UNIVERZA V LJUBLJANI

FAKULTETA ZA RAČUNALNIŠTVO IN INFORMATIKO

Damir Opačak

# Migracije med NoSQL podatkovnimi bazami

**DIPLOMSKO DELO**

UNIVERZITETNI ŠTUDIJSKI PROGRAM PRVE STOPNJE
RAČUNALNIŠTVO IN INFORMATIKA

Mentor: doc. dr. Dejan Lavbič

Ljubljana, 2013

Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

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

Št. naloge: 00137/2013
Datum: 11.04.2013

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

Kandidat: **DAMIR OPAČAK**

Naslov: **MIGRACIJE MED NOSQL-PODATKOVNIMI BAZAMI**
**MIGRATION BETWEEN NOSQL DATABASES**

Vrsta naloge: Diplomsko delo univerzitetnega študija prve stopnje

Tematika naloge:

NoSQL podatkovne baze se vedno bolj uporabljajo tudi v poslovnem okolju. Njihova konkurenčna prednost pred rešitvami, ki temeljijo na relacijskem podatkovnem modelu je predvsem v možnostih skalabilnosti in prilagajanju vedno večjim performančnim zahtevam aplikacijskih rešitev. Zaradi velikih razlik med posameznimi predstavniki skupin NoSQL rešitev je zelo velika težava migracija razvite aplikacije iz ene NoSQL podatkovne baze na drugo. V svetu relacijskih podatkovnih baz je ta problem veliko manjši, predvsem zaradi standardiziranega poizvedovalnega jezika SQL, ki v NoSQL svetu ni prisoten v tolikšni meri, saj vsaka rešitev uporablja svoj API. Naloga študenta v okviru diplomske naloge je pregled in analiza izvedljivosti migracije med posameznimi NoSQL rešitvami. Omenjena primerjava naj bo izvedena na izbrani problemski domeni, kjer je potrebno podati oceno migracije med posameznimi rešitvami glede na različne kriterije.

Mentor:

doc. dr. Dejan Lavbič

Dekan:

prof. dr. Nikolaj Zimic

Fakulteta za računalništvo in informatiko • UNIVERZA V LJUBLJANI • 2

# I Z J A V A  O  A V T O R S T V U

## diplomskega dela

Spodaj podpisani Damir Opačak, z vpisno številko **63090149**, sem avtor diplomskega dela z naslovom:

*Migracije med NoSQL podatkovnimi bazami*

S svojim podpisom zagotavljam, da:

- sem diplomsko delo izdelal/-a samostojno pod mentorstvom doc. dr. Dejan Lavbič

- so elektronska oblika diplomskega dela, naslov (slov., angl.), povzetek (slov., angl.) ter ključne besede (slov., angl.) identični s tiskano obliko diplomskega dela

- soglašam z javno objavo elektronske oblike diplomskega dela v zbirki »Dela FRI«.

V Ljubljani, dne 11. septembra 2013 Podpis avtorja: _____

*Iskreno bi se rad zahvalil vsem, ki ste mi pomagali na poti do moje diplome. Tako gre posebna zahvala mentorju, staršem in ženi.*

**Kazalo**

# SEZNAM UPORABLJENIH KRATIC

| | |
|---|---|
| ACID | Atomicity, Consistency, Isolation, Durability (atomarnost, konsistentnost, izolacija, trajnost – lastnosti transakcij v relacijski podatkovni bazi ) |
| API | Application Programming Interface (aplikacijski programski vmesnik ) |
| BASE | Basically Available, Soft-state, Eventual consistency (osnovnA dostopnost , mehko stanje in eventualna konsistentnost ) |
| CQL | Cassandra Query Language (poizvedovalni jezik za podatkovno bazo Cassandra ) |
| JSON | JavaScript Object Notation (format podatkov za izmenjavo strukturiranih dokumentov na spletu ) |
| LINQ | Language Integrated Query |
| NoSQL | Not Only SQL |
| RDF | Resource Description Framework |
| REST | Representational State Transfer (množica arhitekturnih principov za razvoj spletnih storitev ) |
| SOS | Save Our Systems (teoretično ogrodje, ki omogoča preprost dostop do različnih podatkovnih baz ) |
| SPARQL | Simple Protocol and RDF Query Language (poizvedovalni jezik za poizvedovanje po podatkovnih bazah na osnovi grafov ) |
| SQL | Structured Query Language (poizvedovalni jezik v relacijskih podatkovnih bazah ) |
| UnQL | Unstructured Data Query Language (enotni poizvedovalni jezik za NoSQL ) |
| W3C | World Wide Web Consortium |

# POVZETEK

Diplomska naloga obravnava temo razlik in posledično težav, ki nastopijo pri migracijah med NoSQL podatkovnimi bazami. V uvodnih poglavjih predstavimo problematiko relacijskih podatkovnih baz in odgovorimo na vprašanje nastanka NoSQL podatkovnih baz. Temu sledi pregled področja NoSQL podatkovnih baz, kjer so predstavljeni različni tipi NoSQL podatkovnih baz ter nekateri njihovi predstavniki.

Namen tovrstnega pregleda je prikaz specifičnih značilnosti različnih NoSQL podatkovnih baz in dejstva, da je vsaka izmed njih namenjena reševanju različnih tipov problemov. Posledično imajo različne podatkovne modele ter različne načine poizvedovanja. Predstavljeni so tudi razlogi za migracije med NoSQL podatkovnimi bazami ter načini, kako le-te olajšati.

Glavni del naloge predstavi simulacijo razvoja realne sodobne aplikacije, ki uporablja NoSQL podatkovno bazo. Realizirane in opisane so tudi migracije na druge NoSQL podatkovne baze. Prav tako so podani in realizirani konkretni nasveti glede reševanja problematike migracij med NoSQL podatkovnimi bazami. Sklepno poglavje izpostavi ključne ugotovitve glede migracije med NoSQL podatkovnimi bazami, ki zajemajo spoznanje, da razlike med NoSQL podatkovnimi bazami obstajajo z razlogom in je zato izbira ustrezne podatkovne baze že na samem začetku izjemnega pomena. Z ustrezno programsko abstrakcijo je pa možno težave ob migracijah sicer omiliti.

**Ključne besede:** NoSQL, podatkovne baze, migracije, razvoj.

## ABSTRACT

The thesis discusses the differences and, consequently, potential problems that may arise when migrating between different types of NoSQL databases. The first chapters introduce the reader to the issues of relational databases and present the beginnings of NoSQL databases.

The following chapters present different types of NoSQL databases and some of their representatives with the aim to show specific features of NoSQL databases and the fact that each of them was developed to solve specific types of problems. Subsequently, these databases have different data models and different ways of querying data. The chapter also includes potential reasons for migration between NoSQL databases and solutions to ease those migrations.

The core of the thesis consists of a simulation of development of a real-life, present-day application, using NoSQL database. Migrations to other NoSQL databases are also implemented and described. It also includes advice on saving potential difficulties with migrations between NoSQL databases. The final chapter concludes that the many differences between NoSQL databases exist for a reason, which means that choosing the appropriate databases in the very beginning of a project is crucial, yet by using appropriate code abstractions the difficulties can somewhat be mitigated.

**Keywords:** NoSQL, databases, migration, development

# 1 UVOD

Relacijske podatkovne baze so nastale pred približno štiridesetimi leti, kot rešitev za potrebe tedanjih aplikacij. Do danes se relacijske podatkovne baze niso bistveno spremenile, za razliko od aplikacij in njihovih potreb, ki so drastično drugačne. Včasih so aplikacije podpirale bistveno manjše število uporabnikov, ki so aplikacijo tipično uporabljali znotraj delovnih ur. Aplikacije so imele precej jasno določeno funkcionalnost, ki se ni veliko spreminjala, poleg tega pa je bila tudi sama količina zajetih podatkov precej majhna.

Sodobne aplikacije pa imajo lahko več (sto ) milijonov uporabnikov, ki jo uporabljajo štiriindvajset ur na dan, tristo petinšestdeset dni v letu. Količina zajetih podatkov je bistveno večja kot nekoč, obenem pa se tudi aplikacije same sproti razvijajo in spreminjajo. Odgovor na nastale probleme predstavljajo ne-relacijske podatkovne baze, ki so že od začetka zasnovane tako, da rešujejo določene probleme sodobnih aplikacij. Raznolike potrebe sodobnih aplikacij so privedle do nastanka različnih ne-relacijskih podatkovnih baz, ki rešujejo različne probleme, posledično pa imajo tudi različne podatkovne modele ter različne načine poizvedovanja. Vse sodobne ne-relacijske podatkovne baze pa lahko na kratko označimo kot NoSQL podatkovne baze, kar naj bi pomenilo »ne le SQL« (angl. *not only SQL* ), to pa nakazuje tudi na superiornost NoSQL podatkovnih baz [29].

Velike razlike med NoSQL podatkovnimi bazami negativno vplivajo na možnosti za migracijo med različnimi podatkovnimi bazami. Različne NoSQL podatkovne baze imajo namreč različne podatkovne modele, kar pomeni, da je ob migraciji pogosto potrebno spremeniti način modeliranja podatkov. Še večje pa so razlike pri poizvedovanju, kjer moramo ob migraciji drugače gledati na podatke in na to, kako bomo po le-teh poizvedovali.

Cilj diplomskega dela je predstaviti tematiko migracij med NoSQL podatkovnimi bazami ter pokazati, da so tovrstne migracije precej zahtevne in je torej izbira ustrezne NoSQL podatkovne baze že v samem začetku izredno pomembna. Da pravilno odločitev sprejmemo že na začetku, je potrebno dobro poznavanje različnih NoSQL podatkovnih baz ter njihovih specifičnih lastnosti. Vsaka izmed baz je namreč ustvarjena za reševanje določenega tipa problemov. Razlike med njimi so ogromne, kar posledično zelo oteži morebitne migracije. Ustrezni principi načrtovanja programske kode lahko težave ob migracijah sicer omilijo, na tem mestu pa je zopet izredno pomembno dobro poznavanje značilnosti in namembnosti vsake izmed NoSQL podatkovnih baz.

Diplomska naloga je razdeljena na štiri poglavja. Prvo poglavje – *Zakaj NoSQL?* – predstavlja tematiko NoSQL podatkovnih baz in odgovori na vprašanje, zakaj se je s problemom sploh potrebno ukvarjati.

V drugem poglavju – *Pregled področja NoSQL podatkovnih baz* – se seznanimo z različnimi tipi NoSQL podatkovnih baz in s pomočjo njihovih lastnosti odkrijemo razlike med njihovimi funkcionalnostmi ter namembnostmi, ter posledično tudi glavne razloge za težave pri migracijah med njimi. Na tem mestu prav tako predstavimo glavne razloge za migracijo med NoSQL podatkovnimi bazami in predlagamo rešitve za premoščanje razlik v poizvedovanju med NoSQL podatkovnimi bazami.

V okviru tretjega poglavja predstavimo simulacijo razvoja sodobne spletne aplikacije, ki uporablja več NoSQL podatkovnih baz. Cilj te simulacije je prikaz težavnosti migriranja med NoSQL podatkovnimi bazami.

Četrto poglavje – *Testiranje migracije* – predstavi našo simulacijo, ki vključuje opis realizacije aplikacije na prvotni podatkovni bazi (MongoDB ) in migracije na ostale podatkovne baze.

V zaključnem poglavju so povzete naše ugotovitve in sklepi glede migracij med NoSQL podatkovnimi bazami.

# 2 ZAKAJ NoSQL?

Preden se poglobimo v problematiko razlik med NoSQL podatkovnimi bazami, moramo razumeti, zakaj konkretno so NoSQL podatkovne baze sploh nastale ter katere probleme rešujejo. Relacijske podatkovne baze dominirajo že več kot štirideset let, zato si bomo najprej pogledali njihovo zgodovino, okolje v katerem so nastale ter vse to primerjali s sedanjostjo.

V zadnjih štirih desetletjih se je interaktivna programska oprema bistveno spremenila. Nekdanji »spletni« sistemi so se razvili v današnje spletne in mobilne aplikacije, ki jih uporablja bistveno več ljudi kot nekoč.

Prav tako se je spremenila arhitektura aplikacij. Moderno spletno aplikacijo lahko, z ustreznim porazdeljevanjem bremena na več strežnikov, sočasno uporablja več milijonov uporabnikov. Tako je zmogljivost današnjih aplikacij v veliki meri odvisna od števila strežnikov.

Tehnologija relacijskih podatkovnih baz pa se medtem ni tako drastično spremenila. Relacijske podatkovne baze so bile ustvarjene v 70-ih letih ter posledično optimizirane za aplikacije, uporabnike in infrastrukturo tistega časa. Kljub temu da se niso bistveno spremenile, se danes vendarle množično uporabljajo. Zahteve sodobnih aplikacij se rešujejo s posebnimi tehnikami, kot so denormalizacija, predpomnjenje (angl. *caching* ) in deljenje (angl. *sharding* ). Toda s tem se izgubljajo ključne prednosti relacijskega modela, obenem pa se povečuje cena in kompleksnost takšnih sistemov.

Zaradi vsega naštetega pa so se razvile različne  ne-relacijske podatkovne baze, ki rešujejo najrazličnejše težave. Združene so pod imenom NoSQL, ki naj bi pomenil »ne le SQL« in ne »ne SQL«, kot nekateri zmotno verjamejo. [17]

## 2.1 Sprememba interaktivne programske opreme

### 2.1.1 Porast števila uporabnikov in spremenjena raba aplikacij

V zadnjih nekaj desetletjih je bistveno narastlo število uporabnikov. V 70-ih letih je, na primer, zelo malo aplikacij podpiralo 2000 uporabnikov – večina jih je sprejela bistveno manj. Posledično so le redke organizacije razvijale in uporabljale tako velike aplikacije (med njimi so bile predvsem večje banke in letalske družbe ). Današnje spletne aplikacije pa imajo lahko potencialno dve milijardi uporabnikov, ne glede na to, ali gre za socialno omrežje, računalniško igro, bančniški sistem ali drugo. Dandanes imajo le redke spletne aplikacije zgolj 2000 uporabnikov.

Spremenili pa so se tudi sami uporabniki. Ti so, na primer, nekoč določeno aplikacijo uporabljali ob predvidljivem delovnem času, zaradi česar takšnih sistemov ni bilo težko vzdrževati, saj je bilo dovolj časa za izvajanje načrtovanih posodobitev. Današnje spletne aplikacije pa so v rabi vsak dan v letu, štiriindvajset ur na dan. Uporabniki so aktivni ob različnih urah in različno pogosto: nekateri aplikacijo uporabljajo redno, spet drugi jo uporabijo le nekajkrat, večina pa se jih od aplikacije sploh nikoli ne odjavi. Število uporabnikov lahko skokovito naraste, nova spletna aplikacija pa lahko tako čez noč dobi tudi milijon novih uporabnikov.[5]

### 2.1.2 Spremenjen namen spletnih aplikacij

V 70-ih letih so bile interaktivne aplikacije namenjene predvsem avtomatizaciji ponavljajočih se pisarniških opravil, kot so različne transakcije in rezervacije. Vsi ti procesi so bili že ustaljeni in dobro znani. Danes pa spletne aplikacije nudijo veliko več. Njihova spremenljiva narava stalno ponuja nove načine rabe ter spreminja že ustaljene načine komunikacije, nakupovanja, oglaševanja in zabave. Posledično morajo biti tudi podatkovne baze dovolj fleksibilne, da zadovoljijo spreminjajoče se potrebe sodobnih aplikacij.

### 2.1.3 Sprememba v infrastrukturi aplikacij

V 70-ih letih se je splet šele začel razvijati, pomnilniki so bili dragi, mrežne povezave pa počasne. Dominantno vlogo so zato imeli centralizirani računalniki z majhnim notranjim ter večjim zunanjim pomnilnikom. Danes pa imamo hitre mrežne povezave in cenovno dostopen pomnilnik. Tako se ob vse večjih potrebah aplikacij pojavi dilema, kako naj se aplikacija širi. Na voljo imamo dve možnosti: širitev navzgor in širitev navzven.

Širitev navzgor je centralizirana rešitev, kjer po potrebi izboljšujemo računalnik na katerem se izvaja aplikacija. Kupimo več hitrejših centralnih procesnih enot, več pomnilnika in ostalih potrebnih stvari ter s tem na preprost način rešimo težave. Vendar je tak pristop zelo drag, saj cene računalnikov ne naraščajo linearno. Navsezadnje pa tudi ob neomejeni količini denarja obstaja meja, kako velik računalnik lahko sestavimo. Takšni veliki in dragi sistemi so tudi zelo kompleksni in težavni za vzdrževanje. Velik izziv pa predstavlja tudi zagotavljanje odpornosti na izjemne razmere in napake, saj si bomo le ob obilici denarja lahko privoščili redundanten računalnik takšne velikosti. Prav tako je tudi sam nakup novega strežnika oziroma posodobitev starega zelo pomembna odločitev, saj če bomo kupili preveč virov bomo porabili preveč denarja, če jih bomo kupili premalo, pa bo odzivnost sistema slaba in bo to negativno vplivalo na uporabniško izkušnjo.

Širitev navzven pa je distribuirana rešitev, kjer po potrebi dokupimo več preprostih, cenovno ugodnih računalnikov. Ta rešitev je boljša ne samo zaradi nižjih stroškov, ampak predvsem

zaradi fleksibilnosti, saj se lahko z dodajanjem oziroma odvzemanjem strežnikov enostavno prilagodimo potrebam aplikacije. Obenem so takšni sistemi odporni na napake, kar omogoča nenehno delovanje ne glede na izjemne razmere, kot so izpadi elektrike, okvara računalniške opreme, poplave, potresi in podobno. Omogočajo tudi posodobitev programske opreme brez prekinitve nudenja storitev, saj lahko posodabljamo strežnik za strežnikom. Zaradi vseh navedenih dejstev je distribuirano računanje vse bolj razširjeno in priljubljeno.[5]

## 2.2 Zastarelost relacijskih podatkovnih baz

Relacijske podatkovne baze se v zadnjih štirih desetletjih v svojem bistvu niso opazno spremenile. V sami osnovi transakcijski sistem predvideva centraliziran sistem; da se, torej, celotna podatkovna baza nahaja na enem (velikem ) strežniku. Posledično je ob večjih potrebah aplikacije širitev navzgor edina ne-invazivna rešitev, kar predstavlja precejšnjo oviro za sodobne spletne aplikacije.

Rigidnost relacijskih shem je velika ovira za sodobne aplikacije, saj je tipično nemogoče vnaprej natančno določiti shemo, kot to zahtevajo relacijske podatkovne baze. Aplikacije pa se s časom razvijajo in spreminjajo, kar pomeni, da je sčasoma potrebno spremeniti tudi shemo. To pa je precej zahteven postopek, ki skoraj zagotovo vključuje začasno nedosegljivost aplikacije.

Obstaja pa nekaj tehnik, s katerimi se lahko izognemo širitvi navzgor, vendar s tem izgubimo določene funkcionalnosti, oziroma povečamo kompleksnost samega sistema.

### 2.2.1 Deljenje (angl. *sharding* )

Tehnologija relacijskih podatkovnih baz s svojim transakcijskim sistemom v sami osnovi predvideva centraliziran sistem, torej, da se celotna podatkovna baza nahaja na enem strežniku. Deljenje se tako pogosto uporablja, kadar en sam strežnik ni zmožen obdelati vseh zahtevkov zaradi (pre ) velikega števila sočasnih uporabnikov. Deljenje je realizirano v aplikaciji sami, kjer ročno razdelimo podatke na več strežnikov. Na primer, podatki vseh uporabnikov, ki živijo v vzhodni Evropi, se nahajajo na enem strežniku in podatki vseh uporabnikov, ki živijo v zahodni Evropi, se nahajajo na drugem strežniku.

Deljenje lahko breme sicer uspešno porazdeli na več strežnikov, vendar pa ima tudi nezaželene posledice:

1. **Izgubimo nekatere zelo pomembne prednosti relacijskega podatkovnega modela.** Ne moremo delati stikov (angl. *join* ) med deli, kar pomeni, da če želimo, na primer, poiskati vse kupce, ki so opravili nakup v zadnjem letu, je to poizvedbo potrebno izvesti na vseh strežnikih in potem znotraj aplikacije združiti rezultate. Še večji

problem pa predstavljajo transakcije, kjer mora biti atomarna operacija izvedena nad podatki iz več delov (angl. *shard* ), kar je nemogoče izvesti brez eksternega transakcijskega sistema ali kompleksne logike znotraj aplikacije.

2. **Obstaja velika verjetnost ponovnega deljenja.**
   Deljenje je realizirano znotraj aplikacije, kar pomeni, da je potrebno spremeniti logiko v aplikaciji vsakič, ko je potrebno izvesti novo deljenje. Na primer, če je prej aplikacija morala vedeti, da so vsi podatki uporabnikov iz zahodne Evrope na strežniku X, potem mora sedaj aplikacija vedeti, da se vsi podatki uporabnikov, ki živijo v zahodni Evropi, natančneje v Franciji, Španiji, Portugalski ali Veliki Britaniji, nahajajo na strežniku Y.

3. **Sheme je potrebno vzdrževati na vseh strežnikih.**
   Ko se potrebe aplikacije spremenijo in želimo shranjevati dodatne nove informacije, je potrebno sheme posodobiti na vseh strežnikih, jih normalizirati, optimizirati in ponovno zgraditi tabele.

### 2.2.2 Denormalizacija

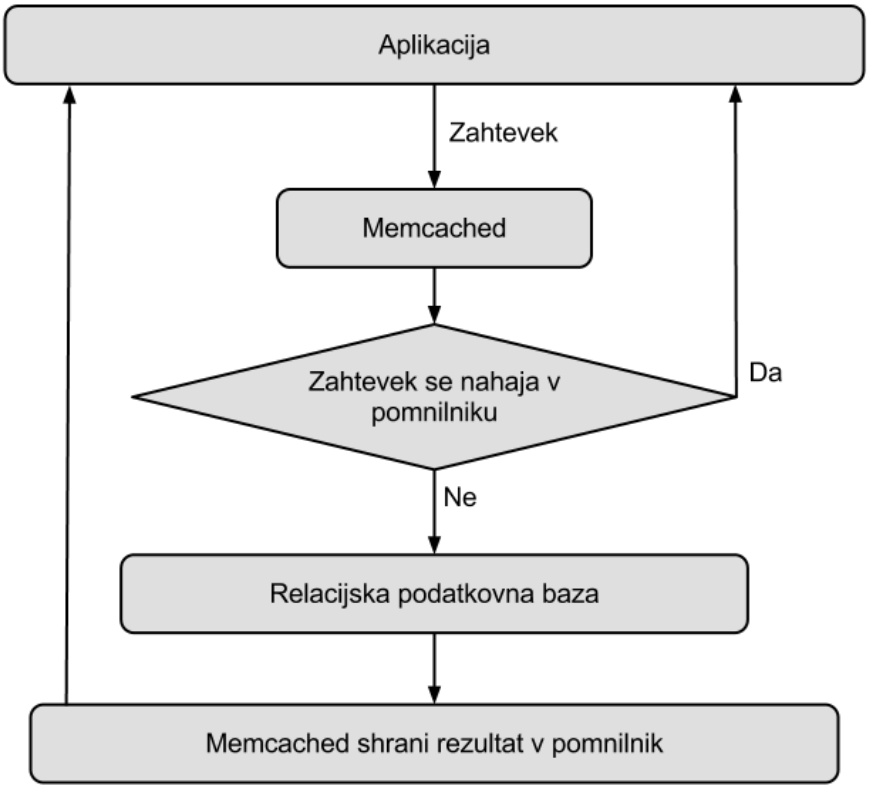
Preden shranimo podatke v relacijsko podatkovno bazo, moramo shemo točno določiti – torej kakšne podatke bomo shranjevali ter kakšni odnosi veljajo med temi podatki. Podatki so razbiti v tako imenovano normalno obliko, kar pomeni, da se nek zapis (gledano s stališča aplikacije ) v podatkovni bazi nahaja v več tabelah. Ko je potrebno zapis posodobiti, mora podatkovna baza zakleniti vse tabele, v katerih se ta zapis nahaja, kar pa resno omeji število sočasnih posodobitev.

Zaradi performančnih razlogov, sodobne aplikacije določene podatke pogosto hranijo v denormalizirani obliki. Tako so podatki tipično podvojeni, kar pomeni, da jih je potrebno posodabljati na več mestih hkrati. Ekstremen primer denormalizacije je hranjenje v ključ - vrednost obliki, kjer je primarni ključ povezan z binarnim zapisom. To sicer olajša deljenje in omogoča hiter razvoj, vendar pa je s tem izgubljena tudi skoraj vsa prednost relacijskih podatkovnih baz.

### 2.2.3 Distribuiran predpomnilnik (angl. *distributed cache* )

Precej učinkovita rešitev za povečanje zmogljivosti relacijskih baz je tudi uporaba distribuiranih predpomnilniških tehnologij, kot je na primer Memcached [15]. Pri tem gre za vmesni sloj med aplikacijo in relacijsko podatkovno bazo. Deluje kot posrednik, ki hrani zapise v pomnilniku, do katerih se je dostopalo nazadnje, na enem ali več strežnikih. Ko aplikacija zahteva nek zapis od relacijske baze, ta zahtevek Memcached prestreže in preveri

ali se zapis morda nahaja kje v pomnilniku med nazadnje obiskanimi zapisi. Če se iskani zapis nahaja med temi zapisi, ga vrne, sicer pa posreduje zahtevek podatkovni bazi ter potem vrne dobljeni zapis, obenem pa ga shrani v pomnilnik za potencialni naslednji dostop.




**Slika 1: Shema, ki prikazuje uporabo in delovanje Memcached. Aplikacija pošlje zahtevek, ki ga prevzame Memcached. Memcached preveri ali se zahtevek nahaja v pomnilniku, če se ga takoj posreduje aplikaciji, sicer pa ga zahteva od relacijske podatkovne baze. Ko zahtevek dobi ga shrani v pomnilnik in posreduje aplikaciji.**

Danes Memcached uporablja velika večina najrazličnejših aplikacij in podjetij, med drugim tudi Wikipedia, Facebook, Twitter in Google. Skoraj vse sodobne spletne aplikacije, ki uporabljajo relacijske podatkovne baze že v samem razvoju aplikacije, integrirajo uporabo Memcached.

Memcached prinaša veliko koristi in bistveno izboljša zmogljivost relacijskih baz, vendar ima vseeno nekaj pomanjkljivosti:

1. **Aplikacije lahko od Memcached postanejo preveč odvisne.**

   Ponekod lahko že izpad enega samega Memcached strežnika pomeni veliko zgrešenih dostopov do Memcached, kar pomeni, da jih mora obdelati relacijska podatkovna

baza. Le-ta pa lahko zaradi prevelike obremenitve (pre ) počasi rešuje zahtevke, kar negativno vpliva na uporabniško izkušnjo aplikacije.

2. **Obvladovanje še enega nivoja arhitekture.**
   To prinaša dodatno kompleksnost, finančne stroške in povečuje možnosti za napake v aplikaciji.

3. **Pospešuje le bralne dostope in ne pisanja.**
   Memcached pospeši bralne dostope do zapisov tako, da zapise, do katerih se je dostopalo nazadnje, hrani v pomnilniku. Vsa pisanja izvaja relacijska podatkovna baza, saj bi v nasprotnem primeru (če bi jih izvajal Memcached ) ob izpadu električne energije prišlo do izgube podatkov. Posledično ta rešitev ne pomaga vsem aplikacijam, ampak le tistim, pri katerih je bistveno več bralnih kot pa pisalnih dostopov do relacijske podatkovne baze.

## 2.3 Značilnosti NoSQL podatkovnih baz

Vse prej omenjene tehnike jasno kažejo na preprosto dejstvo: relacijske podatkovne baze so marsikje, predvsem v primeru spletnih aplikacij, vsiljena in nenaravna rešitev. Obstoječi proizvajalci relacijskih podatkovnih baz vztrajajo pri razvoju relacijskih tehnologij, saj jim to prinaša velik dobiček. Posledično pa so določena podjetja predvsem za lastne potrebe sama razvila alternativne rešitve, kot so Dynamo (Amazon ) in BigTable (Google ). Od takrat je bilo razvitih še precej drugih NoSQL podatkovnih baz, ki so večinoma odprtokodne. Novonastale NoSQL podatkovne baze tudi dosti bolj ustrezajo potrebam sodobnih spletnih aplikacij. Ker so si med sabo zelo različne, so migracije med njimi zelo zahtevne, kljub temu pa imajo tudi nekaj skupnih lastnosti:

- **Ne potrebujemo sheme.**
  Podatke lahko shranjujemo v NoSQL podatkovno bazo, ne da bi predhodno definirali shemo. Posledično lahko kadarkoli spremenimo format podatkov, ki jih shranjujemo, brez da bi morali aplikacijo ali podatkovno bazo začasno onemogočiti za uporabo. To tako aplikaciji, kot tudi podjetju, daje veliko fleksibilnost.

- **Elastičnost (avtomatsko deljenje ).**
  NoSQL podatkovna baza avtomatsko porazdeli podatke med strežnike, brez posredovanja aplikacije. Strežnike lahko dodajamo ali odstranjujemo iz podatkovnega dela, brez da bi aplikacijo začasno onemogočili. Poleg tega večina NoSQL podatkovnih baz omogoča

podvajanje podatkov na različnih strežnikih, tako znotraj enega podatkovnega centra, kot tudi med različnimi podatkovnimi centri. Pravilno vzdrževana NoSQL podatkovna baza bi morala delovati 24 ur na dan, vse dni v letu, ne glede na izredne okoliščine, kot so naravne katastrofe, večje spremembe v aplikaciji in podobno.

- **Vgrajeno predpomnjenje (angl. *caching* ).**
  Napredne NoSQL podatkovne baze avtomatsko in za aplikacijo transparentno shranjujejo zapise, do katerih se je dostopalo nazadnje, v sistemskem pomnilniku. Pri relacijskih podatkovnih bazah pa je shranjevanje takšnih elementov tipično realizirano kot poseben nivo infrastrukture, ki ga je potrebno posebej razviti, namestiti na ločene strežnike in vzdrževati.

- **Podpora za distribuirane poizvedbe.**
  Deljenje negativno vpliva na zmožnost izvajanja kompleksnih poizvedb relacijskih podatkovnih baz, medtem ko NoSQL podatkovne baze lahko izvajajo poizvedbe neodvisno od števila strežnikov, na katerih se nahajajo podatki.[5]

# 3 PREGLED PODROČJA NoSQL PODATKOVNIH BAZ

V tem poglavju bomo spoznali osnovne tipe podatkovnih baz ter njihove znane predstavnike. Dobro poznavanje NoSQL podatkovnih baz je ključnega pomena za razumevanje razlik med NoSQL podatkovnimi bazami in za ustrezno ukrepanje ob morebitni migraciji. Poznavanje lastnosti posameznih podatkovnih baz in njihovih namembnosti pa je izjemno pomembno tudi za razumevanje dejstva, da je bila vsaka izmed njih ustvarjena za reševanje določenega tipa problemov.

Na primer, aplikacije za podporo pri odločanju potrebujejo predvsem podatkovne baze, ki nudijo posebne vrste indeksiranja. Spletne analitične aplikacije, aplikacije za finančno modeliranje in podobne aplikacije pa iščejo predvsem podatkovne baze, ki jim omogočajo distribuirano procesiranje velikih količin podatkov. Aplikacije, ki procesirajo spletne transakcije, stremijo k čim večji zanesljivosti. Spletne aplikacije, z ogromnim številom uporabnikov, kot so Facebook in Amazon, pa se odrekajo ACID standardom za voljo BASE, kar jim omogoča, da učinkovito strežejo ogromni količini uporabnikov.[34]

BASE teorem je kratica za osnovno dostopnost (angl. *Basically Available* ), mehko stanje (angl. *Soft-state* ) in eventualno konsistentnost (angl. *Eventual consistency* ). Osnovna dostopnost pomeni, da se lahko zgodi, da je trenutno dostopen le del podatkov, mehko stanje pomeni, da so v določenem trenutku podatki nekonsistentni, eventualna konsistentnost pa nakazuje, da bodo vsi podatki po določenem času postali konsistentni.[37]

Te razlike so eden izmed razlogov, zakaj so različne NoSQL podatkovne baze vse bolj popularne. Podobne so specializiranemu orodju, za razliko od SQL podatkovnih baz, ki so kot nekakšen švicarski nož. Ravno zaradi velikih razlik med ne-relacijskimi podatkovnimi bazami, so migracije med njimi še toliko bolj težavne.[34]

Podatkovne baze, ki so predstavljene v tem poglavju, so uporabljene tudi kasneje v praktični demonstraciji simulacije razvoja sodobne spletne aplikacije, kjer so realizirane tudi migracije med temi podatkovnimi bazami.

Poizvedovanje po NoSQL podatkovnih bazah se bistveno razlikuje od poizvedovanja po relacijskih podatkovnih bazah. Medtem ko se relacijske podatkovne baze med seboj le malo razlikujejo, so razlike med NoSQL podatkovnimi bazami ogromne, posledično pa so razlike med poizvedovanjem po NoSQL podatkovnimi bazami še večje. Kot bomo videli v nadaljevanju, so te razlike enostavno prevelike, da bi jih lahko zakrili z enotnim poizvedovalnim jezikom.

Zato poizvedovanje po NoSQL podatkovnih bazah poteka preko posebnih vmesnikov, ki so specifični za določeno podatkovno bazo. Veliko NoSQL podatkovnih baz omogoča

upravljanje s podatkovno bazo preko protokola HTTP, kar pomeni, da je lahko aplikacija napisana v poljubnem programskem jeziku in kljub temu uporablja podatkovno bazo. REST vmesniki so sicer univerzalni, vendar tipično precej neprijazni za hitro razvijanje programske opreme. Zaradi tega obstajajo knjižnice API, ki so specifične za določen programski jezik in olajšajo upravljanje s podatkovno bazo. Nekatere NoSQL podatkovne baze same nudijo vmesnike, medtem ko so drugje vmesniki nastali kot odprtokodni projekti.

Vmesniki pa ne skrijejo podrobnosti in delovanja NoSQL podatkovnih baz, ampak le olajšajo dostop do njih s tem, da nudijo integracijo s programskim jezikom, v katerem poteka razvoj aplikacije. Olajšave so tipično v obliki avtomatske oziroma polavtomatske serializacije in deserializacije objektov in v skrivanju podrobnosti protokola. Za uporabo vmesnika prilagojenega določenemu programskemu jeziku, pa je vseeno potrebno dobro poznavanje podatkovne baze. Potrebno se je zavedati arhitekture podatkovne baze, njenega ravnanja, funkcionalnosti, načina indeksiranja ter poizvedovanja, ipd. Za razliko od relacijskih podatkovnih baz, kjer je dovolj, da poznamo nekaj posebnosti izbrane podatkovne baze, je pri NoSQL podatkovnih bazah potrebno dobro poznavanje NoSQL podatkovne baze v ozadju.

V splošnem NoSQL podatkovne baze delimo v štiri skupine: dokumentne podatkovne baze, ključ – vrednost podatkovne baze, podatkovne baze z grafi in podatkovne baze na osnovi družine stolpcev.[14]

## 3.1 Dokumentne podatkovne baze

Glavna ideja dokumentnega modela je, da zamenja tradicionalno vrstico iz relacijskega modela z bolj fleksibilnim modelom – s tako imenovanim dokumentom. Vsaka dokumentna podatkovna baza pa to po svoje implementira. Pri vseh dokumentnih podatkovnih bazah predstavlja dokument množico podatkov, zapisano na nek standarden način. Pogosto uporabljeni standardi so XML, YAML, JSON in BSON, ponekod pa so v rabi tudi binarni formati, na primer PDF in Microsoft Office formati.[9]

Dokumentne podatkovne baze se zavedajo vsebine dokumentov, kar olajša uporabo kompleksnih struktur, kot so gnezdeni objekti in polja. Omogoča tudi poizvedovanje po poljubnih (vnaprej določenih ) atributih. Obenem pa je shema popolnoma fleksibilna in lahko zato dokumentu kadarkoli dodamo oziroma odstranimo želeni atribut.[30]

Različne implementacije nudijo različne možnosti grupiranja in organiziranja dokumentov. Nekatere izmed njih so zbirke (angl. *collections* ), hierarhije direktorijev, nevidni meta-podatki in oznake (angl. *tags* ). Najpogosteje so v rabi zbirke, ki so v primerjavi z relacijskimi podatkovnimi bazami podobne tabelam, dokumenti pa so posamezni zapisi v tej tabeli. Kljub tej podobnosti, pa je ta pristop dosti bolj fleksibilen od relacijskega, saj ima lahko vsak

dokument v zbirki različna polja, za razliko od relacijskega, kjer je tabela natančno definirana.

Dostop do dokumentov poteka preko unikatnega ključa, ki predstavlja dokument. Poleg tipičnega dostopa ključ – dokument, pa večina dokumentnih podatkovnih baz nudi tudi poizvedovalni jezik ali knjižnico API, preko katere je možno poizvedovati po dokumentih tudi preko njihove vsebine.

Na prvi pogled so si dokumentne podatkovne baze precej sorodne in pričakovati bi bilo enoten poizvedovalni jezik. Zato je prav presenetljivo, da za dokumentne NoSQL podatkovne baze standardiziranega poizvedovalnega jezika ni.

V razvoju naj bi bil skupen jezik UNQL, ki bi združeval vse NoSQL podatkovne baze. Razvijajo ga ustanovitelji podatkovne baze SQLite in ustanovitelji NoSQL podatkovne baze Couchbase [25]. Razvoj samega jezika je na začetku izgledal precej obetavno, sedaj, v času pisanja, pa je videti, kot da je popolnoma zamrl. Jezik še vedno praktično ne podpira niti ene podatkovne baze, ampak je ostal v razvoju, da bi podpiral SQLite in Couchbase. Projekt je sicer precej obetaven, vendar so avtorji najverjetneje opazili, da so razlike med NoSQL podatkovnimi bazami pogosto prevelike, da bi jih pokrili z enotnim poizvedovalnim jezikom.[12]

Posledično poizvedovanje po dokumentnih podatkovnih bazah poteka preko posebnih vmesnikov, ki so specifični za določeno dokumentno podatkovno bazo, ki pa tipično omogoča upravljanje tudi preko HTTP protokola. Format dokumentov je bolj ali manj standarden, tako da se dokumenti večinoma prenašajo v JSON formatu.

### 3.1.1 MongoDB

Ena izmed najznačilnejših dokumentnih podatkovnih baz je MongoDB, ki je odprtokodna NoSQL podatkovna baza napisana v C++. Omogoča tako širitev navzven, kot tudi funkcionalnosti podobne relacijskim podatkovnim bazam, na primer sekundarne indekse, porazdeljene poizvedbe in sortiranje. Uporabljanje dokumentnega podatkovnega modela omogoča veliko fleksibilnost pri razvoju aplikacij. [16]

Med glavne značilnosti podatkovne baze MongoDB lahko uvrstimo:

- **Dokumentni model**
  Podatki so shranjeni v BSON formatu, ki je podoben JSON formatu, torej omogoča dinamične sheme. Edini omejitvi sta velikost dokumenta ter nujno potrebno polje »_id«, ki vsebuje unikatno ne-gnezdeno vrednost. Polja oziroma celotne dokumente je možno gnezditi znotraj drugih dokumentov, kar omogoča predstavitev kompleksnih hierarhičnih

odnosov znotraj enega samega zapisa. Dokumenti so shranjeni v zbirkah, pri čemer lahko posamezna zbirka vsebuje poljubno število dokumentov, ki pa so si lahko med seboj popolnoma različni.[27]

- **Pokrite zbirke (angl. *capped collecions* )**
  Pokrite zbirke so posebne zbirke s fiksno velikostjo. Ohranjajo vrstni red vstavljanja, ko dosežejo predefinirano velikost pa se pričnejo obnašati kot krožna vrsta. Koristne so za specifične potrebe, kot so, na primer, beleženje dogodkov (angl. *logging* ).[27]

- **Indeksiranje**
  MongoDB omogoča pol-avtomatsko indeksiranje vseh polj znotraj dokumentov, poleg tega pa nudi tudi podporo za sekundarne indekse.[16]

- **Ad hoc poizvedbe**
  MongoDB omogoča poizvedovanje po poljih, znotraj razpona vrednosti in poizvedovanje z regularnimi izrazi. Poizvedbe lahko vrnejo celotne dokumente ali pa le določena polja iz dokumentov.[16]

- **Agregacija**
  MapReduce se uporablja za procesiranje večjih količin podatkov in agregacijo nad njimi.[16]

- **Replikacija**
  MongoDB omogoča gospodar – suženj (angl. *master – slave* ) replikacijo. Gospodar lahko torej izvaja pisanja in branja, suženj pa le branja. Suženj kopira podatke od gospodarja in se posledično uporablja le za branja in varnostno kopijo. Če gospodar odpove, pa lahko sužnji avtomatsko izvolijo novega gospodarja.[16]

- **Porazdeljevanje obremenitve (angl. *load balancing* )**
  MongoDB omogoča širitev navzven s pomočjo deljenja (angl. *sharding* ). Razvijalec izbere ključ za deljenje, ki določa, kako se bodo podatki razdelili znotraj množice. MongoDB potem avtomatsko razdeli podatke med različne gospodarje. Možna je enostavna uporaba več strežnikov, saj podatkovna baza sama skrbi za porazdeljevanje obremenitev med strežniki in za podvajanje podatkov, kar omogoča, da podatkovna baza prenese tudi fizično okvaro strežnikov.
  Avtomatsko konfiguracijo je možno z lahkoto namestiti, slednja pa nato omogoča enostavno dodajanje novih strežnikov.[16]

- **JavaScript na strežniški strani**
  Razvijalci lahko shranijo funkcije JavaScript na strežniku, kar nadomešča shranjene procedure.[16]

- **Shranjevanje dokumentov**
  MongoDB se lahko uporablja tudi kot navaden sistem za shranjevanje dokumentov (angl. *file* ). S tem dobimo učinkovit sistem, ki je odporen na napake, saj omogoča porazdeljevanje podatkov med strežnike.[27]

Poizvedovanje po MongoDB je zaradi napredne in uporabniku prijazne arhitekture zelo enostavno. MongoDB ima vgrajenih veliko funkcionalnosti, ki skrivajo detajle tipičnih NoSQL podatkovnih baz. Obenem pa podatkovna baza ne dela nepotrebnih predpostavk, ampak pusti, da uporabnik sam določi njeno delovanje, saj uporabnik tipično sprejme boljše odločitve. Tako uporabnik določi ključ za deljenje dokumentov znotraj zbirk, saj sam najverjetneje dobro ve, kakšni dokumenti se shranjujejo v določeni zbirki. Uporabnik prav tako določi, katera polja bo indeksiral, saj podatkovna baza avtomatsko indeksira le ključna polja.

Poizvedovanje se izvaja v JavaScriptu, za hiter razvoj aplikacije in nasploh pregledovanje podatkovne baze pa je možno pisanje poizvedb kar v posebni konzoli, kjer lahko z JavaScript-om upravljamo s podatkovno bazo.

Po sami zgradbi je MongoDB precej podobna relacijskim podatkovnim bazam. V MongoDB se dokumenti vedno shranjujejo samo v zbirke, kar je podobno relacijskim podatkovnim bazam, kjer se zapisi vedno shranjujejo v določeno tabelo. Vendar pri MongoDB zbirke ni potrebno predhodno definirati ali kreirati, saj za to poskrbi kar podatkovna baza. Vlogo stolpcev v relacijskih podatkovnih bazah igra v MongoDB element dokumenta. Shema dokumentov znotraj iste zbirke je lahko različna, za razliko od relacijskih podatkovnih baz, kjer ima vsak zapis znotraj tabele iste stolpce.

Poizvedujemo lahko po kateremkoli elementu znotraj dokumenta, tako po elementih preprostega tipa, kot tudi po kompleksnih tipih kot so polja, poizvedovanje pa je možno tudi po elementih znotraj ugnezdenega dokumenta.

Poizvedovanje je po funkcionalnosti zelo podobno poizvedovalnemu jeziku SQL. Tako je možno poizvedovanje s pomočjo operatorjev, kot so enakost in neenakost, vsebovanost, iskanje po nizih pa je možno tudi z regularnimi izrazi. Vgrajene so tudi naprednejše funkcionalnosti, kot so štetje, sortiranje po vrednosti elementa in vračanje unikatnih rezultatov. Rezultate poizvedbe je možno pregledovati po delih, določene dele pa je mogoče

tudi preskočiti ali pa samo omejiti število rezultatov. Vse te napredne funkcionalnosti v veliki meri zmanjšajo potrebo po pisanju funkcij MapReduce, ki so marsikomu tuje.[27]

MongoDB nudi tudi veliko uradno podprtih gonilnikov za različne programske jezike za dostop do podatkovne baze, poleg njih pa obstaja tudi veliko neuradnih gonilnikov. Te knjižnice API nudijo dobro integracijo z jeziki in z njimi je uporaba MongoDB podatkovne baze še preprostejša.

Spodaj je prikazana povezava med poizvedovalnim jezikom SQL in poizvedovanjem po MongoDB podatkovni bazi.

**Stavki Update**

| **Stavki SQL** | **Stavki MongoDB** |
|---|---|
| UPDATE recipes<br>SET status = “CALORIE-DENSE”<br>WHERE calories > 800 | db.recipes.update(<br>{ calories: { $gt: 800 } },<br>{ $set: { status: “CALORIE-DENSE” } },<br>{ multi: true }<br>) |
| UPDATE recipes<br>SET score = score + 2<br>WHERE status = “LOW-CALORIE” | db.recipes.update(<br>{ status: “LOW-CALORIE” },<br>{ $inc: { score: 2 } },<br>{ multi: true }<br>) |

**Slika 2 Primerjava stavkov Update med SQL in MongoDB [23]**

**Stavki Select**

| **Stavki SQL** | **Stavki MongoDB** |
|---|---|
| SELECT *<br>FROM recipes | db.recipes.find() |
| SELECT name, calories<br>FROM recipes | db.recipes.find(<br>{ },<br>{ name: 1, calories: 1 }<br>) |
| SELECT *<br>FROM recipes<br>WHERE calories > 300<br>AND calories <= 600 | db.recipes.find(<br>{ calories: { $gt: 300, $lte: 600 } }<br>) |
| SELECT *<br>FROM recipes<br>WHERE name like “%cake%” | db.recipes.find(<br>{ name: /cake/ }<br>) |
| SELECT *<br>FROM recipes<br>WHERE name like “cake%” | db.recipes.find(<br>{ name: /^cake/ }<br>) |
| SELECT *<br>FROM recipes<br>WHERE score = 5<br>ORDER BY calories ASC | db.recipes.find( { score: 5 } ).sort( { calories: 1 } ) |

**Slika 3 Primerjava med poizvedovanjem z jezikom SQL in poizvedovanjem po podatkovni bazi MongoDB [23].**

**Stavki Delete**

| Stavki SQL | Stavki MongoDB |
|---|---|
| DELETE FROM recipes<br>WHERE score = 1 | db.recipes.remove( { score: 1 } ) |
| DELETE FROM recipes | db.recipes.remove() |

**Slika 4 Primerjava med odstranjevanjem zapisov s pomočjo SQL in MongoDB**

Kot je razvidno iz preglednic, je možno upravljati z MongoDB na način, ki je zelo podoben jeziku SQL.

### 3.1.2 CouchDB

Še ena izmed pomembnejših predstavnic dokumentnih podatkovnih baz je Apache CouchDB, oziroma na kratko kar CouchDB. Gre za odprtokodno NoSQL podatkovno bazo, ki shranjuje podatke v obliki dokumentov. Vsak dokument vsebuje svoje podatke ter podatke o svoji strukturi. CouchDB nudi zanimive in napredne funkcionalnosti, ki pa so težje razumljive kot pri MongoDB.

Shranjevanje dokumentov v CouchDB je enostavno. Za shranjevanje namreč ni potrebno vnaprej definirati sheme, niti ne določiti, kam naj se dokument shrani. Edino kar določimo je ime podatkovne baze, kamor želimo dokument shraniti. Pri CouchDB pojem zbirk ne obstaja; podobno je tudi z indeksi, nad katerimi uporabnik nima neposrednega nadzora. Poraja se vprašanje, kako se potem izvaja poizvedovanje, če so vsi dokumenti nestrukturirano shranjeni, obenem pa ne moremo neposredno določiti indeksov.[32]

V nadaljevanju si oglejmo bistvene značilnosti podatkovne baze CouchDB:

- **Dokumentni model**
  Vsi podatki so shranjeni v obliki dokumentov. Dokument sestavlja množica parov ključ-vrednost, ki je realizirana v JSON formatu, kar pomeni, da sheme ni potrebno vnaprej definirati. Vrednosti so lahko preprostega tipa, kot so številke, nizi znakov, datumi, lahko pa so tudi kompleksni, kot na primer asociativna polja in urejeni seznami. Vsakemu shranjenemu dokumentu CouchDB dodeli še dodaten par vrednosti, ki predstavlja unikatno identifikacijsko številko.[6]

- **Pogledi (angl. *views* )**
  Shranjene dokumente strukturiramo s pomočjo tako imenovanih pogledov. Vsak pogled je zgrajen kot funkcija Map / Reduce, napisana v JavaScriptu. Funkcija vsak dokument transformira v neko vrednost, ki jo tudi vrne in potem shrani. CouchDB indeksira poglede in jih tudi ustrezno posodablja, ko so novi dokumenti dodani, spremenjeni ali posodobljeni. Z njimi lahko povzamemo vsebino dokumentov, ki se nahajajo v podatkovni bazi. So popolnoma neodvisni od dokumentov nad katerimi izvajajo funkcijo pogleda, prav tako pa tudi ni nobene omejitve glede tega, v koliko pogledih se lahko posamezen dokument nahaja. Pogledi so shranjeni kot tako imenovani dizajn dokumenti, za njihovo sinhronizacijo in varnost pa skrbi podatkovna baza na enak način, kot skrbi za ostale navadne dokumente.

  Ob prvi rabi pogleda CouchDB pregleda vse dokumente v podatkovni bazi in nad njimi izvrši funkcijo pogleda, njene rezultate pa shrani v posebno tabelo kot ključ – vrednost zapis. Nato zgrajeno tabelo transformira v drevo B. Posledično lahko ta proces traja zelo dolgo. Dejansko trajanje pa je odvisno od velikosti podatkovne baze in kompleksnosti funkcije pogleda. Vendar je že ob naslednji rabi pogleda rezultat dostopen skoraj instantno, saj so rezultati shranjeni v drevesu B. Ob spremembi dokumenta se funkcija pogleda izvrši samo nad spremenjenim dokumentom in ustrezno posodobi drevo B. [6]

  Obstajajo trajni in začasni pogledi; zgoraj opisani pogledi so trajni. Začasni pogledi pa so tisti, ki se izbrišejo, ko niso več v rabi, kar pomeni, da jih je potrebno zelo pogosto na novo graditi. Zaradi tega so začasni pogledi skrajno neučinkoviti in tako primerni zgolj za testiranje.[32]

- **Eventualna konsistenca**
  Po določenem času je zagotovljeno, da se bodo vse spremembe med različnimi strežniki ustrezno propagirale in bodo vsi strežniki vsebovali enake podatke.[6]

- **Več verzij**
  CouchDB uporablja več verzij dokumentov, kar pomeni, da podatkovne baze med pisanjem ni potrebno zakleniti. To omogoča veliko propustnost pisanj in branj. Zaradi tega pa lahko pride do konfliktov, ki jih mora razrešiti aplikacija. [6]

- **ACID**
  CouchDB ustreza ACID standardom, to pa je realizirano s pomočjo več verzij dokumentov. [6]

- **Distribuirana arhitektura in podpora za delo s prekinjeno mrežno povezavo**
  CouchDB omogoča sinhronizacijo med različnimi podatkovnimi bazami, tudi če so le-te bile brez povezave. Kar pomeni, da lahko imajo različne replike različne kopije istih podatkov, ki jih lahko potem spremenijo in sinhronizirajo kasneje. Takšna replikacija je možna med gospodarjem in sužnji ter med dvema ali več gospodarjema.[6]

- **REST knjižnica API**
  Poizvedovanje se izvaja v JavaScript preko REST knjižnice API. REST knjižnica API omogoča, da lahko podatkovno bazo uporablja kdorkoli s pomočjo HTTP protokola, torej preko POST, GET, PUT in DELETE klicev. [32]

- **Administracija**
  Upravljanje s podatkovno bazo je možno preko spletnega vmesnika Futon, preko katerega lahko med drugimi urejamo varnostne nastavitve, pregledujemo podatke ter spreminjamo in dodajamo poglede.[32]

## 3.2 Podatkovne baze na osnovi grafov

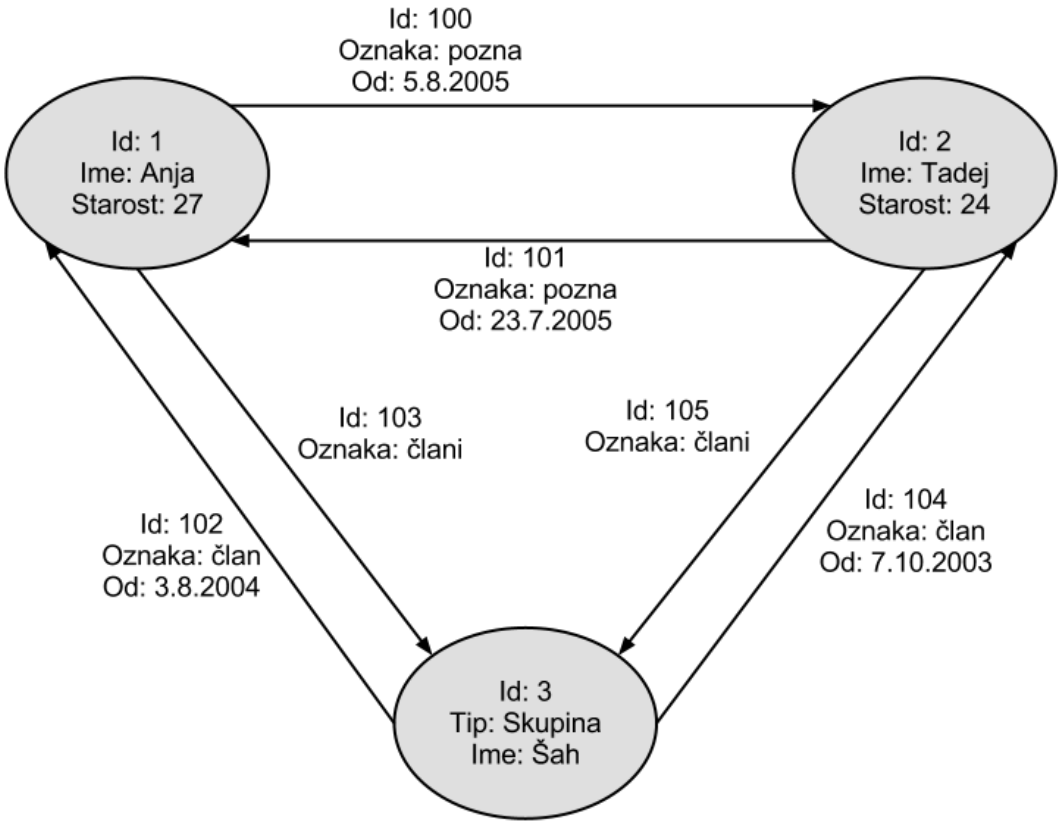
Druga večja skupina NoSQL podatkovnih baz, ki jo bomo na tem mestu omenili, so podatkovne baze na osnovi grafov. Podatkovna baza, ki ima za osnovo graf, je po definiciji vsaka podatkovna baza, ki nudi dostop, ki ni vezan na indeks. To pomeni, da ima vsak element kazalec na sosednji element ter indeksi tako niso potrebni. Te podatkovne baze uporabljajo posebne strukture, ki so značilne za grafe. Tako z vozlišči, povezavami in lastnostmi predstavljajo in shranjujejo podatke. Med seboj se v osnovi delijo v dve podskupini, in sicer na podatkovne baze, ki lahko shranijo kakršnekoli grafe, ter na specializirane baze, ki lahko shranijo le določene vrste grafov, na primer shrambe trojčkov (angl. *triplestore* ) in mrežne podatkovne baze. [10]

Podatkovne baze na osnovi grafov se od ostalih podatkovnih baz precej razlikujejo. Na nek način so še najbolj podobne dokumentnim podatkovnim bazam, vendar imajo še dodatne lastnosti, ki dokumente med seboj povezujejo. Podatki so tipično shranjeni v RDF formatu, ki predstavlja podatke s pomočjo koncepta »subjekt, predikat in objekt«. Ta format je namenjen predvsem standardizaciji opisa spletnih virov, ki omogoča poizvedovanje tudi računalnikom.

Zaradi njegove razširjenosti in primernosti za uporabo pri podatkovnih bazah z grafi, ga le-te pogosto uporabljajo. Zaradi pogoste rabe RDF formata pa je nastalo tudi nekaj poizvedovalnih jezikov nad RDF formatom, med katerimi je daleč najbolj razširjen poizvedovalni jezik SPARQL. [20, 22]

### 3.2.1 Struktura podatkovnih baz na osnovi grafov

Njihova struktura temelji na teoriji grafov, torej na vozliščih, povezavah in lastnostih. To je precej podobno objektno usmerjeni paradigmi. Vozlišča predstavljajo entitete, na primer osebe, ki jih hranimo v podatkovni bazi. Lastnosti so informacije, ki so vezane na določeno povezavo, oziroma odnos med dvema entitetama. Povezave pa so vezi, ki povezujejo vozlišča z drugimi vozlišči ter z lastnostmi. Najpomembnejše informacije so shranjene ravno na povezavah, vendar le-te pridejo do izraza predvsem ob analizi večjega grafa, kjer se pogosto jasno vidijo zanimivi vzorci. [10]




**Slika 5: Primer grafa. Vozlišča so entitete, usmerjene povezave med vozlišči kažejo na odnos med vozliščema, na povezavah pa so lastnosti, ki povedo več o tipu povezave.**

### 3.2.2 Shrambe trojčkov (angl. *triplestore* )

Shrambe trojčkov so posebna vrsta podatkovnih baz na osnovi grafov, ki je namenjena shranjevanju trojčkov. Trojček je sestavljen iz subjekta, predikata in objekta, na primer »Janez pozna Majo«. Poizvedovanje po teh podatkovnih bazah je optimizirano za iskanje trojčkov, posledično pa za to obstajajo tudi posebni poizvedovalni jeziki. Pogosto pa podatkovna baza trojčke sprejema oziroma vrača v standardnem formatu, najpogosteje v RDF formatu. RDF (angl. *Resource Description Framework* ), je meta podatkovni model. Uporablja se za

konceptualen opis informacij, ki so tipično vezane na splet, ter na sploh za predstavitev določenih vrst znanj, kot so, na primer, odnosi. [10, 24]

### 3.2.3 Značilnosti podatkovnih baz na osnovi grafov

V primerjavi z relacijskimi podatkovnimi bazami so podatkovne baze z grafi bistveno hitrejše pri poizvedovanju po asociativnih podatkih. Obenem se tudi bolje prilegajo objektno usmerjenim aplikacijam. Tipično se lažje širijo, saj za poizvedovanje niso potrebni časovno in prostorsko potratni stiki (angl. *join* ). Fleksibilna shema omogoča lažje ad-hoc poizvedbe in lažje spreminjanje podatkov. Te podatkovne baze so primerne predvsem za poizvedovanje po grafih, kot so, na primer, iskanje tipa »kdo koga pozna« ali pa »iskanje najkrajše poti« in podobno.

### 3.2.4 BrightstarDB

BrightstarDB je komercialna NoSQL podatkovna baza, namenjena predvsem za .NET platformo. Podatke shranjuje kot posebne dokumente v RDF formatu.

BrightstarDB Entity Framework je visokonivojsko ogrodje za dostop do podatkovne baze BrightstarDB. Poizvedovanje poteka preko LINQ vmesnika, ki pa je precej omejen ter v času pisanja diplomske naloge ne podpira marsikaterih funkcionalnosti. Ker poteka vse preko LINQ vmesnika, so vse poizvedbe močno tipizirane, kar po svoje bistveno pospeši in olajša razvoj aplikacije.

Za BrightstarDB so značilne naslednje lastnosti:

- **Fleksibilen W3C RDF format**
  BrightstarDB uporablja RDF format ter omogoča poizvedovanje tako preko vmesnika SPARQL, kot tudi preko integracije s programskim jezikom C#. Vmesnik SPARQL je popolnoma kompatibilen s standardizirano različico jezika SPARQL 1.1, kot ga je določil W3C (angl. *World Wide Web Consortium* ) – mednarodna organizacija za standardizacijo interneta. Pred shranjevanjem podatkov ni potrebno definirati sheme, saj ima lahko vsak dokument svojo shemo. RDF model uporablja trojčke za predstavitev podatkov, le-ti pa sestavljajo asociativni model. Vsak trojček povezuje lastnost z neodvisnim virom. Dostop do posamezne entitete je predstavljen z njej lastno vrednostjo URI. Zbirka RDF modelov predstavlja usmerja graf, ki omogoča združevanje podatkov iz različnih virov, ne da bi za to skrbela aplikacija.

- **Transakcije**
  BrightstarDB ustreza ACID standardom za transakcije. Pisanja v bazo nikoli ne blokirajo branj, branja pa nikoli ne blokirajo posodobitev, obenem pa branja nikoli ne dostopajo do nepopolnih pisanj.

- **Avtomatsko indeksiranje**
  Podatkovna baza že ob pisanju v podatkovno bazo avtomatsko indeksira vse odnose ter tudi drugi vrednosti.

- **Zgodovina**
  BrightstarDB nikoli ne prepiše podatkov, kar omogoča dostop do podatkov, kot so izgledali v preteklosti. Možno je tudi poizvedovanje po podatkih, kot so izgledali v katerikoli točki v času, ter vračanje teh podatkov nazaj v to preteklo stanje. Če je potrebno, pa se zgodovina lahko tudi pobriše in s tem sprosti podatkovno bazo.

- **SPARQL**
  BrightstarDB implementira poizvedovalni jezik SPARQL, ter omogoča dostop do baze tudi preko SPARQL vmesnika, kar posledično omogoča univerzalno rabo podatkovne baze.

- **Administracija in poizvedovanje**
  Polaris je orodje za nadzor nad podatkovno bazo BrightstarDB. Omogoča hitro pisanje in testiranje SPARQL poizvedb, kar olajša razvoj aplikacij in nadzor nad podatkovnimi bazami.

BrightstarDB nudi še tri načina poizvedovanja, in sicer s pomočjo knjižnic API za programski jezik C#. Med seboj se te knjižnice razlikujejo po nivoju fleksibilnosti in podrobnosti v nadziranju podatkovne baze.

Prva izmed njih je RDF Client API, ki nudi nizkonivojski dostop do podatkovne baze ter ne zagotavlja močne tipiziranosti. Poizvedovanje poteka preko poizvedovalnega jezika SPARQL, ki ga v obliki niza znakov podamo kot argument ustrezni funkciji.

Data Object Layer pa je samo ovojnica nad nižje ležečo RDF plastjo. Omogoča delo z močno tipiziranimi objekti, vendar obenem nudi precej dinamike, saj skoraj vse funkcije sprejemajo nize znakov za argumente, kar omogoča dinamično grajenje poizvedb. Možno je tudi poizvedovanje s pomočjo poizvedovalnega jezika SPARQL, ki ga podamo kot niz znakov za argument ustrezni funkciji.

Dynamic API je tenka ovojnica nad Data Object Layer, ki dodatno olajša interakcijo med okoljem .NET in RDF formatom. Omogoča dinamično kreiranje objektov, brez vnaprej definirane sheme ali razredov, poizvedovanje pa zopet poteka s pomočjo SPARQL poizvedb.[3]

## 3.3 Ključ – vrednost podatkovne baze

Tretja večja skupina NoSQL podatkovnih baz, podatkovne baze ključ – vrednost, temeljijo na konceptu parov ključ – vrednost. Ta koncept je v programskih jezikih poznan kot razpršena tabela. Kljub temu da je ta podatkovni model zelo preprost, so te podatkovne baze zelo odzivne in propustne ter lahko vsebujejo ogromne količine podatkov.[33]

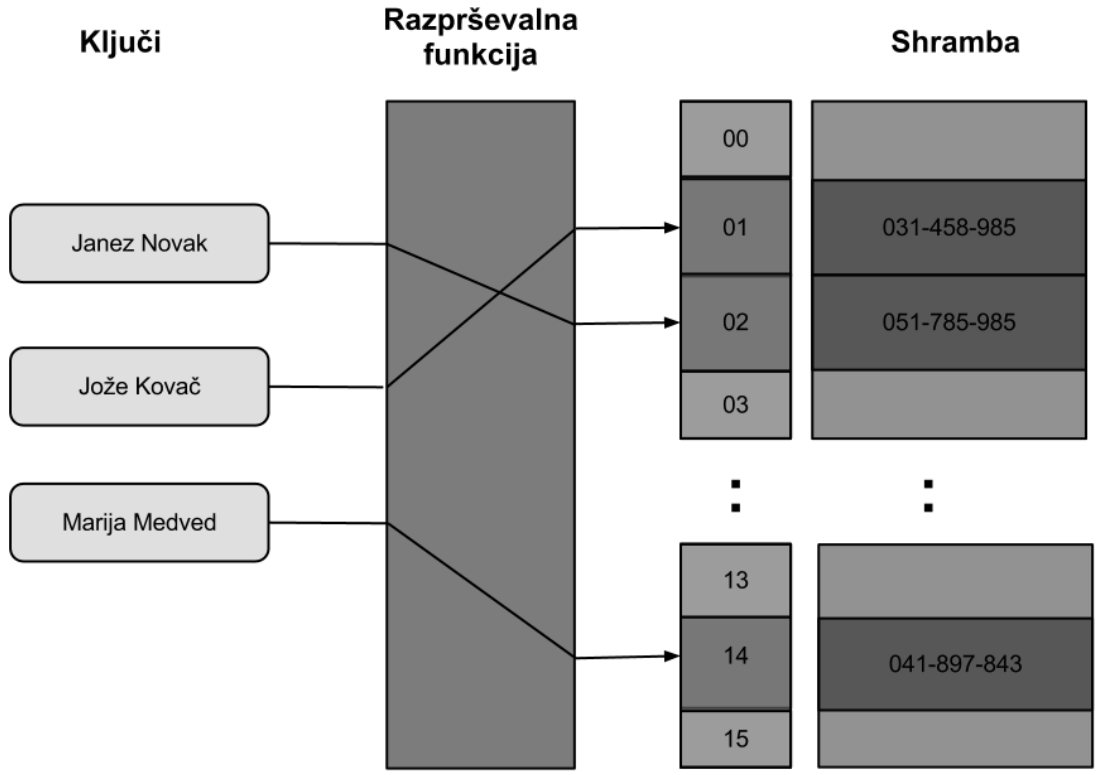
Ključ – vrednost podatkovne baze so tipično razvijalcu najmanj prijazne, saj predstavljajo samo osnovo vseh ostalih podatkovnih baz. Ne delajo velikih predpostavk v zvezi s tem, kako naj bi jih drugi uporabljali, ampak to raje prepustijo razvijalcem. Tako te podatkovne baze, v primerjavi z dokumentnimi podatkovnimi bazami, nudijo zelo malo funkcionalnosti. Obenem pa nudijo bistveno večjo fleksibilnost, saj morajo večji del funkcionalnosti, ki jih podpirajo dokumentne podatkovne baze, po potrebi implementirati razvijalci sami. Posledično ključ – vrednost podatkovne baze prenesejo več bremena in nudijo boljšo odzivnost. Ker večinoma držijo vse podatke v pomnilniku, se pogosto uporabljajo za predpomnjenje in obdelovanje začasnih podatkov [30].

Razlike med dokumentnimi podatkovnimi bazami ter ključ – vrednost podatkovnimi bazami se občutno poznajo pri poizvedovanju po ključ – vrednost podatkovnih bazah. Tako lahko velike razlike opazimo že v sami zasnovi podatkovnega modela, kjer je že vnaprej potrebno upoštevati, kako bomo kasneje poizvedovali po podatkovni bazi. Tudi same poizvedbe so precej drugačne, saj v sami osnovi operiramo samo z zelo omejenim naborom funkcionalnosti, ki se razlikujejo od ene do druge ključ – vrednost podatkovne baze. Poizvedovanje je tako tipično realizirano, da želene atribute in pripadajoče ključe shranimo v neki vnaprej določeni obliki, po kateri potem tudi dejansko poizvedujemo.[33]

### 3.3.1 Razpršena tabela

Razpršena tabela je podatkovna struktura, ki uporablja razprševalno funkcijo, s katero poveže tako imenovane ključe z njihovimi pripadajočimi vrednostmi. Razprševalna funkcija transformira ključ v indeks, na katerem se nahaja pripadajoča vrednost. Razpršena tabela shranjuje pare ključev in njihovih vrednosti. Ko želimo nekaj shraniti v razpršeno tabelo, podamo ključ, oziroma unikatno identifikacijo, ter njeno pripadajočo vrednost. Ko želimo to vrednost spet prebrati, enostavno podamo ključ, s katerim smo shranili vrednost. Vse te

operacije so zelo hitre (izvedejo se v konstantnem času ), saj je razprševalna funkcija konstantna.[11]




**Slika 6: Shema delovanja razpršene tabele za primer telefonskega imenika. Nad ključem (ime in priimek ) izračunamo razprševalno funkcijo s katero dobimo položaj kamor naj shranimo oziroma od koder naj dobimo pripadajočo vrednost (telefonsko številko ).**

### 3.3.2 Redis

Redis je odprtokodna in izjemno hitra NoSQL podatkovna baza, ki temelji na principu ključ – vrednost. V zadnjem času je vse bolj popularna zaradi svoje izjemne hitrosti in obilice funkcionalnosti. Tako je poizvedovanje po Redis podatkovni bazi bistveno bolj prijazno uporabniku. Sicer pa vse skupaj še vedno deluje na principu ključ – vrednost. Torej, če želimo nek podatek dobiti, moramo vedeti, s katerim ključem smo ta podatek shranili. Redis podpira tudi nekatere bolj kompleksne načine shranjevanja in vsak izmed njih nudi dodatne možnosti za poizvedovanje po podatkih.[18]

V nadaljevanju si torej podrobneje oglejmo poglavitne značilnosti podatkovne baze Redis:

- **Nahajanje vseh podatkov v pomnilniku**
  Redis shranjuje vse podatke v pomnilnik, kar omogoča izjemno hitra pisanja in branja. Prav tako pa nudi tudi možnost asinhronega shranjevanja podatkov na disk. Vsi podatki se morajo nahajati v pomnilniku, četudi so shranjeni na disku. To pomeni, da je podatkovna baza omejena z velikostjo pomnilnika. To naj bi se v prihodnjih verzijah spremenilo, ker naj bi Redis potem v pomnilniku imel le še bolj pogosto priklicane elemente.

- **Podatkovni model kot slovar**
  Podatkovni model je slovar, kjer vsakemu ključu pripada določena vrednost. Redis se od tipičnih slovarjev loči po tem, da lahko shranjujemo ne le nize znakov, ampak tudi druge podatkovne strukture.

- **Podatkovni tipi**
  - Nizi znakov
    Nizi znakov so osnovna vrednost v Redis-u. So binarno varni, kar pomeni, da lahko vsebujejo katerekoli podatke, na primer JPEG sliko ali serializiran objekt. Omejeni so na 512 megabajtov.
    Redis nad nizi znakov nudi veliko koristnih operacij, nekatere izmed njih so atomični števci in dodajanje k nizu znakov.
  - Seznami
    Seznami so seznami nizov znakov, urejeni po vrstnem redu vstavljanja. Seznam lahko vsebuje največ 2^32 – 1 elementov, kar je več kot 4 milijarde. Operacije nad elementi, ki so blizu začetka in konca seznama so hitre (konstantne ), medtem ko so operacije nad elementi iz sredine seznama počasnejše ( O(n) ).
    Redis nad seznami prav tako nudi koristne operacije. Nekatere izmed njih so tudi pridobitev zadnjih N vstavljenih elementov in krajšanje seznama na zadnjih N elementov dolžine. Pri seznamu je možno hitro dobiti, nastaviti ali odstraniti prvi oziroma zadnji element seznama. Prav tako je možno dobiti tudi vse elemente na določenem razponu znotraj seznama.
  - Množice
    Množice so neurejene zbirke nizov znakov. V konstantnem času je možno dodati ali odstraniti element ali pa preveriti, ali množica vsebuje element. Množica vedno vsebuje največ eno pojavitev elementa, kar posledično pomeni, da če v množico dodamo večkrat isti element, se bo le-ta v množici nahajal le enkrat. Možna sta tudi dostop do naključnega elementa iz množice ali pa do vseh elementov množice ter preverjanje vsebovanosti elementa v množici.

- Razpršene tabele
  Razpršene tabele povezujejo ključe in vrednosti, kar jih naredi koristne za predstavitev objektov. Razpršene tabele z malo elementi (do približno 100 ) so učinkovito implementirane in zasedejo zelo malo pomnilnika. Možen je dostop do vseh elementov razpršene tabele ali pa le do določenih elementov, ki se nahajajo pod podanim ključem.
- Urejene množice
  Urejene množice so posebna vrsta množic, pri katerih ima vsak element v množici pripadajočo oceno, s pomočjo katere je urejena množica po vrsti od elementov z najmanjšo oceno do tistih z najvišjo. Elementi znotraj množice morajo biti unikatni, ocene pa se lahko podvajajo. Omogočajo hitro (O(log N) ) dodajanje, odstranjevanje in dostop do vseh elementov znotraj določenega razpona, tudi do tistih na sredini, glede na indeks ali pa glede na oceno.[35]

- **Replikacija**
  Redis omogoča gospodar-suženj replikacijo. Podatki iz kateregakoli Redis strežnika se lahko podvojijo na katerokoli število sužnjev. Suženj je lahko gospodar drugemu sužnju. Redis sužnji omogočajo tako bralne, kot tudi pisalne dostope, kar dovoljuje namerno in nenamerno nekonsistenco med strežniki. Posledično je replikacija tipično koristna samo za večjo skalabilnost branj, ne pa tudi pisanj.

Kot je razvidno iz zgoraj omenjenega, je za kompleksnejše poizvedovanje pri tej podatkovni bazi odgovoren razvijalec sam. Podatke lahko na različne načine shranimo v podatkovno bazo, ter jih potem ustrezno prikličemo, filtriranje oziroma poizvedovanje pa izvedemo v aplikaciji. To se zdi sicer potratno, ker pa se Redis nahaja v pomnilniku, gre vseeno za hitro operacijo. Podatke lahko sicer tudi »indeksiramo« s pomočjo struktur Redis, kot je, na primer, seznam. Na primer, za realizacijo iskanja po določenih atributih, ki jih imajo določeni elementi v podatkovni bazi, lahko za vsak atribut naredimo seznam, v katerem hranimo ključe elementov, ki vsebujejo ta atribut. Poizvedovanje po atributih je potem preprosto sprehajanje po seznamu za ta določen atribut. Ta rešitev je hitrejša, vendar pa je prostorsko potratna. Z večjo porabo pomnilnika lahko »indeksiranje« še pohitrimo s pomočjo urejenih seznamov. Glede na to, da je Redis omejen z velikostjo pomnilnika, ki je še vedno precej drag in omejen vir, je potrebno dobro premisliti, ali je takšno indeksiranje resnično potrebno. Potrebno je izbrati pravo razmerje med procesorskim časom in porabo pomnilnika.

## 3.4 Podatkovne baze na osnovi družine stolpcev (angl. *column family* )

Četrto, in tudi zadnjo skupino NoSQL podatkovnih baz, prestavljajo podatkovne baze na osnovi družine stolpcev, ki so na prvi pogled podobne klasičnim relacijskim podatkovnim bazam. Pri relacijskih podatkovnih bazah imamo tabele sestavljene iz stolpcev, kjer tabela

predstavlja neko entiteto, stolpec njen atribut, vrstica pa konkreten primer / instanco te entitete. Stolpce moramo vnaprej določiti in vsi zapisi v tabeli, torej vse vrstice, vsebujejo vse stolpce. [36]

Podatkovne baze na osnovi družine stolpcev so redke večdimenzionalne razpršene tabele, sestavljene iz ključ-vrednost parov. Redke v tem kontekstu pomeni, da ni nujno, da ima vsak ključ (vrstica ) iste stolpce, kot jih imajo drugi, za razliko od relacijskih podatkovnih baz.[31] Večdimenzionalne pa pomeni, da prvo-nivojski ključ enolično določa zapis v podatkovni bazi, torej družino stolpcev, drugo-nivojski pa določa stolpec [36]. Vsak stolpec je sestavljen iz imena, ki je obenem ključ, s katerim dostopamo do stolpca, njegovo vrednostjo in datumom zadnje spremembe (angl. *timestamp* ), ki ga podatkovna baza interno uporablja za razreševanje konfliktov. [4]

Te podatkovne baze so ene izmed prvih NoSQL podatkovnih baz, za lastne potrebe pa so jih razvili spletni giganti kot so Google, Amazon in Facebook. Predstavljajo nadgradnjo navadnih ključ – vrednost podatkovnih baz, obenem pa predstavljajo poenostavitev relacijskih podatkovnih baz. Njihov glavni namen je visok nivo elastičnosti – zmožnosti, da podatkovna baza brez težav obdeluje ogromne količine podatkov in zahtevkov ob dovolj velikem številu (navadnih – angl. *comodity* ) računalnikov, na katerih je le-ta nameščena. Ravno zaradi tega nudijo zelo omejen nabor funkcionalnosti in večino dela prepustijo razvijalcem aplikacij, ki si lahko po svoje realizirajo dodatne funkcionalnosti, kot so napredno indeksiranje in podobno. Ravno zaradi velikega bremena, ki ga morajo nositi razvijalci aplikacij, ki želijo uporabljati nekatere napredne funkcionalnosti, je ta tip podatkovne baze tipično najbolj primeren za tiste aplikacije, ki jim takšen podatkovni model ustreza in pa za tiste, ki potrebujejo izjemno zmogljive podatkovne baze. Pri slednjih je potrebno dobro razumeti, da je govora o resnično velikih potrebah, kot jih imajo le redki, na primer Google, Facebook in Amazon. Pri manjših potrebah je možno brez večjih težav uporabljati tudi druge vrste podatkovnih baz.[31]

### 3.4.1 Cassandra

Ena izmed najpomembnejših in najbolj znanih podatkovnih baz na osnovi družin stolpcev je Apache Cassandra – odprtokodna NoSQL podatkovna baz, ki je distribuirana, decentralizirana, elastično skalabilna, visoko dostopna in odporna na napake. Njen podatkovni model temelji na Googlovem Bigtable, zasnova za distribucijski model pa na Amazonovem Dynamo.[31]

Med glavne značilnosti podatkovne baze Cassandra lahko uvrstimo:

- **Distribuiranost in decentraliziranost**
  Cassandra je zasnovana tako, da uspešno deluje ne samo na različnih računalnikih, ampak je optimizirana tudi za delovanje v različnih podatkovnih centrih, ter celo za situacije, ko se ena gruča (angl. *cluster* ) nahaja v geografsko ločenih podatkovnih centrih.[31]

  Podatki so razdeljeni znotraj gruče tako, da različna vozlišča hranijo različne podatke, ampak, ker ni nobenega gospodarja, lahko vsako vozlišče obdela katerikoli zahtevek. To pomeni, da ni ene same šibke točke (angl. *point of failure* ).[2]

  Zasnova enakovrednih vozlišč pa omogoča preprostejšo postavitev gruče, kot v primeru podatkovnih baz, ki temeljijo na principu gospodar-suženj.[31]

- **Elastičnost**
  Število sočasnih zahtevkov, ki jih lahko podatkovna baza obdela, narašča linearno z dodajanjem novih računalnikov. Spreminjanje števila vozlišč v gruči ne povzroča večjih motenj v delovanju aplikacije. Tako ni potrebno ponovno zagnati aplikacije ali spremeniti poizvedbe, niti ni potrebno ročno prestavljati podatkov na novo vozlišče.[31]

- **Odpornost na napake**
  Podatki so samodejno podvojeni na različna vozlišča, podprto pa je tudi podvajanje med različnimi podatkovnimi centri. Okvarjeno vozlišče lahko zamenjamo brez da bi to negativno vplivalo na delovanje aplikacije.[2]

- **Nastavljiva konsistentnost**
  Konsistentnost v osnovi pomeni, da branja vedno vrnejo najnovejšo različico zapisa. Cassandra omogoča nastavljivo konsistentnost s pomočjo replikacijskega faktorja in nivoja konsistence. Replikacijski faktor določa na koliko vozlišč se bo določena sprememba (dodajanje, spreminjanje, brisanje ) propagirala. Tako si lahko za ceno odzivnosti sistema zagotovimo višjo konsistenco. Nivo konsistence pa določa, koliko vozlišč mora potrditi določeno operacijo (branje ali pisanje ), da se le-ta šteje kot uspešna. [31]

- **Podatkovni model kot večdimenzionalna razpršena tabela**
  Cassandrin podatkovni model temelji na družini stolpcev. Poleg klasičnih stolpcev podpira tudi tako imenovane super stolpce, ki lahko vsebujejo druge stolpce. Vendar je uporaba slednjih precej nezaželena, obstajajo pa tudi govorice, da te funkcionalnosti v prihodnjih verzijah Cassandre ne bo več.

- **Prosti format**
  Preden začnemo shranjevati podatke, moramo določiti prostor ključev (angl. *keyspace* ), ki je v bistvu samo imenski prostor (angl. *namespace* ), ki združuje družine stolpcev in določene konfiguracije. Družine stolpcev so imena za asociirane podatke in njihov vrstni red. Samih stolpcev ni potrebno vnaprej določiti, ampak jih lahko sproti dodajamo po potrebi.[31]

- **Izjemna zmogljivost**
  Cassandra je bila že od začetka zasnovana tako, da dobro izkorišča večprocesorske / večjedrne računalnike in da se izvaja na velikem številu računalnikov razdeljenih v različnih podatkovnih centrih. V praksi je bilo dokazano, da se zelo dobro izkaže tudi ob velikih bremenih.[31]

- **Poizvedovanje**
  Cassandra nudi integracijo s Hadoop-om ter podporo za poizvedbe MapReduce. CQL (Cassandra Query Language) je SQL alternativa za podatkovno bazo Cassandra.[2]

Cassandra je odlična rešitev za aplikacije, ki jim ustreza podatkovni model na osnovi družine stolpcev ter za aplikacije, ki potrebujejo izjemno zmogljivo podatkovno bazo. Izjemna zmogljivost pa pride z veliko ceno – precej preprostim naborom funkcionalnosti. Kar pa pomeni, da imajo razvijalci aplikacij popolno svobodo, kako bodo določene stvari realizirali. Cassandra nudi le osnovno ključ – vrednost poizvedovanje po glavnem ključu ter po ostalih stolpcih, nad katerimi imamo indekse. Ne nudi pa podpore za primerjalne poizvedbe; na primer, če nas zanimajo vse vrednosti stolpca X, ki so večje od 5. Če bi želeli omogočiti takšne poizvedbe, bi morali vključiti v poizvedbo vsaj eno poizvedbo po direktni vrednosti indeksiranega stolpca, na primer: zanimajo nas vsi zapisi, kjer je stolpec Y enak 1 in stolpec X večji od 5. Za nekatere potrebe je to sprejemljivo, za druge pa spet ni. Lahko uporabimo tudi imena stolpcev za shranjevanje dejanskih vrednosti, kar nam lahko tudi malce olajša primerjalne poizvedbe. Včasih pa moramo sami razviti svoj sistem indeksiranja. Indekse lahko shranjujemo kot zapise v posebno družino stolpcev, ki si jo ustvarimo posebej za namene indeksiranja. Ta pristop seveda zahteva veliko dela, koliko konkretno, pa je odvisno od potrebe aplikacije. Večje spremembe v aplikaciji so iz istega razloga lahko precej zahtevne, saj moramo sami skrbeti za podrobnosti shranjevanja in poizvedovanja po podatkih. Na tem mestu je smiselno ponovno omeniti pomembnost čim bolj ustreznega izbora podatkovne baze za določeno problematiko, saj so različne podatkovne baze namenjene reševanju različnih problemov.

## 3.5 Razlogi za migracijo med NoSQL podatkovnimi bazami

Razlogov za migracijo med različnimi NoSQL podatkovnimi bazami je več in se jih je ob zasnovi aplikacije potrebno zavedati. Za večino vzrokov za migracijo so odgovorni ravno razvijalci, ki se na začetku napačno odločijo, bodisi zaradi prezgodnje optimizacije, bodisi zaradi napačnega razumevanja aplikacije. Včasih se spremenijo tudi potrebe uporabnikov, vendar tipično ne gre za drastične spremembe, ki jih izkušeni razvijalci ne bi mogli vnaprej predvidevati in se posledično že na začetku ustrezno odločiti. [30]

- **Razširitev aplikacije z novimi funkcionalnostmi**
  Tipično želijo razvijalci uporabljati podatkovno bazo, ki omogoča precej preprosto razvijanje aplikacije. Mogoče se določena podatkovna baza na začetku zdi primerna, saj veliko obljublja, vendar razvijalci sčasoma ugotovijo, da morajo določene funkcionalnosti, ki bi jih lahko imela podatkovna baza, realizirati kar sami. Ko to s časom preraste v dovolj velik problem, je pametno razmisliti še o drugih podatkovnih bazah. Primer tega prestavlja potreba po sinhronizaciji med dvema gospodar strežnikoma.

- **Težave pri razvoju in / ali rabi trenutne NoSQL podatkovne baze**
  Določena podatkovna baza se lahko na začetku izkaže za idealno, saj nudi hiter in dober razvoj aplikacije, vendar kasneje pri nadaljnjem razvoju aplikacije razvijalci naletijo na nepredvidene težave. Podatkovna baza ne deluje po pričakovanjih, posledično je potrebno dosti več dela vložiti za zagotavljanje želenega delovanja. Včasih problem predstavlja napačna rabe podatkovne baze, včasih pa podatkovna baza enostavna ne ustreza pričakovanjem razvijalcev. Primer je prevelika raba pomnilnika.

- **Neustrezen tip NoSQL podatkovne baze**
  Včasih se zaradi neizkušenosti razvijalci odločijo za napačen tip podatkovne baze. Na samem začetku razvijanja aplikacije se morda ne vidi jasno, kateri tip podatkovne baze je najbolj primeren za določeno aplikacijo. Potem pa se tekom razvoja vse bolj in bolj jasno vidi, kateri tip podatkovne baze bi bil primeren za določeno aplikacijo. To lahko predstavlja precej velik problem, saj lahko neizkušeni razvijalci na svoj način rešijo težave, ki jih je prinesla napačna izbira tipa NoSQL podatkovne baze. Problem pa se še poveča, ko je aplikacija v produkciji in se ob dovolj veliki obremenitvi izkaže, da je ročno reševanje težav, ki jih prinaša napačen tip NoSQL podatkovne baze, popolnoma neprimerno.

- **Neintuitivnost podatkovne baze**
  Izbrana podatkovna baza se morda na prvi pogled za določeno aplikacijo zdi zelo primerna. Nudi vse funkcionalnosti, ki jih aplikacija potrebuje ter je na sploh videti obetavno, vendar pa je raba podatkovne baze zelo neintuitivna in težavna. Na začetku se to morda zdi še sprejemljivo, toda tipično se prej ali slej izkaže, da je zaradi tega v aplikaciji bistveno več napak, razvoj aplikacije poteka počasneje ter posledično predstavlja tudi večje finančno breme.

## 3.6 Predlogi za premoščanje razlik med NoSQL podatkovnimi bazami

Razlike med NoSQL podatkovnimi bazami so smiselne, saj so različne podatkovne baze specializirane za reševanje različnih problemov. Od tod izhaja tudi ideologija »Uporabi pravo orodje za delo«, ki je zelo razširjena v skupnosti NoSQL.[30]

Najprej je potrebno zelo dobro poznavanje različnih tipov NoSQL podatkovnih baz, ter nekaj njihovih predstavnikov. Potrebno je dobro razumeti, čemu so različni tipi podatkovnih baz namenjeni, katere probleme rešujejo ter kakšne funkcionalnosti nudijo. Nato moramo dobro preučiti podatke, ki jih nameravamo shranjevati, da bi lahko čim bolje identificirali primeren podatkovni model, ki ne prinaša nepotrebne kompleksnosti. Obenem pa moramo določiti, kakšne poizvedbe potrebujemo, ter se zavedati, kako jih lahko z določeno podatkovno bazo realiziramo, saj le-to močno vpliva na zasnovo podatkovnega modela. Večinoma dokumentne podatkovne baze nudijo fleksibilen podatkovni model, po katerem je možno zelo napredno poizvedovanje, ključ – vrednost podatkovne baze so primerne predvsem za hitre in preproste operacije, podatkovne baze na osnovi grafov so primerne predvsem za domene, kjer so entitete enako pomembne kot odnosi med njimi, podatkovne baze na osnovi stolpcev pa so primerne predvsem za ogromne količine podatkov, z zelo velikim številom zahtevkov.[30]

Tako je izjemno pomembno ustrezno poznavanje trenutnih potreb aplikacije ter tudi sposobnost predvidevanja razvoja potreb aplikacije v prihodnosti. Z ustreznim izborom podatkovne baze se lahko izognemo, oziroma si vsaj bistveno olajšamo kasnejšo migracijo na drugo podatkovno bazo. Zelo pomembno je, da na začetku pravilno izberemo tip podatkovne baze, ki ga bomo uporabljali, saj so razlike med podatkovnimi bazami istega tipa bistveno manjše, kot pa med različnimi tipi. Posledično je tudi migracija na podatkovno bazo sorodnega tipa preprostejša kot med nesorodnimi tipi. Pri migraciji na soroden tip podatkovne baze je tipično potrebno narediti le manjše spremembe v načinu shranjevanja podatkov in poizvedovanju. Bolj ali manj gre za manjše razlike v funkcionalnostih podatkovnih baz. Navadno predstavlja večji problem nova knjižnica API za dostop do podatkovne baze.

Pri migraciji na nesorodno podatkovno bazo pa moramo večinoma spremeniti tudi sam podatkovni model, način shranjevanja podatkov in predvsem način poizvedovanja po

podatkih. Poizvedovanje se tu bistveno razlikuje že na konceptualnem nivoju, kar pomeni, da se še bolj izrazito razlikuje v sami implementaciji.

### 3.6.1 Programska ogrodja

Za različne programske jezike obstajajo različna ogrodja, ki naj bi olajšala razvoj aplikacij. Nekatera ogrodja med drugim nudijo dostop do relacijskih in NoSQL podatkovnih baz na relativno enoten način. Pomanjkljivosti teh ogrodij so predvsem v slabi integraciji z NoSQL podatkovnimi bazami ter v zelo omejenem naboru NoSQL podatkovnih baz, ki jih podpirajo. Zelo pogosto moramo sami implementirati večino stvari v povezavi s podatkovno bazo. Posledično so programska ogrodja priporočljiva rešitev le, če smo se za uporabo ogrodja odločili zaradi vzrokov, ki niso povezani s podatkovnimi bazami, predvsem pa ne z NoSQL podatkovnimi bazami. Izbor programskih ogrodij z namenom olajšati migracije med podatkovnimi bazami je precej slaba odločitev, saj celotna aplikacija temelji na ogrodju, medtem ko z dobrim načinom programiranja brez večjih težav lahko sami poskrbimo za svoje lastno specifično ogrodje, ki uravnava dostop do želenih NoSQL podatkovnih baz.

### 3.6.2 Ustrezni principi načrtovanja programske kode

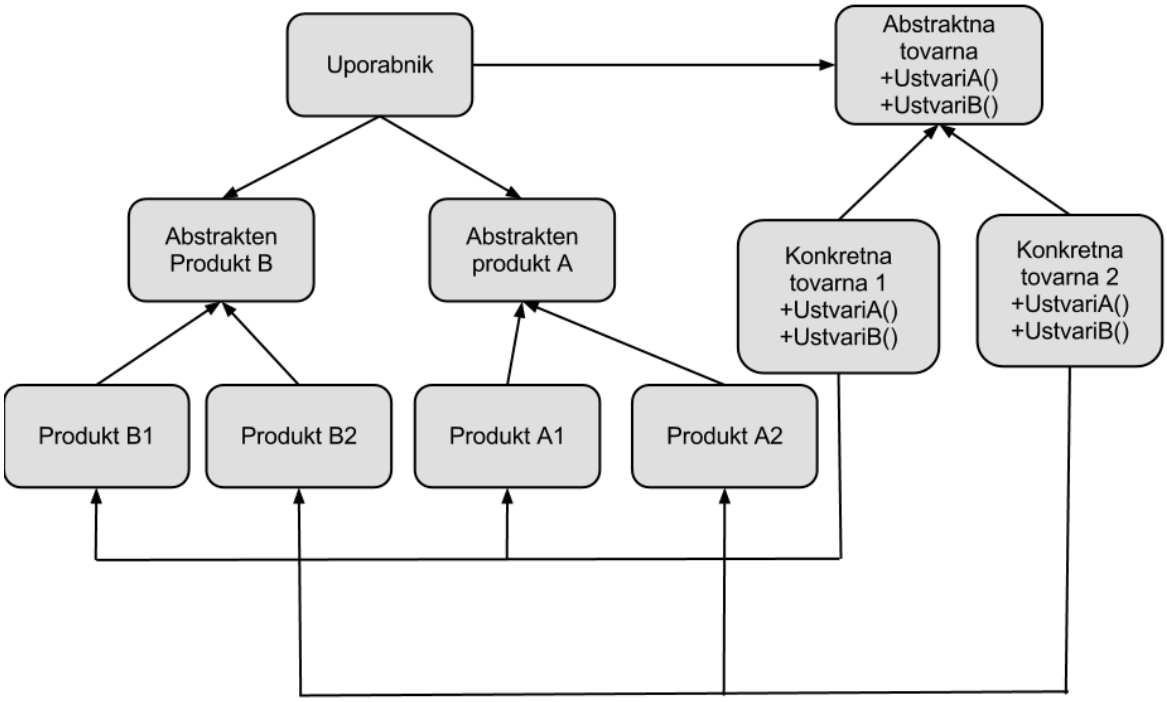
Ustrezni principi načrtovanja programske kode so ključnega pomena za pisanje uporabne kode. To se zelo jasno kaže pri migraciji med različnimi podatkovnimi bazami, še posebej jasno pa kadar migriramo med NoSQL podatkovnimi bazami, kjer so razlike res ogromne.

**Abstraktna tovarna**

Abstraktna tovarna je princip abstrakcije, katere namen je nuditi vmesnik za ustvarjanje družin povezanih ali odvisnih objektov, brez da bi točno določili konkretne implementacije, torej razrede.

Elementi abstraktne tovarne so naslednji: abstraktna tovarna, konkretna tovarna, abstrakten produkt, konkreten produkt in uporabnik.

Abstraktna tovarna deklarira vmesnik za operacije (funkcije ), ki ustvarijo abstraktne razrede (produkte ). Konkretna tovarna pa implementira operacije abstraktne tovarne, torej operacije, ki naredijo konkretne razrede. Abstrakten produkt je vmesnik, ki abstraktno opisuje objekt (produkt ), medtem ko je konkreten produkt implementacija abstraktnega produkta, obenem pa ta produkt ustreza produktu, ki ga ustvari konkretna tovarna. Uporabnik uporablja samo vmesnike, ki jih definirajo abstraktne tovarne in abstraktni produkti.[28]




**Slika 7: Primer sheme abstraktne tovarne. Uporabnik uporablja samo abstraktno tovarno in abstraktne produkte, v ozadju pa v resnici uporablja konkretno tovarno s konkretnimi produkti.**

Abstraktna tovarna nam ne samo olajša migracijo med podatkovnimi bazami, ampak nam lahko omogoča tudi simultano upravljanje z več različnimi podatkovnimi bazami, ne da bi s tem morali bistveno vplivati na logiko aplikacije.

# 4 SIMULACIJA RAZVOJA SODOBNE SPLETNE APLIKACIJE, KI UPORABLJA VEČ NoSQL PODATKOVNIH BAZ

Problematiko razlik, in posledično težav z migracijami, med NoSQL podatkovnimi bazami želimo bolje prikazati na praktičnem primeru razvoja dejanske aplikacije. S tem želimo nazorno ilustrirati tako razlike med NoSQL podatkovnimi bazami, kot tudi vzroke za le-te. Obenem pa želimo s pomočjo konkretne implementacije prikazati, kako olajšati migracijo na drugo NoSQL podatkovno bazo ter kako uporabljati več NoSQL podatkovnih bazah naenkrat.

V nadaljevanju sta tako predstavljena opis izbire ustreznega programskega jezika ter postopek namestitve posameznih NoSQL baz, ki smo jih potrebovali za prikaz migracije med njimi na primeru dejanske aplikacije. Temu sledi opis izvorne podatkovne baze ter predstavitev naše aplikacije, s katero smo preverjali svojo domnevo o težavnosti migracij med NoSQL podatkovnimi bazami.

## 4.1 Izbor programskega jezika in razvojnega okolja

Izbor programskega jezika za razvoj simulacije je relativno pomemben za testiranje migracije. Izbirali smo med JavaScript, Java in C#. JavaScript je v zadnjem času vse bolj popularen tudi kot strežniški jezik, vendar še vedno le majhen delež aplikacij uporablja JavaScript kot strežniški jezik. Ker je cilj diplomske naloge večini čim bližje predstaviti problematiko migracije, se za JavaScript nismo odločili. Java in C# sta si precej podobna, vendar C# po avtorjevem mnenju omogoča malce lepše in lažje programiranje. Glavni razlogi za izbor C# pa so hiter in konstanten razvoj jezika ter platforme .NET in Mono, ter posledično njegova vse večja popularnost.

Razvoj je potekal v okolju Windows (7 in 8), predvsem zaradi odličnega orodja za razvoj programske opreme Visual Studio. Zaradi preprostosti so tudi vse podatkovne baze bile nameščene v istem okolju, torej v okolju Windows. Z izjemo BrightstarDB, bi lahko celoten projekt razvijali tudi v okolju Linux s pomočjo platforme Mono. Računalnik, na katerem smo izvajali testiranje, je bil starejši PC (Intel Dual Core 2.5 GHZ, 4 GB RAM ), vendar je kljub vsemu popolnoma ustrezal potrebam tega testiranja.

## 4.2 Namestitev NoSQL podatkovnih baz

### MongoDB

Namestitev podatkovne baze MongoDB poteka hitro in enostavno. Podatkovno bazo smo enostavno sneli z omrežja, jo postavili v želeni direktorij ter ji ustrezno nastavili pot za shranjevanje podatkov. S tem je osnovno nameščanje MongoDB končano. Za uporabo v projektu pa smo dodali še ustrezne gonilnike MongoDB ter gonilnike za C# knjižnico API.

Uradna 10gen C# knjižnica API za dostop do MongoDB je precej preprosta za uporabo – enostavno jo dodamo v projekt. Omogoča nam preprosto uporabo že obstoječih razredov, saj bolj ali manj avtomatsko poskrbi za ustrezno serializacijo v (oziroma deserializacijo iz) formata JSON. Nudi tudi dobro integracijo z obstoječo infrastrukturo C#.[7]

**CouchDB**

Novejše različice podatkovne baze CouchDB se namestijo precej preprosto, saj ni potrebno posebej nameščati programskega jezika Erlang in ostalih potrebnih okolij za delovanje CouchDB. Vse to po potrebi namesti nalagalnik sam. CouchDB omogoča tudi preprosto registracijo z Windows Service.

Za dostop do podatkovne baze smo uporabili odprtokodno C# knjižnico API, znano pod imenom Hammock oziroma Relax. Pred uporabo knjižnico enostavno dodamo v projekt, obenem pa je potrebno dodati še knjižnico Newtonsoft JSON, ki skrbi za serializacijo in deserializacijo objektov. Knjižnica API nudi precej dobro integracijo z obstoječo infrastrukturo C#, vendar ne tako dobro in intuitivno kot MongoDB.[19]

**BrightstarDB**

BrightstarDB se namesti tako enostavno, kot navadna aplikacija za Windows. Omogoča enostavno integracijo z Windows service. Ker je ciljna platforma te podatkovne baze .NET, smo večinoma uporabljali uradni BrightstarDB Entity Framework, ki nudi dobro integracijo z programskim jezikom C#. Za bolj nizkonivojski dostop, pa smo uporabili že vključene knjižnice API, »Data Object Layer«, »Dynamic API« in »RDF Client API«. Bolj kot je knjižnica prijazna uporabniku in vključena v .NET paradigmo, več podrobnosti skrije in obratno. Posledično smo na različnih mestih uporabili različne knjižnice API.[3]

**Redis**

Redis uradno podpira samo UNIX sisteme, kar pa ne pomeni, da aplikacij za Redis ne moremo razvijati na okolju Windows. Obstaja nekaj odprtokodnih rešitev za izvajanje Redisa na platformi Windows. Pri tem je najbolj uspešna odprtokodna rešitev tako imenovana MSOpenTech Redis, ki je dejanska implementacija Redisa za okolje Windows. Razvoj je sproten in konstanten, ter podpira tudi najnovejše verzije Redisa. Sicer ni priporočljivo uporabljati Redis na okolju Windows v produkciji, kar pa ne pomeni, da ga ni priporočljivo uporabljati na okolju Windows za (hitrejši ) razvoj aplikacije. Tako smo tudi mi uporabljali MSOpenTech Redis, ki ga enostavno prenesemo s spleta v obliki (Visual Studio ) .sln projekta. Projekt vsebuje že vnaprej zgrajene komponente Redis, ki jih enostavno zaženemo.[18]

Za dostop do podatkovne baze Redis smo uporabljali vgrajeno konzolo Redis CLI, ki nudi nadzor nad podatkovno bazo. Za programski dostop do podatkovne baze Redis pa smo uporabili odprtokodno knjižnico API ServiceStack.Redis, ki nudi dobro integracijo s programskim jezikom C# in je tudi uradno priporočena knjižnica API za programski jezik C#.[21]

**Cassandra**

Cassandra je napisana v programskem jeziku Java, kar pomeni, da jo lahko uporabljamo na vseh sistemih, na katerih je nameščen JVM (Java Virtual Machine ). Razlike med sistemi Unix in Windows so samo v različnih konfiguracijskih datotekah, saj operacijska sistema uporabljata različno strukturo direktorijev. Za dostop do podatkovne baze smo uporabljali odprtokodne gonilnike podjetja Datastax, ki smo jih namestili kar preko sistema NuGet. [8]

Gonilniki nudijo odlično integracijo s programskim jezikom C#, zelo dobro integracijo z LINQ, ter pretvorbo poizvedb LINQ v poizvedovalni jezik CQL. Omogočajo pa tudi uporabo poizvedovalnega jezika CQL.

## 4.3 Izvorna podatkovna baza

Izvorna podatkovna baza naj bi bila podatkovna baza, ki se pogosto uporablja, je enostavna za uporabo in čim bolj vsestranska. Najbolj vsestranske podatkovne baze so ključ – vrednost podatkovne baze, vendar te nudijo le omejeno funkcionalnost, medtem ko dokumentne podatkovne baze nudijo bistveno več funkcionalnosti ter so še najbližje relacijskim podatkovnim bazam. Izbrali smo podatkovno bazo MongoDB, tako zaradi njene popularnosti, kot tudi zaradi pogoste rabe. Razloga za njeno popularnost pa sta predvsem preprostost uporabe in učinkovitost.

## 4.4 Predstavitev problemske domene

Problemska domena je zastavljena kot zelo okrnjena različica spletne strani z recepti. Zaradi poudarka na samem poizvedovanju in migraciji med NoSQL podatkovnimi bazami, se ne uporablja nobena avtentikacija, ampak je simuliran predvsem tisti del aplikacije, ki se nanaša na shranjevanje podatkov in poizvedovanje, logika aplikacije pa je simulirana le toliko, da lažje predstavi predstavljeno problematiko.

Omogoča naslednje funkcionalnosti:

- **Prijavljanje uporabnikov v sistem**.
  Aplikacija omogoča registriranje novih uporabnikov ter prijavljanje že registriranih uporabnikov. Zaradi preprostosti avtentikacija ni simulirana, posledično je za vsakega uporabnika shranjeno samo uporabniško ime in elektronski naslov.

- **Dodajanje osnovnih živil**
  Uporabniki lahko dodajajo osnovna živila, iz katerih so sestavljeni recepti.

- **Iskanje osnovnih živil**
  Uporabniki lahko pregledujejo informacije o osnovnih živilih

- **Dodajanje novih receptov**.
  Uporabniki lahko dodajajo nove recepte, ki so sestavljeni iz že obstoječih osnovnih živil.

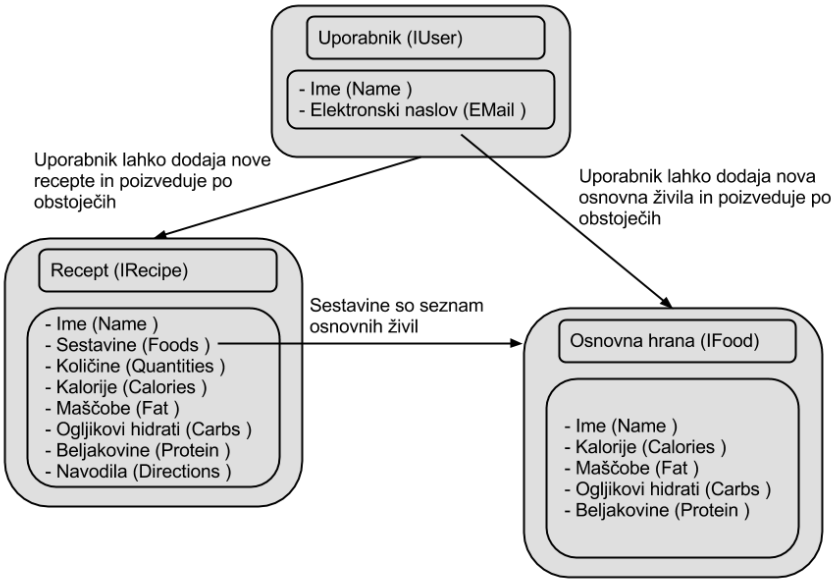
- **Poizvedovanje po receptih.**
  Omogočeno je iskanje po konkretnem imenu recepta (primer: recept z imenom »Skutna torta« ), iskanje po imenu recepta (primer: vsi recepti, ki vsebujejo ime torta ), iskanje po hranilnih vrednostih recepta (primer: vsi recepti, kjer je na 100g več kot 10g beljakovin ) in iskanje po sestavinah (primer: vsi recepti, ki vsebujejo skuto ).

- **Pregledovanje receptov.**
  Vsi uporabniki lahko pregledujejo recepte.

## 4.5 Podatkovni model

Na podlagi prej predstavljenih funkcionalnosti smo sestavili preprost podatkovni model, ki ga interno uporablja aplikacija. Aplikacija je zasnovana tako, da čim bolj olajša migracijo, zato je tudi celoten podatkovni model predstavljen le kot množica vmesnikov, ki jih potem ustrezno implementira vsaka uporabljena podatkovna baza posebej. Podatkovni model sestavljajo naslednji vmesniki[1]:

[1] Imena v oklepajih predstavljajo imena, ki so dejansko uporabljena v kodi.




**Slika 8 Podatkovni model, ki je bil uporabljen za testiranje migracij.**

## 4.6 Realizacija poizvedb

Vse poizvedbe so abstrahirane s pomočjo abstraktne tovarne. Definirane so kot funkcije, ki na vhodu sprejemajo ustrezne argumente ter vrnejo rezultate poizvedb v standardni obliki. Te funkcije ustrezajo prej definiranim specifikacijam aplikacije. Poizvedbe so torej:

- Iskanje uporabnika z določenim uporabniškim imenom (primer: uporabnik z uporabniškim imenom »abc123« )
- Iskanje osnovne hrane, glede na ime osnovne hrane (primer: osnovna hrana z imenom »skuta« )
- Iskanje po konkretnem imenu recepta (primer: recept z imenom »Skutna torta« )
- Iskanje po imenu recepta (primer: vsi recepti, ki vsebujejo ime »torta« )
- Iskanje po hranilnih vrednostih recepta (primer: vsi recepti, kjer je na 100g več kot 10g beljakovin )
- Iskanje po sestavinah recepta (primer: vsi recepti, ki vsebujejo skuto )

## 4.7 Uporabljena abstrakcija

Zaradi utemeljenih predvidevanj, da je migracija med NoSQL podatkovnimi bazami zahtevna operacija, smo se odločili za čim večjo abstrakcijo testne aplikacije, z namenom čim bolj olajšati migracije. Vse operacije vezane na podatkovno bazo so abstrahirane s pomočjo tako

imenovanega principa abstraktne tovarne. Ker ima vsaka podatkovna baza svoj način dostopa do podatkov, je ta abstrakcija zelo primerna.

Elementi abstraktne tovarne s pomočjo konkretnih primerov:

- **Abstraktna tovarna - IModelFactory**
  - Abstraktna tovarna deklarira vmesnik za operacije (funkcije ), ki ustvarijo abstraktne razrede (produkte ).
  - Primer: IModelFactory ustvari IUser, IFood, IRecipe.

- **Konkretna tovarna - MongoModelFactory**
  - Konkretna tovarna implementira operacije abstraktne tovarne, torej operacije, ki naredijo konkretne razrede.
  - Primer: MongoModelFactory ustvari MongoUser, MongoFood in MongoRecipe.

- **Abstrakten produkt**
  - Abstrakten produkt je vmesnik, ki abstraktno opisuje objekt (produkt ).
  - Primer: IUser, IFood, IRecipe.

- **Konkreten produkt**
  - Konkreten produkt je implementacija abstraktnega produkta, obenem pa ta produkt ustreza produktu, ki ga ustvari konkretna tovarna.
  - Primer: MongoUser, MongoFood, MongoRecipe.

- **Uporabnik – Glavni program**
  - Uporabnik uporablja samo vmesnike, ki jih definirajo abstraktne tovarne in abstraktni produkti.

Ta princip abstrakcije nudi tudi možnost dostopa do več različnih podatkovnih baz naenkrat. Posledično lahko brez veliko dodatnega truda poizvedujemo po več različnih podatkovnih bazah. Določeno poizvedbo samo pošljemo vsem modulom, ki skrbijo za želene podatkovne baze, ter njihov rezultat združimo.

# 5 TESTIRANJE MIGRACIJE

Za namene testiranja migracij med različnimi NoSQL podatkovnimi bazami, smo najprej napisali popolnoma funkcionalno aplikacijo, ki ustreza prej določenim zahtevam ter za izhodiščno podatkovno bazo uporablja MongoDB. Na primeru te aplikacije smo nato prikazali najprej migracijo na sorodno, nato pa še na nesorodne podatkovne baze.

## 5.1 Izvorna implementacija dostopa do podatkovne baze

Za dostop do podatkovne baze MongoDB smo uporabili uradne gonilnike za programski jezik C#. Tako je razvoj aplikacije potekal hitro in enostavno. Gonilniki omogočajo odlično integracijo z programskim jezikom C#, obenem pa nudijo odličen nadzor nad podatkovno bazo.

### 5.1.1 Implementacija podatkovnega modela

Implementacija podatkovnega modela je hitra in enostavna. Vsi podatkovni razredi dedujejo od skupnega abstraktnega razreda, ki implementira skupen vmesnik. Vsakemu razredu je potem dodan še dodaten atribut ID, tipa ObjectID, ki ga zahteva MongoDB. Ta atribut predstavlja unikatno identifikacijo dokumenta, oziroma drugače povedano, znotraj podatkovne baze je to ključ, vrednost pa je ostala vsebina dokumenta.

```
public class MongoRecipe : Recipe, IRecipe
{
    public ObjectId Id { get; set; }

    public MongoRecipe():base()
    {
    }
}
```

**Slika 9: Primer adaptacije najkompleksnejšega razreda na MongoDB podatkovno bazo.**

Ker smo vsepovsod uporabljali vmesnike, je bilo potrebno ročno definirati, katero implementacijo vmesnikov naj gonilnik MongoDB uporabi pri deserializaciji objektov. To je zelo nedvoumno, saj enostavno dodamo vse preslikave med vmesniki in implementacijami ter s tem rešimo problem.

### 5.1.2 Implementacija poizvedb

Vmesnik MongoDB nudi bolj ali manj direktno preslikavo med izvornimi ukazi MongoDB, obenem pa doda funkcionalnosti, ki olajšajo delo s podatkovno bazo v konkretnem jeziku. Poizvedovanje je izvedeno v treh korakih. Najprej določimo, nad katero zbirko bomo izvajali

poizvedbo, potem določimo operacijo, ki jo želimo izvesti nad to zbirko, nato pa določimo še nad katerimi elementi zbirke bomo akcijo izvedli.

### Zbirka

Vsi dokumenti so v MongoDB shranjeni v zbirkah, ki tipično nimajo omejene velikosti. Velikost zbirk je omejena samo pri posebnih zbirkah, ki omogočajo avtomatsko urejanje elementov zbirke. Zbirka lahko vsebuje dokumente različnih tipov, z različno strukturo dokumentov. Vendar je to večinoma nesmiselno; pogosto znotraj ene zbirke shranjujemo sorodne oziroma iste dokumente, ki pa se mogoče razlikujejo v kakšnem polju, kjer je to pač smiselno. Zato smo tu ustvarili tri zbirke: zbirko uporabnikov, zbirko osnovnih živil in zbirko receptov. Objektom, ki jih shranjujemo v zbirke, lahko določimo, katera polja naj se shranijo, oziroma kdaj naj se katera polja shranijo in kaj narediti, če dokument ob deserializaciji določenega polja nima. Shranjevanje objektov v zbirkah ni isto kot tipično shranjevanje objektov v objektnih podatkovnih bazah. Zbirke ni potrebno vnaprej ustvariti, ampak jo samo prikličemo z njenim imenom in jo potem naprej normalno uporabljamo, kot da bi zbirka že obstajala. Če dodamo elemente v zbirko, bo MongoDB zbirko tudi dejansko ustvarila.

```
users = database.GetCollection<NSM.MongoDB.Models.MongoUser>("users");
foods = database.GetCollection<MongoFood>("foods");
recipes = database.GetCollection<MongoRecipe>("recipes");
```

**Slika 10: Primer dostopa do zbirke.**

### Akcija

Akcija se izvede nad določenimi elementi zbirke. Akcij je kar precej, najbolj pogosto uporabljene pri poizvedovanju so Find, FindOne in FindAll, za dodajanje in odstranjevanje pa se uporabljata Insert in Remove.

```
var result = recipes.Find(query);
```

**Slika 11: Primer akcije, ki izvede poizvedbo ter vrne kurzor za dostop do vseh receptov, ki ustrezajo podani poizvedbi.**

### Izbor elementov

Izbor elementov je odlično izveden, saj lahko poizvedbe gradimo s pomočjo razreda *Query*. To je razred za grajenje poizvedb, ki mu določimo operator ter podamo argumente za določen operator. Operatorjev je veliko, med drugimi tudi: LT, GT, LTE, GTE, EQ, AND, ALL, ANY, LIKE, WHERE, NOT, MOD, IN, IN RADIUS. Operatorjem podamo tudi pripadajoča argumenta, ki sta pri določenih operatorjih (primer AND ) lahko kar druge poizvedbe, kar dejansko omogoča grajenje kompleksnih poizvedb. Tipično pa je prvi argument ime polja

znotraj dokumenta, po katerem želimo poizvedovati, drugi argument pa je vrednost, s pomočjo katere se izvede operacija nad dejansko vrednostjo podanega elementa (primer EQ ).

```
var query = Query.And(Query.Matches("Foods.Name", name), Query.GT("protein", value));
```

**Slika 12: Primer grajenja poizvedbe. Poizvedba vrne vse recepte, ki so narejeni iz hrane katere ime se ujema s podanim imenom in imajo več beljakovin kot je določena vrednost.**

Poizvedovanje je možno tudi na nekaj drugih načinov, med drugimi tudi s pomočjo LINQ, vendar je prej opisan način poizvedovanja priporočen. Opisan način poizvedovanja omogoča, da iščemo po poljih dokumenta, ki jih sami definiramo kot niz znakov, kar bistveno olajša dinamično poizvedovanje, saj lahko v eni vrstici opišemo poizvedbo po poljubnem polju.

```
var results = recipes.Find(Query.EQ(field, val));
```

**Slika 13: Primer dinamičnega poizvedovanja, ki poišče vse dokumente, katerih vsebina polja z imenom definiranim v "field" ima enako vrednost kot je vrednost "val"**

## 5.2 Migracija na sorodno podatkovno bazo

Migracija na sorodno podatkovno bazo se na prvi pogled zdi precej preprosta operacija. Podoben podatkovni model in podoben način poizvedovanja po podatkovni bazi se zdita smiselna argumenta za to trditev. Vendar je brez ustrezne abstrakcije takšna migracija lahko zelo težavna, saj so razlike med NoSQL podatkovnimi bazami precej velike tudi znotraj iste skupine NoSQL podatkovnih baz.

### 5.2.1 Migracija na dokumentno podatkovno bazo CouchDB

Migracija na CouchDB se je izkazala za precej zahtevno. Precej več truda je bilo potrebno vložiti v implementacijo dostopa do CouchDB, kot pa v realizacijo dostopa do podatkovne baze MongoDB. Glavne razlike so posledica popolnoma drugačnega koncepta shranjevanja dokumentov. MongoDB shranjuje vse dokumente v zbirkah, CouchDB pa shranjuje vse dokumente nestrukturirano. MongoDB poizveduje po zbirkah, CouchDB pa za poizvedovanje ustvari poglede nad nestrukturiranimi podatki. MongoDB posledično podpira ad-hoc poizvedbe, medtem ko je pri CouchDB grajenje pogledov dolgotrajen proces.

CouchDB nudi samo HTTP REST knjižnico API za dostop do podatkovne baze iz določenega programskega jezika. Posledično obstajajo samo neuradne odprtokodne knjižnice API, ki skrijejo podrobnosti protokola HTTP. Odločili smo se za odprtokodno knjižnico »Hammock / Relax«, saj je projekt bolj aktiven od ostalih, obenem pa nudi preprosto in učinkovito integracijo s programskim jezikom C#.[19]

### Migracija podatkovnega modela

Migracija podatkovnega modela se ni izkazala za preveč zahtevno. Preprostih razredov ni bilo potrebno spreminjati, ampak so le dedovali od abstraktnih razredov.

```
public class CouchDBFood : Food, IFood
{
}
```

**Slika 14: Primer implementacije preprostih razredov.**

Precej trivialna je tudi implementacija kompleksnejših razredov; razredi sicer morajo implementirati skupni vmesnik, ampak realno je možno skopirati celotno kodo abstraktnega razreda. Edini razlog, zakaj je to potrebno storiti je, da lahko tako določimo, kako se bodo kakšna polja objekta serializirala. Knjižnica Relax uporablja knjižnico NewtonsoftJSON za serializacijo, ki med drugim omogoča dekoriranje polj v razredu z atributi, ki določajo način serializacije in deserializacije [13]. To je pomembno predvsem kadar so polja kompleksna, kot je na primer seznam vmesnikov. Pri preprostejših tipih je tipično dovolj dobra avtomatska odločitev knjižnice za serializacijo. V kolikor želimo posebne lastnosti serializacije, pa lahko z ustrezno dekoracijo polj dosežemo tudi posebne učinke, kot je na primer posebno ime polj.

```
public class CouchDBRecipe : IRecipe
{
    public string Name { get; set; }

    [JsonProperty(ItemTypeNameHandling = TypeNameHandling.Auto)]
    public List<IFood> Foods { get; set; }

    public List<Double> Quantities { get; set; }
    public string Directions { get; set; }

    public double Calories { get; set; }
    public double Fat { get; set; }
    public double Carbs { get; set; }
    public double Protein { get; set; }
```

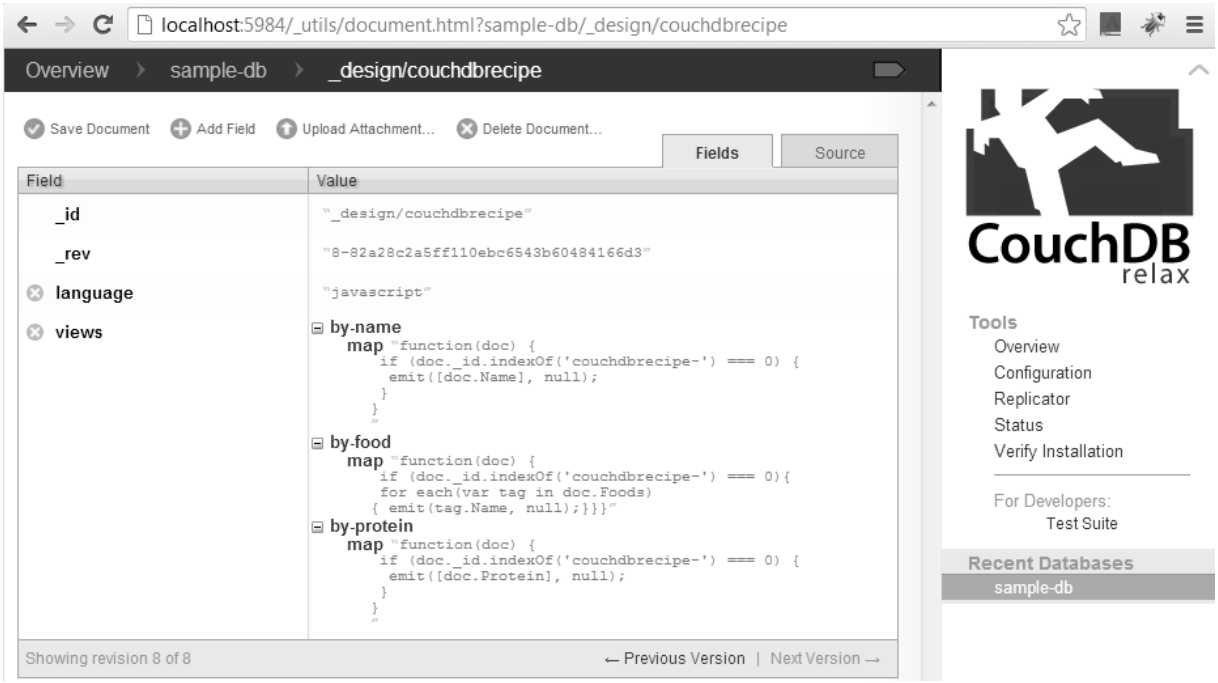
**Slika 15: Primer implementacije kompleksnejših razredov. Razred ne more samo dedovati od abstraktnega razreda ampak mora implementirati skupni vmesnik, saj le tako lahko dekodira polja z atributi za serializacijo. Tukaj je uporabljen atribut ItemTypeNameHandling, ki določa kako se serializirajo elementi znotraj seznama.**

### Migracija poizvedb

Implementacija poizvedb v CouchDB je dosti težja kot pa pri podatkovni bazi MongoDB. Najprej je potrebno dobro razumevanje delovanja CouchDB, shranjevanja dokumentov in pomena pogledov. Tudi delovanje MongoDB je potrebno dobro razumeti, vendar je delovanje MongoDB malo bolj intuitivno in lažje razumljivo. CouchDB shranjuje vse podatke na nestrukturiran način in namesto klasičnega poizvedovanja uporablja tako imenovane poglede,

ki predstavljajo rezultate Map funkcij nad nestrukturiranimi podatki. Celotno poizvedovanje se potem izvaja nad rezultati pogledov, ki so shranjeni v drevesu B. S pogledi ustvarimo posebne množice dokumentov, ki ustrezajo pogojem definiranim v Map funkciji, obenem pa določimo indekse oziroma poizvedovalne ključe za vsak dokument.

Futon je spletni administrativni uporabniški vmesnik, namenjen za nadzor nad podatkovno bazo CouchDB. Med drugim omogoča tudi pisanje začasnih pogledov ter kreiranje trajnih pogledov. Pisanje začasnih pogledov preko Futona je najboljši pristop k pisanju poizvedb, saj so rezultati pri majhni množici podatkov hitro vidni. Če pa delamo nad veliko množico dokumentov, je priporočljivo narediti majhno podmnožico dokumentov in potem nad njo razvijati nove poglede, saj tako razvoj poteka hitreje.




**Slika 16: Spletni vmesnik Futon. Prikazani so tudi različni pogledi povezani z poizvedovanjem po receptih.**

## 5.3 Migracija na nesorodno podatkovno bazo

Migracije na nesorodne podatkovne baze so po pričakovanju težka opravila. Razlike med NoSQL podatkovnimi bazami so precej velike že med sorodnimi podatkovnimi bazami, nesorodne podatkovne baze pa so si tipično zelo različne in so namenjene reševanju drugih problemov. Tako je težko govoriti samo o migraciji med nesorodnimi podatkovnimi bazami, ker tipično v primeru, da menjamo tip podatkovne baze, to kaže tudi na druge potrebe in / ali težave v aplikaciji. Z vsako podatkovno bazo lahko rešimo skoraj vsak problem, vendar ne bo vsaka podatkovna baza pri tem enako učinkovita in primerna nalogi.

V tej diplomski nalogi smo se osredotočili samo na migracijo podatkovne baze in ne na spreminjanje delovanja aplikacije ob menjavi podatkovne baze. Tipično pa pri prehodu na drug tip NoSQL podatkovne baze dodamo tudi kakšno novo funkcionalnost, zaradi katere potrebujemo novo podatkovno bazo.

### 5.3.1 Migracija na podatkovno bazo na osnovi grafov BrightstarDB

Implementacija dostopa do Graph Based NoSQL podatkovne baze BrightstarDB je vzela dosti več časa in truda, v primerjavi z MongoDB in CouchDB. Glavni razlogi za to so ravno v načinu integracije gonilnikov BrightstarDB s programskim jezikom C#. BrightstarDB nudi programski dostop do podatkovne baze preko različnih vmesnikov; edini univerzalno dostopni je vmesnik SPARQL. Vsi ostali so namenjeni le programskemu jeziku C# ter nudijo različno podroben nadzor nad podatkovno bazo.

**Migracija podatkovnega modela**

BrightstarDB Entity Framework je podoben klasičnemu Entity Framework-u. Modele je možno ustvariti preko posebnih vmesnikov, vendar morajo vsi tipi, ki nastopajo v vmesniku, biti ali osnovni ali pa del BrightstarDB Entity Framework-a, kar pomeni, da so od zbirk dovoljene samo zbirke tipa »ICollection« in da so edini vmesniki, ki so dovoljeni del vmesnikov BrightstarDB. Drugače povedano, vmesniki za generiranje entitet BrightstarDB tipično ne morejo dedovati od skupnih vmesnikov. Posledično smo morali narediti vmesnike, ki so v vsem enaki skupnim vmesnikom, razlikujejo pa se v tem, da vsebujejo vmesnike BrightstarDB in ne skupne vmesnike.

```
public interface IRecipe
{
    string Name { get; set; }
    List<IFood> Foods { get; }
    List<Double> Quantities { get; }
    string Directions { get; set; }
```

**Slika 17: Vmesnik BrightstarDB ne more dedovati od tega vmesnika, saj ta vmesnik vsebuje seznam elementov tipa IFood, pri čemer je IFood skupni vmesnik in ne vmesnik BrightstarDB.**

```
[Entity]
public interface IBrightstarDBRecipe
{
    /// <summary>
    /// Get the persistent identifier for this entity
    /// </summary>
    string Id { get; }

    string Name { get; set; }
    ICollection<IBrightstarDBFood> Foods { get; set; }
    ICollection<Double> Quantities { get; set; }
    string Directions { get; set; }
```

**Slika 18: Primer vmesnika BrightstarDB, ki ustreza prevedbi zgoraj omenjenega vmesnika. Od zgornjega vmesnika se razlikuje v dodatnem polju Id in v spremenjenem tipu seznama.**

Posledično se je bistveno zakomplicirala dejanska implementacija modelov. Modeli morajo še vedno implementirati skupne vmesnike, zato so modeli realizirani kot ovojnice do vmesnika BrightstarDB. Implementirajo skupni vmesnik ter pri dostopu do lastnosti po potrebi ustrezno predelajo lastnosti entitet BrightstarDB.

```
public class BrightstarRecipe :IRecipe
{
    public IBrightstarDBRecipe BrightstarEntity { get; set; }

    public BrightstarRecipe(IBrightstarDBRecipe recipe)
    {
        BrightstarEntity = recipe;
    }

    public string Name
    {
        get
        {
            return BrightstarEntity.Name;
        }
        set
        {
            BrightstarEntity.Name = value;
        }
    }


    public List<IFood> Foods
    {
        get
        {
            var foods = new List<IFood>(BrightstarEntity.Foods.Count());
            foreach (var food in BrightstarEntity.Foods)
            {
                foods.Add(new BrightstarFood(food));
            }
            return foods;
        }
```

**Slika 19: Primer realizacije modelov BrightstDB. Model implementira skupni vmesnik ter ustrezno po potrebi pretvarja vrednosti, podane ob *get* oziroma *set* klicih.**

### Migracija poizvedb

Velika konceptualna razlika med podatkovnimi bazami z grafi in ostalimi podatkovnimi bazami močno vpliva tudi na način poizvedovanja. Poizvedovanje po podatkovnih bazah z grafi tipično poteka v poizvedovalnem jeziku SPARQL. BrightstarDB nudi tudi zelo dobro integracijo s programskim jezikom C#, tako da imamo na voljo BrightstarDB Entity Framework, ki omogoča poizvedovanje s pomočjo LINQ ter Dynamic API, Data Object Layer in RDF Client API, preko katerih poizvedujemo s poizvedovalnim jezikom SPARQL. Tako smo osnovne poizvedbe izvedli s pomočjo LINQ, nekatere bolj kompleksne pa s pomočjo jezika SPARQL.

Pisanje poizvedb s pomočjo LINQ poteka hitro in enostavno, vendar BrightstarDB ne implementira vseh funkcionalnosti LINQ, tako da je potrebno prej preveriti, ali je želena funkcionalnost sploh podprta.

```
query = context.BrightstarDBRecipes.Where(
    recipe => recipe.Foods.Where(
        food => food.Name.Equals(name)).Any());
```

**Slika 20: Primer LINQ poizvedbe. Poizvedba vrne vse recepte, ki vsebujejo določeno hrano.**

Poizvedovalni jezik SPARQL je sprva malce težje razumljiv, vendar se izkaže za zelo učinkovitega in precej preprostega, ko ga enkrat spoznamo. Pisanje kompleksnih poizvedb s poizvedovalnim jezikom SPARQL je precej preprosto. Spodaj je primer poizvedbe SPARQL.

```
var squery = @"prefix loc: <http://brightstardb.com/namespaces/default/>
                SELECT ?name WHERE
                    {
                        ?recipe  a loc:BrightstarDBRecipe.
                        ?recipe loc:name ?name .
                        ?recipe loc: protein ?macro .
                        FILTER (?macro > 20 )
                     }";
```

**Slika 21: Primer poizvedovanja s poizvedovalnim jezikom SPARQL. Poizvedba vrne imena vseh receptov, ki imajo več kot 20g beljakovin.**

Poizvedovanje s pomočjo poizvedovalnega jezika SPARQL je realizirano kot izvajanje poizvedb SPARQL, ki so nizi znakov, kar omogoča dinamično grajenje poizvedb, ki lahko bistveno olajša in skrajša razvojni čas poizvedb.

```
var squery = @"prefix loc: <http://brightstardb.com/namespaces/default/>
                SELECT ?name WHERE
                    {
                        ?recipe  a loc:BrightstarDBRecipe.
                        ?recipe loc:name ?name .
                        ?recipe loc:" +smacro+@" ?macro .
                        FILTER (?macro "+ comp + " " + sValue + @")
                     }";
```

**Slika 22: Primer dinamičnega poizvedovanje s poizvedovalnim jezikom SPARQL. Poizvedba vrne vse recepte katerih lastnost je v določenem razmerju (<, >, =, ...) z podano vrednostjo.**

### 5.3.2 Migracija na Ključ – vrednost podatkovno bazo Redis

Migracija na podatkovno bazo tipa ključ – vrednost je ena izmed težjih, saj te podatkovne baze nudijo zelo omejene funkcionalnosti, ter s tem zagotavljajo boljšo odzivnost podatkovne baze. Večino funkcionalnosti poizvedovanja moramo kot razvijalci sami razviti. Že ob samem načrtovanju strukture shranjevanja podatkov moramo misliti na poizvedovanje, saj je dostop do podatkov možen samo preko znanega ključa.

Za dostop do podatkovne baze Redis smo uporabili Redis ServiceStack knjižnico API [21]. Knjižnica omogoča dobro integracijo s programskim jezikom C#, saj nudi različno podroben nadzor nad podatkovno bazo.

IRedisNativeClient nudi zelo nizkonivojski dostop do podatkovne baze, saj predstavlja direktno preslikavo vseh funkcij, ki jih nudi Redis, kar pomeni, da so marsikateri argumenti funkcij kar polja bajtov. IRedisClient nudi srednje-nivojski dostop do podatkovne baze, ki skrije detajle bajt polj. IRedisTypedClient pa je visokonivojski vmesnik, ki omogoča enostavno delo z objekti.

### Migracija podatkovnega modela

Migracija podatkovnega modela je hitra in enostavna – najpreprostejša od vseh do sedaj. Modeli zgolj dedujejo od skupnih abstraktnih tipov, kar je za potrebe Redis podatkovne baze popolnoma dovolj.

```
public class RedisRecipe :Recipe, IRecipe
{
}
```

**Slika 23: Primer implementacije najkompleksnejšega modela. Zadostuje le dedovanje od skupnega abstraktnega razreda.**

### Migracija poizvedb

Migracija poizvedb pa je bistveno kompleksnejša kot pri ostalih podatkovnih bazah. Vgrajenega indeksiranja ni, tipično dostopamo do elementa samo preko njegovega ključa. Posledično je potrebno ključe generirati na nek sistematičen način. Na primer, pri shranjevanju uporabnikov naredimo ključ tako, da uporabniškemu imenu dodamo skupno predpono značilno za uporabnike. Tako bomo vedno ob določenem uporabniškem imenu lahko dobili ustrezne podatke o tem uporabniku.

```
bool success = users.SetEntryIfNotExists(usernamePrefix + user.UserName, redisUser);
```

**Slika 24: Dodajanje uporabnika, je realizirano kot shranjevanje objekta uporabnik s ključem, ki je sestavljen iz predpone za uporabnike in uporabniškega imena.**

Poizvedovanje po receptih pa je bistveno bolj kompleksno. Shranjevanje receptov je sestavljeno na podoben način, kot shranjevanje uporabnikov; recept je torej shranjen s ključem, ki je sestavljen iz predpone za recept ter iz imena recepta. Obenem smo naredili poseben seznam, ki vsebuje ključe vseh receptov. Tako je vso poizvedovanje in filtriranje nad recepti realizirano znotraj aplikacije, saj od Redis podatkovne baze zahtevamo le vse elemente iz našega seznama, ki vsebuje ključe vseh receptov.

```
resultNames = redisClient.GetAllItemsFromSet(recipeNamesSetId).ToList();
result = new List<RedisRecipe>(resultNames.Count);

foreach (string name in resultNames)
{
    var recipe = GetRecipe(name) as RedisRecipe;
    if (recipe == null)
        continue;

    if (recipe.Foods.Any(food => food.Name.Contains(queryName)))
        result.Add(recipe);
}
```

**Slika 25: Primer poizvedbe: iskanje vseh receptov, ki vsebujejo določeno sestavino. Najprej pridobimo vsa imena receptov, potem pregledamo vse recepte ter izberemo tiste, katerih sestavine vsebujejo podan niz znakov.**

Ta rešitev je sicer časovno potratna, ker pa se Redis nahaja v pomnilniku, je to še vedno izjemno hitro. To je možno tudi pospešiti, vendar na račun večje rabe pomnilnika. Ena možna rešitev je, da bi za vsako konkretno polje, po katerem želimo poizvedovati, ustvarili poseben seznam, v katerem bi hranili ključe do vseh elementov, ki vsebujejo to polje. Na primer, za poizvedovanje po sestavinah, bi za vsako možno hrano določili seznam, ki bi vseboval ključe vseh receptov, ki vsebujejo to hrano. To je seveda prostorsko zelo potratno, ker je pomnilnik precej drag in redek vir in ker je za Redis to ključnega pomena, je v večini primerov bolje več dela preložiti na procesor.

### 5.3.3 Migracija na podatkovno bazo na osnovi družine stolpcev – Cassandra

Migracija na podatkovno bazo na osnovi družine stolpcev je lahko izjemno zahtevna, saj ta tip podatkovne baze omogoča le osnovne funkcionalnosti ter razvijalcem ponudi vse potrebno, da si po želji in potrebi sami uredijo in zagotovijo kompleksnejše operacije, kot so kompleksne poizvedbe.

Za dostop do podatkovne baze smo uporabljali Datastax C# Driver For Apache Cassandra knjižnico API. Knjižnica nudi tako visoko-nivojsko upravljanje s podatkovno bazo preko LINQ, kot tudi nizko-nivojski dostop preko jezika CQL.[8]

#### Migracija podatkovnega modela

Migracija podatkovnega modela je bila precej enostavna. Potrebno je bilo implementirati abstraktne razrede, saj je bilo potrebno dodati posebne atribute, ki označujejo, kakšen pomen imajo določene lastnosti znotraj družine stolpcev; torej, ali gre za glavni ključ, sekundarni ključ, ipd.

```
[AllowFiltering]
public class CassandraDBRecipe
{
    [PartitionKey]
    public string Name { get; set; }

    public List<string> Foods { get; set; }

    public List<double> Quantities { get; set; }

    public string Directions { get; set; }

    [SecondaryIndex]
    public double Calories { get; set; }

    [SecondaryIndex]
    public double Fat { get; set; }

    [SecondaryIndex]
    public double Carbs { get; set; }

    [SecondaryIndex]
    public double Protein { get; set; }
```

**Slika 26: Primer implementacije IRecipe vmesnika za podatkovno bazo Cassandra.**

Potrebno je bilo tudi spremeniti način shranjevanja določenih kompleksnejših podatkovnih tipov, saj Cassandra podpira le shranjevanje osnovnih podatkovnih tipov, kot so različne vrste števil, tekstovnega zapisa , števec, datum, seznam in zbirka ter binarni zapisi.

### Migracija poizvedb

Migracija osnovnih poizvedb je bila zelo preprosta. Zataknilo pa se je pri migraciji kompleksnih poizvedb, saj Cassandra ne podpira primerjalnega poizvedovanja po poljubnih stolpcih, niti ne podpira drugih potrebnih kompleksnejših poizvedb.

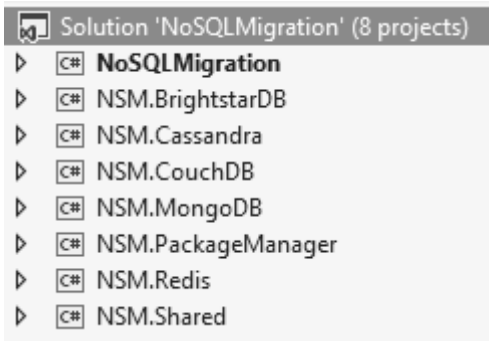
Na tem mestu smo zato ocenili možne rešitve. Ena izmed možnosti je uporaba super stolpcev, ki pa jih skupnost dandanes označuje za precej neprimerne, obstajajo pa tudi govorice, da bodo v prihodnjih različicah Cassandra-e umaknjeni. Druga možnost pa je gradnja lastnih indeksov. To pa je zaradi velikega in raznolikega števila polj, ki jih je potrebno indeksirati, zelo zamudno in zahtevno delo. Obenem pa to še bolj oteži možne nadaljnje spremembe v podatkovnem modelu.

Zato smo na tem mestu ocenili, da je takšna podatkovna baza neprimerna izbira za tovrstno problemsko domeno, saj negativno vpliva na razvoj aplikacije. Naša aplikacija vsebuje precej veliko med seboj povezanih elementov ter kompleksnih poizvedb. Sama aplikacija pa ne potrebuje izjemno zmogljive podatkovne baze, saj največja obstoječa aplikacija (Allrecipes [1] ) uporablja SQL podatkovno bazo. Iz tega lahko sklepamo, da bi za te potrebe lahko brez

težav uporabljali bolj primerno podatkovno bazo, na primer eno izmed dokumentnih podatkovnih baz.

## 5.4 Arhitektura aplikacije

Aplikacija je bila zgrajena modularno, tako da omogoča enostavno dodajanje in spreminjanje vmesnikov za dostop do podatkovne baze. Vsak modul je predstavljen s svojim projektom, saj ga je možno neodvisno razvijati od ostalih modulov.




**Slika 27: Modularna arhitektura aplikacije, vsak modul je predstavljen s svojim projektom.**

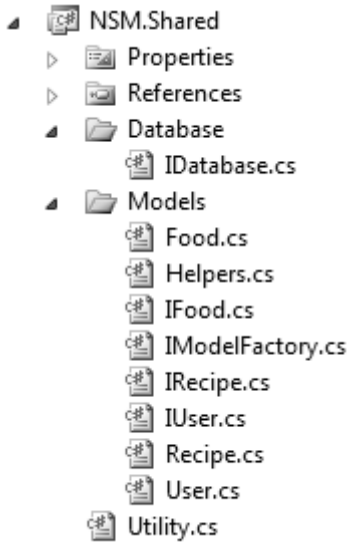
Arhitektura aplikacije temelji na arhitekturi SOS [26], katere glavni cilj je narediti enoten dostop do različnih NoSQL podatkovnih baz, ki je neodvisen od določene aplikacije. To jim sicer uspe, vendar ta pristop podpira le osnovne operacije (PUT, GET in DELETE ), kar je za večino primerov popolnoma nekoristno, saj najkompleksnejši in najpomembnejši del dostopa do podatkovne baze predstavljajo ravno različne poizvedbe. V našem delu smo zato raje naredili dostop do podatkovne baze specifičen aplikaciji ter s tem ohranili vse prednosti in funkcionalnosti različnih NoSQL podatkovnih baz.

### Glavni modul

NoSQL Migration je ime glavnega dela projekta, kjer je realizirana programska logika. Glede na to, da je to samo simulacija realnega projekta in je cilj diplomske nalogi prikazati težave migracije med NoSQL podatkovnimi bazami, je celotna logika aplikacije kar v enem modulu. V realnem projektu bi bila tudi glavna aplikacija razdeljena na več modulov, na primer, na uporabniški vmesnik in na DAL (Data Access Layer – modul za manipulacijo s podatki ). Glavni modul je odvisen od Shared modula, kjer se med drugimi nahajajo vmesniki za dostop do podatkovne baze, ter od PackageManager modula, ki povezuje vse module, ki skrbijo za dostop do podatkovne baze. PackageManager priskrbi glavnemu modulu želene funkcionalnosti določene podatkovne baze.

### Shared modul

Modul NSM.Shared vsebuje stvari, ki so skupne vsem modulom, ki uporabljajo podatkovne baze. Notri so med drugimi definirani vmesniki, ki predstavljajo modele, njihove abstraktne implementacije in vmesnik za dostop do podatkovne baze.




**Slika 28: Zgradba modula NSM.Shared, ki je razdeljena na dva dela, del za upravljanje s podatkovno bazo in del z modeli. Vsi vmesniki imajo pred imenom črko 'I'.**

### PackageManager modul

NSM.PackageManager skrbi za urejanje in povezovanje vseh modulov, ki skrbijo za dostop do podatkovne baze. Povezuje module za delo s podatkovno bazo z glavnim modulom. Glavni modul samo določi katero podatkovno bazo želi uporabljati, PackageManager pa mu potem priskrbi ustrezne implementacije.

```
public enum DatabaseType
{
    MongoDB, CouchDB, Neo4J, BrightstarDB, Redis, Cassandra
}

public class FactoryManager
{
    private DatabaseType databaseType;

    public FactoryManager(DatabaseType databaseType)
    {
        this.databaseType = databaseType;
    }

    public IModelFactory GetModelFactory()
    {
        switch (databaseType)
        {
            case DatabaseType.MongoDB:
                return new MongoModelFactory();
            case DatabaseType.CouchDB:
                return new CouchDBModelFactory();
            case DatabaseType.BrightstarDB:
                return new BrightstarModelFactory();
            case DatabaseType.Redis:
                return new RedisModelFactory();
            case DatabaseType.Cassandra:
                return new CassandraModelFactory();
            default:
                return null;
        }
    }
```

**Slika 29: Zgoraj prikazan je del PackageManagerja, ki glavnemu modulu posreduje ustrezno tovarno za grajenje modelov.**

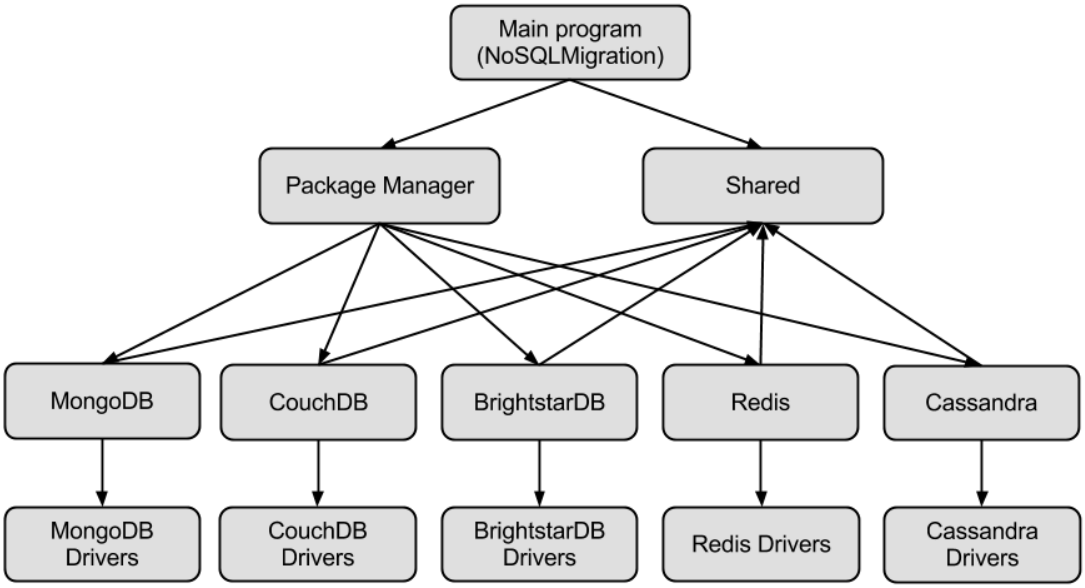
PackageManager je odvisen tako od Shared modula ter od vseh modulov, ki implementirajo dostop do podatkovne baze, kot ga definira Shared modul.

### Podatkovna baza

Moduli, ki skrbijo za dostop do določene NoSQL podatkovne baze, implementirajo ustrezne vmesnike, ki so definirani v modulu NSM.Shared. Implementirati morajo podatkovne modele, vmesnik, ki skrbi za grajenje modelov ter vmesnik, ki skrbi za dostop do podatkovne baze. Ti moduli imajo tudi različne odvisnosti, tipično so to knjižnice API, ki nudijo dostop do želene podatkovne baze. Lahko pa so tam tudi različne druge knjižnice, na primer za serializacijo v ustrezen format (primer Newtonsoft.JSON ).

**Celotna arhitektura**

Celotna arhitektura je zasnovana tako, da so moduli med sabo čim bolj neodvisni. Tako lahko brez težav spremenimo samo določen modul, brez da bi s tem vplivali na ostale.




**Slika 30: Diagram prikazuje odvisnosti med različnimi moduli. Smer puščice kaže, da je začetni modul odvisen od končnega. Na zgornjem nivoju je glavni modul, na drugem nivoju so moduli, ki povezujejo module s podatkovnimi bazami. Na predzadnjem nivoju so moduli, ki skrbijo za integracijo s podatkovno bazo, na zadnjem nivoju pa so njihovi pripadajoči gonilniki in ostale knjižnice API, ki jih ti moduli potrebujejo.**

## 5.5 Povzetek testiranja migracij

Testirali smo težavnost migracij iz izvorne podatkovne baze, ki je v našem primeru bila MongoDB, na druge NoSQL podatkovne baze. Migracija na sorodno podatkovno bazo CouchDB je bila najlažja, saj se obe podatkovni bazi uvrščata med dokumentne podatkovne baze. Največ dela je bilo z drugačnim načinom poizvedovanja, saj CouchDB uporablja tako imenovane poglede.

Migracija na podatkovno bazo na osnovi grafov BrightstarDB je bila zahtevnejša. Vložiti je bilo potrebno več truda v migracijo podatkovnega modela. Večji del dela pa je predstavljala migracija poizvedb, saj je bilo ponekod potrebno poizvedbe napisati v poizvedovalnem jeziku SPARQL.

Redis, ključ – vrednost podatkovna baza je potrebovala še več truda za migracijo. Najtežavnejša je bila migracija poizvedb, saj je bilo potrebno popolnoma spremeniti način razmišljanja o poizvedovanju po podatkih. Že ob shranjevanju podatkov smo morali

razmišljati o tem, kako bomo kasneje poizvedovali po njih, ter sami smo morali skrbeti za vsa indeksiranja.

Podatkovna baza na osnovi družine stolpcev Cassandra, pa je terjala daleč največ truda. Tako kot pri Redisu, smo tudi tukaj sami bili zadolženi za reševanje »osnovnih« problemov, kot so poizvedovanje in indeksiranje.

Spodnja preglednica prikazuje različne atribute opisanih migracij.

| **Podatkovna baza** | **Težavnost prenosa podatkovnega modela** | **Razlika v načinu poizvedovanja** | **Težavnost prenosa poizvedb** | **Možnost migracije** |
|---|---|---|---|---|
| CouchDB | Majhna | Srednje velika | Majhna | Zelo velika |
| BrightstarDB | Srednje velika | Srednje velika | Srednje velika | Velika |
| Redis | Majhna | Velika | Zelo velika | Majhna |
| Cassandra | Velika | Zelo velika | Ogromna | Zelo majhna |

# 6 SKLEPNE UGOTOVITVE

NoSQL podatkovne baze so se razvile z namenom, da rešujejo določene probleme sodobnih aplikacij, ki jim relacijske podatkovne baze niso bile več kos. Med NoSQL podatkovnimi bazami obstajajo velike razlike, ki jih ni lahko premostiti. Vendar problem ni v velikih razlikah, saj jih ravno ta raznolikost naredi izjemno koristne. Vsak tip podatkovne baze je namenjen reševanju drugih problemov, vsaka NoSQL podatkovna baza, tudi tiste istega tipa, pa je različno primerna za različne potrebe. Velike razlike med NoSQL pa močno otežujejo morebitne migracije med njimi. Diplomsko delo se tako ukvarja z vprašanjem težav, ki nastopijo ob migracijah med NoSQL podatkovnimi bazami, ter išče rešitve, ki bi te migracije lahko vsaj delno omilile.

Najboljša rešitev za težave ob migracijah je torej dobro poznavanje potreb lastne aplikacije; tako trenutnih, kot tudi morda drugačnih v prihodnosti. Samo poznavanje potreb aplikacije ni dovolj, dobro moramo biti seznanjeni tudi z različnimi podatkovnimi bazami, njihovimi značilnostmi, načinu delovanja in namenu uporabe. Le tako bomo lahko dobro izbrali podatkovno bazo ter se s tem izognili težavam, ki jih prinaša migracija podatkovne baze.

Če pa se migraciji vendar ne moremo izogniti, obstajajo tudi rešitve, ki morebitno migracijo delno olajšajo. Najprej je potrebno jasno in dobro ločiti programsko kodo, na kakšno različico modela, pogleda in vmesnika. Dobro je uporabljati vmesnike ter abstraktno tovarno ali pa inverzijo kontrole, kar bistveno olajša morebitne spremembe v kodi. V nekaterih programskih jezikih lahko uporabimo tudi določena ogrodja, ki (pol ) avtomatsko omogočajo dostop do različnih NoSQL podatkovnih baz, vendar nam tipično skrijejo preveč detajlov ter nudijo dostop mogoče le do dveh ali treh NoSQL podatkovnih baz. Lastna, aplikaciji specifična ogrodja so večinoma najboljša rešitev. Ogrodja na principu abstraktne tovarne pa nam omogočajo tudi upravljanje z več (NoSQL ) podatkovnimi bazami simultano, kar je za nekatere aplikacije lahko zelo koristno.

## VIRI IN LITERATURA


[1] (2013) Allrecipes. Dostopno na: http://allrecipes.com/

[2] (2013) Apache Cassandra. Dostopno na: http://en.wikipedia.org/wiki/Apache_Cassandra

[3] (2013) BrightstarDB Documentation. Dostopno na: http://brightstardb.readthedocs.org/en/latest/

[4] (2013) Column family. Dostopno na: http://en.wikipedia.org/wiki/Column_family

[5] (2013) Couchbase. Why NoSQL? Dostopno na: http://www.couchbase.com/sites/default/files/uploads/all/whitepapers/NoSQL-Whitepaper.pdf

[6] (2013) CouchDB. Dostopno na: http://en.wikipedia.org/wiki/CouchDB

[7] (2013) CSharp Language Center. Dostopno na: http://docs.mongodb.org/ecosystem/drivers/csharp/

[8] (2013) Datastax C# Driver for Apache Cassandra (Beta). Dostopno na: https://github.com/datastax/csharp-driver

[9] (2013) Document-oriented database. Dostopno na: https://en.wikipedia.org/wiki/Document-oriented_database

[10] (2013) Graph database. Dostopno na: http://en.wikipedia.org/wiki/Graph_database

[11] (2013) Hash Table. Dostopno na: http://en.wikipedia.org/wiki/Hash_table

[12] "Is UNQL Dead?," vol. 2013, ed, 2012.

[13] (2013) Json.NET. Dostopno na: http://james.newtonking.com/pages/json-net.aspx

[14] (2013) List of NoSQL Databases. Dostopno na: http://nosql-database.org/

[15] Memcached. Dostopno na: http://memcached.org/

[16] (2013) MongoDB. Dostopno na: http://en.wikipedia.org/wiki/MongoDB

[17] (2012). *NoSQL Database Technology*. Dostopno na: www.couchbase.com

[18] (2013) Redis. Dostopno na: http://en.wikipedia.org/wiki/Redis

[19] (2013) Relax with CouchDB. Dostopno na: https://code.google.com/p/couchdb-relax/

[20] (2013) Resource Description Framework. Dostopno na: http://en.wikipedia.org/wiki/Resource_Description_Framework

[21] (2013) ServiceStack.Redis. Dostopno na: https://github.com/ServiceStack/ServiceStack.Redis

[22] (2013) SPARQL. Dostopno na: http://en.wikipedia.org/wiki/SPARQL

[23] SQL to MongoDB Mapping Chart. Dostopno na: http://docs.mongodb.org/manual/reference/sql-comparison/

[24] (2013) Triplestore. Dostopno na: http://en.wikipedia.org/wiki/Triplestore

[25] (2013) UnQL Query Language Unveiled by Couchbase and SQLite. Dostopno na: http://www.couchbase.com/press-releases/unql-query-language

[26] P. Atzeni, F. Bugiotti, and L. Rossi, "Uniform access to NoSQL systems," *Information Systems*, 2013.

[27] K. Chodorow and M. Dirolf, *MongoDB: The Definitive Guide*, 1st ed. Sebastopol, CA: O'Reilly Media, 2010.

[28] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, 1st ed. United States of America: Addison-Wesley, 1994.

[29] J. Han, Haihong E, and G. Le. (2011) Survey on NoSQL Database.

[30] R. Hecht and S. Jablonski, "NoSQL Evaluation: A Use Case Oriented Survey," presented at the International Conference on Cloud and Service Computing, Hong Kong, 2011.

[31] E. Hewitt, *Cassandra: The Definitive Guide*. Sebastopol, CA: O'Reilly Media, Inc., 2011.
[32] J. Lennon, *Beginning CouchDB*, 1st ed. United States of America: Paul Manning, 2009.
[33] C. J. M. Tauro, S. Aravindh, and A. B. Shreeharsha, "Comparative Study of the New Generation, Agile, Scalable, High Performance NOSQL Databases," *International Journal of Computer Applications*, vol. 48, 2012.
[34] K. North, "The NoSQL Alternative," *InformationWeek*, pp. 33-35,38-39, 2010.
[35] K. Seguin. (2012). *The Little Redis Book*. Dostopno na: http://openmymind.net/redis.pdf
[36] S. Tiwari, *Professional NoSQL*, 1st ed. Indianapolis, Indiana: John Wiley & Sons, Inc., 2011.
[37] G. Wang and J. Tang, "The NoSQL Principles and Basic Application of Cassandra Model," presented at the International Conference on Computer Science and Service System, Nanjing, 2012.