (false).

4.10.2 CONSTRUCT

CONSTRUCT je ukaz, ki nam omogoča, da lahko RDF podatke spreminjamo, kopiramo in ustvarjamo. Kot rezultat poizvedbe dobimo RDF graf, vendar se moramo zavedati, da z ukazom ne spremenjamo dejanske strukture grafa, tako kot z ukazi SPARQL Update, ki jih bomo spoznali v nadaljevanju. Kot rezultat poizvedbe dobimo RDF graf, ki je takšne oblike, kot ga definiramo znotraj predloge CONSTRUCT ukaza. Znotraj WHERE dela pa definiramo pogoje in vzorce glede na to, kakšne rezultate želimo dobiti iz vhodnega grafa in potem podatke bodisi spreminjamo, kopiramo ali ustvarjamo znotraj predloge. Ukaz je zelo uporaben v primerih, kjer želimo neko množico podatkov iz vhodnega grafa uporabiti tako, da ustvarimo neke spremenjene vrednosti za nove namene, ali če želimo nekim podatkom dodati nove vrednosti ali pa za kopiranje podatkov, ki imajo neko povezavo med seboj in jih bomo kasneje uporabljali za neko drugo množico oz. domeno podatkov. Poizvedba sestavljena iz CONSTRUCT ukaza ne vpliva na originalno vhodno datoteko ali graf, vendar pa lahko rezultat shranimo v datoteko in uporabljamo za kasnejše delo. Dobljeni rezultati poizvedb so v obliki trojic, format, v katerem jih dobimo, pa je odvisen od SPARQL procesorja, katerega uporabljamo [11].

Sintaksa:

CONSTRUCT { predloga }
WHERE { vzorec }

Tako kot ostali ukazi, ki jih podpira SPARQL 1.1, tudi CONSTRUCT ukaz omogoča, da pišemo napredne poizvedbe, kot so poizvedovanje po dato-

tekah, grafih znotraj množice podatkov, dostop do oddaljenih točk, uporabo praznih vozlišč, delo s funkcijam, nizi itd. V primerih, kjer sta vzorec znotraj WHERE bloka ter predloga znotraj CONSTRUCT bloka enaka ter imamo osnovni vzorec, ki ne vsebuje FILTER ukazov in podobnih kompleksnejših vzorcev, lahko uporabimo krajšo verzijo ukaza brez predloge [30].

Sintaksa krajše verzije:

CONSTRUCT WHERE { vzorec }

Spreminjanje podatkov

Z uporabo ukaza CONSTRUCT lahko kopiramo, dodajamo ter spreminjamo podatke, vendar uporaba tega ukaza ne vpliva na spremembo podatkov, shranjenih v grafu, po katerem poizvedujemo. Uporaba ukaza za spreminjanje podatkov je primerna, če uporabljamo dve ali več podobnih množic podatkov, ki pa sicer uporabljata različne predpone in to želimo spremeniti tako, da bodo vse množice podatkov uporabljale iste predikate. Nato je potebno rezultat poizvedbe shraniti še v lokalno RDF datoteko ali na neko oddaljeno točko, da bomo po teh podatkih kasneje lahko poizvedovali. Ukaz pa lahko uporabimo tudi v primerih, kadar želimo spremeniti samo določen predmet, predikat ali trojico, pri čemer je sprememba odvisna od pogoja, ki ga definiramo znotraj WHERE bloka. Znotraj bloka WHERE lahko uporabljamo tudi porazdeljene poizvedbe za oddaljen dostop, ostale napredne ukaze, razne funkcije, ipd. Poizvedba je sestavljena tako, da znotraj predloge ukaza CONSTRUCT definiramo trojice, katerim bomo spremenili poljubne vrednosti. Znotraj bloka WHERE pa definiramo, kateri podatki se bodo spremenili, katere nove vrednosti bodo imeli ipd [11].

Primer:

```
PREFIX bp: <http://test.local/bandprefixes#>
CONSTRUCT {
   ?band bp:clani ?members ;
         bp:zalozba ?labels ;
         bp:wikipedija ?wiki ;
         bp:zvrst ?genre
}
WHERE {
   ?band foaf:name "Bon Jovi"@en ;
         dbpprop:currentMembers ?members ;
         dbpedia-owl:recordLabel ?labels ;
         dbpprop:genre ?genre
   OPTIONAL { ?band foaf:primaryTopic ?wiki }
}
```

Kopiranje podatkov

Kopiranje podatkov v smislu besede CONSTRUCT pomeni, da ukaz uporabimo tako, da glede na vhodne podatke dobimo na izhodu kopijo grafa, ki ustreza vzorcu znotraj bloka WHERE. Ukaz je podoben ukazu SELECT, vendar pri SELECT ukazu dobimo rezultat v obliki trojic. Pri CONSTRUCT ukazu pa dobimo rezultat v obliki trojic (grafa), kjer je oblika odvisna od procesorja [11].

Primer:

```
CONSTRUCT
{ ?country ?p ?o }
WHERE
{
   ?country foaf:name "Slovenia"@en ;
```

```
        ?p ?o
}
```

Ustvarjanje novih podatkov

Ukaz CONSTRUCT lahko uporabimo tudi za ustvarjanje novih trojic. Lahko ustvarimo nove predmete, predikate, objekte ali celotne trojice. Če želimo samo dodati nove trojice, brez obstoječih trojic iz nekega grafa, potem pustimo WHERE blok prazen. V primeru, da želimo neki množici podatkov iz grafa, ki ustreza določenem pogoju definiranem znotraj bloka WHERE, dodati še poljubne vrednosti, pa lahko to znotraj bloka WHERE tudi definiramo. Če želimo uporabiti neke nove predpone, ki trenutno še niso definirane v grafu, po katerem poizvedujemo, je le-te potrebno definirati [11].

Primer 1:

```
PREFIX news: <http://opendata.lavbic.net/news/>
PREFIX dct: <http://purl.org/dc/terms/>
CONSTRUCT{
  <http://www.test.local/novica/novica1> sioc:Post
      "novica1-293a932i" .
  <http://www.test.local/novica/novica1> news:Location "Ljubljana" .
  <http://www.test.local/novica/novica1> dct:title "Poizvedovalni
      jezik SPARQL" .
}
WHERE{}
```

Primer 2:

```
PREFIX ad: <http://www.test.si/additiondata#>
CONSTRUCT
{
    ?city ad:prefixTownLabel ?prefixLabel .
    ?city ad:newPopulation2 ?newPopoulation
}
WHERE
{
   ?city foaf:name "Ljubljana"@en .
   ?city dbpedia-owl:country ?country .
   ?city rdfs:label ?label .
   ?city dbpedia2:populationTotal ?pop .
   ?city dbpprop:leaderName ?leaderName ;
         ?p ?o .
   FILTER (str(?label) = "Ljubljana")
   BIND(SUBSTR("Ljubljana", 1,2) AS ?prefixLabel)
   BIND((?pop*2) AS ?newPopoulation)
}
```

4.11 SPARQL/Update

SPARUL oz. SPARQL/Update je dodatek k osnovnemu SPARQL poizvedovalnem jeziku in se je pojavil šele v verziji SPARQL 1.1. Ukazi in možnosti, ki smo jih spoznali do sedaj, nam ponujajo široko paleto, kaj vse lahko delamo s SPARQL poizvedovalnim jezikom. Spoznali smo tudi ukaz CONSTRUCT, ki se uporablja za dodajanje, spreminjanje ter kopiranje podatkov, vendar ta ne vpliva ne strukturo RDF grafa. Če pa želimo, da lahko podatke shranjene v RDF obliki dejansko dodajamo, urejamo in brišemo, potrebujemo SPARQL 1.1 Update. S to možnostjo, ki v verziji 1.0 še ni bila omogočena, postane SPARQL mnogo več kot le poizvedovalni jezik [11].

4.11.1 Vstavljanje podatkov

Specifikacija za izvajanje SPARQL Update poizvedb predlaga da datoteke, ki izvajajo Update zahtevke, vsebujejo končnico ".ru" [11].

Sintaksa:

INSERT {} ali
INSERT INTO <uri> WHERE ali
INSERT {} WHERE {} ali
INSERT DATA {} ali
INSERT DATA INTO <uri> {}

Ukaz INSERT lahko primerjamo z ukazom CONSTRUCT. Vendar glavna razlika med njima je ta, da ukaz INSERT, tako kot vsi ostali SPARQL/Update ukazi, lahko spremeni podatke iz originalne datoteke ali grafa podatkov, nad katerim izvajamo poizvedbe, CONSTRUCT pa ne more. Pri večji količini spreminjanja podatkov je bolje uporabiti INSERT DATA kot INSERT, ker je ta način hitrejši. INSERT WHERE pa nam omogoča, da vstavimo podatke, ki so odvisni od vzorca znotraj WHERE bloka. V sintaksi imamo še ostale možnosti za ukaze, ki vsebujejo ukaz INTO, ki se uporablja, da določimo IRI naslov za imenovani graf, ki ga želimo posodobiti.

Primer:

```
INSERT {
   <http://www.testnovice.local/novice/novica321>
              news:nComments "0"^^xsd:integer ;
              news:nRatings "0"^^xsd:integer .
};
INSERT {
   <http://www.testnovice.local/novice/novica322>
              news:nComments "0"^^xsd:integer ;
              news:nRatings "0"^^xsd:integer .
```

```
}
```

V zgornjem primeru smo uporabil dva INSERT ukaza, ki v graf dodata dvakrat po dve trojici. S tem dodamo novicama 321 ter 322 podatek o številu komentarjev ter številu glasov. Obe vrednosti postavimo na nič ter ju definiramo kot celi števili (xsd:integer).

Primerjava s SQL-om:

Povpraševalni jezik SQL tudi ponuja ukaz INSERT za dodajanje vsebine v bazo podatkov, ki omogoča dva načina dodajanja podatkov. Prvi, kjer podamo vse vrednosti za vse stolpce v tabeli, ter drugi, ki omogoča, da lahko napolnimo poljubne stolpce v tabeli, ostali stolpci pa dobijo vrednost NULL oz. tisto, ki je privzeto nastavljena [14]. Prednost SPARQL-a je v tem, da omogoča dodajanje podatkov v nek oddaljen graf. Prav tako kot SPARQL nam tudi SQL ponuja, da znotraj INSERT ukaza pišemo SELECT ukaze, ki pridobijo podatke iz že obstoječih tabel, ter tako nastavimo vrednosti določenim stolpcem glede na že obstoječe podatke.

SQL Sintaksa:

INSERT INTO tabela VALUES (vred1, vred2, ...)
INSERT INTO tabela (stolpec1, stolpec2, ...) VALUES (vred1, vred2, ...)

Primer:

```
INSERT INTO dbo.Novica (ID, Vsebina, Avtor, Povzetek,
            DatumNastanka, SteviloKomentarjev, SteviloGlasov)
VALUES (6, 'Vsebina 6', (SELECT ID FROM dbo.Avtor
            WHERE dbo.avtor.Ime LIKE 'Janez'), 'Povzetek 6 ...',
            '2014-01-12', 0, 0)
```

4.11.2 Brisanje podatkov

Drugi pomemben način poizvedb SPARQL/Update je brisanje podatkov. Z ukazoma DELETE ter DELETE DATA lahko izbrišemo poljubne vzorce podatkov glede na vzorec znotraj WHERE bloka ali tako, da sami napišemo trojice, ki jih želimo izbrisati [11].

Sintaksa:

DELETE {} ali
DELETE FROM <uri> {} WHERE {} ali
DELETE {} WHERE {} ali
DELETE DATA {} ali
DELETE DATA FROM <uri>

Sintaksa ukazov za brisanje je zelo podobna sintaksi za vstavljanje podatkov, vendar pri brisanju podatkov ne moremo uporabljati praznih vozlišč ter spremenljivk. Ukaz DELETE DATA tako kot INSERT DATA ne dovoli, da ta vsebuje WHERE blok, ter je pri večjem številu brisanja podatkov hitrejši kot sam DELETE ukaz. Uporaba tega ukaza je primerna, ko točno vemo, katere trojice želimo izbrisati. Drugi ukaz, ki nam omogoča isto možnost za brisanje, je ukaz DELETE, vendar ta omogoča več fleksibilnosti, če ga uporabimo skupaj z ukazom WHERE, kjer določimo pogoje oz. omejitve nad podatki, ki jih želimo izbrisati.

Tako kot pri dodajanju, imamo pri brisanju možnost določiti, iz katerega grafa želimo brisati podatke. V sintaksi imamo možnosti za ukaze DELETE, ki vsebujejo še ukaz FROM, s katerim določimo, iz katerega naslova oz. grafa bomo brisali podatke [11].

Primer 1:

```
DELETE {
   <http://www.testNovice.local/novice/novica41>
         sioc:UserName "Janez Novak"
}
WHERE { }
```

Zgornji primer bi iz množice podatkov odstranil trojico, kjer je uporabniško ime osebe, ki je ustvarila novico 41, enako Janez Novak, če le ta obstaja.

Primer 2:

```
DELETE {
   ?novica ?pred ?obj
}
WHERE {
   ?novica news:location ?location .
   FILTER (str(?location) = "Ljubljana")
   ?novica ?pred ?obj
  }
```

V primeru 2 bi iz množice podatkov odstranili vse tiste trojice, pri katerih nek predmet vsebuje predikat news:location ter ima vrednost Ljubljana.

Primerjava s SQL-om:

SQL za brisanje vsebine iz baze podatkov ponuja ukaz DELETE. Po navadi uporabimo za brisanje podatkov primarni ključ, s katerim v WHERE pogoju določimo, katero vrstico v bazi bomo zbrisali. Lahko pa tudi uporabimo kateri drugi stolpec, po katerem bomo brisali, ali celo več stolpcev [14]. Tako kot pri INSERT ukazu je prednost SPARQL-ovega ukaza DELETE v tem, da omogoča brisanje podatkov iz nekega oddaljenega grafa. SPARQL omogoča

tako brisanje vseh podatkov, katere predstavlja nek predmet, kot tudi brisanje samo poljubnih trojic, ki so del nekega predmeta. To storimo tako, da znotraj DELETE bloka napišemo, katere trojice želimo izbrisati (glej primer 1), če pa želimo izbrisati vse trojice, ki predstavljajo nek predmet, pa uporabimo znotraj DELETE in WHERE bloka enako trojico treh spremenljivk, ki izbriše vse podatke glede na določen pogoj (glej primer 2).

SQL Sintaksa:

DELETE FROM tabela
WHERE stolpec=vrednost;

Sintaksa za izbris vseh podatkov iz tabele:

DELETE FROM tabela; ali
DELETE * FROM tabela;

Primer:

```
DELETE FROM dbo.Novica
WHERE ID = 2
```

V zgornjem primeru smo iz tabele Novica izbrisali novico, ki je imela ID enak dva. Znotraj WHERE ukaza lahko definiramo še več pogojev. DELETE ukaz pa lahko uporabimo tudi brez WHERE ukaza. Uporabna sta tudi ukaza ORDER BY ali LIMIT, s katerima določimo, koliko podatkov iz baze želimo izbrisati.

4.11.3 Spreminjanje podatkov

Do sedaj smo spoznali dva načina, kako dodati nove podatke ter kako izbrisati neke podatke. V tem odstavku pa se bomo lotili načina za spreminjanje podatkov, ki nam pogosto pride prav, če želimo nad podatki narediti neke male ali večje spremembe.

SPARQL ponuja zaporedje ukazov DELETE / INSERT / WHERE, kjer SPARQL procesor pridobi rezultat glede na vzorec znotraj WHERE bloka, nato izvede vse, kar je znotraj DELETE bloka, ter na koncu še vse, kar je znotraj INSERT bloka, pri tem pa WHERE del v poizvedbi ni obvezen. Če ne uporabimo WHERE bloka, se bodo spremenili vsi podatki oz. trojice, ki ustrezajo predlogama znotraj ukaza DELETE in INSERT. Tako lahko podatke uspešno posodobimo, bodisi da gre za predmet, predikat ali pa objekt, odvisno, kako zastavimo poizvedbo. Poleg preprostih poizvedb za spreminjanje podatkov lahko pišemo kompleksne poizvedbe, pri čemer lahko ustvarimo nove vrednosti na podlagi prejšnjih, spremenimo predikate in podobno [11].

Sintaksa:

DELETE {predloga}
INSERT {predloga}
[WHERE {vzorec}]

Primer:

```
DELETE {
   ?post sioc:UserName ?name
}
INSERT {
   ?post foaf:name ?name ;
         news:staticstics/news:avgRating ?avgRatNew
}
WHERE {
   ?post sioc:UserName ?name ;
          news:staticstics/news:avgRating ?avgRat
   FILTER(?avgRat = 0)
   BIND((?avgRat + 1) AS ?avgRatNew)
}
```

V zgornjem primeru posodobimo podatke za tiste novice, ki imajo predikat sioc:UserName (spremenimo v predikat foaf:name) ter imajo določeno povprečje glasov enako nič. Tem trojicam s predikatom news:avgRating povečamo število za 1.

Primerjava s SQL-om:

SQL ponuja ukaz UPDATE za spreminjanje vsebine podatkov v relacijskih podatkovnih bazah. Sama sintaksa je malenkost bolj preprosta kot pri poizvedovalnem jeziku SPARQL, deluje pa tako, da posodobi vsebino zapisov oz. vrstic v tabeli, ki ustrezajo določenemu pogoju znotraj WHERE bloka. Znotraj SET bloka nastavimo nove vrednosti zapisov kot to storimo pri SPARQL-u znotraj INSERT bloka. Oba poizvedovalna jezika omogočata, da pišemo kompleksne poizvedbe za spreminjanje podatkov, tako da pri SQL-u uporabimo dodatne SELECT ali JOIN ukaze, ki pridobijo podatke iz že obstoječih tabel, pri SPARQL-u pa to preprosto storimo z uporabo trojic znotraj WHERE bloka. Uporaba WHERE bloka je pri obeh poizvedovalnih jezikih neobvezna, kar pomeni, da lahko posodobimo vse zapise v določeni tabeli.

SQL Sintaksa:

UPDATE ime_tabele
SET stolpec1 = vrednost1, stolpec2 = vrednost2, ...
[WHERE stolpecX = vrednostX;]

Primer:

```
UPDATE dbo.Novica
SET SteviloKomentarjev = (SteviloKomentarjev + 2),
    Povzetek = 'Nova vsebina za povzetek ...'
WHERE ID = 9
```

4.11.4 Ostale možnosti

SPARQL/Update nam ponuja mnogo več kot le zgornje tri možnosti za delo s podatki. Ukazi, ki so nam na voljo, so še: ukaz DROP za brisanje grafov iz podatkovnih množic, ukaz CLEAR, ki se uporablja za brisanje trojic iz grafov, LOAD ukaz, ki je primeren za kopiranje trojic iz oddaljenega grafa v poljubni graf, MODIFY ukaz, ki je primeren za spreminjanje podatkov, kot smo ga spoznali višje, vendar tukaj še določimo, kateri graf želimo spremeniti, ukaz CREATE, ki ustvari nov imenovani graf z določenim IRI naslovom, ukaz SILENT, ki ignorira napako, če se le-ta pojavi, in razne kombinacije vseh teh ukazov [33].

4.12 Tabelarična primerjava med SPARQL-om in SQL-om

Zaključimo lahko, da SPARQL v primerjavi z SQL-om ponuja kar nekaj več funkcionalnosti. Predvsem se to pozna pri naprednih ukazih, kot so CONSTRUCT, GRAPH, SERVICE, razne funkcije ipd., ki ponujajo programerju zelo veliko možnosti za delo s podatki. Predvsem pa je SPARQL poizvedovalni jezik za sorazmerno mlado vejo podatkov – semantični splet.

Ukaz	SPARQL	SQL
OPTIONAL	Omogoča, da dobimo rezultat tudi v primeru, da spremenljivka znotraj OPTIONAL bloka nima vrednosti.	Podoben ukaz je zunanji stik, ki lahko med seboj poveže več tabel in omogoča, da dobimo rezultat, tudi če nekatere vrednosti v poljubnem stolpcu nimajo vrednosti.
FILTER	Omogoča pisanje pogojev znotraj poizvedbe. Deluje nad celotno skupino, v kateri se pojavi. Ukaz se vedno izvede na koncu poizvedbe.	Pozna ukaz WHERE, ki ima isto funkcijo kot ukaz FILTER.
MINUS	Odstrani vrednosti z leve strani ukaza, ki imajo isto vrednost kot tiste, ki so veljavne na desni strani ukaza.	Ne vrača podvojenih vrednosti. Število stolpcev ter tipi le teh morajo biti enaki na obeh straneh ukaza MINUS.
Regex	Omogoča napredno iskanje po nizih z regularnimi izrazi.	Uporablja se znotraj LIKE ukaza ter prav tako omogoča napredno iskanje po nizih.
NOT IN / IN	Uporablja se za ujemanje poljubnih vrednosti znotraj polja, znotraj katerega definiramo poljubne vrednosti.	Omogoča isto funkcijo za možnost ujemanja podatkov.
UNION	Omogoča združevanje več vzorcev. Pogoj na eni strani ukaza se nanaša na celotno skupino v kateri se nahaja.	Ne vrača podvojenih vrednosti, vendar pozna tudi ukaz UNION ALL, ki jih vrača. Imena stolpcev ter tipi stolpcev se morajo ujemati.

Ukaz	SPARQL	SQL
BIND	Uporablja se za dodeljevanje vrednosti poljubni spremenljivki. Lahko ga uporabimo znotraj SELECT stavka, WHERE bloka ali v povezavi z agregarnimi funkcijami.	Uporablja se za dodeljevanje začasnih poljubnih imen stolpcev in tabel v v času poizvedbe.
GRAPH	Ukaz za poizvedovanje po oddaljenih ali lokalnih RDF grafih, katere lociramo z IRI naslovom.	Ne omogoča oddaljenega poizvedovanja, vendar nekateri ponudniki baz omogočajo poizvedovanje iz več loklanih podatkovnih baz hkrati.
SERVICE	Omogoča poizvedovanje po več kot eni oddaljeni množici podatkov, ki se nahaja kjerkoli na spletu.	Ne omogoča oddaljenega poizvedovanja.
INSERT	Omogoča dodajanje podatkov tako na oddaljenih kot lokalnih množicah podatkov.	Omogoča dodajanje podatkov samo v poljubno tabelo, ki se nahaja znotraj podatkovne baze.
UPDATE	Omogoča spreminjanje podatkov tako na oddaljenih kot lokalnih množicah podatkov.	Omogoča spreminjanje podatkov samo v poljubni tabeli, ki se nahaja znotraj podatkovne baze.
DELETE	Omogoča brisanje podatkov tako na oddaljenih kot lokalnih množicah podatkov.	Omogoča brisanje podatkov samo v poljubni tabeli, ki se nahaja znotraj podatkovne baze.

Poglavje 5

Program

5.1 O programu

V sklopu diplomske naloge smo razvili program, ki omogoča poizvedovanje po podatkih na oddaljeni točki [1], ki zajema podatke o novicah s spletnega portala 24ur [5], ter Dbpedia, ki zajema podatke o virih iz Wikipedije [6]. Omogočili smo tudi ustvarjanje novih novic, ki se shranijo v lokalno datoteko, ter poizvedovanje po le teh. Naredili smo tudi prikaz uporabe porazdeljenih poizvedb na oddaljeno točko DBpedia [3]. S tem smo pokazali, kako se uporabi ukaz SERVICE, ki v semantičnem spletu pride še kako prav zaradi hkratnega poizvedovanja na več oddaljenih točkah. Prav tako kot pri lokalnem in oddaljenem poizvedovanju smo že prej napisali nekatere poizvedbe, vendar pa je program napisan tako, da omogoča uporabniku, da tudi sam napiše poizvedbe in pošilja zahtevke, bodisi na oddaljeno SPARQL točko ali lokalno poizveduje po podatkih, shranjenih v RDF datoteki.

Program je bil napisan v programskem jeziku Java ter v razvojnem okolju Eclipse. Uporabili smo okolje Swing, ki je Javino primarno grafično okolje, ki vsebuje gradnike in orodja za izgradnjo uporabniškega vmesnika. Odločili smo se, da bo program napisan v Javi, saj je trenutna spletna stran [12], iz katere smo izhajali, napisana v JavaScript-u in omogoča poizvedovanje na oddaljeno točko [1]. Zato je bilo bolj zanimivo in mogoče tudi bolj uporabno

narediti takšno aplikacijo, ki ne ponuja samo poizvedovanja na to točko, temveč smo naredili bolj obširno rešitev, ki ponuja tudi ustvarjanje lastnih novic, poizvedovanje po le-teh ter shranjevanje rezultatov v datoteke tipa JSON, RDF ali HTML.

Za izgradnjo dizajna programa smo uporabili vtičnik Windows Builder, katerega smo v okolje Eclipse najprej namestili, ker v standardni verziji še ni bil nameščen. Vtičnik omogoča programerju, da lahko hitreje in bolj učinkovito sestavi obliko programa ter na njega postavi komponente, kamor koli želi. Deluje po načinu "povleci in pusti", pri čemer se Java koda, za katero bi programer s pisanjem na roke porabil veliko več časa, generira sama.

5.2 Implementacija

Program je sestavljen iz treh ključnih delov, poizvedovanje po podatkih iz oddaljenih SPARQL točk, ustvarjanje novic, ki se shranijo lokalno v datoteko, ter poizvedovanje po le teh. Program je napisan tako, da uporabnika z uporabo opozoril obvešča o njegovih napakah ter korakih, ki jih mora storiti. Opozorila se pojavijo, če pride do napak pri pisanju ali branju datoteke, ali če uporabnik ne vnese vseh potrebnih polj pri ustvarjanju novic ali pri vnosu podatka o imenu datoteke. Za opozorila smo uporabili komponento dialog – JOptionPane, kjer uporabnika obvestimo bodisi o napaki ali o tem, kar mora narediti. V primeru napake pri pisanju poizvedb program vrne napako v samo polje za rezultat poizvedbe. Tako lahko uporabnik pri naslednjem pisanju poizvedbe oz. popravkih vidi, kje se je zgodila napaka in kaj mora spremeniti oz. katero sintakso mora uporabiti, da bo poizvedba delovala. S tem smo se približali zahtevam o prijaznosti uporabniških programov do uporabnika.

V prvem delu programa smo se lotili poizvedovanja po dveh oddaljenih SPARQL točkah [1, 3]. Za ta namen smo napisali funkcijo exRemoteQuery( ). Ker smo v prvem delu programa (oddaljeno poizvedovanje)

omogočili možnost pisanja lastnih poizvedb, bodisi na eno izmed dveh oddaljenih SPARQL točk, funkcija exRemoteQuery() najprej preveri, ali želimo sami napisati funkcijo, ali želimo izvajati že prednapisane poizvedbe. Znoraj projekta smo ustvarili še dva razreda (LavbicEndpoint ter Dbpedia), znotraj katerih smo napisali funkcije, ki jih potebujemo za pridobivanje podatkov. Znotraj razreda Dbpedia smo napisali dve funkciji, ki se uporabljata za poizvedovanje po podatkih iz DBpedije. Prva funkcija getPrefixDbpedia() nam vrne predpone, ki jih potrebujemo za pisanje poizvedb, da ne rabimo pisati dolgih IRI naslovov, ter druga getData(String printType, String poizvedba), ki sprejme dva parametra (tip prikaza rezultata ter samo kodo poizvedb) ter izvede poizvedbo in vrne rezultat glede na tip, ki ga izberemo znotraj programa (JSON, XML ali HTML). Razred LavbicEndpoint pa uporabljamo za poizvedovanje po novicah iz portala 24ur. Znotraj tega razreda smo napisali tri funkcije za poizvedovanje po podatkih iz oddaljene točke [1]. Prva funkcija getPrefixLabvic() se uporablja tako kot pri razredu Dbpedia za predpone, druga funkcija getData(String printType, String poizvedba) sprejme dva parametra (tip prikaza rezultata ter samo kodo poizvedbe) in vrne rezultat poizvedbe glede na način prikaza, ki ga izberemo (JSON, XML ali HTML), tretja funkcija getHTMLresults(String resultsString), ki sprejme kot prvi parameter rezultat poizvedbe v obliki JSON, pa se uporablja da vrne rezultat poizvedbe v HTML obliki.

Funkcija getData - poizvedovanje po novicah iz 24ur

Kot smo višje omenili, funkcija sprejme dva parametra, eden je za način prikaza podatkov, drugi pa sama koda poizvedbe, ki se pošlje na oddaljeno SPARQL točko, ki to poizvedbo izvede in uporabniku vrne rezultate poizvedbe. Pri tej funkciji smo želeli uporabiti knjižnico Jena, ki se uporablja za izvajanje SPARQL poizvedb, vendar smo naleteli na težavo pri dostopu, kajti pri uporabi oddaljene točke je potrebno vnesti uporabniško ime in geslo. Zaradi te težave smo se odločili za drugo knjižnico (Apache Http Client), ki pošlje zahtevek na poljubno spletno stran in vrne odgovor zahtevka v poljubni

obliki. V programu smo omogoči tri načine prikazovanje podatkov (JSON, XML ter HTML), zato je bilo potrebno glede na način prikaza podatkov v glavo zahtevka podatki poseben "header".

Uporaba knjižnice Apache Http Client:

```
HttpClient client = new DefaultHttpClient();
HttpGet request = null;
String resultsString = "";
request = new
    HttpGet("http://opendata.lavbic.net/news/sparql?query=" +
    URLEncoder.encode(prefixString + poizvedba));
String basic_auth = new
    String(Base64.encodeBase64(("uporabnisko_ime" + ":" +
    "geslo").getBytes()));
request.addHeader("Authorization", "Basic " + basic_auth);
```

Zgornji primer Java kode je iz našega programa, kjer se lepo vidi način uporabe knjižnice Apache Http Client. Zahtevek (HttpGet) smo ustvarili tako, da smo podali URL naslov do oddaljene točke, ki vsebuje še parameter query, znotraj katerega smo podali zakodirano poizvedbo s predponami. Le to je bilo potebno zakodirati zaradi posebnih znakov (?, !, ., @, "), da dobimo pravilni URL naslov. V zahtevek smo dali še parameter za avtorizacijo, ki je potrebna za dostop do URL naslova. Parameter sprejme uporabniško ime in geslo, katerega smo na koncu še zakodirali ter dodali v glavo zahtevka.

Ker imamo v programu možnost izpisa rezultata v treh oblikah, ta funkcija vsebuje še tri if pogoje, ki preverijo kateri način prikaza smo izbrali. V primeru da izberemo JSON način prikaza, smo v glavo zahtevka dodali nasleden parameter. Naslednji parameter smo dodali tudi, če želimo prikazat rezultate v HTML obliki, vendar smo tam klicali še dodatno funkcijo getHTMLresults( ), ki smo jo napisali tako, da vrne rezultat v HTML obliki.

Parameter za prikaz rezultata v obliki JSON:

```
request.addHeader("Accept", "application/sparql-results+json");
```

Če želimo rezultat poizvedbe prikazati v XML obliki, smo v glavo zahtevka dodali naslednji parameter.

Parameter za prikaz rezultata v obliki XML:

```
request.addHeader("Accept", "application/sparql-results+xml");
```

Koda, ki pridobi podatke, je dokaj kratka in enostavna. Napisali smo jo tako, da v primeru, da pride do napake pri poizvedovanju, program vrne opozorilo. HttpClient izvede zahtevek na URL naslov, ki smo navedli, rezultat pa shranimo v BufferReader, da se lahko nato z while zanko sprehodimo čez vsako vrstico in shranimo rezultat v spremenljivko resultsString, tako da ločimo vsako vrstico z znakom za novo vrstico zaradi lepšega izpisa rezultata.

Koda, ki pridobi podatke iz oddaljene točke opendata.lavbic.news:

```
HttpResponse response;
try {
     response = client.execute(request);
     BufferedReader rd = new BufferedReader (new
         InputStreamReader(response.getEntity().getContent()));
     String line = "";
     while ((line = rd.readLine()) != null) {
          resultsString += line + "\n";
     }
} catch (IOException e) { }
```

Če za način izpisa izberemo HTML, se koda razlikuje samo v tem, da na koncu kode, ki smo jo zgoraj prikazali, kličemo še funkcijo getHTMLresults(String resultsString), ki sprejme za prvi parmater rezultat v JSON

obliki in nato generira HTML kodo, ki se potem prikaže v programu. Za generiranje HTML dokumenta smo uporabili knjižnico JSON Simple 1.1.1, ki omogoča, da iz JSON oblike pridobimo poljubne spremenljivke, katere nato shranimo v spremenljivko tipa String, samo HTML kodo pa smo generirali ročno.

Koda, ki iz JSON oblike generira HTML dokument:

```
String content = resultsString;
String HTMLresult = "";
try {
     String genreJson = content;
     JSONObject genreJsonObject = (JSONObject)
         JSONValue.parseWithException(genreJson);

     JSONObject heads = (JSONObject) genreJsonObject.get("head");
     String vars = heads.get("vars").toString().replace("[",
         "").replace("]", "").replace("\"", "");
     String [] varArr = vars.split(",");
     try {
          HTMLresult += "<html><head></head><body><table
              cellspacing=\"0\" rowspacing=\"0\"><tr><th
              style='font-size:x-small; text-align:left;
              border-bottom:thin gray solid; " +
              "background-color:#FAFAFA;'>St.</th>";
          for(String a : varArr){
               HTMLresult += "<th style='font-size:x-small;
                   text-align:left; border-bottom:thin gray solid;
                   border-left:thin gray solid; " +
                   "background-color:#FAFAFA;'>" + a + "</th>";
          }
          JSONObject results1 = (JSONObject)
              genreJsonObject.get("results");
```

```
            JSONArray arr1 = (JSONArray) results1.get("bindings");

             for(int i=0; i<arr1.size(); i++){
                 JSONObject resultVal = (JSONObject) arr1.get(i);
                 if(i == arr1.size()-1){
                     HTMLresult += "<tr><td style='text-align:right;
                         color:#000000; font-size:small;'>" + (i+1) +
                         "</td>";
                 }else{
                     HTMLresult += "<tr><td style='text-align:right;
                         color:#000000; font-size:small;
                         border-bottom:thin gray solid;'>" + (i+1) +
                         "</td>";
               }
               for(String a : varArr){
                   JSONObject bindingsArr = (JSONObject)
                       resultVal.get(a);
                   if(i == arr1.size()-1){
                       HTMLresult += "<td style='text-align:left;
                           font-size:small; border-left:thin gray
                           solid;'>" + bindingsArr.get("value") +
                           "</td>";
                   }else{
                       HTMLresult += "<td style='text-align:left;
                           font-size:small; border-bottom:thin gray
                           solid; border-left:thin gray solid;'>" +
                           bindingsArr.get("value") + "</td>";
                   }
                } HTMLresult += "</tr>";
           }
           HTMLresult += "</table></body></html>";
    } catch (Exception e) { }
    return HTMLresult;
```

Pri poizvedovanju na oddaljeno točko DBpedia pa ni bilo potrebno uporabiti avtentikacije pri dostopu, zato smo lahko uporabili knjižnico Jena, ki omogoča, da izvajamo poizvedbe tako nad lokalnimi podatki kot nad oddaljenimi storitvami. V primeru poizvedovanja na oddaljeno točko DBpedia smo napisali funkcijo getData(String printType, String poizvedba), ki prejme kot prvi parameter način izpisa, kot drugi pa celotno poizvedbo.

Koda, ki pridobi podatke iz oddaljene točke DBpedia:

```
String printResults = "";
String service = "http://dbpedia.org/sparql/";
String prefixString = getPrefixDbpedia();

QueryExecution qe = QueryExecutionFactory.sparqlService(service,
    prefixString + poizvedba);
ResultSet rs = qe.execSelect();
if (rs.hasNext()) {
     if(printType == "HTML"){
          printResults = ResultSetFormatter.asText(rs);
     }else if(printType == "XML"){
          printResults = ResultSetFormatter.asXMLString(rs);
     }else if(printType == "JSON"){
          ByteArrayOutputStream b = new ByteArrayOutputStream();
          ResultSetFormatter.outputAsJSON(b, rs);
          printResults = b.toString( );
     }
}
qe.close();
return printResults;
```

Znotraj kode (3 vrstica) najprej kličemo funkcijo getPrefixDbpedia(), ki smo jo napisali zato, da nam ni potrebno pisati dolgih URI naslovov znotraj poizvedbe. Nato pa z uporabo knjižnice Jena izvedemo poizvedbo na

oddaljeno točko Dbpedia in podatke shranimo v spremenljivko rs, tipa ResultSet. Nato pa z ResultSetFormatter-jem, ki ga ponuja knjižnica Jena, vrnemo podatke glede na željeno obliko (JSON, XML ali HTML). Glede na to, v kateri obliki želimo rezultat, glede na to izvedemo določeno operacijo. HTML oblika - asText(), XML - asXMLString, kjer obe funkciji prejmeta parameter tipa ResultSet. Če pa želimo rezultat v obliki JSON, pa kličemo funkcijo outputAsJSON, vendar tukaj moramo podati dva parametra, prvi je tipa ResultSet, drugi pa ByteArrayOutputStream, ki ga nato pretvorimo v niz tipa String, da dobimo podatke v primeri obliki. Na koncu povezavo še zapremo in tako je izvedba poizvedbe na oddaljeno točko v Javi zaključena.

Znotraj prvega dela programa smo omogočili tudi pisanje lastnih poizvedb. Če izberemo opcijo, da želimo sami napisati poizvedbo, je potebno izbrati eno izmed oddaljenih točk, ki glede na to, katero točko smo izbrali, kličejo eno izmed prej omenjenih funkcij. V tem delu pisanja poizvedb smo omogočili še izvoz rezultata poizvedbe v datoteko. Za to funkcionalnost smo se odločili, ker pogosto želimo delati z neko množico podatkov, katero smo že v preteklosti pridobili z izvajanjem poizvedb, ter da nam ni potrebno pisati istih poizvedb, kajti le-te bomo sedaj imeli shranjene bodisi v HTML, XML ali JSON obliki. Če želimo podatke izvoziti v datoteko, program najprej preveri, ali imamo v polju za rezultat poizvedbe kakšne podatke. V primeru, da je polje prazno, program uporabnika opozori, v nasprotnem primeru pa ga vpraša po imenu datoteke, v katero želi shraniti. Če datoteka že obstaja, program uporabnika o tem tudi obvesti. Ko uporabnik uspešno vnese ime datoteke, se rezultati shranijo bodisi v HTML, XML ali JSON datoteko. Novo nastala datoteka se nahaja v mapi Rezultati poizvedb iz oddaljenega endpointa.

Naslednja funkcionalnost, ki smo jo v programu omogočili, je ustvarjanje novic, ki se hranijo lokalno v rdf datoteki. V programu imamo obrazec, kamor vnesemo obvezna polja (naslov novice, avtor novice, povzetek ter vsebina).

ID novice ter URL novice se generirata sama, pri čemer se ID novice hrani v datoteki novicaID.txt, iz katere program prebere trenutno vrednost. Dokler vse obvezna polja ne izpolnimo, nas program opozarja, katera vnosna polja morajo biti še izpolnjena. Ko so vsa obvezna polja izpolnjena, pa program s pomočjo Jena knjižnice shrani podatke v lokalno rdf datoteko. Jena pozna tako imenovane modele, v katerih lahko začasno hranimo trojice, zato smo s pomočjo ModelFactory ter funkcije createDefaultModel ustvarili spremenljivko "model", ki je tipa Model, in vanj shranili tako predpone kot trojice. Trojice in predpone smo generirali na podlagi vnešenih podatkov znotraj programa. Najprej smo ustvarili vir (Resource), ki je nova novica, v rdf datoteki predstavljena kot rdf:Description element, ki ima še parameter rdf:about, ki unikatno predstavlja posamezno novico. V primeru, da rdf:Description element ne vsebuje parametra rdf:about, gre za prazno vozlišče. Nato smo našemu modelu dodali še vrednosti za Property, ki so v trojicah predstavljeni kot predikati, na koncu pa smo definirali še celotne trojice z ukazoma add ter addLiteral. Ukaz addLiteral smo uporabili zato, da smo določeni vrednosti podali še tip (v našem primeru je to xsd:string), z ukazom add pa smo ustvarili trojico, ki ima za objekt vrednost tipa xsd:date in predstavlja datum nastanka novice. Program je napisan tako, da v primeru, da datoteka "novice.rdf" še ne obstaja, jo ustvari. V nasprotnem primeru pa preveri, ali je ta prazna. V primeru da ni prazna pa v model prebere podatke, po katerih lahko potem poizvedujemo, dodajamo nove trojice in podobno. Na koncu, ko ustvarimo in zgeneriramo novo novico, podatke iz posodobljenega modela shranimo nazaj v datoteko.

Koda, ki generira rdf datoteko trojic:

```
Model model;
File fileNovice = new File("novice.rdf");
if(fileNovice.length() == 0){
    model = ModelFactory.createDefaultModel();
}else{
```

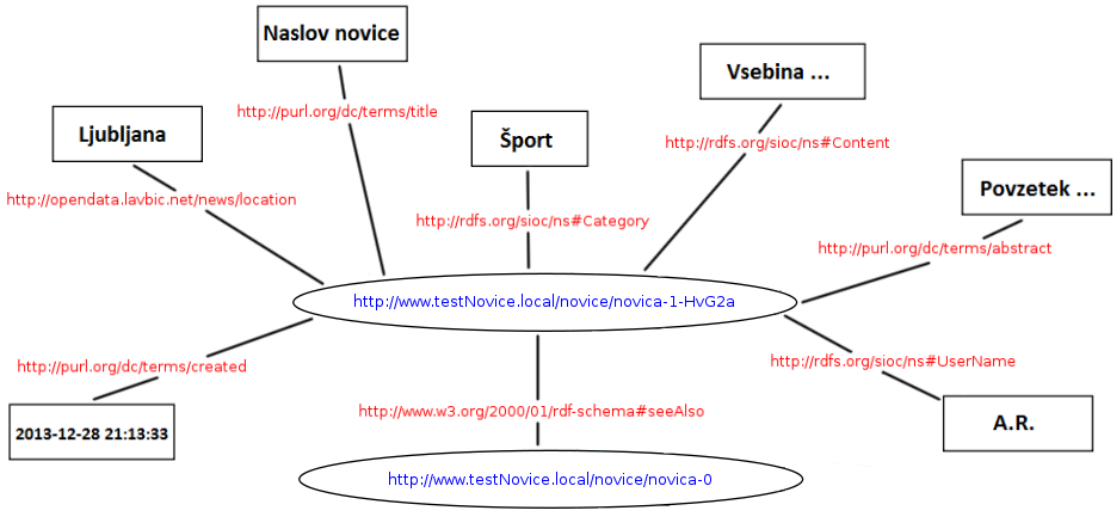
```
    InputStream in = FileManager.get().open("novice.rdf");
    model = ModelFactory.createDefaultModel();
    model.read(in, null);
}

String dcterms = "http://purl.org/dc/terms/";
String sioc = "http://rdfs.org/sioc/ns#";
String news = "http://opendata.lavbic.net/news/";
String rdf = "http://www.w3.org/1999/02/22-rdf-syntax-ns#";
String rdfs = "http://www.w3.org/2000/01/rdf-schema#";
String xsd = "http://www.w3.org/2001/XMLSchema#";

model.setNsPrefix("dct", dcterms);
model.setNsPrefix("sioc", sioc);
model.setNsPrefix("news", news);
model.setNsPrefix("rdf", rdf);
model.setNsPrefix("rdfs", rdfs);
model.setNsPrefix("xsd", xsd);

Resource res = model.createResource(novicaUniqueID);

Property titleProp1 = model.createProperty(dcterms + "title");
Property categoryProp2 = model.createProperty(sioc + "Category");
Property locationProp3 = model.createProperty(news + "location");
Property userNameProp4 = model.createProperty(sioc + "UserName");
Property abstractProp5 = model.createProperty(dcterms +
   "abstract");
Property contentProp6 = model.createProperty(sioc + "Content");
Property seeAlsoProp7 = model.createProperty(rdfs + "seeAlso");
Property createdProp8 = model.createProperty(dcterms + "created");

String timeStampNow = new SimpleDateFormat("yyyy-MM-dd
   HH:mm:ss").format(Calendar.getInstance().getTime());
```

```
model.addLiteral(res, titleProp1, naslovNovice).addLiteral(res,
    userNameProp4, avtorNovice).addLiteral(res, abstractProp5,
    povzetekNovice).addLiteral(res, contentProp6,
    vsebinaNovice).add(res, createdProp8,
    ResourceFactory.createTypedLiteral(timeStampNow,
    XSDDatatype.XSDdate)).addLiteral(res, seeAlsoProp7,
    "http://www.testNovice.local/novice/novica-" +
    textFieldIDNovice.getText());
if(!temaNovice.equals("")){
     model.addLiteral(res, categoryProp2, temaNovice);
}
if(!lokacijaNovice.equals("")){
     model.addLiteral(res, locationProp3, lokacijaNovice);
}

try{
     File file = new File("novice.rdf");

      if(!file.exists()){
        file.createNewFile();
     }

     FileWriter fileWritter = new FileWriter(file.getName());
     BufferedWriter bufferWritter = new BufferedWriter(fileWritter);
     model.write(bufferWritter);
     bufferWritter.close();

}catch(IOException ee){
     ee.printStackTrace();
}
```

RDF datoteka, ki se generira, ko ustvarimo novo novico:

```
<rdf:RDF
    xmlns:news="http://opendata.lavbic.net/news/"
    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns:dct="http://purl.org/dc/terms/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
    xmlns:sioc="http://rdfs.org/sioc/ns#"
    xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#" >
  <rdf:Description
      rdf:about="http://www.testNovice.local/novice/novica-0-7Yp8g">
    <dct:title
        rdf:datatype="http://www.w3.org/2001/XMLSchema#string">Naslov
        novice</dct:title>
    <sioc:UserName
        rdf:datatype="http://www.w3.org/2001/XMLSchema#string"> A.
        R.</sioc:UserName>
    <dct:abstract
        rdf:datatype="http://www.w3.org/2001/XMLSchema#string">
        Povzetek ...</dct:abstract>
    <sioc:Content
        rdf:datatype="http://www.w3.org/2001/XMLSchema#string">
        Vsebina ...</sioc:Content>
    <dct:created
        rdf:datatype="http://www.w3.org/2001/XMLSchema#date">
        2014-01-20 16:46:51</dct:created>
    <rdfs:seeAlso
        rdf:datatype="http://www.w3.org/2001/XMLSchema#string">
        http://www.testNovice.local/novice/novica-0</rdfs:seeAlso>
  </rdf:Description>
</rdf:RDF>
```

Lokalni podatkovni model novic



Slika 5.1: Lokalni podatkovni model novic

V tretjem delu programa pa smo omogočili še poizvedovanje po podatkih, ki jih shranimo lokalno v rdf datoteko. Ta možnost ima že prednapisane poizvedbe, vendar omogoča uporabniku pisanje svojih poizvedb. Tudi tukaj smo omogočili izvoz podatkov v datoteko, pri čemer se podatki shranijo bodisi v HTML, JSON ali RDF datoteko. Koda, ki izvede poizvedbo, je napisana s pomočjo knjižnice Jena. Tudi tukaj pa imamo možnost izbire načina prikaza rezultata poizvedbe, bodisi v HTML, JSON ali RDF obliki.

Koda, ki pridobi podatke iz lokalne rdf datoteke:

```
String queryString = prefixString + "\n" +
    textAreaNapisiPoizvedbo.getText();
QueryExecution qexec = null;
try{
     Query query = QueryFactory.create(queryString);
     qexec = QueryExecutionFactory.create(query, model);
     ResultSet results = qexec.execSelect();
     if(comboBoxNacinPrikazaLocal.getSelectedItem() == "HTML"){
          textPane.setContentType("text/html");
          textPane1.setText(ResultSetFormatter.asText(results));
     }else if(comboBoxNacinPrikazaLocal.getSelectedItem() ==
         "JSON"){
          ByteArrayOutputStream b = new ByteArrayOutputStream();
          ResultSetFormatter.outputAsJSON(b, results);
          textPane1.setText(b.toString());
     }else{
          textPane1.setText(ResultSetFormatter.asXMLString(results));
     }
}catch(Exception ee){
     textPane1.setText(ee.toString());
}
finally{
     qexec.close();
}
```

Program, ki smo ga napisali, je bil narejen v projektu Maven in preko Git orodja, ki ga ponuja Eclipse okolje, prenešen na spletno stran github [35]. Izvorna koda programa se nahaja na naslovu [36].

5.3 Primer poizvedovanja iz več oddaljenih SPARQL točk hrati

Poizvedba:

```
PREFIX owl: <http://www.w3.org/2002/07/owl#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX movie: <http://data.linkedmdb.org/resource/movie/>
PREFIX dcterms: <http://purl.org/dc/terms/>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX dbpprop: <http://dbpedia.org/property/>
PREFIX dbpedia-owl: <http://dbpedia.org/ontology/>

SELECT ?movie ?movieLabel ?subject ?releaseDate ?dbpediaLink
    ?movieTitle ?noLangTag ?producer ?movieDistributor ?runtime
    ?name ?desc ?actor ?abstractLen ?actorBPlace ?birthYear
    ?strConatActorData WHERE {
    {
    SERVICE <http://dbpedia.org/sparql> {
        SELECT ?actor ?name ?desc ?actorBday ?actorBPlace
            ?actorDday ?birthYear ?strConatActorData ?abstractLen
            WHERE {
           ?actor rdfs:label ?name .
           ?actor dbpprop:shortDescription ?desc .
           ?actor dbpedia-owl:abstract ?abstract .
           FILTER (lang(?abstract) = "en")
           BIND (CONCAT("St. znakov povzetka: ", strlen(?abstract))
               AS ?abstractLen)
           OPTIONAL { ?actor dbpedia-owl:birthDate ?actorBday .
                      BIND (SUBSTR(str(?actorBday), 1, 4) AS
                          ?birthYear)
```

```
        }
        OPTIONAL { ?actor dbpprop:dateOfDeath ?actorDday }
        OPTIONAL { ?actor dbpprop:birthPlace ?actorBPlace }
        FILTER (regex(str(?desc), "actor", "i"))
        BIND (CONCAT(?name, ", Leto rojstva: ", ?birthYear, ",
            Rojstni kraj: ", ?actorBPlace) AS ?strConatActorData)
    }
    LIMIT 70
}
}
UNION
{
SERVICE <http://data.linkedmdb.org/sparql> {
    ?movie rdf:type movie:film ;
           rdfs:label ?movieLabel ;
           owl:sameAs ?dbpediaLink ;
           movie:initial_release_date ?releaseDate
    FILTER (regex(str(?dbpediaLink), "dbpedia", "i"))
    FILTER (?releaseDate >= "1960-01-01")
}
SERVICE <http://dbpedia.org/sparql> {
       ?dbpediaLink dcterms:subject ?subject ;
                    foaf:name ?movieTitle
       FILTER (langMatches(lang(?movieTitle), "en"))
       BIND (str(?movieTitle) AS ?noLangTag)
       ?dbpediaLink dbpedia-owl:producer ?producer .
       OPTIONAL { ?dbpediaLink dbpprop:distributor
           ?movieDistributor . }
       OPTIONAL { ?dbpediaLink dbpprop:runtime ?runtimeZac }
       BIND (str(?runtimeZac) AS ?runtime)
}  }
}
LIMIT 200
```

V zgornjem primeru smo prikazali uporabo naprednega gradnika za poizvedovanje iz oddaljenih SPARQL točk (porazdeljene poizvedbe - ukaz SERVICE). V primerjavi s SQL-om vidimo, da je glavna prednost poizvedovalnega jezika SPARQL poizvedovanje iz oddaljenih virov, ki omogoča uporabniku pridobiti naenkrat več različnih množic podatkov. V primeru smo prikazali uporabo poizvedbe, ki pridobi podatke iz dveh oddaljenih točk (DBpedia [3], ter LinkedMdb [4]), pri čemer smo uporabili tri ukaze SERVICE. Prvi SERVICE ukaz ter naslednja dva smo ločili z ukazom UNION, tako da v rezultatu dobimo podatke, ki so ločeni (leva ter desna stran ukaza UNION nista povezani med seboj). Znotraj prvega ukaza smo uporabili še ukaz OPTIONAL, ki omogoči, da se rezultat ne zavrne, tudi v primeru, ko se trojica znotraj OPTIONAL ukaza ne izvede. Prikazali smo uporabo ukaza BIND in vidimo, da se ta lahko uporabi bodisi znotraj ukaza OPTIONAL ali izven. S funkcijo CONCAT smo prikazali, kako je možno smiselno sestavljati nize. Težava, na katero smo naleteli, je število podatkov, ki jih lahko pridobimo z jezikom SPARQL. Trenutno je pri semantičnem spletu še vedno težava v tem, da ne moremo pridobiti velike količine podatkov naenkrat, predvsem takrat, ko poizvedujemo iz več oddaljenih točk naenkrat, zato je potrebno število podatkov omejiti (to smo storili z uporabo ukaza LIMIT). Znotraj prvega SERVICE ukaza smo uporabili tudi ukaz SELECT, vendar ta ukaz ni potreben (v naslednjih dveh SERVICE ukazih ga nismo uporabili). Večina oddaljenih točk ima omejeno število podatkov, ki jih lahko vrne, ali pa imajo nastavljen maksimalen čas za izvedbo poizvedbe, to pa zaradi tega, ker bi prvič kompleksne poizvedbe zavzele veliko virov na strežniku, drugič, ker lahko pride do težave v omrežju in oddaljena točka ne vrne rezultata ipd. Zaradi tega se je potrebno pri pisanju porazdeljenih poizvedb osredotočati samo na tiste podatke, ki jih potrebujemo. Na drugi strani pa imamo poizvedovalni jezik SQL, ki se uporablja za poizvedovanje po relacijskih podatkovnih bazah, vendar predvsem na lokalnih. Nekateri ponudniki SQL-a imajo možnost, da se lahko preko programov povežemo na neko oddaljeno bazo, vendar to ni bistvo SQL-a. Vlogo oddaljenega poizvedovaja ter poizvedova-

nja po podatkih, ki se nahajajo na različnih spletnih virih, ima SPARQL, ki je ključen jezik za poizvedovanje na semantičnem spletu.

V drugem in tretjem SERVICE ukazu smo prikazali, kako med seboj povezati dve oddaljeni točki. Med seboj smo povezali podatke iz LinkedMdb ter DBpedije, pri čemer smo najprej iz baze filmov (LinkedMdb) pridobili filme, ki imajo URL naslov do DBpedije ter so nastali po letu 1960. Da smo preverili, ali nek film vsebuje povezavo do DBpedije, smo uporabili predikat owl:sameAs, ki vsebuje podatek do neke druge spletne strani ter uporabili še FILTER ukaz, znotraj katerega smo napisali funkcijo regex, ki v spremenljivki ?dbpediaLink najde niz "dbpedia", kot tretji parameter pa smo podali še zastavico "i", ki funkciji pove, da naj ne loči med velikimi ter malimi črkami. V tretjem SERVICE ukazu pa smo se povezali na URL naslov oddaljene točke DBpedija, kjer smo kot predmet uporabili spemenljivko ?dbpediaLink, ki predstavlja URL naslov (podatek smo pridobili z oddaljene točke LinkedMdb) do filma, ki se nahaja na oddaljeni točki DBpedija. Nato smo z uporabo funkcije langMatches() ter znotraj nje funkcije lang() preverili, ali ima naslov filma jezikovno značko, ki lahko predstavlja katero koli različico angleškega jezika. Znotraj tega SERVICE bloka smo napisali še trojico, ki pridobi podatek o producentu filma, ter dve, kateri smo napisali znotraj OPTIONAL ukaza, v primeru, da ena izmed njiju ne velja, se rezultat za to poizvedbo ne bo zavrnil. Na koncu pa smo dodali še ukaz BIND, ki odstrani tip(http://dbpedia.org/datatype/second) iz spremenljivke ?runtimeZac. Na koncu poizvedbe smo dodali še ukaz LIMIT, da ne bi dobili opozorila s strani serverja, da je čas za poizvedbo potekel.

5.4 Java

Java je objektno usmerjen, prenosljiv programski jezik, ki se je v začetku imenoval Oak (hrast) in je bil razvit kot zamenjava za programski jezik C++. Podoben jezik je JavaScript, vendar ju ne smemo enačiti. Dandanes poznamo več vrste Jave, kot je J2SE, ki je standardna različica Jave za osebne

računalnike, Java ME, ki je različica Jave za mini naprave, kot so mobiteli, pametni televizorji ipd, tretja vrsta pa je Java EE, ki pa je poslovna različica Jave, nato pa poznamo še Java Card ter Java FX. Java se je v zadnjem desetletju zelo razširil v svetu računalništva in do danes postal eden najbolj priljubljenih programskih jezikov, ki se najpogosteje uporablja pri spletnih aplikacijah odjemalec-strežnik. Java ponuja zelo obsežno knjižnico, ki je zelo dobro dokumentirana, tako da omogoča lažje in kvalitetno objektno programiranje. Ko govorimo o objektnem programiranju, glavno vlogo igrajo objekti, ki so skupek podatkov, s katerimi želimo upravljati kot s celoto, vsak objekt pa pripada nekemu razredu.

Java je zelo zanimiv programski jezik tudi zaradi tega, ker deluje na vseh platformah. Vse, kar je potrebno storiti, je, da datoteko prenesemo na platformo, kjer mora biti nameščeno Java okolje (vendar to ne bi smel biti problem, kajti Java okolje uporabljamo vsak dan na različnih spletnih straneh). Najboljši dve alternativi za pisanje programov v Javi pa sta programa Eclipse in NetBeans [38].

5.5 Apache Jena

Jena je odprto kodno Java ogrodje, ki se uporablja za izgradnjo programov v semantičnem spletu ter povezanih podatkih (Linked Data). V programih se uporablja kot knjižnica, katero uvozimo v program, ter lahko potem z njeno uporabo delamo z RDF podatki. Za primer takih podatkov lahko vzamemo DBpedijo, ki omogoča, da so podatki iz Wikipedije shranjeni v RDF obliki. Jena nam omogoča, da lahko podatke pišemo in beremo iz RDF grafov. Grafi so predstavljeni kot abstraktni modeli, ki pa imajo vir podatkov bodisi iz datotek, podatkovnih baz, URL naslovov ali kombinacijo le teh. Po podatkih lahko poizvedujemo s SPARQL programskim jezikom, kot smo to naredili v sklopu diplomske naloge, ali spreminjamo oz. posodabljamo s SPARUL oz. SPARQL/Update, ki je dodatek poizvedovalnem jeziku SPARQL [41].

Jena podpira različne serializacije grafov, kot so RDF/XML, Turtle, N3,

N-Triple. V sklopu diplomske naloge smo opazili, da tudi Jena ne podpira vseh možnosti za popolno delo, saj smo imeli težavo z dostopom do oddaljene točke [1], pri čemer je bilo potrebno vnesti uporabniško ime in geslo, zato je včasih potrebno uporabiti tudi nekatere druge knjižnice. V našem primeru smo uporabili knjižnico Apache Http Client [42].

5.6 Eclipse

Eclipse je alternativa za pisanje programov v Javi, bodisi za preproste ali kompleksne aplikacije, in je integrirano razvojno okolje (IDE), ki vsebuje osnovno delovno okolje ter sistem za vtičnike, kot je WindowsBuilder, katerega smo uporabili pri izdelavi diplomske naloge, ter mnoge druge, s katerimi lahko prilagajamo svoje razvojno okolje. Eclipse je napisan večinoma v Javi in prvotno tudi namenjen pisanju programov v tem jeziku. Uporabnikom pa ne omogoča pisanja programov samo v Javi, temveč omogoča tudi ostale programske jezike, kot so Ada, C, C++, JavaScript, Python, Ruby on Rails, Fortran, Cobol, Groovy ipd. Uporablja se tudi za razvoj paketov, ki se lahko kasneje uporabijo v programu Mathematica. Program deluje na različnih operacijskih sistemih, kot so Windows, Linux, Solaris ter Mac OS X. Naš program je bil napisan v zadnji verziji Eclipsa - Kepler, ki je izšla v mesecu juniju 2013. Prva verzija programa pa je izšla leta 2004, od takrat naprej pa so vsako leto v mesecu juniju izdali novo različico programa [43].

5.7 Izboljšave in nadgradnja programa

Program, ki smo ga razvili v sklopu diplomske naloge, dela nad podatki iz oddaljenih SPARQL točk ter podatki oz. novicami, ki jih sami ustvarimo. Prva izboljšava, ki bi se lahko naredila, je, da bi program razširili in povezali z modelom novic na oddaljeni točki [1], kamor bi lahko preko programa dodajali nove novice. Po možnosti bi lahko še spreminjali in posodabljali vsebino podatkov, če bi bili administrator. Za urejanje novic bi bilo pametno dodati

obrazec za prijavo, preko katerega bi omogočili to funkcijo samo tistim, ki imajo avtoriteto, torej v tem primeru novinarjem, ki so zaposleni v podjetju. S tem, ko bi novico dodali preko programa v sistem, bi bilo smiselno narediti tudi povezavo s strežnikom, ki bi ob nastanku in objavi novice omogočila, da bi se ta pojavila na spletni strani, kjer so prikazane novice podjetja, v tem primeru 24ur [5]. Ob tem bi bilo potrebno omogočiti še urejevalnik vsebine, kjer bi bilo omogočeno dodajanje in spreminjanje slik, tabel, pisave in podobno.

Glede na to, da je program zastavljen tako, da je namenjen poizvedovanju po podatkih ter ustvarjanju novic, in ker vemo, da se v podatkih skriva veliko informacij, bi bilo smiselno in koristno implementirati zgornjo izboljšavo, s tem pa bi bil program uporaben za veliko dodatnih možnosti, kot je obdelava podatkov in prikaz statistike objave novic, komentarjev, glasovanj in podobno, če bi se osredotočili samo na problemsko domeno novic iz 24ur. Program bi bil s tema dvema izboljšavama že precej uporaben in primeren za resno delo.

Poglavje 6

Zaključek

Namen diplomske naloge je bil predstaviti napredne gradnike poizvedovalnega jezika SPARQL, ki je danes, ko je semantični splet že tako razširjen, nepogrešljiv poizvedovalni jezik za poizvedovanje po RDF podatkih. Obenem smo naredili primerjavo s poizvedovalnima jezikoma XQuery ter SQL, nato opisali RDF format, ki omogoča, da so podatki na spletu organizirani in lahko nato z njimi uspešno delamo, ter obenem predstavili načine za shranjevanje podatkov.

V glavnem delu pa smo se osredotočili na temo diplomske naloge – poizvedovalni jezik SPARQL, kjer smo podrobno opisali večino pomembnih funkcionalnosti ter naprednih gradnikov, ki jih ta ponuja in so nepogrešljive pri resnem delu, ter naredili primerjavo z jezikom SQL. Bistvo naloge pa je bilo izdelati program, ki prikazuje, kako se z naprednimi gradniki poizveduje po RDF podatkih. Celoten program stoji na problemski domeni novic iz 24ur [5] ter podatkov iz DBpedie, ter novicah, ki jih lahko s programom sami ustvarimo. Program ima prednapisane poizvedbe na omenjeni oddaljeni točki, pri čemer smo prikazali uporabo naprednega gradnika SERVICE, ki je zelo pomemben pri poizvedovanju s poizvedovalnim jezikom SPARQL, kajti ta nam omogoča, da lahko poizvedujemo naenkrat z več različnih oddaljenih točk in tako pridobimo podatke iz več kot enega vira hkrati.

V programu smo prikazali tako poizvedovanje kot shranjevanje podat-

kov, ki so shranjeni v RDF formatu, vendar pa se nam je ob koncu razvoja programa odprlo še veliko novih idej, kako bi program lahko izboljšali in nadgradili. Zato smo v poglavju o izboljšavah in nadgradnjah opisali, kako bi lahko program naredili boljši in s tem dobili program, ki bi bil primeren za konkretno delo, ne samo s poizvedovanjem po podatkih, temveč tudi uporabo statistike, ter dodajanje novic preko programa na splet, kar bi verjetno vzelo kar nekaj časa za razvoj, vendar bi se zagotovo izplačalo.

Literatura

[1] OpenData news endpoint, dostopno na:
http://opendata.lavbic.net/news/sparql

[2] DBpedia, dostopno na:
http://dbpedia.org/

[3] Virtuoso SPARQL Query Editor, dostopno na:
http://dbpedia.org/sparql

[4] Linked Movie Data Base, dostopno na:
http://data.linkedmdb.org/sparql

[5] 24ur, dostopno na:
http://www.24ur.com/

[6] Wikipedia, dostopno na:
http://en.wikipedia.org/

[7] Mohorič, Podatkovne baze. 2. popravljena izdaja. Ljubljana: BI-TIM, 2002. ISBN 961-6046-12-8.

[8] RDF Primer, dostopno na:
http://www.w3.org/TR/rdf-primer/

[9] Resource Description Framework (RDF), dostopno na:
http://en.wikipedia.org/wiki/Resource_Description_Framework

[10] Couchbase, dostopno na:
http://www.couchbase.com/kr/why-nosql/nosql-database

[11] Bob DuCharme, Learning SPARQL: Querying and Updating with SPARQL 1.1. 1st edition. Sebastopol, CA 95472: O'Reilly Media, 2011. ISBN: 978-1-449-30659-5.

[12] OpenData news, dostopno na:
http://opendata.lavbic.net/news/

[13] Postman, dostopno na:
http://www.getpostman.com/

[14] Alan Beaulieu, Learning SQL. 2nd edition. Sebastopol, CA 95472: O'Reilly Media, 2009. ISBN: 978-0-596-52083-0.

[15] RDF Vocabulary Description Language 1.0: RDF Schema, dostopno na:
http://www.w3.org/TR/rdf-schema/

[16] SPARQL 1.1 Query Language, dostopno na:
http://www.w3.org/TR/sparql11-query/#GroupPatterns

[17] SPARQL 1.1 Query Language, dostopno na:
http://www.w3.org/TR/sparql11-query/#propertypaths

[18] SPARQL Query Language for RDF, dostopno na:
http://www.w3.org/TR/rdf-sparql-query/#optionals

[19] ARQ - SPARQL Tutorial - Union, dostopno na:
http://jena.sourceforge.net/ARQ/Tutorial/union.html

[20] SPARQL Query Language for RDF, dostopno na:
http://www.w3.org/TR/rdf-sparql-query/

[21] SQL UNION Operator, dostopno na:
http://www.w3schools.com/sql/sql_union.asp

[22] SPARQL 1.1 Query Language, dostopno na:
http://www.w3.org/TR/sparql11-query/#assignment

[23] SPARQL 1.1 Query Language, dostopno na:
http://www.w3.org/TR/sparql11-query/#selectExpressions

[24] SPARQL 1.1 Query Language, dostopno na:
http://www.w3.org/TR/sparql11-query/#groupby

[25] SQL Aliases, Dostopno na:
http://www.w3schools.com/sql/sql_alias.asp

[26] Sparql Endpoints - W3C Wiki, dostopno na:
http://www.w3.org/wiki/SparqlEndpoints

[27] SPARQL 1.1 Query Language, dostopno na:
http://www.w3.org/TR/sparql11-query/#specifyingDataset

[28] SPARQL 1.1 Query Language, dostopno na:
http://www.w3.org/TR/sparql11-query/#specDataset

[29] SPARQL 1.1 Query Language, dostopno na:
http://www.w3.org/TR/sparql11-query/#rdfDataset

[30] SPARQL 1.1 Query Language, dostopno na:
http://www.w3.org/TR/sparql11-query/#construct

[31] Christof Strauch, NoSQL Databases Lectures, dostopno na:
https://oak.cs.ucla.edu/cs144/handouts/nosqldbs.pdf

[32] SQL INSERT INTO Statement, dostopno na:
http://www.w3schools.com/sql/sql_insert.asp

[33] SPARQL Update, dostopno na:
http://www.w3.org/Submission/SPARQL-Update/

[34] Richard E. Smith, Authentication: from passwords to public keys. Inc Boston, MA, USA, Addison-Wesley Longman Publishing Co., 2002. ISBN:0-201-61599-1.

[35] GitHub, dostopno na:
https://github.com/

[36] Izvorna koda programa, dostopno na:
https://github.com/scorpioalen/diplomskaNaloga

[37] Microsoft, What is RDF Blank Node, dostopno na:
http://msdn.microsoft.com/en-us/library/aa303696.aspx

[38] Java (programming language), dostopno na:
http://en.wikipedia.org/wiki/Java_(programming_language)

[39] Comparision of Triple Stores, dostopno na:
http://www.bioontology.org/wiki/images/6/6a/Triple_Stores.pdf

[40] Uniform Resource Identifier (URI): Generic Syntax, 2005, dostopno na:
http://greenbytes.de/tech/webdav/rfc3986.pdf

[41] Data - W3C, dostopno na:
http://www.w3.org/standards/semanticweb/data

[42] HttpClient, dostopno na:
http://hc.apache.org/httpclient-3.x/

[43] Eclipse (software), dostopno na:
http://en.wikipedia.org/wiki/Eclipse_(software)

[44] ARQ - Property Paths, dostopno na:
http://jena.sourceforge.net/ARQ/property_paths.html

[45] RDFa 1.1 Primer - Second Edition, dostopno na:
http://www.w3.org/TR/xhtml-rdfa-primer/

[46] LargeTripleStores - W3C Wiki, dostopno na:
http://www.w3.org/wiki/LargeTripleStores

[47] SPARQL 1.1 Property Path, dostopno na:
http://www.w3.org/TR/sparql11-property-paths/

[48] Peter Buneman, Mary Fernandez in Dan Suciu, UnQL: a query language and algebra for semistructured data based on structural recursion, 2000, št. 9, str. 76-110.

[49] Priscilla Walmsley, XQuery. 1st edition. Sebastopol, CA 95472: O'Reilly Media, 2007. ISBN: 978-0-596-00634-1.

[50] Semantični splet, dostopno na:
http://www.lavbic.net/delo-in-raziskovanje/semanticni-splet/

[51] XQuery 1.0 and XPath 2.0 Functions and Operators (2nd Edition), dostopno na:
http://www.w3.org/TR/xpath-functions/#regex-syntax