UNIVERZA V LJUBLJANI

FAKULTETA ZA RAČUNALNIŠTVO IN INFORMATIKO

Matej Rojko

Integracija povpraševanj nerelacijskih podatkovnih baz in doseganje visoke razpoložljivosti v računalniškem oblaku

MAGISTRSKO DELO

ŠTUDIJSKI PROGRAM DRUGE STOPNJE
RAČUNALNIŠTVA IN INFORMATIKE

MENTOR: prof. dr. Matjaž Branko Jurič

Ljubljana, 2014

IZJAVA O AVTORSTVU MAGISTRSKEGA DELA

Spodaj podpisani Matej Rojko, z vpisno številko **63080182**, sem avtor magistrskega dela z naslovom:

Integracija povpraševanj nerelacijskih podatkovnih baz in doseganje visoke razpoložljivosti v računalniškem oblaku.

S svojim podpisom zagotavljam, da:

- sem magistrsko delo izdelal samostojno pod mentorstvom prof. dr. Matjaža Branka Juriča,
- so elektronska oblika magistrskega dela, naslov (slov., angl.), povzetek (slov., angl.) ter ključne besede (slov., angl.) identični s tiskano obliko magistrskega dela,
- soglašam z javno objavo elektronske oblike magistrskega dela v zbirki »dela FRI«.

V Ljubljani, 14. dec 2014 Podpis avtorja:

Zahvala

Za strokovno vodenje in usmerjanje se zahvaljujem svojemu mentorju, prof. dr. Matjažu Branku Juriču.

Še posebej bi se rad zahvalil svojim staršem ter prijateljema Vidu Kanduču in Evi Verk za vsa priporočila in podporo skozi študij.

KAZALO

KAZALO SLIK

KAZALO TABEL

Seznam uporabljenih kratic

kratica	pojem	pomen
NoSQL	Not only SQL	Podatkovne baze »ne samo SQL«[1]
BLOB	Binary large object	Veliki binarni objekt
RAM	Random access memory	Bralno-pisalni pomnilnik
HTML	Hypertext markup language	Označevalni jezik za oblikovanje večpredstavnostnih dokumentov
W3C	World Wide Web Consortium	Konzorcij svetovnega spleta
URI	Uniform resource identifier	Enotni označevalnik vira
RDF	Resource description framework	Ogrodje za opisovanje virov
JSON	Javascript object notation	Notacija JavaScript objektov
XML	Extensible markup language	Razširljivi označevalni jezik
BSON	Binary JSON	Binarni JSON
YAML	YAML Ain't Markup Language	Format za serializacijo uporabniku berljivih podatkov
REST	Representational state transfer	Predstavitveni prenos stanja
SPARQL	SPARQL Protocol and RDF Query Language	Protokol in poizvedovalni jezik RDF
SOAP	Simple object access protocol	Protokol za spletne storitve, ki temelji na XML
MD5	Message digest	Zgoščevalna funkcija
MVC	Model – View – Controller	Model – Pogled – Krmilnik
IaaS	Infrastructure as a Service	Infrastruktura kot storitev
PaaS	Platform as a Service	Računalniško okolje kot storitev
SaaS	Software as a Service	Programje kot storitev
SOA	Service-oriented architecture	Storitveno usmerjena arhitektura
MOM	Message oriented middleware	Sporočilna vmesna oprema
GIS	Geographic Information System	Geografski informacijski sistem
PDF	Portable document format	Format, ki je neodvisen od računalniškega okolja
SLA	Service Level Agreement	Sporazum o ravni storitve

[1] V tem delu jih naslavljamo kot nerelacijske podatkovne baze.

Povzetek

Nerelacijske podatkovne baze NoSQL (Not only SQL) v kombinaciji z računalništvom v oblaku ponujajo shranjevanje in upravljanje velikih količin podatkov. Povpraševanje po več različnih tipih nerelacijskih podatkovnih baz in razširljivost sistema prinašata velike izzive. Za rešitev integracije povpraševanja nad različnimi tipi podatkovnih baz je v magistrski nalogi implementiran pristop pretvorbe dokumentov v zapise RDF/XML in oglaševanje slednjih na končnih točkah. Z uporabo enotnega poizvedovalnega jezika SPARQL (SPARQL Protocol and RDF Query Language) smo prikazali način integracije povpraševanj nad dvema različnima vrstama nerelacijskih podatkovnih baz, MongoDB in Neo4j.

Za hitrejše izvajanje operacij branja je v magistrski nalogi predlagana arhitektura z uporabo metode črepinjenja ter analiziran vpliv vertikalne oziroma horizontalne razširljivosti sistema. Na porazdeljeni sistem naprav smo shranjevali podatke in merili čas. Ugotovili smo, da metoda črepinjenja izboljša čas operacije branja, medtem ko se čas operacije pisanja z uporabo metode črepinjenja nekoliko poveča. Oba pristopa, integracijo povpraševanj in metodo črepinjena, smo prikazali na primeru.

Ključne besede

NoSQL, SPARQL, vertikalna razširljivost, horizontalna razširljivost, replikacija

Abstract

Non-relational databases NoSQL offer in combination with cloud computing abilities to store and manage large amounts of data. Querying over multiple NoSQL databases and system scalability can be major challenges. To solve the integration of queries over different types of NoSQL databases, we have implemented the approach of converting documents into RDF/XML formats and advertising them on the endpoint. By using common query language SPARQL (SPARQL Protocol and RDF Query Language), we have shown the method of query integration over two different types of non-relational NoSQL databases, MongoDB and Neo4j.

To accelerate the execution of read operations, we have proposed an architecture using the sharding method. We have analyzed the impact of vertical or horizontal scalability of the system. We have stored data on distributed storage and measured the time. We have found out that the shrading method improves the read operations time, while the time for write operations slightly increases. We have demonstrated both approaches, query integration and shrading method, on an example.

Keywords

NoSQL, SPARQL, Vertical Scalability, Horizontal Scalability, Replication

1. Uvod

Nerelacijske podatkovne baze so pridobile pomen, saj so se povečale zahteve po shranjevanju nestrukturiranih podatkov in izvajanju učinkovitih analiz nad temi podatki. Z uporabo nerelacijskih podatkovnih baz pridobimo večjo fleksibilnost, saj večina predstavnikov teh podatkovnih baz ne zahteva vnaprej definirane podatkovne sheme. Z namestitvijo instanc na horizontalno porazdeljeni sistem naprav, nerelacijske podatkovne baze učinkovito rešujejo problem razširljivosti (*angl. Scalability*). Poznamo štiri glavne skupine: *dokument (angl. Document), graf (angl. Graph), ključ-vrednosti (angl. Key-Value)* in *vrste razširljivi zapisi (angl. Column Family Store)*.

Izvajanje povpraševanj in njihova integracija nad različnimi tipi nerelacijskih podatkovnih baz predstavlja velik izziv. Ta se pojavi zaradi uporabe unikatnega poizvedovalnega jezika različnih proizvajalcev teh baz. Za povpraševanje nad različnimi podatkovnimi bazami bi morali na vsaki izmed podatkovnih baz izvesti poizvedbo in rezultate poizvedb pozneje programsko združiti v celoto. Namesto tega bomo v magistrski nalogi uporabili rešitev, ki temelji na enotnem poizvedovalnem jeziku SPARQL [1], ki iz različnih končnih točk samodejno združi pridobljene rezultate. Za uporabo tega poizvedovalnega jezika moramo implementirati enega od pristopov, t.j. premik poizvedb bližje k podatkom oziroma premik podatkov bližje k poizvedbam. Prvi od nas zahteva pretvorbo poizvedbe v poizvedovalni jezik izbrane podatkovne baze, medtem ko drugi način predvideva pretvorbo podatkov v zapis RDF/XML in oglaševanje slednjih na končni točki.

Za shranjevanje velikih količin podatkov in hitro izvajanje operacije branja je potrebna razširljivost sistema. Razširljivost je zmožnost prilagajanja sistema na povečanje zahtev [2]. Poznamo vertikalno oziroma horizontalno razširitev. Prva predvideva povečanje količine virov ene naprave, medtem ko druga zahteva namestitev instanc nerelacijske podatkovne baze na večje število povezanih naprav. V primeru izbire horizontalne porazdelitve sistema moramo na podatkovnem nivoju izvajati metodo črepinjenja. Ta z izbiro ustreznega ključa črepinjenja zagotavlja drobljenje podatkov na gručo porazdeljenih naprav. Naprave, uporabljene pri horizontalni porazdelitvi sistema, so fizične ali navidezne. Navidezne naprave običajno proti plačilu lahko najamemo na javnih ponudnikih storitev v oblaku (npr. Amazon AWS, Microsoft Azure, Google Cloud). Za preprečitev izgube podatkov nerelacijske podatkovne baze z gručo instanc omogočajo namestitev replikacije sistema v realnem času.

V magistrski nalogi bomo naslovili problem povpraševanja nad različnimi tipi nerelacijskih podatkovnih baz in predlagali rešitev, ki temelji na pretvorbi podatkov v dokumente zapisa

RDF/XML ter oglaševanje le-teh na končni točki. Za analizo učinkovitosti vpliva vertikalne oziroma horizontalne porazdelitve sistema na izvajanje operacije branja in pisanja bomo na navidezne naprave, gostujoče na javnem ponudniku storitev v oblaku Amazon AWS, shranjevali dokumente in merili čas izvajanja. Predvidevamo, da se bo čas izvajanja spreminjal v odvisnosti od izbire tipa navidezne naprave ter ključa črepinjenja. Analizirali bomo čas operacije branja in pisanja podatkov na eno instanco podatkovne baze, nameščeno na navidezni napravi tipa *t2.micro* oziroma *t2.medium* na javnem ponudniku storitev v oblaku Amazon AWS. Drugo analizo bomo izdelali na porazdeljenem sistemu navideznih naprav tipa *t2.micro* oziroma *t2.medium*, kjer bomo pri uporabi metode črepinjenja za ključ izbrali izvleček dokumenta (*angl. Hash Key*). Pri tretji analizi bomo na isti arhitekturi za ključ metode črepinjenja izbrali ključ z definiranim obsegom (*angl. Range Key*). Za zagotavljanje visoke odzivnosti sistema bomo analizirali delovanje replikacije nad navideznimi napravami, gostujočimi na dveh različnih ponudnikih storitev v oblaku Amazon AWS in Microsoft Azure. Za namen povezave vseh omenjenih delov ter shranjevanja velikih količin podatkov bomo predlagali arhitekturo, ki predvideva uporabo metode črepinjenja ter replikacije na porazdeljenem sistemu navideznih naprav.

Zgradba magistrske naloge je sledeča. V drugem poglavju pregledamo različne tipe nerelacijskih podatkovnih baz ter opišemo lastnosti in primere uporabe za vsakega izmed njih. Sledi poglavje o kratkem pregledu računalništva v oblaku in arhitekturnih gradnikih. V tem poglavju razložimo kategorizacijo računalništva v oblaku na storitvene in namestitvene modele ter obrazložimo večnivojsko arhitekturo ter programski model MVC. Četrto poglavje opisuje integracijo povpraševanja med različnimi nerelacijskimi podatkovnimi bazami. V prvem delu tega poglavja analiziramo integracijo z uporabo neposrednega dostopa do podatkovnih baz, integracijo z uporabo storitev, integracijo z uporabo ETL ter integracijo s semantičnim spletom in ontologijo. Pri integraciji s semantičnim spletom opišemo ontologijo, nato pa predstavimo našo rešitev s pretvorbo dokumentov v zapis RDF/XML ter oglaševanje teh na končni točki AllegroGraph [3]. V petem poglavju predlagamo arhitekturno zasnovo rešitve za integracijo povpraševanja nad različnimi vrstami nerelacijskih podatkovnih baz. Arhitektura predvideva shranjevanje velikih količin podatkov na porazdeljen sistem navideznih naprav, gostujočih na dveh javnih ponudnikih storitev v oblaku Amazon AWS in Microsoft Azure.

Za učinkovito shranjevanje podatkov v šestem poglavju opišemo metodo črepinjenja, algoritma, po katerih metoda deluje, ter dobre in slabe prakse pri izbiri ključa črepinjenja. V nadaljevanju poglavja naredimo natančno analizo vpliva vertikalne oziroma horizontalne razširljivosti sistema. Po analizi sledi predstavitev ugotovitev in v zadnjem delu tega poglavja analiziramo različne topologije replikacije in možnost uporabe navideznih naprav, gostujočih na dveh javnih ponudnikih storitev v oblaku – Amazon AWS in Microsoft Azure ter v lokalnemu omrežju. V

sedmem poglavju predstavimo implementacijo rešitev integracije različnih vrst nerelacijskih podatkovnih baz in analize vertikalne oziroma horizontalne razširitve sistema. Sledijo zaključek ter uporabljena literatura.

2. Pregled nerelacijskih podatkovnih baz

V svetu nerelacijskih podatkovnih baz moramo opustiti misel, pri kateri je ena nerelacijska podatkovna arhitektura primerna za shranjevanje vseh različnih vrst podatkov. Prednost omenjenega pristopa je predvsem v tem, da so nerelacijske podatkovne baze optimizirane za shranjevanje in upravljanje z vnaprej določeno vrsto podatkov. Vrste nerelacijskih podatkovnih baz se poleg zapisa podatkov, ki jih hranijo, ločijo tudi glede na primere uporabe. Za razliko od relacijskih podatkovnih baz nerelacijske podatkovne baze ne zahtevajo načrtovanja podatkovne sheme (*angl. Database Schema*). Kljub temu pa moramo natančno preučiti, katero vrsto nerelacijske podatkovne baze bomo izbrali za podporo rešitvi domenskega problema [4]. V spodnji tabeli [Tabela 1] je predstavljen kratek pregled štirih vrst nerelacijskih podatkovnih baz:

Vrsta podatkovne baze	***Opis***	***Primeri uporabe***
Ključ-vrednosti	Dvodimenzionalna tabela na logičnem nivoju prikazuje pomen nerelacijske podatkovne baze ključ-vrednosti. Ključ je predstavljen z nizom znakov, ki enolično določajo vrednost, predstavljeno z binarnim objektom BLOB (*angl. Binary large object*).	Slovar, shranjevanje multimedijskih vsebin …
Graf	Nerelacijska podatkovna baza graf vsebuje množico vozlišč in povezav, kjer kombinacija teh predstavlja graf. Nerelacijska podatkovna baza *graf* z algoritmi nad grafi omogoča izvajanje analiz kompleksnih povezav.	Socialna omrežja, poizvedbe kot so prijatelji – prijateljev. Analiza povezav v bioinformatiki – projekt Bio4j.
Vrste-razširljivi zapisi	Nerelacijska podatkovna baza *vrste-razširljivi zapisi* združuje posamezne stolpce v družine stolpcev (*angl. Column-family)*. Z uporabo tega pristopa je omogočeno hitro	Analiza podatkov geografskega informacijskega sistema (*angl. Geographic information system*). Analiza velikih tabel (*angl. BigTable)*.

	spreminjanje vrednosti stolpcev, saj se združeni podatki nahajajo na isti lokaciji na trdem disku.	
Dokument	Nerelacijska podatkovna baza dokument vsebuje dokumente z metapodatki (*angl. Metadata*), ki opisujejo vsebino dokumentov. S samodejnim indeksiranjem podatkov omogoča hitro iskanje po vsebini dokumentov.	Usmerjeno oglaševanje (*angl. Targeting Advertising*). Lokalno shanjevanje kompleksnih objektov ter možnost poznejše sinhronizacije s strežnikom.

Tabela 1: Primeri uporabe nerelacijskih podatkovnih baz.

2.1 Nerelacijska podatkovna baza *ključ-vrednosti*

Nerelacijsko podatkovno bazo *ključ-vrednosti* lahko na logičnem nivoju predstavimo kot dvodimenzionalno tabelo [Slika 1]. Ključ je predstavljen z enostavnim nizom znakov, ki enolično določajo vrednost predstavljeno z binarnim objektom BLOB [5].

Ključ	Vrednost
…	…

Slika 1: Logična predstavitev nerelacisjke podatkovne baze *ključ-vrednosti*.

2.1.1 Osnove nerelacijske podatkovne baze *ključ-vrednosti*

Za enostavnejše razumevanje podatkovne baze *ključ-vrednosti* si poglejmo primer kjer upodobimo slovar. Slovar je določen s ključem in pripadajočim objektom [Tabela 2]. Ključ je predstavljen s tekstovnim besedilom, ki ga lahko definiramo na različne načine [4], med drugim kot:

- absolutna pot,
- relativna pot,
- izvleček, ki je generiran iz vrednosti objekta,

- klic spletne storitve REST ter
- poizvedba SQL.

Podatkovna baza *ključ-vrednosti* je primerna za shranjevanje poljubnih tipov podatkov. V osnovi mora aplikacija zagotavljati pravilno upravljanje z relacijami med ključi in vrednostmi, saj je dostop do vrednosti omogočen le z navedbo unikatnega ključa.

Primeri uporabe	***Ključ***	***Vrednost***
Ime slike	slika.jpg	Binarna slika
Spletni naslov	http://www.fri.uni-lj.si/	Spletna stran HTML
Pot do datoteke	C:\\dokument.pdf	Dokument PDF
Izvleček MD5	d21097b807cbbd278c2a2200ecb980ca	Rojko
Klic spletne storitve REST	http://webservices.amazon.com/onca/xml?Service=AWS	[{ime: Matej, priimek: Rojko}]
Poizvedba SQL	Select Priimek from tabela where id=1	<Priimek>Rojko</Priimek>

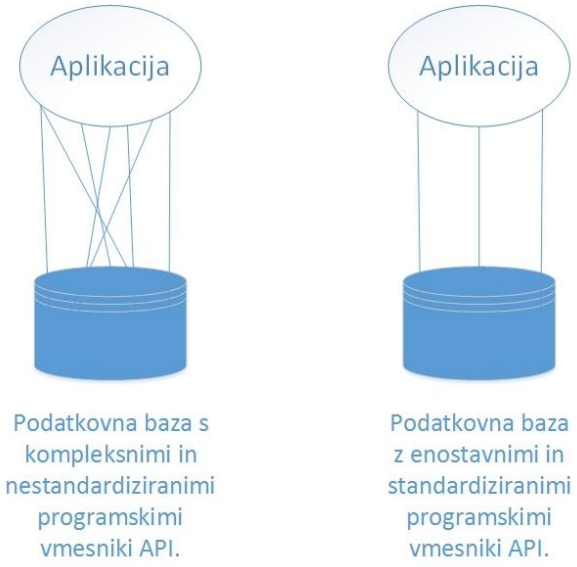
Tabela 2: Primeri uporabe ključev in vrednosti v podatkovni bazi *ključ-vrednosti*.

Prednosti uporabe podatkovne baze *ključ-vrednosti* so:

- razvijalci se zaradi enostavnosti uporabe lahko osredotočijo na kakovost delovanja aplikacij,
- razširljivost in zanesljivost sistema ter
- ponovna uporabljivost podatkovne arhitekture.

Nerelacijska podatkovna baza *ključ-vrednosti* ne podpira pisanja kompleksnih poizvedb, zaradi česar lahko pospešimo iskanje vrednosti, ki jih določajo ključi. Zaradi svoje enostavnosti so razvijalci lahko osredotočeni na druge parametre, ki izboljšujejo kakovost storitev (npr. prenos in shranjevanje 100.000.000 objektov).

Nerelacijska podatkovna baza *ključ-vrednosti* zaradi preprostega načina shranjevanja podatkov omogoča razvoj standardiziranih programskih vmesnikov in ponovno uporabljivost podatkovne baze v drugih aplikacijah. Nizka ponovna uporabljivost pri svojem delovanju uporablja veliko nestandardiziranih programskih vmesnikov med podatkovno bazo in aplikacijo (prikazano na levi strani, [Slika 2]). Medtem ko visoka ponovna uporabljivost (prikazano na desni strani, [Slika 2]) implementira le standardizirane klice *PUT, GET, DELETE*. Te klice lahko brez večjih težav ponovno implementiramo na drugih aplikacijah, ki se sklicujejo na iste podatkovne vire.



Slika 2: Nizka ponovna uporabljivost (levo) in visoka ponovna uporabljivost (desno).

2.1.2 Primer uporabe nerelacijske podatkovne baze *ključ-vrednosti*

Kot primer uporabe lahko navedemo iskalnik Google, ki za indeksiranje spletnih strani uporablja algoritem, imenovan Googlebot [6]. Ta algoritem sodi v skupino algoritmov za indeksiranje spletnih strani, ki jih s skupnim generičnim izrazom imenujemo pajek (*angl. Crawler*). Delovanje algoritma, ki ga uporablja Googlebot, je v grobem definirano v dveh korakih:

- obisk spletne strani in indeksiranje podatkov na spletni strani ter
- nadaljevanje indeksiranja podatkov na vseh podstraneh, ki so navedene kot povezave na osnovni strani.

Vsaka beseda na spletni strani predstavlja indeks/ključ, ki vsebuje informacijo o dokumentih in lokacijo v besedilu, kjer se je določena beseda pojavila. Ko uporabnik vnese iskalni niz, se kot zadetek pojavijo spletne strani, ki vsebujejo iskano besedilo. Za pravilno razvrstitev dokumentov, torej prikaz tistih zadetkov, ki jih uporabnik išče, algoritem upošteva naslednje kriterije: število pojavitev, lokacijo besede, sinonime in rang spletne strani [7].

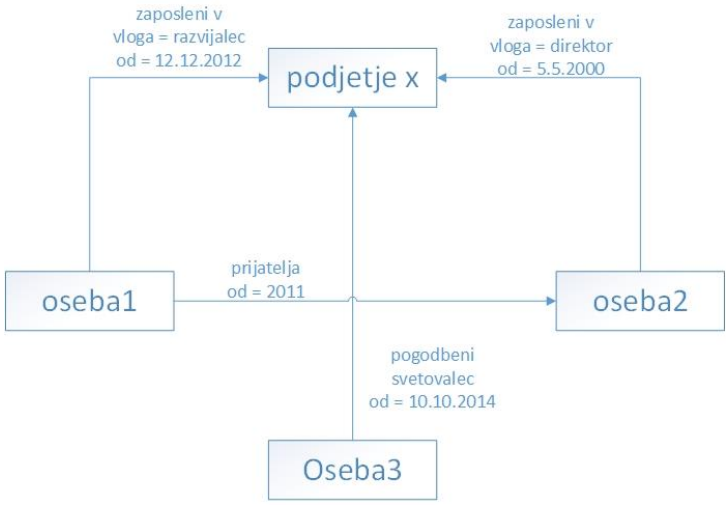
Predstavniki nerelacijske podatkovne baze *ključ-vrednosti* so:

- Berkeley DB,

- Amazon SimpleDB in
- Riak.

2.2 Nerelacijska podatkovna baza *graf*

Nerelacijska podatkovna baza *graf* je sistem, ki vsebuje množico vozlišč ter povezav, kjer kombinacija slednjih predstavlja graf [8]. Za opis povezav in vozlišč uporabljamo lastnosti, ki grafu dodajo večji semantični pomen [Slika 3].



Slika 3: Logična predstavitev nerelacijske podatkovne baze *graf*.

2.2.1 Osnove nerelacijske podatkovne baze *graf*

Nerelacijska podatkovna baza *graf* je sistem, predstavljen s pomočjo:

- vozlišč (*angl. Nodes*),
- povezav med vozlišči (*angl. Relationships*) in
- lastnosti (angl. *Properties*).

Nerelacijska podatkovna baza *graf* je namenjena aplikacijam, ki temeljijo na analizi grafov. V to skupino aplikacij sodijo številna področja, med drugim:

- socialna omrežja,
- iskanje na podlagi podobnosti in pravil,
- analize mutacij v bioinformatiki ter
- geografski informacijski sistemi.

Vsem tem področjem je skupna analiza kompleksnih povezav med vozlišči. Nerelacijska podatkovna baza *graf* je odlična rešitev za tovrstne probleme, saj je možno s pomočjo algoritmov nad grafi analizirati kompleksne povezave med vozlišči.

V vozlišča podatkovne baze *graf* lahko shranjujemo poljubne vrednosti (npr. osebne podatke, spletne strani, podatke o računalniških sistemih in omrežjih, informacije o bioloških celicah živih organizmov).

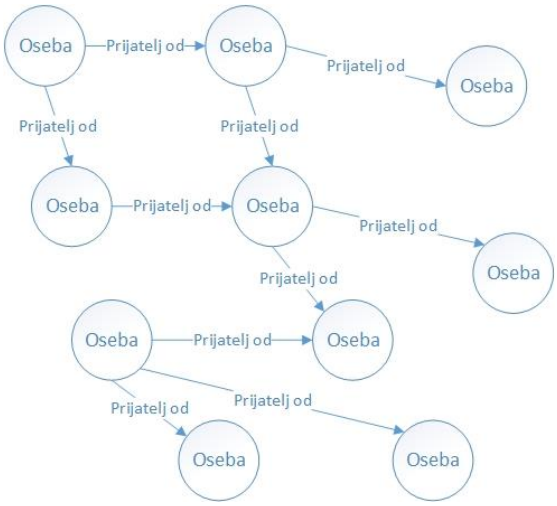
Primeri poizvedb in s tem povezanimi vprašanji, ki se lahko izvajajo nad to obliko podatkovne baze, vključujejo:

- V grafu izpiši najkrajšo povezavo med dvema vozliščema.
- Katera vozlišča imajo sosede s specifičnimi lastnostmi?
- Kot parameter poizvedbi podamo dve vozlišči in se vprašamo, kako podobni so si sosedje teh dveh vozlišč.

V nasprotju z relacijskimi podatkovnimi bazami, ki za shranjevanje relacij potrebujejo dodatne informacije, torej stolpce, ki nam vrnejo informacijo o povezavah, podatkovna baza *graf* vsakemu vozlišču dodeli notranji identifikator in te uporabi za namen povezav med vozlišči. Za razliko od ostalih nerelacijskih podatkovnih struktur ima podatkovna baza *graf* omejitve pri razširljivosti. Ta je lahko razširjena na več strežnikov in optimizirana za branje. Pisanje na tovrstni porazdeljeni sistem predstavlja veliko kompleksnost pri implementaciji. Podatkovna baza *graf* najbolje deluje, kadar ima na voljo zadostne količine delovnega spomina (*angl. RAM*), saj za analizo kompleksnih povezav potrebuje hiter dostop do vozlišč in lastnosti povezav.

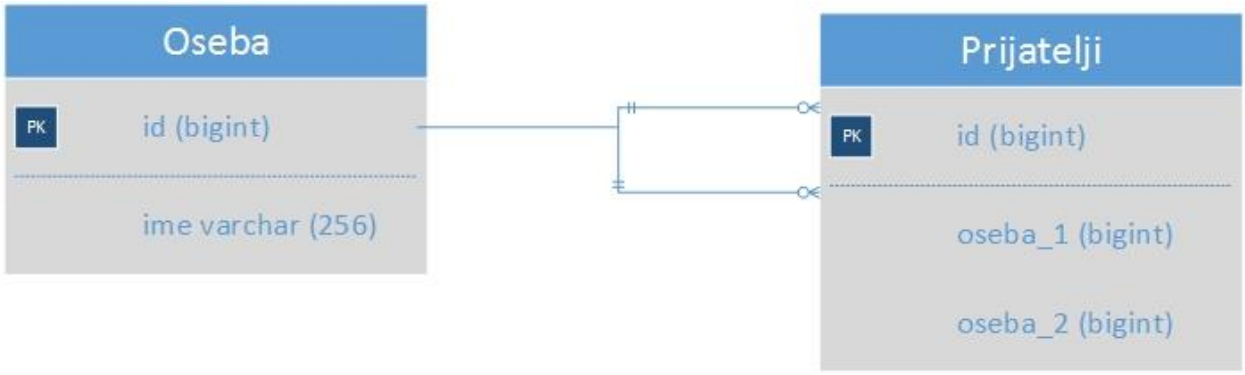
2.2.2 Primer uporabe nerelacijske podatkovne baze *graf*

Eden izmed primerov uporabe nerelacijske podatkovne baze graf so socialna omrežja, kjer želimo uporabnikom ponuditi listo morebitnih poznanih prijateljev. To poizvedbo imenujemo poizvedba prijatelji–prijateljev (*angl. Friends-of-friends*). Za lažje razumevanje si poglejmo primer, kjer predpostavimo, da obdelujemo uporabnika, ki ima sto prijateljev. Če bi želeli za vsakega izmed njegovih stotih prijateljev pridobiti seznam nadaljnjih sto prijateljev, torej prijateljev na drugem nivoju, bi poizvedba v relacijski podatkovni bazi vrnila skupno 10.000 vrstic (100 x 100). Lahko si predstavljamo, da analiza tovrstnih poizvedb predstavlja zahtevno operacijo za relacijske podatkovne baze, kar pa ne velja za podatkovno bazo *graf*, katere glavna prednost je analiza povezav [9]. Primer bi lahko predstavili z grafom vozlišč in povezav [Slika 4].



Slika 4: Graf prikazuje relacije poznanstev med prijatelji.

V relacijski podatkovni bazi bi te povezave predstavili z naslednjo podatkovno shemo [Slika 5].



Slika 5: Shema podatkovne baze (prijatelji–prijateljev).

V relacijski podatkovni bazi bi poizvedbo prijatelji–prijateljev na drugem in tretjem nivoju implementirali na način, prikazan spodaj [Programska koda 1].

```
select count(distinct pri2.*) from Prijatelji pri1
inner join Prijatelji pri2 on pri1.oseba_1 = pri2.oseba_2
where pri1.oseba_1 = ?

select count(distinct pri3.*) from Prijatelji pri1
inner join Prijatelji pri2 on pri1.oseba_1 = pri2.oseba_2
inner join Prijatelji pri3 on pri2.oseba_1 = pri3.oseba_2
where pri1.oseba_1 = ?
```

Programska koda 1: Poizvedbi prijatelji–prijateljev na drugem in tretjem nivoju.

Ta pristop je popolnoma pravilen, vendar glavno težavo predstavljajo povezave (*angl. Join*) v relacijskih podatkovnih bazah. Pri izločanju podatkov moramo najprej narediti vse povezave med primarnimi in tujimi ključi in pozneje med tem naborom podatkov izločiti tiste, ki ne ustrezajo pogojem. Tak način pri veliki količini podatkov in večnivojskih povezavah predstavlja veliko časovno kompleksnost.

Za razliko od relacijskih podatkovnih baz, lahko za učinkovito analizo povezav v nerelacijski podatkovni bazi *graf* uporabimo algoritem *Graph traversals* [10]. Algoritem pri svojem delovanju upošteva le tista vozlišča, ki ustrezajo pogojem. Za izvedbo poizvedbe nam predhodno ni potrebno izvajati časovno zahtevne operacije združevanja nad celotnim naborom podatkov. Torej v poizvedbi izberemo začetno vozlišče in pogoje, ki jih bo algoritem upošteval pri izbiri oziroma izločanju vozlišč.

Zaradi drugačnega načina delovanja lahko v nerelacijsko podatkovno bazo *graf* shranimo velike količine podatkov in nad njimi učinkovito izvajamo poizvedbe. V knjigi [9] so merili čase za poizvedbo prijatelji–prijateljev petega nivoja na relacijski podatkovni bazi MySQL in jih primerjali s časi na nerelacijski podatkovni bazi *graf* Neo4j. Avtorji so prišli do zaključka, da je poizvedba z uporabo nerelacijske podatkovne baze *graf* kar 10 milijonkrat hitrejša od poizvedbe nad podatki, shranjenimi v relacijski podatkovni bazi MySQL.

Predstavniki nerelacijske podatkovne baze *graf* so:

- Neo4j,
- InfoGrid in
- Infinitive Graph.

2.3 Nerelacijska podatkovna baza *vrste-razširljivi zapisi*

Nerelacijsko podatkovno bazo *vrste-razširljivi zapisi* lahko na logičnem nivoju predstavimo z razpredelnico. V te nerelacijski podatkovni bazi so stolpci združeni v družine stolpcev [Slika 6]. Družine stolpcev so na trdi disk shranjene skupaj na isti lokaciji. To omogoča hitro spreminjanje vrednosti v stolpcih [11].

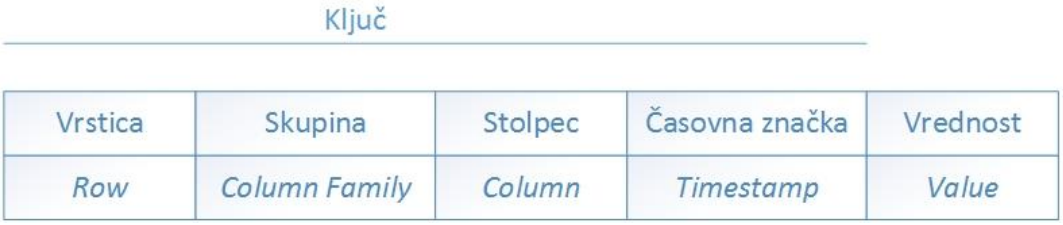
Slika 6: Predstavitev podatkov v nerelacijski podatkovni bazi *vrste-razširljivi zapis*.

2.3.1 Osnove nerelacijske podatkovne baze *vrste-razširljivi zapisi*

Za poizvedovanje nad podatki lahko uporabimo poizvedovalni jezik CQL (*Cassandra Query Language*) [12] oziroma programski model MapReduce [13], ki omogoča paralelno procesiranje poizvedb nad gručo horizontalno porazdeljenih naprav. Ta model je definiran z dvema osnovnima korakoma:

- ***Map*** – razdeli celotno operacijo na podoperacije, ki jih paralelno izvaja na različnih strežnikih, povezanih v eno podatkovno arhitekturo ter
- ***Reduce*** – priskrbi sestavljen rezultat iz vseh rezultatov funkcije map, ki smo jih prejeli z različnih strežnikov/vozlišč.

Za naslavljanje podatkov v nerelacijski podatkovni bazi *vrste-razširljivi zapisi* uporabljamo sestavljene ključe iz kombinacije vrstic, družine stolpcev, stolpcev ter časovnih značk [4]. Časovne značke omogočajo shranjevanje ločenih vrednosti za enako kombinacijo vrstic in stolpcev [Slika 7].



Slika 7: Prikaz sestavljenega ključa v podatkovni bazi *vrste-razširljivi zapisi*.

Prednost te nerelacijske podatkovne baze se pokaže, kadar le manjši odstotek celic (*angl. Sparse Matrix)* vsebuje podatke. Za prazne celice nerelacijska podatkovna baza *vrste-razširljivi zapisi* ne shranjuje ničelnih vrednosti (*angl. Null)*. Torej podatki na disku niso zapisani, kot to velja za

relacijsko podatkovno bazo SQL [5]. Horizontalna razširljivost te nerelacijske podatkovne baze je enostavna, saj omogoča ločeno shranjevanje družine stolpcev v različne podatkovne centre.

2.3.2 Primer uporabe nerelacijske podatkovne baze *vrste-razširljivi zapisi*

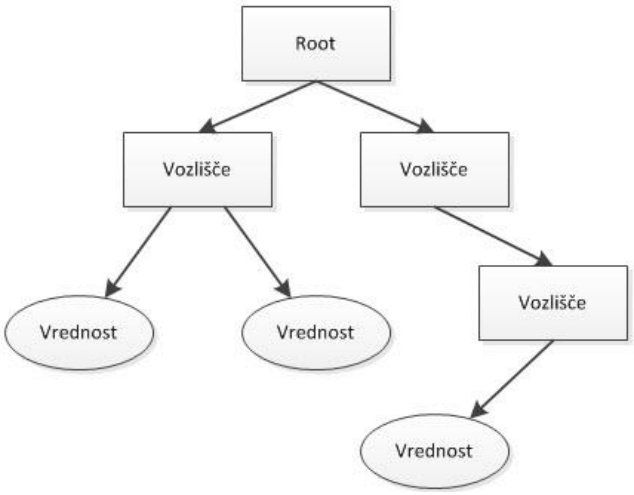
Za to skupino nerelacijskih podatkovnih baz poznamo kar nekaj primerov uporabe. Med znane spletne aplikacije, ki uporabljajo to vrsto nerelacijske podatkovne baze, sodita Google Analytics in Google Maps. Pri razvoju geografskega informacijskega sistema karta s satelitskimi posnetki vsebuje koordinate. Koordinate s pomočjo družin stolpcev združujemo v manjša območja z dodatnimi podatki (npr. slike okolice, interesne točke). Ko se uporabnik zanima za izbrano območje na karti je z nerelacijsko podatkovno bazo *vrste-razširljivi zapisi* s pomočjo družin stolpcev enostavno pridobiti podatke o območju. Te iste poizvedbe lahko implementiramo tudi drugače (npr. MongoDB Geospatial Indexes and Queries), vendar je nerelacijska podatkovna baza *vrste-razširljivi* zapisi optimizirana za združevanje podobnih stolpcev in s tem hitro pridobitev rezultatov o izbrani lokaciji in okolici.

Predstavniki nerelacijske podatkovne baze *vrste-razširljivi* zapisi so:

- Apache Cassandra,
- HBase in
- Google BigTable (dostopen kot del Google App Engine).

2.4 Nerelacijska podatkovna baza *dokument*

Nerelacijska podatkovna baza je namenjena shranjevanju samo-strukturiranih (*angl. Self-structured*) dokumentov, torej dokumentov z metapodatki [14]. Za logično predstavitev nerelacijske podatkovne baze *dokument* lahko uporabimo drevesno strukturo [Slika 8]. Samodejno indeksiranje vstavljenih dokumentov omogoča gradnjo drevesne strukture in s tem hitro iskanje po vsebini dokumentov.



Slika 8: Drevesna struktura nerelacijske podatkovne baze *dokument.*

2.4.1 Osnove nerelacijske podatkovne baze *dokument*

Podatkovna baza *dokument* se od ostalih razlikuje v tem, da ne shranjuje in vrača objektov BLOB. V to vrsto nerelacijske podatkovne baze lahko shranjujemo dokumente standardiziranih zapisov, med drugimi [8]:

- XML (angl. *Extensible Markup Language*),
- JSON (angl. *JavaScript Object Notation*),
- BSON (angl. *Binary JSON*),
- YAML (*angl. YAML Ain't Markup Language*),
- HTML (*angl. Hypertext markup language*) ali
- PDF (*angl. Portable Document Format*).

Med shranjevanjem objektov poskrbi za samodejno indeksiranje in s tem omogoči hitro iskanje vsake vstavljene vrednosti. Za namen indeksiranja in iskanja po vrednostih je vsakemu vstavljenemu dokumentu samodejno dodeljen enostavni ključ, ki predstavlja enolično identifikacijo dokumenta.

Podatki v dokumentih so zaradi samodejnih indeksov predstavljeni kot drevesna struktura, s pomočjo katere podatkovna baza *dokument* izlušči podatke, ki jih potrebuje za namen vračanja rezultatov poizvedbam. Kot primer lahko navedemo uporabnika, ki želi izpisati iskano poglavje v knjigi. Podatkovna baza *dokument* bo s pomočjo indeksov izluščila ustrezno poglavje in ga vrnila kot rezultat poizvedbe. To pomeni, da za namen poizvedb ne potrebujemo prenašati celotne vsebine dokumentov v delovni pomnilnik.

Drevesna struktura je predstavljena z vrhnjim elementom, na katerega so pripeta ostala vozlišča. Vsako podrejeno vozlišče vsebuje informacije o navigaciji od vrhnjega vozlišča do poljubne vrednosti [Slika 8]. Nerelacijska podatkovna baza *dokument* lahko hitro izloči tiste veje drevesa, ki ne ustrezajo iskalnim parametrom. Vrednosti so predstavljene v listih drevesa.

V podatkovni bazi dokument so dokumenti povezani v skupine, poimenovane zbirke (*angl. Collections*). Te zbirke si lahko predstavljamo kot mape v operacijskem sistemu UNIX ali Windows. Za shranjevanje velikih količin podatkov so lahko zbirke uporabljene na vrsto različnih načinov:

- hierarhični pristop shranjevanja dokumentov (primer drevesnih struktur),
- logično združevanje podobnih dokumentov v isto kolekcijo ali
- shranjevanje poslovnih pravil

2.4.2 Primeri uporabe podatkovne baze *dokument*

Nerelacijsko podatkovno bazo *document* lahko uporabimo za shranjevanje strojno generiranih podatkov (npr. sistemski dnevnik) ali točkovno oglaševanje, torej prilagajanju oglasov na podlagi zgodovine klikov [15]. Storitev oglaševanja mora imeti lastnost visoke dostopnosti (*angl. High Avaliablility*), saj izpad storitev pomeni potencialno izgubo denarja za ponudnike oglasov. Za zagotavljanje visoke dostopnosti mora podatkovna baza delovati simultano v več različnih podatkovnih centrih. Podatkovna baza *dokument* v svoji osnovi omogoča naslednje lastnosti:

- samodejno drobljenje podatkov (*angl. Sharding*),
- replikacijo sistema (*angl. Replication*) in
- upravljanje z zahtevki (*angl. Load Balancing*).

Predstavnika nerelacijske podatkovne baze *dokument* sta:

- MongoDB in
- CouchDB.

2.5 Ugotovitve

Pred končno izbiro nerelacijske podatkovne baze je smiselno narediti natančno analizo primerov uporabe ter pregledati produkte, ki ponujajo rešitve izbranih domenskih problemov. Za hitro naslavljanje vrednosti in uporabo enostavnih ključev, bi uporabili nerelacijsko podatkovno bazo *ključ-vrednosti.* V primeru analize kompleksnih povezav bi morali izbrati nerelacijsko podatkovno bazo *graf*, ki omogoča uporabo algoritmov nad grafi. V aplikaciji, ki bi v pretežni meri izvajali operacijo pisanja, bi morali premisliti o izbiri nerelacijske podatkovne baze *vrste-razširljivi zapisi.* Kot zadnja v te skupini je nerelacijska podatkovna baza *document,* ki omogoča shranjevanje samo-strukturiranih dokumentov.

3. Kratek pregled računalništva v oblaku in arhitekturnih gradnikov

Računalništvo v oblaku je model za omogočanje omrežnega dostopa do deljenega bazena nastavljivih računalniških virov (npr. omrežja, strežniki, podatkovne shrambe, aplikacij in storitev), do katerih je mogoče hitro dostopati [16]. Z možnostjo uporabe navideznih naprav na zahtevo lahko implementiramo različne topologije/arhitekture, ki jih uporabljamo za učinkovito in cenovno sprejemljivo možnost pri analizi velikih količin podatkov. Za načrtovanje predlagane arhitekture smo sledili principu trinivojske arhitekture ter za razvoj aplikacije uporabili model MVC (Model-View-Controller) [17].

3.1 Računalništvo v oblaku

Računalništvo v oblaku v osnovi ponuja koriščenje storitev preko interneta, s pristopom obračunavanja po uporabi (*angl. Pay-As-You-Go*). Glavne prednosti računalništva v oblaku so predvsem možnost uporabe virov, ki jih v danem trenutku potrebujemo, razširljivost ter visoka dostopnost.

Za končnega uporabnika je infrastruktura, torej izbira namestitvenega modela (*angl. Deployment Model*) oziroma storitvenega modela (*angl. Service Model*), nepomembna, vse dokler uporabnika oskrbujemo s kakovostnimi storitvami, ki so zanj pomembne. Ponudniki se morajo pri upravljanju infrastrukture držati sklenjenih dogovorov o ravni storitev (*angl. Service Level Agreement – SLA).* To je dogovor, ki se sklene med ponudnikom ter uporabnikom storitve. Ta vsebuje opis storitve IT in odgovornosti, ki jih imata tako ponudnik kot uporabnik [18].

Obstajajo tri velike skupine storitev, ki jih ponuja računalništvo v oblaku [19]:

- programska oprema kot storitev (*angl. Software as a Service, krat. SaaS*),
- platforma kot storitev (*angl. Platform as a Service, krat. PaaS*) in
- infrastruktura kot storitev (*angl. Infrastructure as a Service, krat. IaaS*).

Navedene storitve so lahko dostopne v različnih namestitvenih modelih računalništva v oblaku:

- zasebni oblak (*angl. Private Cloud*),
- javni oblak (*angl. Public Cloud*),
- skupnostni oblak (*angl. Community Cloud*) in
- hibridni oblak (angl. Hybrid Cloud).

Pravilno načrtovana arhitektura aplikacij omogoča hkratno uporabo različnih vrst računalništva v oblaku.

3.1.1 Namestitveni modeli

Namestitveni modeli se razlikujejo po lastniku, velikosti in dostopu do infrastrukture, na kateri je nameščena storitev v oblaku. Največjo marketinško pozornost so dobili ponudniki javnih storitev v oblaku, ki uporabnikom proti nizkim začetnim investicijskim stroškom omogočajo hitro razpoložljivost navideznih naprav in storitev. Organizacije, ki upravljajo z občutljivejšimi podatki, se običajno nagibajo k izgradnji svojega zasebnega oblaka ali uporabi t. i. hibridnega oblaka, ki združuje prednosti javnega ter zasebnega oblaka v celoto. Pri uporabi deljene infrastrukture med sorodnimi organizacijami govorimo o skupnostnem oblaku.

3.1.1.1 Zasebni oblak

V primeru upravljanja z občutljivimi podatki se veliko podjetji odloča za izgradnjo svojega zasebnega oblaka. Pri izbiri tega načina podjetja upravljanje zaupajo svojim sistemskim administratorjem ali pogodbenemu zunanjemu izvajalcu. Pri uporabi privatnega zasebnega oblaka imajo podjetja možnost koristiti vse prednosti storitev v oblaku (npr. razširljivost, visoka dostopnost).

3.1.1.2 Javni oblak

Javni oblak je namenjen vsem, ki običajno proti plačilu želijo imeti dostop do navideznih strežnikov preko interneta. Pri uporabi javnega oblaka so ponudniki storitev odgovorni za namestitev (*angl. Installation*), upravljanje (*angl. Management*), oskrbovanje (*angl. Provisioning*) in vzdrževanje (*angl. Maintenance*). *Med* vodilne ponudnike javnih storitev v oblaku štejemo Amazon Web Service, Microsoft Azure, Google Apps, SalesForce.com, OpenStack, OpenShift.

3.1.1.3 Skupnostni oblak

Uporaba skupnostnega oblaka je značilna za organizacije, ki imajo podobne zahteve. V tem primeru si organizacije delijo vire skupnostnega oblaka med seboj. Prednost pri uporabi te vrste namestitvenega modela je predvsem hierarhija zaupanja. Podjetja običajno bolj zaupajo v infrastrukturo, deljeno med svojimi partnerskimi podjetji, kot tisto, ki je namenjena poljubnemu uporabniku.

3.1.1.4 Hibridni oblak

Hibridni oblak združuje prednosti javnega ter zasebnega oblaka v celoto. Pri tem namestitvenem modelu lahko zahteve po večjih zmogljivosti razširimo na javni oblak, medtem ko občutljive podatke s pomočjo privatnega oblaka hranimo v svojem podatkovnem centru. Omenjeni način je v podjetjih pritegnil veliko pozornosti, s tem pa ponudnikom javnih oblakov dal dodatno motivacijo za razvoj posebnih modulov, ki poenostavijo združitev med različnimi namestitvenimi načini (npr. AWS Direct Connect, Windows Azure ExpressRoute).

3.1.2 Storitveni model

Storitveni modeli opisujejo tip storitve, ki jih ponudniki storitev ponujajo in so dosegljive preko platforme v oblaku [19]. Storitvene modele delimo v tri prevladujoče skupine:

- programska oprema kot storitev
- platforma kot storitev ter
- infrastruktura kot storitev.

Infrastruktura kot storitev predstavlja osnovo računalništva v oblaku. V ta nivo sodi strojna oprema ter z njo storitev oskrbovanja. Drugi nivo – platforma kot storitev – omogoča enostavni najem programske opreme, ki je potrebna za učinkovit razvoj tretjega nivoja – programska oprema kot storitev.

3.1.2.1 Programska oprema kot storitev

To vrsto storitve si v osnovi lahko predstavljamo kot že razvite uporabniške storitve, do katerih dostopamo preko interneta. Kot primer lahko navedemo različne storitve za e-pošto, aplikacije za upravljanje s plačili, spletni urejevalnik besedil. Pri uporabi te storitve uporabnik uporablja funkcionalnosti, ki jih ponuja izbrana storitev. Uporabniki s pomočjo spletnega vmesnika dostopajo do aplikacij, pri čemer so nekatere brezplačne (npr. Gmail, Facebook), medtem ko druge za dostop zahtevajo plačilo (npr. Microsoft Office 365). Ključne prednosti uporabe programja kot storitve, so predvsem: enostavnost uporabe, hitra integracija v podjetja, razširljivost in zanesljivost sistema ter zmanjšanje potreb po zmogljivi strojni opremi osebnih računalnikov (npr. Adobe Creative Cloud). Kot zanimiv člen v te skupini se pojavi tudi varnost kot storitev SaaSS (*angl Software as a Secure Service*) [19], ki je pripeta obstoječim SaaS aplikacijam. Ta omogoča dodatne možnosti varovanja pri uporabi programja kot storitve, torej predstavlja vmesni nivo med uporabnikom in ponudnikom.

3.1.2.2 Platforma kot storitev

Razvoj programja je kompleksen proces, ki za delovanje zahteva strojno opremo, podatkovne baze ter aplikacijske strežnike. Ti sistemi morajo biti nameščeni in vzdrževani s strani sistemskih administratorjev, ki skrbijo za varnostne kopije. Ideja računalniškega okolja kot storitve je omogočiti lažji, cenovno ugodnejši in hitrejši razvoj aplikacij. Razvijalci se morajo pri uporabi teh storitev osredotočiti zgolj na programsko kodo in uporabljati programske vmesnike API, ki jih ponujajo ponudniki storitev. Od računalniškega okolja kot storitve lahko pričakujemo storitve, kot so varnost, razširljivost, možnost shranjevanja podatkov, integracija podatkovnih baz, sistem za dokumentiranje sprememb. V to skupino zagotovo lahko umestimo Amazon Storage Service, vrste SQS, SNS, OpenShift.

3.1.2.3 Infrastruktura kot storitev

Infrastruktura kot storitev je v večji meri namenjena ponudbi navideznih naprav kot storitve. Pri tem lahko uporabniki po lastni volji izberejo različne skupine, ki se razlikujejo predvsem po zmogljivostnih lastnostih. Po izbiri skupine se samodejno ustvari navidezna naprava, do katere lahko uporabniki dostopajo z oddaljenim dostopom SSH ali Telnet. Prednost infrastrukture kot storitve so nižji stroški, pri čemer uporabnikom ni potrebno skrbeti za nadgradnjo strojne opreme. Poleg nižjih stroškov ne smemo pozabiti na možnost razširitve ter visoke dostopnosti na porazdeljenem sistemu naprav [20]. Ta vrsta storitve predstavlja najnižji sklop računalništva v oblaku, na katerem so grajene druge vrste omenjenih storitev v oblaku.

V storitev IaaS sodijo Amazon Elastic Cloud Computing (EC2), Windows Azure navidezne naprave, OpenStack ...

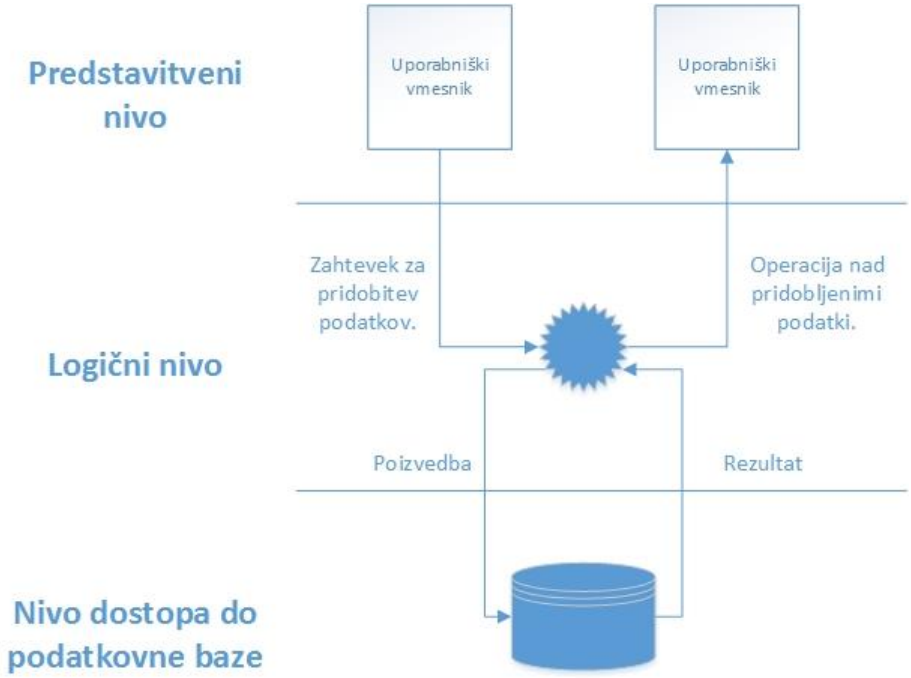
3.2 Večnivojska arhitektura

Za dosego aplikacijske šibke sklopljenosti (*angl. Loose Coupling*) med posameznimi nivoji aplikacije in s tem zagotavljanje robustnosti in fleksibilnosti smo za načrtovanje predlagane arhitekture izbrali pristop, ki ga zagovarja trinivojska arhitektura. Ta arhitektura je sestavljena iz treh glavnih delov:

- predstavitveni nivo (*angl. Presentation Tier*),
- logični nivo (*angl. Logic Tier*) in
- nivo dostopa do podatkovne baze (*angl. Data Tier*).

Komunikacija med navedenimi deli poteka samo neposredno preko soseda. To z drugimi besedami pomeni, da uporabniki preko predstavitvenega nivoja ne morejo neposredno dostopati

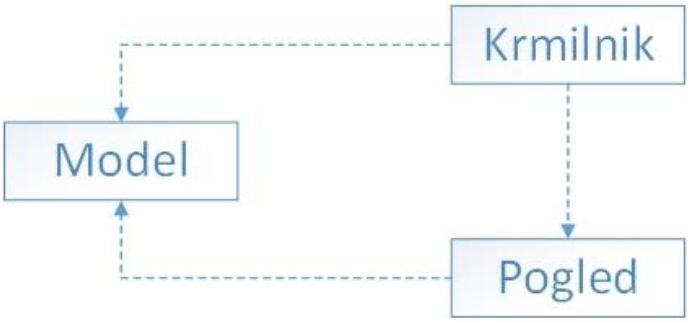
do podatkov, shranjenih v podatkovni bazi, vendar morajo zahtevki potovati preko logičnega nivoja, ki implementira poslovno logiko aplikacije [Slika 9].



Slika 9: Predstavitev trinivojske arhitekture.

3.3 Model MVC (Model-View-Controller)

Za namen razvoja spletnih aplikacij lahko velikokrat zasledimo pojem MVC Model-Pogled-Krmilnik (*angl. Model-View-Controller*). Ta v logičnem smislu sledi principom trinivojske arhitekture, vendar z glavno razliko, da je model MVC namenjen opisovanju vzorcev razvoja spletnih aplikacij (*angl. Programming Design Pattern*) [21]. Glavna prednost tega pristopa je predvsem šibka sklopljenost med deli aplikacije in s tem možnost lažjega razvoja spletnih aplikacij za podporo različnim platformam (npr. razvoj aplikacije za različne spletne brskalnike, tablične računalnike …). Pri uporabi modela MVC je potrebno vedeti, da sta modula *pogled* in *krmilnik* odvisna od modula *model,* ta pa ni odvisen od njiju [Slika 10]. To predstavlja veliko prednost, saj lahko modul *model* testiramo ali zamenjamo neodvisno od ostalih dveh. Drugi korak pri zagotavljanju šibke sklopljenosti je delitev odvisnosti med moduloma *krmilni* ter *pogled.* Pri razvoju spletnih aplikacij je ta delitev vidna z uporabo razvoja aplikacije na strani strežnika (*angl. Server Side*) in strani uporabnika (*angl. Client Side*), pri čemer prvi predstavlja modul *krmilnik* in drugi modul *pogled.*



Slika 10: Model MVC.

3.3.1 Model

Modul *model* se uporablja za delo s podatkovnim nivojem aplikacije. Po svoji značilnosti je pomemben, saj upravlja z vsemi podatki, ki jih uporablja aplikacija. Dobro načrtovana aplikacija MVC ne bo nikoli neposredno dostopala do podatkovne baze, vendar bo vse podatke hranila v objektih, katerih shema je definirana s pomočjo razredov. Vsi podatki, ki se uporabljajo v aplikaciji, naj bi bili dostopni preko objektov, saj poslovni nivo aplikacije s temi objekti rešuje domenski problem.

3.3.2 Pogled

Modul *pogled* je predstavljen kot uporabniški vmesnik, ki ga lahko razvijemo z različnimi pristopi (npr. HTML, JavaScript, AngularJS, CSS, JSP in ASPX). Če spletno aplikacijo gradimo z mislijo o šibki sklopljenosti delov modela MVC, potem lahko modul pogled prilagodimo za podporo različnim napravam (npr. računalnik, tablični računalnik, mobilni telefoni). V tem primeru nam ni potrebno spreminjati programske logike implementirane v modulu *krmilnik* ter dostopa do podatkovne baze implementiranega v modulu *model.*

3.3.3 Krmilnik

Modul *krmilnik* je vmesni nivo med moduloma *pogled* in *model*. Predstavljen je kot glavni del sistema, katerega naloga je upravljanje z zahtevki in odgovori ter nastavitev povezav do podatkovne baze.

4. Integracija povpraševanj med različnimi nerelacijskimi podatkovnimi bazami

V tem poglavju bomo združili problem izvajanja povpraševanj nad različnimi tipi nerelacijskih podatkovnih baz in integracije rezultatov povpraševanj v enoten odgovor. Ta problem je še posebej poudarjen pri uporabi različnih tipov nerelacijskih podatkovnih baz, saj se te baze razlikujejo tako v načinu shranjevanja podatkov, kakor tudi v načinu izvajanja povpraševanj. Glavna motivacija integracije je z enotnim poizvedovalnim jezikom sočasno poizvedovati po različnih vrstah podatkovnih virov.

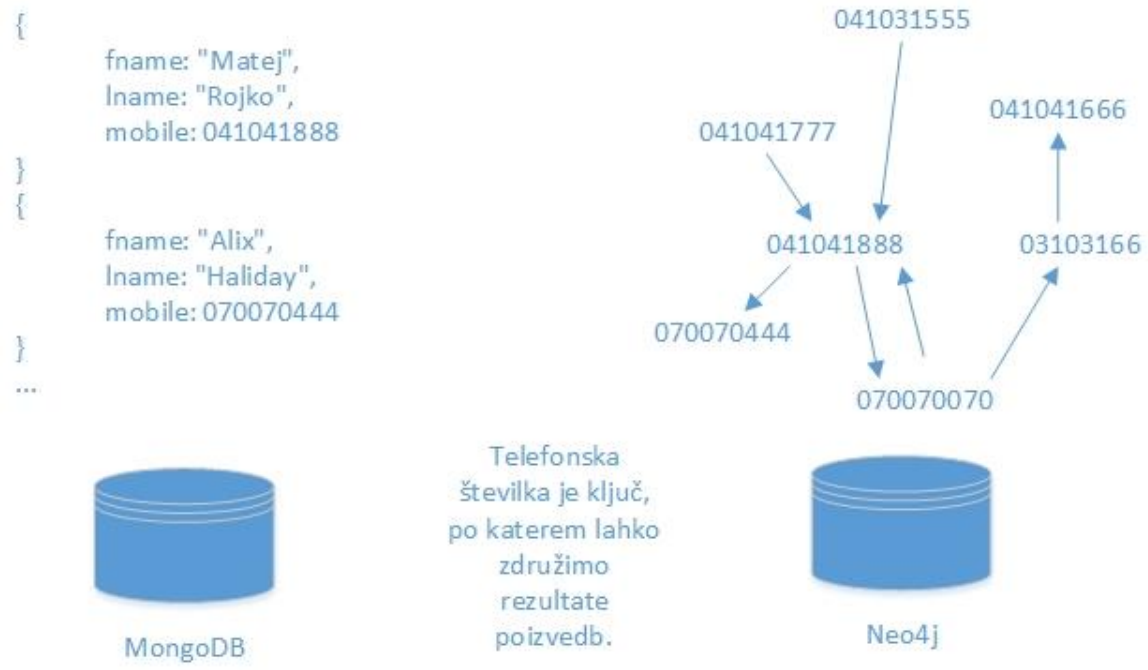
V tem poglavju bomo pregledali možnosti integracije povpraševanj nad nerelacijskimi podatkovnimi bazami s ciljem, da bi dosegli rezultat, kjer bomo s skupnim poizvedovalnim jezikom zmožni pridobiti podatke iz različnih vrst podatkovnih baz.

Za namen integracije poizvedb nad različnimi vrstami podatkovnih baz bomo analizirali naslednje načine [22]:

- integracija z uporabo neposrednega dostopa do podatkovnih baz,
- integracija z uporabo spletnih storitev,
- integracija z uporabo ETL ter
- integracija s semantičnim spletom in ontologijo.

Nato bomo predstavili način integracije povpraševanj nad različnimi tipi podatkovnih baz z uporabo poizvedovalnega jezika SPARQL. V ta namen smo razvili program, ki omogoča samodejno pretvorbo podatkov v zapis RDF/XML ter oglaševanje teh preko končne točke SPARQL (*angl. SPARQL endpoints*).

Pri implementaciji integracije smo uporabili primer oseb in telefonskih številk. Nerelacijska podatkovna baza MongoDB hrani podatke o osebah (ime, priimek, telefonska številka), medtem ko nerelacijska podatkovna baza Neo4j hrani podatke s telefonskimi številkami pošiljateljev in prejemnikov sporočil [Slika 11]. Navedeni primer se smiselno prilagaja izbranima vrstama nerelacijskih podatkovnih baz *dokument* in *graf*.



Slika 11: Prikaz domene oseb in telefonskih številk.

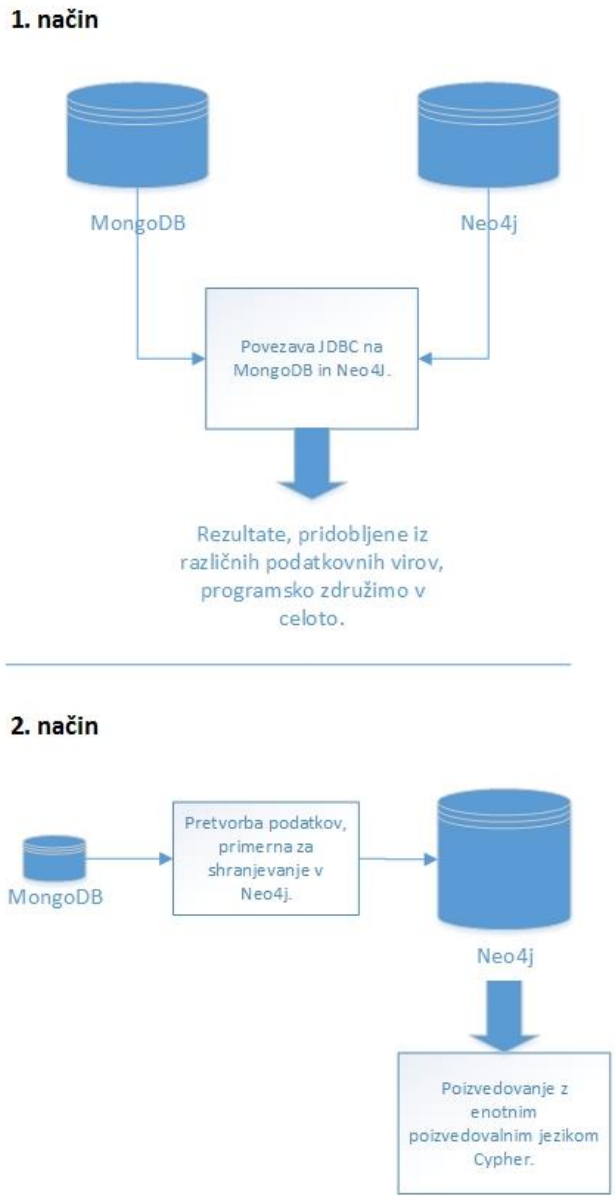
4.1 Integracija z uporabo neposrednega dostopa do podatkovnih baz

Način z neposredno povezavo na podatkovno bazo je najenostavnejši način, ki ga poznamo pri skupnem poizvedovanju ali integraciji podatkovnih baz. Vse nerelacijske podatkovne baze [2. Pregled nerelacijskih podatkovnih baz], iz različnih programskih jezikov (npr. Java, Python …), omogočajo neposredno povezavo na izbrano podatkovno bazo. Za poizvedovanje podatkov po različnih podatkovnih virih bi morali uporabiti nizkonivojski vmesnik, pri čemer bi sprva poizvedovali po prvem in nato po drugem podatkovnem viru. Pri tem načinu bi pozneje programsko združili rezultate in jih vrnili uporabniku (prvi način) [Slika 12].

Za nekatere primere bi lahko poleg združevanja rezultatov sprva pretvorili podatke domene prve podatkovne baze v zapis podatkov druge podatkovne baze in jih shranili v drugo podatkovno bazo. V našem primeru bi to pomenilo, da bi podatke o osebah, shranjene v nerelacijski podatkovni bazi MongoDB, pretvorili v vozlišča nerelacijske podatkovne baze Neo4j. Nad temi podatki bi skupaj z obstoječimi podatki lahko izvajali poizvedbe z enotnim poizvedovalnim jezikom nerelacijske podatkovne baze Neo4j (drugi način) [Slika 12]. Kot smo omenili [2. Pregled nerelacijskih podatkovnih baz], vsi podatki niso primerni za vsako vrsto nerelacijske podatkovne baze, zato moramo biti pri tem postopku previdni. Pri uporabi drugega načina bi

povzročili, da bi ena podatkovna baza vsebovala podatke drugih podatkovnih baz, pri čemer bi morali zagotoviti tudi sinhronizacijo med pripetimi podatkovnimi bazami.

Za primer lahko predpostavimo, da bi podatke o operaterjih pridobivali s povezavo na dodatni podatkovni vir. V tem primeru bi morali vse podatke ponovno pretvoriti in jih shraniti v svojo nerelacijsko podatkovno bazo Neo4j.



Slika 12: Prikaz težav pri uporabi integracije z neposrednim dostopom do podatkovnih baz.

4.2 Integracija z uporabo spletnih storitev

Spletne storitve za svoje delovanje uporabljajo izmenjavo sporočil in protokol HTTP. Konzorcij W3C je standardiziral spletne storitve tako, da le-te omogočajo komunikacijo med računalniki (*angl. Machine-To-Machine*). Običajno se sporočila med sistemi prenašajo v zapisu XML ali JSON. Spletne storitve so za razliko od neposrednega dostopa do podatkovnih baz bolj primerne, saj so aplikacije, ki se naslajajo na njih šibko sklopljene. Implementacija aplikacij z uporabo spletnih storitev omogoča gradnjo kompleksnih sistemov.

Če bi naredili ločnico med konceptualnim in tehničnim nivojem spletnih storitev, potem bi ugotovili, da konceptualno spletne storitve predstavljajo način standardiziranega oglaševanja podatkov preko protokola HTTP. Tehnično gledano pa so lahko spletne storitve implementirane na vrsto različnih načinov. Med najpopularnejše sodita SOAP (*Service-Oriented Architecture Protocol*) in RESTful (*Representational State Transfer*). Poleg omenjenih poznamo tudi druge arhitekturne stile distribuiranega računalništva CORBA, DCOM …

Implementacija spletnih storitev s tehnologijo SOAP je namenjena predvsem poznejši gradnji kompleksnih vendar robustnih sistemov. Prednost, ki jo prinaša definiranje spletnih storitev s tehnologijo SOAP, je standardizirani opis razpoložljivih spletnih storitev ali končnih točk, imenovan WSDL (*Web Service Description Language*). Poleg standardiziranega definiranja vsakega izmed vmesnikov moramo dokumente zapisa XML, ki so namenjeni izmenjavi oviti v telo SOAP [23].

Drugi način za implementacijo in uporabo spletnih storitev se imenuje RESTFul. Ta je mnogo enostavnejši, saj za delovanje ne zahteva razvoja vmesnika WSDL. Hkrati lahko slednje predstavlja tudi slabost, saj je vmesnik potrebno opisati drugače, ne nujno na računalniku razumljivi način, kar onemogoča samodejno generiranje programske kode odjemalcev. Za opis vmesnika lahko uporabimo WADL (*Web Application Description Language*) [24].

Nerelacijski podatkovni bazi MongoDB in Neo4j, ki smo jih izbrali za implementacijo arhitekture, za dostop do podatkov omogočata uporabo spletnih storitev RESTful. MongoDB za uporabo spletnih storitev preko zahtevka GET prejme parametre, ki jih uporabi pri izvajanju poizvedb. Neo4j za razliko od MongoDB zahteva ovijanje poizvedovalnega jezika Cypher. V obeh primerih bosta nerelacijski podatkovni bazi izvedli poizvedbo in vrnili rezultate v zapisu JSON.

Uporaba spletnih storitev ne zahteva nameščanja dodatnih programskih vtičnikov API. Kljub vsemu bi morali v primeru uporabe spletnih storitev izvesti delne poizvedbe nad vsakim podatkovnim virom ter ob pridobitvi vseh podatkov programsko združiti rezultate v celoto.

4.3 Integracija z uporabo ETL

Na nek način smo idejo sistema ETL (*Extract, transform, load)* že omenili v poglavju o integraciji [4.1 Integracija z uporabo neposrednega dostopa do podatkovnih baz] z uporabo neposrednega dostopa do podatkovnih baz preko programskih vtičnikov, kjer smo opisali, da je možno podatke iz ene podatkovne baze pretvoriti v obliko podatkov druge podatkovne baze. Proces ETL definira tri osnovne korake [25]:

- izvoz podatkov iz podatkovnih virov,
- pretvorba podatkov in
- uvoz podatkov.

4.3.1 Izvoz podatkov

Namen izvoza podatkov je predvsem izbira tistih podatkov, ki bodo vključeni v poizvedbe. Pri procesu izvoza moramo paziti, kaj se zgodi ob poznejši spremembi podatkov v izvirni podatkovni bazi. Za zagotavljanje sinhroniziranega načina med podatkovnimi viri moramo proces prilagoditi tako, da bo odziven na spremembe v izvirni podatkovni bazi.

4.3.2 Pretvorba podatkov

Pretvorba podatkov se pri relacijskih podatkovnih bazah nanaša predvsem na spremembo v shemi, medtem ko se pri nerelacijskih podatkovnih bazah nanaša na različne zapise in oblike dokumentov.

4.3.3 Uvoz podatkov

V nerelacijskih podatkovnih bazah s pomočjo različnih modulov (npr. MongoDB – mongoimport, Neo4j – LOAD CSV, Cassandra – COPY) uvozimo podatke. Pri uvozu podatkov lahko nove podatke prepišemo z obstoječimi ali pa jih dodamo v ciljno podatkovno bazo. Za uspešen uvoz podatkov moramo zagotoviti, da so podatki v zahtevani obliki ter zapisu ciljne podatkovne baze [26].

4.4 Integracija s semantičnim spletom in ontologijo

V poglavju [2. Pregled nerelacijskih podatkovnih baz] smo omenili, da je vsaka nerelacijska podatkovna baza optimizirana za učinkovito shranjevanje in poizvedovanje vnaprej določene vrste podatkov. Podatke bi sicer lahko integrirali v eno hibridno nerelacijsko podatkovno bazo, ki podpira različne vrste (npr. ArangoDB), vendar bi s tem izgubili možnost dinamičnega pripenjanja oziroma poizvedovanja nad različnimi podatkovnimi viri v realnem času, torej bi

morali podatke uvoziti v izbrano nerelacijsko podatkovno bazo. Za implementacijo tega pristopa bi moral algoritem samodejno pridobivati podatke iz različnih javno dostopnih podatkovnih virov in jih shranjevati v izbrano nerelacijsko podatkovno bazo, pri čemer bi morali zagotoviti odzivnost na spremembe v izvirnih podatkovnih virih. V nasprotju nam integracija s semantičnim spletom omogoča uporabo skupnega poizvedovalnega jezika SPARQL, s katerim podatke ohranimo v izvirnih podatkovnih bazah.

Svetovni splet je infrastruktura, ki združuje skoraj 3 milijarde uporabnikov [27], med katerimi večina shranjuje podatke. Veliko dodano vrednost predstavlja povezovanje in združevanje podatkov iz različnih podatkovnih virov. Glavna prednost semantičnega spleta je predvsem pridobivanje podatkov iz drugih javno dostopnih podatkovnih baz in združevanje teh s podatki, ki jih hranimo pri sebi.

4.4.1 Predstavitev semantičnega spleta in ontologije

Semantični splet je razširitev obstoječega svetovnega spleta, kjer so podatki z vključitvijo semantike razumljivi računalniškim sistemom. Semantični splet se v osnovi naslaja na ogrodje ontologije, ki predstavlja skupek pravil za definiranje shem.

Z uporabo semantičnega spleta in ontologije je mogoče implementirati integracijo nerelacijskih podatkovnih baz [28].

4.4.1.1 Semantični splet

Semantični splet lahko opišemo z definicijo [29]: Skupek standardov in dobrih praks za deljenje podatkov in pomena teh podatkov preko spleta na način, ki je razumljiv tudi računalniškim sistemom.

Če razdelimo to definicijo na tri dele, potem dobimo naslednjo predstavo o semantičnem spletu:

1. **Del definicije – *Skupek standardov***

Semantični splet je zgrajen na osnovi pravil, ki jih določa organizacija W3C. V svoji osnovi pozna štiri osnovne standarde:

- podatkovni model RDF,
- poizvedovalni jezik SPARQL,
- shema RDF in
- standard OWL za definiranje ontologije.

2. **Del definicije – Skupek *dobrih praks za deljenje podatkov preko spleta na način, ki je razumljiv tudi računalniškim sistemom***

Semantični splet je bil načrtovan z namenom, da je razumljiv tako ljudem kot računalniškim sistemom. Spletne storitve, ki omogočajo iz različnih spletnih aplikacij pridobiti informacije o npr. najcenejših letalskih vozovnicah, so dandanes nekaj običajnega. Nekoč so tovrstne spletne storitve pridobivale te podatke na način, kjer je algoritem obiskal spletno stran prevoznika in s pomočjo pretvorbe dokumentov HTML izluščil podatke o vozovnicah. Ob vsaki spremembi spletnih strani, so morali razvijalci prilagoditi algoritem za pridobivanje podatkov. Z uporabo semantičnega spleta se tem težavam lahko hitro izognemo. Arhitektura Linked Data [30] je tista, ki predstavlja dobre prakse za deljenje podatkov med sistemi.

3. **Del definicije – Skupek *pomena podatkov***

Za deljenje pomena podatkov moramo uporabljati naslove URI, ki enolično določajo vire. Pomembno lastnost za širjenje pomena podatkov predstavlja predvsem ontologija, s katero definiramo pomen objektov, ki so porazdeljeni na različnih podatkovnih virih.

Semantični splet je torej porazdeljen in heterogen sistem, ki dviguje svetovni splet na višjo raven. V svoji osnovi predstavlja medij za kolaboracijo med sistemi in omogočanje razumevanja podatkov s strani računalniških sistemov. Tako svetovni splet kot semantični splet sta namenjena povezovanju virov, ki so predstavljeni z naslovi URI. Glavna razlika med svetovnim in semantičnim spletom je v tem, da svetovni splet uporablja protokol HTTP za namen predstavitve vsebine spletnih strani, medtem ko semantični splet poizkuša vsebino predstaviti na način, ki je razumljiv računalniškim sistemom.

4.4.1.2 Ontologija

Ontologija je v osnovi uporabljena za namen medsebojnega povezovanja (*angl. Interoperability*) in enotnega razumevanja podatkov med različnimi sistemi. Ideja ontologije je predvsem definiranje skupnega besednjaka in uporaba preslikovalnega jezika, ki bo uspel preslikati podatke, porazdeljene v različnih domenah, v skupno logično domeno, razumljivo vsem sistemom, vpetim v proces integracije. Ontologija je sestavljena iz treh ključnih delov [28]:

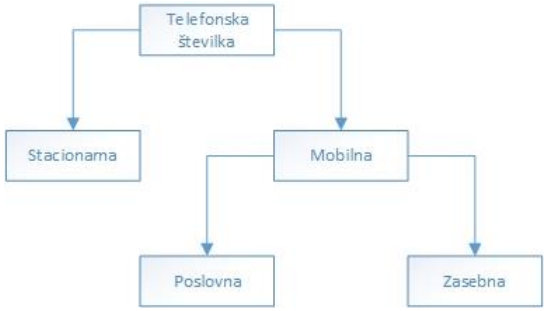
- instanc,
- razredov ter
- relacij.

Instance

Instance vključujejo objekte, kot so osebe, telefonske številke, molekule … Cilj ontologije je dodeljevanje pomena vsem tem objektom.

Razredi

Razredi so abstrakcija instanc za opis domene in so predstavljeni kot kolekcija objektov, povezanih v hierarhični graf. Za definiranje domene poznamo glavi razred in podrazrede [Slika 13].



Slika 13: Hierarhija razredov, definiranih v ontologiji.

Relacije

Relacije imajo največji pomen pri opisu domene. Poznamo naslednjo vrsto relacij:

- relacije med razredi,
- relacije med instancami,
- relacije med instancami in razredi,
- relacije med instanco in kolekcijo ter
- relacije med kolekcijami.

S pomočjo teh vrst relacij in opisa pomena je možno zelo natančno opisati domeno in jo predstaviti kot povezan graf med objekti in razredi.

Kadar bi želeli integrirati podatkovne baze z javno dostopnimi, bi morali veliko energije vložiti za pravilno definiranje ontologije [31]. Na temo o integraciji ontologij (torej integracije opisa domenskega problema) obstaja vrsto znanstvenih člankov. Za implementacijo ontologije bi morali definirati lokalne ontologije nad posameznimi nerelacijskimi podatkovnimi bazami. Nad slojem lokalnih ontologij bi morali za namen integracije ontologije definirati globalno ontologijo, ki bi združevala pomen različnih instanc [32] . V magistrski nalogi smo predpostavili, da želimo integrirati podatkovne vire, ki hranijo podatke o podobni problemski domeni, kjer uporabnik ve, katere podatke želi združiti med seboj.

4.4.2 Integracija s pomočjo semantičnega spleta

Za uporabo semantičnega spleta se uporablja skupni poizvedovalni jezik SPARQL. Poizvedovalni jezik SPARQL deluje nad dokumenti zapisa RDF in lahko hkrati izvaja poizvedbe nad različnimi podatkovnimi viri. To omogoča SPARQL SERVICE, ki se med izvajanjem poizvedbe poveže na t. i. končne točke. Ko poizvedovalni jezik SPARQL iz različnih podatkovnih virov pridobi potrebne podatke, jih na koncu samodejno združi v celoto.

Pri integraciji nerelacijskih podatkovnih baz poznamo dva pristopa, ki se med seboj ločita predvsem v načinu predstavitev podatkov poizvedovalnemu jeziku SPARQL:

1. možnost preslikave poizvedovalnega jezika SPARQL v poizvedovalni jezik za izbrano nerelacijsko podatkovno bazo [33] in
2. preslikava dokumentov zapisa JSON v dokumente zapisa RDF/XML in oglaševanje teh na končni točki SPARQL [34].

Glavna razlika med navedenima načinoma je predvsem v vprašanju ali želimo poizvedbe prestaviti bližje k podatkom ali pa podatke bližje k poizvedbam. V naslednjih dveh poglavjih bomo natančno opisali oba načina. Za implementacijo *proof of concept* smo izbrali drugi način, pri katerem smo s pomočjo transformacije XSLT pretvorili podatke zapisa JSON v podatke zapisa RDF/XML in jih shranili na končno točko, ki je dostopna poizvedovalnemu jeziku SPARQL.

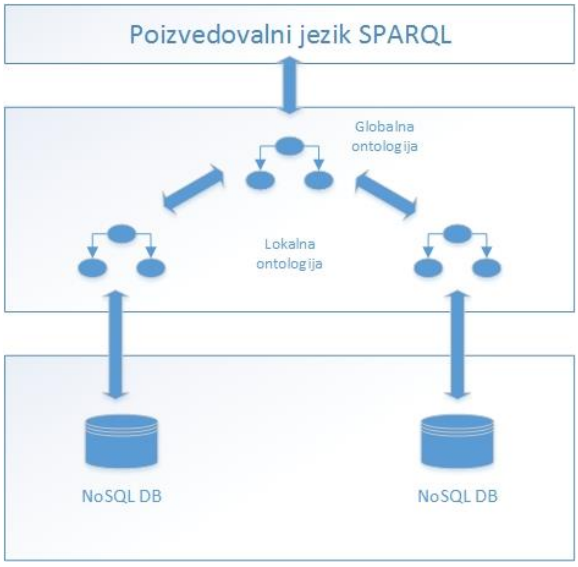
4.4.3 Preslikava poizvedovalnega jezika SPARQL v poizvedovalni jezik za izbrano nerelacijsko podatkovno bazo

Nerelacijske podatkovne baze za svoje delovanje ne potrebujejo vnaprej definiranih shem, kar omogoča večjo fleksibilnost pri shranjevanju podatkov.

V članku, ki govori o integraciji podatkovnih baz [33], so predvsem z vrsto nivojev in pretvorb uspeli preslikati skupni poizvedovalni jezik SPARQL v poizvedovalni jezik nerelacijske podatkovne baze MongoDB. Za uspešno preslikavo poizvedovalnega jezika SPARQL v poizvedovalni jezik MongoDB moramo okvirno slediti naslednjemu postopku:

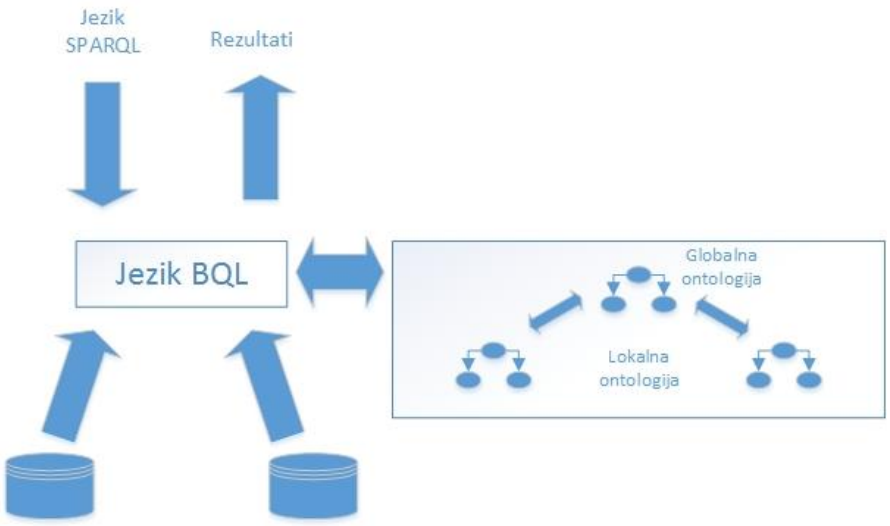
- za vsak podatkovni vir moramo definirati t. i. lokalno ontologijo, ki je namenjena opisu domene podatkov,
- nad tem nivojem je potrebno definirati globalno ontologijo, ki iz različnih lokalnih ontologij preslika pomen podatkov. Globalna ontologija predstavlja referenco za naslavljanje objektov v podatkovnih bazah ter

- skupni poizvedovalni jezik SPARQL pretvorimo v poizvedovalni jezik izbrane nerelacijske podatkovne baze. Za dosego tega cilja uporabimo jezik BQL (Bridge Query Language) [35]. Nerelacijske podatkovne baze v osnovi ne podpirajo poizvedovalnega jezika, kot ga poznamo pri relacijskih podatkovnih bazah (SQL). Te za razliko uporabljajo proceduralni način, ki ga poznamo v programskem jeziku Java (torej uporaba razredov in objektov) [Slika 14].



Slika 14: Osnovna arhitektura, ki prikazuje omenjen pristop.

Za izvajanje poizvedb SPARQL je potrebno te preslikati v poizvedovalni jezik, ki ga omogoča izbrana nerelacijska podatkovna baza. Pri naslavljanju objektov lahko uporabljamo globalno ontologijo, ki zagotavlja pravilno naslavljanje objektov, opisanih v lokalnih ontologijah. Torej če poenostavimo ta pristop, potem uporabljamo slovar (t.j. globalno ontologijo), s katerim si pomagamo, da lahko v jeziku BQL implementiramo funkcije, ki naslavljajo objekte, po katerih želimo iskati. Jezik BQL je primeren za namen preslikave iz abstraktnega poimenovanja objektov v implementacijo proceduralnega jezika poizvedb [Slika 15]. Končne rezultate, ki jih pridobimo iz različnih lokalnih podatkovnih virov, moramo programsko združiti v celoto in vrniti kot skupni rezultat.

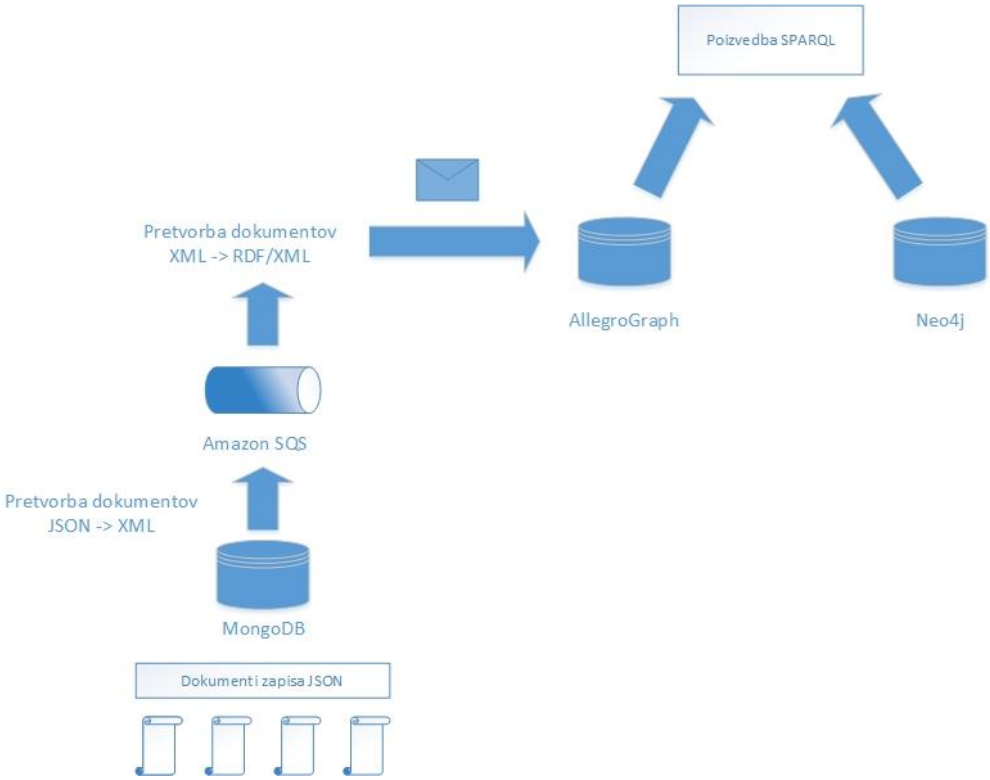


Slika 15: Postopek preslikave poizvedovalnega jezika BQL in vračanje rezultatov.

4.4.4 Preslikava dokumentov zapisa JSON v dokumente zapisa RDF/XML in oglaševanje teh preko končne točke SPARQL

Za namen integracije izbranih nerelacijskih podatkovnih baz smo želeli podpreti poizvedovalni jezik SPARQL, ki lahko sočasno poizveduje nad porazdeljenim sistemom nerelacijskih podatkovnih baz. Če smo v prejšnjem poglavju govorili predvsem o preslikavi jezika SPARQL v proceduralni jezik, ki ga ima vsaka nerelacijska podatkovna baza implementiranega na svoj način, bomo v tem poglavju govorili o preslikavi dokumentov, shranjenih na različnih nerelacijskih podatkovnih bazah na način, da jih lahko ponudimo jeziku SPARQL kot parameter za poizvedovanje po njih.

Glavna ideja je torej prestavitev podatkov bližje poizvedbam in ne poizvedb bližje k podatkom. Za ta namen smo v praktičnem delu implementirali algoritem, ki skrbi za pravilno pretvorbo dokumentov zapisa JSON v dokumente zapisa RDF/XML ter nad nerelacijsko podatkovno bazo MongoDB namestili dodatno podatkovno bazo, imenovano AllegroGraph, ki je namenjena predvsem oglaševanju končne točke SPARQL. Razvita programska oprema v celoti podpira proces pretvorbe dokumentov za namen uspešnega izvajanja poizvedb SPARQL. Delovni tok, ki smo ga uporabili pri implementaciji, je prikazan v nadaljevanju [Slika 16].



Slika 16: Proces pretvorbe podatkov za namen izvajanja sočasnih razširjenih poizvedb SPARQL.

Proces pretvorbe podatkov se prične z branjem dokumentov zapisa JSON v nerelacijski podatkovni bazi MongoDB. Ti dokumenti so z uporabo različnih metod in vrste Amazon AWS pretvorjeni v zapis RDF/XML in poslani na končno točko AllegroGraph.

Dokumenti, shranjeni na nerelacijski podatkovni bazi Neo4j, so predstavljeni kot točka-točka-povezava, zato je te dokumente mnogo lažje pretvoriti v dokumente zapisa RDF/XML ter jih z namestitvijo posebnega modula oglaševati poizvedovalnemu jeziku SPARQL.

V naslednjih poglavjih bomo predstavili podrobnejši opis vsakega izmed korakov.

4.4.4.1 Predstavitev podatkov shranjenih na nerelacijski podatkovni bazi MongoDB

Nerelacijska podatkovna baza MongoDB shranjuje podatke v dokumente zapisa JSON/BSON. Zapis JSON/BSON omogoča poljubno dodajanje novih polj. Prav zaradi izjemne fleksibilnosti dokumentov JSON se je potrebno zavedati, da je težko zagotoviti generičen pristop, ki bi bil zmožen pretvoriti vse možne oblike dokumentov zapisa JSON v dokument, ki bi bil razumljiv poizvedovalnemu jeziku SPARQL. Zaradi omenjene težave smo v naši implementaciji nekoliko

omejili fleksibilnost in uporabnikom omogočili izbiro prvonivojskih polj, katerih vrednosti bodo pozneje vključene v poizvedbo SPARQL. Ob navedbi integracijskega ključa (npr. telefonska številka) poizvedovalni jezik SPARQL samodejno združi podatke in vrne skupni rezultat. Pri tem načinu integracije, torej pretvorbi dokumentov v zapis RDF/XML, nam ni potrebno implementirati združevalnega algoritma, kot smo to omenili pri integraciji z uporabo jezika BQL. Izrazita prednost pristopa s pretvorbo podatkov in uporabo poizvedovalnega jezika SPARQL je integracija naših podatkov z javno dostopnimi podatkovnimi bazami, ki s pomočjo semantičnega spleta oglašujejo svoje podatke.

V primeru izvajanja poizvedb nad novo nerelacijsko podatkovno bazo MongoDB bi morali podatke po že opisanem postopku pretvoriti v dokumente zapisa RDF/XML in jih oglaševati preko končne točke. Pri tem ne bi potrebovali vnovičnega definiranja lokalne ontologije, kot bi morali to storiti pri pretvorbi poizvedovalnega jezika SPARQL v proceduralni jezik izbrane podatkovne baze. Za poizvedovanje nad dodatno podatkovno bazo lahko v poizvedovalnem jeziku SPARQL uporabimo storitev, ki na končno točko posreduje poizvedbo in po pridobitvi ustreznih podatkov le-te samodejno združi v celoto.

4.4.4.2 Pretvornik dokumentov

Ta del je predstavljen v dveh korakih; pretvorbe dokumentov iz zapisa JSON v zapis XML ter pozneje v zapis RDF/XML. Pretvornik dokumentov iz zapisa JSON v zapis XML je preprosta funkcija, implementirana v programskem jeziku JavaScript, ki zagotavlja, da se vsi atributi preslikajo v dokument zapisa XML.

Dokumente zapisa XML smo s pomočjo lastne implementacije transformacije XSLT pozneje pretvorili v dokumente zapisa RDF/XML, pri čemer smo upoštevali priporočila organizacije W3C [36].

4.4.4.3 Uporaba sporočilnega sistema Amazon

Za zagotavljanje zanesljivosti sistema smo dokumente zapisa XML sprva shranili v vrsto Amazon SQS (*Simple Queue Service*) in jih pozneje z uporabo razčlenjevalnika Saxon pretvorili v zapis RDF/XML ter shranili v podatkovno bazo AllegroGraph. Poleg spletne storitve Amazon bi lahko v programskem jeziku Java uporabili programski vmesnik JMS, ki podpira uporabo sporočilnih sistemov.

V osnovi poznamo dva načina delovanja sporočilnega sistema, ki se ločita predvsem v načinu posredovanja sporočil končnim uporabnikom.

Sporočilni sistem SNS (*Simple Notification Service*) je imenovan kot izdajatelj–naročnik (*angl. Publish – Subscribe System*). Naročnik se mora prijaviti na izbrano temo in ob vsakršni spremembi, torej pojavitvi novega sporočila, bo naročnik samodejno prejel obvestilo. Ob morebitni neaktivnosti naročnika v času oddaje sporočila, naročnik sporočila ne prejme. Običajno se ta sistem uporablja za storitve kot so e-pošta, socialna omrežja in drugo.

Sporočilni sistem SQS (*Simple Queue Service*) se od sporočilnega sistem SNS loči predvsem v tem, da je sporočilo lahko vzeto iz vrste le enkrat, torej slednji ne podpira posredovanja sporočil več uporabnikom. Ko prejemnik iz vrste vzame sporočilo, tega ni mogoče več ponovno pridobiti. Sistem SQS je običajno uporabljen pri integraciji in zagotavljanju šibke sklopljenosti sistemov. Pri uporabi tega sistema sporočilo čaka v vrsti, dokler uporabnik ni aktiven. S tem nam sistem zagotavlja zanesljivost dostave sporočil.

4.4.4.4 Končna točka AllegroGraph

Podatkovna baza AllegroGraph je ena izmed najbolj popularnih predstavnikov podatkovnih baz vrste *triplestore*. Naloga slednjih je oglaševanje končne točke in shranjevanje dokumentov zapisa *triple*, ki so predstavljeni kot *subject-predicate-object.* Na to sestavo bi lahko gledali tudi drugače, kjer je *subject* predstavljen kot identifikator vira, *predicate* kot ime objekta ter *object* kot vrednost [29]. Za uspešno uporabo poizvedovalnega jezika SPARQL je potrebno dokumente predstaviti v zapisu RDF, kjer je potrebo dosledno uporabljati imenske prostore. V predlagani arhitekturi smo podatkovno bazo AllegroGraph uporabljali za oglaševanje pretvorjenih dokumentov zapisa RDF/XML preko končne točke SOARQL. Za prenos dokumentov na končno točko AllegroGraph smo uporabili programski vmesnik, implementiran v programskem jeziku Python.

Dokumenti zapisa RDF/XML imajo zelo podobno obliko kot dokumenti zapisa XML, s to razliko, da slednji vsebujejo dodatne informacije, ki so nujno potrebne za opis ontologije. Veliko težo nosita atributa *rdf:about* in *rdf:resource.* Prvi predstavlja enolični identifikacijski ključ elementa *Description*, v katerega smo shranili identifikacijski ključ dokumenta, shranjenega v nerelacijski podatkovni bazi MongoDB. Drugi atribut pa predstavlja ime razreda, o katerem navedeni dokument zapisa RDF vsebuje podatke [Slika 17]. Za preverjanje pravilnosti dokumentov lahko uporabimo spletno storitev organizacije W3C [37], ki s pomočjo algoritma preveri pravilnost dokumentov zapisa RDF/XML.

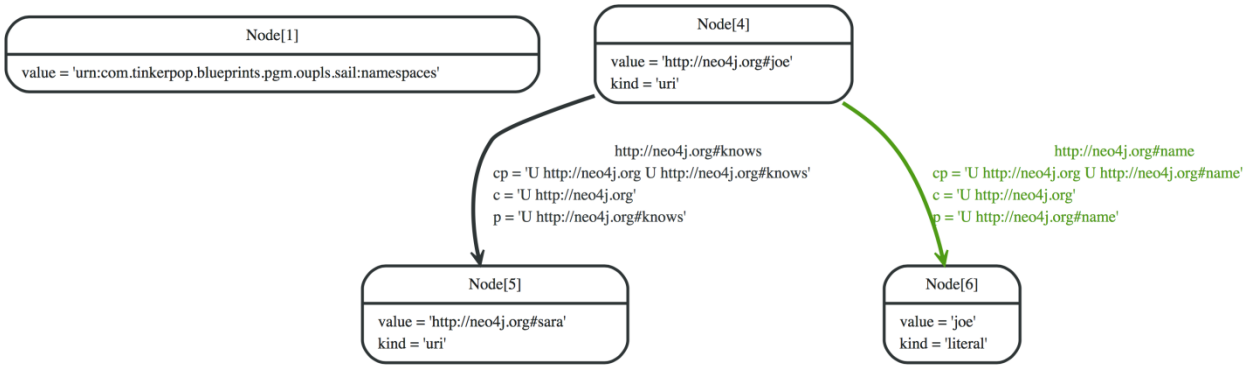
```
<?xml version="1.0" encoding="UTF-8"?>
<RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
     xmlns="http://www.w3.org/1999/02/22-rdf-syntax-ns#">
   <Description
rdf:about="http://example.org/oseba/53a35035a686369ad6581113">
      <type rdf:resource="http://mag.org/ontology/oseba"/>
      <lname xmlns:ns0="http://mag.org/ontology/">Rojko</lname>
      <mobile xmlns:ns0="http://mag.org/ontology/">040040999</mobile>
      <fname xmlns:ns0="http://mag.org/ontology/">Matej</fname>
   </Description>
</RDF>
```

Slika 17: Primer dokumenta zapisa RDF/XML.

4.4.4.5 Končna točka Neo4j

Nerelacijsko podatkovno bazo Neo4j lahko s pomočjo namestitve vtičnika, ki je bil razvit s strani *neo4j-contribute* [38], pretvorimo v končno točko. Kljub uporabi vtičnika moramo za uspešno izvajanje poizvedbe SPARQL dokumente pretvoriti v obliko zapisa RDF in jih ponovno shraniti v obstoječo nerelacijsko podatkovno bazo Neo4j. Pri pretvorbi moramo natančno upoštevati pravila, ki določajo pravilno uporabo imenskih prostorov, s katerimi definiramo lastnosti objektov in povezave med njimi. Za pravilno delovanje vtičnika moramo uporabljati imenske prostore, kot jih prikazuje primer, prikazan na uradni spletni strani Neo4j [39]. Pri definiranju povezav med objekti in opisu lastnosti objekta ločimo dva načina:

- za namen definiranja povezave med objekti moramo uporabljati lastnost *uri* (v spodnjem primeru je uporabljena beseda »knows«) [Slika 18] ter
- za namen opisa lastnosti objektov (ime, priimek …), moramo uporabljati lastnost imenovano *literal* (v spodnjem primeru je uporabljena beseda »name«) [Slika 18].



Slika 18: Predstavitev primera, navedenega na uradni spletni strani Neo4j.

Za ustvarjanje dokumentov zapisa RDF smo uporabili Apache Jena [40]. Ta definira programski vmesnik, s katerim je v programskem jeziku Java omogočeno generiranje dokumentov zapisa RDF. Izmed vseh možnih zapisov (*Turtle, N-Triples, N-Quads, JSON-LD, N3, RDF/XML*) smo izbrali osnovnega, poimenovanega *N-Triples*. Pretvorba dokumentov iz oblike točka-točka-povezava, ki je predstavljena v nerelacijski podatkovni bazi Neo4j, zaradi podobnih lastnosti, ki jih imajo dokumenti zapisa RDF, ne predstavlja večjih težav.

4.4.4.6 Poizvedovalni jezik SPARQL

Poizvedovalni jezik SPARQL se uporablja za poizvedovanje nad dokumenti zapisa RDF in semantičnim spletom. Poizvedovalni jezik SPARQL omogoča izvajanje poizvedb nad lokalnimi ali oddaljenimi podatkovnimi viri, ki so predstavljeni kot končne točke z oglaševanjem dokumentov zapisa RDF. Za razvoj poizvedb lahko uporabljamo vrsto različnih lastnosti poizvedovalnega jezika, kot so filtri, naslavljanje podatkovnih virov, vključitev dodatnega podatkovnega vira z uporabo storitev (*angl Services*), združevanje podatkov, itn. Za namen integracije podatkovnih baz in sočasnega poizvedovanja nad različnimi nerelacijskimi podatkovnimi viri smo implementirali poizvedbo, ki združuje podatke o osebah, shranjene v nerelacijski podatkovni bazi MongoDB, in podatke o poslanih sporočilih med telefonskimi številkami, shranjenimi v Neo4j. Poizvedovalni jezik s pomočjo izbranega ključa sam združi podatke in vrne imena ter priimke vseh oseb, ki so prejeli sporočilo od osebe s telefonsko številko 041041888 [Slika 19].

```
prefix mypref: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
select ?firstAndLastName ?mobilePhoneB where {
filter (?mobilePhonePerson = ?mobilePhoneB){
    SERVICE <http://54.72.50.104:10035/repositories/pythontutorial1>{
        select ?firstAndLastName ?mobilePhonePerson where {
            ?s mypref:fname ?fn .
            ?s mypref:lname ?ln .
            ?s mypref:mobile ?mobilePhonePerson
            bind(concat(str(?fn),\" \",str(?ln)) as ?firstAndLastName)
        }
    }}
    {
        {SELECT ?mobilePhoneB WHERE {
            ?personA <http://neo4j.org#likes> ?personB .
            ?personA <http://neo4j.org#mobile> ?mobilePhoneA .
            ?personB <http://neo4j.org#mobile> ?mobilePhoneB .
            FILTER regex(str(?mobilePhoneA),\"041041888\")
            }
        }
    }
}
group by ?mobilePhoneB ?firstAndLastName
```

Slika 19: Poizvedba, ki vrne podatke o osebah, ki so prejele sporočilo od osebe s telefonsko številko 041041888.

Poizvedbo smo z uporabo programskega orodja Curl [41] neposredno posredovali na nerelacijsko podatkovno bazo Neo4j, s katere smo s pomočjo značke SERVICE izvedli poizvedbo nad oddaljenim podatkovnim virom AllegroGraph, gostujočim na javnem ponudniku storitev v oblaku Amazon AWS. Filter v poizvedbi predstavlja združevanje podatkov, kjer rezultate iz prve poizvedbe z uporabo ključa telefonske številke povežemo z rezultati druge poizvedbe. V prvi poizvedbi (predstavljeni pod značko SERVICE) pridobimo vse osebe z imeni in priimki ter telefonske številke, medtem ko druga poizvedba vrne podatke o relacijah, torej katera telefonska številka je prejela sporočilo od pošiljatelja s telefonsko številko 041041888. Na koncu smo z značko *Group by* združili rezultate, ki predstavljajo isto entiteto.

Iz rezultata poizvedbe [Slika 20] je razvidno, katere osebe so prejele sporočilo od osebe z izbrano telefonsko številko. Zaradi naključnih podatkov, ki smo jih vstavili v Neo4j, v navedenem primeru tudi pošiljatelj predstavlja entiteto prejemnika.

```
[ {
  "mobilePhoneB" : "\"040040666\"",
  "firstAndLastName" : "\"Ruby Wolf\""
}, {
  "mobilePhoneB" : "\"041041888\"",
  "firstAndLastName" : "\"Kaylee Schwarz\""
}, {
  "mobilePhoneB" : "\"070070777\"",
  "firstAndLastName" : "\"Madelyn Phillips\""
}, {
  "mobilePhoneB" : "\"041041999\"",
  "firstAndLastName" : "\"Eliana Mayr\""
} ]
```


Slika 20: Rezultat poizvedbe SPARQL.

4.5 Ugotovitve

V tem poglavju smo analizirali možnosti integracije povpraševanj podatkovnih baz. Menimo, da vsi načini, ki so uporabljeni pri relacijskih podatkovnih bazah, niso primerni za integracijo podatkov v svetu nerelacijskih podatkovnih baz. Vzrok lahko najdemo v količini podatkov in različnih vrstah nerelacijskih podatkovnih baz, med katerimi je vsaka namenjena shranjevanju točno določene vrste podatkov. Za izvajanje poizvedb nad različnimi podatkovnimi viri s skupnim poizvedovalnim jezikom smo predlagali sistem integracije s pomočjo semantičnega spleta. Ta je v osnovi namenjen poizvedovanju nad različnimi podatkovnimi viri. Če bi želeli implementirati generično rešitev, ki bi podpirala integracijo podatkovnih virov s semantičnim spletom, bi morali veliko energije posvetiti teoriji o integraciji ontologij. Ta predstavlja predvsem definiranje besednjaka, ki opisuje pomen podatkov. V magistrski nalogi smo ta problem izpostavili, vendar smo pri implementaciji predpostavili, da uporabnik, ki želi poizvedovati nad podatki, porazdeljenimi na različnih podatkovnih virih, dobro pozna problemsko domeno in je zmožen izbire pravilnega integracijskega ključa.

Za namen integracije s semantičnim spletom smo predlagali dva načina, pri čemer prvi predstavlja preslikavo poizvedovalnega jezika SPARQL v proceduralni jezik in programsko združevanje rezultatov, medtem ko drugi predstavlja pretvorbo in približevanje podatkov poizvedovalnemu jeziku SPARQL. Za integracijo nerelacijskih podatkovnih virov, ki bi hranili izjemno veliko število dokumentov, bi bila izbira prvega načina primernejša, saj podpira izvajanje poizvedb nad kompleksnimi dokumenti zapisa JSON. Za implementacijo prvega načina bi morali implementirati končno točko, kjer bi preko spletnih storitev RESTful na vhodni parameter prejeli poizvedbo SPARQL ter jo pretvoriti v proceduralni jezik, ki bi s pomočjo programskih vmesnikov pridobil zahtevane podatke [42]. Te podatke bi pozneje v implementirani funkciji programsko združili v logično celoto ter jih vrnili kot rezultat poizvedbe. Poleg razvoja

lastne končne točke bi lahko uporabili tudi obstoječe strežnike (npr. Apache Fuseki, AllegroGraph) in jih integrirali z izbrano nerelacijsko podatkovno bazo.

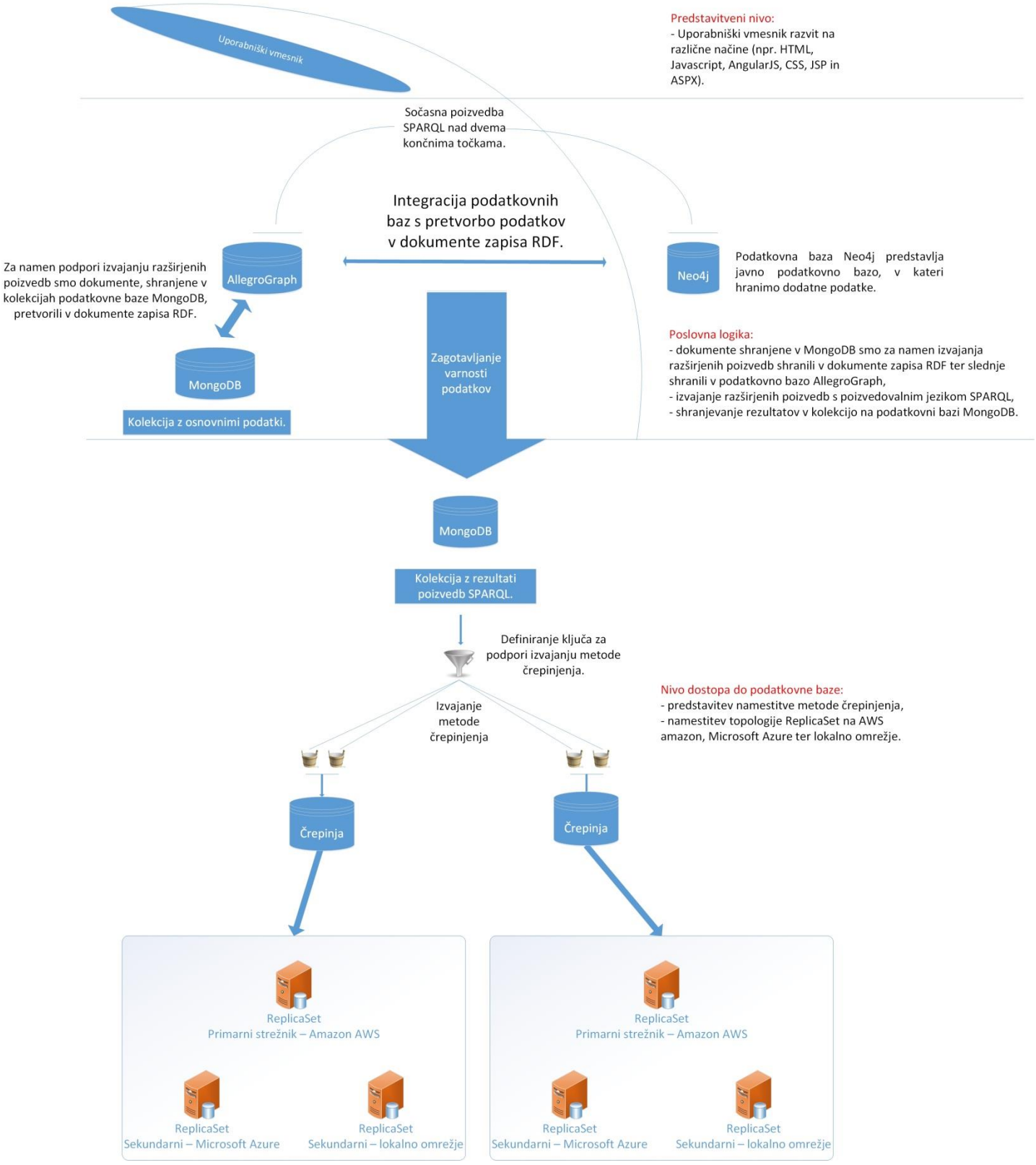
Pri uporabi drugega načina ne potrebujemo preslikave poizvedovalnega jezika SPARQL v proceduralni jezik, vendar moramo z različnimi koraki in transformacijo XSLT pretvoriti dokumente v zapis RDF/XML. Pri pretvorbi dokumentov moramo podpreti sinhronizacijo med nerelacijsko podatkovno bazo MongoDB ter končno točko AllegroGraph. Slednje je možno implementirati z uporabo vtičnika MongoWatch, ki beleži spremembe nad izbrano kolekcijo.

5. Arhitekturna zasnova rešitve za integracijo povpraševanj

Arhitekturna zasnova opisuje izbiro komponent, potrebnih za realizacijo rešitve povpraševanj z uporabo semantičnega spleta in ontologije. Arhitekturo smo ločili v tri dele [Slika 21] in jih povezali z logičnim konceptom trinivojske arhitekture [3.2 Večnivojska arhitektura]:

- predstavitveni nivo,
- poslovna logika in
- nivo dostopa do podatkovne baze.

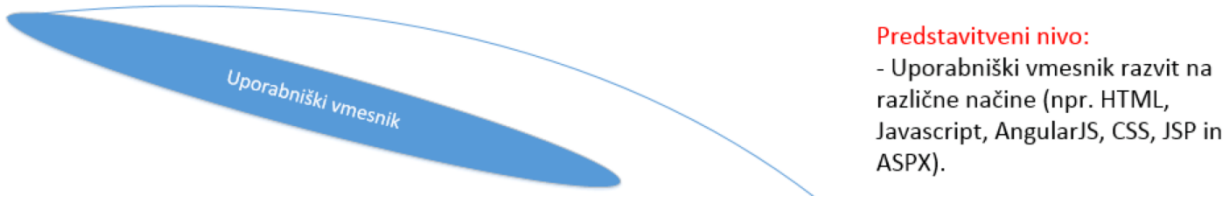
Pri implementaciji predlagane arhitekture smo izbrali dve različni vrsti nerelacijskih podatkovnih baz *dokument* - MongoDB in *graf* - Neo4j. Nerelacijsko podatkovno bazo *dokument* smo uporabili za primarni vir shranjevanja podatkov ter analizo učinkovitosti metode črepinjenja. Za prikaz integracije različnih vrst nerelacijskih podatkovnih baz in izvajanja razširjenih poizvedb SPARQL [4.4.4.6 Poizvedovalni jezik SPARQL] smo del podatkov shranili na nerelacijsko podatkovno bazo *graf*. Arhitektura vključuje prednosti računalništva v oblaku in za namen implementacije *proof of concept* predvideva uporabo modela MVC [3.3 Model MVC (Model-View-Controller)].



Slika 21: Predlagana arhitektura, uporabljena v magistrski nalogi.

5.1 Predstavitveni nivo

Predstavitveni nivo v osnovi predstavlja vmesnik med uporabniki ali drugimi aplikacijami [Slika 22].

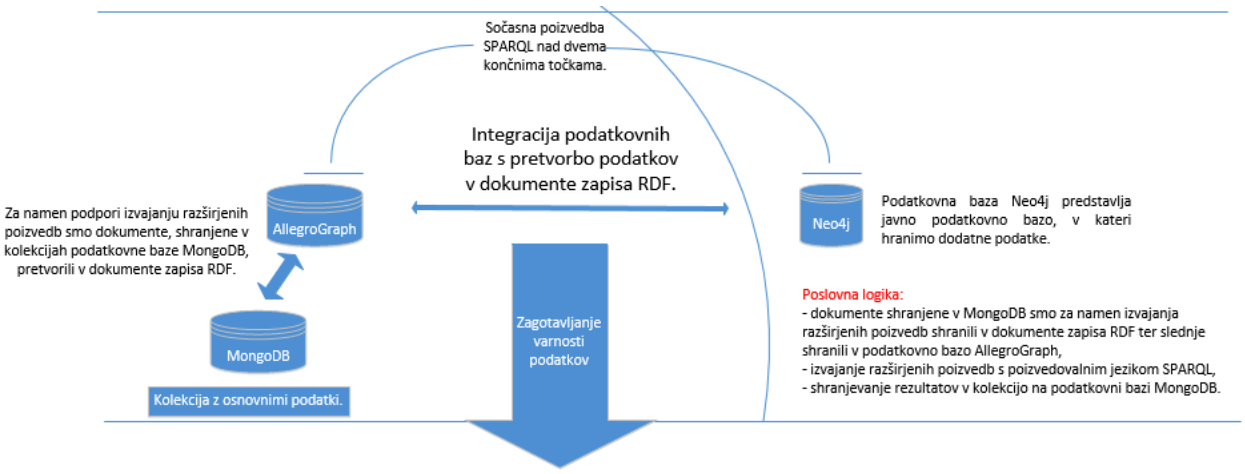


Slika 22: Prikaz predstavitvenega nivoja.

Predstavitveni nivo v modelu MVC je predstavljen z modulom *pogled*. Za namen implementacije aplikacije *Proof of Concept* smo uporabili razvojno ogrodje Django, ki v osnovi sledi konceptu modela MVC. Pri implementaciji predstavitvenega nivoja smo razvili spletno stran, ki omogoča uporabnikom interakcijo pri integraciji nerelacijskih podatkovnih baz ter merjenje vpliva vertikalne oziroma horizontalne razširljivosti sistema. [7.2 Analiza vpliva vertikalne in horizontalne razširljivosti na učinkovitost delovanja sistema]. Za razvoj uporabniškega vmesnika smo uporabili jezik HTML v povezavi s CSS in Javascript.

5.2 Poslovna logika

Poslovna logika [Slika 23] združuje vse pomembne dele magistrske naloge, saj vključuje tako integracijo različnih nerelacijskih podatkovnih baz [4.4.2 Integracija s pomočjo semantičnega spleta] kot izvajanja meritev učinkovitosti metode črepinjenja [7.2 Analiza vpliva vertikalne in horizontalne razširljivosti na učinkovitost delovanja sistema].



Slika 23: Prikaz poslovne logike.

Z analizo različnih vrst nerelacijskih podatkovnih baz [2. Pregled nerelacijskih podatkovnih baz] smo ugotovili, da je izbira vrste nerelacijske podatkovne baze močno odvisna od problemske domene, torej bi za učinkovito analizo relacij med vozlišči izbrali drugo vrsto nerelacijske podatkovne baze kot za shranjevanje geografskih točk. Ta del arhitekture z uporabo sočasnih poizvedb [4.4.2 Integracija s pomočjo semantičnega spleta] predpostavlja integracijo dveh različnih vrst nerelacijskih podatkovnih baz *graf* in *dokument.* Sočasne poizvedbe poznamo pri uporabi semantičnega spleta, kjer skupni poizvedovalni jezik SPARQL omogoča samodejno združevanje rezultatov poizvedb iz različnih podatkovnih virov.

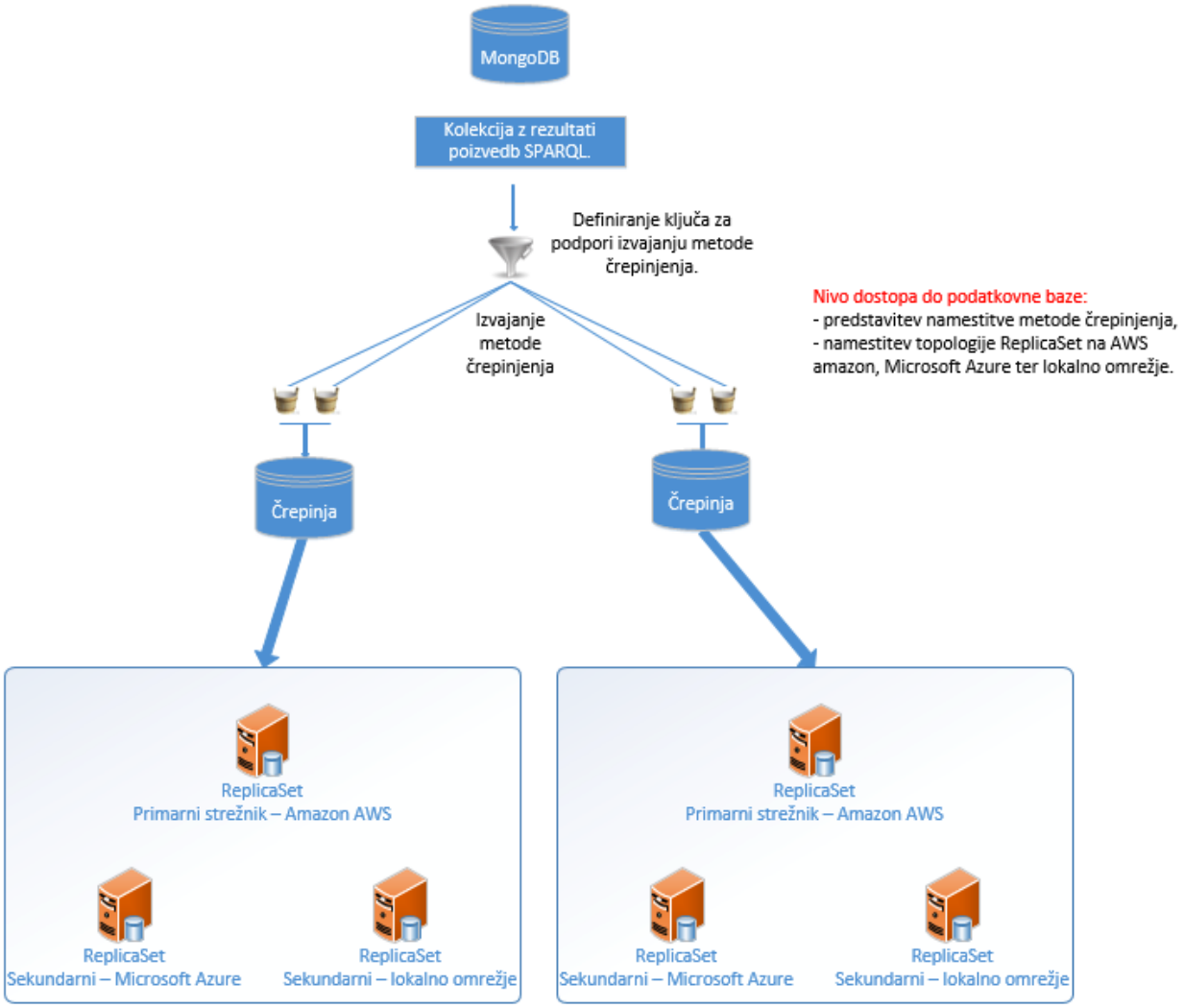
Motivacijo za integracijo nerelacijskih podatkovnih baz smo našli v svetu bioinformatike. To je veda, ki združuje biologijo in računalniško tehnologijo za namen učinkovitega shranjevanja, širjenja in analize podatkov [43]. Podatki v bioinformatiki vsebujejo informacije o sekvenci DNK. Razumevanje in predvsem primerjava podatkov z drugimi javno dostopnimi podatkovnimi bazami (npr. Uniref, Enzyme, RefSeq, NCBI Taxonomy) je ključ za uspeh pravilne interpretacije mutacij. V svetu bioinformatike imamo opravka z veliko količino podatkov, ki jih moramo za namen poznejših analiz učinkovito shraniti v podatkovno bazo. Za to opravilo lahko namesto relacijskih izberemo nerelacijske podatkovne baze (npr. MongoDB, Neo4j, Hadoop).

Poleg možnosti primerjanja podatkov z drugimi javno dostopnih podatkovnimi bazami obstaja na področju bioinformatike zanimiv projekt Bio4j [44], ki s pomočjo semantičnega spleta in dokumentov zapisa RDF združuje večino pomembnih podatkovnih baz v svetu bioinformatike. Ta projekt za osnovo podatkovne arhitekture uporablja nerelacijsko podatkovno bazo *graf* -Neo4j [2.2 Nerelacijska podatkovna baza *graf*].

Projekt Bio4j je potrebno gostovati na javnem ponudniku storitev v oblaku Amazon AWS, kjer instanca zavzema približno 200 GB podatkovnega prostora. Na sliki navidezne naprave je za takojšen začetek dela nameščena vsa potrebna programska oprema. Če bi želeli instanco nerelacijske podatkovne baze *graf* gostovati na drugem ponudniku javnih storitev v oblaku ali lokalnem omrežju, bi morali za ta namen namestiti vso zahtevano programsko opremo ter preslikati podatke. Zaradi sistemskih zahtevnosti, predvsem zadostne količine bralno-pisalnega pomnilnika, moramo nerelacijsko podatkovno bazo *graf,* projekta Bio4j gostovati na navidezni napravi (*angl. Virtual Machine*) tipa *X-Large*. Kompleksnost podatkov, ki se nanašajo na specifično problemsko domeno bioinformatike, ter stroški izvajanja, ki so ob 14-dnevnem testiranju presegli 50 EUR, sta prevladujoča razloga, zakaj za prikaz integracije nerelacijskih podatkovnih baz nismo izbrali projekta Bio4j. Namesto tega smo uporabili domeno oseb in pošiljanje tekstovnih sporočil med njimi.

5.3 Nivo dostopa do podatkovne baze

Nivo dostopa do podatkovne baze [Slika 24] v našem primeru predstavlja shranjevanje podatkov na visoko razpoložljivi sistem črepinj ter uporabo replikacije, nameščene na različnih ponudnikih računalništva v oblaku ter lokalnim omrežjem [6. Visoka razpoložljivost, razširljivost in črepinjenje]. Učinkovitost črepinjenja podatkov je v veliki meri odvisna od izbire ključa, po katerem bo metoda črepinjenja drobila podatke [6.2.1 Izbira ustreznega ključa]. Za zagotavljanje visoke razpoložljivosti smo vsako črepinjo namestili na sistem *ReplicaSet*, kjer v primeru izpada ene pripete instance v sistemu vlogo prevzamejo druge [6.5.2 Različni načini sistema replikacije na podatkovni bazi MongoDB].



Slika 24: Prikaz nivoja dostopa do podatkovne baze.

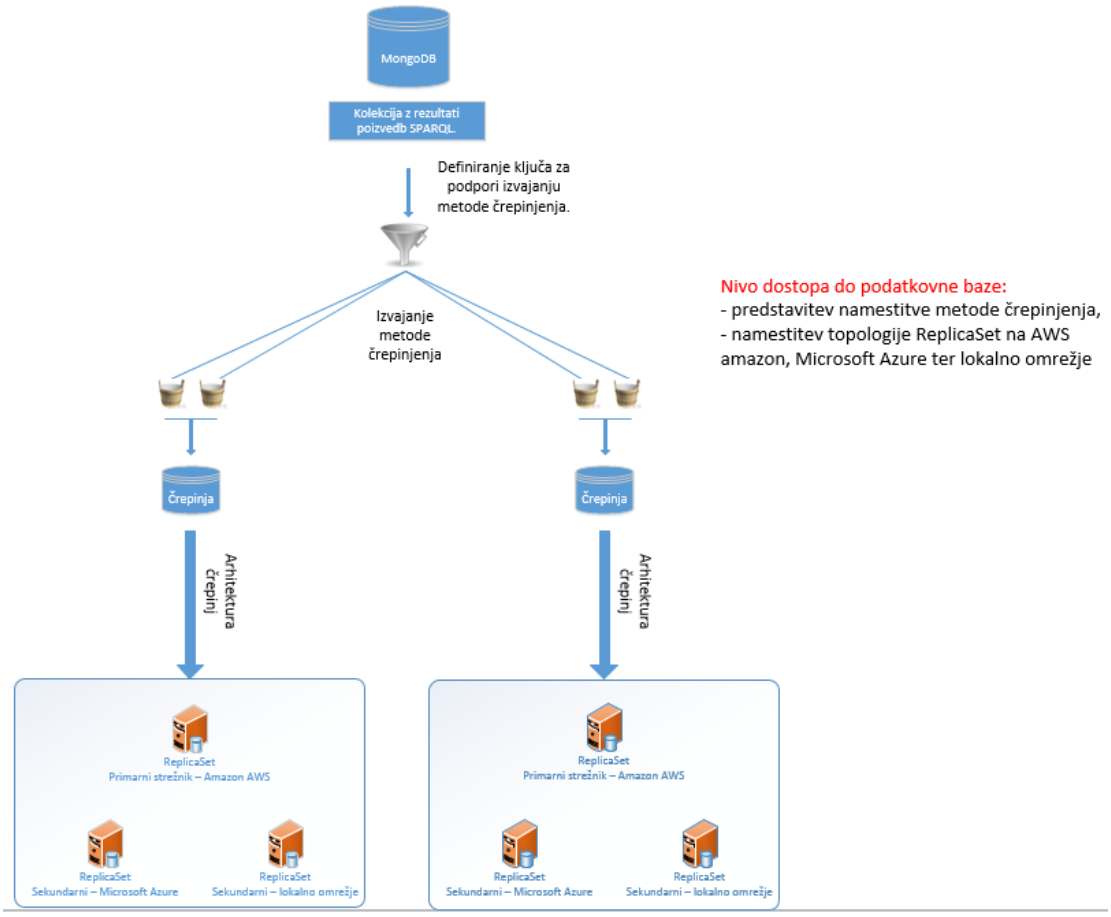
Za namen implementacije predlagane arhitekture smo v pretežnem delu uporabili storitveni model IaaS, ki ga ponujata ponudnika javnega oblaka Amazon AWS in Microsoft Azure. Za zagotavljanje visoke razpoložljivosti arhitekture smo na različne navidezne naprave tipa *micro* namestili nerelacijsko podatkovno bazo MongoDB in s pravilnimi nastavitvami povezali instance v logično gručo. V tem delu se nam je zdela zanimiva ideja o namestitvi instanc na dva različna ponudnika storitev v oblaku ter lokalno omrežje [6.6 Namestitev replikacije sistema na različne ponudnike storitev v oblaku].

Pri izdelavi analize vpliva horizontalne oziroma vertikalne razširljivosti sistema [6.4 Analiza učinkovitosti] smo uporabili navidezne naprave tipa *t2.micro* oziroma *t2.medium* in na topologijo povezanih instanc shranjevali dokumente zapisa JSON ter merili čase.

6. Visoka razpoložljivost, razširljivost in črepinjenje

Z uporabo horizontalne oziroma vertikalne razširljivosti sistema povečamo učinkovitost na nivoju navideznih naprav. Nivo navideznih naprav je osnova za uporabo metode črepinjenja ter replikacije sistema, ki skupaj zagotavljata visoko razpoložljiv sistem na podatkovnem nivoju [45].

Podatke, ki bi bili pridobljeni z integracijo različnih podatkovnih virov, bi v primeru velikih količin morali shraniti na porazdeljeni sistem instanc. Ta sistem zahteva uporabo metode črepinjenja, kjer se podatki z enim izmed dveh osnovnih algoritmov drobijo na črepinje sistema. Vsaka črepinja naj bi bila podprta z uporabo replikacije, ki zagotavlja nenehno delovanje tudi v primeru izpada ene ali več navideznih naprav [Slika 25].



Slika 25: Nivo dostopa do podatkovne baze.

6.1 Razširljivost sistema

Ko govorimo o razširljivosti sistema, govorimo predvsem o možnosti dodeljevanja virov, ki so potrebni za tekoče delovanje aplikacij. Razširljivost merimo s številom uporabnikov, ki jih lahko izbrana aplikacija gosti v določenem času. Ta doseže svoje meje takrat, ko kritični deli arhitekture presegajo zmogljivost strojne opreme. Za zagotavljanje razširljivosti sistema poznamo dva prevladujoča pristopa [46]:

- vertikalno razširljivost sistema (*angl. Scale Up*) in
- horizontalno razširljivost sistema (*angl. Scale Out*).

6.1.1 Vertikalna razširljivost sistema

Vertikalna razširljivost sodi med preproste načine, s katerimi se je večina sistemskih administratorjev srečala v preteklosti. Ideja je predvsem dodajanje novih sistemskih virov (npr. procesorska moč, delovnega pomnilnika, diskovnega polja …) obstoječim strežnikom. Izbira tega načina za zagotavljanje razširljivosti sistema ne prinaša večje kompleksnosti ali tveganj, saj se koncept delovanja za obstoječe aplikacije ne spremeni. Pri omenjenem načinu lahko kljub nadgradnji strojne opreme hitro dosežemo nove omejitve.

6.1.2 Horizontalna razširljivost sistema

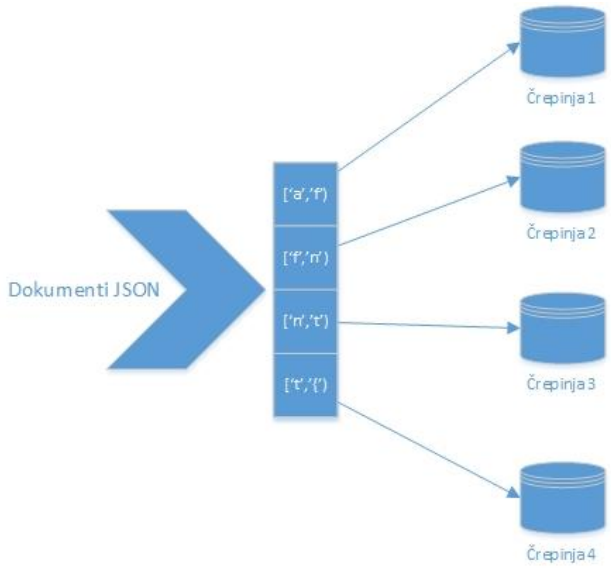
Horizontalna razširljivost sistema poveča odzivnost aplikacij z možnostjo dodajanja novih vozlišč. Vozlišča so lahko homogena, torej vsebujejo enako količino dodeljenih virov navideznim napravam, ali heterogena. Zagotavljanje razširljivosti sistema s homogenimi vozlišči je preprostejše, saj lahko z izenačevalnikom obremenitev zagotovimo porazdelitev obremenjenosti sistema. V osnovi izenačevalnik obremenitev zagotavlja oskrbovanje z dodajanjem ali odstranjevanjem navideznih naprav ter dodajanje in odvzemanje razpoložljivih virov posamezne navidezne naprave. Dodajanje oziroma odvzemanje virov se lahko izvaja samodejno ali ročno. Pri ročnem načinu zagotovimo razpoložljive vire, kamor lahko izenačevalnik usmerja zahteve. Za svoje delovanje lahko izenačevalnik obremenitev uporablja enega od navedenih algoritmov [47] (npr. Round-robin, Ant Colony, Map Reduce, MaxMin, MinMin …). Prednost dinamične alokacije navideznih naprav ali virov posamezne navidezne naprave je samodejni odziv na število uporabnikov, ki v danem trenutku dostopajo do aplikacije sistema. Prav zaradi dodajanja in odvzemanja strežnikov je cena gostovanja aplikacije na javnem ponudniku storitev v oblaku odvisna od števila aktivnih uporabnikov. Med izenačevalnike obremenitev sodijo (AWS Elastic Load Balancing, Azure Traffic Manager, OpenStack – Load Balancer as a Service).

6.2 Črepinjenje

Metodo črepinjenja smo v predlagano arhitekturo umestili v nivoju dostopa do podatkovne baze, kjer metoda drobi podatke na predhodno nameščen in horizontalno razširjen sistem. Pri uporabi metode črepinjenja mora biti logični pogled na porazdeljene podatke enak podatkom, shranjenim na eni instanci podatkovne baze. Pred pričetkom izvajanja metode črepinjenja je potrebno definirati ključe, s katerimi delimo podatke na porazdeljeni sistem naprav. Pravilno definiranje ključev je močno povezano z učinkovitostjo sistema. V primeru izbire napačnega ključa sistem ne omogoča enostavne zamenjave le-tega. Proces zamenjave ključa bi zahteval izvoz in ponovni uvoz podatkov ter definiranje novega ključa. Zaradi drobljenja ter velike količine podatkov lahko zahtevnost menjave ključa enačimo s spremembo sheme v relacijski podatkovni bazi.

6.2.1 Izbira ustreznega ključa

Za namen analize delovanja metode črepinjenja smo simulirali rezultat integracije nerelacijskih podatkovnih baz MongoDB in Neo4j. Učinkovitost metode črepinjenja je povezana z izbiro ključa. Če bi na podlagi rezultatov poizvedbe iz prvega dela magistrske naloge izbrali za ključ črepinjenja priimke oseb, potem bi metoda črepinjenja drobila podatke v vedra *(angl. Bucket)* po začetnih črkah priimkov [Slika 26].



Slika 26: Deljenje podatkov na porazdeljeni sistem črepinj in definiranje ključev.

Nerelacijska podatkovna baza MongoDB pri izvajanju metode črepinjenja vse podatke shranjuje v vedra in le-te za namen približno enakomerne porazdelitve prenaša med navideznimi

napravami. Ker strežniki, ki predstavljajo črepinje sistema, ne vsebujejo zgolj enega območja podatkov, se ob dodajanju nove črepinje podatki delno kopirajo na dodano navidezno napravo [48].

6.2.1.1 Slabe prakse za izbiro ključev

Izbira ključa s t. i. binarno vrednostjo, ki omogoča deljenje podatkov zgolj na dve črepinji, je napačna. Poglejmo primer, kjer bi uporabnike, ki smo jih shranili v nerelacijsko podatkovno bazo MongoDB, delili po spolu. Ob zadostni količini podatkov bi metoda črepinjenja razdelila dokumente glede na spol. Ob dodajanju novih uporabnikov bi se obremenjenost črepinj le povečala. Ker smo izbrali napačni ključ, ki ne omogoča dodatnega drobljenja podatkov, bi morali za nadaljnje drobljenje podatkov na aplikacijskem nivoju implementirati svoj algoritem črepinjenja. Slednje bi privedlo do velike kompleksnosti sistema.

Izbira časovne značke za ključ bi zadnje osvežene podatke vedno drobila na enem samem strežniku. Ker so uporabniki poslovnih aplikacij običajno zainteresirani za zadnje podatke (npr. pošiljanje sporočil med uporabniki), se srečamo s primerom vroče točke (*angl. Hot Spot*). Vroča točka je definirana kot najbolj obremenjeni del sistema.

6.2.1.2 Dobre prakse za izbiro ključev

Z metodo črepinjenja si želimo doseči, da so podatki logično združeni na eni črepinji, torej da za večino poizvedb ne potrebujemo naslavljati vseh črepinj v sistemu. Za dosego tega cilja je primer dobre prakse izbira sestavljenega ključa [48].

Primer sestavljenega ključa je kombinacija datuma in tistega polja, nad katerim običajno izvajamo poizvedbe. Če navedemo primer, pri katerem bi za sestavljeni ključ izbrali datum ter priimke oseb, potem bi sistem za npr. mesec maj delil podatke na naslednji način:

$$((-\infty, -\infty), (05, \text{»Rojko«})) \text{ in } [(05, \text{»Rojko«}),(\infty, \infty))$$

Ob zadostni količini podatkov bi sistem ustvaril nova vedra, ki bi vsebovala podatke za mesec (npr. avgust). Ker naše poizvedbe ne bi več pogosto dostopale do podatkov za mesec maj, bi bili ti podatki odstranjeni iz delovnega pomnilnika.

6.3 Uporaba metode črepinjenja in horizontalne razširljivosti za zagotavljanje visoke razpoložljivosti sistema

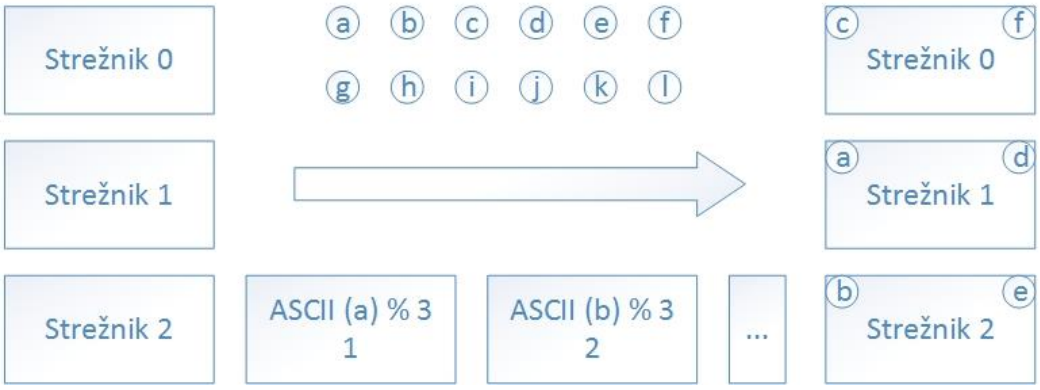
Metoda črepinjenja in horizontalna razširljivost predstavljata uporabo porazdeljenega sistema. Ker so podatki v relacijskih podatkovnih bazah porazdeljeni po tabelah, uporaba metode črepinjenja zaradi kompleksnosti združevanja podatkov v preteklosti ni bila zelo priljubljena. V tem primeru so bili podatki deljeni na nivoju aplikacije, kar je predstavljalo dodatni nivo kompleksnosti. Z nerelacijskimi podatkovnimi bazami se je zgodba predvsem zaradi predstavitve celovitih podatkov na enem mestu (npr. zapis JSON, vrednost v nerelacijski podatkovni bazi *ključi-vrednosti* ...) obrnila. S tem veliko nerelacijskih podatkovnih baz omogoča samodejno drobljenje podatkov (npr. MongoDB, Apache HBase, Cassandra) in posledično podatkovnem nivoju uporabo horizontalne razširljivosti sistema.

V nerelacijskih podatkovnih bazah poznamo naslednje algoritme za drobljenje podatkov [49]:

- osnovni algoritem črepinjenja podatkov (*angl. Regular Sharding),*
- algoritem konsistentnega zgoščevalnega obroča (*angl. Consistent Hash Rings)* in
- algoritem z definiranjem obsega ključev (*angl. Range Partitioning*).

6.3.1 Osnovni algoritem črepinjenja podatkov

Osnovni algoritem za drobljenje podatkov uporablja ostanek pri deljenju. Za osnovno črepinjenje podatkov lahko predpostavimo, da želimo drobiti vrednosti ASCII na tri različne strežnike. Torej za vrednost ASCII črke »a« bi veljalo (*97 % 3 = 1*) [Slika 27].

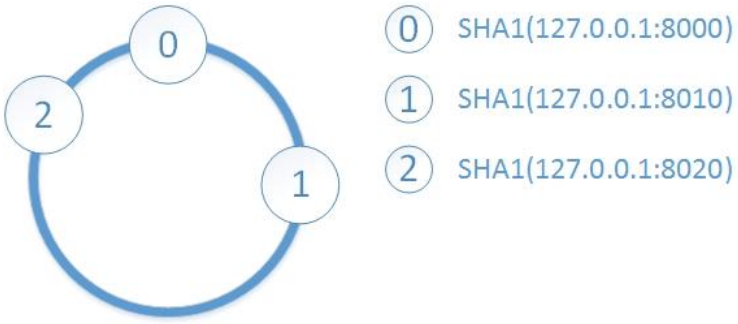


Slika 27: Primer z uporabo osnovnega algoritma črepinjenja podatkov.

Ta algoritem je preprost in deluje vse dokler ne dodamo ali odvzamemo strežnika iz gruče. V tem primeru bi se rezultat ostanka deljenja spremenil in s tem obstoječi podatki ne bi bili več pravilno naslovljeni. Rešitev slednjega problema je ponovni izračun vrednosti za obstoječe podatke.

6.3.2 Algoritem konsistentnega zgoščevalnega obroča

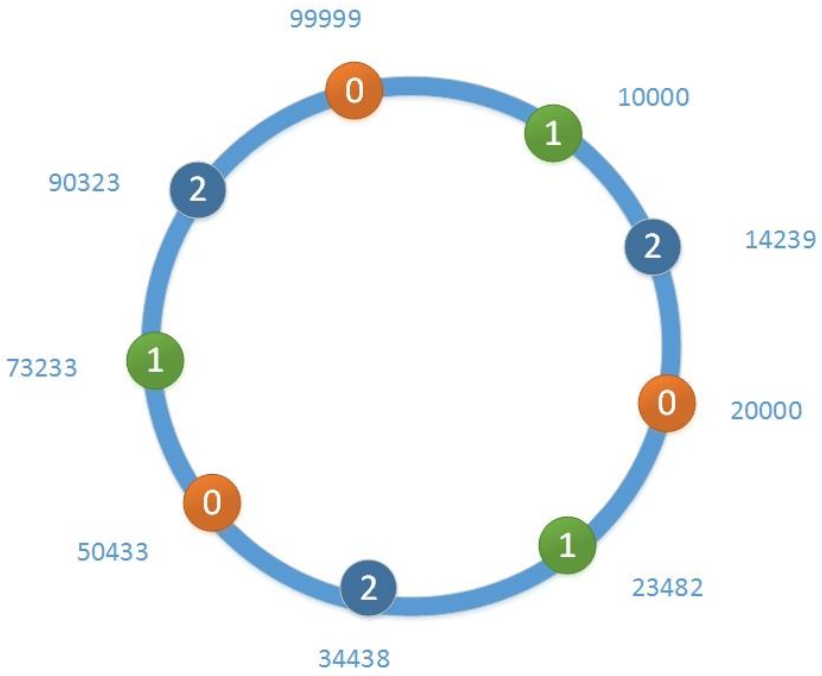
Dobra zgoščevalna funkcija porazdeli podatke na približno enakomeren način. To omogoča razdelitev dokumentov, predstavljenih s ključi in vrednostmi na horizontalno porazdeljen sistem strežnikov. Osnovna ideja algoritma je izbira enoličnega identifikacijskega ključa (npr. kombinacija lokalnega naslova IP ter vrat). Z izračunom izvlečka, kjer lahko uporabimo enega izmed možnih algoritmov, med drugim: CRC32, SHA1, MD, določimo obsege na obroču. Ker rezultati izvlečkov predstavljajo začetne vrednosti posameznega strežnika, so strežniki na obroču porazdeljeni neenakomerno. To je logična posledica generiranja izvlečkov, saj rezultat teh funkcij predstavljajo naključne vrednosti [Slika 28]. Neenakomerna porazdelitev strežnikov na obroč pomeni neenakomerno porazdelitev količine podatkov na strežnike.



Slika 28: Porazdelitev strežnikov na obroč.

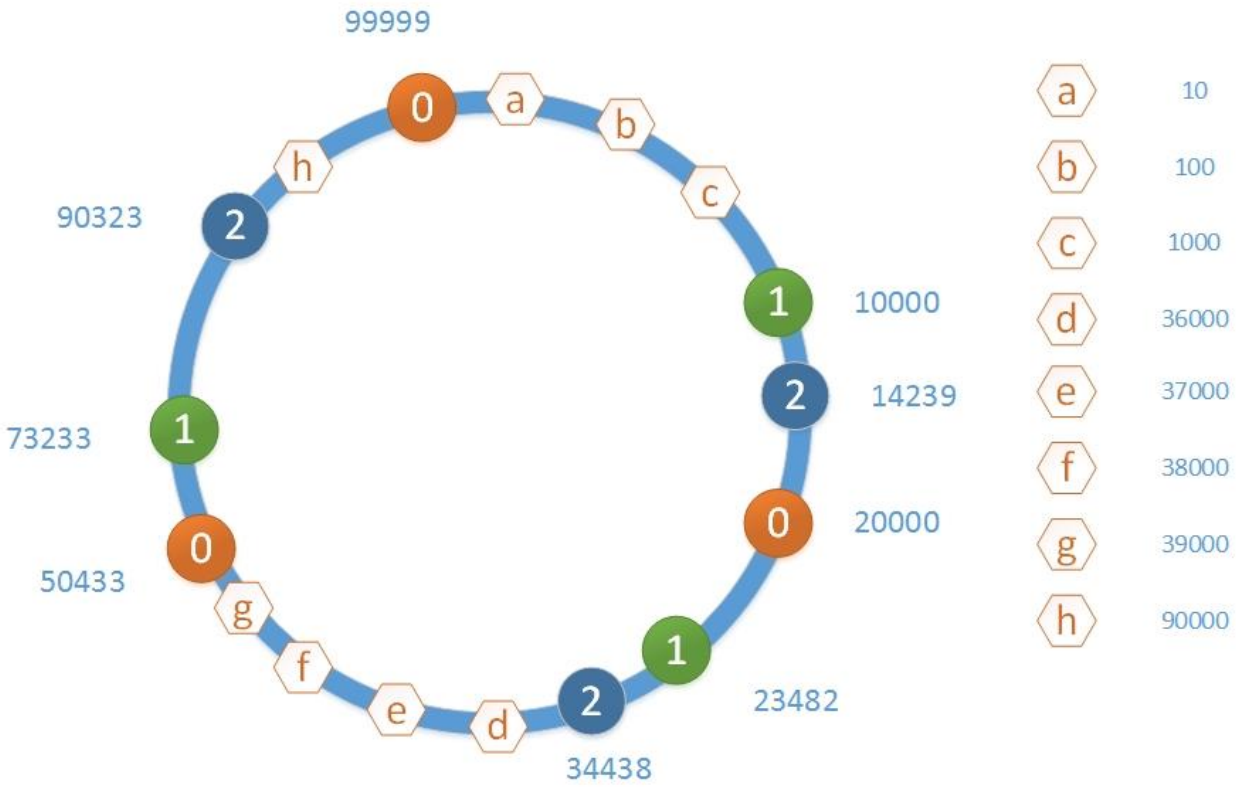
Ob morebitnem izpadu strežnika 1, bi bili vsi novi podatki nenadoma preusmerjeni na strežnik 2. To bi predstavljalo veliko obremenitev, zato algoritem težavo rešuje z dodelitvijo navideznih vozlišč (*angl. Virtual Nodes)* za vsak fizični strežnik.

Če bi strežniku 0 dodelili tri navidezna vozlišča, potem bi enolično identifikacijsko številko razširili z dodatno vrednostjo (npr. *0-127.0.0.1:8000, 1-127.0.0.1:8000, 2-127.0.0.1:8000*) in izračunali izvlečke za vse strežnike in navidezna vozlišča. Na tej točki algoritem zaradi večjega števila navideznih strežnikov porazdeljenih na obroč, zagotavlja približno enakomerno porazdelitev fizičnih strežnikov [Slika 29].



Slika 29: Porazdelitev navideznih vozlišč.

Za pravilno delovanje moramo za vsak dokument sprva izračunati izvleček, s katerim lahko algoritem usmeri dokument na pravilno navidezno vozlišče [Slika 30]. Ob rasti količine podatkov bi bili podatki približno enakomerno porazdeljeni na vsa tri vozlišča [Slika 30]. Ob izpadu strežnika 0 bi v navedenem primeru izgubili le tri dokumente, medtem ko bi se brez uporabe navideznih naprav to število lahko močno povečalo. Algoritem omogoča ustvarjanje večjega števila navideznih vozlišč za zmogljivejše fizične strežnike. Ta lastnost zagotavlja, da lahko zmogljivejši fizični strežniki hranijo večje količine podatkov. Ob morebitnem izpadu fizičnega strežnika se dokumenti izgubijo, vendar je odstotek izgubljenih dokumentov manjši kot ob neuporabi navideznih vozlišč.



Slika 30: Obroč s porazdeljenimi podatki.

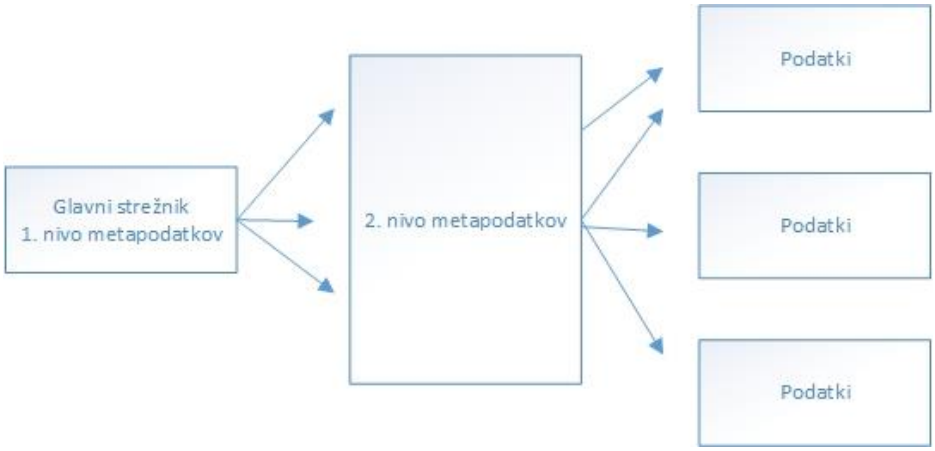
6.3.3 Algoritem z določanjem obsega ključa

Pri metodi črepinjenja, ki deluje z algoritmom definiranja obsega ključev, izenačevalnik hrani metapodatke o tem, kateri strežnik v sistemu vsebuje podatke z določenim obsegom ključev. Ti metapodatki so uporabljeni za preusmerjanje poizvedb na strežnike, ki vsebujejo ustrezne podatke. Tako kot algoritem zgoščevalnih obročev tudi tu definiramo obsege ključev, kjer pa ključi niso izračunani s pomočjo zgoščevalne funkcije. Slednje pomeni, da je definiranje učinkovitega ključa črepinjenja izjemno pomembno, saj bodo podatki zaporedno razvrščeni po sistemu črepinj. Za zagotavljanje približne porazdelitve podatkov algoritem omogoča krčenje obsega ključev za tiste obsege, v katere se podatki največkrat preslikajo.

Algoritem z določanjem obsega ključev, ki ga uporablja Google BigTable, je opisan v članku [50]. Za storitve, ki jih ponuja podjetje Google (npr. Google Maps, Google Earth), je črepinjenje ključnega pomena. Pri navedenih sistemih govorimo o popolnoma drugačni količini podatkov, kot smo jih obravnavali pri izdelavi magistrske naloge.

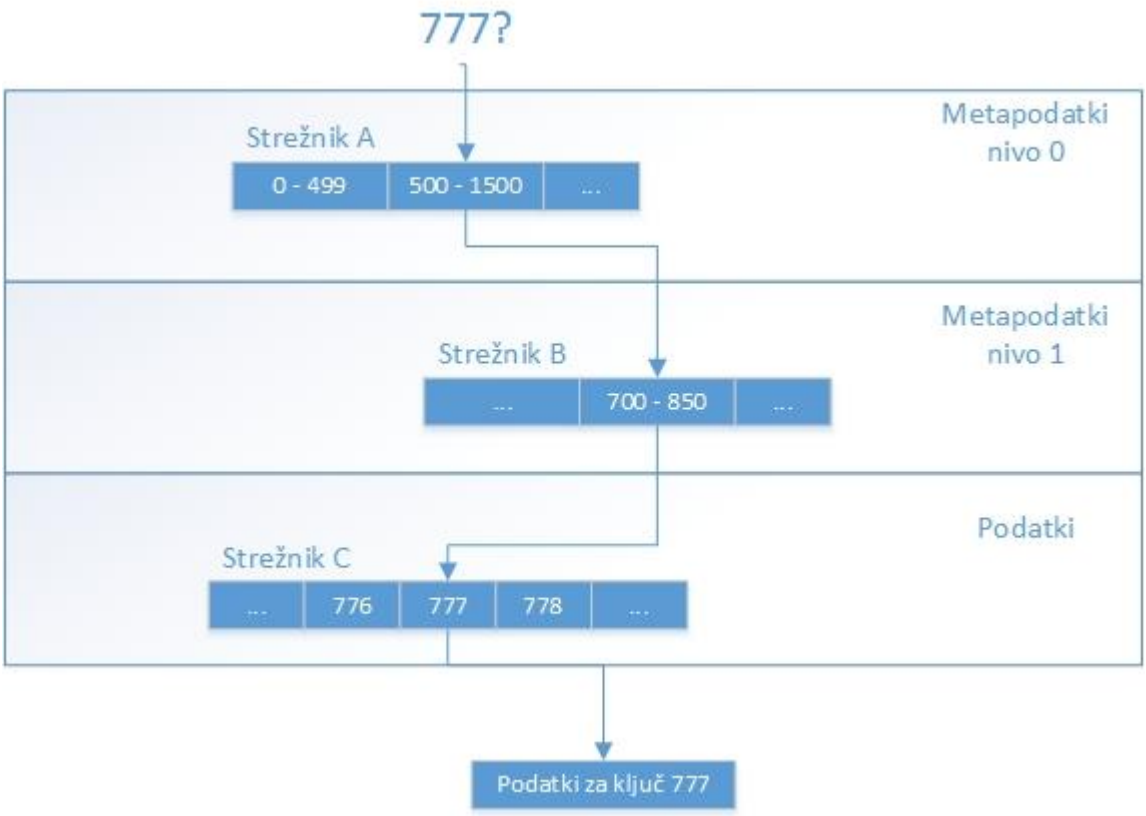
V osnovi so strežniki razdeljeni v tri nivoje [Slika 31], med katerimi prvi in drugi hranita metapodatke ter tretji končne podatke, shranjene v sistemu. Drugi nivo, ki hrani metapodatke, se

pojavi predvsem zaradi shranjevanja enormnih količin podatkov »BigData«, pri katerih lahko pričakujemo, da bomo delili tudi metapodatke na dodatne strežnike. Poizvedba mora pri metodi črepinjenja s pomočjo metapodatkov pridobiti informacije o naslovu strežnika, ki hrani končne podatke.



Slika 31: Uporaba metode črepinjenja pri podatkih velikosti BigData.

Za lažjo predstavo si poglejmo primer, kjer želimo dostopati do podatkov s ključem vrednosti 777 [Slika 32].



Slika 32: Algoritem naslavljanja dokumenta.

V primeru naslavljanja vozlišča, ki hrani vrednost ključa 777, mora poizvedba pridobiti metapodatke iz prvonivojskega strežnika A. Metapodatki, shranjeni na tem strežniku, hranijo informacijo o naslovu drugonivojskega strežnika, ki hrani informacijo o končnem naslovu podatkov. Uporabnik mora za identifikacijo pravilnega strežnika venomer izvesti poizvedbo nad strežnikom prvega in drugega nivoja. Šele z metapodatki, pridobljenimi z drugega nivoja, lahko poizvedba pridobi želene podatke s strežnika C, ki se nahaja na tretjem nivoju. Za izboljšanje hitrosti naslavljanja podatkov lahko uporabnik hrani že pridobljene naslove s prvih dveh nivojev v predpomnilniku (*angl. Caching*). Slednje uporabniku omogoča izboljšanje hitrosti naslavljanja istih dokumentov.

6.3.4 Izbira ustreznega algoritma črepinjenja

Za namen utemeljitve izbire ustreznega algoritma bi morali analizirati učinkovitost posamezne nerelacijske podatkovne baze (npr. Riak in MongoDB), ki uporablja algoritem konsistentnega zgoščevalnega obroča in algoritem z določanjem obsega ključev ter na javni ponudnik storitev v oblaku shraniti velike količine podatkov. Ker smo bili za namen tovrstnih analiz omejeni s finančnimi sredstvi, lahko le teoretično opredelimo, delovanje katerega algoritma bi bilo ustrezno

za izbrano opravilo. Algoritem z definiranjem obsega ključa bi bil primeren, kadar bi naše poizvedbe večinoma poizvedovale nad zaporednimi obsegi ključev. Ker so ključi razvrščeni in urejeni po velikosti, lahko hitro naslavljamo strežnik z ustreznimi podatki.

Na drugi strani algoritem konsistentnega zgoščevalnega obroča nudi izjemno dobro porazdelitev podatkov v sistemu.

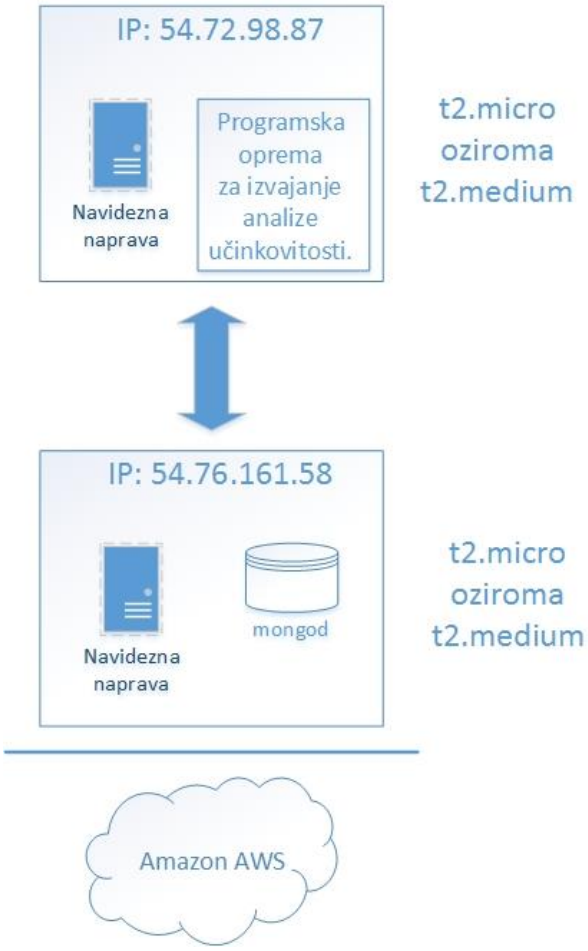
Izbira algoritma je močno odvisna od izbire produkta nerelacijske podatkovne baze, pri kateri se moramo predvsem vprašati, katero vrsto podatkov bomo shranjevali in katere operacije bodo prevladovale v sistemu (npr. operacija branja ali operacija pisanja). Šele po tovrstni opredelitvi bi lahko izbirali med različnimi ponudniki nerelacijskih podatkovnih baz in jih primerjali po učinkovitosti delovanja. V predlagani arhitekturi smo izbrali nerelacijsko podatkovno bazo MongoDB, ki uporablja algoritem črepinjenja na osnovi definiranja obsega ključev.

6.4 Analiza učinkovitosti

Pri izdelavi analize učinkovitosti smo generirali od 1000 do 10 milijonov naključnih dokumentov zapisa JSON, ki so predstavljali rezultat integracije med nerelacijskima podatkovnima bazama MongoDB in Neo4j. Na javnem ponudniku storitev v oblaku Amazon AWS smo na ločeni navidezni napravi izvajali programsko opremo, ki je pošiljala podatke na eno navidezno napravo oziroma lokalno instanco *mongos*, ki je z uporabo metode črepinjenja drobila podatke ter jih pošiljala na različni navidezni napravi. S tema topologijama smo zagotovili približno enake pogoje, saj so se podatki vedno prenašali preko omrežja. Za branje podatkov smo implementirali poizvedbo, v kateri smo želeli pridobiti število sporočil, ki so bila poslana iz izbrane telefonske številke.

6.4.1 Meritve vpliva vertikalne in horizontalne razširljivosti sistema

Za analizo vertikalne razširljivosti smo na javnem ponudniku storitev v oblaku Amazon AWS izbrali navidezni napravi tipa *t2.micro* oziroma *t2.medium* [Slika 33] z različno količino delovnega pomnilnika (1 GB oziroma 4 GB). Rezultati analize [Tabela 3] so merjeni v časovni enoti sekunda.

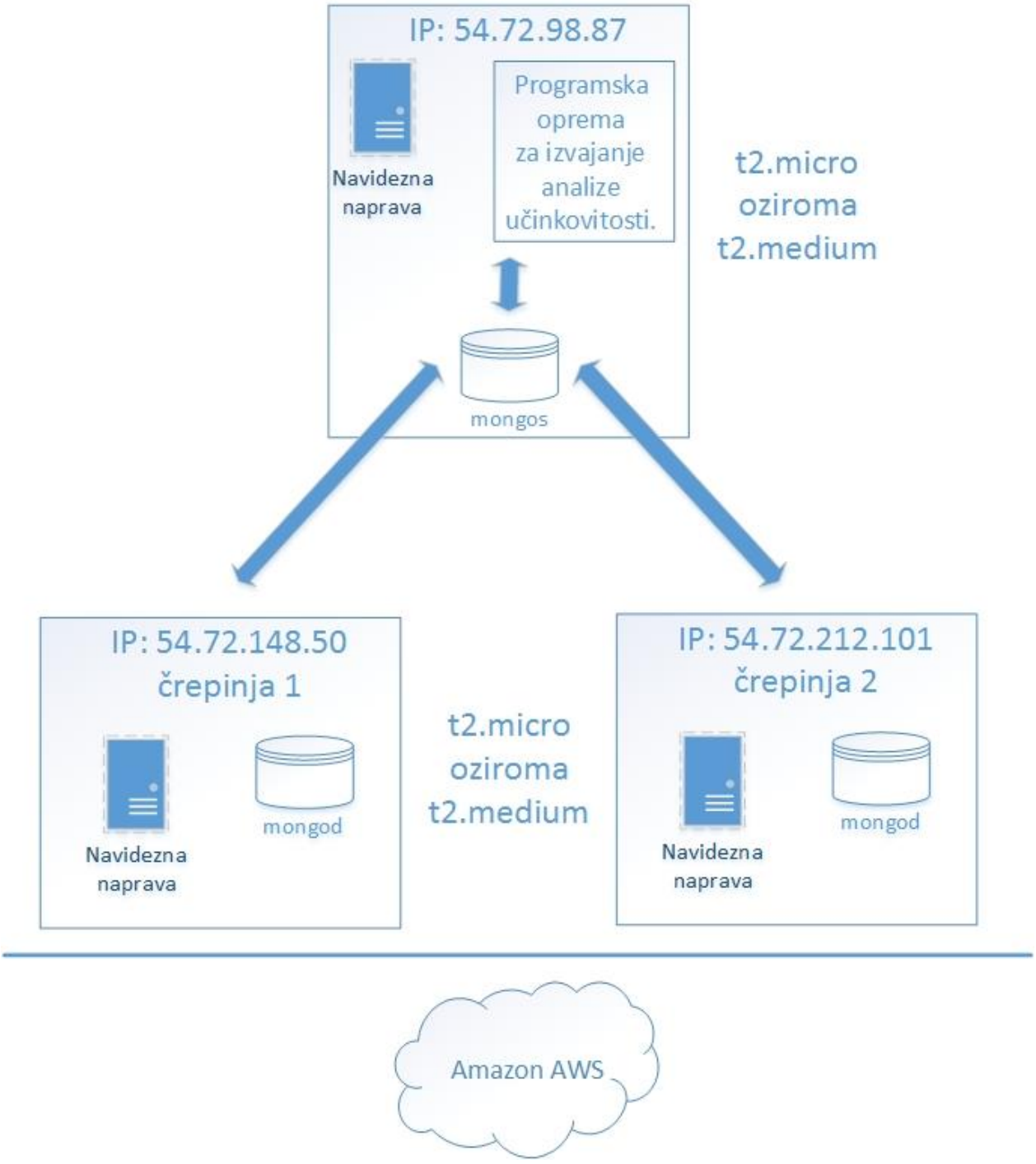


Slika 33: Topologija sistema, uporabljena pri analizi vertikalne razširljivosti.

	Vertikalna razširljivost (t2.micro)		*Vertikalna razširljivost (t2.medium)*	
Št. dokumentov	***Pisanje*** *(v sekundah)*	***Branje*** *(v sekundah)*	***Pisanje*** *(v sekundah)*	***Branje*** *(v sekundah)*
1 000	0.468	0.003	0.436	0.003
10 000	1.890	0.006	1.546	0.006
100 000	16.590	0.037	12.281	0.037
1 000 000	136.103	0.340	102.200	0.346
2 000 000	183.245	0.664	183.987	0.689
3 000 000	273.118	1.016	271.709	1.028
10 000 000	3319.625	6.229	3347.356	3.483

Tabela 3: Vertikalna razširljivost sistema.

Poleg vpliva vertikalne razširljivosti na eno instanco podatkovne baze smo naredili analizo vertikalnega vpliva na horizontalno porazdeljeni sistem dveh instanc podatkovne baze [Slika 34]. Za podporo tej analizi smo namestili sistem črepinjenja in tako kot v zgornjem primeru na javnem ponudniku storitev v oblaku izbrali navidezni napravi tipa *t2.micro* oziroma *t2.medium*. Za ključ črepinjenja smo izbrali izvleček (*angl. Hash)* in pridobili naslednje rezultate [Tabela 4], predstavljene v časovni enoti sekunda.



Slika 34: Topologija, uporabljena pri meritvi vertikalnega vpliva na horizontalno porazdeljeni sistem.

	Horizontalna razširljivost (t2.micro)		*Horizontalna razširljivost (t2.medium)*	
Št. dokumentov	***Pisanje*** *(v sekundah)*	***Branje*** *(v sekundah)*	***Pisanje*** *(v sekundah)*	***Branje*** *(v sekundah)*
1 000	1.129	0.009	1.078	0.008
10 000	10.881	0.009	10.474	0.009
100 000	111.278	0.015	106.900	0.015
1 000 000	1150.708	0.069	1104.822	0.071
2 000 000	2328.457	0.133	2383.244	0.129
3 000 000	3954.537	0.222	3422.579	0.204

Tabela 4: Vpliv vertikalne razširljivosti na horizontalno porazdeljeni sistem. Za ključ črepinjenja je bil uporabljen izvleček dokumenta.

Poleg vpliva razširljivosti sistema smo analizirali tudi vpliv zamenjave ključa, kjer smo na istem porazdeljenem sistemu uporabili ključ črepinjenja z definiranjem obsega [Tabela 5].

	Horizontalna razširljivost (t2.micro)		*Horizontalna razširljivost (t2.medium)*	
Št. dokumentov	***Pisanje*** *(v sekundah)*	***Branje*** *(v sekundah)*	***Pisanje*** *(v sekundah)*	***Branje*** *(v sekundah)*
1 000	0.842	0.007	0.142	0.005
10 000	1.311	0.008	1.056	0.008
100 000	10.307	0.019	10.112	0.020
1 000 000	537.044	0.087	441.835	0.094
2 000 000	1424.163	0.177	1408.724	0.181
3 000 000	2377.485	0.268	2435.96	0.347

Tabela 5: Vpliv vertikalne razširljivosti na horizontalno porazdeljeni sistem. Za ključ črepinjenja smo uporabili ključ z določanjem obsega.

6.4.2 Interpretacija rezultatov

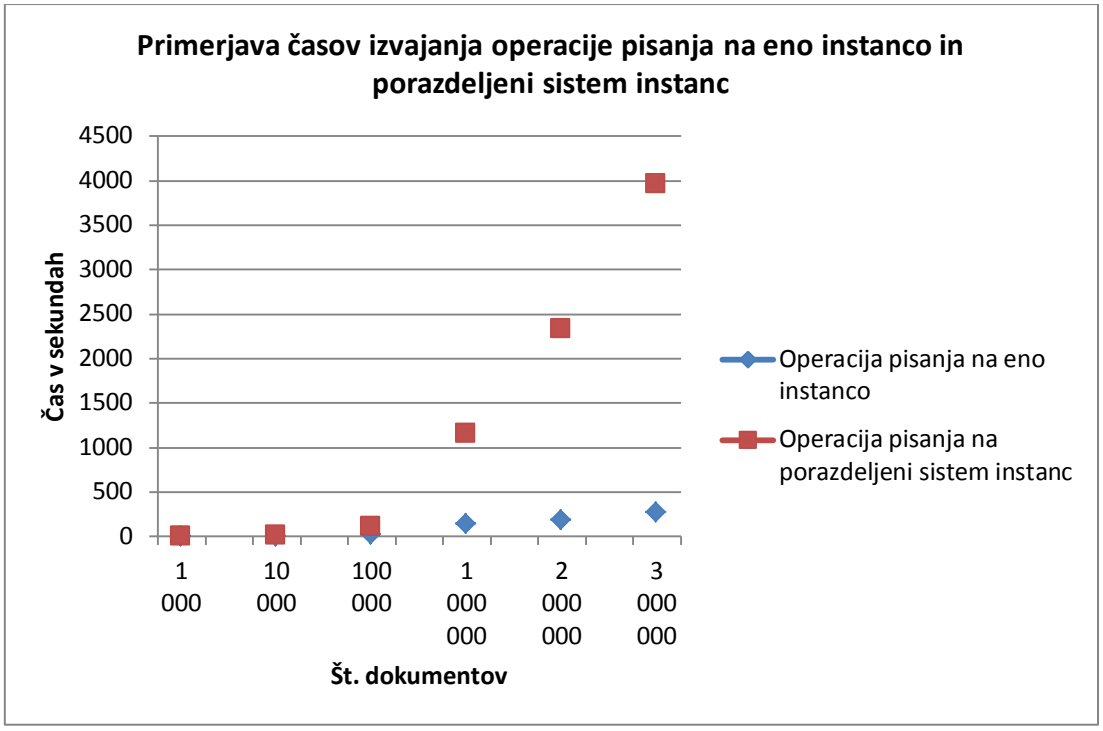
6.4.2.1 Analiza rezultatov vertikalne razširljivosti sistema

Če se osredotočimo na prvi del analize, kjer smo iz oddaljene navidezne naprave shranjevali podatke na eno navidezno napravo z različno količino delovnega pomnilnika, lahko iz tabel razberemo [Tabela 3], da so nekateri časi boljši pri napravi z manj delovnega pomnilnika, drugi pa pri napravi z več delovnega pomnilnika. To je posledica omrežne zakasnitve pri komunikaciji med navideznima napravama z nameščeno programsko opremo ter instanco podatkovne baze. Verjamemo, da bi z odstranitvijo omrežne zakasnitve in shranjevanjem večje količine podatkov prišli do razlik v prid napravi z več pomnilnika.

6.4.2.2 Analiza rezultatov pri izvajanju operacij na eno instanco podatkovne baze oziroma porazdeljeni sistem instanc

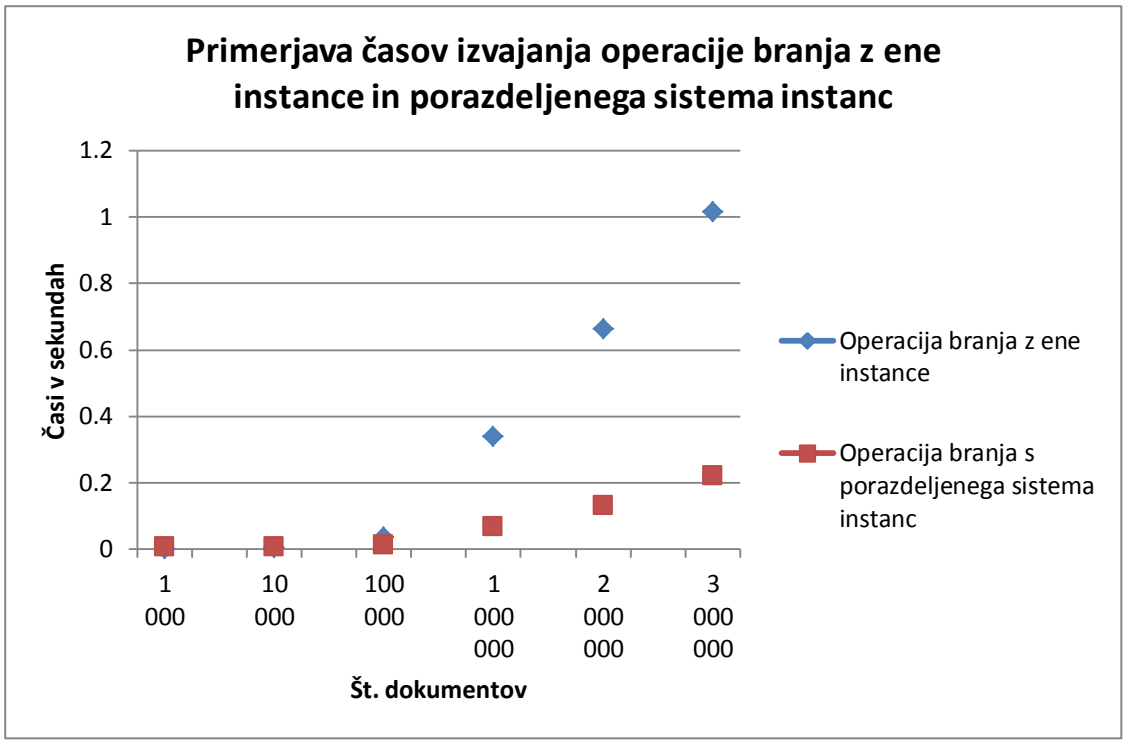
Pri primerjanju časov izvajanja operacije pisanja na eno instanco podatkovne baze s časi na porazdeljeni sistem instanc opazimo občutno počasnejše izvajanje operacije pisanja na porazdeljenem sistemu instanc [Slika 35]. Z uporabo metode črepinjenja in usmerjanjem podatkov na ustrezno instanco podatkovne baze se povečajo časi izvajanja operacije pisanja. Pri

obeh topologijah, torej shranjevanju podatkov na eno instanco oziroma porazdeljeni sistem instanc, smo zagotovili, da so se podatki prenašali preko interneta.



Slika 35: Primerjava časov izvajanja operacije pisanja na eno instanco in porazdeljeni sistem instanc.

V nasprotju z operacijo pisanja se časi operacije branja pri uporabi porazdeljenega sistema zmanjšajo v primerjavi s časi izvajanja operacije branja na eni instanci podatkovne baze [Slika 36].



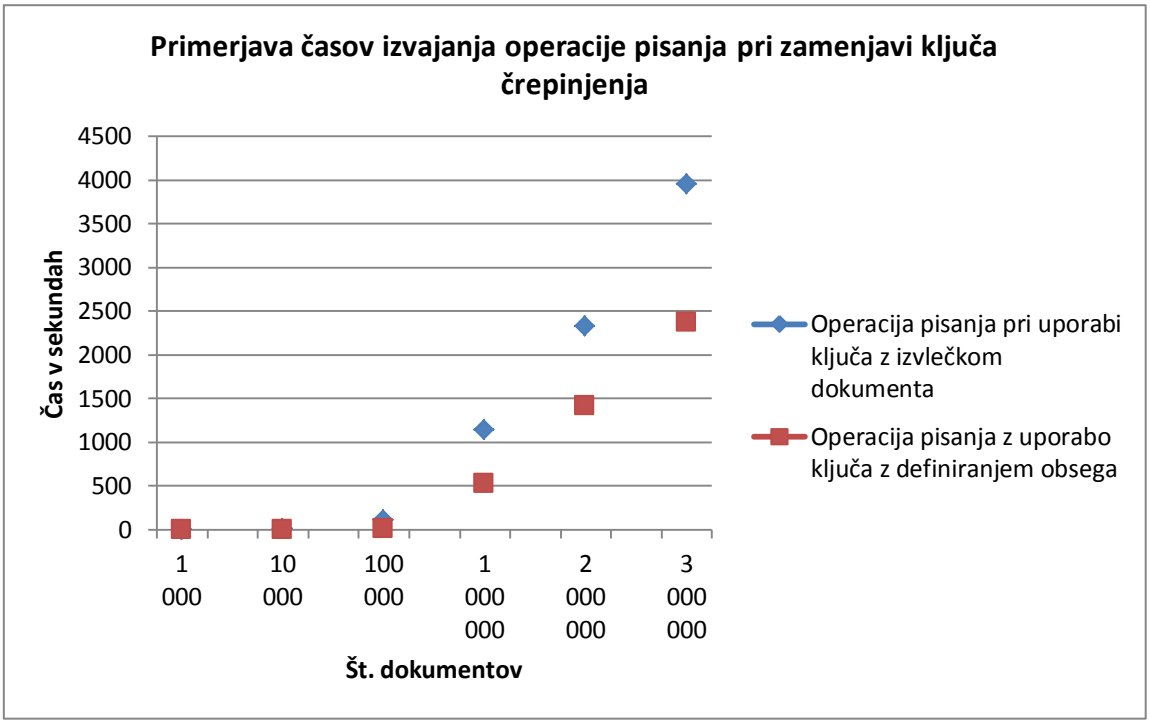
Slika 36: Primerjava časov izvajanja operacije branja z ene instance in porazdeljenega sistema instanc.

6.4.2.3 Analiza rezultatov vpliva vertikalne razširljivosti na horizontalno porazdeljeni sistem

Zaradi majhne količine podatkov, ki smo jo shranjevali na porazdeljeni sistem instanc, vertikalna razširitev ni očitno pripomogla k izboljšanju časov. Tako kot pri prvi analizi [6.4.2.1 Analiza rezultatov vertikalne razširljivosti sistema], so bili rezultati pri shranjevanju 3 milijonov dokumentov [Tabela 5] boljši pri uporabi navidezne naprave tipa *t2.micro*. Odstopanje se pojavi zaradi omrežne zakasnitve pri uporabi navideznih naprav na javnem ponudniku storitev v oblaku Amazon AWS. Verjamemo, da bi v primeru shranjevanja večje količine podatkov ter zmanjšane omrežne zakasnitve, vertikalna razširljivost pripomogla k izboljšanju časa.

6.4.2.4 Analiza rezultatov pri zamenjavi ključa črepinjenja

Pri primerjavi časa izvajanja operacije pisanja pri zamenjavi ključa črepinjenja opazimo, da smo z zamenjavo ključa, torej z izbiro ključa z definiranim obsegom, izboljšali čase [Slika 37]. Pri isti analizi se pri izvajanju operacije branja časi niso bistveno razlikovali.

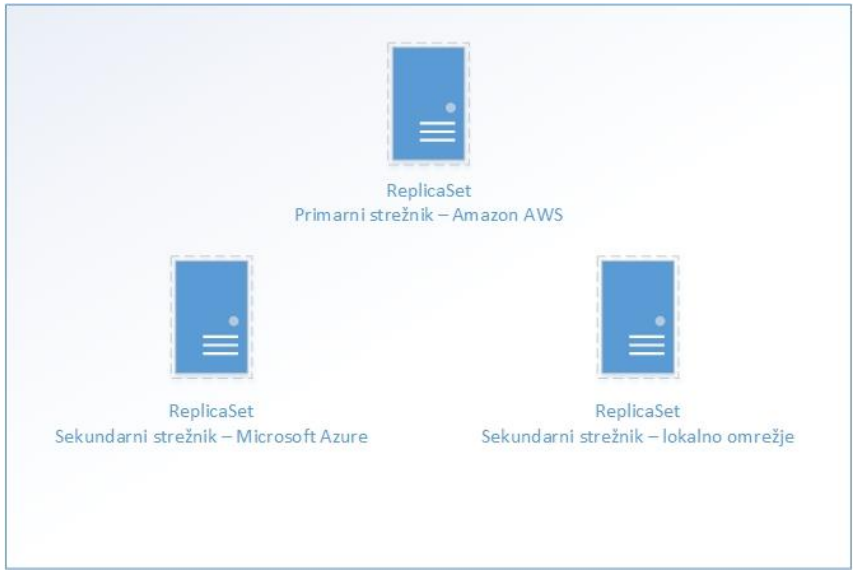


Slika 37: Primerjava časov pri zamenjavi ključa črepinjenja.

Pri izdelavi analiz vertikalna razširljivost ni bistveno pripomogla k hitrejšemu delovanju sistema. Verjamemo, da bi vertikalna razširljivost bila izrazitejša s shranjevanjem večje količine podatkov. Tega nam zaradi finančnih omejitev ni uspelo izvesti. Za nerelacijsko podatkovno bazo MongoDB smo zasledili napotke, da naj bi z vertikalno razširitvijo razširili sistem do količine 250 GB podatkov [51]. V primeru dosega te omejitve bi nadaljevali z metodo črepinjenja, pri čemer bi bilo smiselno črepinje namestiti na replikacijo sistema, kot je to prikazano v predlagani arhitekturi [Slika 21]. Replikacija sistema omogoča visoko razpoložljive navidezne naprave in porazdelitev obremenitev. Poleg namestitve porazdeljenega sistema bi morali veliko pozornosti nameniti izbiri ustreznega ključa črepinjenja ter indeksov nad tistimi polji, nad katerimi izvajamo poizvedbe. Pri količini 2,5 TB podatkov naj bi vsako črepinjo podprli z izolirano gručo replikacije, pri čemer bi delovanje metode črepinjenja ostalo nespremenjeno. Izolirana gruča replikacije pomeni namestitev vsake instance nerelacijske podatkovne baze na ločeno navidezno napravo.

6.5 Replikacija sistema

Za zagotavljanje visoke razpoložljivosti sistema je replikacija ključnega pomena. V predlagani arhitekturi [Slika 21] smo replikacijo sistema vključili za podporo strežnikov/črepinj [Slika 38]. Navidezne naprave, ki so združene v logično celoto in predstavljajo replikacijo sistema, smo namestili na različne ponudnike storitev v oblaku ter lokalno omrežje.



Slika 38: Replikacija sistema.

6.5.1 Osnove replikacije sistema

S sistemom replikacije smo želeli zagotoviti naslednje lastnosti:

- razširljivost,
- trajnost/zanesljivost ter
- izoliranost sistema.

6.5.1.1 Razširljivost

Z razširljivostjo sistema izboljšamo naslednji dve lastnosti:

- učinkovitost/odzivnost: ko namestimo podatkovno bazo na gručo navideznih naprav, s tem omogočimo paralelno delovanje sistema in
- redundanco sistema: ta zagotavlja, da se kopija dokumenta nahaja na različnih napravah ali podatkovnih centrih. To pomeni, da lahko uporabnike preusmerimo na tiste instance strežnikov, ki so na dani lokaciji najbližji. S tem zmanjšamo zakasnitve dostopa do sistema.

6.5.1.2 Trajnost/zanesljivost

Metoda replikacije se običajno uporablja za izognitev težavam, ki nastanejo zaradi sistemskih napak. Še posebej pa so priporočljive za uporabo v naslednjih primerih:

- kadar želimo imeti duplikat podatkovne baze ter
- kadar imamo veliko količino podatkov, ki se neprestano spreminjajo. V tem primeru običajna varnostna kopija ni uporabna.

6.5.1.3 Izoliranost sistema

V svetu podatkovnih baz obstajajo procesi, ki drastično znižujejo učinkovitost sistema. V teh primerih želimo imeti sinhronizirano kopijo sistema in tovrstne procese izvajati na njej. Kot primer lahko navedemo:

- ustvarjanje poročil ali varnostnih kopij: te se lahko izvajajo na strežnikih, ki so predstavljeni kot sekundarni strežniki sistema (*angl. Slaves*) ter
- dostopanje programerjev za namen razvoja programske opreme do osveženih podatkov. Za izognitev težavam v primeru napak v programski kodi je bolje omogočiti dostop do sinhroniziranih izoliranih sistemov

6.5.2 Različni načini sistema replikacije na podatkovni bazi MongoDB

Za podporo sistemu replikacije poznamo različne arhitekturne pristope [52]**,** pri čemer smo pri izdelavi magistrske naloge izbrali najnaprednejšo, imenovano *ReplicaSet.* V nerelacijski podatkovni bazi MongoDB poznamo tudi druge namestitvene topologije replikacije:

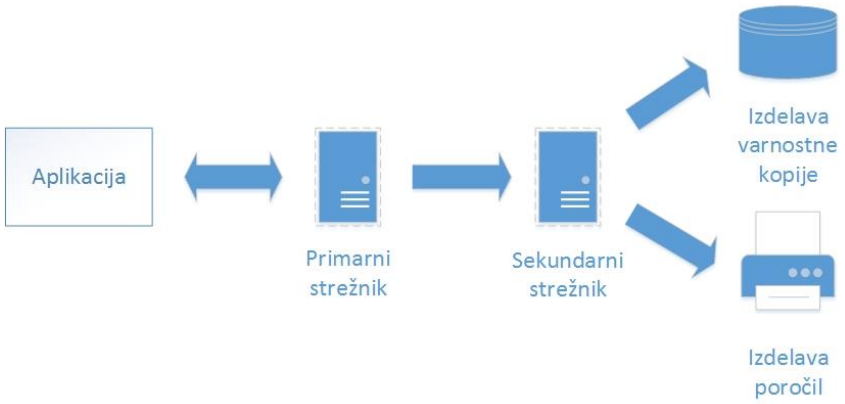
- ena primarna/ena sekundarna instanca (*angl. Single master/single slave replication*),
- ena primarna/več sekundarnih instanc (*angl. Single master/multiple slave replication*),
- več primarnih/ena sekundarna instanca (*angl. multiple master/single slave replication)*,
- replikacija Cascade (*angl. Cascade replication*),
- prepletena replikacija (*angl. Interleaved replication*) in
- replikacija *ReplicaSet*.

Sistem replikacije se je skozi čas močno spreminjal, zato lahko trenutno za zagotavljanje visoke razpoložljivosti sistema uporabljamo zelo napredno tehnologijo imenovano *ReplicaSet.* Za pravilno delovanje sistema *ReplicaSet* MongoDB uporablja OpLog [53]. Ta zagotavlja sinhronizacijo med vsemi strežniki v sistemu. Vse instance podatkovnih arhitektur, ki delujejo v načinu primarnih strežnikov za namen sinhronizacije s sekundarnimi strežniki, upravljajo s sistemom OpLog. Sekundarni strežniki s pomočjo poizvedb pridobijo osvežene podatke. OpLog

za svoje delovanje uporablja časovne značke, ki zagotavljajo, da bo sistem v primeru izpada preveril stanje. Ob tem OpLog uporablja lastnost časovnega okna, kar pomeni, da se čas hranjenja sprememb sistema vedno spreminja. Sinhronizacija sistema se odvija s pomočjo asinhronih funkcij.

6.5.2.1 Topologija ena primarna/ena sekundarna instanca

Ta način je najenostavnejši in ne omogoča samodejnega odziva na napake. Aplikacija v tem primeru komunicira s primarnim strežnikom, medtem ko je sekundarni strežnik uporabljen za izdelavo poročil in varnostnih kopij [Slika 39].



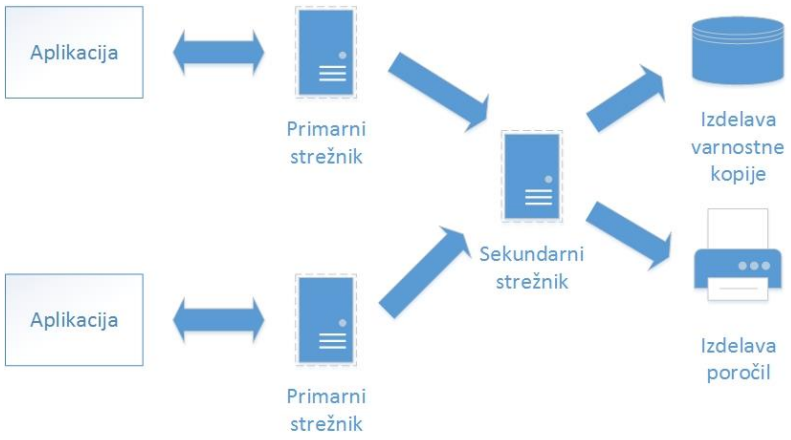
Slika 39 Topologija primarne in sekundarne instance strežnika.

6.5.2.2 Topologija ena primarna/več sekundarnih instanc

Za podporo večji količini izdelave poročil lahko namestimo več sekundarnih instanc, ki omogočajo paralelno izdelavo poročil in varnostnih kopij.

6.5.2.3 Topologija več primarnih/ena sekundarna instanca

Slednji način uporabljamo v primeru, ko sistem gostuje več aplikacij ter vsaka aplikacija uporablja svojo primarno instanco podatkovne arhitekture [Slika 40]. V tem primeru se sekundarna instanca uporablja za namen varnostne kopije in poročil celotnega sistema. Za ta način se lahko odločimo, kadar proces izdelave varnostne kopije in poročila ne presega zmogljivosti sekundarne instance.



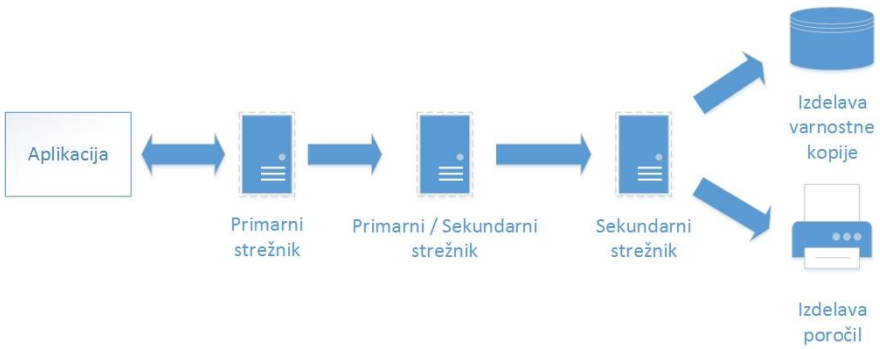
Slika 40: Topologija več primarnih in ene sekundarne instance strežnika.

6.5.2.4 Topologija Cascade

Ta topologija sistema replikacije zahteva tri strežnike:

- primarni strežnik,
- ½ primarni in ½ sekundarni in
- sekundarni strežnik.

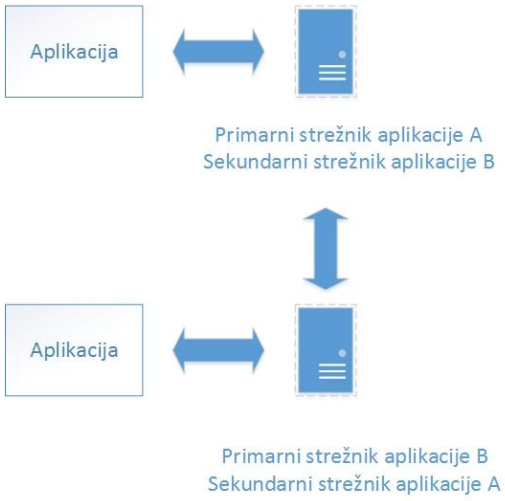
Vsi strežniki so povezani v verigo in skupaj omogočajo, da vmesni strežnik prejme kopijo primarne podatkovne baze v realnem času, medtem ko sekundarni strežnik prejme kopijo v zapoznelem času [Slika 41]. Vmesni strežnik je predstavljen kot sekundarni strežnik za prvi primarni strežnik ter hkrati primarni strežnik za drugi sekundarni strežnik.



Slika 41: Topologija cascade.

6.5.2.5 Topologija prepletene replikacije

Ta način uporabljamo, kadar si več aplikacij deli isto topologijo podatkovne baze. Tako se prvi aplikaciji strežnik predstavi kot primarni, medtem ko se drugi aplikaciji predstavi kot sekundarni strežnik [Slika 42].



Slika 42: Prepletena replikacija.

6.5.2.6 Topologija ReplicaSet

Topologija *ReplicaSet* je sodobna topologija, uporabljena v nerelacijski podatkovni bazi MongoDB, ki se uporablja na področju sistema replikacije. Za uspešno namestitev sistema moramo določiti tako primarne kot sekundarne strežnike, katerih naloga je skrb za pravilno replikacijo sistema. Poleg primarnih in sekundarnih strežnikov moramo izbrati tudi aktivne in pasivne strežnike. Strežniki, ki bodo opredeljeni za pasivne, nikoli ne bodo postali primarni strežniki sistema. Ta lastnost ima odlično prednost, saj sistemu ne dovolimo, da bi za primarni strežnik izbral tistega, ki ni dovolj zmogljiv za preračunavanje in strežbo vseh zahtevkov. Pasivni strežniki so primerni za izdelave varnostnih kopij in poročil. V omejitev, da bi skrbniki sistema namestili topologijo *ReplicaSet* na eno samo instanco, je tej onemogočeno, da bi podpirala lokalni naslov (*angl. Localhost*). Pred začetkom namestitve moramo zagotoviti vidnost vseh instanc, ki bodo vključene v sistem replikacije.

Replikacija sistema izboljšuje:

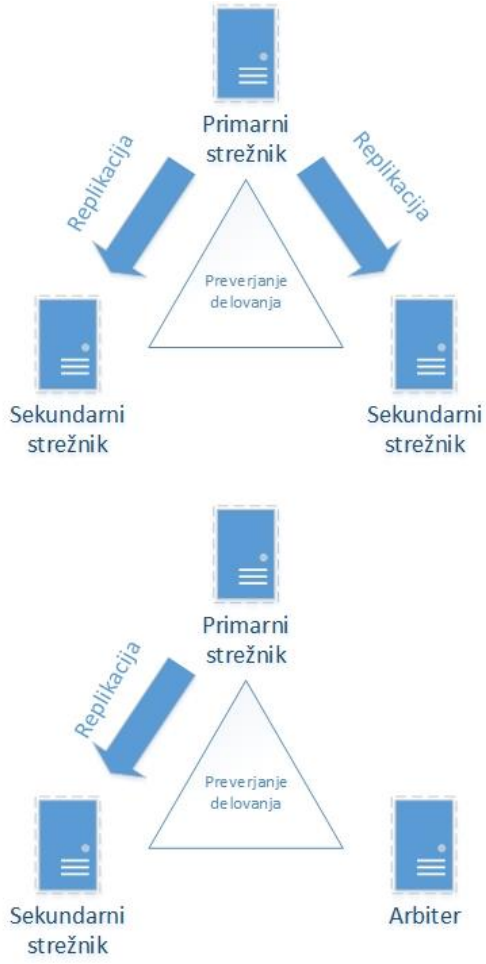
- zmogljivost sistema in
- redundančnost sistema.

Zmogljivost sistema

V aplikaciji lahko podpremo paralelno branje podatkov. Paralelni način poteka simultano na vseh instancah v sistemu. S tem občutno povečamo zmogljivost sistema ob branju podatkov.

Redundančnost sistema

Redundančnost zagotavlja sinhronizacijo instanc v različnih podatkovnih centrih med seboj. To je zagotovljeno z lastnostjo, da primarne instance sistema prejmejo vse zahtevke, ki jih pozneje sinhronizirajo s svojimi sekundarnimi strežniki. V primeru izpada sistema se zažene t. i. proces volitev, kjer se določi primarni strežnik sistema. Eno izmed instanc podatkovne baze lahko določimo za arbitra volitev [Slika 43], ki odloča o rezultatu izbire novega primarnega strežnika. Glavna razlika med primarnim in sekundarnim strežnikom je, da uporabniku ni dovoljeno pisati na sekundarni strežnik sistema. Vsak aktivni sekundarni strežnik v primeru izpada primarnega lahko prevzame vlogo vodilnega strežnika v sistemu.



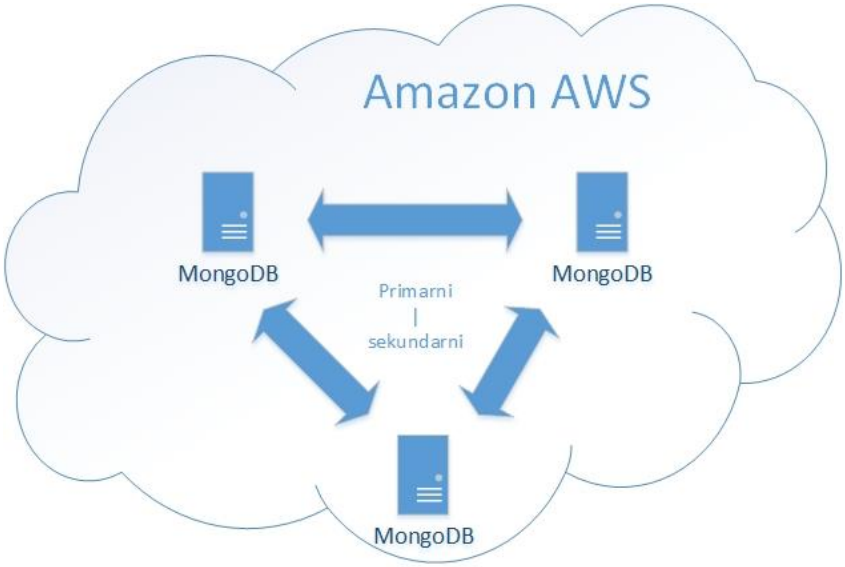
Slika 43: Primer delovanja sistema replikacije v načinu brez in z arbitrom.

6.6 Namestitev replikacije sistema na različne ponudnike storitev v oblaku

Replikacija sistema z uporabo *ReplicaSet* za svoje delovanje uporablja gručo navideznih naprav. Pri uporabi sistema *ReplicaSet* nas je zanimalo, če lahko z uporabo različnih ponudnikov storitev v oblaku ter lokalnim omrežjem brez večjih težav vzpostavimo replikacijo sistema. Za ta namen smo uporabili navidezne naprave na javnem ponudniku storitev v oblaku Amazon AWS in Microsoft Azure.

6.6.1 Primer namestitve instanc na oblačno storitev Amazon AWS

Na javnem ponudniku storitev v oblaku Amazon AWS smo ustvarili tri instance in s pomočjo pravil v požarnem zidu (*angl. Firewall*) zagotovili vidnost med njimi. Na te instance smo namestili topologijo replikacije *ReplicaSet* [Slika 44].



Slika 44: Prikazan je primer topologije, ki je bil nameščen izključno na ponudniku storitev v oblaku Amazon AWS.

Za namestitev replikacije sistema moramo slediti naslednjemu postopku:

- na vsako navidezno napravo namestimo nerelacijsko podatkovno bazo MongoDB in ji dodelimo ime gostitelja (*angl. Hostname*),
- datoteko */etc/hosts* spremenimo tako, da imena ostalih instanc sovpadajo z omrežnimi naslovi IP,

- za namen shranjevanja podatkov na datotečnem sistemu ustvarimo mapo (*mkdir –p /db/active/data*),
- za uspešno komunikacijo med instancami moramo na javnem ponudniku storitev v oblaku Amazon AWS ustvariti varnostno skupino (*angl. Security Group*), ki omogoči komunikacijo med instancami,
- po uspešni dosegljivosti instanc moramo vsako izmed njih z naslednjimi ukazi dodati v sistem *ReplicaSet:*

```
./mongod –dbpath /db/active1/data –port 27021 –replSet magset/VM-I2-S2:27022 –rest
#Po zagonu prve instance se povežemo nanjo.
./mongo VM-I1-S1:27021
#Spremenimo nastavitve replicaSet.
cfg = {
      _id : 'magset'
      members : [
            { _id:0, host:'VM-I1-S1:27021' },
            { _id:1, host:'VM-I2-S2:27022' },
            { _id:2, host:'VM-I3-S3:27023' }
            ]
      }
rs.initiate(cfg) //Inicializacija nastavitve.
rs.status() //Preverimo stanje ReplicaSet sistema.
rs.stepDown() //Simuliramo izpad primarne instance.
```

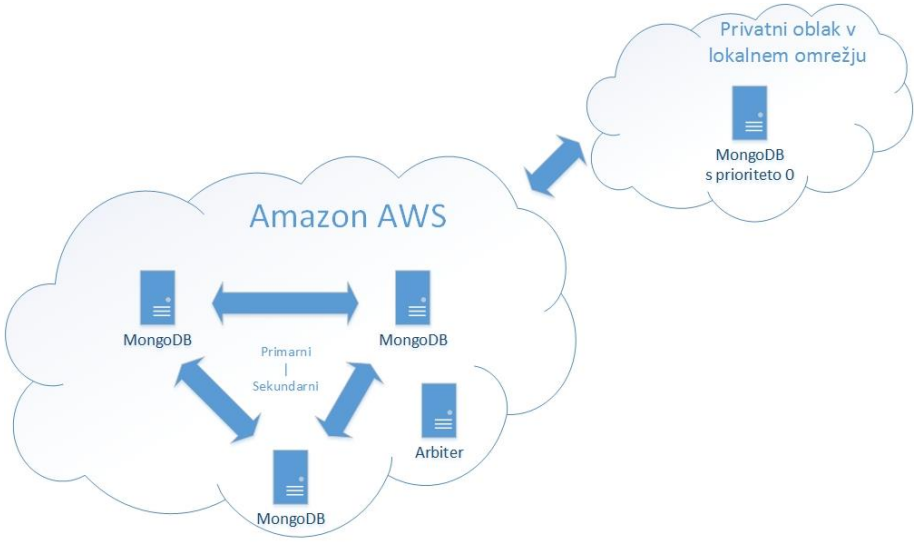
Za testiranje delovanja replikacije sistema smo z uporabo ukaza simulirali izpad primarne navidezne naprave sistema [Slika 45]. V tem primeru je sistem samodejno zagotovil zamenjavo primarne instance sistema.

```
connecting to: VM-I1-S1:27021/test
magset:PRIMARY> show collections
system.indexes
test
magset:PRIMARY> db.test.find()
{ "_id" : ObjectId("5271195e4701cf16182061c8"), "item" : "card", "qty" : 15 }
magset:PRIMARY> db.test.insert({"item":"avto", "qty":1})
magset:PRIMARY> db.test.find()
{ "_id" : ObjectId("5271195e4701cf16182061c8"), "item" : "card", "qty" : 15 }
{ "_id" : ObjectId("52711cddeb4fba002826f8ae"), "item" : "avto", "qty" : 1 }
magset:PRIMARY> rs.stepDown()
Wed Oct 30 14:51:25.905 DBClientCursor::init call() failed
Wed Oct 30 14:51:25.906 Error: error doing query: failed at src/mongo/shell/query.js:78
Wed Oct 30 14:51:25.906 trying reconnect to VM-I1-S1:27021
Wed Oct 30 14:51:25.907 reconnect VM-I1-S1:27021 ok
magset:SECONDARY> exit
bye
root@VM-I1-S1:/download/mongodb-linux-x86_64-2.4.7/bin# ./mongo VM-I3-S3:27023
MongoDB shell version: 2.4.7
connecting to: VM-I3-S3:27023/test
magset:PRIMARY> db.test.find()
{ "_id" : ObjectId("5271195e4701cf16182061c8"), "item" : "card", "qty" : 15 }
{ "_id" : ObjectId("52711cddeb4fba002826f8ae"), "item" : "avto", "qty" : 1 }
magset:PRIMARY>
```

Slika 45: Rezultat namestitve sistema *ReplicaSet* in simuliranje izpada primarnega strežnika.

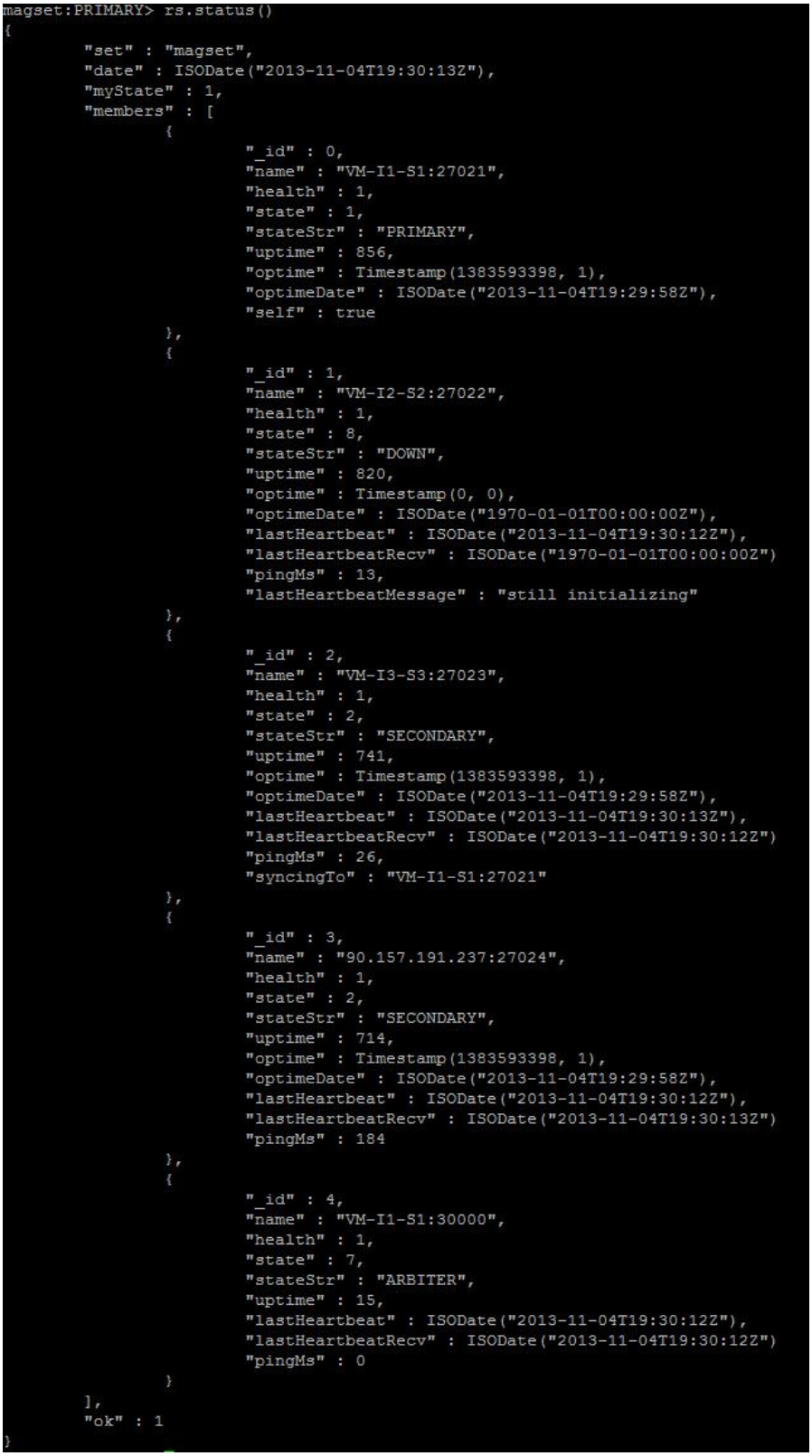
6.6.2 Primer namestitve instanc na oblačno storitev Amazon AWS in lokalno omrežje

Topologija *ReplicaSet* nas ne omejuje pri namestitvi navideznih naprav na javni ponudnik storitev v oblaku oziroma v lokalno omrežje [Slika 46].



Slika 46: Prikazan je primer topologije ločenih instanc nerelacijske podatkovne baze na storitvi v oblaku Amazon AWS ter lokalnim omrežjem.

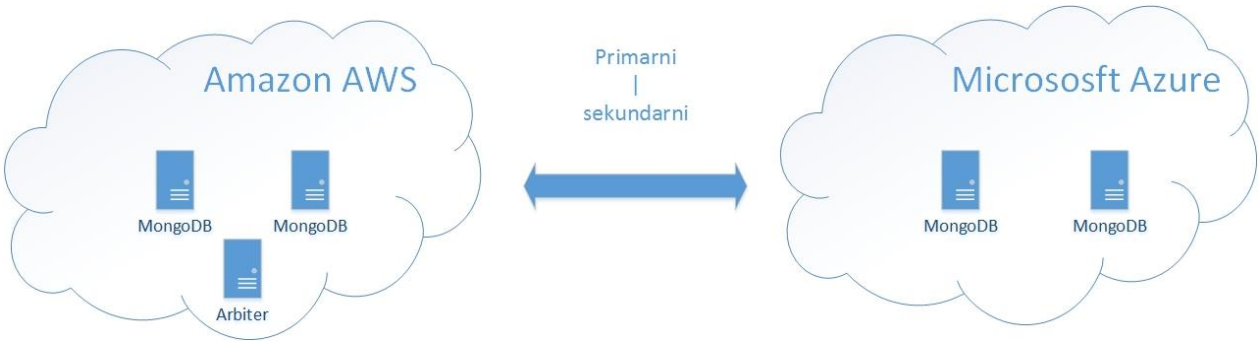
V tem načinu smo namestili dodatno instanco v lokalnem omrežju in jo kot sekundarni strežnik povezali v sistem *ReplicaSet* [Slika 47]. Z dodatnim strežnikom v lokalnem omrežju je bilo v sistemu *RepliciSet* povezanih sodo število strežnikov. Slednje je predstavljalo težavo pri volitvah, zato smo morali dodatno namestiti arbitra volitev. Za uspešno vključitev lokalne navidezne naprave v topologijo *ReplicaSet* smo vsem navideznim napravam na javnem ponudniku storitev v oblaku Amazon AWS dodelili elastični naslov IP (*angl. Elastic IP Address*) in z nastavitvijo požarnega zidu zagotovili vidnost med instancami. Za zagotavljanje varnosti sistema lahko na navidezne naprave namestimo programje za vzpostavitev navideznega zasebnega omrežja (*angl. Virtual Private Network*) (npr. OpenVPN, LogMeIn Hamachi, Shrew Soft), ki zagotavlja izoliranost, šifriranje, preverjanje pristnosti (*angl. Authentication*) ter avtorizacijo navideznih naprav.



Slika 47: Rezultat namestitve instance na javnem ponudniku storitev v oblaku Amazon AWS in lokalnemu omrežju.

6.6.3 Primer namestitve instanc na dva različna ponudnika oblačnih storitev (Amazon AWS in Microsoft Azure)

Za zagotavljanje visoke dostopnosti sistema smo analizirali primer namestitve topologije *ReplicaSet* na dva ponudnika storitev v oblaku Amazon AWS in Microsoft Azure [Slika 48]. Za njun izbor smo se odločili, saj sodita med popularnejše ponudnike teh storitev.



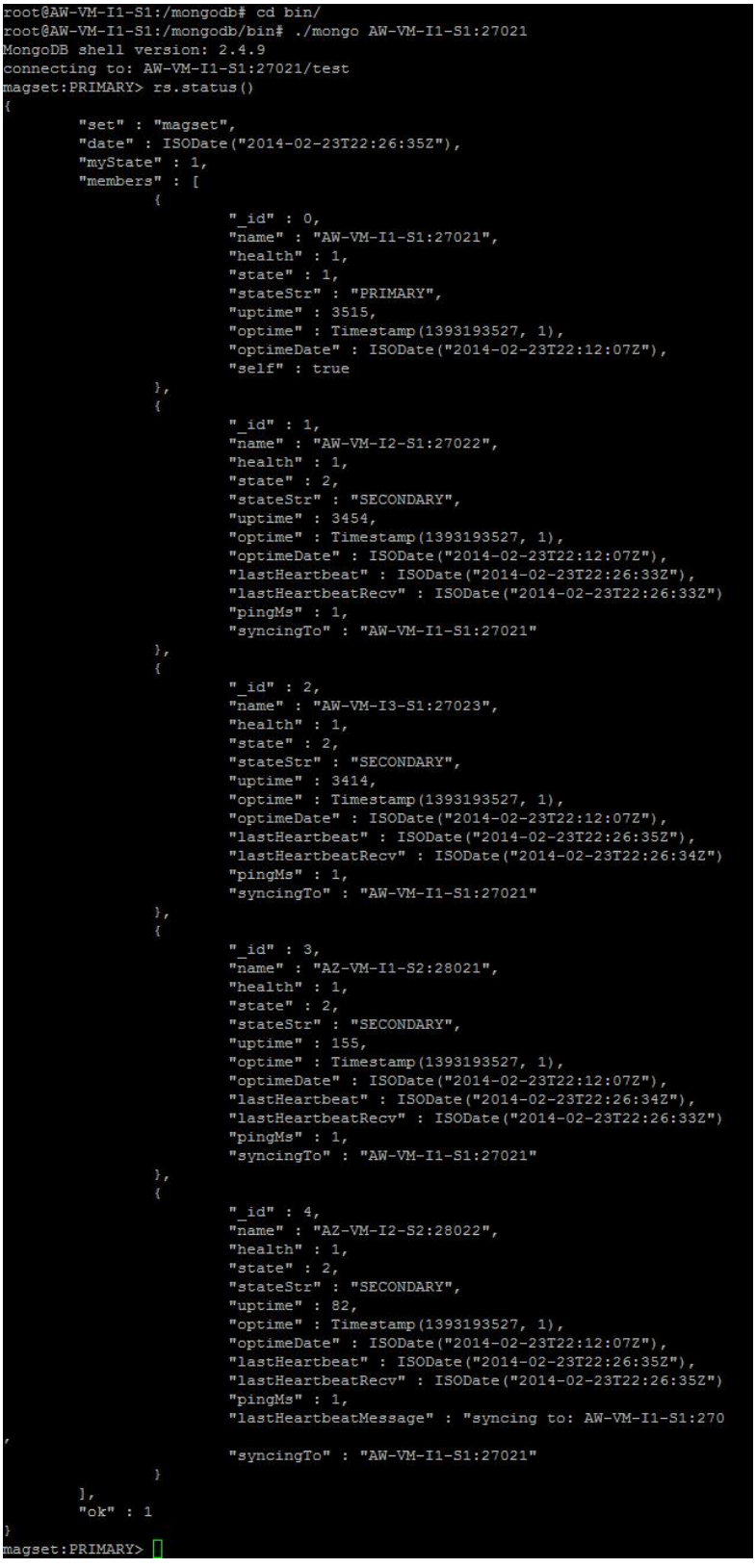
Slika 48: Prikazan je primer topologije, kjer smo instance namestili na dva različna ponudnika storitev v oblaku (Amazon AWS in Microsoft Azure).

Za uspešno delovanje smo morali na navidezne naprave namestiti instanco podatkovne baze MongoDB in zagotoviti povezljivost med navideznimi napravami [Slika 49]. Povezljivost instanc podatkovne baze MongoDB smo zagotovili s pomočjo naslednjih nastavitev.

```
cfg = {
_id: 'magset',
members: [
      { _id: 0, host: 'AW-VM-I1-S1:27021' },
      { _id: 1, host: 'AW-VM-I2-S1:27022' },
      { _id: 2, host: 'AW-VM-I3-S1:27023' }
]
}
```

Na navideznih napravah gostujočih na javnem ponudniku storitev v oblaku Microsoft Azure smo s spodnjimi ukazi instance vključili v sistem ReplicaSet.

```
AZ-VM-I1-S2: ./mongod –dbpath /db/active4/data –port 28021 –replSet
magset/AW-VM-I1-S1:27021 –rest –smallfiles
AZ-VM-I2-S2: ./mongod –dbpath /db/active5/data –port 28022 –replSet
magset/AW-VM-I1-S1:27021 –rest –smallfiles

#Na primarni instanci smo dodali dve novi instanci v obstoječo
topologijo ReplicaSet:
rs.add("AZ-VM-I1-S2:28021")
rs.add("AZ-VM-I2-S2:28022")
```



Slika 49: Prikaz statusa sistema *ReplicaSet* po uspešni vključitvi vseh instanc v ta sistem.

6.7 Ugotovitve

Visoka razpoložljivost sistema je zelo pomembna lastnost, ki jo moramo zagotoviti pri implementaciji aplikacij. Za ta namen smo visoko razpoložljivost v predlagani arhitekturi zagotovili na dveh nivojih, t.j. nivoju navideznih naprav ter podatkovnem nivoju. Na nivoju navideznih naprav smo namestili topologijo replikacije sistema na različne ponudnike storitev v oblaku ter lokalno omrežje. Pri namestitvi topologije nismo imeli večjih težav, saj je bilo za uspešno delovanje navideznih naprav potrebno zagotoviti le vidnost med navideznimi napravami. To smo storili predvsem s pravilno nastavitvijo požarnega zidu.

Drugi nivo, ki zagotavlja visoko razpoložljivost sistema, je predstavljen na podatkovnem nivoju. Na tem nivoju namestitev in nastavitev visoko razpoložljivega sistema zahtevata poglobljeno znanje o problemski domeni. Visoko razpoložljivost na podatkovnem nivoju smo zagotovili z uporabo metode črepinjenja. Ta metoda z uporabo enega izmed navedenih algoritmov (algoritem zgoščevalni obroč ali algoritem z definiranjem obsega ključev) drobi podatke na porazdeljen sistem naprav. Kot smo omenili, nerelacijska podatkovna baza MongoDB pri svojem delovanju uporablja algoritem z definiranjem obsega ključev. Ključno vprašanje za učinkovito delovanje tega algoritma je predvsem pravilna izbira ključa, po katerem bo izbrani algoritem drobil podatke. Nepravilno izbiro ključa lahko enačimo z nepravilnim načrtovanjem podatkovne sheme v relacijskih podatkovnih bazah, saj se lahko srečamo s problemom, da podatkov zaradi napačne izbire ključa ne moremo več drobiti. V tem primeru bi morali ponovno definirati ključe črepinjenja ali pa metodo črepinjenja implementirati na aplikacijskem nivoju, kar bi predstavljalo veliko kompleksnost. Najboljše prakse nam narekujejo, da je za učinkovito delovanje metode črepinjenja potrebno izbrati sestavljeni ključ v kombinaciji z najpogostejšimi iskalnimi polji ter časovno značko shranjevanja podatkov [48].

Tako horizontalna kot vertikalna razširljivost sistema sta uporabljeni za implementacijo visoko razpoložljive arhitekture. Vertikalno razširljivost sistema lahko v povezavi z računalništvom v oblaku implementiramo z izbiro zmogljivejših naprav. V ta namen javni ponudnik Amazon AWS deli navidezne naprave v več skupin, s poudarkom na procesorski moči, diskovni kapaciteti ali delovnem pomnilniku. V nasprotju s tem pa horizontalna razširljivost sistema omogoča izboljšanje zmogljivosti z dodajanjem novih navideznih naprav. Drobljenje podatkov je tista metoda na podatkovnem nivoju, ki zagotavlja učinkovito uporabo horizontalno razširljivega sistema.

Za testiranje in uporabo predlagane arhitekture [Slika 21] ter analize učinkovitosti smo v programskem jeziku Python simulirali rezultate integracije in jih shranili na eno navidezno napravo oziroma porazdeljeni sistem črepinj. Za učinkovito upravljanje z večjimi količinami

podatkov (npr. rezultati analiz *Next Generation Sequences*), bi morali z vertikalno razširljivostjo sistema podpreti horizontalno porazdeljeni sistem.

7. Predstavitev implementacije

Praktični del magistrske naloge zajema implementacije pretvorbe dokumentov za namen integracije nerelacijskih podatkovnih baz z uporabo semantičnega spleta, namestitev arhitekture na porazdeljeni sistem navideznih naprav, gostujočih na dveh javnih ponudnikih računalništva v oblaku ter lokalnemu omrežju ter implementacijo funkcije za testiranje vpliva vertikalne oziroma horizontalne razširljivosti na učinkovitost shranjevanja in branja podatkov. Implementacijo in namestitev nekaterih delov arhitekture smo prikazali v teoretičnem delu, kjer smo s teorijo podprli motivacijo za implementacijo te arhitekture. Dele predlagane arhitekture [Slika 21] smo namestili na javni ponudnik storitev v oblaku Amazon AWS [Slika 50].

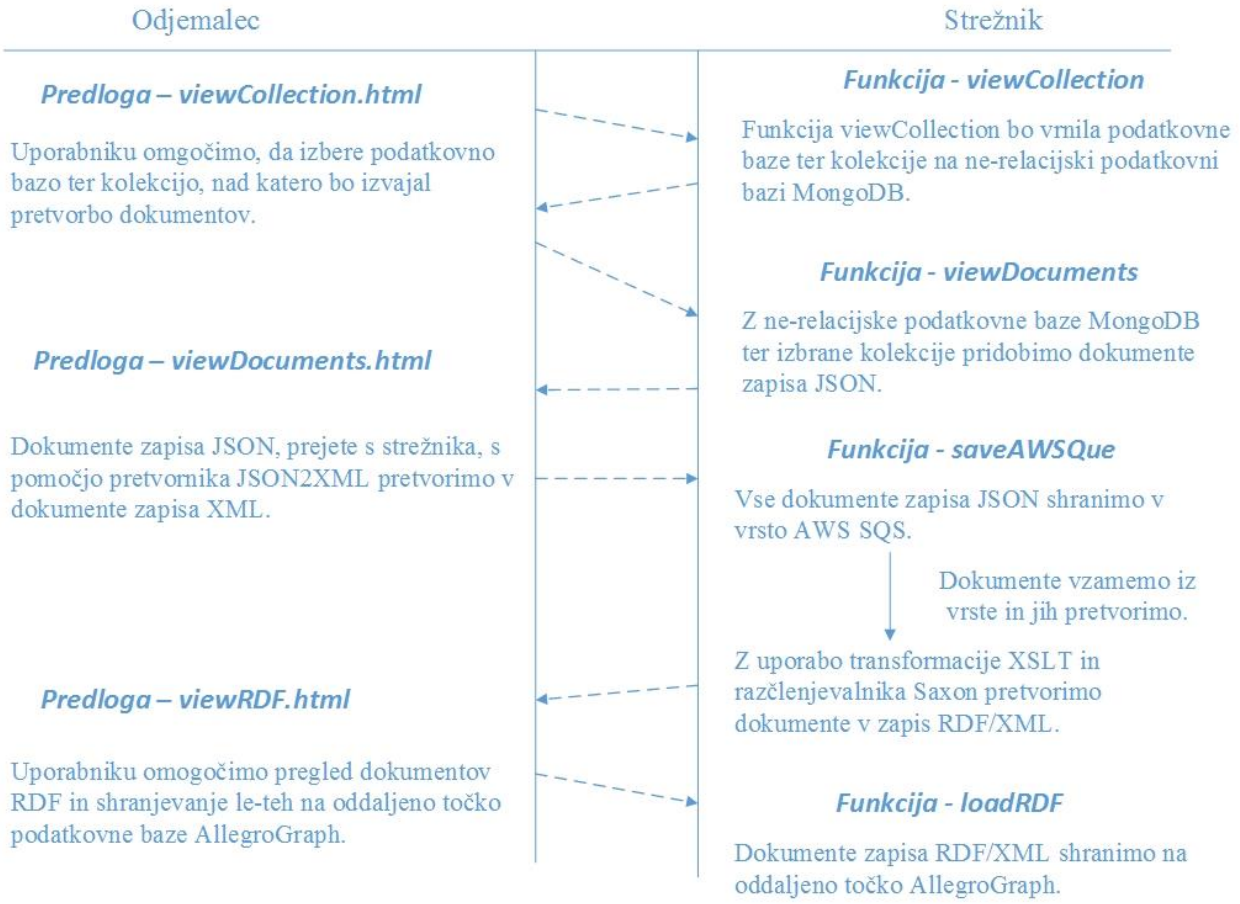
Name	Instance ID	Instance Type	Availability Zone	Instance State	Status Checks	Alarm Status	Public DNS	Public IP
AllegroGraph	i-4896570a	t1.micro	eu-west-1c	stopped		None	ec2-54-72-50-104.eu-w...	54.72.50.104
AW-VM-I1-S1	i-26c18b67	t1.micro	eu-west-1a	stopped		None		
AW-VM-I2-S1	i-4ac18b0b	t1.micro	eu-west-1a	stopped		None		
AW-VM-I3-S1	i-d0c58f91	t1.micro	eu-west-1a	stopped		None		
AW-VM-MongoDB	i-a08642e2	t1.micro	eu-west-1c	stopped		None	ec2-54-72-213-53.eu-w...	54.72.213.53
AW-VM-Neo4j	i-70c8c431	t1.micro	eu-west-1a	stopped		None	ec2-54-72-222-121.eu-...	54.72.222.121
AW-VM-Shard1	i-a8e2fbeb	t1.micro	eu-west-1c	stopped		None	ec2-54-72-92-7.eu-wes...	54.72.92.7
AW-VM-Shard2	i-074b7b46	t1.micro	eu-west-1a	stopped		None	ec2-54-76-10-174.eu-w...	54.76.10.174

Slika 50: Navidezne naprave na javnem ponudniku storitev Amazon AWS.

Za pravilno delovanje navedene arhitekture (integracije, podpora črepinjenju in replikaciji) smo uporabili različne instance nerelacijskih podatkovnih baz MongoDB, Neo4j ter AllegroGraph. Za namen povezave aplikacije in arhitekture smo uporabili programske vmesnike API, ki omogočajo komunikacijo z izbrano podatkovno bazo (npr. *pymongo-MongoDB, os- AllegroGraph*).

7.1 Predstavitev implementacije pretvorbe dokumentov za namen integracije

Za integracijo podatkovnih baz s pomočjo semantičnega spleta ter možnost izvajanja skupnega poizvedovalnega jezika SPARQL smo dokumente zapisa JSON, shranjene v nerelacijski podatkovni bazi MongoDB, pretvorili v dokumente zapisa XML. Pozneje smo te dokumente pretvorili v dokumente zapisa RDF/XML in jih shranili na oddaljeno podatkovno bazo AllegroGraph. Spletno aplikacijo smo implementirali v razvojnem okolju Django, kjer smo z uporabo programskega modela MVC podprli interakcijo z uporabniki. Dokumente smo pretvarjali po postopku, prikazanem spodaj [Slika 51].



Slika 51: Koraki za pretvorbo dokumentov zapisa JSON v zapis RDF/XML.

Pri integraciji nerelacijskih podatkovnih baz mora uporabnik predhodno izbrati polja dokumentov zapisa JSON, katerih vrednosti bodo uporabljene pri izvajanju sočasnih razširjenih poizvedb. Uporabnik lahko izbere poljubno število polj, ki jih bo vključil v poizvedovalni jezik SPARQL. Namen te metode je predhodno izključiti tiste dokumente, ki ne bodo vključeni v poizvedbo. S tem načinom prihranimo čas in podatkovni prostor, ki je potreben za pretvorbo dokumentov. Nerelacijska podatkovna baza MongoDB podpira uporabo funkcij, implementiranih v programskem jeziku JavaScript. V programskem jeziku Python smo s funkcijo *eval* na nerelacijsko podatkovno bazo poslali implementirano funkcijo v programskem jeziku JavaScript. Ta predstavlja poizvedbo MapReduce, ki pridobi vsa imena polj vseh dokumentov, shranjenih v izbrani kolekciji. Postopek za pridobitev vseh dokumentov z izbranimi polji poteka v dveh korakih. Sprva uporabnik s seznama izbere ime podatkovne baze ter ime kolekcije nad katero bo poizvedba MapReduce pridobila imena polj v dokumentih zapisa JSON. V drugem koraku uporabnik izbere tista polja, ki bodo uporabljena pri integraciji nerelacijskih podatkovnih baz [Programska koda 2].

```
$('#cbx_collections').on('select', function (event)
{
    var args = event.args;
    if (args) {
        var value = args.item.value;
        selectedCollection = value;
        $.ajax({
            type: "POST",
            url: "/getkeysAjax",

            data:{'params':JSON.stringify({'db':selectedDB,'collection':sel
            ectedCollection})},
            dataType: "json",
            success : function(jsonrec) {
                data = jsonrec.server_response;
                $("#lbx_keys").jqxListBox({width: 200, source: data,
            height: 250, multiple: true});
                $("#cbx_collections").jqxComboBox({ disabled: true });
            }

        });
    }
});
```

Programska koda 2: Pošiljanje izbranih polj preko asinhrone funkcije Ajax.

Ko modul *krmilnik* od uporabnika prejme zahtevek POST z imenom izbrane podatkovne baze ter kolekcije, funkcija na strani strežnika nad nerelacijsko podatkovno bazo izvede poizvedbo MapReduce [Programska koda 3]. Ta v kolekcijo poimenovano *MagManager* shrani vsa imena polj, ki se nahajajo v dokumentih zapisa JSON.

```
@csrf_exempt
def getkeysAjax(request):
    #Metoda POST, ki pridobi podatke o izbrani podatkovni bazi in kolekciji.
    if request.method == 'POST':
        jsonParam = request.POST.getlist('params')
        jsonStr = json.loads(jsonParam[0])
        pymongoconn = MongoClient(MONGODB_IP,MONGODB_PORT)
        db = pymongoconn[jsonStr['db']]
        #Funkcija MapReduce vrne polja, ki so navedena v dokumentih zapisa JSON.
        #Funkcijo posljemo neposredno na podatkovno bazo z ukazom eval.
        #V podatkovno bazo MagManager shranimo rezultat poizvedbe MapReduce.
        db.eval("""function (collName){
        db.runCommand({
          "mapreduce" : collName,
          "map" : function() {
            for (var key in this) { emit(key, null); }
          },
          "reduce" : function(key, stuff) { return null; },
          "out": {replace: "RDFkeys",db: "MagManager"}
        })
        }""",jsonStr['collection'])

        objects_list = []


        for s in pymongoconn['MagManager'].RDFkeys.find():
            objects_list.append(str(s["_id"]))

        #Oblikujemo JSON odgovor.
        response_dict = {}
        response_dict.update({'server_response': objects_list })
        pymongoconn.close()


    return
HttpResponse(json.dumps(response_dict),mimetype='application/json')
```

Programska koda 3: Podpora asinhronim klicem Ajax in pošiljanje poizvedbe MapReduce na nerelacijsko podatkovno bazo MongoDB.

V nadaljevanju postopka mora uporabnik izbrati ustrezna polja, ki jih na strani odjemalca, programska oprema preko asinhronega zahtevka Ajax pošlje na stran strežnika. Tam so vsi prejeti podatki shranjeni v sejo. Za podporo shranjevanja vrednosti v sejo smo morali v razvojnem okolju Django uporabiti poseben modul (*mongo_sessions*), ki v nerelacijski podatkovni bazi ustvari kolekcijo za shranjevanje sej. Poleg namestitve modula se je potrebno v nastavitveni datoteki (*Settings.py)* sklicevati na uporabo tega modula [Programska koda 4].

```
#Nastavitev v nastavitveni datoteki Settings.py.
SESSION_ENGINE = 'mongo_sessions.session'
connection = MongoClient('192.168.1.143',27017)
MONGO_CLIENT = connection.MagManager
MONGO_SESSIONS_COLLECTION = 'mongo_sessions'

#Primer uporabe shranjevanja vrednosti v spremenljivke seje.
session_engine['db'] = jsonStr['db']
session_engine['collection'] = jsonStr['collection']
session_engine['selectedKeys'] = jsonStr['selectedKeys']
session_engine.save()
```

Programska koda 4: Nastavitev modula v nastavitveni datoteki (Settings.py) in primer uporabe.

Po prejemu vseh ustreznih parametrov programska oprema izvede pretvorbo dokumentov iz zapisa JSON v zapis XML, pri čemer uporablja knjižnico, implementirano v programskem jeziku JavaScript *(Json2XML)*. Ob uspešni pretvorbi dokumentov zapisa JSON v zapis XML programska oprema s funkcijo [Programska koda 5] pretvori dokumente v zapis RDF/XML. Za ta namen smo v projekt vključili razčlenjevalnik *Saxon*, ki ob navedbi dokumentov zapisa XML ter datoteke zapisa XSLT pretvori dokumente v zapis RDF/XML. Funkcija, ki predstavlja pretvornik dokumentov, v prvem koraku najprej shrani dokumente zapisa XML na datotečni sistem.

```
...
# Preberemo sejni ključ.
session_data = session_engine.load()
#Vse dokumente zapisa XML sprva shranimo na datotečni sistem,
#kjer ime datoteke vsebuje sejni ključ.

os.makedirs(os.path.dirname(os.path.realpath(__file__))+"/Pretvorniki/Saxon
/MAGPrimeri/XMLDokumenti/XMLFolder"+session_data.get('SessionID'))
xmlArrayCounter = 0
for xmlDoc in xmlArray:

      text_file =
      open(os.path.dirname(os.path.realpath(__file__))+"/Pretvorniki/Saxon/
      MAGPrimeri/XMLDokumenti/XMLFolder"+session_data.get('SessionID')+"/XM
      LDoc"+session_data.get('SessionID')+"_"+str(xmlArrayCounter)+".xml",
      "w")
      text_file.write(xmlDoc)
      text_file.close()
      xmlArrayCounter = xmlArrayCounter +1
...
```

Programska koda 5: Shranjevanje dokumentov zapisa XML na datotečni sistem.

V zadnjem koraku pretvorbe funkcija pridobi dokumente zapisa XML in jih s klicem razčlenjevalnika Saxon, implementiranega v programskem jeziku Java, in navedbo transformacije

XSLT pretvori v zapis RDF/XML [Programska koda 6]. Za uspešno izvedbo klica datoteke *JAR* smo uporabili modul *os.system*.

```
...
#Iz mape, katere ime je sestavljeno v odvisnosti od ključa seje, preberemo
datoteke. Za vsako prebrano datoteko s pomočjo razčlenjevalnika Saxon
izvedemo transformacijo dokumentov.

ListXMLDocumentsName =
os.listdir(os.path.dirname(os.path.realpath(__file__))+"/Pretvorniki/Saxon/
MAGPrimeri/XMLDokumenti/XMLFolder"+session_data.get('SessionID'))

os.makedirs(os.path.dirname(os.path.realpath(__file__))+"/Pretvorniki/Saxon
/MAGPrimeri/RDFDokumenti/RDFFolder"+session_data.get('SessionID'))

RDFArrayCounter = 0

for item in ListXMLDocumentsName:
      osPath = os.path.dirname(os.path.realpath(__file__));

      os.system('java -jar '+osPath+'/Pretvorniki/Saxon/saxon9he.jar -t
      '+osPath+'/Pretvorniki/Saxon/MAGPrimeri/XMLDokumenti/XMLFolder'+sessi
      on_data.get('SessionID') + '/' + item +' '+ osPath+
      '/Pretvorniki/Saxon/MAGPrimeri/XML2RDF.xsl >
      '+osPath+'/Pretvorniki/Saxon/MAGPrimeri/RDFDokumenti/RDFFolder'+sessi
      on_data.get('SessionID')+'/RDF'+session_data.get('SessionID')+'_'+str
      (RDFArrayCounter)+'.rdf')

      RDFArrayCounter = RDFArrayCounter + 1
...
```

Programska koda 6: S pomočjo razčlenjevalnika pretvorimo dokumente zapisa XML v dokumente zapisa RDF/XML.

Transformacija XSLT [Programska koda 7] skrbi za pravilno pretvorbo dokumentov zapisa XML v dokumente zapisa RDF/XML.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE rdf:RDF [
  <!ENTITY rdf "http://www.w3.org/1999/02/22-rdf-syntax-ns#">
  <!ENTITY basic "http://www.w3.org/1999/02/22-rdf-syntax-ns#">
  <!ENTITY ns "http://www.w3.org/1999/02/22-rdf-syntax-ns#">
  <!ENTITY about "http://mag.org/">
  <!ENTITY resource "http://mag.org/ontology/">
  <!ENTITY ontology "http://mag.org/ontology/">
  <!ENTITY ns0 "http://www.w3.org/1999/02/22-rdf-syntax-ns#">
]>
#Registracija osnovnih imenskih prostorov.
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
version="2.0" xmlns:rdf="&rdf;" xmlns="&basic;">
<xsl:strip-space elements="*"/>
<xsl:output indent="yes"/>
<xsl:param name="ns-prefix" select="'ns0'"/>
<xsl:param name="ns-namespace" select="'http://example.org/ontology/'"/>

<xsl:template match="/">
      #Ustvarimo element RDF, ki mu s pomočjo <xsl:template match='*'>
      dodamo preostale vrednosti.
      <RDF>
            <xsl:copy>
                  <xsl:apply-templates />
            </xsl:copy>
      </RDF>
</xsl:template>

<xsl:template match="*" name="element">
<Description>

#Element description z atributom about vsebuje id dokumenta, shranjenega v
MongoDB.
<xsl:attribute name="rdf:about">&about;<xsl:value-of select="local-
name()"/>/<xsl:value-of select="id"/></xsl:attribute>

#Element type predstavlja ime razreda (npr. oseba).
<xsl:element name="type">
      <xsl:attribute name="rdf:resource">&resource;
      <xsl:value-of select ="local-name()"/></xsl:attribute>
</xsl:element>

#V tem delu pridobimo ostale elemente zapisa XML in jih dodamo v dokument
zapisa RDF/XML.
<xsl:for-each select="*[(name() != 'id')]">
      <xsl:element name="{local-name()}">
            <xsl:namespace name="{$ns-prefix}" select="$ns-namespace"/>
            <xsl:value-of select="."/>
      </xsl:element>
</xsl:for-each>
</Description>
</xsl:template>
</xsl:stylesheet>
```

Programska koda 7: Transformacija XSLT.

Za sinhronizacijo med podatkovnima bazama MongoDB in AllegroGraph smo namestili modul, imenovan *MongoWatch,* ter razvili funkcijo v programskem jeziku JavaScript [Programska koda 8], ki v izbrano kolekcijo shranjuje vse spremembe, ki se dogajajo v nerelacijski podatkovni bazi MongoDB. Pozneje lahko te dokumente z enakim postopkom, kot je bil naveden zgoraj, pretvorimo v dokumente zapisa RDF/XML in jih shranimo v podatkovno bazo AllegroGraph.

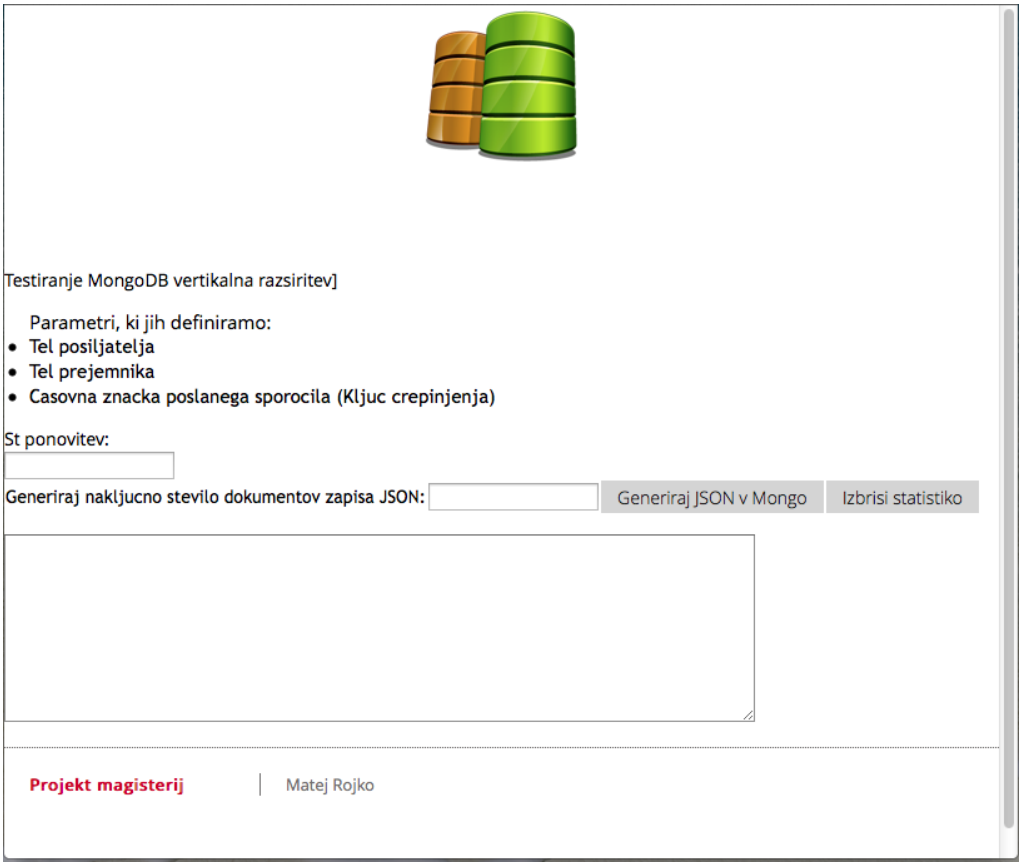
```
db.system.js.save(
{ _id: "MagMongoWatch",value:
    function() {
    //Uporabimo OpLog.
    var coll = db.oplog.rs;
    var lastTimeStamp = coll.find().sort({ '$natural' : -1 })[0].ts;
    while(1){

        cursor = coll.find({ ts: { $gt: lastTimeStamp } });
        cursor.addOption( 2 );
        cursor.addOption( 32 );

        while( cursor.hasNext() ){
            //Dokument BSON, ki vsebuje zadnjo spremembo.
            var doc = cursor.next();
            lastTimeStamp = doc.ts;
            //Dokument shranimo v kolekcijo.
            db = db.getSiblingDB('MagMongoWatchDB');
            db.MagMongoWatchColl.insert(doc.o);
        }
}}})
```

Programska koda 8: Funkcija JavaScript, ki zagotavlja shranjevanje sprememb v temu namenjeno kolekcijo na nerelacijski podatkovni bazi MongoDB.

7.2 Analiza vpliva vertikalne in horizontalne razširljivosti na učinkovitost delovanja sistema

7.2.1 Analiza vpliva vertikalne razširljivosti na učinkovitost delovanja sistema

Za izvedbo analize učinkovitosti smo v razvojnem okolju Django pripravili spletno stran [Slika 52], kjer uporabnik določa število ponovitev ter število dokumentov, ki jih želi shraniti. Kot rezultat prejme tekstovno datoteko, ki vsebuje izmerjeni čas.



Slika 52: Grafični vmesnik za uporabo merjenja vpliva vertikalne razširljivosti na učinkovitost delovanja sistema.

Za namen izdelave analize vertikalne razširljivosti smo na javnem ponudniku storitev v oblaku izbrali navidezni napravi *t2.micro* oziroma *t2.medium*, z dodeljeno različno količino delovnega pomnilnika. Zaradi boljše možnosti primerjanja rezultatov meritev med vertikalno oziroma horizontalno porazdelitev sistema, smo podatke na instanco nerelacijske podatkovne baze MongoDB pošiljali z neodvisne navidezne naprave. Pri merjenju časov smo v programskem jeziku Python uporabili modul *Time* [Programska koda 9].

```
#Inicializacija povprečnega časa branja in pisanja.
avgTimeRead = 0
avgTimeWrite = 0

#Inicializacija datoteke, v kateri bomo shranili statistiko.
file_ =
open(os.path.dirname(os.path.realpath(__file__))+'/CPUStatistics/Statistics.txt',
'a')

#Preko zahtevka POST s strani uporabnika prejmemo število dokumentov ter število
ponovitev.
numberOfDocuments = request.POST['numberOfDocuments']
numOfLoop = int(request.POST['numOfLoop'])

#Izvajamo število ponovitev.
for i in range(0, numOfLoop):
       #Za vsako ponovitev vzpostavimo novo povezavo.
        conn = MongoClient('54.72.98.87',27017)
       #Ob vsaki ponovitvi izbrišemo obstoječo podatkovno bazo.
        conn.drop_database("bulkDB")
        db = conn.bulkDB
       #Inicializacija začetnega časa.
        start = time.time()
        #Ustvarjanje naključnih dokumentov.
db.bulkCollection.insert(({'sT':
str(generatePhoneNumbers()),'rT':str(generatePhoneNumbers()),'TS':datetime.datetime
.fromtimestamp(int("1284101485")).strftime('%Y-%m-%d')} for i in
xrange(int(numberOfDocuments))))

       #Konec meritve.
       end = time.time()
       #V spremenljivko za povprečni čas shranimo čas prve iteracije.
        avgTimeWrite += (end - start)
        file_.write(numberOfDocuments+" : (pisanje) - "+ str(end - start)+" (branje)
- ")
        #Inicializacija začetnega časa za namen meritev branja.
        start = time.time()
        #Zanima nas število sporočil, poslanih iz ene telefonske številke na drugo
        db.bulkCollection.find({"sT":"1","rT":"5"}).count()
        end = time.time()
        avgTimeRead += (end - start)
        file_.write(str(end - start)+'\n')

        conn.close()

#Izračunamo ter zapišemo povprečni čas.
file_.write(numberOfDocuments+" : (Povprecje pisanje) - "+
str(avgTimeWrite/numOfLoop)+" (Povprecje branje) - ")
file_.write(str(avgTimeRead/numOfLoop)+'\n')

file_.close()
file_ =
open(os.path.dirname(os.path.realpath(__file__))+'/CPUStatistics/Statistics.txt',
'r+')

#Uporabniku vrnemo podatke o statistiki.
reader = file_.read()
file_.close()

return HttpResponse(json.dumps({"reader":reader}),mimetype='application/json')
```

Programska koda 9: Merjenje vpliva vertikalne razširljivosti sistema.

7.2.2 Analiza vpliva horizontalne razširljivosti na učinkovitost delovanja sistema

Za podporo merjenja vpliva horizontalne razširljivosti smo na dve navidezni napravi z operacijskim sistemom Ubuntu namestili nerelacijsko podatkovno bazo MongoDB in za podporo izvajanju metodi črepinjenja namestili nastavitvene strežnike (*angl. Config Servers*), ki shranjujejo informacije o nastavitvah črepinj in naslovih podatkov na posameznih črepinjah. Namestitveni strežnik smo namestili na neodvisni navidezni strežnik, imenovan *mongos*.

Strežnik (Mongos – IP: 54.72.98.87):

```
#Ustvarimo datoteko za namen hranjenja podatkov v podatkovni bazi.
mkdir -p /db/config/data

#Na virtualni napravi Mongos, zaženemo instance mongod in kot parameter podamo
configsvr.
./mongod --port 29021 --dbpath /db/config/data --configsvr
```

Za uspešno usmerjanje podatkov smo namestili usmerjevalnik črepinj (*angl. Shard Controller*), katerega nalogo si lahko predstavljamo kot usmerjevalnik podatkov med črepinjami in uporabniki. Ta skrbi za približno enakomerno porazdelitev veder med navideznimi napravami, ki predstavljajo črepinje sistema.

```
#Zaženemo instanco mongos in kot parameter podamo naslov IP kontrolerja ter
številko vrat instance mongos.

./mongos --configdb Mongos:29021 --port 30001
```

Na navideznem strežniku smo zagnali prvo instanco nerelacijske podatkovne baze, ki je predstavljala črepinjo sistema.

Strežnik (Shard1 - IP: 54.72.148.50):

```
#Na strežniku (Shard1) zaženemo instanco ne-relacijske podatkovne baze
MongoDB, ki predstavlja prvo črepinjo v sistemu.
sudo mkdir -p /db/shard0/data
./mongod --port 30023 --dbpath /db/shard0/data --shardsvr
```

Na drugem navideznem strežniku, imenovanem ***Shard2***, smo namestili drugo instanco nerelacijske podatkovne baze MongoDB, ki je predstavljala črepinjo sistema.

Strežnik (Shard2 – IP: 54.72.212.101):

```
#Na strežniku (Shard2), zaženemo instanco ne-relacijske podatkovne baze
MongoDB, ki predstavlja drugo črepinjo v sistemu.
sudo mkdir -p /db/shard1/data
./mongod --port 30022 --dbpath /db/shard1/data --shardsvr
```

Z naslednjimi ukazi smo dodali instanci v sistem črepinj in pozneje izpisali status [Slika 53].

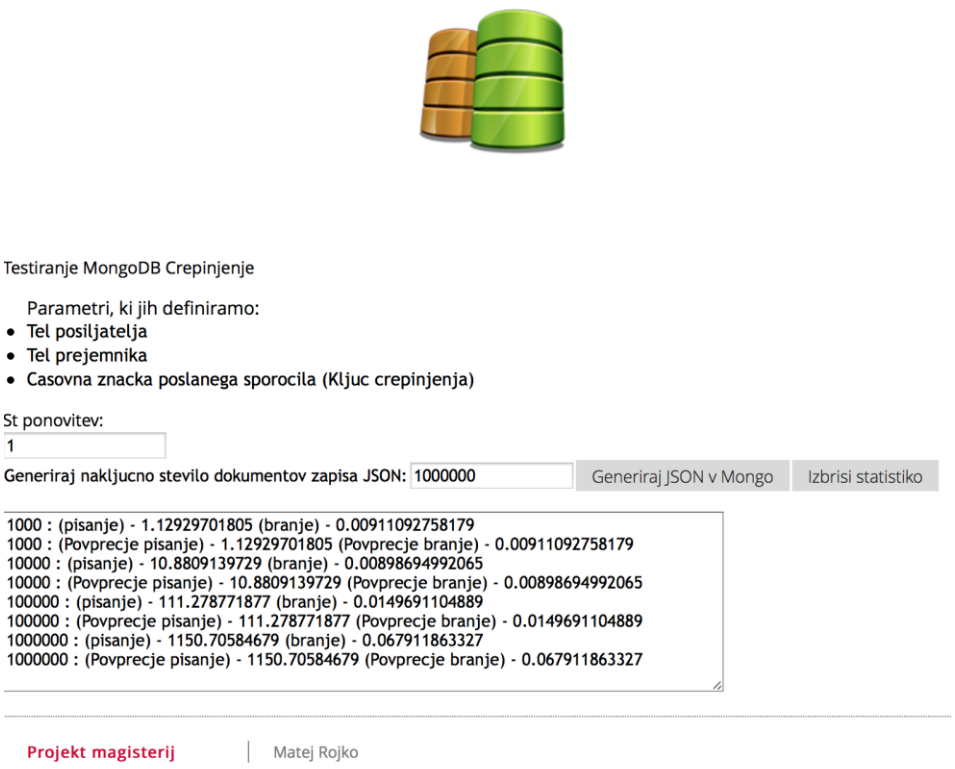
Strežnik (Mongos – IP: 54.72.98.87)

```
#Za dodajanje nove instance moramo vzpostaviti povezavo na kontroler, ki se
nahaja na prvem navideznem strežniku.

./mongo Mongos:30001

#Uporabimo kolekcijo admin.
use admin

#Dodamo obstoječi črepinji v sistem črepinjenja.
db.runCommand({addShard: "Shard1:30023"})
db.runCommand({addShard: "Shard2:30022"})

#Izpis vseh črepinj, ki jih hranimo v sistemu.
db.runCommand({listshards:1})
```

```
mongos> db.runCommand({listshards:1})
{
        "shards" : [
                {
                        "_id" : "shard0000",
                        "host" : "Shard1:30023"
                },
                {
                        "_id" : "shard0001",
                        "host" : "Shard2:30022"
                }
        ],
        "ok" : 1
}
```

Slika 53: Izpis statusa črepinj.

Programska rešitev za izvajanje meritev se za analizo vpliva horizontalne razširljivosti sistema na učinkovitost delovanja sistema veliko ne razlikuje od programske rešitve za merjenje rezultatov vertikalne razširljivosti sistema [Programska koda 9]. Kljub vsemu smo za slednjo meritev pripravili podobni uporabniški vmesnik [Slika 54], ki uporabniku omogoča določanje števila ponovitev ter število dokumentov, ki jih želi shraniti na porazdeljeni sistem. Naj omenimo, da je lastnost nerelacijske podatkovne baze, kjer za uporabo metode črepinjenja ni potrebno opraviti velikih sprememb programske opreme, zelo dobrodošla. Nerelacijska podatkovna baza MongoDB nam zagotavlja, da nad nivojem črepinj uporabljamo instanco, imenovano *mongos*, ki v sodelovanju z usmerjevalnikom črepinj skrbi za pravilno drobljenje podatkov.



Slika 54: Grafični vmesnik za uporabo merjenja vpliva horizontalne razširljivosti na učinkovitost delovanja sistema.

Za pridobitev natančnih rezultatov smo ob vsaki iteraciji zanke izbrisali podatkovno bazo ter ponovno nastavili metodo črepinjenja, ki se izvaja nad izbrano kolekcijo [Programska koda 10]. Za namen povezovanja na podatkovno bazo smo v programskem jeziku Python uporabili programski vmesnik *pymongo*. Programska koda [Programska koda 10] se od že navedene [Programska koda 9] razlikuje predvsem v nastavitvi metode črepinjenja in povezavi na instanco, imenovano *mongos*.

```
…
#Povezava na instanco mongos, nameščeno na navidezni napravi Shard1.
conn = MongoClient('54.72.98.87',30001)

#Ob vsaki iteraciji izbrišemo obstoječo podatkovno bazo.
conn.drop_database("bulkDB")
db = conn.bulkDB

#Nastavitev metode črepinjenja.
conn.admin.command('enableSharding','bulkDB' )
conn.admin.command('shardCollection', 'bulkDB.bulkCollection', key={'_id':
'hashed'})
…
```

Programska koda 10: Merjenje vpliva horizontalne razširljivosti sistema.

Med izvajanjem metode shranjevanja podatkov smo preverili število shranjenih podatkov na instanci *mongos* ter posamezni črepinji [Slika 55].

Slika 55: Prikaz števila vstavljenih dokumentov na instanci *mongos* ter posamezni črepinji.

8. Sklepne ugotovitve

Motivacijo za integracijo povpraševanj različnih vrst nerelacijskih podatkovnih baz smo našli v svetu bioinformatike. Zanimalo nas je, na kakšen način lahko s skupnim poizvedovalnim jezikom poizvedujemo nad različnimi nerelacijskimi podatkovnimi bazami, kjer vsaka izmed njih uporablja svoj poizvedovalni jezik. Za zagotavljanje učinkovite razširiljivosti nas je predvsem zanimal vpliv uporabe metode črepinjenja, vertikalne razširljivosti sistema ter uporabnost replikacije na dveh različnih ponudnikih storitev v oblaku Amazon AWS in Microsoft Azure ter lokalnim omrežjem.

Sprva smo v magistrski nalogi analizirali različne tipe nerelacijskih podatkovnih baz *ključ-vrednosti, graf, vrste-razširljivi zapisi* ter *dokument* in prikazali najpogostejše primere uporabe. V nadaljevanju smo pregledali storitvene in namestitvene modele računalništva v oblaku ter večnivojsko arhitekturo za dosego aplikacijske šibke sklopljenosti.

V poglavju o integraciji povpraševanj različnih vrst nerelacijskih podatkovnih baz smo v začetnem delu teoretično predstavili različne možnosti in izpostavili dva načina. Prvi predstavlja premik poizvedb bližje k podatkom, kar od nas zahteva pretvorbo poizvedb SPARQL v dotični poizvedovalni jezik izbrane podatkovne baze in implementacijo končne točke za pravilno vračanje rezultatov. Drugi način zahteva premik podatkov bližje k poizvedbami in s tem pretvorbo podatkov v dokumente zapisa RDF/XML ter oglaševanja slednjih na končni točki. V tem delu smo implementirali program, ki omogoča samodejno pretvorbo podatkov, shranjenih v nerelacijski podatkovni bazi MongoDB in oglaševanje le-teh preko končne točke AllegroGraph. Z implementacijo tega dela smo uspeli z enotnim poizvedovalnim jezikom SPARQL, brez programskega združevanja podatkov, poizvedovati nad dvema različnima nerelacijskima podatkovnima bazama, MongoDB in Neo4j. Za prikaz delovanja smo na nerelacijsko podatkovno bazo MongoDB shranjevali podatke o osebah, medtem ko smo nerelacijsko podatkovno bazo Neo4j uporabili za beleženje pošiljanja sporočil med uporabniki. Tako kot smo omenili v tem poglavju, bi podobno rešitev lahko uporabili v bioinformatiki za namen primerjave podatkov o variacijah z javno dostopnimi podatkovnimi bazami, ki za svoje oglaševanje uporabljajo semantični splet.

V nadaljevanju magistrske naloge smo predlagali arhitekturno zasnovo rešitve, ki za namen izdelave analize vpliva vertikalne oziroma horizontalne razširitve sistema ter uporabo replikacije, predvideva shranjevanje podatkov na porazdeljeni sistem navideznih naprav.

Za namen drobljenja podatkov smo uporabili metodo črepinjenja in sprva opisali dobre ter slabe prakse pri izbiri ključa črepinjenja, nato pa se osredotočili na osnovna algoritma, po katerih

metoda črepinjenja deluje (t.j. algoritem konsistentnega zgoščevalnega obroča in algoritem z določanjem obsega ključa). Nerelacijska podatkovna baza MongoDB, ki smo jo v osnovi uporabili pri implementaciji predlagane arhitekture, za izbiro ključa omogoča uporabo izvlečka dokumenta oziroma izbiro ključa z definiranim obsegom. V primeru izbire ključa z definiranim obsegom moramo biti še posebej previdni, saj napačna izbira ključa lahko povzroči nezmožnost nadaljnjega črepinjenja ali pa neenakomerno porazdelitev podatkov na črepinje sistema. V primeru zahteve po zamenjavi ključa moramo ponoviti proces drobljenja podatkov.

Za izvedbo analize vpliva vertikalne oziroma horizontalne razširljivosti smo implementirali programsko rešitev, kjer smo dokumente zapisa JSON shranjevali na izbrano navidezno napravo. Pri izvajanju meritev smo v podatkovno bazo shranjevali od 1000 do 10 milijonov dokumentov zapisa JSON in na koncu izvajali operacijo branja. Pri izdelavi prve analize, kjer smo analizirali vertikalni vpliv, se pri shranjevanju naše količine podatkov rezultati niso očitno razlikovali. Zaradi omrežne zakasnitve pri komunikaciji med navideznima napravama s programsko opremo in instanco nerelacijske podatkovne baze, vertikalna razširljivost ni pripomogla k občutno hitrejšemu izvajanju operacij branja ter pisanja.

Pri drugi analizi smo enake zapise podatkov z metodo črepinjenja drobili na porazdeljeni sistem dveh navideznih naprav. Rezultati so bili slabši, saj smo za shranjevanje 3 milijonov dokumentov porabili približno enako časa, kot pri shranjevanju 10 milijonov dokumentov na eno instanco nerelacijske podatkovne baze MongoDB. Z uporabo metode črepinjenja smo za operacijo branja pridobili boljše čase kot s primerjavo ene instance nerelacijske podatkovne baze MongoDB.

Ugotovili smo, da se je z uporabo metode črepinjenja povečal čas pisanja v podatkovno bazo, medtem ko se je čas izvajanja operacije branja v primerjavi z eno instanco podatkovne baze zmanjšal. Z zamenjavo ključa smo uspeli izboljšati čas izvajanja operacije pisanja.

V nadaljevanju tega poglavja smo se posvetili replikaciji sistema in namestitvi le-tega na dva različna ponudnika storitev v oblaku Amazon AWS, Microsoft Azure ter lokalno omrežje. Pri uporabi distribuiranega sistema replikacije na različnih ponudnikih storitev v oblaku ter lokalnim omrežjem smo ugotovili, da v sistem lahko pripnemo poljubne fizične ali navidezne naprave. Za delovanje je pomembno zagotoviti vidnost in izmenjavo podatkovnega prometa med napravami. Z uporabo replikacije in nastavitvijo sekundarnega strežnika zagotovimo, da bomo v lokalnem omrežju hranili vse kopije podatkov, ki jih shranjujemo na javne ponudnike storitev v oblaku.

V magistrski nalogi smo se osredotočili na povpraševanje po različnih vrstah nerelacijskih podatkovnih baz ter analizirali vpliv vertikalne oziroma horizontalne razširljivosti sistema. S pretvorbo podatkov smo zagotovili uspešno uporabo enotnega poizvedovalnega jezika nad

različnima vrstama nerelacijskih podatkovnih baz *dokument* in *graf*. Rezultati tega dela predstavljajo integracijo podatkov, ki jih hranimo pri sebi, z javno dostopnimi podatki, do katerih dostopamo preko končnih točk. Integracijo s semantičnim spletom bi lahko izboljšali z razvojem lastne končne točke in pretvorbo poizvedovalnega jezika SPARQL v poizvedovalni jezik izbrane podatkovne baze.

Z analizo učinkovitosti smo prikazali vpliv vertikalne oziroma horizontalne razširljivosti sistema na porazdeljeni sistem navideznih naprav na javnem ponudniku storitev v oblaku Amazon AWS. Rezultati tega dela predstavljajo pozitivni učinek uporabe horizontalne razširljivosti za izvajanje operacije branja ter počasnejše delovanje izvajanja operacije pisanja. Predvsem za prikaz vpliva vertikalne razširljivosti sistema verjamemo, da bi s shranjevanjem večje količine podatkov zagotovili natančnejše rezultate.

Literatura

[1] (2008) SPARLQ Query Language. Dostopno na: http://www.w3.org/TR/rdf-sparql-query/ (december 2014).

[2] (2014) What is Scalability? Dostopno na: http://msdn.microsoft.com/en-us/library/aa578023.aspx (december 2014).

[3] (2014) AllegroGraph. Dostopno na: http://franz.com/agraph/allegrograph/ (maj 2014).

[4] D. McCreary in A. Kelly, *Making Sense of NoSQL: A guide for managers and the rest of us*, Shelter Island: Manning, 2014, pogl. 4, str. 64, str. 82.

[5] T. Shashank, *Professional NoSQL*, Canada: John Wiley & Sons, 2011, str. 14–17, str. 60, str. 74–79.

[6] (2014) Googlebot. Dostopno na: http://support.google.com/webmasters/answer/182072?hl=en (september 2014).

[7] Google Crawling and Indexing. Dostopno na: http://www.google.com/intl/en/insidesearch/howsearchworks/crawling-indexing.html (september 2014).

[8] V. Gaurav, *Getting Started with NoSQL*, United Kingdom: Packt Publishing Ltd, 2013, str. 29, str. 43–46.

[9] J. Partner, A. Vukotic, N. Watt, *Neo4j in Action*, Shelter Island: Manning, 2012, pogl. 1.

[10] (2012) Graph traversals. Dostopno na: http://www.cs.cornell.edu/courses/CS2112/2012sp/lectures/lec24/lec24-12sp.html (oktober 2014).

[11] A. Nayak, A. Poriya, D. Poojary, "Type of NoSQL Databases and its Comparison with Relational Databases", *International Journal of Applied Information Systems (IJAIS)*, št. 4, zv. 5, str. 17, 2013.

[12] (2013) CQL Language Reference. Dostopno na: http://www.datastax.com/docs/1.1/references/cql/index (december 2014).

[13] (2014) MapReduce. Dostopno na: http://www.ibm.com/software/data/infosphere/hadoop/mapreduce (avgust 2014).

[14] (2012) Document-oriented database. Dostopno na: http://en.wikipedia.org/wiki/Document-oriented_database (december 2014).

[15] (2013) NoSQL and Ad Targeting. Dostopno na: http://info.couchbase.com/rs/northscale/images/Couchbase_Whitepaper_NoSQL_and_Ad_Targeting.pdf (december 2014).

[16] P. Mell, T. Grance, *The NIST Definition of Cloud Computing*, Gaithersburg: NIST, 2011, str. 2.

[17] P. Dragos-Paul, A. Adam, *Designing an MVC Model for Rapid Web Application Development*, Romania: Romanian-American University, 2013, pogl. 3, 4.

[18] (2008) Service Level Agreement (SLA). Dostopno na: http://www.knowledgetransfer.net/dictionary/ITIL/en/Service_Level_Agreement.htm (december 2014).

[19] Z. Mahmood, R. Hill, *Cloud Computing for Enterprise*, London: Springer, 2011, pogl. 3, str. 8.

[20] (2014) 5 Important Benefits of Infrastructure as a Service. Dostopno na: http://www.statetechmagazine.com/article/2014/03/5-important-benefits-infrastructure-service (december 2014).

[21] (2011) Critical technology. Dostopno na: http://criticaltechnology.blogspot.de/2011/09/mvc-in-three-tier-architecture.html (marec 2014).

[22] (2012) System Integration in the Cloud Era - Api vs. Integration Framework vs. Enterprise Service Bus (ESB). Dostopno na: http://www.slideshare.net/KaiWaehner/systems-integration-in-the-cloud-era-api-vs-integration-framework-vs-enterprise-service-bus-esb (avgust 2014).

[23] (2013) Web Services. Dostopno na: http://docs.oracle.com/javaee/6/tutorial/doc/giqsx.html (oktober 2014).

[24] (2014) Project Siena: Creating a WADL Configuration File. Dostopno na: http://social.technet.microsoft.com/wiki/contents/articles/23838.project-siena-creating-a-wadl-configuration-file.aspx (julij 2014).

[25] (2014) Overview of Extraction, Transformation, and Loading. Dostopno na: http://docs.oracle.com/cd/B19306_01/server.102/b14223/ettover.htm (julij 2014).

[26] (2013) ETL - Load. Dostopno na: http://www.immagic.com/eLibrary/ARCHIVES/GENERAL/WIKIPEDI/W121031E.pdf (december 2014).

[27] (2014) Internet usage statistics. Dostopno na: http://www.internetworldstats.com/stats.htm (november 2014).

[28] M. T. Mohammad, *Understanding Semantic Web and Ontologies: Theory and Applications*, Philadelphia: Philadelphia University, 2010.

[29] B. DuCharme, *Learning SPARQL*, Sebastopol: O'Reilly, 2013, str. 19–20, pogl. 2.

[30] (2013) Linked data. Dostopno na: http://www.w3.org/standards/semanticweb/data (november 2014).

[31] C. Namyoun, S. Yeol, H. Hyoil, *A Survey on Ontology Mapping*, Philadelphia: Drexel University, 2006.

[32] K. Sharifullah, S. Muhammad, *Semantic matching in hierarchical ontologies*, Pakistan: National University of Science and Technology, 2014.

[33] O. Cure, M. Lamolle in C. L. Duc, *Ontology Based Data Integration Over Document and Column Family Oriented NoSQL stores*, France: University Paris, 2011.

[34] (2010) Integrate disparate data sources with Semantic Web technology. Dostopno na: http://www.ibm.com/developerworks/library/x-disprdf/ (maj 2014).

[35] (2012) BQL. Dostopno na: http://senseidb.github.io/sensei/bql.html (december 2014).

[36] (2014) RDF 1.1 XML Syntax. Dostopno na: http://www.w3.org/TR/REC-rdf-syntax (maj 2014).

[37] (2014) Validation Service. Dostopno na: http://www.w3.org/RDF/Validator/ (oktober 2014).

[38] (2014) Neo4j-contribute. Dostopno na: https://github.com/neo4j-contrib/sparql-plugin (maj 2014).

[39] (2013) Neo4j SPARQL Plugin. Dostopno na: http://neo4j-contrib.github.io/sparql-plugin/ (maj 2014).

[40] (2014) An introduction to RDF. Dostopno na: https://jena.apache.org/tutorials/rdf_api.html (junij 2014).

[41] (2012) Curl. Dostopno na: http://curl.haxx.se/ (julij 2014).

[42] E. Wilde, M. Hausenblas, *RESTful SPARQL? Your Name It!*, Ireland: Nationa University of Ireland, 2009.

[43] (2011) What is bioinformatics? Dostopno na: http://www.ncbi.nlm.nih.gov/pubmed/11552348 (december 2014).

[44] (2014) Bio4j. Dostopno na: http://bio4j.com (marec 2014).

[45] T. Clarence, S. Aravindh, A. Shreeharsha, "Comperative Study of the New Generation, Agile, Scalable, High Performance NoSQL Databases", *International Journal of Computer Applications*, št. 20, zv. 48, 2012.

[46] B. Wilder, *Cloud Architecture Patterns*, Sebastopol: O'Reilly, 2012, pogl. 1, 2.

[47] M. Katyal, A. Mishra, *A Comparative Study of Load Balancing Algorithms in Cloud Computing Environment*, 2013.

[48] K. Chodorow, *Scaling MongoDB*, Sebastopol: O'Reilly, 2011, str. 10, str. 19–25.

[49] A. Marcus, *The NoSQL Ecosystem*, The Architecture of Open Source Application, št. 1, 2014.

[50] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, R. E. Gruber, Bigtable: A Distributed Storage System for Structured Data, 2006.

[51] (2014) Compose blog. Dostopno na: http://blog.compose.io/how-we-scale-mongodb/ (november 2014).

[52] E. Plugge, P. Membrey, T. Hawkins, *The Definitive Guide to MongoDB*, USA: Paul Manning, 2010, pogl. 10.

[53] (2013) MongoDB OpLog. Dostopno na: http://docs.mongodb.org/manual/core/replica-set-oplog/ (julij 2014).

[54] H. V. Jagadish, "Big Data and It's Technical challenges", *Communications of the ACM*, št. 7, zv. 57, str. 86–94, 2014.