Univerza v Ljubljani
Fakulteta za računalništvo in informatiko ter
fakulteta za matematiko in fiziko

Andrej Slapnik

Uporaba objektno-relacijskega preslikovanja nad Apache Cassandra podatkovno bazo

Employment of object-relational mapping with Apache Cassandra database

diplomsko delo

univerzitetni interdisciplinarni študijski program računalništva in matematike

Mentor: doc. dr. Dejan Lavbič

Ljubljana 2012

Besedilo je oblikovano z urejevalnikom besedil LaTeX.

Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

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

Št. naloge: 00040/2012
Datum: 03.09.2012

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

Kandidat: **ANDREJ SLAPNIK**

Naslov: **UPORABA OBJEKTNO-RELACIJSKEGA PRESLIKOVANJA NAD APACHE CASSANDRA PODATKOVNO BAZO**

EMPLOYMENT OF OBJECT-RELATIONAL MAPPING WITH APACHE CASSANDARA DATABASE

Vrsta naloge: Diplomsko delo univerzitetnega študija

Tematika naloge:

Količina, frekvenca nastajanja in predvsem potreba po obdelavi podatkov na svetovnem spletu raste z izjemno hitrostjo. Z uporabo sistemov za upravljanje relacijskih podatkovnih baz pri obvladovanju takšne kompleksnosti pogosto naletimo na težave. Ena izmed možnosti je uporaba rešitev iz družine NoSQL podatkovnih baz. Raziščite prednosti in slabosti uporabe ORM mehanizma nad NoSQL podatkovno bazo Cassandra. Zamislite si konkreten problem, ki ga poskušajte rešiti s predlaganim pristopom ter ga primerjajte z obstoječimi pristopi v svetu relacijskih podatkovnih baz. V primerjavo vključite tudi test zmogljivost.

Mentor:

doc. dr. Dejan Lavbič

Fakulteta za računalništvo in informatiko • UNIVERZA V LJUBLJANI • 2

Dekan Fakultete za računalništvo in informatiko:

prof. dr. Nikolaj Zimic

Dekan Fakultete za matematiko in fiziko:

akad. prof. dr. Franc Forstnerič

UNIVERZA V LJUBLJANI • Fakulteta za matematiko in fiziko • 1

Izjava o avtorstvu diplomskega dela

Spodaj podpisani Andrej Slapnik, z vpisno številko **63060240**, sem avtor diplomskega dela z naslovom:

Uporaba objektno-relacijskega preslikovanja nad Apache Cassandra podatkovno bazo

S svojim podpisom zagotavljam, da:

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

V Ljubljani, dne 25. novembra 2012 Podpis avtorja:

Zahvalil bi se vodji razvoja v podjetju Zemanta mag. Dušanu Omerčeviću za korektno in strokovno vodenje med nastajanjem te diplomske naloge.

Kazalo

Povzetek

Abstract

1 Uvod **1**

2 Osnovni pojmi **3**

2.1 Uvod 3

2.2 Porazdeljene zgoščevalne tabele 3

2.3 Apache Cassandra 9

2.4 Object-relational mapper (ORM) 15

2.5 Normalizacija in denormalizacija 16

3 ORM nad Apache Cassandra PSUPB-jem **19**

3.1 Prednosti in slabosti 21

3.2 Obstoječi odjemalci za Apache Cassandra 22

4 Implementacija osnovnega ORM nad Apache Cassandra **25**

4.1 Uvod 25

4.2 Osnovne potrebe 26

4.3 Implementacija 27

5 Reševanje realnega problema - iskanje najbližjega prostega termina **31**

5.1 Opis problema 31

5.2 Izračun gostote poslovalnic 32
5.3 Reševanje problema z relacijskim SUPB-jem 34
5.4 Reševanje problema z Apache Cassandra PSUPB-jem 36
5.5 Uporaba ORM-ja v praksi 38
5.6 Testiranje zmogljivosti SUPB-jev 39

6 Razprava **43**
6.1 Uporaba SUPB-ja 43
6.2 Načrtovanje sheme podatkovne baze 44
6.3 Implementacija ORM-ja 45

Slike

2.1 Zgoščevalna tabela 4
2.2 Konsistentno zgoščevanje 12

4.1 Rezredni diagram ORM-ja in njegove uporabe 29

5.1 Spletni vmesnik aplikacije. Rdeči označevalnik prikazuje našo lokacijo, modri označevalniki pa prikazujejo lokacije poslovalnic. 38
5.2 Rezultati obremenitve MySQL podatkovne baze 40
5.3 Rezultati obremenitve PostgreSQL podatkovne baze 40
5.4 Rezultati obremenitve Apache Cassandra podatkovne baze . . 40

Povzetek

V diplomski nalogi sem preveril kako na reševanje splošnega realnega problema vpliva uporaba nerelacijskega sistema za upravljanje podatkovne baze, in ali uporaba ORM-ja nad nerelacijskim SUPB-jem pripomore k učinkovitejšem razvoju aplikacije.

V prvem delu sem predstavil nerelacijski sistem za upravljanje s podatkovno bazo Apache Cassandra ter opisal porazdeljene zgoščevalne tabele, ki so osrednja podatkovna struktura v ozadju tega sistema. Nadaljeval sem z opisom ideje o dodatnem aplikacijskem nivoju v obliki ORM-ja in pregledal prednosti ter slabosti ob uporabi z Apache Cassandro.

V osrednjem delu sem opisal idejo in razvoj lastnega ORM-ja kot dodatni nivo med aplikacijo napisano v programskem jeziku PHP in nerelacijsko podatkovno bazo. Reševanje realnega problema in izkušnje, ki sem jih pridobil med načrtovanjem in razvojem sem opisal v petem poglavju.

Diplomsko nalogo sem zaključil s predstavitvijo ugotovitev in vodil, ki naj bi se jih razvijalec držal, ko se odloča katere tehnologije bo uporabil pri reševanju svojega problema.

Ključne besede: Podatkovna baza, relacija, Apache Cassandra, ORM, porazdeljene zgoščevalne tabele

Abstract

In this thesis I have checked what is the impact of using non-relational database managment system to solving one of the general problem. I have also checked how using ORM layer with non-relational DBMS contribute to efficent development.

In the first part I have presented non-relational database managment system Apacahe Casasndra and descibed what distributed hash tables are. I have continued with description of the idea of additional applicative layer ORM and checked pros and cons of using it with Apache Cassandra.

I the main part I have descibed the idea and development of my own ORM between PHP application and non-relational database. In chapter five I have descibed solving general problem and experiences I have got at designing and developing application.

In the last part I have presented my findings and written few guides to other developers which can help at making decisions which technologies should they use at solving their problem.

Keywords: Database, relation, Apache Cassandra, ORM, distributed hash tables

Poglavje 1

Uvod

Načrtovalec aplikacije in njen razvojnik se vsakodnevno srečujeta z vprašanji kaj naj uporabita kot optimalno rešitev svojega problema pri razvoju programske opreme. Razvoj mora potekati hitro, razvijalec pa naj bi se osredotočal predvsem na reševanje problemov in ne le na kodiranje. Odgovori na vprašanja kot so: ali naj uporabim relacijski sistem za upravljanje podatkovne baze ali nerelacijski, ali naj pri razvoju aplikacije uporabim ORM ali ne, ali naj se odločim za prosto dostopno odprtokodno rešitev ali naj napišem svojo, na razvoj aplikacije odločilno vplivajo.

V tem diplomskem delu se bom postavil v kožo načrtovalca aplikacije, ki mora svojemu naročniku predstaviti rešitev za nek standarden problem. Rešitev mora delovati dovolj hitro, v mislih moramo imeti možnosti razširitve infrastrukture, razvojnikom pa moramo olajšati njihovo delo s pravilnim izborom tehnologij. Osredotočil se bom predvsem na način shranjevanja podatkov in način kako razvojnik v aplikaciji dostopa do podatkov. Predstavil bom nerelacijski odprtokodni in porazdeljeni sistem za upravljanje podatkovne baze (PSUPB) Apache Cassasndra in tehnologije na katerih temelji. PSUPB Apache Cassandra bom primerjal s klasičnima relacijskima sistemoma za upravljanje s podatkovnimi bazami PostgreSQL in MySQL. Ob tem bom preveril ali lahko s pomočjo ORM-ja za nerelacijsko podatkovno bazo pohitrimo razvoj aplikacije, oziroma ali nam ta v nerelacijskem svetu

povzroča le preglavice. Za zaključek bom predstavil tudi rešitve realnega problema z izbranimi tehnologijami.

Poglavje 2

Osnovni pojmi

2.1 Uvod

Za uvod bom predstavil osnovne tehnologije s katerimi sem se tekom priprave diplomske naloge ukvarjal. Pričel bom z opisom zgoščevalnih tabel in njihovih porazdeljenih različic. Porazdeljene zgoščevalne tabele so osnova za PSUPB Apache Cassandra, ki ga bom opisal v nadaljevanju. Bralec bo lahko več izvedel o osnovni ideji ter zgodovini, podatkovnem modelu in infrastrukturi. Nadaljeval bom z idejo in razlogih za uporabo ORM. Opisal bom kaj ORM sploh je, katere so njegove prednosti in katere slabosti. Zaključil bom s kratko predstavitvijo procesov normalizacije in denormalizacije. Opisal bom kje omenjena procesa uporabljamo in katere so njune prednosti.

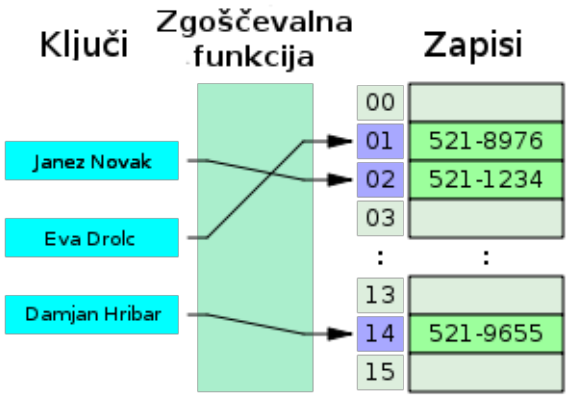
2.2 Porazdeljene zgoščevalne tabele

2.2.1 Zgoščevalne tabele

Zgoščevalna tabela je podatkovna struktura, ki shranjuje elemente v obliki parov [ključ,vrednost]. Najpomembnejša lastnost zgoščevalne tabele je dostopni čas do željenega elementa, ki se mora zgoditi v konstantnem času glede na število elementov. Posledično se branje in brisanje elementa lahko

izvedeta v konstantnem času, hitrost zapisovanja elementa pa je odvisna od gostote same zgoščevalne tablele. Najpomembnejši problem, ki ga moramo rešiti pri implementaciji zgoščevalne tabele, je problem sovpadanja lokacij oz. zgoščenih ključev.

Za implementacijo zgoščevalne tabele največkrat uporabimo polje. Če polje lahko shrani n vrednosti, se element naprimer shrani na mesto ključ mod n. Dobra implementacija zgoščevalne tabele zahteva tudi dobro definirano zgoščevalno funkcijo. Funkcija mora generirati svoje vrednosti čimbolj enakomerno porazdeljeno in se čimbolj približati injektivnosti. Ker v relanosti ne poznamo števila vseh parov vrednosti, ki jih bomo hoteli shraniti, je idealno funkcijo nemogoče izbrati. Pri idelani zgoščevalni funkciji ne bi prihajalo do trkov, najslabši čas branja podatkov pa bi se izvedel v konstantnem času. Pogoj za idealno zgoščevalno funkcijo je tudi idelana velikost polja za shranjevanje. Ta bi bila natanko toliko, kolikor elementov bi v polje hoteli shraniti, tega podatka pa kot smo že omenili predhodno, žal nimamo.



Slika 2.1: Zgoščevalna tabela

Zgoščevalna tabela mora zaradi svojega dizajna in velikih časovnih prihrankov pri dostopnem času do elementa žal tudi kaj žrtvovati. Kot smo že omenili moramo velikost zgoščevalne tabele določiti vnaprej. To ima za posledico dve težavi: Ali je tabela prevelika in nam zato po nepotrebnem zaseda prostor v

spominu, ali pa je premajhna, kar ima za posledico povečano število trkov. V najslabšem primeru nam celo zmanjka prostora za shranjevanje.

Vkolikor se zgoščevalna tabela zaradi prevelike gostote neha obnašati optimalno, moramo zgraditi novo – večjo. Ta proces imenujemo ponovno zgoščevanje, ki pa ponavadi terja veliko časa. Ker si tega v nekaterih tipih aplikacij enostavno ne moremo privoščiti, to smatramo za drugo slabost.

Tretja slabost je neprilagodljivost zgoščevalne funkcije. Ker v realnosti ponavadi ne poznamo porazdeljenosti elementov, se nam v najslabšem primeru zgoščevalna tabela lahko izrodi v seznam.

Kot zadnjo večjo slabost bi poudaril neoptimalnost oz. neučinkovitost nekaterih standardnih operacij nad podatki shranjenimi v tabeli. Iskanje največjega ali najmanjšega elementa, razvrščanje elementov in ostale operacije, ki temeljijo na primerjanju zaradi samega dizajna ni mogoče učinkovito izvesti.

2.2.2 Porazdeljenost

Porazdeljena zgoščevalna tabela se od navadne zgoščevalne tabele ne razlikuje veliko. Razlika med njima je predvsem v lokaciji shranjevanja elementov. Če se elementi pri navadni zgoščevalni tabeli shranjujejo na eni lokaciji, naprimer v enem polju, se elementi pri porazdeljeni zgoščevalni tabeli shranjujejo na več lokacijah, ki jih imenujemo vozlišča. Ta vozlišča nato tvorijo mrežo, največkrat v obliki obroča (ring). Zaradi različnih lokacij shranjevanja elementov moramo prilagoditi implementacije osnovnih operacij nad strukturo.

Med osnovne operacije štejemo dodajanje, branje in brisanje elementa. Za vse te operacije mora veljati, da se jih mora dati izvesti za katerikoli element na kateremkoli vozlišču ter iz kateregakoli vozlišča. Pravtako se morajo operacije izvesti podobno hitro kot na navadi zgoščevalni tabeli. Zavedati se moramo, da moramo poleg operacije same, izvesti še iskanje vozlišča v katerem se element nahaja. Kakšen je čas iskanja, je odvisno od strukture mreže, ki ji rečemo topologija. Tipičen iskalni čas vozlišča v mreži je

$\mathcal{O}(logn)$ kjer je n število vozlišč, možne pa so tudi topologije, kjer je iskalni čas vozlišča konstanten glede na število vozlišč. Tak iskalni čas nam omogoča gostota povezav med vozlošči.

Stopnja vozlišča	Dolžina poti
$\mathcal{O}(1)$	$\mathcal{O}(n)$
$\mathcal{O}(logn)$	$\mathcal{O}(logn)$
$\mathcal{O}(\sqrt{n})$	$\mathcal{O}(1)$

Tabela 2.1: Dolžina poti glede na stopnjo vozlišč

Standardne topologije imajo obliko drevesa in jim rečemo drevesne. Obstajajo pa tudi topologije, ki jim rečemo hiper-kocke. V hiper-kocki vsa vozlišča oštevilčimo, sosednost pa predstavljajo razlike v bitih teh števil.

Vzdrževanje ključev v tabeli je porazdeljeno med vozlišča, potrebno pa je zagotoviti, da lahko vsako vozlišče za vsak ključ ugotovi v katerem vozlišču se nahaja. Več o lastnostih porazdeljenih zgoščevalnih tabel si bomo ogledali v nadaljevanju.

2.2.3 Zgodovina

Na idejo o porazdeljenih zgoščevalnih tabelah so najprej prišli razvijalci in raziskovalci peer to peer (P2P) omrežij, ki so želeli povečati zmogljivosti svojih omrežij. Največji problem sta jim predstavljala prepustnost omrežja in velikost oziroma majhnost prostora za shranjevanje podatkov. Prišli so na idejo, da bi lahko vsako vozlišče v omrežju (vsak odjemalec) namenil določeno količino svojih virov v skupno dobro. Tako so se v devetdesetih letih razvila prva P2P omrežja kot so Napster, Freenet in Gnutella. Seveda so se omrežja v arhitekturi in podrobnostih razlikovala med seboj, delila pa so si isto idejo.

Omrežje Gnutella je iskalo elemente tako, da je o elementu povprašalo vsa vozlišča v omrežju (flooding), omrežje Kazaa je uporabljalo t.i. supervozlišča, ki so shranjevala podatke o lokacijah elementov, omrežje Napster

pa je imelo centralni strežnik, kjer so se nahajali vsi podatki o lokacijah elementov. Kot vidimo, je imelo vsako omrežje svojo veliko slabost. Omrežje Gnutella je po nepotrebnem zasedalo omrežje, omrežje Napster pa je imelo ozko grlo in bilo lahka tarča napadov prav zaradi centralnega strežnika. Raziskovalci in znanstveniki z vsega sveta so se posledično začeli truditi, da bi našli omrežje, ki ne bi imelo teh očitnih slabosti, bi bilo hitro, povsem porazdeljeno in bilo lahko razširljivo. Omrežje bi moralo maksimizirati uporabo shranjevalnega prostora ter optimizirati pretok podatkov po omrežju ne glede na to ali je v mreži sto, tisoč ali več milijonov vozlišč. Omrežje bi moralo biti robustno, odporno na odpovedi vozlišč, sama vozlišča pa bi lahko enostavno dodajali. Glavno vprašanje, ki se je v tistem času porajalo v glavah raziskovalcev je bilo, ali tako omrežje sploh obstaja in ali smo ga zmožni implementirati.

Leta 2001 se je začelo vzporedno izvajati več večjih pomembnih projektov, ki so zaznamovali razvoj porazdeljenih zgoščevalnih tabel. V akademskih krogih so bili to CAN (Content addressable network) [7], Chord [8], Pastry [9] in Tapestry, med internetno populacijo pa je najbolj znan projekt BitTorrent. Vsi ti projekti so začeli dokazovati da omrežja, ki zadoščajo našim željam in omejitvam, obstajajo.

2.2.4 Lastnosti porazdeljenih zgoščevalnih tabel

Nekaj osnovnih lastnosti, ki jih mora imeti porazdeljena zgoščevalna tabela, smo opisali že v prejšnjem poglavju. To so: neodvisnost vozlišč, decentralizirano omrežje, možnost hitre razširljivosti, lahko dodajanje vozlišč in odpornost na odpoved vozlišč. Ne smemo pa pozabiti tudi na druge lastnosti kot so: izenačevanje obremenitve, celovitost podatkov, zmogljivost, ter seveda hitrost. V nadaljevanju bomo podrobneje opisali pomembnejše izmed njih.

Neodvisnost in avtonomnost vozlišč

Neodvisnost vozlišč pomeni, da nobeno od vozlišč ni odvisno od kateregakoli drugega vozlišča. Posledično morebitna odpoved katerega izmed vozlišč ne vpliva kritično na nobeno drugo vozlišče. Avtonomnost se sklicuje na samozadostnost – vsako vozlišče bi lahko delovalo tudi samo. Posledično imamo lahko v mreži poljubno število (seveda do nekega n) vozlišč.

Decentralizirano omrežje

Veliko omrežij v preteklosti se je samoupravljalo s pomočjo centralnega strežnika ali več njih. Omrežje Napster je naprimer delovalo s pomočjo centralnega strežnika, ki je koordiniral vsa vozlišča ter vodil seznam datotek ter njihovih lokacij. Takšna topologija je sicer zelo lahka za implementacijo, vendar prinaša več varnostnih in zmogljivostnih problemov. Ti so med drugim možnost zlorabe strežnika, možnost napada na strežnik ter s tem onemogočanje celotnega omrežja, očitno pa je strežnik tudi ozko grlo celotnega omrežja.

Decentraliziranost nam torej omogoča, da rešimo vse te opisane probleme in prenesemo breme celotnega omrežja na vsa vozlišča. Implementacija takšnega omrežja je vse prej kot lahka naloga. Poskrbeti moramo za pravilno komunikacijo med vozlišči, za varnost pred vdori in slabonamernimi vozlišči, omrežje pa mora še vedno delovati zelo hitro. Različna omrežja so te probleme reševala na različne načine o katerih bomo več povedali kasneje.

Hitra razširljivost

Potrebe bo dodatnih zmogljivostih se na uspešnih projektih povečujejo izredno hitro. Mogoče nam v začetnih fazah zadostuje le nekaj strežnikov, a ob povečanem prometu moramo čimhitreje dodati nove. Želimo si, da je dodajanje novih stržnikov/vozlišč nezamudno, najbolje celo, da je to dodajanje avtomatizirano. Potrebujemo torej omrežje, ki mu brez posebnega truda in dodatnih nastavitev lahko dodamo novo vozlišče, omrežje pa ga takoj sprejme za svojega in nanj preusmeri promet.

2.2.5 Prednosti in slabosti

Do tega trenutka smo dodobra spoznali porazdeljene zgoščevalne tabele. Hitro vidimo, da je njihova prednost predvsem v porazdeljenem shranjevanju elementov, dobri razširljivosti, same poskrbijo za porezdelitev elementov med vozlišči, odpornosti na odpovedi posameznih vozlišč (podatki se samodejno prestavijo na druga, sosednja vozlišča), v omrežju pa ni centralnega strežnika in torej ne očitnega ozkega grla.

Med slabosti porazdeljenih zgoščevalnih tabel lahko štejemo razdrobljenost podatkov. Podobna podatka se lahko nahajata na povsem različnih lokacijah. Posledično to pomeni, da moramo podobne podatke iskati po celotnem omrežju in se ne moremo zanašati na lokacijsko bližino.

2.3 Apache Cassandra

Apache Cassandra je odprtokoden in prostodostopen porazdeljen sistem za upravljanje s podatkovnimi bazami (PSUPB). Njegovi začetki segajo v leto 2008, ko sta prvo različico razvila Avinash Lakshman in Prashant Malik, zaposlena v podjetju Facebook. Apache Cassandra je napisan v programskem jeziku Java, njegov podatkovni model je podoben Googlovemu BigTable-u [3], za delovanje pa uporablja podobno infrastrukturo kot Amazonov Dynamo, opisan v članku Amazon's Dynamo [1]. Članek je bil tudi osnova za realizacijo in pričetek dela na odprtokodno različici Amazonovega Dynama, kasneje poimenovanega Cassandra. Leta 2009 ga je pod okrilje vzel Apache Foundation, ki ga ja preimenoval v Apache Cassandra ter označil za njihov "top-level" projekt. Od takrat Apache Foundation skrbi za razvoj in širitev Apache Cassandra projekta.

2.3.1 Podatkovni model

V času pisanja tega teksta je bila najnovejša stabilna različica Apache Cassandra 1.1.6. Podatkovni model, ki ga bom opisal v nadaljevanju torej velja

za to različico.

Kot smo omenili podatkovni model Apacha Cassandra temelji na Googlovi rešitvi BigTable [2]. A Apache Cassandra gre še dlje. Podatkovni model je nekakšen hibrid med BigTable-om in relacijsko podatkovno shemo. V splošnem iz obeh modelov vzame najboljše lastnosti in jih združi v nekakšen super model.

Osnovna skupina podatkov v Apache Cassandra je t.i. prostor ključev (ang. "keyspace"). Ta v podatkovnem modelu igra enako vlogo kot shema ali baza v relacijskem podatkovnem modelu. V prostoru ključev lahko določimo več družin atributov (ang. "column family"), katere lahko primerjamo z relacijami v relacijskem podatkovnem modelu. Vsaka družina atributov ima tudi skupino atributov, podobno kot jo ima relacija v relacijskem podatkovnem modelu. Tu pa opazimo največjo razliko. V Apache Cassandra lahko družini atributov določimo skoraj neomejeno atributov, ti pa se lahko uporabijo tudi za shranjevanje podatkov. Prav tako je ena izmed večjih razlik med modeloma v tem, da ima vsak zapis v relacijskem modelu natanko vse atribute kot jih ima vsak drugi zapis v neki relaciji. Če zapis neke vrednosti nima, se ponavadi na to mesto zapiše vrednost NULL. V podatkovnem modelu Apache Cassandra imajo lahko različni zapisi v eni družini atributov povsem različne atribute, vprašanje pa je do katere mere je te zapise potem še smiselno združevati v družine. Največja prednost te lastnosti je prihranek prostora, pozabiti pa ne smemo tudi na skorajšnjo neomejenost pri shranjevanju podatkov.

Naj omenim še nekaj dodatnih posebnosti podatkovnega modela, ki ga uporablja Apache Cassandra. Vsak zapis v družini atributov je določen s ključem, ki je v tej družini unikaten. Ta nam omogoča primarni indeks, kateri nam omogoča hitro iskanje elementov. Zapisu lahko določimo tudi sekundarne indekse, ki pa imajo smisel le, če je unikatnost podatkov v indeksiranem atributu dovolj velika. Atributom vsakega zapisa moramo določiti tudi podatkovni tip. Izbiramo lahko med standardnimi tipi kot so tekst, celo število, realno število, decimalno število, datum ali zaporedje znakov.

Izberemo pa lahko tudi tip super-atribut (ang. "Super-column"), ki nam omogoča, da v ta atribut zapišemo več vrednosti oblike parov [ključ,vrednost]. Ta tip atributa nam omogoča, da lahko podatke zapišemo v kakršnikoli obliki hočemo. Omogoča nam tudi popolno denormalizacijo podatkov, kar je izrednega pomena pri hitrosti poizvedb.

2.3.2 Infrastruktura

Infrastruktura, ki jo uporablja Apache Cassandra je povzeta po Amazonovem Dynamu [1]. V nadaljevanju bom opisal vsako izmed pomembnejših funkcionalnosti infrastrukture, ki skupaj omogočajo, da sistem lahko deluje nemoteno in zelo hitro.

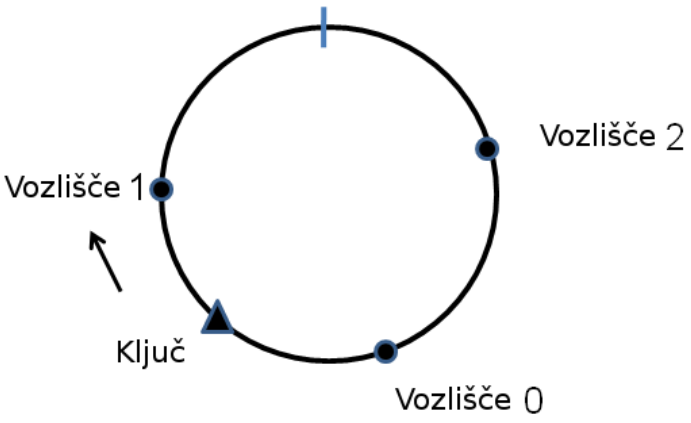
Razdelitev podatkov

Ena od zahtev dobre implementacije PSUPB-ja je možnost postopnega skaliranja. Da lahko učinkovito skaliramo obstoječo podatkovno bazo moramo imeti podatke dobro razdeljene. Apache Cassandra uporablja tehniko kosistentnega zgoščevanja [5] (ang. "consistent hashing"). Osnovna ideja konsistentnega zgoščevanja je v tem, da vozlišča razporedimo po navidezni krožnici, vsako vozlišče pa je odgovorno za podatke desno od samega sebe.

Replikacija

Za dosego dobre razpoložljivosti in trajnosti podatkov Apache Cassandra omogoča replikacijo podatkov. Implementacija in uporaba sta zelo enostavni, saj lahko stopnjo replikacije nastavimo kar preko parametra. S parametrom lahko določimo koliko kopij podatkov želimo v podatkovni bazi. Če to število nastavimo na n, potem je vsako vozlišče odgovorno in shranjuje poleg svojih podatkov še podatke svojih n-1 predhodnikov v krogu. Podatki so tako kljub morebitni odpovedi enega izmed vozlišč na voljo uporabnikom.

Posledica replikacije podatkov je ta, da imamo lahko v nekem trenutku v podatkovni bazi za isti podatek več različnih vrednosti. Kako skladni naj



Slika 2.2: Konsistentno zgoščevanje

bodo podatki ob branju ali pisanju, lahko določimo sami. Apache Cassandra za obe operaciji podpira več možnosti:

Stopnje skladnosti, ki so na voljo pri pisanju podatkov:

- ANY - Podatek mora biti uspešno zapisan vsaj na eno vozlišče
- ONE - Podatek mora biti uspešno zapisan vsaj na eno vozlišče, ki vsebuje kopijo tega podatka
- TWO - Podatek mora biti uspešno zapisan vsaj na dve vozlišči, ki vsebujeta kopijo tega podatka
- THREE - Podatek mora biti uspešno zapisan vsaj na tri vozlišča, ki vsebujejo kopijo tega podatka
- QUORUM - Podatek se zapiše na vozlišča, ki jih dobimo kot rezultat kvoruma. Kvorum je število vozlišč, katerih rezultate potrebujemo za uspešno akcijo. V Apache Cassandri se kvorum izračuna s formulo stopnja_replikacije/2 + 1

- LOCAL_QUORUM - Podobno kot QUORUM, le da sistem prisilimo, da se podatek zapiše v istem podatkovnem centru kot je koordinator ter s tem izboljšamo odzivnost
- EACH_QUORUM - Podobno kot QUORUM, le da mora biti podatek zapisan v vsak podatkovni center
- ALL - Podatek mora biti zapisan na vsa vozlišča

Stopnje skladnosti, ki so na voljo pri branju podatkov:

- ONE - Prebere podatek najbližje kopije
- TWO - Prebere najažurnejši podatek izmed najbližjih dveh kopij
- THREE - Prebere najažurnejši podatek izmed najbližjih treh kopij
- QUORUM - Prebere najažurnejši podatek izmed vseh kopij, ki sodelujejo v kvorumu
- LOCAL_QUORUM - Podobno kot QUORUM, le da v kvorumu sodelujejo le kopije iz lokalnega podatkovnega centra ter s tem izboljšamo odzivnost
- EACH_QUORUM - Podobno kot QUROUM, le da v kvorumu sodelujejo vozlišča iz vseh podatkovnih centrov
- ALL - Prebere najažurnejši podatek izmed vseh kopij

Različice podatkov

Podatki v podatkovni bazi večino časa niso povsem konsistentni, saj bi popolno transakcijsko okolje bistveno upočasnilo sistem. V vozliščih tako lahko ob nekem določenem času najdemo nekonsistennte različice podatkov, ki pa tekom časa slej ko prej postanejo konsistentne (ang. “eventual consistency”) oz. enake. To dosežemo z uporabo vektorskih ur, ki so nič drugega

kot pari verzije in predhodne verzije podatka, iz katerih na koncu izračunamo končo vrednost podatka.

Ravnanje ob odpovedih vozlišč

Odpoved vozlišča je le ena od neprijetnosti, ki se nam lahko zgodi na porazdeljenih omrežjih, a se ji ni moč izogniti. Zaznavanje odpovedi vozlišč je v Apache Cassandri implementirano kot površno sklepanje (ang. "sloppy quorum"). Ker imamo vozlišča razporejena po krožnici, se v vsakem primeru podatek zapiše v enega izmed delujočih vozlišč. Če podatek pride do vozlišča, ki primarno ni odgovorno zanj, to pomeni da je eno izmed predhodnih vozlišč odpovedalo. Podatek se vseeno shrani na to vozlišče, le v drug del baze. Ti deli baze se občasno pregledujejo in če niso prazni, obvestijo o tem druga vozlišča. V Apache Cassandri lahko nastavimo parametra, ki sistemu določita, koliko vozlišč moramo povprašati o morebitni odpovedi in koliko jih mora odgovoriti negativno, da napaka zares obstaja.

Dodajanje in odstranjevanje vozlišč

Dodajanje in odstranjevanje vozlišč lahko izvedemo zelo enostavno. Na krožnico dodamo novo vozlišče kamor želimo (ponavadi v najbolj obremenjeni del), podatki pa se v tem delu samodejno razporedijo na novo vozlišče in razbremenijo sosednja vozlišča. Za zagotovitev enakega bremena na vseh vozliščih moramo nato vsa vozlišča ponovno razporediti po krožnici. To storimo tako, da izračunamo nove lokacije glede na novo število vozlišč, podatki pa se nato prerazporedijo sami.

2.3.3 Omejitve

PSUPB Apache Cassandra ima tudi nekaj omejitev. Nekatere so začasne, prisotne v trenutni različici (a naj bi bile odpravljene v eni izmed prihodnjih različic), druge so predvidoma trajne.

Omejitve, ki naj bi kot take ostale so naslednje. Velikost vsakega zapisa

v družini atributov je navzgor omejena z velikostjo diska na posameznem vozlišču v posamezni gruči. Druga omejitev je velikost posameznega atributa zapisa, ki ne sme presegati dveh gigabajtov. Zapis ima lahko največ 10^9 atributov, velikosti vseh imen atributov ter vrednosti ključa pa so navzgor omejene s 64 kilobajti.

Trenutne omejitve, ki naj to ne bi bile več v kasnejših različicah so naslednje. Apache Cassandra ima dva nivoja indeksov, vendar obstaja tudi tretji nivo, ki se nahaja v super-atributih, ki pa ni indeksiran. V splošnem se moramo torej izogibati večjemu številu super-atributov. Druga omejitev je odsotnost pretočnega (ang. "streaming") dostopa do podatkov, kar pomeni, da mora vsak podatek, ki ga želimo zapisati ali ga prebrati biti manjši od velikosti RAM pomnilnika.

Prejšnje različice Apache Cassandre so imeli tudi druge omejitve a so bile odpravljene.

2.4 Object-relational mapper (ORM)

Object-relational mapper (ORM) je nivo v aplikaciji, ki poskrbi za preslikovanje nekompatibilnih oblik podatkov v objekte in obratno [11]. V večini aplikacijah, ki jih sprogramiramo v objektnem stilu, moramo nekatere podatke hraniti dlje časa, večkrat pa pridemo do problema kako to storiti. Seveda lahko za vsak tip podatka to počnemo ročno, vendar je to delo zelo zamudno in gre ponavadi le za kodiranje s katerim ne rešujemo nobenega konkretnega problema. Ob uporabi relacijske podatkovne baze za shranjevanje podatkov je pretvorba razredov v relacije in objektov v zapise v relacijah samoumevna. Samoumevna sta tudi pretvorba atributov razredov v atribute relacij ter uporaba tujih ključev za povezave med relacijami kot opis povezave med razredi.

Vendar pa se postavlja vprašanje kako objekte shranjevati v nerelacijsko bazo? Kako prikazati odvisnosti med razredi, kako zapisati atribute? Ali lahko podpremo vse značilnosti objektnega programiranja kot sta naprimer

dedovanje in enkapsulacija? Če uporabimo nerelacijsko podatkovno bazo za shranjevanje podatkov ORM kar na enkrat ni več ORM. Nič več ne preslikujemo razredov v relacije in objekte v zapise. Prilagoditi se moramo namreč načinu shranjevanja podatkov, kot ga uporablja željena baza. Lahko, da je to shanjevanje podatkov v obliki parov [ključ,vrednost], baza lahko uporablja shranjevanje podatkov v obliki dokumentov ali pa uporablja kakšno izmed hibridnih variant. V vsakem primeru moramo za preslikovanje poskrbeti ročno in kar se le da učinkovito. Kot smo že omenili je to delo ponavadi dokaj trivialno a zelo zamudno. Zato bi radi tudi med nerelacijsko bazo in našo apliakcijo imeli dodaten nivo, ki bi poskrbel za to vrsto preslikav, predvsem zato, da prihranimo čas razvijalcu in zmanjšamo možnosti neoptimalih preslikav.

Uporaba ORM v aplikaciji ima seveda dobre in slabe lastnosti. S pomočjo ORM-ja lahko hitreje pišemo kodo, koda je bolj optimizirana, ni se nam potrebno ukvarjati s specifikami posameznih SUPB-jev, olajša nam testiranje aplikacije, navsezadnje pa omogoča uporabo podatkovne baze manj veščim razvijalcem. Med manj dobre strani ORM-jev štejemo strah razvijalcev pred "še enim"dodatnim nivojem in tehnologijo. Praksa je pokazala tudi, da včasih ne potrebujemo vseh podatkov, ki nam jih vrne ORM in zato zgubljamo čas pri združevanju relacij ali branjem nepomembnih podatkov.

2.5 Normalizacija in denormalizacija

Normalizacija v kontekstu relacijskega modela podatkovne baze je postopek s katerim odpravimo čimveč odvisnosti in podvajanj podatkov v podatkovni bazi. Namen normalizacije je izolirati različne podatke, preprečiti podvajanje podatkov ter prikazati odvisnosti med podatki. Relacije po normalizaciji postanejo manjše in manj kompleksne. Kompleksnejše so le odvisnosti med njimi, katerim rečemo tuji ključi. Rezultat dobre normalizacije je tudi to, da imamo nek podatek zapisan le na enem mestu. Če želimo podatek spremeniti ali zbrisati, moramo to narediti le na enem mestu. Predstavil jo je E.F. Codd

leta 1970 v svojem članku z naslovom *A relational model of data for large shared data banks* [4]. Najbolj znane stopnje normalizacije so prva, druga in tretja normalna oblika, v zadnjih letih pa so raziskovalci določili stopnje celo do šeste normalne oblike.

Denormalizacija je postopek, ki se večinoma uporablja v nerelacijskem modelu podatkovne baze iz katere se podatki večinoma berejo ali pa vsaka transakcija pusti v bazi konsistentno stanje. Zapise v bazi iščemo preko ključa, zapisi pa se v bazi lahko ponavljajo ter s tem omogočajo hitrejši dostop. Dovoljeni so tudi atributi, ki vsebujejo več vrednosti saj s tem preprečimo iskanje in združevanje pripadajočih vrednosti. Večinoma se uporabljata dve stopnji denormalizacije. To sta prva normalna oblika (enako kot pri normalizaciji), ter neprva normalna oblika pri kateri atribut lahko vsebuje več vrednosti.

Poglavje 3

ORM nad Apache Cassandra PSUPB-jem

Na vprašanje zakaj bi želeli imeti ORM nad Cassandro bi lahko odgovorili zelo filozofsko, a najbolje se je držati dejstev in realnosti. V zadnjih letih se večino programske kode piše objektno. Razlogov za to je več. Koda je bolj pregledna, razredi se lahko uporabijo na več projektih, aplikacije lahko enostavno testiramo, vse to pa nam na dolgi rok prinese veliko boljše rezultate, kot če bi programirali proceduralno. Razredi in objekti so lahko razumljivi večini programerjev, enostavno pa si jih je predstavljati tudi v realnem življenju. Klasični primer je, če si za razred izberemo avtomobil, ki ima atributa znamko in model, objekt tega razreda pa naredimo, če mu ta dva atributa konkretno določimo. Če v programskem jeziku delamo z objekti in bi te objekte radi shranili v podatkovno bazo za kasnejšo uporabo, je potrebno objekte pretvoriti v obliko, ki je primerna za izbran tip SUPB-ja. Pri relacijskih SUPB-jih je pretvorba objektov zelo intuitivna. Vsak objekt predstavlja eno vrstico v relaciji, vsak atribut en stolpec. Kaj pa se zgodi, če hočemo razredu dodati bolj kompleksen atribut, npr. naslov, ki je ponavadi sestavljen iz več dodatnih atributov (kraj, hišna številka, pošta), ali več naslovov (stalnega in začasnega ter mogoče še naslov vikenda za obalo)? Ali pa če hočemo pri neki osebi prikazati graf družinskega drevesa v katerem

se nahaja? Vidimo, da se moramo hitro zateči k več relacijam in med njimi voditi povezave, ki jim pravimo tuji ključi. Pri poizvedbi moramo vse te tabele povezati med sabo, kar hitro poveča kompleksnost poizvedbe. Pri shranjevanju in ažuriranju je potrebno podatke dodati ali spremeniti v vsaki tabeli posebej, prav tako brisati. Vse to pripelje do veliko dela za programerja, saj bi moral ves čas skrbeti za ažuriranje podatkov vseh tabelah naenkrat.

Pri tem delu mu na pomoč priskoči ORM saj zanj opravi vse potrebne poizvedbe na podatkovni bazi. Za programerja je brisanje objekta povsem transparentno in ga ne zanima, kaj se dogaja v ozadju, prav tako bi lahko rekli za branje objekta. Kar programer hoče je le to, da ko zahteva nek določen objekt, tega pridobi iz podatkovne baze s čimmanj dela in čimmanj porabljenega časa. ORM mu pri tem zelo pomaga. Ampak še vedno nismo odgovorili na vprašanje, zakaj bi potrebovali ORM pri nerelacijskem SUPB-ju kot je Cassandra. Največji razlog je spet v hitrosti razvoja. Programerju namreč ni več potrebno skrbeti za ažuriranje objektov, za to poskrbi ORM. ORM lahko tudi poskrbi za optimalno shranjevanje objekta ter tako pomaga slabšim programerjem. Ampak kdaj se odločiti za relacijski SUPB in kdaj za nerelacijskega? Splošnega odgovora na to vprašanje ne bomo našli. Izkušnje so pokazale, da ob veliki količini podatkov nerelacijski SUPB daje boljše rezultate kot relacijski. Zavedati pa se moramo, da je bilo potrebno za to, da nerelacijski SUPB hitro izvaja operacije, skoraj vedno prilagoditi podatkovne strukture tako, da so te ustrezale ciljnim SUPB-jem. Da signifikantno izboljšamo hitrost, se skoraj vedno žrtvuje prostor na disku (ali v RAM-u). Podatke v ta namen denoramliziramo do željene strukture, podatkom dodajamo redundanco, v nekaterih SUPB-jih pa uporabimo tudi atribute, ki lahko vsebujejo drug objekt ali več njih. Nad vsem tem pa ponavadi bdi izjemno hiter in optimiziran indeks, ki željene podatke najde zelo hitro, včasih celo v konstantnem ($\mathcal{O}(1)$) času.

Kadarkoli želimo uporabiti nerelacijski SUPB moramo pri strukturi podatkovne baze izhajati iz poizvedb, katere želimo izvajati na njej. To je na-

jvečja in hkrati bistvena razlika v primerjavi z relacijsko podatkovno bazo. Pri relacijski podatkovni bazi si najprej zamislimo strukturo za podatke kakršne bomo hranili, nato pripravimo poizvedbe, dodamo potrebne indekse in upamo na najboljše rezultate. Tega si v Cassandri (in ostalih nerelacijskih SUPB-jih) ne moremo privoščiti. Razlog za to je predvsem v tem, da v splošnem, poleg primarnega, nerelacijski SUPB-ji ne poznajo indeksov. Sicer pri Cassandri lahko dodamo dodatne indekse, vendar se ti redko obnesejo tako kot si želimo. Problematične so predvsem poizvedbe, pri katerih ne vemo kaj natančno iščemo. Poizvedbe, kjer nas zanimajo elementi z določeno vrednostjo nekega atributa se bodo izvajale hitro. Prav nasprotno pa se zgodi pri poizvedbah kjer naprimer želimo elemente med sabo primerjati. Takšne operacije so naprimer večje od, manjše od, največji element ali najmanjši element. Poizvedbe se bodo izvedle izjemno neučinkovito, v najslabšem primeru mora SUPB iti čez vse elemente v svoji podatkovni bazi.

Pokazali smo torej, da je izbira vrste SUPB-ja odvisna predvsem od poizvedb, ki jih želimo nad podatkovno bazo izvajati. Žal še vedno nimamo splošnega odgovora na vprašanje kaj je bolje, v pravo smer pa se bomo odločili že, če vemo katere probleme bomo reševali in, če bomo pregledali kako so drugi reševali podobne probleme.

3.1 Prednosti in slabosti

Arhitekti aplikacij in njihovi razvijalci so glede uporabe ORM-ja v svojih aplikacijah razdeljeni v dve skupini. Nekateri prednosti uporabe ORM ne vidijo, drugi aplikacije brez ORM nivoja sploh nočejo razvijati. V nadaljevanju bom opisal nekaj prednosti in nekaj slabosti, ki jih prinese uporaba ORM nivoja v apliakcijah.

Prednosti, ki jih zagovarjajo privrženci so:

- Razvijalcu ni potrebno skrbeti kako so podatki shranjeni in kakonaj dostopa do njih
- Količina potrebne kode se zmanjša

- Zamenjava SUPB-ja ne zahteva nobene spremembe v kodi. Zamenjati moramo le ORM-jev gonilnik
- Sledenje entitetam
- Verižna ažuriranja in brisanja podatkov
- Veriženje objektov in dostopanje do željenih atributov
- Lažje prakticiranje TDD (Test Driven Development)
- Za varnost pred SQL vrivanjem je že poskrbljeno

Na drugi strani nasprotniki zagovarjajo svoje stališče z naslednjimi dejstvi:

- Dodaten čas za učenje nove tehnologije
- Ažuriranje velikih skupin podatkov je lahko zelo zamudno
- Razvijalci začnejo pozabljati kako sistem deluje v ozadju
- V splošnem je delovanje aplikacije, ki uporablja ORM počasnejše
- Pisanje kompleksnih poizvedb je ponavadi nemogoče
- Razvijalci nimajo dovolj nadzora nad generiranjem poizvedb
- Dodaten nivo abstrakcije - možno uhajanje abstrakcije, saj mora razvijalec vseeno pisati poizvedbe ročno

3.2 Obstoječi odjemalci za Apache Cassandra

Na idejo o uporabi ORM-ja kot dodaten, transparenten nivo med programskim jezikom in podatkovno bazo v Cassandri je prišlo že kar nekaj razvijalcev

Odjemalec	Programski jezik	Je ORM
Pycassa	Python	Ne
Telephus	Python	Ne
PlayOrm	Java	Da
Astyanax	Java	Ne
Hector	Java	Da
Kundera	Java	Da
Cassandrelle	Java	Ne
Cascal	Scala	Ne
Helenus	Node.js	Ne
Clj-hector	Clojure	Ne
Aquiles	.NET	Ne
Cassandraemon	.NET	Da
Fauna	Ruby	Ne
Phpcassa	PHP	Ne
Pandra	PHP	Da
SimpleCassie	PHP	Ne
libQtCassandra	C++	Ne
Cassy	Haskell	Ne

Tabela 3.1:

po svetu. Obstaja kar nekaj odjemalcev za uprorabo Cassandre za shranjevanje podatkov, a le nekaj izmed teh je res ORM-jev. V tabeli spodaj sem prikazal najbolj znane med njimi, dopisal programski jezik za katerega je odjemalec namenjen in pa oznako ali gre za ORM ali ne.

Poleg programskih jezikov, ki jih najdemo v tabeli, so podprti tudi nekateri drugi ne tako znani oz. popularni. Sam sem se odločil, da bom kot razvijalec v PHP jeziku nagradil odjemalec Phpcassa in mu dodal enostaven ORM, ki bo zmožen osnovnih operacij kot so kreiranje, branje, pisanje in brisanje zapisov.

Poglavje 4

Implementacija osnovnega ORM nad Apache Cassandra

4.1 Uvod

Za nalogo sem si postavil implementacijo enostavnega ORM sistema nad sistemom za upravljanje podatkovne baze Apache Cassandra. Kot smo že omenili ja Apache Cassandra nerelacijski PSUPB optimiziran za hitro in enostavno razširljivost. Implementacija ORM sistema za nerelacijski SUPB v splošnem ni enostavno rešljiv problem. Ker želim pokazati kako se lotiti svoje implementacije ORM, se bom omejil le na osnovne potrebe ORM sistema. To so kreiranje, branje, ažuriranje in brisanje objektov (CRUD – create, read, update, delete).

Ker sem sam najbolj domač v programskem jeziku PHP sem se odločil, da bom svoj ORM implementiral v tem jeziku. PHP (PHP Hipertext Preprocessor) je prostodostopen in odprt programski jezik, primarno namenjen procesiranju na strežniku, predvsem za pripravo spletnih strani. Je najbolj razširjen jezik na svojem področju, za svoje delo pa ga uporablja milijone razvijalecv po celem svetu.

Po kratki raziskavi sem našel le na en delno izveden ORM nad Cassandro za programski jezik PHP (Pandra). Čeprav že ima implementirane osnove

funcionalnosti, projekta že dolgo časa nihče ne vzdržuje. Posledično podpira le dve leti staro različico Apache Cassandre 0.7. Druga lastnost Pandre zaradi katere sem se odločil pisanja svojega ORM, je statično tipiziranje. Vse atribute družine atributov je namreč potrebno določiti vnaprej, ker razvijalcu delno omeji možnosti razvoja.

4.2 Osnovne potrebe

ORM mora omogočati hitro inicializacijo objektov. Pravtako mora omogočati branje objektov iz podatkovne baze, njihovo urejanje, shranjevanje in brisanje. Znati mora pretvarjati enostavne objekte v obliko primerno za zapis v podatkovno bazo in obratno. Če skupina atributov ne obstaja, jo mora znati zgraditi. Zavedati se mora ali objekt ima pripadajoči ključ ali ne. Navsezadnje pa mora delovati hitro, delovanje pa mora biti dovolj optimizirano, da ne opazimo sprememb v delovanju sistema.

ORM sem razdelil na dva dela. Prvi del je zadolžen za komunikacijo z odjemalcem Cassandre ter definicijo sheme in družin atributov, drugi del pa skrbi za definicijo razredov, ki jih bo razvijalec potreboval pri reševanju problemov.

Za povezavo do odjemalca in nato do Cassandrine podatkovne baze skrbi razred Keyspace. To je statični razred z atributi, ki jih potrebuje za povezavo na podatkovno bazo. Nastavimo lahko IP naslov podatkovne baze oz. enega izmed njenih vozlišč, port na katerem vozlišče posluša, ter shemo (Keyspace) v katero bomo shranjevali svoje podatke. Drugi razred je nadrazred za vse razrede, katerih objekte bomo shranjevali v podatkovno bazo, poimenoval pa sem ga ColumnFamilyElement. Zadolžen je za vse CRUD operacije, vsebuje pa tudi metode za lažje in hitrejše programiranje. Tretji razred je namenjen urejanju družin atributov, ki podpirajo super-atribute. Razred SuperColumnFamilyElement je le nadgradnja razreda ColumnFamilyElement, kateremu smo metode spremenili do te mere, da zna upravljati s super-atributi.

V delu ORM-ja, ki ga razvijalec prilagodi svojim potrebam, najprej na-

jdemo prazno mapo. V to mapo razvijalec dodaja razrede, ki morajo dedovati od ColumnFamilyElement ali SuperColumnFamilyElement razreda. Razvijalcu ni potrebno določiti nobenih atributov ali dodatnih metod, saj za osnovno uporabo za vse poskrbita nadrejena razreda.

Vzemimo za primer razred Uporabnik, ki je deduje po razredu ColumnFamilyElement. Vkolikor bomo hoteli shraniti objekt tega razreda v podatkovno bazo, se bo ta objekt shranil v družino atributov (Column family) z imenom Uporabnik. Če ta družina atributov še ne obstaja v shemi, jo bo razred naredil sam. Če objekt z danim ključem že obstaja v družini atributov, bomo objekt prepisali z novimi atributi. Število atributov načeloma ni omejeno (omejitev seveda obstaja, vendar je v realni uporabi ne bi smeli nikoli doseči). Branje elementa oz. njegovo isknje enostavno poteka preko njegovega ključa. Če ta obstaja, vrnemo pripadajoči objekt, če ne obstaja, naredimo nov objekt. Ta ORM bom uporabil pri reševanju svojega problema in rezultate primerjal z drugimi načini dostopa do podatkovnih baz.

4.3 Implementacija

Kot smo že omenili je ORM sestavljen iz dveh osnovnih delov. Prvi del, ki ga imenujemo jedro, je namenjen komunikaciji s podatkovno bazo in upravljanju z objekti, drugi ,uporabniški del, pa je namenjen uporabniku, kjer lahko določi katere razrede želi uporabiti in ali naj ti razredi podpirajo super atribute.

4.3.1 Jedro ORM-ja

Razred Keyspace

Razred Keyspace skrbi za povezavo na podatkovno bazo ter za upravljanje z njeno strukturo, tj. dodajanje družin atributov. Konstruktor razreda zahteva, da objektu ob inicializaciji določimo lokacijo strežnika in mu povemo katero podatkovno bazo (ang. “Keyspace”) na tem strežniku želimo uporabl-

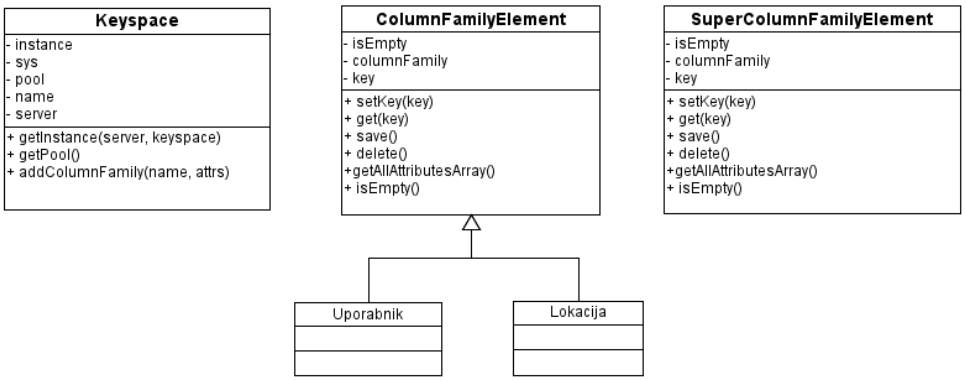
jati. Razred je pripravljen tako, da v pomnilniku hranimo le eno različico tega razreda.

Razreda ColumnFamilyElement in SuperColumnFamilyElement

Razreda ColumnFamilyElement in SuperColumnFamilyElement sta namenjena dedovanju, vsak dedovan razred pa skrbi za svojo družino atributov. Razlika med razredoma je le v tipu atributov, ki jih podpirata. Medtem, ko razred SuperColumnFamilyElement podpira tudi super atribute, jih razred ColumnFamilyElement ne.

Razreda imata vse potrebne metode za opravljanje s posameznimi zapisi v družini atributov. Omogočata kreiranje novega zapisa, shranjevanje zapisa, brisanje zapisa ter ažuriranje že obstoječega zapisa. Razreda sta sposobna tudi dodati družino atributov v podatkovno bazo vkolikor družina še ne obstaja. Razreda prav tako omogočata tudi branje zapisa, ki še ne obstaja, tako da zapis pripravi v pomnilniku, v podatkovno bazo pa ga shrani, če razvijalec izrecno zahteva shranitev.

Razlog za implementacijo dveh različnih razredov za upravljanje družin atributov je v tem, da ORM podpira dinamično tipizacijo, Apache Cassandra pa ne podpira super atributov v navadni družini atributov in obratno. Razvojnik bi lahko brez predhodne odločitve za kateri tip družine atributov gre tipe mešal, posledično pa bi v sistemu prihajalo do napak. V vsakem primeru je najboljša in edina pravilna rešitev, da se razvijalec ob kreiranju modela odloči kakšen tip podatkov bo shranjeval v dani družini atributov.



Slika 4.1: Rezredni diagram ORM-ja in njegove uporabe

4.3.2 Uporabniški del ORM-ja

Uporabniški del ORM-ja je namenjen razvijalcem. Razvijalec v ta del ORM-ja doda poljubne razrede, katerih objekte bo uporabljal pri razvoju. Razredi morajo dedovati od enega izmed ColumnFamilyElement ali SuperColumnFamilyElement razredov, družina atributov v podatkovni bazi pa bo poimenovana enako kot je poimenovan razvijalčev razred. Razredi v tem delu ne potrebujejo nobenih atributov ali metod, seveda pa jih lahko razvijalec doda, če se mu zdi, da jih potrebuje.

Poglavje 5

Reševanje realnega problema - iskanje najbližjega prostega termina

5.1 Opis problema

Za testiranje sem si izbral splošen problem iskanja in rezervacije najbližjega prostega temina ene izmed poslovalnic kateregakoli podjetja, ki se ukvarja z željeno dejavnostjo. Poslovalnice se nahajajo po celem svetu, proste termine pa posodabljamo v relanem času. Trenutna situacija realnega problema je takšna, da skrbimo za termine približno 3000 poslovalnic v Veliki Britaniji in na Nizozemskem. Predvidena je širitev v Avstralijo, Novo Zelandijo in Združene države Amerike, dodali pa bi tudi poslovalnice iz drugih panog. Ocenjeno število poslovalnic v roku enega leta je preseglo petsto tisoč. Zaradi potrebe po tekočem in zanesljivem delovanju bomo testirali delovanje sistema, ki upravlja z 10^6 poslovalnicami. Podjetja in poslovalnice bomo zaradi potrebe po testiranju naključno zgenerirali.

Predpostavimo lahko, da imamo podatkovno bazo podjetij velikostnega reda 10^5 zapisov. Ker ima vsako podjetje v povprečju 10 poslovalnic, imamo v podatkovni bazi vsaj 10^6 poslovalnic. Prosti termini nas zanimajo za nasled-

nji mesec, kar pomeni, da imamo v podatkovni bazi za vsako poslovalnico približno 30 prostih terminov. Tako vidimo, da ima naša največja struktura več kot $30x10^6$ zapisov. Za rezultat naše poizvedbe hočemo najbližjih 10 poslovalnic, ki imajo najzgodnejše termine. Opazimo, da bomo rezultate razvrstili po dveh različnih parametrih, zemljepisni oddaljenosti od željene lokacije in časovni oddaljenosti od trenutnega časa. Določimo, da ima vsak parameter svojo težo. Za primer vzemimo, da je 1 kilometer zemljepisne oddaljenosti enak 1 uri časovne oddaljenosti in vsaka od teh enot enaka 1 naši "skupni enoti". Tako lahko vidimo, da je poslovalnica z imenom "Poslovalnica1" oddaljena 12 kilometrov in ima najbližji prosti termin čez 24 ur oddaljena 36 "naših enot". Poslovalnica z imenom "Poslovalnica2", ki je oddaljena 34 kilometrov, a ima najbližji prosti termin čez 1 uro, oddaljena 35 "naših enot" in zato bližja kot "Poslovalnica1".

Reševanje problema lahko razdelimo na tri dele. Prvi del predstavlja pridobivanje relavantnih poslovalnic iz podatkovne baze, drugi del razvščanje pridobljenih poslovalnic glede na oddaljenost naše lokacije in trenutnega časa, tretji del pa pridobivanje podatkov o vseh prostih terminih za najbližjih n poslovalnic. V nadaljevanju bom opisal kako bi probleme rešil s pomočjo relacijskega sistema za upravljanje s podatkovno bazo (uporabil bom SUPB PostgreSQL 9.1 ter MySQL 5.5), in kako bi probleme rešil z nerelcijskim sistemom za upravljanje podatkovne baze (uporabil bom sistem Apache Cassandra 1.1.5). Med obema rešitvama bom primerjal hitrost razvoja in intuitivnosti rešitve ter hitrost delovanja sistema. Prav tako bom primerjal kako lahko uporaba ORM orodja pripomore k sami hitrosti razvoja aplikacije.

5.2 Izračun gostote poslovalnic

V tem poglavju bom opisal postopek, kako izračunamo gostoto poslovalnic na nekem področju na planetu, o katerem sem več prebral na spletni spletnem tečaju univerze v Washingtonu [15]. Že na začetku se moramo zavedati, da izračuni gostot ne bodo popolnoma točni, saj bomo planet razdelili na

kvadrante in izračunali gostoto v posameznih kvadrantih. Manjši kot bi bili kvadranti, bolj natančna bi bila gostota, vendar si prevelikega števila kvadrantov ne moremo privoščiti. Pri izračunu bom uporabil kvadrante velikosti ene stopinje zempljepisne širine krat ena stopinja zemljepisne dolžine. Pravtako bomo poenostavili sferično obliko zemlje do te mere, da bodo kvadranti pravokotniki. Zaradi poselitve prebivalstva na planetu in optimizacije samega algoritma bomo gostoto računali le med 60 stopinjami južne zemljepisne dolžine in 75 stopinjami severne zemljepisne dolžine. To pomeni, da moramo izračunati gostote 135x360 = 48600 kvadrantov. Vsaka poslovalnica pripada nekemu kvadrantu, število poslovalnic v nekem kvadrantu pa nam pove osnovno gostoto. Ker izračune delamo na sferi in se razdalja stopinje zemljepisne širine v kilometrih manjša moramo osnovne gostote popraviti z razmerjem velikosti kvadranta proti osnovnemu kvadrantu na ekvatorju. Za ilustracijo lahko v spodnji tabeli vidimo kako se spreminja razdalja stopinje zemljepisne širine, ko se povečuje zemljepisna dolžina.

Zemljepisna dolžina	Razdalja stopinje zemljepisne dolžinen	Razdalja stopinje zemljepisne širine
0°	110.574 km	111.320 km
15°	110.649 km	107.551 km
30°	110.852 km	96.486 km
45°	111.132 km	78.847 km
60°	111.412 km	55.800 km
75°	111.618 km	28.902 km
90°	111.694 km	0.000 km

Tabela 5.1:

Ko imamo tabelo gostot izračunano, je radij kroga v katerem bomo poslovalnice iskali enostavno izračunati. Če se nahajamo v kvadrantu z gostoto poslovalnic 1000 in si želimo za rezultat približno 100 poslovalnic, moramo zajeti poslovalnice v krogu, ki ima ploščino ene desetine ploščine kvadranta.

Stopinjo zemljepisne širine bi seveda popravili glede na zemljepisno dolžino na kateri se nahajamo. Ker želimo desetkrat manj rezultatov, moramo radij zmanjšati $\sqrt{10}$-krat. Torej 55km/$\sqrt{10}$ = približno 17.4km. Če bi se želeli številu rezultatov najbolj približati bi zaradi razmerja ploščine kroga in ploščine kvadrata morali polmer povečati za cca. $2/\sqrt{\pi}$.

Informacija, ki je dobimo je kako daleč od naše lokacije moramo iskati poslovalnice, da dobimo približno željeno število rezultatov.

5.3 Reševanje problema z relacijskim SUPB-jem

5.3.1 Pridobivanje relavantnih podatkov iz podatkovne baze

Relavanten podatek v našem primeru pomeni, da najbližji prosti termin ne sme biti preveč v prihodnosti in, da poslovalnica ne sme biti preveč oddaljena od naše trenutne lokacije. Privzeli bomo, da poslovalnica od naše trenutne lokacije ne sme biti oddaljena več kot petdeset kilometrov in da nas zanimajo prosti termini do najkasneje enega meseca vnaprej. Za izločitev relavantnih poslovalnic bomo zato v poizvedbo poslali tri parametre. To so zemljepisna širina in dolžina naše trenutne lokacije ter trenutni čas. Poizvedba nam mora vrniti le poslovalnice z oddaljenostjo največ petdeset kilometrov in najbližjim prostim terminom ne kasneje kot čez en mesec, tj. 30 dni (ali 2592000 sekund).

V relacijski podatkovni bazi so podatki iste vrste združeni v tabele ali relacije. Vsaka relacija ima svoje ime, potrebne atribute za shranjevanje podatkov, ki imajo določene podatkovne tipe, obstajati pa mora tudi vsaj en atribut, ki ga označimo kot primarni ključ (kot to zahteva željena stopnja normalne oblike). Za rešitev našega problema bomo uporabili tri relacije. Prva bo shranjevala podatke o podjetjih, druga bo shranjevala podatke o poslovalnicah in tretja bo shranjevala podatke o prostih terminih. Lokacije

poslovalnic bomo zapisali v atributa “lat” in “lon”, ki označujeta zemljepisno širino in dolžino lokacije poslovalnice, čas najbližjega prostega termina pa bomo (v sekundah) zapisali v atribut “closest_time”. Z nekaj razmisleka pridemo do SQL poizvedbe za vračanje rezutlatov, ki se glasi:

```
SELECT * FROM poslovalnice
WHERE razdalja<50 AND najblizji_termin<30dni
```

Če bi bilo milijon poslovalnic enakomerno porazdeljenih po površju Zemlje, bi jih v radiju 50km našli približno 14. Ker pa vemo, da veliko površine Zemlje predstavljajo morja in neposeljeno kopno, bi se lahko ta številka povečala za več kot desetkrat. Če hočemo izračune ves čas delati na približno enako pričakovano velikem številu poslovalnic, moramo pri poizvedbi upoštevati tudi gostoto poslovalnic na področju, kjer poizvedujemo za poslovalnicami. Dva robna primera sta naprimer center Londona, kjer lahko v radiju 50 kilometrov najdemo več sto poslovalnic, nasprotni primer pa je kakšno odročno podeželje, kjer v radiju 50 kilometrov najdemo kvečjemu eno poslovalnico. Izračune gosote po kvadrantih po planetu bi, kot smo že omenili, imeli izračunane predhodno. V našem primeru bi nato glede na dano lokacijo in željeno število zadetkov izračunali polmer iskanja, ki ne bi bil večji od 50 km in nato v tem radiju poiskali relavantne poslovalnice. Primer, da v naslednjem mesecu ni prostega termina v kateri izmed poslovalnic bi zanemarili, saj so takšni primeri zelo redki. Končna SQL poizvedba v SUPBju PostgreSQL 9.1, če smo na lokaciji (52,-1) in želimo radij desetine stopinje, bi izgledala takole:

```
SELECT id FROM providers
WHERE |/ ((lat-52)^2+(lon+1)^2)<0.1
```

ali

```
SELECT id FROM poslovalnice
WHERE lat<52.01 and lat>51.99 and lon<-0.99 and lon>-1.01
```

Poizvedba na milijon in dvesto tisoč vrsticah na povprečnem računalniku za prvo poizvedbo traja dobrih 800ms, za drugo dobrih 300ms, ter za drugo

z BTREE indeksom na poljih lat in lon dobrih 4ms. Kljub indeksu se razumljivo trajanje prve poizvedbe ne spremeni

5.3.2 Razvrščanje rezultatov ter pridobivanje podrobnosti poslovalnic

Razvščanje samo lahko implementiramo kar v aplikaciji. Razvščanje nekaj deset elementov glede na neko številsko vrednost je nezamudno delo za katerikoli programski jezik.

Ko imamo določenih prvih deset rezultatov, moramo še vedno za vsakega od njih v podatkovni bazi poiskati podrobnosti. Ker imamo ključe rezultatov oz. poslovalnic že podane, se vse te poizvedbe izvedejo izredno hitro. Na relacijah v podatkovni bazi lahko uporabimo tudi zgoščevalne indekse, kateri nam željeni zapis vrnejo v času $\mathcal{O}(1)$, Realna hitrost je seveda odvisna od zmogljivosti naših strežnikov, vendar poizvedbe za deset elementov ne smejo trajati več kot nekaj milisekund.

5.4 Reševanje problema z Apache Cassandra PSUPB-jem

5.4.1 Pridobivanje relavantnih podatkov iz podatkovne baze

Reševanje problema, kjer poizvedba zahteva iskanje po nekem razponu vrednosti atributa, je v Apache Cassandri zelo neučinkovito. Iskanje je neučinkovito zaradi dizajna sistema, ki zapise shranjuje in porazdeljuje po vozliščih na podlagi ključa. Ker za ključe naših zapisov ne moremo dati lokacij, lahko uporabimo sekundarni indeks na naš atribut. Vendar, če želimo narediti poizvedbo po nekem razponu vrednost atributa, moramo kot pogoj v poizvedbi podati vsaj eno enakost, ki se nanaša na primarni ključ. Kakorkoli razmišljamo in rešujemo ta problem, Apache Cassandra ni najbolj primerna

za reševanje takega problema, če se ne poslužimo dodatnih trikov in si lahko privoščimo tudi neekzaktne rezultate.

Ideja, ki sem jo uporabil za rešitev našega problema je v razdelitvi vsega iskalnega področja na kvadrante, ki so dovolj majhni, da omogočajo zadostno natančnost iskanja in so še vedno dovolj veliki, da končno število kvadrantov ni preveliko. Za velikost kvadrantov sem si izbral področja velika desetino stopinje zemljepisne širine krat desetino stopinje zemljepisne dolžine. Nato sem vsem poslovalnicam določil kvadrante in za ključ zapisa v podatkovni bazi izbral oznako kvadranta. Vsak kvadrant je nato imel v super atributu shranjene poslovalnice, ki se nahajajo v njem.

Po izračunu gostote področja na katerem se nahajamo nato izvedemo le poizvedbe o poslovalnicah, ki se nahajajo v kvadrantih okoli naše lokacije. Če je gostota poslovalnic majhna, poiščemo poslovalnice v večjem obsegu kvadrantov kot, če je gostota velika, kjer se lahko zadovoljimo že s poslovalnicami v kvadrantu v katerem se nahajmo. Ko so poizvedbe končane, imamo v rokah poslovalnice, ki ustrezajo našim pogojem, potrebno jih je le še razvrstiti in za prvih deset poiskati podrobnosti.

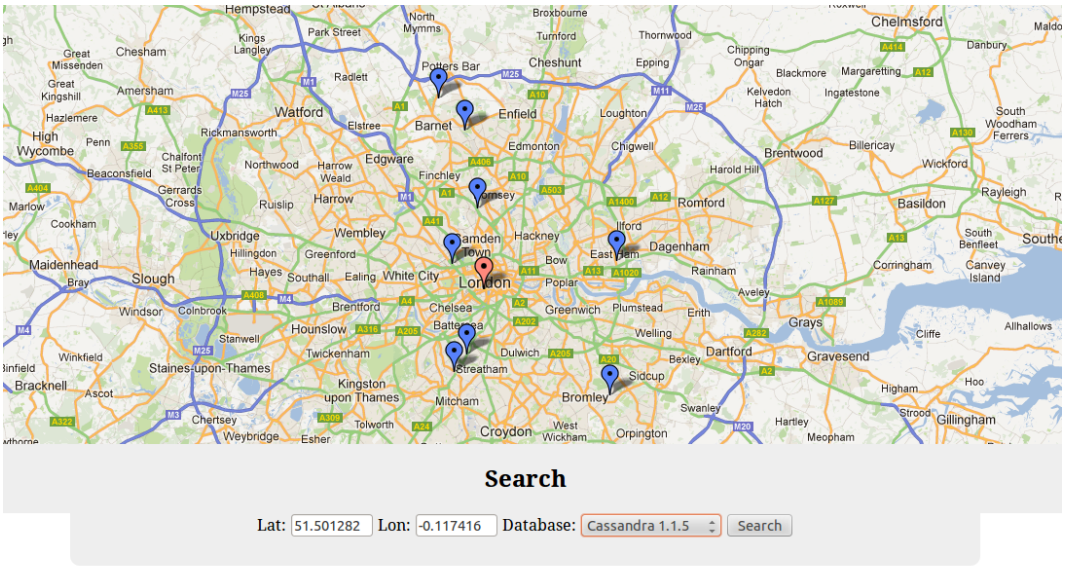
5.4.2 Razvrščanje rezultatov ter pridobivanje podrobnosti poslovalnic

Ko pridobimo željen obseg poslovalnic, razvrščanje in pridobivanje podrobnosti izvedemo na enak način kot pri relacijski podatkovnih bazah. Poslovalnice razvrstimo kar v apliakciji saj gre za relativno enostavno in hitro operacijo. Podrobnosti o prvih desetih poslovalnicah pridobimo z dodatnimi poizvedbami, ki pa se v Apache Cassandri izvedejo izredno hitro. Vsi zapisi so namreč indeksirani s primarnimi ključi, katere pa že imamo. Tako kot pri relacijskih bazah se tudi pri Apache Cassandri take poizvedbe izvedejo v nekaj milisekundah.

5.5 Uporaba ORM-ja v praksi

Da bi stestiral katere prednosti in slabosti prinese uporaba ORM-ja v praksi, sem se lotil razvoja aplikacije, kjer sem svoj ORM uporabil. Zamislil sem si spletni vmesnik za iskanje najbližje poslovalnice, razultate pa bi nato prikazal na zemljevidu. Ker hočem testirati tudi hitrosti različnih SUPB-jev, bi imel na vmesniku tudi možnost izbire iz katere podatkovne baze bi rad podatke.

Z uporabo Google Maps storitve sem pripravil spletni vmesnik, ki ga lahko vidimo na spodnji sliki. Uporabnik lahko s klikom na zemljevid določi svojo lokacijo, izbere podatkovno bazo in sproži iskanje. Vmesnik nato z AJAX [1] klicem pokliče API moje aplikacije, ta pa mu vrne rezultate in čas trajanja poizvedbe. Rezultate in čas trajanja nato izpišem na zaslonu.



Slika 5.1: Spletni vmesnik aplikacije. Rdeči označevalnik prikazuje našo lokacijo, modri označevalniki pa prikazujejo lokacije poslovalnic.

Uporaba ORM-ja je zelo pohitrila razvoj aplikacije v fazi, kjer sem delal

[1] Asynchronous JavaScript and XML - tehnika, ki spletnim vmesnikom omogoča komunikacijo s strežnikom v ozadju

s posameznimi objekti oz. poslovalnicami. Prav tako se je zmanjšal obseg kode ter povečala preglednost. Podatke o eni poslovalnici sem lahko kjerkoli v kodi dobil samo s kreiranjem objekta, kar mi je prihranilo pisanje poizvedbe in ročne povezave na bazo.

Ni pa šlo vse po pričakovanjih. Ko sem v neki fazi potreboval podatke večih poslovalnic naenkrat, sem moral kreirati vsak objekt posebej in se torej za podatke vsake poslovalnice v ozadju povezati na podatkovno bazo. Hitrost aplikacije se je zato drastično zmanjšala. Da bi pospešil delovanje apliakcije, sem uporabil hibridni pristop in sem za pridobitev podatkov o večih poslovalnicah pogledal v podatkovno bazo mimo ORM-ja, ko pa sem obdeloval posamezne poslovalnice sem ponovno uporabil ORM. Z vidika konsistence in preglednosti kode tak pristop ni najlepši in ga ne priporočam.

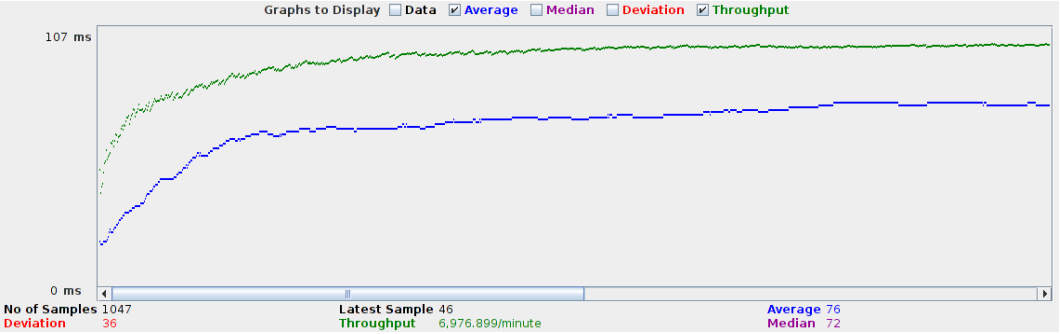
Potrebno se je zavedati, da uporaba ORM-ja pride veliko bolj do izraza, ko veliko delamo z objekti oz. zapisi samimi, ko podatke ne samo beremo, ampak tudi veliko urejamo in brišemo. Če imamo aplikacijo, ki večino časa iz podatkovne baze le bere je potrebno razmisliti ali res potrebujemo ORM in ali bo uporaba dodatnega nivoja abstrakcije odtehtala slabosti, ki jih uporaba ORM-ja prinese s seboj.

5.6 Testiranje zmogljivosti SUPB-jev

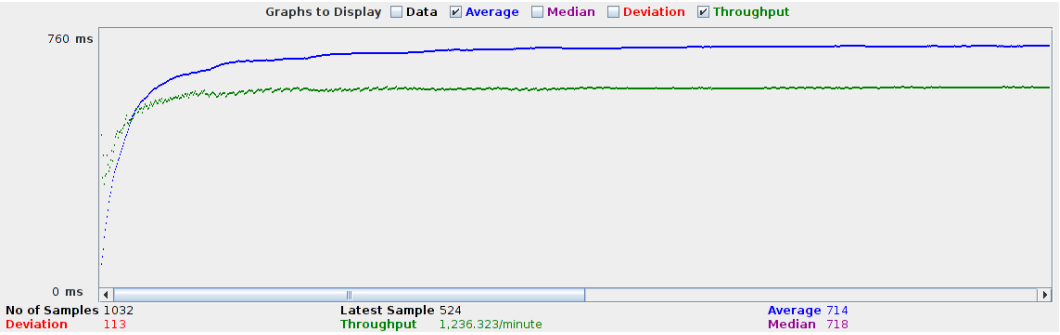
Obremenitvene teste sistemov sem implementiral z aplikacijo JMeter. JMeter je aplikacija, ki je namenjena raznovrstnim testiranjem sistemov in aplikacij. Tako kot Cassandra je tudi JMeter pod okriljem Apache Foundation, ki skrbi za njen nemoten razvoj.

Pričakovana obremenitev aplikacije v realnosti je bila ocenjena na 15 poizvedb na sekundo, zato sem simuliral 150 uporabnikov v 10 sekundah, ki so enakomerno porazdeljeno zahtevali rezultate poizvedbe o poslovalnicah v krogu 50 kilometrov. PostgreSQL in MySQL sta tekla na enoprocesorskem 2Ghz hitrem strežniku s 768MB delovnega pomnilnika. Apache Cassandra je tekla na treh vozliščih, od katerih je imel vsak en procesor z 1Ghz ter 768MB

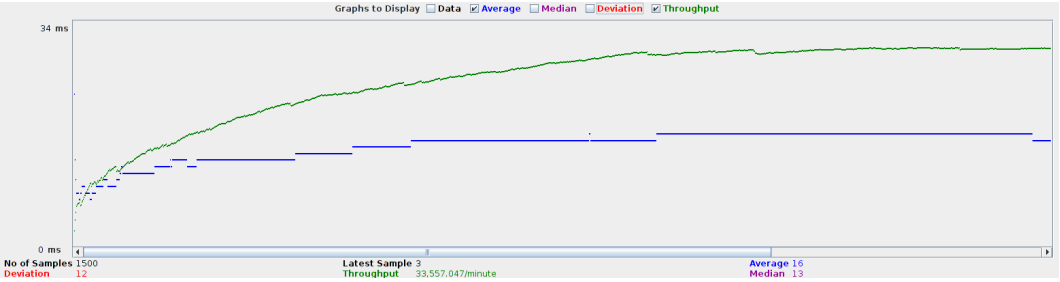
delovnega pomnilnika. Vse strežnike sem poganjal na virtualnih strežnikih na delovni postaji s štirijedrnim procesorjem Intel i7 in 8GB delovnega pomnilnika.



Slika 5.2: Rezultati obremenitve MySQL podatkovne baze



Slika 5.3: Rezultati obremenitve PostgreSQL podatkovne baze



Slika 5.4: Rezultati obremenitve Apache Cassandra podatkovne baze

Vidimo, da bi načrtovano breme vse tri podatkovne baze preživele tudi na strežnikih, kot sem jih uporabil za testiranje. Najslabše se je obnesel sistem s PostgreSQL SUPB-jem, ki bi prenesel približno 20 poizvedb na sekundo, sledi mu sistem z MySQL SUPB-jem, ki bi prenesel približno 115 poizvedb na sekundo, najbolje pa se je pričakovano obnesela gruča z Apache Cassandra SUPB-jem, ki bi prenesla približno 550 poizvedb na sekundo. Čeprav je bil sistem z Apache Cassandro približno trikrat zmogljivejši od sistema z MySQL, je prvi še vedno približno 60% hitrejši.

Poglavje 6

Razprava

Namen te naloge je bil preveriti, kako se razlikuje razvoj rešitve nekega splošnega realnega problema, če pri razvoju uporabimo dodaten nivo v obliki ORM-ja in kako se razlikujejo pristopi k reševanju problema, če za shranjevanje podatkov uporabimo relacijski oz. nerelacijski SUPB. Preveril sem tudi kako zahtevno je implementirati osnovni ORM za nerelacijski SUPB in na kaj vse moramo biti pri razvoju tega ORM pozorni.

6.1 Uporaba SUPB-ja

Kot smo omenili v prejšnjem poglavju uporaba ORM ne prinese vedno le prednosti ampak tudi slabosti. Dobro moramo razmisliti kaj želimo s podatki početi in kakšne omejitve ima izbrani ORM. Če naša aplikacija večino časa podatke bere ali velikokrat iz podatkovne baze zahteva več zapisov hkrati, potem uporabo ORM-ja odsvetujem. Vkolikor je naloga naše aplikacije večinoma urejanje posameznih zapisov v podatkovni bazi ali tudi samo branje posameznih zapisov, potem nam uporaba ORM-ja zelo olajša in pohitri delo. Preveriti moramo tudi ali je ORM dobro optimiziran, da ne zavira delovanja naše aplikacije.

Z izbiro in uporabo različnih tipov SUPB-jev sem si nabral veliko izkušenj. Niso vsi SUPB-ji primerni za reševanje vsakega problema. Če želimo de-

lati poizvedbe po intervalu vrednosti nekega atributa, se nerelacijski SUPB Apache Cassandra izkaže za manj primernega. To je posledica dizajna samega SUPB-ja, saj za indeksiranje ključev uporablja zgoščevanje, ta pa je neučinkovit pri operacijah z intervali. Tu je v prednosti relacijski SUPB, saj lahko na atributih uporabimo indeks drevesne strukture, na katerem so operacije nad intervali zelo učinkovite. Vkolikor delamo poizvedbe s katerimi direktno naslovimo zapise v podatkovni bazi, pa se izkaže neralcijski SUPB za izredno hitrega, predvsem zaradi svojega indeksa. Vendar lahko tudi v relacijski podatkovni bazi na atributu kreiramo zgoščeni indeks s čimer pri poizvedbah tega tipa nerelacijski SUPB ni v prednosti. Prednost nerelacijskega SUPB se pokaže predvsem pri dodajanju atributov zapisom v bazi, ter neomejenosti pri določanju atributov, saj imajo lahko zapisi v isti družini atributov povsem različne množice atributov. Prav tako SUPB ne rezervira prostora vsakemu atributu zapisa, če atribut dodamo le enemu zapisu. Ker nimamo podobnih omejitev, je spreminjanje sheme podatkovne baze enostavno in nezamudno opravilo, ki ne zavira delovanja baze med njenim delovanjem.

6.2 Načrtovanje sheme podatkovne baze

Z uporabo nerelacijskega SUPB-ja se mora spremeniti tudi način prostopa k reševanju problema. Ker se mora shema podatkovne baze določiti glede na poizvedbe, ki jih želimo opravljati, morajo biti vse poizvedbe, ki jih želimo na podatkovni bazi opravljati, vnaprej znane. Shemo podatkovne baze nato prilagodimo potrebam poizvedb in omejitvam izbranega SUPB-ja. Sam sem imel kar nekaj težav z izbiro sheme podatkovne baze, saj sem želel delati poizvedbe po intervalih ne da bi vedel kateri podatek natančno želim. Žal se problema, ki sem si ga zadal, ni dalo ekzaktno rešiti na način, da bi aplikacija delal dovolj hitro. Moral sem sprejeti kompromise, a na koncu sem implementiral shemo, ki je omogočala dovolj hitr poizvedbe.

6.3 Implementacija ORM-ja

Implementacija učikovitega in uporabnega ORM-ja zahteva veliko premisleka, načrtovanja in poznavanja programskega jezika v katerem je napisan in izbranega SUPB-ja. Ravno zato sem se sam odločil za uporabo programskega jezika PHP, ki sem ga najbolj vešč. Vedeti moramo katere operacije bomo z ORM-jem podprli in katerih ne. Poznati moramo omejitve izbranega SUPB-ja. Sam sem moral prebrati kar nekaj zapisov o lastnostih, ki naj bi jih ORM imel in kako so drugi razvijalci reševali probleme na katere so naleteli ob razvoju ORM-ja. Nekaterih potreb se žal ne da zadovoljiti zaradi objektno-relacijskega neujemanja (ang. "Object-relational impedance mismatch" [13]).

Večja odločitev, ki sem jo moral sprejeti je bila ali naj v ORM-ju uporabim dinamično ali statično tipiziranje [12]. Statično tipiziranje vzdržuje red v objektih in aplikaciji, dinamično tipiziranje pa dovoli razvojniku več svobode, a nanju prelaga tudi veliko odgovornosti. Ker tudi Apache Cassandra dovoljuje manjkajoče in dodatne atribute ter omogoča dinamične atribute, sem se odločil, da bom pri implementaciji ORM uporabil dinamično tipizacijo. To pomeni, da razvijalcu ni potrebno predhodno določiti katere atribute ima razred, niti mu ni potrebno preverjati tipov atributov.

Literatura

[1] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swami Sivasubramanian, Peter Vosshall and Werner Vogels, "Dynamo: Amazon's Highly Available Key-Value Store", *Proceedings of the 21st ACM Symposium on Operating Systems Principles, Stevenson, WA* ,Oktober 2007.

[2] Fay Chang, Jefrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber, "Bigtable: A distributed storage system for structured data.", *In Proceedings of the 7th Conference on USENIX Symposium on Operating Systems Design and Implementation - Volume 7*, pages 205-218, 2006.

[3] Avinash Lakshman, Prashant Malik, "Cassandra - A Decentralized Structured Storage System", dosegljivo na *http://www.cs.cornell.edu/Projects/ladis2009/papers/Lakshman-ladis2009.PDF*

[4] E. F. Codd, "A relational model of data for large shared data banks", dosegljivo na *http://www.seas.upenn.edu/ zives/03f/cis550/codd.pdf*

[5] Karger, D., Lehman, E., Leighton, T., Panigrahy, R., Levine, M., and Lewin, "Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the World Wide Web." *STOC '97. ACM Press, New York, NY, 654-663*

[6] Andresen Dave, "Distributed Hash Tables", dosegljivo na *http://www.cs.cmu.edu/ dga/15-744/S07/lectures/16-dht.pdf*

[7] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, Scott Shenker, "A Scalable ContentAddressable Network", dosegljivo na *http://berkeley.intel-research.net/sylvia/cans.pdf*

[8] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, Hari Balakrishnan, "Chord: A Scalable Peertopeer Lookup Service for Internet Applications", dosegljivo na *http://pdos.csail.mit.edu/papers/chord:sigcomm01/chord$_s$igcomm.pdf*

[9] Antony Rowstron, Peter Druschel, "Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems", dosegljivo na *http://www.cs.rice.edu/ druschel/publications/Pastry.pdf*

[10] Pranab Ghosh , "Cassandra Secondary Index Patterns", dosegljivo na *http://pkghosh.wordpress.com/2011/03/02/cassandra-secondary-index-patterns*

[11] Ambler W. Scott, "Mapping Objects to Relational Databases: O/R Mapping In Detail", dosegljivo na *http://www.agiledata.org/essays/mappingObjects.html*

[12] Patel Jay, Ebay tech blog, "Cassandra Data Modeling Best Practices", dosegljivo na *http://www.ebaytechblog.com/2012/07/16/cassandra-data-modeling-best-practices-part-1/* in *http://www.ebaytechblog.com/2012/08/14/cassandra-data-modeling-best-practices-part-2/*

[13] Hewitt Eben, "Cassandra: the definitieve guide" *O'Reilly Media, November 2010*

[14] Več avtorjev , "Slides and presentations from talks at Cassandra Europe", dosegljivo na *http://cassandra-eu.org/presentations*

[15] Več avtorjev , "Projections and Coordinate Systems", dosegljivo na *http://courses.washington.edu/gis250/lessons/projection/*

[16] Več avtorjev, "Object-relational impedance mismatch", dosegljivo na *http://en.wikipedia.org/wiki/Object-relational_impedance_mismatch*