LATVIJAS UNIVERSITĀTE
DATORIKAS FAKULTĀTE

NOSQL DATU BĀŽU ANALĪZE UN TO INTEGRĀCIJAS IESPĒJAS AR ORACLE ADF

MAĢISTRA DARBS

Autors: **Andrejs Daško**
Stud. apl. ad07015
Darba vadītājs: Dr. dat. Alina Vasiļjeva

RĪGA 2013

Anotācija

Pašlaik neeksistē integrācijas risinājumi, kas atļautu lietot *Oracle AD*F, jeb *Oracle* Lietotņu Izstrādes Ietvaru, ar *NoSQL* datu bāzēm. *Oracle ADF* ir tīmekļa ietvars, kas tiek pārsvarā lietots tīmekļa programmu izstrādei lieliem uzņēmumiem. *NoSQL* var būt efektīvāks par tradicionālajām relātrīs-slāņuciju datu bāzēm arī mazākajos uzņēmumos, kuriem jāapstrādā lieli datu apjomi.

Galvenie darba mērķi ir izanalizēt iespējas integrēt *Oracle ADF* ar *NoSQL* datu bāzēm un piedāvāt vienu vai vairākas integrācijas pieejas. Darba rezultātā tika definēti vairāki *NoSQL* labumi, kā, piemēram, augsta nodalījumu tolerance, izstrādes procesa atvieglojums, kartēšanas problēmas risināšana ar agregātu palīdzību, un citi. Tika izstrādātas un aprakstītas divas pieejas, kas atļauj integrēt ADF ietvaru ar ne-relāciju datu bāzēm.

Atslēgvārdi

Oracle ADF, NoSQL, integrācija, trīs-slāņu arhitektūra

Abstract

NoSQL databases and Oracle ADF integration possibilities

Currently there is no solution that would allow using Oracle ADF, or Oracle Application Development Framework, with NoSQL databases. Oracle ADF is a web development framework, which is used to develop web application for enterprises. NoSQL has the potential of being effective even in smaller companies that have to handle large volumes of data.

Main goals of this work are to analyze the possibilities of integrating Oracle ADF with NoSQL databases and propose one or several integration approaches. As a result of this work many of the NoSQL database benefits were outlined, such as high partition tolerance, scale out possibilities, simplification of development process and elimination of the impedance mismatch problem. Two approaches of integrating Oracle ADF with NoSQL databases were defined, described and implemented as a final result of this work.

Key words

Oracle ADF, NoSQL, integration, three-tier architecture

Autoreferāts

Darba izstrādes gaitā tika veikts gan teorētisks pētījums, gan integrācijai nepieciešamo komponenšu izstrāde. Lai paplašinātu savas zināšanas, autors ir piedalījies un veiksmīgi pabeidza divus kursus, kurus rīkoja MongoDB izstrādātāji – *10gen* kompānija. Darba mērķa sasniegšanai un izvirzīto uzdevumu izpildīšanai, autors ir:

- apskatījis un izanalizējis *NoSQL* datu bāžu konceptus,
- salīdzinājis vairākas *NoSQL* datu bāžu arhitektūras,
- paveicis veiktspējas mērījumus,
- aprakstījis *ADF* ietvara arhitektūru,
- aprakstījis un izskatījis ietvara un datu bāžu integrācijas iespējas,
- piedāvājis divas integrācijas pieejas, un
- izstrādājis un aprakstījis modeļu slāņa projektējumu, kas ļauj ērti lietot ne-relāciju datu bāzes sadarbībā ar ADF ietvaru.

APZĪMĒJUMU SARAKSTS

Oracle ADF (Application Development Framework) – Oracle lietotnes izstrādes ietvars, kas būtiski atvieglo Java programmatūras izstrādi, piedāvājot vizuālo un deklaratīvo izstrādes procesu.

SQL (Structured Query Language) – specifiska programmēšanas valoda, visbiežāk izmantota relācijas datu bāžu vaicājumu rakstīšanai.

NoSQL – datu bāzes pārvaldības sistēmas, kuras neizmanto tradicionālo relāciju datu bāzes pārvaldības sistēmu modeli. Tipiski tādas datu bāzes neizmanto tabulas un neizmanto SQL.

MVC (Model-View-Controller) – trīs-slāņu arhitektūra – klientservera arhitektūras tips, ko veido trīs precīzi definēti atšķirīgi procesi: skats, kontrolieris un modelis.

POJO (Plain Old Java Object) – termins, kas apraksta Java objektu, kas neizmanto ne vienu no Java objektu modeliem vai ietvarus.

EJB (Enterprise JavaBeans) – izstrādes specifikācijas, kas apraksta servisa komponentes, kas satur biznesa loģiku.

API (Application programming interface) – lietojumprogrammas saskarne – programmatūras izpausto klašu, metožu, procedūru vai funkciju komplekts.

JPA (Java Persistance API) – Java datu pastāvīguma bibliotēka, kas nodrošina lietojumprogrammas komunikāciju ar datu bāzi.

XML (eXtensible Markup Language) – ir W3C rekomendācija speciālas nozīmes iezīmēšanas valodu veidošanai.

JAX-WS – programmēšanas valoda tīmekļa pakalpojumu izstrādei

JAXB (Java Architecture for XML Binding) – mehānisms, kas atļauj konvertēt XML datus Java objektos un Java objektus XML failos

SATURA RADĪTĀJS

IEVADS 1

1. NOSQL KONCEPTI UN ĪPAŠĪBAS 4

1.1. CAP teorēma 6

1.2. Datu saskaņa 9

1.2.1. Atjauninājumu saskaņa 9

1.2.2. Lasīšanas saskaņa 10

1.2.3. Datu saskaņas atslābināšana 11

1.3. Replicēšana 11

1.3.1. Meistars-Kalps replicēšana 12

1.3.2. Sinhrona un asinhrona replicēšana 14

1.4. Klasterizācija 14

1.5. Agregāts 17

1.6. MapReduce 21

2. DATU BĀZES IZVĒLE 23

2.1. BigTable arhitektūra 23

2.2. Cassandra 24

2.3. HBase 25

2.4. MongoDB 26

2.5. Datu bāzes izvēles pamatojums 28

2.6. Izvēlētas datu bāzes detalizēts apskats 28

2.6.1. MongoDB pielietojumi lielajos uzņēmumos 29

2.6.2. Mongo un SQL komandu salīdzinājums 31

3. VEIKTSPĒJAS TESTĒŠANA 39

3.1. Testēšanas sistēmas specifikācijas 39

3.2. Rakstīšanas pieprasījumi 39

4. ADF PĀRSKATS 42
4.1. Biznesa servisu slānis 44
4.2. Kontrolieru slānis 44
4.3. Skatu slānis 45
5. ADF UN NOSQL DATU BAŽU INTEGRĀCIJA 46
5.1. ADF paplašinājumu iespējas 46
5.2. NoSQL shēmas nepieciešamība 48
5.3. Tīmekļa pakalpojums un tīmekļa pakalpojuma datu kontrole 48
5.3.1. Nepieciešamo komponentu uztaisīšana 48
5.3.2. Tīmekļa pakalpojumu izpaustas metodes 50
5.3.3. Modeļu tīmekļa pakalpojuma un klientu lietotnes mijiedarbība 51
5.4. Datu kontrole no Java klases 59
SECINĀJUMI 62
IZMANTOTĀ LITERATŪRA UN AVOTI 64

IEVADS

NoSQL datu bāzes piedāvā vairākas priekšrocības, kuras vairs neļauj mūsdienu IT uzņēmumiem tās pilnībā ignorēt un turpināt lietot tikai relāciju datu bāzes. Nozarei ir nepieciešami *NoSQL* risinājumi, jo šī veida datu bāzes risina problēmas, kuras radās no relāciju datu bāžu pamatiem un projektējuma risinājumiem. Apskatīsim divas galvenās problēmas, ko *NoSQL* datu bāzes palīdz atrisināt.

Lietotņu izstrādes procesa produktivitāte: liela daļa no izstrādes procesa ir atmiņā glabāto datu struktūru kartēšana ar relāciju datubāzi. *NoSQL* datu bāze var piedāvāt datu modeli, kas labāk der lietotņu vajadzībām, vienkāršojot kartēšanas procesu un samazinot apjomu. Problēma ir veids, kā relāciju datu bāzes glabā datus. Programma gandrīz nekad neapstrādā datus tabulu formā, bet parasti dokumenta formā, kur eksistē tā saucamās „*HAS-A*" attiecības, kur vienas entītijas objekts satur sevī citas entītijas objektus.

Liela mēroga dati: uzņēmumi saprot priekšrocības savākt vairāk datus un apstrādāt tos daudz ātrāk. Tas ir grūti panākams, un, dažos gadījumos ar relāciju datu bāžu palīdzību pat neiespējams. Galvenais iemesls ir tas, ka relāciju datu bāzes ir paredzētas, lai strādātu uz vienas fiziskas mašīnas, bet parasti ir daudz izdevīgāk apstrādāt lielus datu apjomus ar smagu skaitļošanas slodzi klasteros, kas sastāv no vairākām mazākām un lētākām mašīnām.

IT nozares cilvēkiem, kuriem nav labas izpratnes par *NoSQL,* uzskata, ka šāda tipa datu bāzes ir pielietojamas tikai lielajos uzņēmumos ar milzīgu datu apjomu, vairākiem datu centriem, kas, visticamāk, atrodas vairākos kontinentos. *Google* un *Amazon* var apsvērt *NoSQL* kā jēdzīgu risinājumu, bet ne mazie uzņēmumi.

Šāds viedoklis ir ne vairāk ka stereotips. Ir jāsaprot, ka glabāt vairāk datus par lietotni ir daudz vieglāk un ērtāk, bet ar relācijas datu bāzēm, saglabājot daudz informācijas par lietotni, var ātri saskarties ar problēmu, kurai ir piesaistīts termins „Lielie dati". Ar *NoSQL*, pat relatīvi nelielos projektos un lietotnēs, var apstrādāt lielākus datu apjomus ātrāk, drošāk un ar mazāku datu sarežģītības struktūru.

Nevar arī neatzīmēt izstrādes atvieglošanu, izmantojot *NoSQL*. Latvijas valsts mēroga IT projekti var gūt labumu no šīs *NoSQL* īpašības, neatkarīgi no projekta lieluma.

ADF, jeb Lietotņu Izstrādes Ietvars, vēl vairāk atvieglo programmatūras izstrādi. Ja *NoSQL* risina vai pat novērš kartēšanas problēmu starp datu bāzi un lietotni, *ADF* uzdevums ir atvieglot

izstrādi citos aspektos. Ietvars pārņem uz sevi skata un kontroliera slāņu elementu struktūru veidošanu, kas nodrošina, ka izstrādātājiem nav jātērē laiks uz standarta klašu rakstīšanu.

ADF prot pilnībā noģenerēt lietotnes skeletu, lai izstrādātāji varētu koncentrēties uz biznesa loģikas izstrādi. Pārāk daudz no izstrādātāju laika paiet, rakstot, piemēram, saskarnes grafiskus elementus. *ADF* risina to, piedāvājot lielu kolekciju ar gataviem lietotāju saskarnes elementiem.

Vēl labāks izstrādātāju laika zuduma piemērs ir datu pastāvīguma implementācija. Pat izmantojot īpašas pastāvīguma bibliotēkas (*angļ.: persistance*), kā *Hibernate*[8] un *TopLink* [9], ir jātērē daudz laika, lai efektīvi sadarbotos ar datu bāzi. *ADF* piedāvā ļoti plašu rīku skaitu, kas palīdz noģenerēt visus nepieciešamos failus, kas ir vajadzīgi lietotnes saskarnei ar datu bāzi.

Pašlaik *Oracle ADF* nepiedāvā risinājumu, kas varētu apvienot *ADF* lietotni ar *NoSQL* datu bāzēm. Tā kā *ADF* vairākumā tiek izmantots lielajos uzņēmumos, tādi *NoSQL* labumi, kā lielu datu apstrādes vieglums un ātrums, kļūst sevišķi vērtīgi.

2012. gada oktobra beigās autors ir sazinājies ar vienu no *Oracle Red Samurai* pārstāvjiem no Lietuvas – Andrejus Baranovskis, kas atzīmēja, ka *ADF* un *NoSQL* integrācija var būt ļoti noderīga, un ka viņš nebija dzirdējis ne par vienu pašlaik eksistējošu integrācijas piedāvājumu. Tas vēl vairāk pārliecināja autoru par to, ka šī tēma ir aktuāla, un ka integrācijas risinājums var palīdzēt daudziem *ADF* speciālistiem visā pasaulē.

Darba pirmajā nodaļā tiek izpētītas un izanalizētas *NoSQL* tipa datu bāžu īpatnības un galvenie koncepti. Tiek aplūkoti *CAP* teorēmas principi, it sevišķi datu saskaņas vajadzība, jo tas būtiski ietekmē *NoSQL* risinājuma izvēli. Daudz laika tika veltīts klasterizācijas un replicēšanas analīzei, jo šie ir vieni no pašiem svarīgākajiem iemesliem, kāpēc izstrādātāji un datu bāžu administratori izlemj lietot *NoSQL* risinājumu.

Otrajā nodaļā tiek veikta analīze, kas ļauj pamatoti un apzināti izvēlēties vispiemērotāko *NoSQL* risinājumu integrācijas ar *Oracle ADF* ietvaru. Tiek apskatītas dažādu *NoSQL* risinājumu arhitektūras īpatnības. Balstoties uz paveiktās analīzes, tiek izvēlēta *MongoDB NoSQL* datu bāze. Tālākās sadaļās izvēlētais risinājums tiek apskatīts detalizētāk. Lai uzskatamāk parādītu izvēlētā risinājuma saskarni, tiek salīdzinātas *MongoDB* un *SQL* komandas.

Trešajā nodaļā tiek aprakstīti *OracleSQL* un *MongoDB* veiktspējas testu rezultāti. Šīs nodaļas galvenais mērķis ir pierādīt, ka izvēlētais *NoSQL* relāciju risinājums prot strādāt ātrāk par eksistējošo standarta risinājumu, kas ir *OracleSQL* relāciju datubāze. Šajā darbā veiktspējas testēšanai netiek pievērsts daudz uzmanības, jo tas nav viens no darba mērķiem vai uzdevumiem.

Ceturtā nodaļa apskata *Oracle* lietotnes izstrādes ietvaru lieliem uzņēmumiem – *ADF*. Tiek apskatīta ietvara trīs-slāņu arhitektūra, kā arī katra slāņa svarīgākās komponentes. Visvairāk uzmanības ir pievērsts tieši modeļa slānim, kas realizē savienojumu ar datu avotiem un pārvalda lietotnes mijiedarbību ar datu bāzi.

Piektajā nodaļā tiek apkopota visa informācija par *Oracle ADF* un *MongoDB* integrāciju. Vispirms tiek apskatītas ietvara paredzētās paplašinājuma iespējas, kas potenciāli var atļaut izstrādātājam izmantot alternatīvus datu avotus. Tiek izvēlēta viena no divām apskatītajām integrācijas pieejām. Sadaļās tiek definētas nepieciešamās komponentes, kā arī tiek aprakstīts šo komponenšu izveidošanas process. Tālāk tiek piedāvāts autora izstrādāts risinājums, kas atļauj sadarboties ar *MongoDB* datu bāzi efektīvāk, nekā ar 10gen piedāvāto Java dzini. Nodaļas beigās tiek demonstrēti vairāki *ADF* ietvara izmantošanas scenāriji, kuri izmanto autora piedāvāto integrācijas pieeju un autora izstrādāto pastāvīguma slāni.

Maģistra darba mērķis ir piedāvāt pieejas, kas atļauj integrēt Oracle ADF ietvaru ar kādu no NoSQL datu bāzēm. Mērķa sasniegšanai ir izvirzīti sekojoši uzdevumi: 1) apskatīt svarīgākos NoSQL konceptus un īpašības, sevišķi tos, kas būtiski atšķiras no relāciju datu bāzēs konceptiem; 2) identificēt problēmas, kuras var rasties, integrējot ADF un NoSQL; 3) apskatīt ADF datu slāņa paplašināšanas iespējas; 4) atklāt iespējamās pieejas ADF un NoSQL integrācijai.

1.

1. NOSQL KONCEPTI UN ĪPAŠĪBAS

NoSQL ir vēsturiski ļoti vāji definēts akronīms. Lai saprastu, ko tieši tas nozīmē, ir vērts zināt, no kurienes parādījās šis neoloģisms. Ir interesanti, ka pats pa sevi termins *NoSQL* parādījās 90. gados vienas relāciju datu bāzes nosaukumā – *Strozzi NoSQL*. Šī datu bāze glabāja datus *ASCII* failos, kur katrs ieraksts ir attēlots ka teksta rindiņa, un īpašības ir atdalītas ar tabulācijas simbolu. '*NoSQL*' nozīmēja, ka datubāze neizmantoja *SQL* ka vaicājumu valodu, bet gan *UNIX shell* komandas.

NoSQL termins, kādu mēs to saprotam šodien, bija radīts 2009. gadā 11. Jūnijā San Francisko. To izdomāja programmatūras izstrādātājs Johan Oskarsson, kas toreiz strādāja Londonā. *Google* izstrādāta datu bāze *BigTable* un *Amazon* izstrādāta *Dynamo* šokēja un iedvesmoja daudzus cilvēkus IT nozarē visā pasaulē, kas izskaidro šī laika perioda vairākas konferences par alternatīviem datu glabāšanas iespējam. Johan, ka arī daudzi citi, gribēja uzzināt vairāk par šīm jaunajiem datu bāzēm. Tāpēc viņš izlēma norīkot diskusiju, kur jebkurš, kas gribēja, varēja pastāstīt par datu bāzēm, kuras vēlāk tiks apvienotas zem termina *NoSQL*.

Lai diskusija izraisītu interesi, bija nepieciešams izdomāt īsu un pieminamo, un ko katram potenciālām apmeklētājam nevajadzētu ievadīt meklētājprogrammās. Johans gribēja labu atsauces birku, ko varētu izmantot 2009. gada jau ļoti populārā komunikācijas tiklā Twitter. Viņš uztaisīja balsojumu *Cassandra IRC* kanālā, un izvēlēja '*NoSQL*' terminu, ko piedāvāja Eric Evans – izstrādātājs, kas toreiz strādāja *Rackspace* kompānijā.

Kaut gan jaunais termins slikti aprakstīja jaunas alternatīvas datu glabāšanas programmas, tas labi derēja atsauces birkas vajadzībām priekš *Twitter* sistēmas. Toreiz '*NoSQL*' bija domāts ka vienas diskusijas nosaukums.

Termins, ka tas bieži notiek ar kaut ko skanīgo, bet ne īsti labo, izplatījās ar zibenīgo ātrumu, un kļuva par nosaukumu visai tehnoloģijas parādībai, kaut gan tas tomēr neko īsti neaprakstīja. Oriģināls diskusijas plāns bija apspriest „atvērta koda, dalītas apstrādes, ne-relāciju datu bāzes". Līdz šīm brīdim nav vispārpieņemtas definīcijas, tapāt, ka nav institūciju, kas varētu pilnvaroti piedāvāt tādu definīciju. Tāpēc vienīgais, kas ir jāsaprot ar terminu *NoSQL*, ir šī tipa datu bāžu īpašību kopums.

Pirmkārt, liekas acīmredzami, ka *NoSQL* datu bāzes neizmanto *SQL* vaicājumu valodu. Dažām no *NoSQL* datu bāzēm ir savas vaicājumu valodas, kas, protams, ir līdzīgi *SQL* valodai, kas būtiski atvieglo apmācības procesu. Bet ne viena no *NoSQL* datu bāzēm pagaidām nepiedāvā savu standarta vaicājumu valodu.

Cita īpašība ir, ka *NoSQL* datu bāzes ir atvērta koda projekti. Kaut gan *NoSQL* pielieto arī aizvērta koda sistēmām, vismaz sākumā termins attiecās tieši uz atvērta koda datu bāzēm.

Visas datu bāzes, kas tika apspriestas oriģinālajā diskusijā 2009. gada Jūnijā, piedāvāja savus klasterizācijas risinājumus. Tas būtiski ietekmēja šo datu bāžu datu modeļus un datu saskaņas konceptus. Tas nenozīmē, ka visas *NoSQL* datu bāzes ir domātās darbam klasteros. Grafu datu bāzes ir viens no *NoSQL* datu bāžu tipiem, bet to datu modelis ir ļoti līdzīgs relāciju datu bāžu modelim. Grafu datu bāzes māk efektīvi apstrādāt datus ar vairākām sarežģītām saistībām.

NoSQL datu bāzes strādā bez stingri noteiktas shēmas, kas atļauj brīvi pievienot ierakstiem laukus, nemainot datu bāzes struktūru. Tas ir ļoti noderīgi darbam ar netipiskiem datiem, kur lauki atšķiras starp ierakstiem. Relāciju datubāzes risināja šo problēmu, ieviešot tādus lauku nosaukumus, ka `customField6`, vai arī izmantojot modificējamo lauku tabulas, ar kuriem ir grūti strādāt, un kuras ir ļoti grūti saprotamas.

Tomēr saprast terminu *NoSQL* ka „ne tikai *SQL"* nav pilnīgi korekti. Šīm traktējumam ir savi mīnusi. Pirmkārt, ja mēs saprotam ne tikai *SQL* zem šī termina, tam jābūt *NOSQL* (*no angļ.: Not Only SQL*), nevis *NoSQL*. Otrkārt, *Oracle* un *Postgres* datu bāzes pārvaldības sistēmas arī varētu iekļaut 'ne tikai *SQL*' kopā.

Saprast *NoSQL* ka „ne tikai *SQL*" ir tomēr daudz labāk, nekā 'teiksim nē *SQL*ām'. „Ne tikai *SQL*" atver durvis jaunai parādībai, kas ir zināma, ka „*Polyglot persistence*". Šī pieeja piedāvā izmantot dažāda tipa datu bāzes dažādām vajadzībām. Visinteresantākais ir tas, ka tādu pieeju var izmantot vienas lietotnes robežās. Protams, vajag saprast, ka dažāda tipa datu bāžu izmantošana vienas lietotnes robežas ir iespējama tikai tad, ja lietotne izmanto lietotņu datu bāzi, un ne integrāciju datu bāzi.

Kaut gan klasterizācija ir viens no svarīgākajiem *NoSQL* popularitātes iemesliem, tas nemaz nav vienīgais. Gandrīz tik pat svarīga problēma, ko risina *NoSQL* datu bāzes, ir kartēšanas problēma (no angļ. val.: *impedance mismatch*).

Tas, ka uzņēmumi saprot labumu glabāt un apstrādāt lielus datu apjomus, piespieda IT nozares cilvēkus domāt inovatīvi par savām datu glabāšanas vajadzībām. Komandas, kas iepriekš izmantoja tikai relāciju datu bāzes, tagad sāk apsver *NoSQL* iespējas.

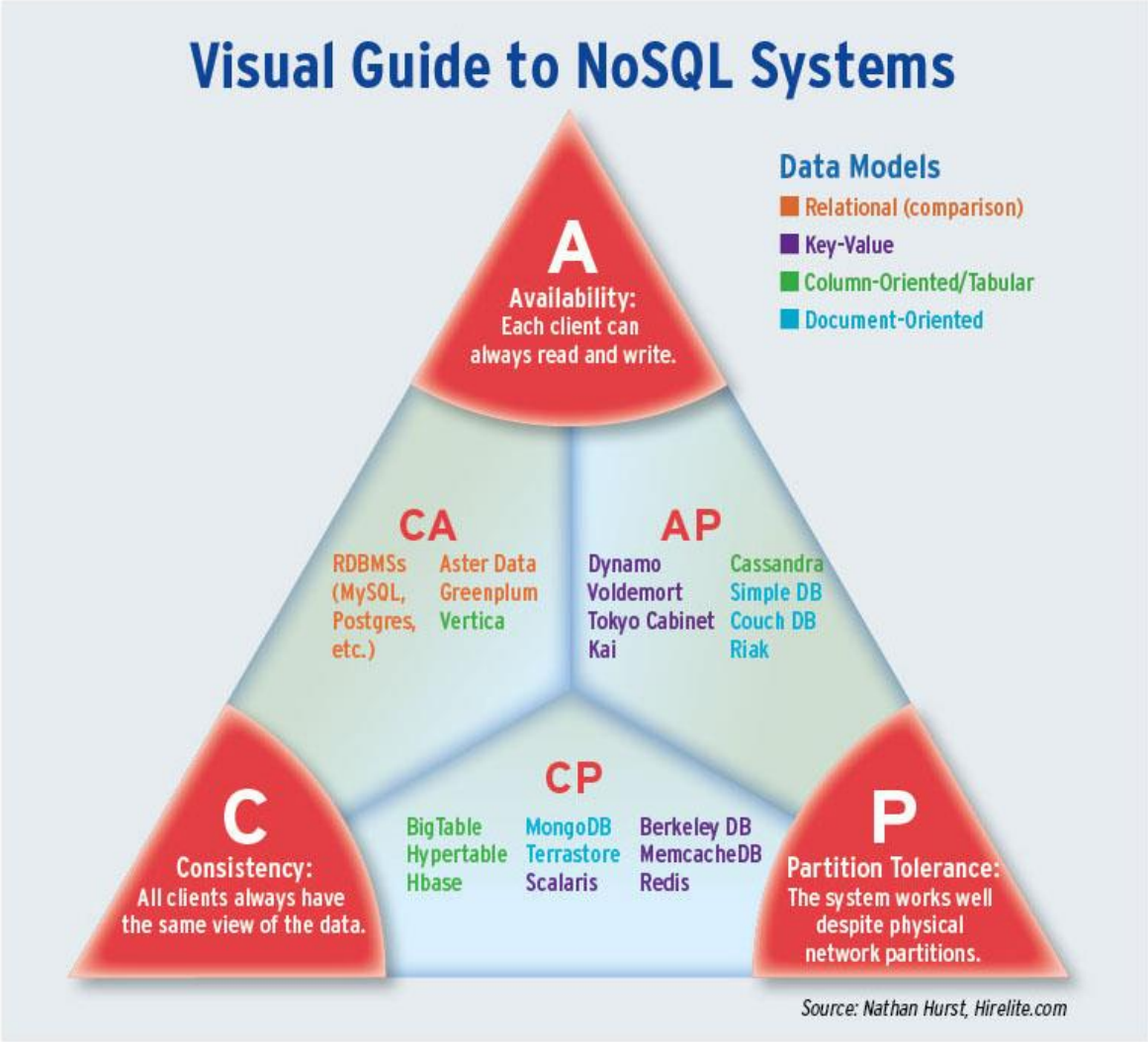
Ja vajag atzīmēt divas vissvarīgākos iemeslus, kāpēc uzņēmumiem ir jāapsver *NoSQL* datu bāzes, es viennozīmīgi atzīmētu *NoSQL* datu bāžu iespēju apstrādāt tāda izmēra datus, kam ir nepieciešams klasteris, un programmatūras izstrādes atvieglošana, risinot kartēšanas problēmu starp datu bāzes un atmiņa glabātiem datiem.

1.1. CAP teorēma

CAP teorēma ir viens no galvenajiem pieradījumiem, kāpēc var būt nepieciešams atslābināt datu saskaņu. 2000. gadā notika 19. simpozijs par sadalīto datošanu [5] (*no angļ.: Symposium on Principles of Distributed Computing*), kur zinātnieks un Kalifornijas universitātes profesors Eriks Breveris (Eric A. Brewer) pirmoreiz noformulēja sekojošo priekšrakstu [6]: jebkāda sadalītas datošanas sistēma, jeb klasterizēta sistēma, spēj nodrošināt tikai divus no šīm trim galvojumiem:

- Datu saskaņa (*no angļ.: Consistency*) – visi mezgli redz vienādu datu versiju;
- Pieejamība (*no angļ.: Availability*) – garantija, ka katrs pieprasījums vienmēr saņems atbildi par to, vai tas izpildījās vai nē;
- Nodalījumu tolerance (*no angļ.: Partition tolerance*) – sistēma turpina darboties, ja viena klastera daļa nevar sazināties ar citu klastera daļu.

Attēlā 1.1. vairākas *NoSQL* datu bāzes ir sadalītas pa grupām – 1) CA – datu saskaņa un pieejamība, 2) CP – datu saskaņa un nodalījumu tolerance, un 3) AP – pieejamība un nodalījumu tolerance. Katra datu bāze koncentrējas uz diviem no trim CAP teorēmas galvojumiem.



1.1. att. **NoSQL sistēmas attēlojums [5]**

Viena servera sistēmas ir klasisks CA sistēmas piemērs, kas nozīmē, ka sistēmā ir augsta līmeņa datu saskaņa, augsta pieejamība un ļoti zema nodalījumu tolerance. Tas ir pilnīgi pieņemami, jo vienai fiziskai mašīnai nevar notikt tīkla nodalījums. Šādu sistēmu var apskatīt kā klasteri ar vienu mezglu – tas vai nu darbojas, un ir pieejams, vai nu nedarbojas. Relāciju datu bāžu sistēmu vairākums strādā šāda režīmā.

Ir teorētiski iespējams uztaisīt CA klasteri, bet tas nozīmēs, ka tīkla nodalījumu gadījumā, visi mezgli pārtrauks darbību un ne viens klients nevarēs sarunāties ne ar vienu mezglu. Tas nozīmē, ka ir jānodrošina, lai tiklu nodalījums šajā klasterī notiek reti un pilnīgi. Šāds gadījums ir

iespējams, ja visi klastera mezgli atrodas vienā datu centrā, bet tas ir ļoti neizdevīgi. Ir svarīgi atzīmēt, ka klastera nodalījumu gadījuma ir nepieciešams ātri identificēt faktu, ka notika nodalījums, kas nav triviāls un pat grūts uzdevums.

Apskatīsim klasteru, kur datu sadales līmenis tiek pakāpeniski samazināts lai uzlabotu nodalījuma toleranci. Pieņemsim, ka viesnīcu tīkla rezervācijas programma strādā uz divu mezglu klastera – viens atrodas Kanādā, un viens Francijā. Jā ir nepieciešam nodrošināt datu saskaņu, ja lietotājs grib norezervēt istabu Kanādas filiālē, šīm mezglam ir jāsazinās ar Francijas mezglu pirms akceptēt rezervāciju. Abiem mezgliem ir jāsaskaņo rakstīšanas pieprasījums. Tas, savukārt, nozīmē, ka, tīkla nodalījumu gadījumā, neviens no filiālēm nevar patstāvīgi rezervēt istabas, kas norādu uz zemu pieejamību.

Šo situāciju ir iespējams izlabot, ja tiek izvēlēta viena meistarmašīna, ar kuru tiek saskaņotas visas rezervācijas. Pieņemsim, ka Francijas mezgls ir meistars. Tīkla nodalījumu gadījumā, klients no Francijas var uztaisīt rezervāciju. Klients no Kanādas redzēs nesaskaņotus datus, bet nevarēs uztaisīt rezervāciju, kas nodrošina, ka nenotiks loģiskas nesaskaņas konflikts.

Lai vel vairāk palielināt pieejamību, ir iespējams atļaut abiem mezgliem akceptēt rezervācijas, pat ja ir noticis tīkla nodalījums. Šī gadījumā draud sistēma, ka Francijas un Kanādas klienti norezervēs pēdējo istabu. Tomēr, atkarībā no šīs viesnīcas politikas, tas varētu būt pieņemams. Vairākas ceļošanas kompānijas pārdod vairākas vietas, nekā fiziski eksistē, lai samazināt zaudējumus no atcelšanas gadījumiem. Līdzīgi, vairākas viesnīcas atstāj dažās tukšas istabas, kad visas pieejamas istabas jau ir norezervētas. Tas atļauj iedot vienam no ar konfliktējošo rezervāciju vienu no šīm rezerves istabām, vai uztaisīt izņēmumu, ja rezervāciju pieprasa klients ar augstāku statusu.

Ir jāsaprot, ka var nodrošināt arī trīs *CAP* teorēmas radītājus, bet vienā vai divas no tiem būs samazinātā līmenī. Un visbiežāk *NoSQL* sistēmās tiek samazināts tieši datu saskaņas līmenis.

1.2. Datu saskaņa

Viena no lielākam atšķirībām starp centralizētu relāciju datu bāzes serveri, kas atrodas uz vienas fiziskas mašīnas, un klasterizētu *NoSQL* serveri ir datu saskaņas jautājumu risinājumi. Relāciju datubāzes izmanto stipras saskaņas pieeju, un problēmas, kuras parādās *NoSQL* pasaulē, tur neeksistē.

Ņemot vērā *CAP* teorēmu, relāciju datu bāzes samaksā par stipro saskaņu un augsto pieejamību ar ļoti sliktu nodalījumu tolerance. Ir svarīgi apspriest saskaņas atslābināšanas iespējas, kā veidu uzlabot nodalījumu toleranci.

1.2.1. Atjauninājumu saskaņa

Apskatīsim piemēru: divi finanšu nodaļas darbinieki grib atjaunot darbinieces apgādībā esošo cilvēku skaitu. Abi darbinieki zina, ka iepriekš darbiniecei apgādībā bija viens cilvēks. Katrs no darbiniekiem palielina apgādībā esošo cilvēku skaitu uz 1. Ja finanšu sistēma nebrīdina, ka dati, kurus redz otrais darbinieks, tika samainīti, kamēr tas uzspieda uz pogu 'Saglabāt', apgādībā esošo cilvēku skaits samainīsies no 1 uz 3. Šo parādību sauc par rakstīšanas-rakstīšanas konfliktu (*no angļ.: write-write conflict*).

Eksistē vairāki veidi datu saskaņas nodrošināšanai, kuri tiek sadalīti divās loģiskās grupās: pesimistiskie un optimistiskie. Pesimistiskas datu saskaņas gadījuma konfliktu situācijas nekad netiek pielaistas. Visbiežāk to nodrošina, izmantojot atjaunojamo datu bloķēšanu, kas nozīmē, ka vienlaikus izmaiņas var veikt tikai viens klients.

Bieži lietojama optimistiska pieeja ir izmantot nosacīto atjauninājumu, kad pirms jebkura atjauninājumu pieprasījuma datu bāze pārliecinās, ka dati nebija modificēti no iepriekšēja lasīšanas pieprasījuma laika. Finanšu darbinieku piemēra gadījumā, otrais darbinieks, uzspiežot pogu 'Saglabāt', redzētu kļūdas paziņojumu, kas informētu viņu par to, ka dati tika modificēti, un ka ir vajadzīgs apstiprinājums atjauninājumu pieprasījuma izpildei.

Cita optimistiska pieeja ir reģistrēt abus konfliktējošus pieprasījumus, un apstrādāt tos. Šo pieeju aktīvi izmanto versiju kontroles sistēmas, kur kodu izpildes konflikti ir notiek regulāri. Nākamais solis arī nāk no versiju kontroles sistēmām – vajag saplūdināt jeb apvienot divu konfliktējošo pieprasījumu rezultātus. Sistēmas var parādīt lietotājam abus vērtības variantus, un

likt viņam izvēlēties jauno vērtību. Ir gadījumi, kad lietotne prot atrisināt konfliktu patstāvīgi, bet tam vienmēr ir nepieciešama īpaši veltīta implementācija.

Pirmoreiz sastopoties ar šīm saskaņas problēmām, cilvēki mēdz izvēlēties pesimistisko pieeju, un, lai gan daudzos gadījumos tas ir pieņemami, pesimistiska saskanas nodrošināšanas pieeja var būtiski samazināt sistēmas atsaucību un izraisīt strupsaķeres, kurus ir ļoti grūti identificēt un atkļūdot.

Replicēšana ievērojami palielina saskaņas problēmu skaitu. Ja dažādi klastera mezgli satur datus, kurus var atjaunināt neatkarīgi no citiem mezgliem, noteikti parādīsies saskaņas problēmas, ja nav implementēti veidi, kas tos specifiski atrisina. Viens no veidiem ir izmantot tikai vienu mezglu rakstīšanai. Šādu replicēšanas modeli sauc par Meistars-Kalps replikāciju, un tā tiek apskatīta replicēšanas nodaļā.

1.2.2. Lasīšanas saskaņa

Apskatīsim sekojošo situāciju: datu bāzē glabājas informācija par pirkumu, kur katrs pirkums satur informāciju par katru nopirkto objektu un kopējo pirkuma summu. Ja pirmais klients pievieno pirkumam vel vienu objektu, ir jāpārrēķina kopēja pirkuma summa. Otrais klients var nolasīt pirkuma kopējo summu pirms atjaunota pirkuma summa ir ierakstīta datu bāzē, un saņemt veco pirkuma summu. Šādu situāciju sauc par nesaskaņotu datu ielasīšanu vai lasīšanas-rakstīšanas konfliktu.

Loģiska datu saskaņa ir pārliecināšanas par to, ka dažādi datu fragmenti ir saskaņoti viens ar otru. Lai izvairīties no lasīšanas-rakstīšanas konfliktiem, relāciju datubāzes izmanto transakcijas. Pieņemot, ka pirmais klients apvieno abus rakstīšanas pieprasījumus vienā transakcijā, otrais klients ielasīs informāciju par abiem pirkuma elementiem vai nu pirms atjauninājuma pieprasījuma izpildes, vai nu pēc tā.

Strādājot ar klasteriem, parādās pilnīgi cits datu nesaskaņas veids. Apskatīsim citu situāciju: e-veikals strādā uz trim klastera mezgliem, viens no kuriem atrodas ASV, viens Austrālijā un viens Latvijā. Lietotājs no Latvijas nopērk pēdējo produkta vienību. Bet atjauninājums ienāk ātrāk ASV mezglā, nekā Austrālijas mezglā. Lietotājs no Austrālijas redz, ka prece ir pieejama, bet lietotājs no ASV redz, ka visas preces ir izpirktas. Šo datu nesaskaņas veidu sauc par replicēšanas saskaņu – pārliecināšanas par to, ka datu vienībai ir vienāda vērtība, pat ja tā tiek ielasīta no dažādiem klastera mezgliem.

1.2.3. Datu saskaņas atslābināšana

Datu saskaņas nodrošināšana ir ļoti svarīgs process jebkuras datu bāzes dizainā. Bet ļoti bieži ir tomēr nepieciešami atteikties no ļoti stingras saskaņas. Ir vienmēr iespējams uzprojektēt sistēmu, kur nevar notikt datu nesaskaņas gadījumi, bet tas izraisīs būtiskus zudumus citās sistēmas pazīmēs. Tāpēc bieži ir nepieciešams pazaudēt datu saskaņas ziņa, lai iegūt kaut ko citu. Ir jāņem vērā, ka daudziem domēniem ir dažādi datu nesaskaņas tolerances līmeņi, un šos līmeņus ir jāņem vērā sistēmas saskaņas projektēšanas brīdī.

Saskaņas līmeņa samazināšana notiek arī viena servera relāciju datu bāžu sistēmās, kur galvenais datu saskaņas nodrošināšanas rīks ir transakcija. Tomēr, transakciju sistēmas bieži piedāvā iespēju atslābināt izolācijas līmeņus, kas atļauj pieprasījumiem ielasīt datus, kuri vel nav ierakstīti datu bāzē. Daudzas sistēmas ir spiestas atslābināt saskaņu no augstākā izolācijas līmeņa (serializācija) lai dabūt optimālo veiktspēju.

Vairākas sistēmas pilnībā atteicas no transakcijām, jo transakciju efekts uz veiktspēju ir pārāk liels. *MySQL* datu bāzei bija liela popularitātē laikā, kad tā pavisam neatbalstīja transakcijas. Izstrādātāji bija tika apmierināti ar *MySQL* ātrumu, ka tie bija ar mieru strādāt bez transakcijām. Tādai milzīgai sistēmai, ka *eBay*, kas strādāja ar *Pritchett* datu bāzi, bija jāatteicas no transakcijām, lai strādāt pieņemamā līmenī – tas ir aktuāli, ņemot vērā šīs sistēmas klasterizācijas nepieciešamības mērogu [1]. Tas parāda, ka daudzas sistēmas, kas apkalpo lielā biznesa uzņēmumus, ir spiestas strādāt bez transakcijām.

1.3. Replicēšana

Replicēšana eksistē gan *NoSQL*, gan relāciju datu bāzu pasaulē. Par replicēšanu *NoSQL* pasaulē ir jāsaprot datu redundantu kopiju glabāšana. Šo parādību bieži sauc par *RAID*, no angļu valodas – redundant array of independent disks, jeb neatkarīgo disku redundants masīvs. Šo terminu bieži izmanto, runājot par dublējuma procesu.

Ja dublējumu procesā, un pavisam kopīgā izpratnē, *RAID* ir fizisko cieto disku masīvs, datu bāžu ziņa *RAID* var būt arī virtuālo mašīnu loģisks masīvs, vai arī neatkarīgo serveru kopa, kas ir apvienoti tā saucamajos replicēšanas kopās.

Izmantot tikai cietos diskus priekš datu bāzes replicēšanas ir diezgan nedroši. Viens no vis biežākiem iemesliem, kāpēc datu bāzes fiziska servera mašīna vairs nav pieejama, ir barošanas bloka sabrukums. Tad visa servera mašīna izslēdzas, un datu bāze kļūst nepieejama. Tāda servera konfigurācija ir domāta datu pieejamībai un drošībai, gadījumā, ja fiziski sabruks viens no *RAID* masīvā cietajiem diskiem.

Vienā no svarīgākām datu bāzes augstas pieejamības radītājiem, ir ka datu bāzei nav vienota sabrukumu punkta (*no angļ.: single point of failure*). Vairākas *NoSQL* datu bāzes atbalsta replikāciju, kur katrs *RAID* masīva elements ir datu bāzes process. Šie datu bāzēs procesi ir apvienoti replicēšanas kopās. Katrs datu bāzes process var dzīvot uz atsevišķas fiziskas mašīnas. Tas nozīmē, ka pat ja sabrūk servera mašīnas barošanas bloks, vai serveru telpā notiek ugunsgrēks, failu redundantas kopijas var glabāties uz citas fiziskas mašīnas citā datu glabāšanas centrā vai citā uzņēmumu ofisā.

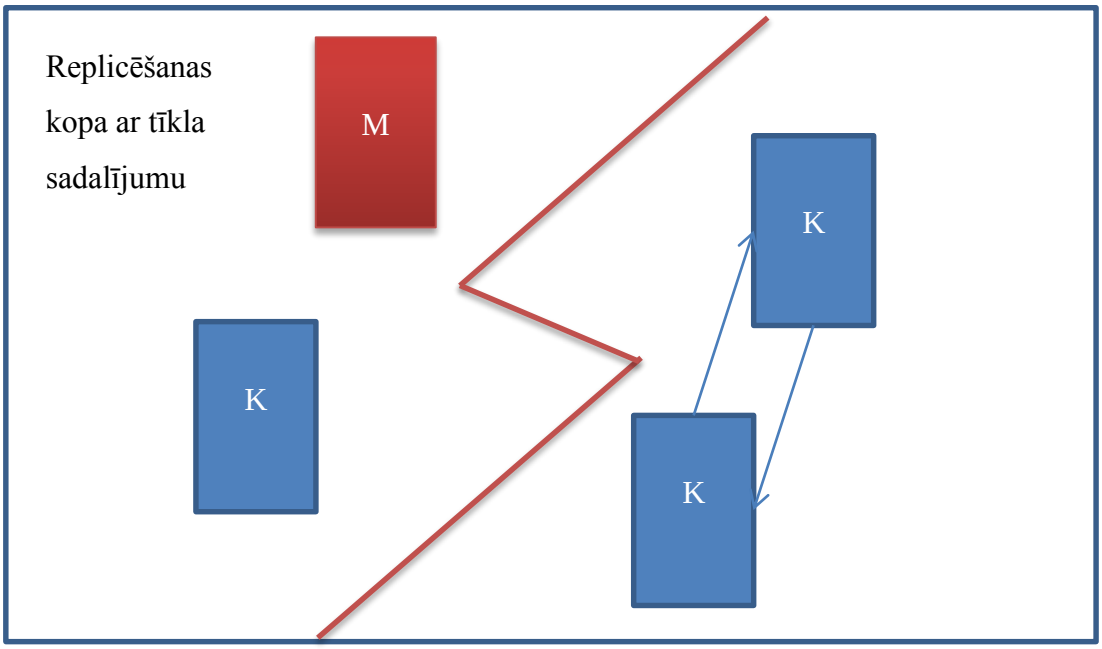
1.3.1. Meistars-Kalps replicēšana

Replicēšanā eksistē jēdziens replicēšanas faktors, kas apraksta cik redundantu kopiju un, attiecīgi, fizisko serveru, tiek izmantots replicēšanai. Par standartu, un vispopulārāko replicēšanas faktoru, tiek uzskatīts replicēšanas faktors 3. Ja eksistē iespēja, ka var sabrukt visi trīs serveri, vai dati ir pārāk svarīgi, un datu saskaņai ir primārā nozīme, tad var izmantot replicēšanas faktoru 5, vai arī 7, lai nodrošinātu gandrīz absolūtu datu saglabātību.

Viens no populārākiem replicēšanas tipiem, kas ir cieši atkarīgs no replikāciju faktora, ir meistars-kalps replicēšana (*no angl.: master slave replication*) – viens no replicēšanas kopas dalībniekiem ir meistars, un visi pārējie dalībnieki ir kalpi. Ja meistara mašīna sabrūk, notiek balsošanas process, par to, kurai mašīnai tagad jābūt par jaunu meistaru. Lai kāda no mašīnām varētu kļūt par jaunu replicēšanas kopas meistaru, tai ir jādabū balsu vairākumu no citiem kopas dalībniekiem. Šo procesu sauc par automātisko atveseļošanu (*no angļ: automatic failover, automatic recovery*), kas nozīmē, ka, meistara mašīnas sabrukšanas gadījumā, nav nepieciešama datu bāzes administratora darbība.

Kad replicēšana strādā automātiskas atveseļošanas režīmā, vienmēr izmanto nepāra skaitļa replicēšanas faktoru. Tas ir svarīgi, jo, lai viens no kalpiem kļūtu par jauno meistaru replicēšanas kopas robežās, tam ir jādabū balsu vairākums. Apskatīsim piemēru ar šādām īpašībām:

- Replikāciju faktors ir 4;
- Notiek tīkla sadalījums – 2 serveri var redzēt viens un otru, bet nevar redzēt pērējus divus serverus (att. 1.2.);
- Meistara mašīna sabrūk.



1.2. att. **Replicēšanas kopa ar tīkla sadalījumu**

Ne vienā no tīkla nodalījumiem nav pietiekami daudz balsu, lai sasniegtu balsu vairākumu un izvēlēties jaunu meistaru. Tāpēc ir svarīgi, lai replicēšanas faktors ir nepāra skaitlis. Ja replicēšanas kopas faktors būtu 5, viena no divām tīklu sadalījuma pusēm būtu trīs mašīnas, kuriem būtu balsu vairākums. Viena no šīm mašīnām tiktu izvēlēta par jaunu meistara mašīnu. Kad vecā meistara mašīna ieslēdzās, tā pieprasīs informāciju no pārējiem replikācijas kopas mezgliem. Ja veca meistara mašīna konstatēs, ka kopā eksistē jauna meistara mašīna, tā kļūs par kalpu mezglu. Pretējā gadījumā vecā meistara mašīna atrodas tīklu sadalījumu pusē, kurā atrodas mazākums mezglu, un jauna meistara mašīna netiks izvēlēta, kamēr nepazudīs tīkla sadalījums.

1.3.2. Sinhrona un asinhrona replicēšana

Replicēšanas procesā ir jāizveļas vel viens svarīgs režīms – gaidīt, kamēr visi replicēšanas kopas dalībnieki apstiprinās, ka ir saņēmuši datus, vai negaidīt. Sinhronas replicēšanas gadījumā datu bāze atbildēs klientam par veiksmīgo ierakstīšanas pieprasījumu tikai tad, kad vairākums, vai visi replicēšanas kopas dalībnieki paziņos, ka ir saņēmuši datus. Šī datu bāzes konfigurācija var būt nepieņemama lietotnes ātrdarbības dēļ, jo tā manāmi palielina katra ierakstīšanas pieprasījuma izpildes laiku.

Tāpēc vairākas *NoSQL* datu bāzes izmanto asinhrono vai daļēji asinhrono replicēšanu. Ar asinhrono replicēšanu ir jāsaprot tādu režīmu, kad datu bāze apstiprina rakstīšanas pieprasījumu kā izpildīto tad, kad dati ir ierakstīti vismaz vienā replicēšanas kopas mezglā. Datu bāze negaida apstiprinājumu no citiem mezgliem un turpina darbu. Replicēšana notiek fonā, pēc apstiprinājuma aizsūtīšanas klientam.

Sistēmas ar sinhrono replikāciju piedāvā augstāku datu saglabātību – lai pazaudētu datus, vajag lai sabruktu visas replicēšanas kopas mašīnas. Asinhronas replicēšanas labums, savukārt, ir veiktspējas palielinājums, jo nav jāgaida apstiprinājums no katras kopas mašīnas. Tomēr datu saglabātības līmenis sistēmās ar asinhrono replicēšanu ir ļoti zems. Ja mašīna, kura saņēma jaunus datus un nosūtīja klientam apstiprinājumu, sabrūk pirms tā paspēj nosūtīt jaunus datus citām replicēšanas kopas mašīnām, dati tiek pazaudēti.

Līdzīgi, kā tas notiek ar datu saskaņu, eksistē domēni un sistēmas, kur zems datu saglabātības līmenis ir pieņemams.

1.4. Klasterizācija

Lai atļautu datu bāzes serverim strādāt ar lieliem datiem, kuru apjoms strauji pieaug, servera mašīna ir jāmodernizē – ir jāpalielina atmiņas izmērs, instalēt jaudīgāku procesoru, un, kas ir ļoti svarīgi, palielināt datu nesēju skaitu. Šo procesu sauc par vertikālo mērogošanu.

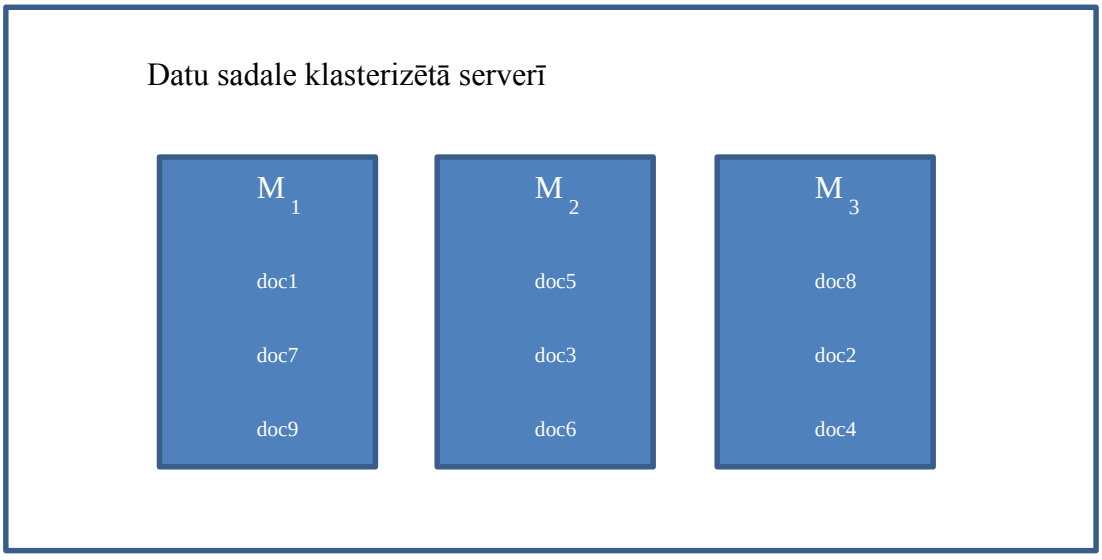
Problēma ar šo metodi ir tas, ka, lai apstrādātu relatīvi lielus datu apjomus, ir vajadzīgs serveris ar tik jaudīgu procesoru, un ar tik lielo datu nesēju skaitu, ka šī servera mašīnas iegādes un apkalpošanas cena kļūst pārāk liela.

Otrā problēma ir, ka izmantojot vienu fizisko servera mašīnu, šī mašīna kļūst par vienoto sabrukuma punktu. Tas nozīmē, ka datu bāzes lietotāji var pilnībā zaudē pieeju pie datu bāzes, ja notiek viena no šīm situācijām:

- Elektrības zudums;
- Barošanas bloka bojājums;
- Ugunsgrēks.

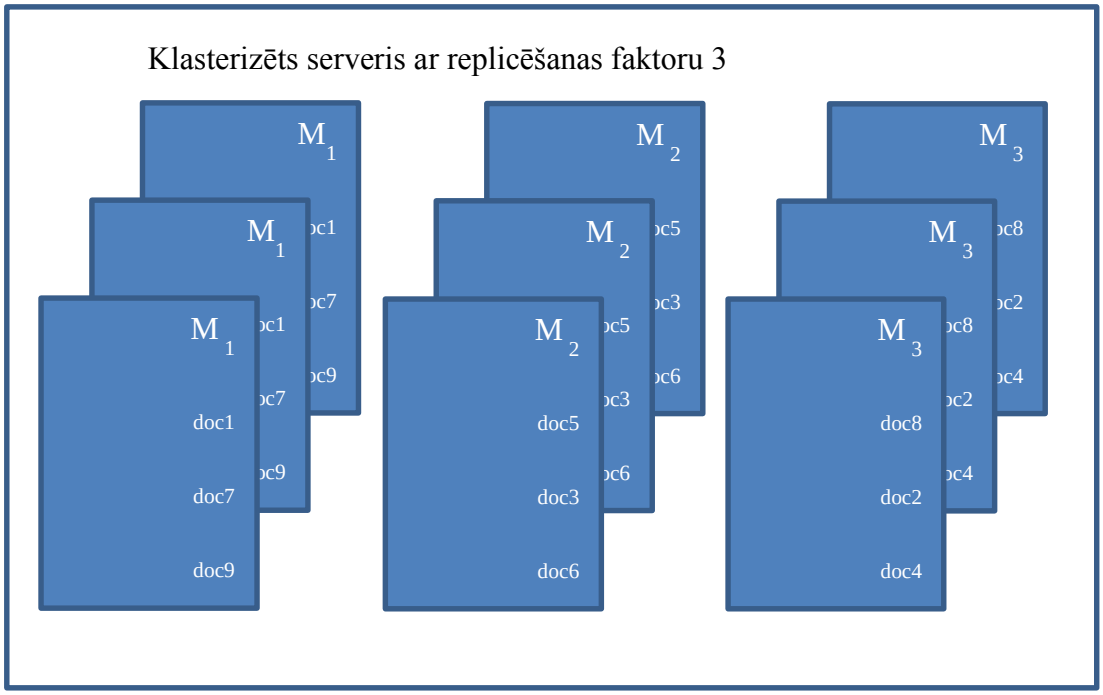
Sistēmās, kur augsts pieejamības līmenis ir ļoti svarīgs, ir daudz izdevīgāk lietot horizontālo mērogošanu. Horizontāla mērogošana paredz, ka datu bāze strādā uz vairākām, relatīvi lētām mašīnām. Izmantojot šādu konfigurāciju, gadījumā, ja viena no mašīnām sabrūk, datu bāzē paliek strādājošā stāvoklī.

Klasteri, atkarībā no replicēšanas mašīnām, glabā dažādus dokumentus. Tas nozīmē, ka viens fizisks dokuments vienā laikā var atrasties vienā un tikai vienā klasterī. Pastāv iespēja, ka datu bāzes darba laika, dokuments tiks pārvietots no viena klastera uz citu. Šo procesu klasterizācijā sauc par datu bilances uzturēšanu. Attēlā 1.3. ir parādīts klasteris bez replicēšanas.



1.3. att. **Datu sadale klasterizētā serverī**

Tā kā katrs unikāls dokuments glabājās tikai vienā fiziskā mašīnā, ir ļoti svarīgi lietot gan klasterizāciju, gan replicēšanu. Attēlā 1.4. ir parādīts klastera piemērs ar replicēšanu.



1.4. att. **Klasterizēts serveris ar replicēšanas faktoru 3**

Kā var redzēt attēlā 1.4., katrs klasteris satur unikālus dokumentus, bet katra klastera replicēšanas kopas dalībnieki glabā redundantu informāciju. Šī konfigurācijā ir ļoti populārā, jo klasterizācija nodrošina servera vertikālo mērogošanu, un replicēšana nodrošina augstu datu pieejamību.

Ir svarīgi atzīmēt, ka reālajās sistēmās ir diezgan bezjēdzīgi izmantot replicēšanu ar replicēšanas faktoru 3, ja eksistē tikai trīs klastera mezgli. Ar šādu konfigurāciju pilnībā pietiek ar replicēšanas faktoru 2, bet, ja mezglu skaits klasterī pārsniedz simtus mašīnu, pastāv liela varbūtība, ka dienas laikā sabrukt var abas replicēšanas kopas mašīnas.

Klastera mezglu atslēgas

Meistars-kalps replicēšanas gadījumā, rakstīšanas pieprasījumus saņem tikai meistarmašīna, un tālāk dati, vai arī pieprasījumu komandas, tiek sūtīti citiem replicēšanas kopas dalībniekiem. Viens no galvenajiem jautājumiem klasterizētā *NoSQL* datu bāzes serveri ir kurā klastera mezglā rakstīt jaunus dokumentus. Var likt jaunus dokumentus mezglos, kur ir visvairāk brīvas vietas uz datu nesējiem. Šīs veids ir visvieglākais un intuitīvs, bet tas neļauj realizēt datu bāzes klastera pilno potenciālu.

Viens no veidiem, kā izlemt uz kuru mezglu sūtīt jaunu dokumentu, ir izmantojot mezglu atslēgas. Pieņemsim, ka datu bāzes administratoram ir jāstrādā ar dokumentu datu bāzi, kurā glabājas kolekcija „Lietotāji". Par mezglu atslēgu ir jāizveļas kolekcijas elementa kādu no īpašībām, piemērām, uzvārdu.

Kad ir izvēlēta mezglu atslēga, ir jādefinē atslēgās diapazons. Piemērām, pirmajā klasterī mēs glabāsim visus lietotāju, kuru uzvārdi sākas ar burtiem no A līdz H. Klasterim ar mezglu skaitu 3 atslēgu diapazoni var izskatīties tā:

- M1: [A..H);
- M2: [H..R);
- M3: [R..Ž].

Šāda konfigurācija dod datu bāzei iespēju optimizēt lasīšanas pieprasījumus, un nesūtīt pieprasījumus uz katru klastera mezglu. Apskatīsim lasīšanas pieprasījuma piemēru:

db.users.find({ "surname" : "Gates" });

Šāds pieprasījums tiks automātiski nosūtīts tikai uz pirmo klastera mezglu, jo citi mezgli nevar saturēt lietotājus ar uzvārdiem, kas sākas ar burtu „G". Analizējot visbiežākās lasīšanas pieprasījumus, kuras tiek veiktas pret konkrēto kolekciju, datu bāzes administrators var izvēlēties vispiemērotāko mezglu atslēgu.

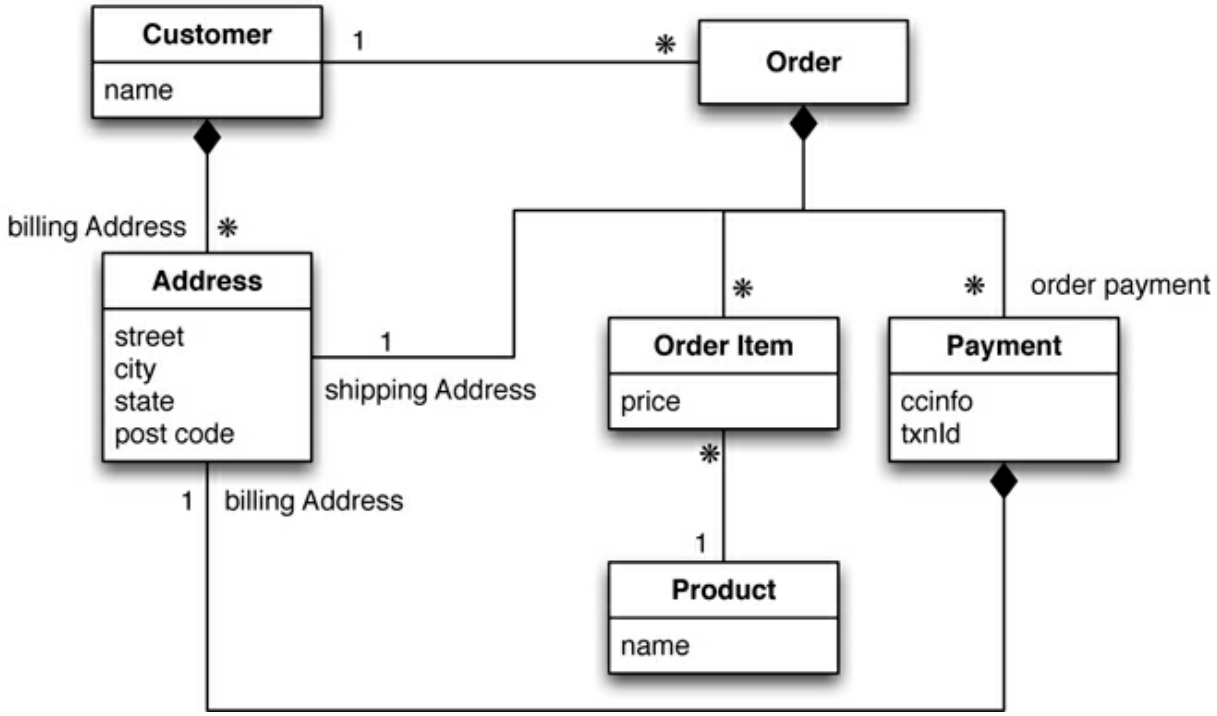
1.5. Agregāts

Relāciju datu bāzes atgriež informāciju kā ierakstu, jeb rindiņu, kopu. Ieraksts ir ierobežota datu struktūra – tas neatbalsta ligzdošanu un neļauj glabāt sarakstu, ka vienu no ieraksta laukiem.

Viens no būtiskiem ar *NoSQL* saistītiem terminiem, kas nāk no *DDD* (*no angļ.: Domain-Driven Development*), jeb domēnu virzītas izstrādes, ir agregāts. Ar agregātu ir jāsaprot dažādo saistīto objektu kolekciju, kas tiek izmantota kā viena vienība. Agregātus izmanto trīs no četrām *NoSQL* datu bāžu tipiem:

- Dokumentu tipa datu bāzes;
- Atslēgas-vērtības datu bāzes;
- Kolonnu-ģimenes datu bāzes.

Agregāti tiek ļoti veiksmīgi izmantoti *NoSQL* pasaulē vairāku pēc vairākiem iemesliem. Pirmkārt, agregāts ir naturāla vienība priekš klasterizācijas. Otrkārt, agregāts risina kartēšanas problēmu, kas būtiski atvieglo izstrādes procesu. Apskatīsim piemēru, lai saprastu, kā agregāti atšķīrās no rindu kolekcijas relāciju datu bāzēs.



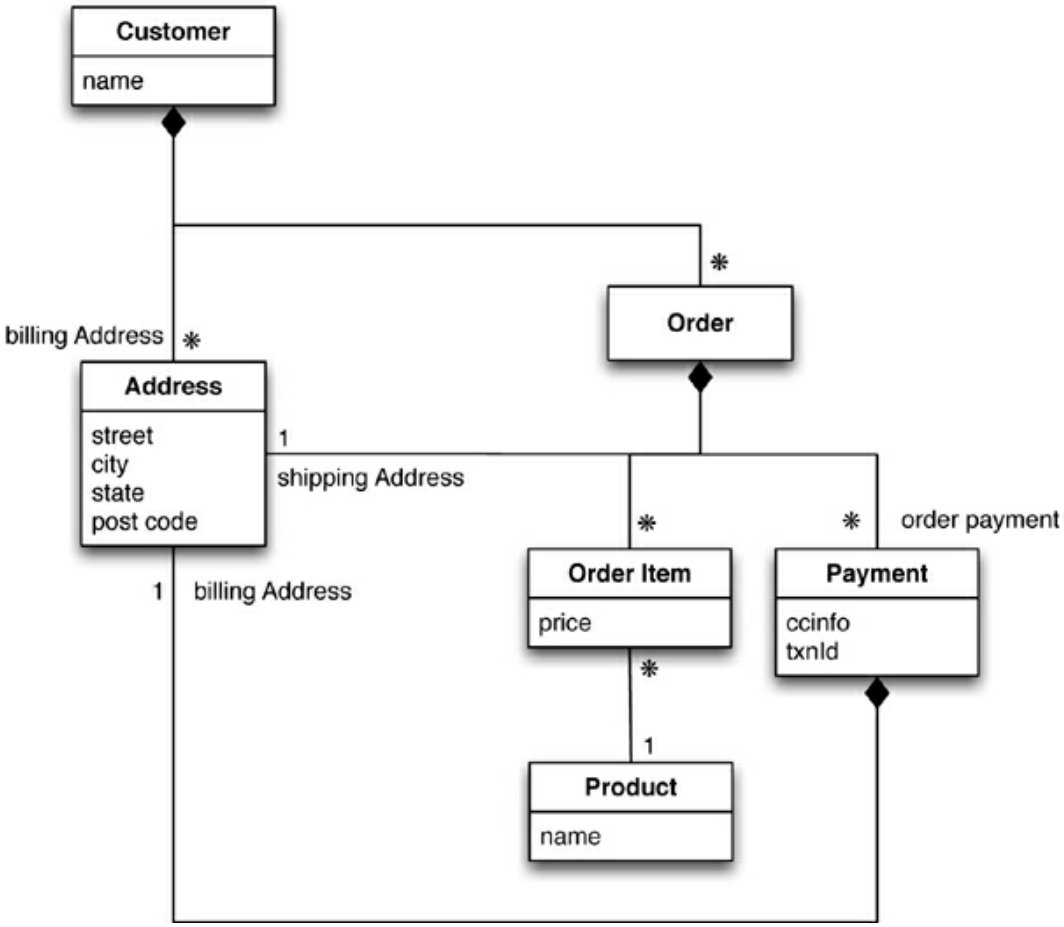
1.5. att. **Relāciju datu modelis [1]**

Attēlā 1.5. ir parādīts datu modelis, kur viss ir normalizēts, lai dati neatkārtojas vairāk kā vienā tabulā. Reālistiskā pasūtījumu sistēmā būtu vairāk tabulu, bet priekš mūsu piemēra šīs datu modelis ir pilnīgi pietiekams. Tagad apskatīsim, kā šo datu modeli var atspoguļot ar agregātu palīdzību [1].

```
// pasūtītāju agregāts
{ "id":1,
"name":"Martin",
"billingAddress":[{"city":"Chicago"}]
}
// pasūtījumu agregāts
{
"id":99,
"customerId":1,
"orderItems":[
  {
  "productId":27,
  "price": 32.45,
  "productName": "NoSQL Databases explained"
```

```
    }
  ],
"shippingAddress":[{"city":"Chicago"}]
    "orderPayment":[
  {
    "ccinfo":"1000-1000-1000-1000",
    "txnId":"abelif879rft",
    "billingAddress": {"city": "Chicago"}
  }
  ],}
```

JSON fragmentā, kas ir standarts datu glabāšanas formāts *NoSQL* pasaulē, mēs varam redzēt, ka datu modelis ir sadalīts divos agregātos – pasūtītājs un pasūtījums. Var redzēt, ka piemērā nav definēts agregāts produkts, bet produkta nosaukums ir iekļauts pasūtījumu agregāta pasūtījumu elementu masīvā. Šī denormalizācija ir ļoti raksturīga agregātu datu modeļos, jo mēs cenšamies minimizēt agregātu skaitu, kas būs nepieciešami vienam pieprasījumam.



1.6. att. **Alternatīvs agregātu datu modelis [1]**

Specifiska datu denormalizācija ir katra *NoSQL* datu bāzes administratora apzināts lēmums. Kā parādīts attēlā 1.6., šo pašu agregātu var nomodelēt arī citādi. Izmantojot attēlā 1.6. aprakstīto datu modeli, agregāti pircējs un pasūtītājs izskatīsies citādi [1]:

```
// in customers
{ "customer": {
"id": 1,
"name": "Martin",
"billingAddress": [{"city": "Chicago"}],
"orders": [
  { "id":99,
    "customerId":1,
    "orderItems":[
    {
    "productId":27,
    "price": 32.45,
    "productName": "NoSQL Distilled"
    }
  ],
  "shippingAddress":[{"city":"Chicago"}]
  "orderPayment":[
    {
    "ccinfo":"1000-1000-1000-1000",
    "txnId":"abelif879rft",
    "billingAddress": {"city": "Chicago"}
    }],
  }]
}}
```

Ir būtiski saprast, ka modelēšanā nekad nebūs vienas pareizas atbildes. Modelējot agregātus, ir jāanalizē konkrēti biznesa pieprasījumi. Ja mums ir nepieciešams atlasīt pasūtītāju kopā ar visiem pasūtījumiem, mums pietiktu ar vienu agregātu, kur katrs pasūtījums atrastos pasūtītāju agregāta masīvā „pasūtījums". Turpretī, ja ir svarīgi atlasīt katru konkrētu pasūtījumu, vai apstrādāt statistiskus datus, saistītus ar pasūtījumiem, mums būtu nepieciešams atlasīt visus pasūtītājus, un izņemt no tiem visus pasūtījumus, kas nebūtu efektīvi. Tādā gadījumā ir loģiski pielietot atsevišķu agregātu prieks pasūtījumiem.

1.6. MapReduce

MapReduce ir koncepts, kuru popularizēja *Google Inc.* – Amerikāņu multinacionālā korporācija, kas specializējas uz internet-bāzētiem pakalpojumiem. Paradigma pirmo reizi tika aprakstīta zinātniskā publikācija[12], kuru 2004. gadā decembrī prezentēja Džefrijs Dīns un Sandžejs Ghemavats. Darbs tika prezentēts sestajā simpozija par operētājsistēmu dizainu un implementāciju.

MapReduce ir algoritmisks ietvars, kas atļauj izpildīt sarežģītus uzdevumus paralēli klastera ietvaros. Ietvars kļuva ļoti populārs *NoSQL* pasaulē, jo palīdzēja efektīvi cīnīties ar Lielo Datu problēmu, izmantojot klasterizāciju.

Ietvara ideja ir sadalīt problēmu divās daļās. Pirmā daļa konvertē datu sarakstu citā tipa sarakstā, izmantojot map() funkciju. Otrā daļa konvertē otro sarakstu vienā vai vairākās skalārās vērtībās, izmantojot reducē() funkciju. Izmantojot šo paradigmu, programma atļauj sistēmai sadalīt uzdevumus mazos komponentos un izpildīt tos lielajā klasterī paralēli.

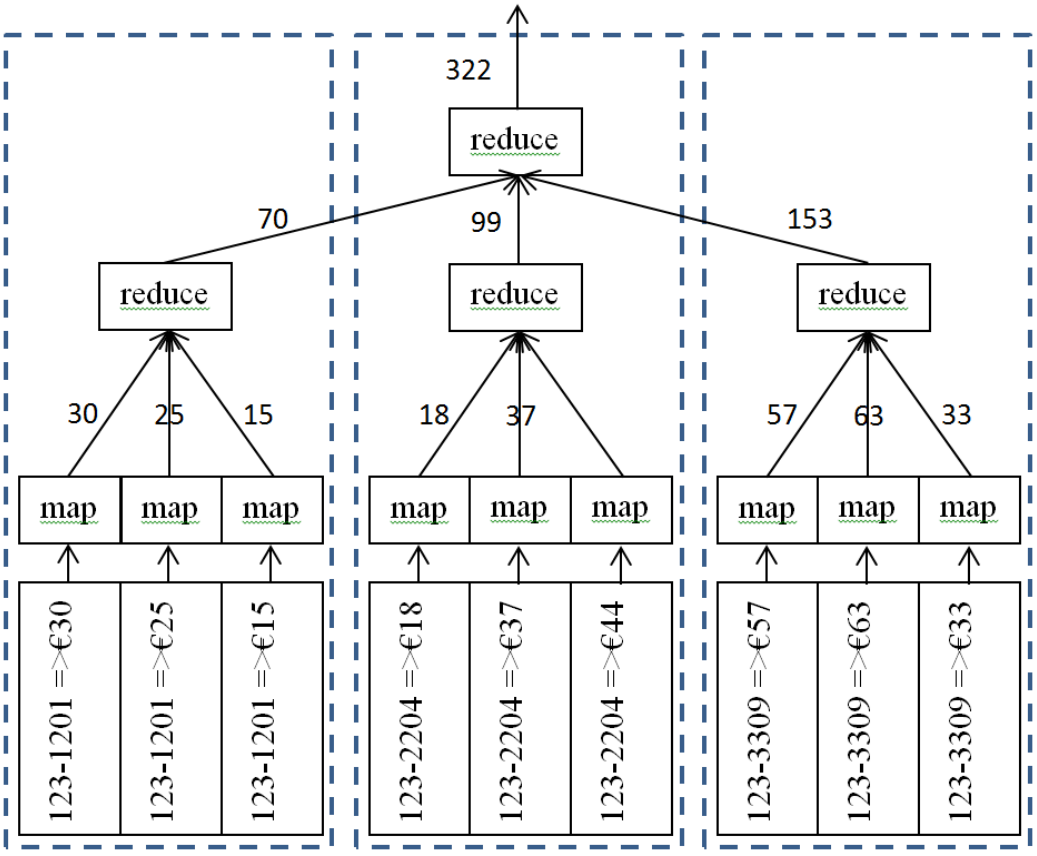
Apskatīsim, kā datu bāzes atlasīšanas pieprasījums tiek izpildīts *SQL* pasaulē.

SELECT * FROM rooms LIMIT <batch_size> OFFSET <last_batch>

Datu bāzes serveris sūta visus rezultātus uz lietotnes serveri paketēs, un tad lietotņu serveris izpilda kaut kādas darbības ar šiem rezultātiem. Tas ir pieejams relatīvi mazos datu apjomos. Bet kad ierakstu skaits kļūs lielāks, sistēma kļūs lēnāka, ja datu bāzei ir nepieciešams lielāks laiks, lai atsūtītu visas rezultātu paketes.

MapReduce strādā otrādi. Lietotņu serveris, jeb klients, sūta tikai algoritmu uz katru no datu bāzes mezglu, un katrs mezgls ir atbildīgs par rezultāta atgriešanu. Tas nozīmē, ka *map()* un *reduce()* funkcijas tiek izpildītas datu bāzes serveros, nevis lietotņu servera mašīnā.

Ir svarīgi atzīmēt, ka izmantojot *MapReduce* ietvaru, izstrādātājs, kuram nav zināšanu un pieredzes par paralelizēto uzdevumu apstrādi, var izmantot datu bāzes klastera priekšrocības. Viss, kas ir nepieciešams ir uzrakstīt *map()* un *reduce()* funkcijas datu bāzes pārvaldības sistēmai pazīstamā sintaksē. Par paralelizāciju un mezglu pieejamību atbild datu bāzes meistara mašīna. Attēlā 1.7. parādīts *MapReduce* pielietojuma piemērs datu bāzes klasterī ar trim mezgliem.



1.7. att. **MapReduce izpildes procesa diagramma trīs mezglu klasterī**

Viens no ļoti efektīviem paņēmieniem *MapReduce* ietvarā ir *reduce()* un *map()* funkciju virknēšana. Normālā gadījumā *map()* funkcijas rezultāts tiek izmantots ka ievada parametrs priekš *reduce()* funkcijas. Bet lai rakstītu patiešām efektīvus un elegantus pieprasījumus, bieži *reduce()* funkciju rezultāti tiek izmantoti ka jauno *map()* funkciju ievada parametri.

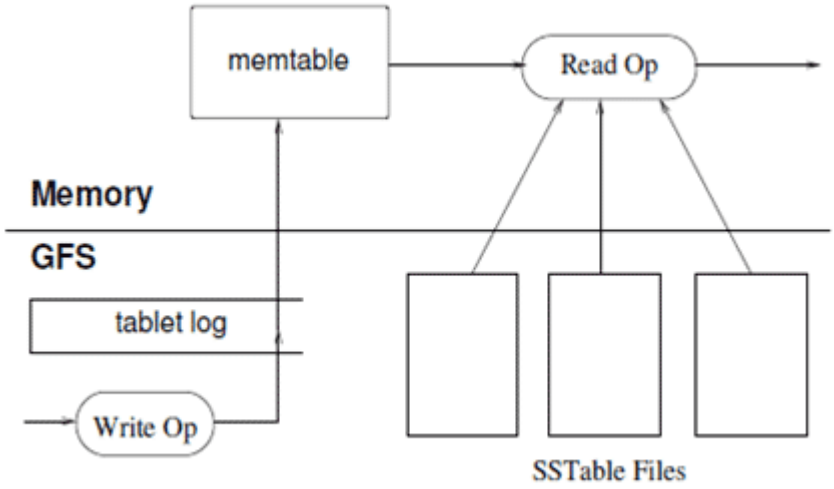
2. DATU BĀZES IZVĒLE

Šī sadaļā tiek apskatītas dažādu *NoSQL* datu bāžu arhitektūras un struktūras koncepti. Tiek definētas līdzības un atšķirības ne-relāciju datu bāžu iekšējā organizācijā. Rezultātā tiks izvēlēta vispiemērotākā ne-relāciju datu bāze integrācijai ar *ADF* ietvaru.

2.1. BigTable arhitektūra

Gan *Cassandra,* gan *HBase* tika ietekmēti ar „*Google BigTable*." Lielu ietekmi uz *Cassandra* datu bāzes attīstību atstāja arī „*Amazon Dynamo*", bet šī nodaļa tiks aprakstītas *BigTable* arhitektūras īpašības, kurus izmanto gan *Cassandra,* gan *HBase* datu bāzes.

„*BigTable*" ir kolonnu-ģimenes tipa datu bāze kas glabā datus daudzlīmeņu kārtotā asociatīvā masīvu formātā (*Multidimensional Sorted Map*). Datu bāzes struktūra ir <ieraksta atslēga, kolonnu ģimene, kolonnas atslēga>. Kolonnu ģimene ir pamata vienība, kas glabā loģiski saistītas kolonnu grupas. Attēlā 2.1. ir parādīta *BigTable* arhitektūras grafisks attēlojums.



2.1. att. **„BigTable" arhitektūra**

„*BigTable*" datu bāzē jaunie dati tiek pievienoti jau eksistējošiem datiem. Tas nozīme, ka kad dati tiek modificēti, jaunie dati tiek pievienoti failam, nevis aizvieto jau eksistējošus datus.

Kad datu bāzes serveris saņem pievienošanas pieprasījumu, dati tiek ievietoti atmiņas apgabalā, ko sauc par '*memtable'*. Ja šīs apgabals ir pilns, tad visi dati tiek glabāti failā, ko sauc par kārtoto virkņu tabulu (*SSTable* jeb *Sorted String Table*). Ja servera mašīna sabrūk pirms

dati tiek pārvietoti no atmiņas tabulas uz kārtoto virkņu tabulu, dati tiek pazaudēti. Līdz ar to, lai nodrošinātu augstu izturīgumu, ir nepieciešams saglabāt komandu vēsturi pirms dati tiks ierakstīti atmiņas tabulā.

Kad serveris saņem atlasīšanas pieprasījumu, vispirms atslēga tiek meklēta atmiņas tabulā. Ja dati ar konkrētu atslēgu nav atrasti iekš atmiņas tabulas, meklēšana notiek kārtoto virkņu tabulā. Atslēga var tikt meklēta vairākās kārtoto virkņu tabulās.

Šī arhitektūrai ir priekšrocība, izpildot pievienošanas pieprasījumus. Vispirms dati tiek ierakstīti atmiņā, un ir pārvietoti uz diska tikai tad, kad ir sakrājies noteikts datu apjoms. Tas ļauj izvairīties no laiksakritīga ievada/izvada problēmas.

Savukārt, veiktspēja būs zemāka, izpildot lasīšanas pieprasījumus, ja lasīšana notiek no kārtoto virkņu tabulas, nevis no atmiņas tabulas. Tiek izmantots filtrs, kas ātri noteic, vai atslēga eksistē kārtoto virkņu tabulā, un uztaisa indeksu tālākai lietošanai. Tomēr, ja eksistē vairākas kārtoto virkņu tabulas, lasīšanas pieprasījuma izpildes laikā notiek daudz ievada/izvada operāciju. Lai uzlabotu lasīšanas veiktspēju, tiek izmantots datu sablīvēšanas paņēmiens (*no angļ.: compaction*) – dati no vairākām kārtoto virkņu tabulām tiek savienoti un kārtoti, ar mērķi samazināt uz diska glabāto tabulu skaitu. Ir svarīgi atzīmēt, ka sablīvēšanas rezultātā tiek uzlabota gan lasīšanas, gan rakstīšanas veiktspēja.

Iepriekšminēto iemeslu dēļ lasīšanas un jauninājumu pieprasījumi datu bāzēs ar šo arhitektūru tiek izpildīti daudz lēnāk, nekā rakstīšanas pieprasījumi.

2.2. Cassandra

Cassandra datu bāze izmanto konsekventu jaukšanu (algoritmiska metode, kas ļauj pārveidot ieraksta atslēgu tā adresē; ieraksta atslēga nosaka datu atrašanās vietu). Galvenā atšķirība starp *Cassandra* un *HBase* ir *CAP* teorēmas akcents – *Cassandra* pievērš uzmanību datu pieejamībai un nodalījumu tolerancei (AP – *Availability and Partition tolerance*), bet *HBase* koncentrējas uz datu saskaņas un nodalījumu tolerances (CP – *Consistency and Partition tolerance*). *CAP* teorēmā ir sīkāk aprakstīta nodaļā 1.1.

Konsekventa jaukšana ir aktīvi izmantota „*Amazon Dynamo*" *NoSQL* datu bāzē. *Dynamo* izmanto konsekventu jaukšanu lai sadalītu datus starp vairākām kalpu mašīnām, kas formē datu bāzes klasteri. Tas atļauj meistara mašīnai efektīvi izlemt, kurā no kalpu mašīnām ierakstīt jaunu dokumentu, vai kurā mašīnām meklēt eksistējošo dokumentu.

Cassandra paaugstina lietojamības līmeni pateicoties datu saskaņai. Šī koncepcija ir saistīta ar replikāciju. Izmantojot sistēmas parametru, ir iespējams definēt nepieciešamo replikācijas mezglu apstiprinājumu skaitu. Piemēram, ja replicēšanas kopa sastāv no trim mezgliem, datu bāzi var sakonfigurēt tādā veidā, lai pievienošanas pieprasījums tiktu uzskatīts par veiksmīgu tikai tad, kad apstiprinājums tika saņemts no visiem replicēšanas kopas mezgliem.

Tomēr, *Cassandra* atļauj pārbaudīt tikai trīs operācijas, pirms vērtība tiks atgriezta. Tas nozīmē, ka rakstīšanas pieprasījumi var būt uzskatīti par veiksmīgiem pat tad, ja kļūdas notika replicēšanas fāzē. Tas būtiski uzlabo lietojamību. Neveiksmīgas rakstīšanas operācijas tiek reģistrētas atsevišķā replikācijas kopas mezglā, un var tikt atkārtoti izpildītas vēlāk. Šī funkcionalitāte *Cassandra* datu bāzē ir saukta par „*hinted handoff*".

Ņemot vērā, ka replicēšanas fāzes veiksmīga izpilde netiek garantēta, datu piemērotība ir pārbaudīta lasīšanas fāzē. Pēc noklusējuma, ja *Cassandra* datu bāzes serveris izmanto replikāciju, lasīšanas laika dati tiek nolasīti no katra replikācijas mezgla. Datu bāzes meistara mašīna ņem vērā, ka starp replikācijas mezgliem var atšķirties. Tāpēc pēc lasīšanas pieprasījuma izpildes, jaunākā datu versija tiek replicēta uz visiem replicēšanas kopas mezgliem. Līdz ar to *Cassandra* izpilda lasīšanas pieprasījumus lēnāk, nekā rakstīšanas pieprasījumus.

2.3. HBase

HBase struktūra ir līdzīga „*Google BigTable*" datu bāzes struktūrai. *BigTable* strādā ar „*tablet unit*" vienībām, bet *HBase* izmanto vienību „*region unit*". Atslēgas ir sadalītas pa diapazoniem, un katra diapazona atrāšanas vieta tiek glabāta meta datu formā atsevišķā meta reģionā. Kad atslēga ir ierakstīta, vispirms ir jāatrod vajadzīgs diapazons, un tad klients pielieto jaukšanu jauniem datiem. *HBase* ievieto datus viena konkrētā diapazonā un sadala diapazonu tad, ja tas kļūs pārāk liels.

HDFS, jeb *Hadoop* veltīta failu sistēma, nodrošina datu glabāšanu un replicēšanu. Rakstot failu *HBase* datu bāzē, *HDFS* sinhroni replicē datus uz visiem replicēšanas kopas mezgliem, bet lasīšanas laika nolasa datus tikai no viena mezgla, kas nodrošina datu integritāti. Atmiņas tabula un kārtoto virkņu tabula tiek glabātas mezglā, kura atrāšanas vieta var mainīties sakarā ar to, ka *Hbase* un *HDFS* strādā atsevišķi. Tas var izraisīt paaugstinātu ievada/izvada procedūru skaitu.

Ja viens no klastera mezgliem, jeb reģions, sabrūk, pārējie mezgli pārņem šī mezgla datu pārvaldi. Operāciju vēsture ir atkārtoti palaista lai uztaisītu sabrukuša mezgla atmiņas tabulu. Ja mezgls, kurš sabruka, saturēja *HDFS* datus, tad attiecīgs datu fragments tiek migrēts uz citu mezglu.

Cassandra un HBase arhitektūras atšķirības

Iepriekšējas sadaļas parāda *Cassandra* un *HBase* vairākas arhitektūru un struktūru līdzīgas īpašības. Sekojošas īpašības ir svarīgākas starpības divu datu bāžu darbībā.

Pirmkārt, *Cassandra* datu bāze saglabā komandu vēsturi failu sistēmā, un dara to tikai vienreiz. *HBase* nosūta komandu vēsturi uz *HDFS*, pēc kā *HDFS* izpilda replicēšanu. Datu fiziska atrāšanas vieta netiek ievērota, līdz ar to datu rakstīšanas pieprasījums var tikt nosūtīts uz citu fizisku klastera mašīnu.

Otrkārt, *Cassandra* datu bāze saņem rakstīšanas pieprasījumu uz visiem replicēšanas kopas mezgliem, bet *HBase* datu bāze izpilda rakstīšanas pieprasījumu tikai vienā reģionā, un apstrādā lasīšanas pieprasījumus tikai vienā mezglā.

Treškārt, lai apstrādātu vienu lasīšanas pieprasījumu, *Cassandra* nolasa datus trīs reizes – no katra replicēšanas kopas mezgla. *HBase* nolasa datus tikai vienreiz.

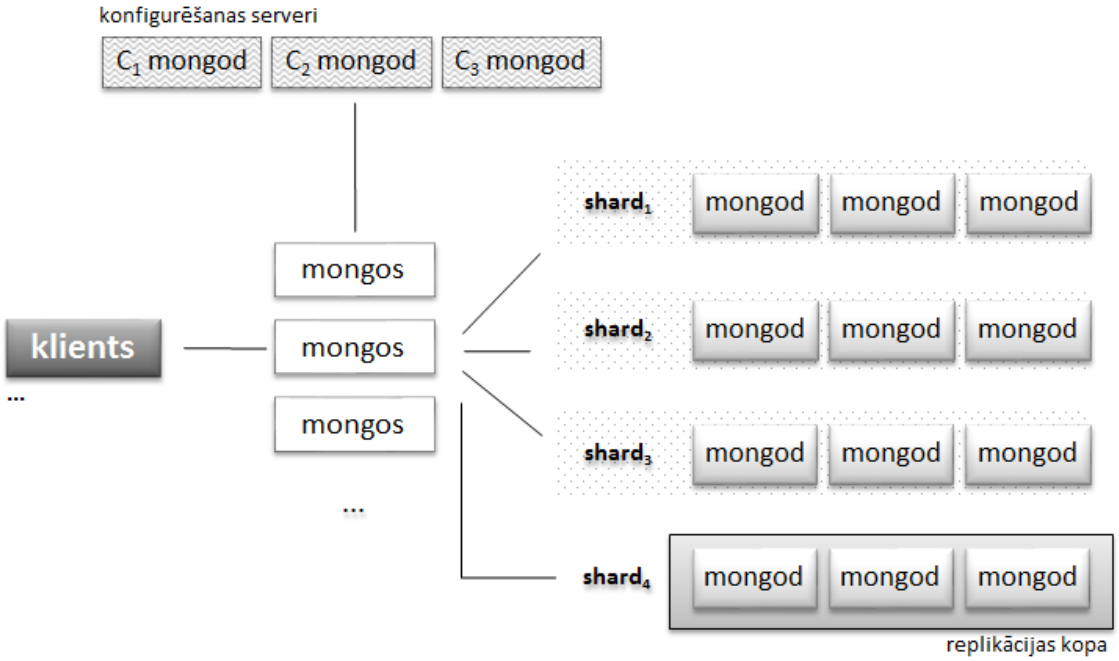
2.4. MongoDB

Mongo datu bāze piedāvā konfigurējamu replicēšanu un datu izplatīšanas konfigurēšanu. *MongoDB* izstrādātāji rekomendē izmantot trīs reģionus (*MongoDB* termins ir „*shard*") ar replicēšanas faktoru 3. Replicēšana ir detalizētāk aprakstīta sadaļa 1.3.

Mongo datu bāzēs izmanto Mestars-Kalps tipa replicēšanu, un atbalsta automātisko atveseļošanu (*no angļ: automatic failover, automatic recovery*). Replicēšanas kopā ir viena meistara mašīna un vairākas kalpu mašīnas. Rakstīšanas pieprasījumus apstrādā tikai meistara mašīna. Ja meistara mašīnas sabrūk, viena no kalpu mašīnām tiek izvēlēta par jauno meistaru.

Datu izplatīšanas sistēma, ko piedāvā *Mongo* datu bāze, ir „*sharding*". Katra kolekcija var tikt sadalīta uz apgabaliem, un viens no kolekcijas laukiem kļūs par apgabalu atslēgu. Līdzīgi *HBase* datu bāzei, „*sharding*" izmanto secību saglabājošo nodalījumu metodi *(order-preserving partitioning*). Ja kolekcija izmanto „*sharding*", tā ir sadalīta diapazonos (*Mongo* termins ir

„*chunk*"). Diapazonu definē trīs vērtības - <kolekcija, minimālā vērtība, maksimālā vērtība>. Diapazona maksimāls izmērs ir 64 megabaiti. Ja diapazons izmērs pārsniedz maksimālo, tas tiek sadalīts divos jaunos diapazonos. Viens, vai abi no jauniem diapazoniem var būt nomigrēti uz citu fizisku klastera mašīnu. Attēls 2.2. demonstrē *MongoDB* klastera organizāciju.



2.2. att. **MongoDB klastera organizācija**

Kad klients savienojas ar *Mongo* datu bāzes klasteri, vispirms tas sarunājās ar 'mongos' procesu, kas atbild par maršrutēšanu un koordinēšanu. Koordinēšana notiek starp vairākiem klastera mezgliem, rezultāti ir apkopoti un atgriezti klientam. Konfigurācijas serveri glabā meta informāciju par šardiem un diapazoniem.

Mongo datu bāze izmanto asinhrono replicēšanas metodi. Kad tiek saņemts rakstīšanas pieprasījums, meistara mašīna saglabā pieprasījumu operāciju vēsturē (Mongo termins – *Oplog*). Operāciju žurnāls ir fiksēta izmēra *FIFO* kolekcija. Operāciju žurnāls strādā cirkulārās rindas veidā – kad nepietiek vietas jauniem datiem, visvecākie dati tiek nodzēsti. Kalpu mašīnas regulāri nolasa un izpilda žurnāla glabātas komandas. Tas salabo lietojamību, jo meistara koordinēšanas procesam nav jāgaida apstiprinājums no replicēšanas kopas kalpu mašīnām. Ja kalpa mezgls nepaspēj laicīgi izpildīt operāciju žurnāla komandas, tas pārtrauc operāciju žurnāla komandu replicēšanu un sinhronizējas, nolasot datus no meistara mašīnas.

Vel viena svarīga *Mongo* datu bāzes operāciju žurnāla īpašība ir „*idempotence"* – žurnāla komandas var tikt izpildītas vairākas reizes ar vienādiem datiem, un rezultāts katru reizi būs vienāds. Tas atļauj izpildīt operāciju žurnāla glabātas komandas vēlāk, piemēram, kad mašīna ieslēdzas pēc pārstartēšanas.

2.5. Datu bāzes izvēles pamatojums

Mongo datu bāzes galvenā priekšrocība ir spēja apstrādāt milzīgus datu apjomus, un, kas ir vēl svarīgāk, milzīgo pieprasījumu skaitu, kas ir paveikts pateicoties replicēšanas pieejai un horizontālai mērogošanai.

ADF tradicionāli strādā ar *OracleDB*, kas ir relāciju datu bāze. *MongoDB* sintakse un komandu struktūra ir ļoti līdzīga *SQL* valodas komandām, ar izņēmumu, ka *Mongo* neatbalsta „*join*" komandas. Turklāt, dokumentu tipa datu bāzes ir daudz līdzīgākas relāciju datu bāzēm, nekā atslēgu-vērtību un kolonnu-ģimenes tipa datu bāzes.

Vel viena svarīga *Mongo* datu bāzes īpašība ir shēmas prombūtne. Tas ir ļoti izdevīgi, implementējot Modelis-Skats-Kontrolieris dizaina šablonu, izstrādājot tīmekļa lietotni. Pievienot jaunu lauku *Mongo* datu bāzē ir tik pat viegli ka pievienot jaunu lauku datu modeles klasē. Tas ļauj secināt, ka visuzskatamāk ir integrēt *ADF* tīmekļa lietotņu izstrādes ietvaru tieši ar *Mongo* datu bāzi.

2.6. Izvēlētas datu bāzes detalizēts apskats

Mongo datu bāzes nosaukums nāk no angļu vārda *Humongous*, kas nozīmē „ārkārtīgi liels" [4]. *MongoDB* galvenie koncepti ir veiktspēja un datiem viegla pieejamība. *MongoDB*, jeb *Mongo*, ir dokumentu tipa datubāze, kas atļauj datiem pastāvēt mantotā veidā.

MongoDB izstrādātājs ir privāta programmatūras izstrādes kompānija '10gen', kuru 2007. gadā dibināja '*DoubleClick*' kompānijas izpilddirektors Dwight Merriman un privātas e-komercijas kompānijas '*Gilt Groupe*' dibinātais Kevin P. Ryan. MongoDB pirmā publiska relīze notika 2009. gadā. Komerciālo tehnisko atbalstu 10gen saka piedāvāt 2010.gada martā [10].

MongoDB ir *JSON* (*no angļu: JavaScript Object Notation* [11]) dokumentu tipa datu bāze, kaut gan dati fiziski tiek glabāti *JSON* binārā formātā – *BSON*, jeb binārs *JSON*. Attēlā 2.3. ir parādīts MongoDB atlasīšanas pieprasījums un atgriezts rezultāts.

2.3. att. **MongoDB JSON dokumenta struktūra**

Mongo pieprasījumam obligāti jāsastāv no sekojošiem elementiem: norāde uz datu bāzi, kolekcijas nosaukums, komandas nosaukums un pieprasījuma parametri.

2.6.1. MongoDB pielietojumi lielajos uzņēmumos

Ir svarīgi apskatīt, kā *MongoDB* tiek lietotā lielajos uzņēmumos visā pasaulē. Vispazīstamākās kompānijas, kas aktīvi lieto *MongoDB* ir:

- *Foursquare* – sociāls tikls bāzēts uz atrāšanas vietām,
- *bit.ly* – URL saīsināšanas serviss, un
- *CERN* – Eiropas organizācijas, kas nodarbojas ar kodolpētniecību.

2012. gadā decembrī MongoDB izstrādātājs 10gen norīkoja divus kursu programmas – *MongoDB* priekš izstrādātājiem un *MongoDB* priekš datu bāzes administratoriem. Izstrādātāju kursa beigās 10gen apmācības nodaļas vadītājs un kompānijas viceprezidents Dwight Merriman intervēja Džonu Hoffmanu, kas pašlaik ir Foursquare kompānijas inženērijas infrastruktūras vadītājs. Autors uzskata, ka ir vērts apskatīt veidu, kurā *Foursquare* datu bāzes administratori un arhitekti izmanto *MongoDB*. *Foursquare* prototips, kas tika izstrādāts 2009. gadā, izmantoja *MySQL* datu bāzi. 2010. gadā bija pieņemts lēmums migrēt kodu uz *Scala* programmēšanas valodu un *Postgres* datu bāzes pārvaldības sistēmu. Kad lietotnes lietotāju skaits saka strauji augt,

parādījās datu bāzes serveru klasterizācijas nepieciešamība. Tas nozīmēja, ka vajadzēja daļēji vai pilnībā atteikties no transakciju pārvaldības un no *SQL JOIN* funkcionalitātes. *Foursquare* datu bāzes arhitekti izvēlējas *MongoDB*.

Migrācijas process no *Postgres* uz *MongoDB* notika pakāpeniski. Pirmkārt, uz jauno NoSQL datu bāzi tikai nomigrēta tikai viena kolekcija – tikšanas vietu kolekcija (*no angļ.: venue*). Dažas nedēļas laikā rakstīšanas pieprasījumi gāja gan uz *Postgres*, gan uz MongoDB datu bāzēm. Arhitekti gribēja pārliecināties, ka jaunā datu bāze strādā pietiekami stabili un ātri. Kad sagaidāmais rezultāts bija sasniegts, pārejas kolekcijas tika migrētas uz *MongoDB*. Pēc tam lasīšanas pieprasījumi tika pārslēgti uz *MonogDB*, un *Postgres* datu bāze strādāja ka replicēšanas datu bāze, kas saņēma tikai rakstīšanas pieprasījumus. Pēc dažām nedēļām, kad *Foursquare* datu bāzes administratori bija pārliecināti, ka *MongoDB* ir stabils risinājums, *Postgres* datu bāze tika pilnīgi izslēgta.

Autoraprāt, interesanti tikai izmantota *MongoDB* shēmas prombūtnes īpašība. Pēc Džona Hoffmana vārdiem, migrējot datus uz *MongoDB*, arhitekti nebija parēķinājuši, cik daudz datus 1aizņems, piemēram, 'checkin' kolekcija. Uz 2012. gada decembri, pēc Džona Hoffmana vārdiem, *Foursquare* saņēma apmēram piecus miljonus jaunus *checkin* dokumentus katrā diennaktī, un kopējais *checkin* dokumentu skaits iekš kolekcijas bija pārsniedzis 2.5 miljardus. Kad kolekcija satur 2.5 miljardus ierakstu, gari lauku nosaukumi sāk aizņemt pietiekami daudz vietas, lai kļūst par optimizācijas iemeslu.

Sava intervijā Jons Hoffmans pateica, ka lielākajās kolekcijās lauku nosaukumi tagad sastāv no viena burta, lai maksimāli notaupīt vietu. Veids, kā *Foursquare MongoDB* administratori izlēma saīsināt lauku nosaukumu garumu ir diezgan inovatīvs, un, autorāprāt, interesants. Lai nedeaktivēt datu bāzi, izstrādātāji pieņēma šādus lēmumus:

- Visi jauni rakstīšanas pieprasījumi izmantos jaunus lauku nosaukumus,
- Kods izmantos veco lauku nosaukumu tikai tādā gadījumā, ja dokuments nesatur jauno laukumu nosaukumu.

Pēc tam tika palaists „*batch"* process, kas nodzēsa visus vecos laukus no datu bāzes, un uztaisīja jaunus laukus tos ierakstos, kuros eksistēja tikai vecie lauki. Šīs pakāpenisks process aizņēma vairākas nedēļas, un nepieprasīja deaktivēt datu bāzi, lai uztaisīt izmaiņas. Tas bija iespējams tikai pateicoties *MongoDB* shēmas prombūtnes īpašībai.

2.6.2. Mongo un SQL komandu salīdzinājums

Lai paradīt, kā *MongoDB* komandas atbilst *SQL* komandām, apskatīsim *CRUD* (*Create Read Update Delete*, jeb rakstīšanas, lasīšanas, atjaunināšanas un dzēšanas) operāciju komandas *MongoDB* sintaksē un *SQL* sintaksē. Tabula 2.1. salīdzina ierakstu pievienošanas komandas.

2.1. tabula

Tabulas uztaisīšanas un modifikāciju komandu salīdzinājums

SQL komanda	**MongoDB komanda**
CREATE TABLE users (id INT NOT NULL AUTO_INCREMENT, user_id Varchar(40) age Number, status char(1), PRIMARY KEY (id))	Kolekcija tiek netieši uztaisīta, izpildot pirmo ierakstu pievienošanas pieprasījumu. Primārā atslēga _id ir pievienota automātiski, ja tā nav tieši norādīta. db.users.insert({ user_id: “abc123”, age: 24, status: "A" })
ALTER TABLE users ADD join_date DATETIME	Kolekcijas neapraksta un neuzspiež definēto dokumentu struktūru.
ALTER TABLE users DROP COLUMN join_date	Kolekcijas neapraksta un neuzspiež definēto dokumentu struktūru.
CREATE INDEX idx_user_id_asc ON users(user_id)	db.users.ensureIndex({ user_id: 1})
CREATE INDEX idx_user_id_asc_age_desc ON users(user_id, age DESC)	db.users.ensureIndex({ user_id: 1, age: -1})
DROP TABLE users	db.users.drop()

Var redzēt, ka nav nepieciešamas speciālas komandas, lai pievienotu jaunas kolonnas, jeb laukus kādai no *Mongo* datu bāzes kolekcijām. Katrs ierakstu pievienošanas pieprasījums var

saturēt laukus, kuri vel neeksistē nevienā eksistējošā kolekcijas dokumentā. Savukārt, nav obligāti, lai jauns ieraksts satur laukus no kolekcijas dokumentos eksistējošiem laukiem. Piemērām, ir iespējams izpildīt pieprasījumu kas pievieno lietotāju, kuram ir norādīts vecums un status, un pēc tam pievienot lietotāju, kuram nav definēts status, bet ir definēts vecums un reģistrācijas datums. Tabulā 2.2. tiek salīdzinātas ierakstu pievienošanas komandas.

2.2. tabula

Ierakstu pievienošanas pieprasījumu salīdzinājums

SQL komanda	**MongoDB komanda**
INSERT INTO users(user_id, age, status) VALUES ("uid312", 24, "A")	db.users.insert({ user_id: "uid312", age: 24, status: "A" })

Jauno ierakstu pievienošanas komanda ir ļoti līdzīga *SQL* komandai, ar izņēmumu, ka lauku nosaukums un vērtība tiek norādīti blakus viens otram, kas padara pievienošanas pieprasījumu daudz lasāmāku.Tabula 2.3. tiek salīdzinātas ierakstu atlasīšanas komandas.

2.3. tabula

Atlasīšanas pieprasījumu komandu salīdzinājums

SQL komanda	**MongoDB komanda**
SELECT * FROM users	db.users.find()
SELECT id, user_id, status FROM users	db.users.find({ }, { user_id: 1, status: 1 })
SELECT * FROM users WHERE status = "A"	db.users.find({ status: "A"})

2.3. tab. turpinājums

SELECT * FROM users WHERE status = "A" AND age = 24	db.users.find({ status: "A", age: 24})
SELECT * FROM users WHERE status = "A" OR age = 24	db.users.find({ $or: [{ status: "A" }, { age: 24 }] })
SELECT * FROM users WHERE age > 24	db.users.find({ age: { $gt: 24 } })
SELECT * FROM users WHERE status = "A" ORDER BY user_id DESC	db.users.find({ status: "A" }).sort({ user_id: 1 })
SELECT COUNT(*) FROM users	db.users.count()
SELECT COUNT(user_id) FROM users	db.users.count({ user_id: { $exists: true } }) or db.users.find({ user_id: { $exists: true } }).count()

Lai vieglāk saprastu, kā strādā *Mongo* atlases komanda find, apskatīsim tas sintaksi:

db.collection.find(<pieprasījums>, <projekcija>)

Pirmais komandas parametrs ir WHERE bloka analogs. Otrais parametrs, kur sauc par projekciju, definē laukus, kuras tiks atlasītas. *SQL* sintaksē kolonnas, kas tiks atlasītas, ir norādītas aiz SELECT atslēgvārda. *Mongo* atlasītie lauki tiek definēti *find* komandas otrajā parametrā, jeb projekcijā. Lai lauks tiktu atlasīts, tā vērtībai projekcijā ir jābūt 1.

Projekcija ir neobligāts parametrs, kuru drīkst nenorādīt. Pēc noklusējuma, ja projekcijas parametrs nav norādīts, tiks atlasīti visi dokumenta lauki. Ja projekcijas lauks nav tukšs, tiks atlasīti

tikai tie lauki, kuru vērtības ir vienādas ar 1. _id lauks ir atlasīts vienmēr, pat ja tas nav norādīts projekcijas parametrā. Lai neatlasītu _id lauku, tā vērtībai projekcijā ir jābūt 0.

Pirmais *find* komandas parametrs arī ir neobligāts. Ja tas nav norādīts, tiks atlasīti vīsi dokumenti. Tas ir analoģisks SQL SELECT komandas palaišanai bez WHERE bloka. Lai acīmredzami parādīt, ka pieprasījums tieši definē, ka vajag atlasīt visus dokumentus, ir labāk norādīt *find* komandas pirmo parametru ka tukšas figūriekavas. Abas zemāk norādītas komandas ir sintaktiski likumīgas, un atgriezīs vienu un to pašu rezultātu:

- db.collection.find()
- db.collection.find({ })

Tabulā 2.4. tiek salīdzināti divu veidu jauninājumu pieprasījumu komandas

2.4. tabula

Jauninājumu komandu salīdzinājums

SQL komanda	**MongoDB komanda**
UPDATE users SET status = "C" WHERE age > 24	db.users.update({ age: { $gt: 24 } }, { $set: { status: "C" } }, { multi: true })
UPDATE users SET age = age + 3 WHERE status = "A"	db.users.update({ status: "A" } , { $inc: { age: 3 } }, { multi: true })

Apskatīsim jauninājumu komandas UPDATE sintaksi:

db.collection.update(<pieprasījums>, <jauninājums>, <opcijas>)

Pirmais parametrs definē dokumentus, kuri tiks atjaunināti. Otrais parametrs apraksta izmaiņas, kas būs pielietoti atlasītiem dokumentiem. Otrais parametrs ir *SQL* bloka SET analogs. Lai izmainīt eksistējošā lauka vērtību, tiek izmantots modifikators $set. Svarīgi pieminēt, ka *Mongo* dokumenti var saturēt masīvus. Lai pievienotu elementu masīvām, kas ir dokumentā, tiek izmantots modifikators $push.

```
db.users.update(
  { _id: 1 },
```

```
  {
   $push: { employers: { company: 'Microsoft', join: 1963, quit: 1988 } }
  }
)
```

Augšējais koda fragments ievieto jaunu elementu dokumenta masīvā. Par *Mongo* masīva elementiem var būt arī pilnvērtīgi dokumenti. *Mongo update* komandas otrais parametrs var saturēt vairākus parametrus, bet šī darba nolūkos nav obligāti detalizēti apskatīt katru no tiem. Pilno jauninājumu parametra modifikatoru saraksts: 1) lauku jauninājumu modifikatori: $inc, $rename, $set, $unset; 2) masīvu jauninājumu modifikatori: $addToSet, $pop, $pullAll, $pull, $pushAll un $push.

Visvairāk no *SQL* komandas *Mongo* komanda atšķiras ar trešo parametru, kas satur sarakstu ar opcijām. Apskatīsim vairākas update komandas opcijas. Pēc noklusējuma, ja trešais parametrs nav norādīts, *Mongo update* komanda ir analoģiska SQL UPDATE komandai ar LIMIT 1 ierobežojumu. Ja trešais parametrs satur opciju multi:true, tad Mongo update komanda ir analoģiska SQL UPDATE komandai bez LIMIT ierobežojuma.

Vel viena vērtīga update komandas opcija ir upsert. Ja trešais parametrs satur opciju upsert: true, vai upsert: 1, update komanda ievietos jaunu dokumentu, ja dokuments pēc definēta pieprasījuma kolekcijā netiks atrasts. Apskatīsim koda fragmentu:

```
db.users.update(
   { firstname: „John”, lastname: „Doe”, age: 18, height: 184 },
   { $set: { age: 19, height: 185 } },
   { upsert: true }
)
```

Ja lietotājs ar vārdu „John” un uzvārdu „Doe” eksistē kolekcijā ‘users’, šī dokumenta vecuma un auguma vērtības tiks izmainītas uz 19 un 185 respektīvi. Ja dokuments ar šo vārdu un uzvārdu neeksistē iekš kolekcijas ‘users’, ir divi iespējami varianti:

- ja jauninājumu parametrs satur tikai laukus un vērtības, jauns dokuments tiks izveidots ar šiem laukiem;
- ja jauninājumu parametrs satur tikai jauninājumu parametra modifikatorus ($set, $inc u.t.t.), jauns dokuments tiks uztaisīts ar laukiem no pieprasījumu parametra, kuriem tiks pielietoti jauninājumu parametra modifikatori.

Tabula 2.5. parāda *Mongo* datu bāzes jauninājumu komandu variantus ar aktīvu ‘*upsert*’ atribūtu un rezultāta dokumentus, demonstrējot atšķirības starp atgrieztiem rezultātiem..

2.5. tabula

Jauninājumu komandas ar aktīvu upsert atribūtu

Update ar upsert: true	**Rezultāts**
db.users.update({ firstname: „John”, lastname: „Doe”, age: 30, height: 180 }, { firstname: „John”, lastname: „Doe”, age: 33, height: 185 }, { upsert: true })	{firstname: „John”, lastname: „Doe”, age: 33, height: 185}
db.users.update({ firstname: „John”, lastname: „Doe”, age: 18, height: 184 }, { $set: { age: 19, height: 185 } }, { upsert: true })	{firstname: „John”, lastname: „Doe”, age: 19, height: 185}

Dokumentu dzēšanas pieprasījumi ir ļoti līdzīgi *SQL* sintaksei. Tabulā 2.6. ir parādīti divi dzēšanas pieprasījumu piemēri: ar un bez atlasīšanas pieprasījuma parametra.

2.6. tabula

Dzēšanas komandu salīdzinājums

SQL komanda	**MongoDB komanda**
DELETE FROM users WHERE status = "D"	db.users.remove({ status: "D" })
DELETE FROM users	db.users.remove( )

Dokumentu dzēšanas *Mongo* komanda remove sintakse definē divus parametrus:

db.collection.find(<pieprasījums>, <tikaiViens>)

Ja abi parametri nav norādīti, tiks nodzēsti visi kolekcijas dokumenti, analoģiski SQL pieprasījumam „DELETE FROM table_name”. Pēc iestatījuma, atšķirībā no jauninašanas komandas update, tiks dzēsti visi dokumenti, kuri būs atlasīti ar pieprasījuma parametru. Lai

nodzēst tikai vienu dokumentu, kas atbilst pieprasījumu parametram, ir jāpadod otrais boolean tipa parametrs. Sekojošs pieprasījums nodzēsīs tikai vienu lietotāju, kuram ir 24 gadi:

```
db.users.remove( { age: 24 }, true )
```

3. VEIKTSPĒJAS TESTĒŠANA

Lai pierādītu, ka *NoSQL* datu bāžu lietošana sadarbībā ar *ADF* ietvaru var sniegt būtiskus uzlabojumus, ir nepieciešams salīdzināt *NoSQL* un *SQL* datu bāžu veiktspēju vairākos scenārijos:

- Ierakstu pievienošanas ātrums tabulai ar/bez indeksiem,
- Dokumentu pievienošanas ātrums ar/bez indeksiem,
- Dokumentu pievienošanas ātrums drošā/nedrošā režīmā.

ADF ietvars atbalsta integrāciju ar vairākām relāciju datu bāzēm, bet visbiežāk tas tiek lietots ar Oracle datu bāzi. Šī veiktspējas testi tiks izpildīti, izmantojot sekojošas datu bāzes: MongoDB v2.2.0 un Oracle Database 11g Express Edition.

3.1. Testēšanas sistēmas specifikācijas

Šai demonstrācijai tiek izmantoti sekojošie sistēma un programmatūra. Sistēmas īpašības:

- Windows 7 Professional 64-bit operatīva sistēma
- Intel(R) Core(TM) i5-3570K CPU @3.40GHz 4 kodoli
- 8 Gb DDRIII RAM

Izmantota programmatūra:

- JDeveloper Studio Edition v. 11.1.1.6.0
- Integrēts WebLogic lietotņu serveris v. 11.1.1.5.0

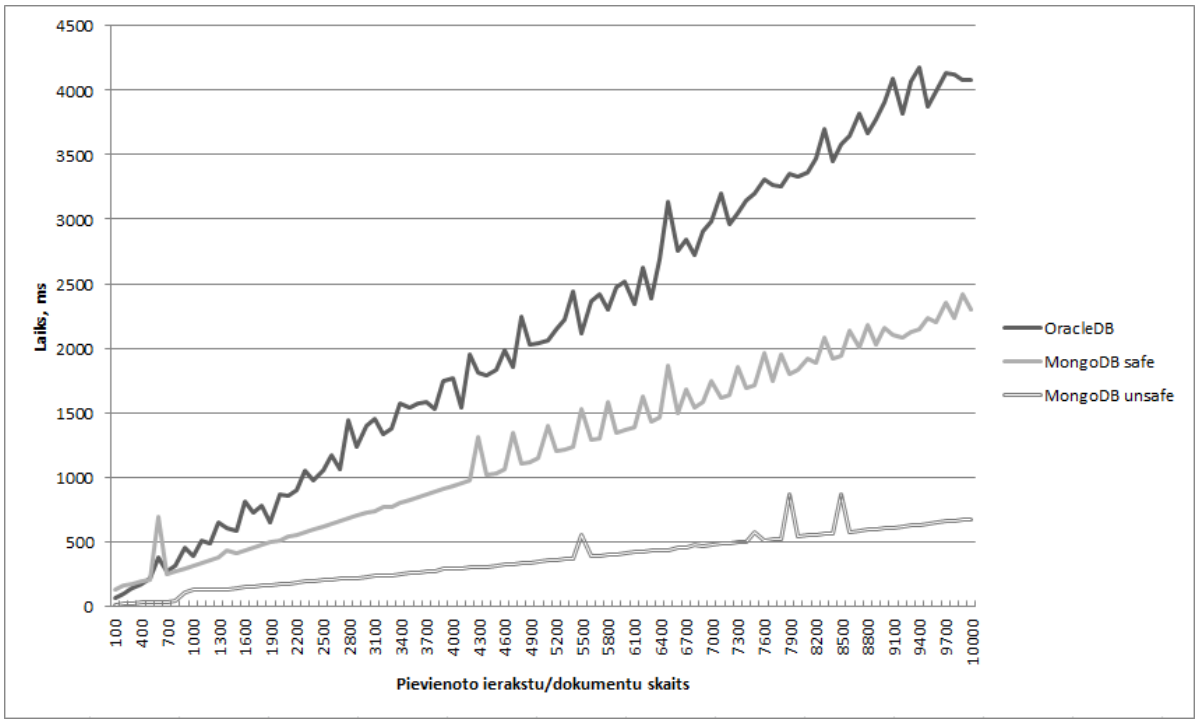
3.2. Rakstīšanas pieprasījumi

NoSQL datu bāžu atbalstītāji galvo, ka šī tipa datu bāzes ir ātrāki par relāciju datu bāzēm, ja pievienošanas pieprasījumu skaits ir liels un katra dokumenta izmērs ir relatīvi mazs. Piemēri būtu vairāki statistikas dati: reitingi, apskatīto bilžu skaits u.t.t.

Pēc noklusējuma *MongoDB* izpilda visus pievienošanas pieprasījumus nedrošā režīmā. Drošā režīmā datu bāzes serveris atgriež atskaiti pēc katra pievienošanas pieprasījuma. Atskaite satur informāciju par izpildīto pieprasījumu. Galvenokārt, vai pieprasījums tika izpildīts veiksmīgi vai nē. Tests tiks izpildīts abos režīmos.

Tests salīdzina laiku, kas ir nepieciešams divām datu bāzēm, lai ierakstītu konkrētu ierakstu skaitu– no 100 līdz 10000. *Mongo* datu bāzes kolekcijas dokuments satur divus laukus: vienu desmit simbolu garu teksta virkni un vienu desmit ciparu skaitli. Oracle datu bāzes tabula satur divas analoģiskas kolonnas.

Pēc katras iterācijas, testa tabulas un kolekcijas tiek iztīrītas. Testa mērījumos netiek iekļauts laiks uz objektu uztaisīšanu un datu dzēšanu no tabulas iterācijas beigās. Tiek skaitīts tikai laiks uz procedūrām ‘collection.insert()’ un ‘statement.execute()’.



3.1.att. **MongoDB un OracleDB veiktspējas salīdzinājums**

Attēlā *3.1.att.* **MongoDB un OracleDB veiktspējas salīdzinājums** skaidri redzam, ka visas trīs funkcijas ir lineāras. Nedrošā režīmā *Mongo* datu bāzes pievienošanas pieprasījumus izpildes ātrums ir būtiski lielāks par Oracle datu bāzi. 10000 ierakstu tiek ierakstīti 4100 milisekunžu laika Oracle datu bāzē, bet *Mongo* nedrošā režīmā ar šo uzdevumu risina 700 milisekunžu laikā. Drošajā režīmā Mongo datu bāzei ir nepieciešams apmēram 2300 milisekundes, lai izpildīt šo uzdevumu.

Ir svarīgi atzīmēt, ka *Oracle* datu bāze vienmēr strādā drošā režīmā. Nedrošo režīmu nevar lietot svarīgo biznesa datu ierakstīšanai – strādājot ar tādiem datiem, ir vienmēr jāsaņem apstiprinājumu no datu bāzes servera par to, ka pievienošanas pieprasījums izpildījās veiksmīgi. Bet ir vairāki specifiski scenāriji, kad nedroša režīma lietošana ir pieņemama.

Nedrošs datu ierakstīšanas režīms ir ļoti populārs reālā laika datorspēļu izstrādē. Tādās lietojumprogrammās spēles serveris saņem no katra spēlētāja milzīgu apjomu informācijas, kas nozīme lielu pievienošanas pieprasījumu skaitu. Serveris ieraksta šo informāciju datu bāzē, lai pēc tam izsūtīt to citiem spēlētājiem. Piemērām, spēlētājam ir jāredz informācija par to, kur uz kartes atrodas viņa komandas biedri, kas notiek dažādos kartes reģionos, cik laika ir palicis līdz nākamajām notikumam u.t.t. Visi šie dati var mainīties vairākas reizes sekundē. Tas nozīmē, ka ja viens no 1000 pievienošanas pieprasījumiem neizpildīsies, lietotājs to pat nepamanīs. Tāpēc nav nepieciešams saņemt apstiprinājumu no datu bāzes servera katram pievienošanas pieprasījumam. Bieži tiek sekojošs paņēmiens – apstiprinājums no servera tiek pieprasīts katrām 1000-jām pievienošanas pieprasījumam. Tas atļauj atrast problēmas, kas neļauj izpildīties vairākiem pieprasījumiem, piemēram, tīkla problēmas starp klientu un serveri.

Lai apstiprināt šī darbā paveiktus veiktspējas testus autors izpētīja E. Intenberga 2012. gada rakstīto maģistra darbu [16]. Darbs pēta vairākas tīmekļa lietojumprogrammu izstrādes ietvarus. Sava darba ietvaros Intenberga kungs analizēja vairāku relāciju un ne-relāciju datu bāžu veiktspējas radījumus. Tika salīdzinātas tādas datu bāzes pārvaldības sistēmas, ka:

- *MongoDB* drošā režīmā (*Mongo safe*)
- *MongoDB* nedrošā režīmā (*Mongo unsafe*)
- *MySQL MyISAM*
- *Memcached*
- *CouchDB*
- *PostgreSQL*

Šo testu rezultāti rāda analoģisku tendenci – visātrāk strādā *MongoDB* nedrošā režīmā. *MongoDB* drošā režīmā strādā lēnāk, bet joprojām ātrāk par visiem *SQL* risinājumiem. Autoraprāt, veiktspējas testi pietiekami uzskatami parāda, ka *MongoDB* ir ātrāks par vairākiem *SQL* datu bāzes pārvaldības sistēmām specifiskajos gadījumos, kas nozīmē, ka šī datu bāzes integrācija ar *ADF* var efektīvi uzlabot sistēmas ātrdarbību.

4. ADF PĀRSKATS

Oracle lietotņu izstrādes ietvars (*no angļ.: Oracle Application Development Framework*), kuru normāli sauc vienkārši par Oracle ADF, ir komerciāls programmatūras izstrādes ietvars lielo uzņēmumu vajadzībām.

ADF komponentes sāka parādīties kopš 1999. gada, kad Oracle izlaida ADF Biznesu Komponentus (*no angļ.: ADF Business Components*), ko tad sauca par „JBO" (*no angļ.: Java Business Objects*), un vēlāk par „*BC4J*" (*no angļ.: Business Components for Java"*). ADF lietotņu izstrāde notiek JDeveloper izstrādes vide. Mūsdienu ADF konfigurācija ar modeļu/saistīšanas slāni parādījās kopā ar programmas JDeveloper 9.0.5 versiju.

Oracle ADF atvieglo *Java* programmatūras izstrādes procesu, ļaujot izstrādātājiem nerakstīt infrastruktūras kodu, un koncentrēties uz lietotnes funkcionalitātes. Vajadzīgus infrastruktūras elementus ietvars prot noģenerēt automātiski. ADF turpmāk atvieglo izstrādes procesu, piedāvājot vizuālo un deklaratīvo Java programmēšanas veidu, ko nodrošina Oracle JDeveloper.

Oracle ADF implementē Modelis-Skats-Kontrolieris dizaina šablonu, piedāvājot integrētus risinājumus katra šablona līmeņa izstrādei. Paplašinot *MVC* šablonu, *ADF* integrējas ar *Oracle SOA* un *WebCenter Portal* ietvariem, kas vienkāršo kompozīto lietotņu izstrādi.

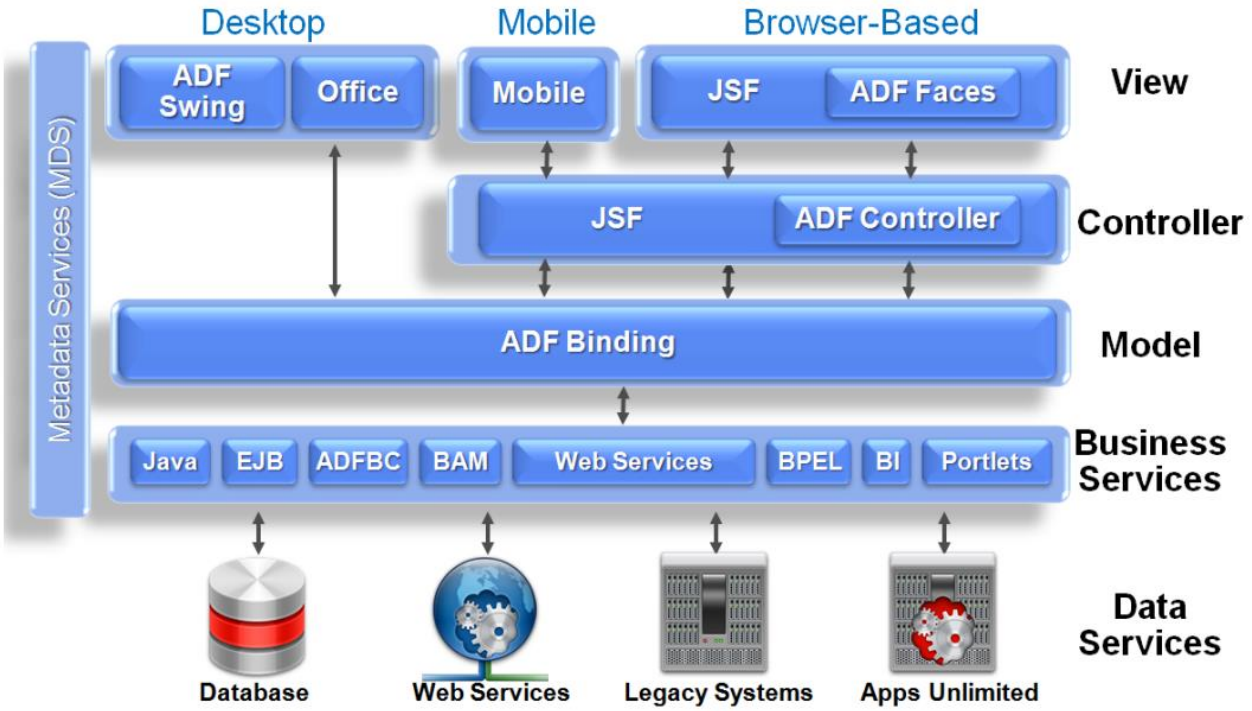
Lietotāju saskarnes izstrādei tiek izmantota vilkšanas un nomēšanas tehnika, kas atļauj ievietot lapā ne tikai grafiskus elementus, bet arī modeļa elementus. JDeveloper spēj pārkonvertēt šos datu avotus tabulās, formās vai grafos. Izstrādātājs var izvēlēties, vai tabulas kolonnas atbalstīs kārtošanu, filtrēšanu, vai būs iespējams atzīmēt tabulas rindu vai rindas, un viss ar grafiska vedņa palīdzību. Rezultātā tiek noģenerēti lapas elementi, kur pirmkoda formā ir *XML* elementi. Tas, savukārt, nozīmē, ka visu šo darbu var izpildīt ari manuāli, pirmkoda līmenī [2].

Oracle ADF cieši izmanto „Modelis-Skats-Kontrolieris (Model-View-Controller jeb *MVC*) projektēšanas šablonu. *MVC* lietotne ir sadalīta uz: 1) modeļu slānis, kas pārvalda lietotnes komunikāciju ar datu bāzēm, 2) skatu līmenis, kas apraksta lietotnes lietotāju saskarni, un 3) kontrolieru slānis, kas pārvalda lietotnes plūsmas un nodrošina komunikāciju starp modeļu un skatu slāņiem. Sadalot lietotni šīs trīs slāņos, vienkāršojas moduļu izstrāde, uzturēšana un izmantošana citās lietotnēs. Slāņi ir vāji sapāroti, kas ir Servisu-orientētas arhitektūras pamatrādītājs.

ADF ne tikai implementē Modeļa-Skata-Kontroliera (*MVC, jeb Model-View-Controller*) šablonu, bet arī paplašina to. *ADF* arhitektūra balstās uz četriem, nevis trīs, slāņiem:

- Biznesa servisu slānis – nodrošina datu pieejamību no dažādiem datu avotiem;
- Modeļu slānis – piedāvā abstrakcijas līmeni virs Biznesa Servisu slāņa, kas atļauj Skatu un Kontrolieru slāņiem efektīvi strādāt ar dažādām Biznesa Servisu slāņa implementācijām;
- Kontrolieru slānis – piedāvā plūsmu kontroles mehānismu;
- Skatu slānis – piedāvā nepieciešamas komponentes lietotāju saskarnes izstrādei.

Oracle ADF atļauj izstrādātājiem izvēlēties tehnoloģijas katra slāņa izstrādei. Attēlā 4.1. dotā diagramma parāda vairākas izvēles opcijas, kas ir pieejams izstrādātājam ar *Oracle ADF*.



4.1. att. **ADF slāņu sadalījums** [2]

Integrācijas slānis, kas atļauj lietot savienot tik lielu skaitu ar *Java EE* tehnoloģijām un pataisa izstrādi tik elastīgu, ir modeļu slānis. *EJB*, tīmekļa pakalpojumi, *JavaBeans*, *JPA/EclipseLink/TopLink* objektus var izmantot ka biznesa servisus *Oracle ADF* modelī. Skatu slānis var iekļaut tīmekļa bāzētas saskarnes, kas ir implementētās ar JSF (*Java Server Faces*), *Dekstop Swing* lietotnes un *MS Office* interfeisus, ka arī mobilo ierīču saskarnes.

4.1. Biznesa servisu slānis

Biznesa servisu slānis pārvalda lietotnes saziņu ar datu avotiem. Biznesa servisu slānis definē, kādā formā lietotnē redzēs datus. Tos var nosaukt par datu avotu iekapsulēšanas apvalku. Biznesa servisu slāni *ADF* ietvarā var implementēt šādos veidos:

- Ka parastas Java klases (*POJO*);
- *EJB*, tīmekļa pakalpojumi;
- *JPA* objekti;
- Oracle *ADF* Biznesa Komponenti.

Pastāv arī iespēja izmantot *XML* un *CSV* failus kā datu avotu, kā arī *REST* (*ReStruturedText*) failus. Šī darba mērķu sasniegšanai ir nepieciešams apskatīt tieši *ADF* biznesa komponentes, jo tie ļauj apstrādāt datus no nestandarta datu avotiem.

4.2. Kontrolieru slānis

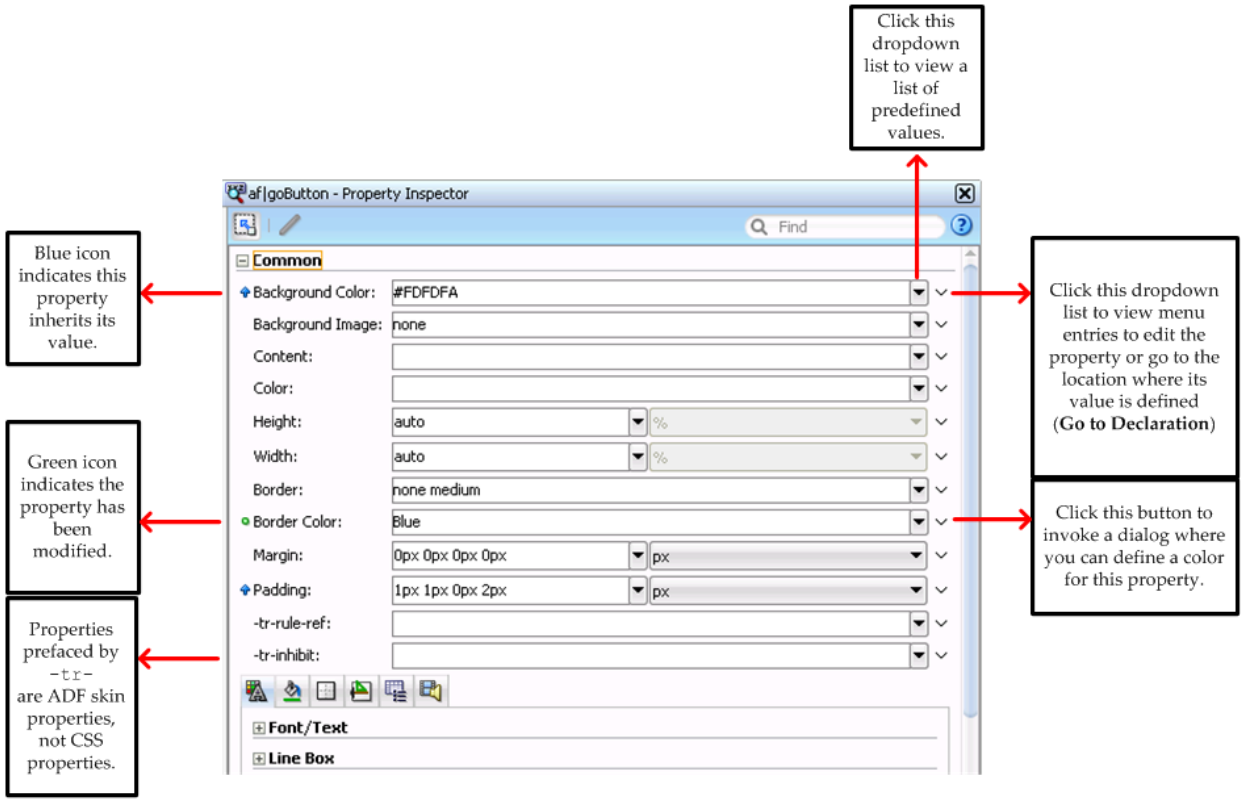
Kontrolieru slānis pārvalda lietotnes plūsmas un sākotnei apstrādā lietotāju ievaddatus. Piemērām, kad lietotājs uzspiež un pogu ‘Meklēt’, kontrolieris norāda, kāda darbība ir jāizpilda, lai izpildās meklēšanas kods, un izpilda navigāciju (rezultātu lapa).

JDeveloper piedāvā divus kontrolieru variantus priekš tīmekļa lietotnēm: standarts JSF kontrolieris vai *ADF* Kontrolieris, kas paplašina JSF kontroliera funkcionalitāti. Izmantojot jebkuru no divu kontrolieru tipiem, *ADF* atļauj izstrādātājam automātiski attēlot navigāciju un darbību izvēli diagrammas formā. Kontrolieri var izstrādāt arī pirmkoda veidā, kas ir XML.

Ar *ADF* Kontrolieri izstrādātājs var sadalīt lietotnes plūsmās vairākās, vieglāk pārvaldāmās, uzdevumu plūsmās (no angļ.:*task flow*), ievietot plūsmās nevizuālas komponentes un lēmumu punktus, veidot lapu fragmentus, kuri strādā lapas reģionos. Šīs piegājiens mudina maksimālo atkārtoto lietotāju saskarnes komponenšu izmantošanu, kā arī atvieglo šo komponentu integrāciju jaunos projektos.

4.3. Skatu slānis

Skatu līmenis reprezentē lietotāju saskarni ar lietotni. Izstrādātāji izmanto gatavo komponentu kopas, un pievieno tas saskarnes elementiem – lapas un fragmenti. Kā arī citos slāņos, *JDeveloper* piedāvā iespēju rediģēt komponentes vairākās veidos: 1) pirmkoda rediģēšana; 2) rediģēšana, izmantojot paleti, ko sauc par atribūtu inspektoru (*no angļ.: Attribute Inspector palette*).



4.2. att. **ADF atribūtu inspektors** [3]

Attēlā 4.2. var redzēt pogas elementa atribūtus, kurus var rediģēt ar grafisko vedni. Tas atļauj apskatīt lietotāju saskarnes elementu cilvēkam draudzīgā formātā, un būtiski samazināt drukas kļūdu skaitu. Vednis māk identificēt nederīgas atribūtu vērtības, kā arī piedāvāt definētas vērtības.

Datu kontroles palete piedāvā iespēju izveidot tabulu vai formu saskarnes elementu, izmantojot vilkšanu un nomēšanu. Biznesa Servisu komponenti tiek nomesti lapās vai lapu fragmentos, un ADF automātiski noģenerē saistīšanas kodu un izveido grafiskas komponentes.

5. ADF UN NOSQL DATU BAŽU INTEGRĀCIJA

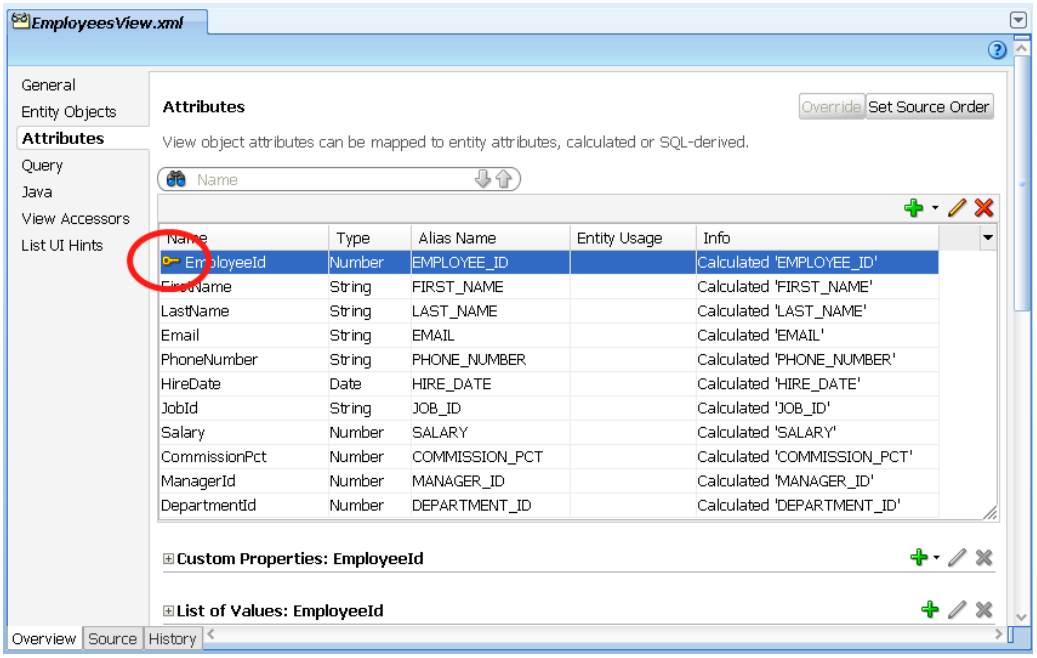
Šajā nodaļā tiek apskatītas *ADF* un *NoSQL* integrācijas iespējas un autora piedāvāti integrācijas problēmu risinājumi. Lai atrast labākas integrācijas pieejas, autoram bija jāņem vērā *Oralce* izstrādes vides *JDeveloper* un vairāku protokolu ierobežojumus.

5.1. ADF paplašinājumu iespējas

Šī darba mērķu sasniegšanai, ir jāapskata tieši *ADF* Biznesu Komponenšu, jeb *ADF* BK paplašinājumu iespējas. Biznesu komponentes atļauj izstrādātājiem nerakstīt lietotnes infrastruktūras kodu, kas ir saistīts ar:

- Lietotnes savienojumu ar datu bāzi;
- Datu lasīšanu (no datu bāzes);
- Datu bloķēšana;
- Transakciju kontroli.

ADF piedāvā noģenerēt gatavus objektus, kurus var izmantot, lai ielasīt, ierakstīt vai modificēt datus no datu bāzes. Galvenie BK instrumenti ir Skatu objekts un Entītiju objekts, jeb VO un EO respektīvi. Izstrādātāji, kas ir rakstījuši kodu, kas savieno datu bāzi ar lietotni, ir rakstījuši entītiju objektus kā parastas Java klases, kur katrs lauks atbilst vienai relāciju datu bāzes tabulas kolonnai.



5.1. att. **ADF skatu objekts**

ADF BK noģenerē šos objektus automātiski, un piedāvā daudz iespēju modificēt šos objektus. lai labāk pielāgot tos lietotnes vajadzībām. Visi BK instrumenti iekš *ADF* ir *XML* objekti, kurus var rediģēt gan pirmkoda veidā, gan ar vedņa palīdzību, ko var redzēt attēlā 5.1.*5.1*.5.1.

Viens no populārākiem datu avotiem, uz kura bāzes tiek uzbūvēti EO un VO ir relāciju datu bāzes tabulas. Izstrādātājs nodrošina savienojumu ar nepieciešamo datu bāzi, un *ADF* veidnis atļauj izvēlēties nepieciešamas tabulas. Var izvēlēties arī dažas, nevis visas, tabulu kolonnas, kā arī definēt, vai VO būs rakstīšanas, vai tikai lasīšanas atļauja.

ADF piedāvā arī citas iespējas un citus datu avotus, uz kuru bāzes var uztaisīt biznesa komponenšu elementus. Šī darba uzdevumu izpildei ir svarīgi apskatīt divas ADF biznesa servisu slāņa paplašinājuma iespējas:

- Programmatiski aizpildītie VO;
- Entītiju objekti, kas balstās uz tīmekļa pakalpojumiem;
- Metodes, kas ir izpaustas no lietotņu moduļiem.

Apskatīsim visus minētus variantus. Programmatiski aizpildītie skatu objekti tiek aizpildīti no koda, un tāpēc teorētiski atļauj aizpildīt savus elementus no *NoSQL* datu bāzēm. Integrācijas darbs šajā gadījuma ir uzprojektēt klases, vai klašu bibliotēku, kas nolasītu datus no *NoSQL* datu bāzes, apstrādātu vai nokartētu tos datus, un programmatiski aizpildītu skatu objektus ar šiem datiem.

Otra pieeja, kuru izmanto daudz biežāk, ir balstīt entītiju objektus uz eksistējošiem tīmekļa pakalpojumiem. Lai integrēt *NoSQL* datu bāzi un *ADF*, izmantojot šo pieeju, vajag uztaisīt tādus tīmekļa pakalpojumus, kurus entītiju objekti varēs uzskatīt par korektu datu avotu.

Cita potenciāla integrācijas pieeja ir izmantot *ADF* lietotnes moduļus, lai iekapsulēt metodes, kas izsauc *NoSQL* datu bāzi.

Pēc dziļākas pētīšanas tika konstatēts, ka programmatiskie skata objekti piedāvā interfeisu, kas ir pārāk grūti pielāgojams risinājumiem, kas neizmanto *Java* valodas *SQL* bibliotēku. Otrais risinājums, kas balstās uz tīmekļa pakalpojumu izmantošanas, piedāvāja lielāku datu sūtīšanas ērtību starp lietojumprogrammas klienta un modeļa daļām. Vel lielāku brīvību var sasniegt iekapsulējot datu bāzes izsaukšanu lietotņu moduļos – *ADF* ietvara kontroliera slāņa elementos. Šo iemeslu dēļ tika izlemts aprakstīt integrācijas iespējas izmantojot tikai divas pēdējās pieejas.

5.2. NoSQL shēmas nepieciešamība

Viens no *NoSQL* raksturīgākām īpašībām ir stingri definētas shēmas prombūtne. Šī īpašība dod datu bāzes arhitektam un lietojumprogrammas izstrādātājiem vairāk brīvības, atļauj pievienot jaunus laukus dokumentiem, kuri tiek ievietoti vecās kolekcijās, neieviešot nekādas izmaiņas datu bāzē. Ir svarīgi atzīmēt, ka izstrādātāja lielāka brīvība, it sevišķi modeļa slānī, var potenciāli uztaisīt kolekciju datus par pārāk nekonsistentiem.

Autoraprāt, lielākais ieguvums no shēmas prombūtnes ir iespēja pārvietot kolekciju shēmu definīciju lietojumprogrammas modeļa slānī. Tas nodrošina, kurus modeļa slānis ieraksta datu bāzē, atbilst pašreizējai kolekcijas shēmai. Shēmas definēšana modeļa slānī dod izstrādātājiem vairākas priekšrocības:

- Lauka nosaukuma jeb atslēgas maiņa,
- Datu objektu klases, kas var tikt izmantotas datu rakstīšanai un lasīšanai,
- Datu objektu klases, kas var tikt izmantotas *ADF UI* elementu aizpildīšanai.

5.3. Tīmekļa pakalpojums un tīmekļa pakalpojuma datu kontrole

Šī integrācijas pieejas realizācijai ir nepieciešams uztaisīt sekojošas komponentes: 1) tīmekļa pakalpojums, 2) tīmekļa pakalpojumu datu kontrole, un 3) JSF lapa ar tabulu. Lai izsauktu tīmekļa pakalpojumu pa taisno no *Java* koda, ir iespējams uztaisīt tīmekļa pakalpojuma starpnieku (*no angļ.: web-service proxy*).

5.3.1. Nepieciešamo komponentu uztaisīšana

Šajā sadaļā tiek aprakstīts nepieciešamo komponentu uztaisīšanas process *JDeveloper* vidē. Komponentu uztaisīšanai tiks izmantoti izstrādes vides komponenšu uztaisīšanas veidni.

5.3.1.1. Tīmekļa pakalpojuma uztaisīšanas process

Šīs pieejas vajadzībām ir jāuztaisa Java EE tīmekļa pakalpojumu. Apskatīsim tīmekļa pakalpojumu izveidošanas procesu JDeveloper izstrādes vidē. Lai uztaisīt vajadzīgo tīmekļa pakalpojumu tika izmantoti sekojošie soli:

1) *File -> New*
2) Kategoriju sarakstā ir jāizvēlas '*Business Tier*' un zem tā '*Web Services*'
3) Jāizvēlas '*Java Web Service*' labajā loga daļā un jāuzspiež uz '*Next*'
4) Jāizvēlas klase, kas satur metodes, kuras būs pieejamās caur tīmekļa pakalpojumu, un spiežam '*Next*' divas reizes
5) Metožu logā ir jāizvēlas metodes, kuras būs pieejamas caur tīmekļa pakalpojumu un spiežam '*Finish*'

5.3.1.2. Tīmekļa pakalpojumu starpnieka uztaisīšanas process

Lai izsauktu tīmekļa pakalpojumu no Java koda, ir nepieciešams saģenerēt tīmekļa pakalpojumu starpnieku (*no angļ.: web service proxy*). Tīmekļa pakalpojumu starpnieka ģenerēšanai ir nepieciešams tikai tīmekļa pakalpojuma WSDL adrese.

WSDL, jeb tīmekļa pakalpojumu definēšanas valoda (*no angļ.: Web Service Description Language*), apraksta tīkla pakalpojumus *XML* valodas forma. *WSDL* dokuments ir viena konkrēta tīmekļu pakalpojuma apraksts. Tas apraksta sagaidāmus ziņojumus, izpaustas metodes, un tīklu protokolus, kurus izmanto tīmekļa pakalpojums. *JDeveloper* izstrādes vides tīmekļu pakalpojuma starpnieka uztaisīšanas veidnis sagaida tīmekļa pakalpojuma *WSDL* dokumenta adresi, kas var būt failu sistēmas ceļš, vai *URL*, kas norāda uz lietotņu serverī izvietoto *WSDL* dokumentu. *WSDL* dokumenta piemēru var apskatīt 1. pielīkumā.

Standarta *ADF* lietotne ir sadalīta divos projektos – modeļa projekts un skata-kontroliera projekts. No projektējuma viedokļa, labāk ir izvietot modeļa tīmekļa pakalpojumu modeļa projektā, bet pakalpojuma starpnieku izvietot skatu-kontrolieru projektā. Lai saģenerētu tīmekļa pakalpojuma starpnieku *JDeveloper* izstrādes vidē, izmantosim uztaisīšanas veidni. Sekojošie soļi apraksta pakalpojuma starpnieka uztaisīšanas procesu:

1) *File -> New*
2) Kategoriju sarakstā ir jāizvēlas '*Business Tier*' un zem tā '*Web Service*s'
3) Jāizvēlas '*Web Service Proxy*' labajā loga daļā un jāuzspiež uz '*Next*'
4) '*WSDL Document URL*' ievadlaukā ievadam *WSDL* dokumenta adresi failu sistēma vai serverī izvietota *WSDL* dokumenta *URL* adresi, un uzspiest '*Next*'.
   WSDL URL piemērs: http://localhost:7101/JavaMongoWeb-ViewController-context-root/MapService?WSDL
5) Jāievada *Java* pakotni, kurā tiks saģenerēts starpnieks, un jāuzspiež '*Finish*'.

Izstrādes vides veidnis saģenerēs visus nepieciešamus failus, un izvietos tos 5. solī norādītajā pakotnē. Lai izsaukt tīmekļa pakalpojumu caur starpnieku, izstrādes vide uztaisa klienta failu. Tas satur tīmekļa pakalpojuma klases pārstāvja instanci, kura satur visas izpaustas metodes. Ir svarīgi atzīmēt, ka tīmekļa pakalpojuma starpnieka pārģenerēšanas procesā šīs fails tiks pārrakstīts, tāpēc tīmekļa pakalpojuma izsaukšanas loģiku ir vērtīgi glabāt atsevišķā failā.

5.3.2. Tīmekļa pakalpojumu izpaustas metodes

Tālāk tiks aprakstīta tīmekļu pakalpojumu iespējamo pielietojumu analīze. Visparastākais veids ir rakstīt un izpaust metodes, kas izpilda konkrētu uzdevumu. Piemērām, apskatīsim sekojošas metodes deklarāciju un Java dokumentāciju.

```
/**
 * Metode sanem timekla lapas identifikatoru un izpilda sekojoso komandu
 * db.collection.update( { "pageId" : pageId }, { $inc : { „hitCount” : 1 } } );
 * @param pageId - lapas identifikators
 */
public void addHit(int pageId);
```

Vairākas pastāvīguma bibliotēkas, piemēram „*Hibernate*”, piedāvā konceptu, ko sauc par iepriekšdefinētiem pieprasījumiem (*Hibernate* termins – *Named Query*). Metode *addHit()* ir iepriekšdefinēta pieprasījuma analogs.

Tīmekļa pakalpojums tiek izvietots serverī atsevišķi no klienta lietojumprogrammas. Tas atļauj izmantot šo tīmekļa pakalpojumu ka patstāvīgu modeli, jeb datu avotu. To var izmantot vairākas lietotnes vai lietotnes komponentes. Galvenais mīnus ir tas, ka lai pievienot vai rediģēt iepriekšdefinēto pieprasījumu, ir nepieciešams atkārtoti izvietot tīmekļa pakalpojumu uz servera.

Apskatīsim alternatīvu viedu, kā izmantot tīmekļa pakalpojumu ka datu modeli. Tas atļaus izpaust metodes, kas ir līdzīgākas pastāvīguma bibliotēku metodēm. Ir nepieciešami vairāki dizaina elementi:

- Datu objekta abstrakta klase,
- Datu objekta klasi mantojošas klases, un
- Shēmas enumerācijas klase, kas ir iekļauta datu objekta bērnu klasē

Shēmas enumerācijas klase palīdz nodefinēt kolekcijas dokumentos eksistējošus laukus. Shēmai, kas ir definēta lietojumprogrammas klientu pusē, ir priekšrocības, jo tas atļauj datu objekta lauku aizpildīšanai izmantot piekļūvējmetodes, un neizmantot lauku nosaukumus, kuri var mainīties. Pilnus klases definīcijas ir apskatāmas 3. pielikumā.

5.3.3. Modeļu tīmekļa pakalpojuma un klientu lietotnes mijiedarbība

Tā ka ADF ir bāzēts uz Java programmēšanas valodas, tiek nodemonstrētas Java klases, kas pārstāv kolekcijas shēmu, un datu objekti. Lai nodrošinātu stabilu datu modeļa slāņa interfeisu, tiek izmantotas sekojošas klases:

- *DataObject* – abstrakta datu objekta klase
- *PersonDO* – datu objekta implementācija
- *Schema* – enumerācijas klase

Abstraktā klasē *DataObject* tiek definētas visas metodes, kuras ir jāpārlādē šīs klases mantotājiem. Ir iemesls, kāpēc šai lomai tiek izmantota tieši abstrakta klase, nevis Java interfeiss – Java tīmekļa pakalpojumu parametru tipa ierobežojumi.

```
1. public int insert(List<DataObjectInterface> listOfDataObjects);
2. public int insert(List<DataObjectAbstractClass> listOfDataObjects);
```

Sarakstā pirmais elements ir metodes deklarācija, kuru nevar izpaust ka tīmekļa pakalpojuma metodi. Otrs piemērs ir likumīga tīmekļa pakalpojuma metodes deklarācijas piemērs, kas saņem kolekciju ar abstrakto klasi mantojošas klases instancēm.

JAXB un *JAX-WS* tīmekļa pakalpojumu standarti neatbalsta interfeisa parametrus, izņemot kolekcijas. Tas nozīme, ka *Map* interfeisu nevar izmantot datu objektu padošanai, ka arī nedrīkst izmantot sarakstu sarakstus (*List<List<String>*). Vienīgais veids, kas ir pieejams, ir padot divus sarakstus ka parametrus: viens satur lauku nosaukumus, un otrais lauku vērtības.

Datu objektu implementācijas klase, piemēram, *PersonDO.java*, implementē visas abstraktas metodes, kas ir deklarētas klasē *DataObject*.

Shēmu aprakstoša enumerācijas klase satur sekojošo informāciju:

- Datu bāzes nosaukums,
- Kolekcijas nosaukums,
- Lauka nosaukums datu objektā,
- Lauka nosaukums datu bāzē,
- Vecs lauka nosaukums datu bāzē (opcionāls).

Aplūkosim rakstīšanas pieprasījumu metodes deklarāciju. Lai ierakstīt datus, ir nepieciešama sekojoša informācija: datu bāzes nosaukums, kolekcijas nosaukums, un viena vai vairāku dokumentu informācija - katra lauka nosaukums, jeb atslēga, un katra lauka vērtība. Metodi ar šādu deklarāciju var izpaust ka tīmekļa pakalpojuma metodi:

public boolean insert(String db, String collection, List<DataObject> documents);

Lai dabūt katra dokumenta lauku nosaukumus un vērtības, *DataObject* abstrakta klase deklarē metodes *getFieldNames()* un *getFieldValues()*. Ir iespējams apvienot šīs divas metodes, un deklarēt vienu metodi, kas atgriež *Map<String, Object>*, kur atslēga ir lauka nosaukums datu bāzē, un vērtība ir šī lauka vērtība. Bet, ņemot vērā, ka tīmekļa pakalpojumu protokoli neļauj izpaust metodes ar *Map* parametru, ir jāizmanto divas metodes, kas atgriež sarakstus.

5.3.3.1. Datu nolasīšana

Pieradīsim, ka ir iespējams nolasīt datus no *Mongo* datu bāzes, un parādīt tos JSF lapā. Pirmkārt, ir jāsagatavo datu avots, kuru ADF varēs izmantot. Tiek uztaisīta *POJO* klase, kuru tiek izmantota ka entītiju objekts. Koncepta pieradījumam tiek atlasīts viens lauks no viena dokumenta no kolekcijas 'students', un tā vērtība tiek attēlota ar *af:outputText* JSF komponenti.

Pirmkārt, tiek uztaisīts tīmekļa pakalpojums, kurā ir izpaustas divas metodes:

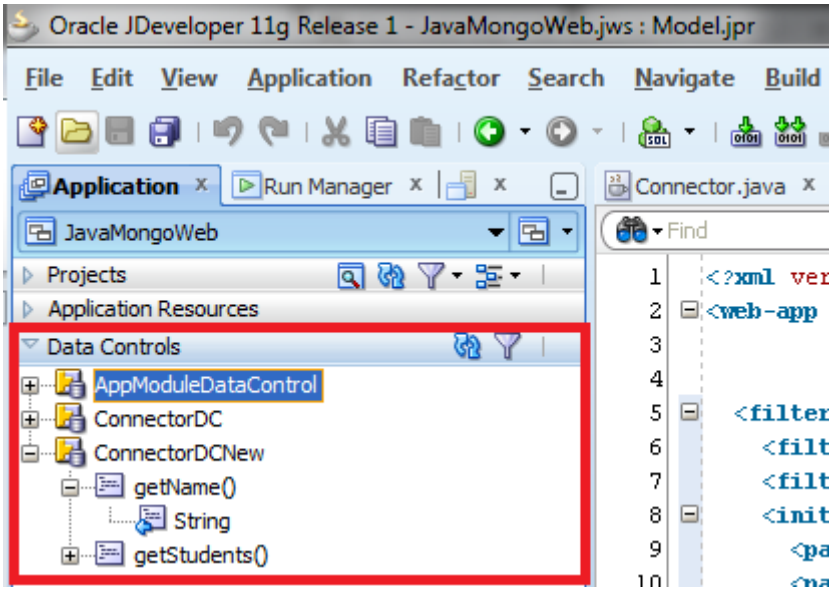
- *getName()* – atgriež pirmā dokumenta '*name*' lauka vērtību no kolekcijas 'students'
- *getStudents()* – atgriež *StudentsDO* klases instanču sarakstu

Tīmekļa pakalpojuma uztaisīšanas process ir aprakstīts sadaļā 5.3.1.1.

Nākamais solis – uztaisīt datu kontroli no eksistējošā tīmekļu pakalpojuma. Lai to izdarīt, ir jāizpilda sekojošie soli:

1) *File -> New*
2) Kategoriju sarakstā izvēlamies '*Business Tier*' un zem tā '*Web Services*'
3) Jāizvēlas '*Web Service Data Control*' labajā loga daļā
4) Ir jādefinē datu kontroles nosaukums un ievadam tīmekļa pakalpojuma adresi. Tīmekļa pakalpojuma adrese ir *URL* uz tā *WSDL* dokumentu Piezīme: uz datu kontroles uztaisīšanas momentu tīmekļa pakalpojuma *WSDL* ir jābūt pieejamam. Spiežam '*Next*'
5) Ja tīmekļa pakalpojums ir pieejams, nākamajā logā parādīsies metožu izvēlne. Ir jāizvēlas tīmekļa pakalpojuma metodes. Šai demonstrācijai ir jāizvēlas metode *getName()* un jānospiež '*Finish*'.

Gatavo datu kontroli var apskatīt '*Data Controles*' palīglogā, kas atrodas izstrādes vides galvenā loga kreisajā pusē, un ir parādīts attēlā *5.2*.



5.2. att. **Datu kontroles palīglogs**

Tālāk jāizmanto uztaisīto datu kontroles elementu, lai parādīt *getName()* metodes rezultātu *JSF* lapā. Jaunajā JSF lapā vajag pavilkt un nomest *getName()* metodi no datu kontroles palīgloga formas elementā. *ADF* piedāvā vairākus datu avotu attēlojumu iespējas. Šai demonstrācijai ir izmantota *ADF* poga. Uzspiežot uz šo pogu, tiks palaista tīmekļa pakalpojuma metode *getName()*, kas atgriezīs nepieciešamo vērtību. Lai šī vērtība parādītos uz lapas, iemetīsim *getName()* metodes rezultātu no datu kontroles palīgloga kā teksta izvada elementu '*ADF Output Text*'.

5.3. att. **Nolasīšanas pieprasījuma izpilde izmantojot JSF komponentes**

Attēls 5.3. parāda lapas izskatu pēc tā atvēršanas un pēc pogas *getName()* uzspiešanas. Uz ekrāna var redzēt vērtību, kas tika atlasīta no *Mongo* datu bāzes. Šīs piemērs uzskatami pierada, ka tīmekļa pakalpojums savienojumā ar *ADF* datu kontroles mehānismu atļauj atlasīt datus no NoSQL datu bāzes un parādīt tos izmantojot *ADF* lietotāju saskarnes komponentus.

5.3.3.2. Dokumentu atjaunināšana

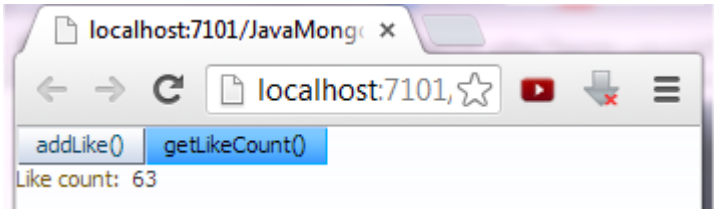
Tagad jāpierāda, ka ir iespējams atjaunināt eksistējošus *Mongo* dokumentus. Šai demonstrācijai izmantosim kolekcijas 'posts' dokumentu 'likeCount' lauku. Ir jānodefinē divas metodes: viena palielinās '*likeCount*' laika vērtību uz 1, un otra nolasīs tekošo '*likeCount*' lauka vērtību no datu bāzes. Izmantosim sekojošas metožu deklarācijas:

```
public void addLike(int studentId) {
public int getLikeCount(int studentId) {
```

Metode *addLike()* saņem studenta identifikatoru ka parametru, palielina lauka '*likeCount*' vērtību uz 1. Metode *getLikeCount()* saņem studenta identifikatoru ka parametru, atlasa studenta dokumentu un atgriež šī dokumenta '*likeCount*' lauka vērtību. Tālāk ir atkārtoti jāizvieto tīmekļa pakalpojumi uz servera, lai jaunas metodes būtu pieejamas. Pēc tam ir jāpārtaisa datu kontroles elements, izmantojot atjaunoto tīmekļa pakalpojumu.

Lai uzskatami nodemonstrēt funkcionalitāti, metodi *addLike*() ir jāievelk lapā no datu kontroles palīglogā, un izvēlēties '*ADF Button*' attēlošanas elementu. Ņemot vērā, ka metodei *addLike()* ir viens parametrs, *JDeveloper* piespiež izstrādātāju norādīt šī parametra vērtību. Šīs demonstrācijas nolūkos ir pietiekami norādīt konstanto vērtību, bet reālas dzīves scenārijā šī vērtība var būt aizpildīta, piemērām, caur teksta ievadlauku.

Otra metode *getLikeCount*() šī piemērā ir arī attēlota pogas veidā, kaut gan izmantojot *ADF* uzdevumu plūsmas (*no angļ.: task flow*) ir iespējams atjaunot vērtību, uzspiežot uz pirmās *addLike()* pogas. Attēls 5.4. parāda lapu, kas satur trīs lietotāju grafiskas saskarnes elementus: divas pogas kas izsauc datu bāzes metodes, un teksta izvada elementu vērtību attēlošanai.



5.4. att. **Modifikācijas pieprasījumu izpilde izmantojot JSF komponentes**

Tagad, kad ir pieradīts, ka *ADF* var izmantot lai gan nolasītu vērtības no *Mongo* datu bāzes, gan atjaunot vērtības, ir vērtīgi pierādīt, ka ir iespējams izmantot tādus *MongoDB* vērtīgas funkcionalitātes kā *MapReduce*.

5.3.3.3. Dati tabulas formā

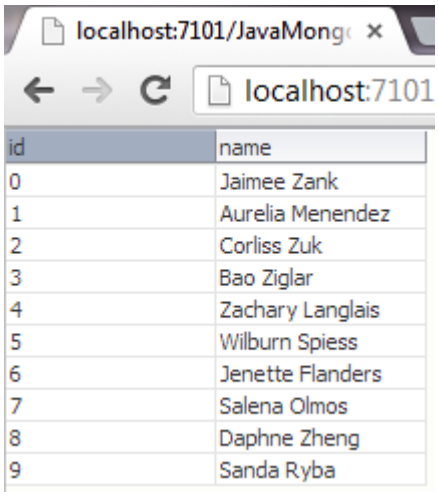
Tagad ir jāpierāda, ka ir iespējams attēlot atlasītus datus no MongoDB tabulas veidā. Šīm nolūkam '*Connector*' klase satur metodi *getStudents()* ar sekojošo deklarāciju:

```
public List<StudentDO> getStudents();
```

Šī klase atlasa visus kolekcijā eksistējošus studentu dokumentus. Ir jāatzīmē, ka datu objekta klasei '*StudentDO*' ir jābūt serializējamai:

```
public class StudentDO implements Serializable {
```

Lai būtu iespējams izmantot *getStudents()* metodi JSF lapā, ir nepieciešams pārģenerēt datu kontroles elementu un pievienot šo metodi redzamo metožu sarakstam. Pēc tam ir nepieciešams atkārtoti izvietot lietotni uz servera. Nākamais solis ir ievilkt datu kontroles elementu JSF lapā un izvēlēties attēlošanas veidu '*Table -> ADF Table*'. Attels 5.5. parāda pirmos desmit ierakstus no studentu kolekcijas, kas ir attēlotas JSF tabulas formā.



id	name
0	Jaimee Zank
1	Aurelia Menendez
2	Corliss Zuk
3	Bao Ziglar
4	Zachary Langlais
5	Wilburn Spiess
6	Jenette Flanders
7	Salena Olmos
8	Daphne Zheng
9	Sanda Ryba

5.5. att. **Dati no Mongo datu bāzes tabulas formā**

5.3.3.4. MapReduce

Nākamais solis ir nodemonstrēt *MapReduce* ietvara lietojamību, izmantojot *ADF*. Apskatīsim *Mongo* dokumentu no kolekcijas, kura tiks izmantota šīm piemēram:

```
> bills.findOne()
{       "_id" : ObjectId("5173fd2cbbc5bdb95b10b271"),
        "accNumber" : "123-1201",
        "billableAmmount" : 30 }
```

Kolekcija '*bills*' satur rēķinus. Katrs rēķins ir piesaistīts vienam un tikai vienam pieslēguma numuram. Šī piemēra uzdevums ir atlasīt visus kolekcijas dokumentus, un atgriezt kopējo summu

apmaksai katram unikālam pieslēguma numuram. Tālāk tiek nodemonstrēta šī uzdevuma risinājums izmantojot *MapReduce* klasterizētā datu bāzes serverī.

Uzrakstīsim *map()* un *reduce()* funkcijas. No katra dokumenta, kas atrodas '*bills*' kolekcijā, mums ir jāuztaisa asociatīva masīva elementu, kura atslēga ir pieslēguma numurs, un vērtība ir masīvs ar summām apmaksai no katra rēķina zem šī pieslēguma numura.

```
map() {
  emit(this.accNumber, this.billableAmmount); }
```

Šīs funkcijas rezultātā dabūsim asociatīvo masīvu, jeb vārdnīcu, ko varam izmantot, ka ievada parametru priekš *reduce()* funkcijas.

```
reduce(accNumber, valuesPrices) {
        return Array.sum(valuesPrices);}
```

Rezultātā saņemam vairākus dokumentus ar diviem laukiem: '*_id'* un '*value*'. Šī gadījumā, '*_id'* ir pieslēguma numurs, un '*value*' ir kopēja summa apmaksai šīm pieslēgumam.

```
{ "_id" : "123-1201" , "value" : 70.0}
{ "_id" : "123-2204" , "value" : 99.0}
{ "_id" : "123-3309" , "value" : 153.0}
```

Apskatot attēlu *1.7. att.* **MapReduce izpildes procesa diagramma trīs mezglu klasterī** var redzēt, kā *map()* un *reduce()* funkcijas tiek izpildītas atsevišķos datu bāzes mezglos. Pēc tam *reduce()* funkcijas rezultāti tiek atgriezti uz meistara mašīnu, kas apvieno rezultātus no visiem datu bāzes mezgliem vienā rezultātā.

Šādā veida, izmantojot '*MapReduce*' ir iespējams apkopot datus par katru pieslēguma numuru un attēlot šo informāciju izmantojot *ADF* ietvara grafiskas saskarnes komponentes, piemērām, tabulas formā.

5.3.3.5. Lauka nosaukuma maiņa

Šajā sadaļā tiek apskatīta problēma, ar ko var satikties datu bāzes administratori un arhitekti. Šo problēmu 2011. gadā bija jārisina kompānijas *Forsquare* datu glabāšanas komandai. Šīs problēmas ir detalizēti aprakstīts sadaļā 2.6.1.

Uzdevums ir samainīt *Mongo* datu bāzes kolekcijas lauka nosaukumu tā, lai datu bāze paliktu pieejama lietotājiem bez pārtraukumiem. Lai atrisināt šo uzdevumu, tiek piedāvāts sekojošs risinājums: shēmas enumerācijas klasei pievienots viens papildus Java lauks, kas glabās veco datu

bāzes vērtību. Piemērs demonstrē shēmas enumerācijas klases stāvokli vairākos lietojumprogrammas dzīves posmos:

Kad datu bāze tiek uztaisīta, un datu apjoms ir mazs, lauku nosaukumi neaizņem pārāk daudz datus. Trešais Java lauks netiek izmantots, un paliek tukšs.

```
public enum Schema implements Serializable {
    NAME("name", "full name", ""),
    AGE("age", "age", ""),

    private String currentDOFieldName;
    private String currentDBFieldName;
    private String oldDBFieldName;
...
```

Kad ir nepieciešams izmainīt datu bāzes lauka nosaukumu, pašreizējs datu bāzes lauka nosaukums kļūst par veco un tiek glabāts trešajā laukā. Jaunais datu bāzes lauka nosaukums tiek saglabāts otrajā laukā.

```
public enum Schema implements Serializable {
    NAME("name", "fn", "Full name"),
    AGE("age", "a", "age");

    private String currentDOFieldName;
    private String currentDBFieldName;
    private String oldDBFieldName;
...
```

Shēmas konfigurācija ir pabeigta. Tagad apskatīsim izmaiņas, kas ir nepieciešamas datu modeļa *CRUD* metodēs:

- rakstīšana: visi jauni dokumenti tiek pievienoti ar jaunu lauku nosaukumu;
- lasīšana: ir iespējams implementēt vairākas lasīšanas metožu variantus –
    - atgriež visus dokumentus, gan ar veciem, gan ar jauniem lauku nosaukumiem;
    - atgriež tikai dokumentus ar veciem lauku nosaukumiem;
    - atgriež tikai dokumentus ar jauniem datu nosaukumiem;
- atjaunināšana: tiek atjaunināti dokumenti ar jauniem un ar veciem lauku nosaukumiem;
- dzēšana: tiek dzēsti dokumenti gan ar jauniem, gan ar veciem lauku nosaukumiem.

Tālāk tiek apskatīti piedāvāta risinājuma labumi: lauku nosaukuma izmēra samazināšana – kolekcijā, kura lauks glabā 10 simbolu teksta virkņu tipa identifikatoru, un lauka nosaukums ir arī 10 simbolu teksta virknē, pusi no patērētas vietas aizņem tieši atslēgas, jeb lauku nosaukumi.

Otrkārt, aprakstīta arhitektūra atļauj izmainīt lauka nosaukumu, neieviešot izmaiņas kodā gan tīmekļa pakalpojumā (datu modelis), gan lietojumprogrammas klientu daļā. Vienīgais, kas tiek mainīts, ir datu objekta shēmas klases definīcija.

Tālāk izstrādātājam ir jāizvēlas viena no divām stratēģijām, kuras attiecās uz lasīšanas pieprasījumiem. Pirmkārt, ir jāizvēlas eksistējošo datu pārvēršanas (*no angļ.: data conversion*) pieeja – aktīvā vai pasīvā. Pēc jauna nosaukuma ieviešanas, kolekcija saturēs dokumentus gan ar jauniem, gan ar veciem lauku nosaukumiem. Pasīvā datu pārvēršana strādā sekojošā veida: kad dati tiek nolasīti, visi dokumenti, kas satur vecos lauku nosaukumus, tiek pārsaukti, un tad atgriezti klientam. Aktīva datu pārvēršana notiek datu bāzes apkalpošanas laikā, vai ar skripta palīdzību, kad tiek atlasīti visi dokumenti, kas satur kaut vienu veca parauga lauku nosaukumu, un visi lauki ar veco nosaukumu tiek pārsaukti.

Kompānijas *Foursquare* gadījumā, kas plašāk aprakstīts sadaļā 2.6.1., kolekcija '*checkins*' saturēja datus, kuras gandrīz nekad netika atjauninātas. Pie tam, dokumentu skaits kolekcija bija tik liels, ka bija nepieciešams īsā laikā samazināt lauku nosaukumu garumus. Tāpēc datu glabāšanas nodaļas darbinieki izlēma, kad tiks izmantota aktīva datu pārvēršanas pieeja, un dati tiks apstrādāti ar skripta palīdzību, kas tika palaists katrā klastera mezglā un replikācijas mašīnā. Tādā veida dati tika atjaunināti īsā laikā, un datu bāze palika pieejama visā datu pārvēršanas procesa laikā.

5.3.3.6. Jauna laika pievienošana

Tālāk tiek aprakstīta pieeja, kas izmanto šī darbā piedāvāto datu modeļa arhitektūru (abstrakta klase, mantojoša klase un enumerācijas klase), lai pievienot jaunu lauku eksistējošai datu bāzes kolekcijai, ieviešot pēc iespējas mazāk izmaiņas.

Lai pievienot jaunu lauku vispirms ir jāievieš izmaiņas gan abstrakto datu objektu mantojošā klasē, gan datu objekta shēmas enumerācijas klasē. Izmaiņas, kas ir nepieciešamas datu objekta klasē: ir nepieciešams pievienot jaunu Java lauku un piekļuvējmetodes (*no angļ.: accessors*). Lielākas izmaiņas tiek ieviestas tieši datu objekta shēmas klasē. Ir nepieciešams nodefinēt jauno lauku reprezentējošo enumerācijas elementu, kurā ir norādīts Java laka nosaukums, datu bāzes jaunā lauka nosaukums, un vecs datu bāzes lauka nosaukums, kas ir tukšs jaunam kolekcijas

laukam. Pēc tam paliek atjaunot vietas klienta kodā, lai aizpildīt jauno datu objekta lauku. Vairāk nekādas izmaiņas nav nepieciešamas.

5.4. Datu kontrole no Java klases

Izmantojot tīmekļa pakalpojumu un tīmekļa pakalpojuma starpnieku ir iespējams definēt metodes, kas atlasa vai modificē specifiskus laukus. Lai atlasīt vai ierakstīt sarežģītus objektus un objektu kolekcijas, ir nepieciešams izmantot *ADF* ietvara iespēju ģenerēt datu kontroles komponenti no *Java* klases.

Šī pieeja atļauj pilnībā izmantot autora piedāvāto datu pastāvības slāni, kas sastāv no abstraktas datu objektu klases, datu objektu mantojošām klasēm un shēmas definējošām klasēm. Tālāk tiek apskatītas lasīšanas un rakstīšanas metodes, kuras sadarbojas pa tiešo ar *MongoDB*. Pirmkārt, tiek apskatīts autora konstruēts interfeiss, kas tiek piedāvāts klienta aplikācijai:

public void insert(DataObject documents);

public void insert(List<DataObject> documents);

public List<DataObject> read(Class<? extends DataObject> clazz, String searchQuery);

Izmantojot autora piedāvāto datu modeļa projektējumu, ir iespējams izmantot abstrakto klasi metožu deklarācija, un izmantot šīs metodes ar jebkādu klienta definētu objekta klasi. Vienīgais nosacījums ir lai klienta definēta datu objektu klase manto no klases *DataObject.java*, un pārlādē šīs klases visas abstrakta metodes.

Tālāk tiek aprakstītas metodes, kas fiziski izsauc datu bāzi izmantojot *MongoDB Java* dzini. Autoraprāt, ir svarīgi apskatīt šo metožu pirmkodu fragmentus, jo tie ir autora praktiskā darba galvenie elementi. Pirmkārt, apskatīsim, kā dati tiek ierakstīti datu bāzē:

```
private static void insert(DataObject document) {
    Mongo mongo = getMongo();
    DB db = mongo.getDB(document.getDBName());
    DBCollection collection = db.getCollection(document.getCollectionName());
    BasicDBObject dbObject = new BasicDBObject();
    Map<String, Object> map;
    map = twoListsToMap(document.getCurrentDBFieldNames(),
document.getValueList());
    Object value;
    for (Map.Entry<String, Object> entry : map.entrySet()) {
        value = entry.getValue();
        if (value != null) {
            dbObject.put(entry.getKey(), value);
        }
```

```
    }
    collection.insert(dbObject);
}
```

Koda fragments demonstrē, ka metode *insert()* var apstrādāt jebkuru DataObject.java mantojošo klasi, jo izmanto abstrakta klases metodes *getDBName()*, *getCollectionName()*, *getCurrentDBFieldNames()* un *getValueList()*.

Lai būtu iespējams izmantot lasīšanas metodi ar jebkuru datu objektu, ir nepieciešams izmantot *Java Reflection*[15] bibliotēku.

```
LocalDataObject document = null;
try {
    document = clazz.newInstance();
} catch (IllegalAccessException iae) {
    ...
}
...
while (cursor.hasNext()) {
    DBObject cursorEntry = cursor.next();
        try {
            DataObject dataObject = clazz.newInstance();
            List<String> currentDBFieldNames = document.getCurrentDBFieldNames();
            List<String> currentDOFieldNames = document.getCurrentDOFieldNames();
            for (int i = 0; i < currentDBFieldNames.size(); i++) {
                Field field = clazz.getDeclaredField(currentDOFieldNames.get(i));
                Object fieldValue = cursorEntry.get(currentDBFieldNames.get(i));
                field.set(dataObject, fieldValue);
            }
            resultList.add(dataObject);
        } catch (IllegalAccessException iae) {
            ...
        }
}
```

Ir svarīgi atzīmēt, ka izmantojot tikai klases referenci, ir iespējams rekonstruēt datu objektu mantojošas klases instanci. *Reflection* bibliotēka atļauj konstruēt laukus pēc to nosaukumiem un aizpildīt šo lauku vērtības. Datu objekta instance tiek aizpildīta ar informāciju no kursora, kuru atgriež datu bāze.

ADF atļauj uztaisīt datu kontroles elementu no jebkuras *Java* klases, bet korekti būtu glabāt visas ar datu modeli saistītas klases lietotņu moduļos. Lietotņu moduli ir specifiski domāti datu modeļu elementu izpaušanai skata un kontroliera slāņiem. Normālā *ADF* lietojumprogramma satur vairākus lietotņu moduļus, kuri satur sevī skatu objektus un entītiju objektus. Darba mērķu sasniegšanai metodes, kas atlasa datus no *MongoDB* datu bāzes, ir vērtīgi izvietot lietotņu moduļos, un izpaust šīs metodes skatu slānim. *ADF* automātiski uztaisa datu kontroles elementu no katra lietotnes moduļa. Tas nozīmē, ka datu kontroles elementi no metodēm, kas izsauc *MongoDB* un

atgriež lietotāja definētu datu objektus, var tikt iemesti *JSF* lapās un attēloti lietotāju grafiskas saskarnes elementu formā.

SECINĀJUMI

Maģistra darba galvenais mērķis tika pilnīgi sasniegts. Mērķa sasniegšanai tika izpildīti visi darba ievadā izvirzītie uzdevumi. Darbs ietver gan teorētiskā materiāla analīzi, gan praktisko integrācijas slāņa pielietojuma aprakstu.

NoSQL konceptu un īpašību pētījums parāda, ka ne-relāciju datu bāzes piedāvā daudz vairāk labumu, nekā lielāki rakstīšanas ātrumi. Horizontālās mērogošanas iespējas atļauj lieliem uzņēmumiem uzlabot lietojumprogrammu stabilitāti, palielināt pieejamības laiku un būtiski samazināt izmaksas. '*MapReduce'* koncepts atļauj pielietot vairākas priekšrocības, kuras piedāvā horizontāli mērogota sistēma. Izmantojot '*MapReduce'* ne-relāciju datu bāzes klasterī, ir iespējams atrisināt tādas problēmas, ka šablonbāzēta meklēšana un sadalīta datu kārtošana.

Lai pamatoti izvēlēties vispiemērotāko risinājumu *ADF* un *NoSQL* integrācijai, tika apskatītas vairākas eksistējošo ne-relāciju datu bāžu arhitektūras. Tika konstatētas vairākas atšķirības eksistējošo risinājumu arhitektūru dizainā, kas tālāk liecina par to, ka katrs no *NoSQL* risinājumiem ir izstrādāts specifiskam problēmu spektram. Risinājuma izvēle ir pilnā mērā atkarīga no klientu industrijas un industrijai specifiskiem pieprasījumiem. Ņemot vērā, ka Oracle *ADF* ietvars ir cieši integrēts ar Java valodu, tika izvēlēts *10gen* kompānijas risinājums – *MongoDB*. Datu bāzes izvēle ir pamatota ar vairākiem faktoriem – sintakses vienkāršība, attīstīts datu bāzes dzinis Java valodai un ierakstu glabāšana agregāta formā.

Lai iepazītos ar *MongoDB* ne-relāciju datu bāzi, autors piedalījās divos kursos, kurus rīkoja *MongoDB* izstrādātāju kompānija *10gen* – „M101J: *MongoDB* priekš izstrādātājiem" un „M102: *MongoDB* priekš datu bāzes administrētājiem". Abi kursi tika veiksmīgi pabeigti ar gala atzīmi 88% un 95% respektīvi. Autora kursu pabeigšanas sertifikāti ir apskatāmi 4. pielikumā. Minētie kursi autoram iedeva milzīgu teorētisko un praktisko zināšanu apjomu par *MongoDB* datu bāzi un ne-relāciju datu bāzēm kopumā.

Autoram ir divu gadu pieredze, strādājot ar *Oracle* produktiem, un pusotra gada pieredze, strādājot ar *Oracle ADF* ietvaru. Zināšanas, iegūtas darbā, palīdzēja izanalizēt un aprakstīt ietvara integrācijas iespējas ar ne-relāciju datu bāzēm. Pētījumu laikā tika konstatēts, ka viena no izvirzītajām integrācijas pieejam ir daudz efektīvāka un ērtāk realizējama. Programmatiskie skata

objekti piedāvā saskarni, kas ir pārāk grūti pielāgojama risinājumiem, kas neizmanto *Java* valodas *SQL* bibliotēku. Otrais risinājums, kas balstās uz tīmekļa pakalpojumu izmantošanas, piedāvāja lielāku datu sūtīšanas ērtību starp lietojumprogrammas klienta un modeļa daļām. Vēl vairāk iespēju piedāvā lietotņu moduļi – ADF ietvara kontroliera slāņa elementi. Šo iemeslu dēļ tika izlemts aprakstīt integrācijas iespējas, izmantojot tikai divas pēdējās pieejas.

Integrācijas risinājuma izstrādes procesā autors izvirzīja pieeju, kas atļauj *ADF* lietojumprogrammas izstrādātājam ērti manipulēt ar MongoDB datu bāzi. Ar Java Reflection bibliotēkas palīdzību, risinājums dod iespēju izmantot izstrādātāju definētus datu objektus datu rakstīšanai un lasīšanai no *Mongo* datu bāzes. Izstrādātājam ir jāpārliecinās tikai par to, ka datu objekti atbilst definētai saskarnei. Viena no galvenajām risinājuma priekšrocībām ir iespēja definēt datu bāzes shēmas lietojumprogrammas klientu pusē.

Lielu ietekmi uz izstrādāto modeļa slāņa arhitektūru atstāja 2012. gada decembrī notikuša intervija ar Džonu Hoffmanu. Autora piedāvātā pieeja veiksmīgi risina tādas problēmas, kā kolekcijas lauka nosaukuma maiņa un jauna datu lauka pievienošana kolekcijai. Izmantojot šo pieeju, vairākas izmaiņas datu modelī aizņem daudz mazāk laika un pieprasa daudz mazāk izmaiņu kodā, nekā izmantojot standartu *MongoDB Java* dzini.

Darba izstrādes procesā tika pierādīts un nodemonstrēts, ka, izmantojot autora izstrādāto integrācijas pieeju, var izmantot *ADF* ietvara vairākas datu attēlošanas iespējas. Tika parādīts, ka datu rakstīšana ir iespējama, izmantojot *JSF* grafiskās saskarnes elementus.

Lai turpmāk attīstītu izstrādāto risinājumu, autors plāno sagatavot bibliotēku, kas palīdzēs jebkuram Java izstrādātājam efektīvāk strādāt ar MongoDB. Viens no turpmākā darba virzieniem ir pielāgot risinājumu citām ne-relāciju datu bāzēm. Vel viens potenciāls attīstības virziens ir risinājuma saskarnes pielāgošana nestandarta vaicājumu apstrādei. Bet visvairāk autoru interesē risinājuma adaptācija *MongoDB* klasterim, un to veiktspējas salīdzināšana ar *SQL* klasteri.

Autoraprāt, darba mērķis ir sasniegts pilnībā un visi darba uzdevumi ir izpildīti. Autors ļoti augsti vērtē maģistra darba laikā iegūtās zināšanas. Tika paplašinātas ne tikai zināšanas par *NoSQL* datu bāzēm, bet arī par *ADF* ietvara iespējām.

IZMANTOTĀ LITERATŪRA UN AVOTI

1. Pramod J. Sadalage and Martin Fowler. NoSQL Distilled – A brief guide to the Emerging World of Polyglot Persistence. – Addison-Wesley. - USA, 2011. – 340 p.
2. Shaun O'Brien and Shay Shmeltzer. Oracle White Paper, Oracle Application Development Framework Overview. – Oracle. - USA, 2011. – 11 p.
3. Walter Egan. Skin Editor User's Guide for Oracle Application Development Framework. Oracle. - USA, 2011. – 138 p.
4. Eric Redmond, Jim R. Wilson. Seven Databases in Seven Weeks: A Guide to Modern Databases and the NoSQL Movement. Pragmatic Bookshelf. –USA, 2012. – 352 p.
5. Nineteenth ACM Symposium on Principles of Distributed Computing (PODC 2000) [tiešsaiste]. – [atsauce 16.04.2001.]. Pieejams: http://www.podc.org/html/podc2000
6. Program for PODC 2000 Computing [tiešsaiste]. – [atsauce 21.07.2000.]. Pieejams: http://www.podc.org/html/podc2000/program.html
7. Visual Guide to NoSQL Systems [tiešsaiste]. – [atsauce 15.03.2010.]. Pieejams: http://blog.nahurst.com/visual-guide-to-nosql-systems
8. Hibernate Project [tiešsaiste]. – [atsauce 07.01.2012.]. Pieejams: http://www.hibernate.org
9. Oracle Toplink [tiešsaiste]. – [atsauce 07.01.2012.]. Pieejams: http://www.oracle.com/technetwork/middleware/toplink/overview/index.html
10. Open Source MongoDB Gets Commercial Support [tiešsaiste] – [atsauce 01.04.2013.]. Pieejams: http://www.databasejournal.com/sqletc/article.php/3873406/Open-Source-MongoDB-Gets-Commercial-Support.htm
11. Introducing JSON [tiešsaite]. – [atsauce 01.04.2013.]. Pieejams: http://www.json.org/
12. MapReduce: Simplified Data Processing on Large Clusters [tiešsaiste]. – [atsauce 20.04.2013]. Pieejams: http://research.google.com/archive/mapreduce.html
13. „MongoDB priekš izstrādātājiem" kursa apraksts [tiešsaiste]. [atsauce 25.05.2013.]. Pieejams: https://education.10gen.com/courses/10gen/M101P/2013_June/about
14. „MongoDB priekš administratoriem" kursa apraksts [tiešsaiste]. [atsauce 25.05.2013.]. Pieejams: https://education.10gen.com/courses/10gen/M102/2013_July/about
15. Java Reflection bibliotēkas apraksts [tiešsaiste]. – [atsauce 26.05.2013.]. Pieejams: http://docs.oracle.com/javase/tutorial/reflect/
16. E. Intenbergs, „PHP ietvaru salīdzinājums tīmekļa vietņu izstrādē", maģistra darbs, LU Datorikas fakultāte, Latvijas Universitāte, Rīga, 2012

PIELIKUMI

1. pielikums

Autora izstrādāta datu pastāvīguma slānis

DatabaseHelper.java

```
package com.mongojava.main.proxy;

import com.mongodb.BasicDBObject;
import com.mongodb.DB;
import com.mongodb.DBCollection;
import com.mongodb.DBCursor;
import com.mongodb.DBObject;
import com.mongodb.Mongo;

import com.mongojava.main.proxy.util.DataObject;

import java.lang.reflect.Field;

import java.net.UnknownHostException;

import java.util.LinkedHashMap;
import java.util.LinkedList;
import java.util.List;
import java.util.Map;


public class DatabaseHelper {
    /**
     * Ieraksta sarakstu ar datu objektu instancem datu baze
     * @param documents
     */
    public static void insertList(List<DataObject> documents) {
        if (documents != null && !documents.isEmpty()) {
            Mongo mongo = getMongo();
            DB db = mongo.getDB(documents.get(0).getDBName());
            DBCollection collection =
db.getCollection(documents.get(0).getCollectionName());
            for (DataObject document : documents) {
                insert(collection, document);
            }
        }
    }

    /**
     * Ieraksta datu objekta instanci datu baze
     * @param document
     */
    public static void insert(DataObject document) {
        if (document != null) {
            Mongo mongo = getMongo();
            DB db = mongo.getDB(document.getDBName());
            DBCollection collection = db.getCollection(document.getCollectionName());
            insert(collection, document);
```

```
        }
    }

    /**
     * Ieraksta datu objekta instanci datu baze
     * @param document
     */
    private static void insert(DBCollection collection, DataObject document) {
        BasicDBObject dbObject = new BasicDBObject();
        Map<String, Object> map;
        map = twoListsToMap(document.getCurrentDBFieldNames(),
document.getValueList());
        Object value;
        for (Map.Entry<String, Object> entry : map.entrySet()) {
            value = entry.getValue();
            if (value != null) {
                dbObject.put(entry.getKey(), value);
            }
        }
        collection.insert(dbObject);
    }

    /**
     * Atlasa un atgriez visus dokumentus no datu objektam piesaistitas kolekcijas
     * datu objektu forma
     * @param clazz
     * @return
     */
    public static List<DataObject> readListBothSchemas(Class<? extends DataObject>
clazz) {
        DataObject document = getInstance(clazz);
        return readListBothSchemas(clazz, bothSchemasSearchQuery(document));
    }

    /**
     * Atlasa un atgriez visus dokumentus no datu objektam piesaistitas kolekcijas
     * datu objektu forma
     * @param clazz
     * @param searchQuery
     * @return
     */
    private static List<DataObject> readListBothSchemas(Class<? extends DataObject>
clazz, BasicDBObject searchQuery) {
        List<DataObject> resultList = new LinkedList<DataObject>();

        DataObject document = null;
        document = getDocument(clazz);

        Mongo mongo = getMongo();
        DB db = mongo.getDB(document.getDBName());
        DBCollection collection = db.getCollection(document.getCollectionName());
        DBCursor cursor = collection.find(searchQuery);
        while (cursor.hasNext()) {
            DBObject cursorEntry = cursor.next();
            try {
                DataObject dataObject = clazz.newInstance();
                List<String> currentDBFieldNames = document.getCurrentDBFieldNames();
```

```
                List<String> oldDOFieldNames = document.getOldDOFieldNames();
                List<String> oldDBFieldNames = document.getOldDBFieldNames();
                for (int i = 0; i < document.getOldDBFieldNames().size(); i++) {
                    Field field = clazz.getDeclaredField(oldDOFieldNames.get(i));
                    Object fieldValue;
                    fieldValue = cursorEntry.get(oldDBFieldNames.get(i));
                    if (fieldValue == null) {
                        fieldValue = cursorEntry.get(currentDBFieldNames.get(i));
                    }
                    field.set(dataObject, fieldValue);
                }

                resultList.add(dataObject);
            } catch (IllegalAccessException iae) {
                iae.printStackTrace();
            } catch (InstantiationException ie) {
                ie.printStackTrace();
            } catch (SecurityException se) {
                se.printStackTrace();
            } catch (NoSuchFieldException nsfe) {
                nsfe.printStackTrace();
            }
        }
        return resultList;
    // }

    /**
     * Atlasa un atgriez tikai tadus dokumentus, kur satur vismaz vienu lauku
     * ar veco nosaukumu
     * @param clazz
     * @return
     */
    public static List<DataObject> readListOldSchema(Class<? extends DataObject>
clazz) {
        DataObject document = getInstance(clazz);
        return readListOldSchema(clazz, oldSchemaSearchQuery(document));
    }

    /**
     * Atlasa un atgriez tikai tadus dokumentus, kur satur vismaz vienu lauku
     * ar veco nosaukumu
     * @param clazz
     * @param searchQuery
     * @return
     */
    private static List<DataObject> readListOldSchema(Class<? extends DataObject>
clazz, BasicDBObject searchQuery) {
        List<DataObject> resultList = new LinkedList<DataObject>();

        DataObject document = null;
        try {
            document = clazz.newInstance();
        } catch (IllegalAccessException iae) {
            iae.printStackTrace();
        } catch (InstantiationException ie) {
            ie.printStackTrace();
        }
```

```
    Mongo mongo = getMongo();
    DB db = mongo.getDB(document.getDBName());
    DBCollection collection = db.getCollection(document.getCollectionName());
    DBCursor cursor = collection.find(searchQuery);
    while (cursor.hasNext()) {
        DBObject cursorEntry = cursor.next();
        try {
            DataObject dataObject = clazz.newInstance();
            List<String> oldDOFieldNames = document.getOldDOFieldNames();
            List<String> oldDBFieldNames = document.getOldDBFieldNames();
            for (int i = 0; i < document.getOldDBFieldNames().size(); i++) {
                Field field = clazz.getDeclaredField(oldDOFieldNames.get(i));
                field.set(dataObject, cursorEntry.get(oldDBFieldNames.get(i)));
            }

            resultList.add(dataObject);
        } catch (IllegalAccessException iae) {
            iae.printStackTrace();
        } catch (InstantiationException ie) {
            ie.printStackTrace();
        } catch (SecurityException se) {
            se.printStackTrace();
        } catch (NoSuchFieldException nsfe) {
            nsfe.printStackTrace();
        }
    }
    return resultList;
}

/**
 * Genere pieprasijumu, kas atgriez tikai tadus dokumentus, kas satur kaut vienu
 * veco datu bazes lauka nosaukumu
 * @param document
 * @return
 */
private static BasicDBObject oldSchemaSearchQuery(DataObject document) {
    BasicDBObject searchQuery;
    List<BasicDBObject> criteria = new LinkedList<BasicDBObject>();
    BasicDBObject criterion = null;
    for (String fieldName : document.getOldDBFieldNames()) {
        criterion = new BasicDBObject();
        criterion.append(fieldName, new BasicDBObject("$exists", true));
        criteria.add(criterion);
    }
    searchQuery = new BasicDBObject("$or", criteria);
    return searchQuery;
}

private static BasicDBObject bothSchemasSearchQuery(DataObject document) {
    return new BasicDBObject();
}

/**
 * Konstrue datu objekta instanci no klases references
 * @param clazz
 * @return
```

```
     */
    private static DataObject getInstance(Class<? extends DataObject> clazz) {
        DataObject document = null;
        try {
            document = clazz.newInstance();
        } catch (IllegalAccessException iae) {
            iae.printStackTrace();
        } catch (InstantiationException ie) {
            ie.printStackTrace();
        }
        return document;
    }

    /**
     * Metode atrod visas datu objekta kolekcijas dokumentus ar veciem lauku
nosaukumiem
     * un pardeve uz jauniem nosaukumiem
     * @param clazz
     */
    public static void migrateOldToNewSchema(Class<? extends DataObject> clazz) {
        DataObject document = getInstance(clazz);
        List<String> oldDBFieldNames = document.getOldDBFieldNames();
        List<String> currentDBFieldNames = document.getCurrentDBFieldNames();
        // TODO: Throw exception if size is different

        Mongo mongo = getMongo();
        DB db = mongo.getDB(document.getDBName());
        DBCollection collection = db.getCollection(document.getCollectionName());
        for (int i = 0; i < oldDBFieldNames.size(); i++) {
            collection.update(new BasicDBObject(), new BasicDBObject("$rename", new
BasicDBObject(oldDBFieldNames.get(i), currentDBFieldNames.get(i))), false, true);
//            criterion.append("$rename", );
        }
    }

    private static Map<String, Object> twoListsToMap(List<String> names, List<Object>
values) {
        Map<String, Object> map = new LinkedHashMap<String, Object>();
        for (int i = 0; i < names.size(); i++) {
            map.put(names.get(i), values.get(i));
        }
        return map;
    }

    private static DataObject getDocument(Class<? extends DataObject> clazz) {
        DataObject document;
        try {
            document = clazz.newInstance();
        } catch (IllegalAccessException iae) {
            iae.printStackTrace();
        } catch (InstantiationException ie) {
            ie.printStackTrace();
        }
        return document;
    }

    private static Mongo getMongo() {
```

```
        Mongo mongo = null;
        try {
            mongo = new Mongo("localhost", 27017);
        } catch (UnknownHostException uhe) {
            uhe.printStackTrace();
        }
        return mongo;
    }
}
```

DataObject.java abstrakta klase

```
package com.mongojava.main.proxy.util;

import java.util.List;

public abstract class DataObject {

    public abstract String getDBName();

    public abstract String getCollectionName();

    public abstract List<Object> getValueList();

    public abstract List<String> getDOFieldNames();

    public abstract List<String> getCurrentDBFieldNames();

    public abstract List<String> getOldDBFieldNames();
}
```

PersonDO.java klase ar iekļauto Shemas Enum klasi

```
package com.mongojava.main.proxy.util;


import java.io.Serializable;

import java.util.LinkedList;
import java.util.List;


public class PersonDO extends DataObject implements Serializable {

    @SuppressWarnings("compatibility:5393877295368927156")
    private static final long serialVersionUID = 1L;
    public String name;
    public Integer age;
    public String hobby;

    /**
     * Apraksta kolekcijas tekojošo shēmu
     */
    public enum Schema implements Serializable {
```

```
        NAME("name", "n", "Full name"),
        AGE("age", "age", "a"),
        HOBBY("hobby", "hobby", "h");

        protected static final String DB_NAME = "performance";
        protected static final String COLLECTION_NAME = "plainInsert";

        private String currentDOFieldName;
        private String currentDBFieldName;
        private String oldDBFieldName;

        Schema(String currentDOFieldName, String currentDBFieldName, String
oldDBFieldName) {
            this.currentDOFieldName = currentDOFieldName;
            this.currentDBFieldName = currentDBFieldName;
            this.oldDBFieldName = oldDBFieldName;
        }

        public static List<String> getCurrentDOFieldNames() {
            List<String> fieldNameList = new LinkedList<String>();
            for (Schema person :
com.mongojava.main.proxy.util.PersonDO.Schema.values()) {
                fieldNameList.add(person.getCurrentDOFieldName());
            }
            return fieldNameList;
        }

        public static List<String> getCurrentFieldNames() {
            List<String> fieldNameList = new LinkedList<String>();
            for (Schema person :
com.mongojava.main.proxy.util.PersonDO.Schema.values()) {
                fieldNameList.add(person.getCurrentDBFieldName());
            }
            return fieldNameList;
        }

        public static List<String> getOldDBFieldNames() {
            List<String> fieldNameList = new LinkedList<String>();
            for (Schema person :
com.mongojava.main.proxy.util.PersonDO.Schema.values()) {
                fieldNameList.add(person.getOldDBFieldName());
            }
            return fieldNameList;
        }

        public String getCurrentDOFieldName() {
            return currentDOFieldName;
        }

        public String getCurrentDBFieldName() {
            return currentDBFieldName;
        }

        public String getOldDBFieldName() {
            return oldDBFieldName;
        }
```

```
        public static String getDBName() {
            return DB_NAME;
        }

        public static String getCollectionName() {
            return COLLECTION_NAME;
        }
    }

    public static long getSerialVersionUID() {
        return serialVersionUID;
    }

    public List<Object> getValueList() {
        List<Object> list = new LinkedList<Object>();
        list.add(name);
        list.add(age);
        list.add(hobby);
        return list;
    }

    public List<String> getCurrentDOFieldNames() {
        return
com.mongojava.main.proxy.util.PersonDO.Schema.getCurrentDOFieldNames();
    }

    public List<String> getCurrentDBFieldNames() {
        return com.mongojava.main.proxy.util.PersonDO.Schema.getCurrentFieldNames();
    }

    public List<String> getOldDBFieldNames() {
        return com.mongojava.main.proxy.util.PersonDO.Schema.getOldDBFieldNames();
    }

    public String getDBName() {
        return com.mongojava.main.proxy.util.PersonDO.Schema.getDBName();
    }

    public String getCollectionName() {
        return com.mongojava.main.proxy.util.PersonDO.Schema.getCollectionName();
    }

    @Override
    public String toString() {
        return "name: " + name + "; age: " + age + "; hobby: " + hobby;
    }

    // Piekļuvējmetodes
    public void setName(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }

    public void setAge(Integer age) {
```

```
        this.age = age;
    }

    public Integer getAge() {
        return age;
    }

    public void setHobby(String hobby) {
        this.hobby = hobby;
    }

    public String getHobby() {
        return hobby;
    }
}
```

Tīmekļa pakalpojumu WSDL dokumenta piemērs

```
<?xml version='1.0' encoding='utf-8'?>
<!-- Published by JAX-WS RI at http://jax-ws.dev.java.net. RI's version is
Oracle JAX-WS 2.1.5. -->
<!-- Generated by JAX-WS RI at http://jax-ws.dev.java.net. RI's version is
Oracle JAX-WS 2.1.5. -->
<definitions xmlns:soap12="http://schemas.xmlsoap.org/wsdl/soap12/"
xmlns:tns="http://tempuri.org/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns="http://schemas.xmlsoap.org/wsdl/"
targetNamespace="http://tempuri.org/" name="MapService">
  <types>
    <xsd:schema>
      <xsd:import namespace="http://tempuri.org/"
      schemaLocation="http://localhost:7101/JavaMongoWeb-ViewController-
context-root/MapService?xsd=1" />
    </xsd:schema>
  </types>
  <message name="insert">
    <part name="parameters" element="tns:insert" />
  </message>
  <message name="getFieldNames">
    <part name="parameters" element="tns:getFieldNames" />
  </message>
  <message name="getFieldNamesResponse">
    <part name="parameters" element="tns:getFieldNamesResponse" />
  </message>
  <message name="test">
    <part name="parameters" element="tns:test" />
  </message>
  <message name="testResponse">
    <part name="parameters" element="tns:testResponse" />
  </message>
  <message name="printArray">
    <part name="parameters" element="tns:printArray" />
  </message>
  <message name="printList">
    <part name="parameters" element="tns:printList" />
  </message>
  <message name="insertTwoLists">
    <part name="parameters" element="tns:insertTwoLists" />
  </message>
  <message name="insertDOList">
    <part name="parameters" element="tns:insertDOList" />
  </message>
  <message name="insertList">
    <part name="parameters" element="tns:insertList" />
  </message>
  <message name="insertWithSchema">
```

```
  <part name="parameters" element="tns:insertWithSchema" />
</message>
<portType name="MapService">
  <operation name="insert">
    <input message="tns:insert" />
  </operation>
  <operation name="getFieldNames">
    <input message="tns:getFieldNames" />
    <output message="tns:getFieldNamesResponse" />
  </operation>
  <operation name="test">
    <input message="tns:test" />
    <output message="tns:testResponse" />
  </operation>
  <operation name="printArray">
    <input message="tns:printArray" />
  </operation>
  <operation name="printList">
    <input message="tns:printList" />
  </operation>
  <operation name="insertTwoLists">
    <input message="tns:insertTwoLists" />
  </operation>
  <operation name="insertDOList">
    <input message="tns:insertDOList" />
  </operation>
  <operation name="insertList">
    <input message="tns:insertList" />
  </operation>
  <operation name="insertWithSchema">
    <input message="tns:insertWithSchema" />
  </operation>
</portType>
<binding name="MapServiceSoap12HttpPortBinding"
type="tns:MapService">
  <soap12:binding transport="http://www.w3.org/2003/05/soap/bindings/HTTP/"
  style="document" />
  <operation name="insert">
    <soap12:operation soapAction="" />
    <input>
      <soap12:body use="literal" />
    </input>
  </operation>
  <operation name="getFieldNames">
    <soap12:operation soapAction="" />
    <input>
      <soap12:body use="literal" />
    </input>
    <output>
      <soap12:body use="literal" />
    </output>
  </operation>
```

```
    <operation name="test">
      <soap12:operation soapAction="" />
      <input>
        <soap12:body use="literal" />
      </input>
      <output>
        <soap12:body use="literal" />
      </output>
    </operation>
    <operation name="printArray">
      <soap12:operation soapAction="" />
      <input>
        <soap12:body use="literal" />
      </input>
    </operation>
    <operation name="printList">
      <soap12:operation soapAction="" />
      <input>
        <soap12:body use="literal" />
      </input>
    </operation>
    <operation name="insertTwoLists">
      <soap12:operation soapAction="" />
      <input>
        <soap12:body use="literal" />
      </input>
    </operation>
    <operation name="insertDOList">
      <soap12:operation soapAction="" />
      <input>
        <soap12:body use="literal" />
      </input>
    </operation>
    <operation name="insertList">
      <soap12:operation soapAction="" />
      <input>
        <soap12:body use="literal" />
      </input>
    </operation>
    <operation name="insertWithSchema">
      <soap12:operation soapAction="" />
      <input>
        <soap12:body use="literal" />
      </input>
    </operation>
  </binding>
  <service name="MapService">
    <port name="MapServiceSoap12HttpPort"
    binding="tns:MapServiceSoap12HttpPortBinding">
      <soap12:address location="http://localhost:7101/JavaMongoWeb-
ViewController-context-root/MapService" />
    </port>
```

```
  </service>
</definitions>
```

Tīmekļa pakalpojuma un starpnieka sadarbība 1. veids

```
public abstract class DataObject {

    public abstract String getDBName();

    public abstract String getCollectionName();

    public abstract List<Object> getValueList();

    public abstract List<String> getFieldNames();
}

public class PersonDO extends DataObject implements Serializable {

    @SuppressWarnings("compatibility:-4872898270401628590")
    private static final long serialVersionUID = 1L;
    private String name;
    private Integer age;
    private String hobby;

    /**
     * Apraksta kolekcijas tekojošo shēmu
     */
    public enum Schema implements Serializable {
        NAME("name"),
        AGE("age"),
        HOBBY("hobby");

        protected static final String DB_NAME = "performance";
        protected static final String COLLECTION_NAME = "plainInsert";

        private String fieldName;

        Schema(String fieldName) {
            this.fieldName = fieldName;
        }
```

```
    public static List<String> getFieldNames() {
        List<String> fieldNameList = new LinkedList<String>();
        for (Schema person : PersonDO.Schema.values()) {
            fieldNameList.add(person.getFieldName());
        }
        return fieldNameList;
    }

    public String getFieldName() {
        return fieldName;
    }

    public static String getDBName() {
        return DB_NAME;
    }

    public static String getCollectionName() {
        return COLLECTION_NAME;
    }
}

// Intefeisa metodes
public String getDBName() {
    return Schema.getDBName();
}

public String getCollectionName() {
    return Schema.getCollectionName();
}

public List<Object> getValueList() {
    List<Object> list = new LinkedList<Object>();
    list.add(name);
    list.add(age);
    list.add(hobby);
    return list;
```

```
    }

    public List<String> getFieldNames() {
        return Schema.getFieldNames();
    }

    // Piekļuvējmetodes
    public void setName(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }

    public void setAge(Integer age) {
        this.age = age;
    }

    public Integer getAge() {
        return age;
    }

    public void setHobby(String hobby) {
        this.hobby = hobby;
    }

    public String getHobby() {
        return hobby;
    }
}
```

4. pielikums

"10 gen" izsniegti kursu veiksmīgas pabeigšanas sertifikāti

10gen | The MongoDB Company

CERTIFICATE
Dec. 24th, 2012

This is to certify that

Andrejs Daško

successfully completed

M102: MongoDB for DBAs

a course of study offered by 10gen, The MongoDB Company

Dwight Merriman
Chief Executive Officer
10gen, Inc.

Andrew Erlichson
Vice President, Education
10gen, Inc.

Authenticity of this certificate can be verified at https://education.10gen.com/downloads/certificates/a8bdbf7fcf524564b3920c9655484596/Certificate.pdf

10gen | The MongoDB Company

CERTIFICATE
Dec. 24th, 2012

This is to certify that

Andrejs Daško

successfully completed

M101: MongoDB for Developers

a course of study offered by 10gen, The MongoDB Company

Dwight Merriman
Chief Executive Officer
10gen, Inc.

Andrew Erlichson
Vice President, Education
10gen, Inc.

Authenticity of this certificate can be verified at https://education.10gen.com/downloads/certificates/a26641a185484856bc809aa646eba77a/Certificate.pdf