UNIVERZA V LJUBLJANI

FAKULTETA ZA RAČUNALNIŠTVO IN INFORMATIKO

Nejc Škerjanc

# PRIMERJAVA UČINKOVITOSTI NOSQL IN RELACIJSKE PODATKOVNE BAZE

DIPLOMSKO DELO

VISOKOŠOLSKI STROKOVNI ŠTUDIJSKI PROGRAM PRVE STOPNJE

RAČUNALNIŠTVO IN INFORMATIKA

Mentor: viš. pred. dr. Aljaž Zrnec

Ljubljana, 2012

Rezultat diplomskega dela so intelektualna lastnina avtorja in Fakultete za računalništvo in informatiko Univerze v Ljubljani. Za objavljanje ali izkoriščanje rezultatov diplomskega dela je potrebno pisno soglasje avtorja, Fakultete za računalništvo in informatiko ter mentorja.

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: 00251/2012
Datum: 05.04.2012

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

Kandidat: **NEJC ŠKERJANC**

Naslov: **PRIMERJAVA UČINKOVITOSTI NOSQL IN RELACIJSKE PODATKOVNE BAZE**

**PERFORMANCE COMPARISON BETWEEN NOSQL AND RELATIONAL DATABASE**

Vrsta naloge: Diplomsko delo visokošolskega strokovnega študija prve stopnje

Tematika naloge:

Relacijske podatkovne baze predstavljajo standard na področju upravljanja s podatki že 30 let. V svoji osnovi se niso bistveno spremenile in veliko strokovnjakov je bilo prepričanih, da se na tem področju ne bo razvilo nič novega. S pojavom socialnih omrežij, računalništva v oblaku in zahtev po procesiranju tokov podatkov, pa relacijske baze niso več mogle slediti zahtevam po visoki razpoložljivosti in skalabilnosti. V okviru diplomske naloge predstavite podatkovne baze NoSQL. Predstavite razloge za pojav teh baz, kakšne vrste NoSQL baz poznamo in predstavite njihove lastnosti. V okviru praktičnega dela izdelajte performančno analizo za NoSQL bazo Cassandra in klasično relacijsko bazo MySQL. Pri tem je potrebno obe podatkovni bazi namestiti na več vozlišč.

Mentor:

viš. pred. dr. Aljaž Zrnec

Dekan:

prof. dr. Nikolaj Zimic

Fakulteta za računalništvo in informatiko
2
UNIVERZA V LJUBLJANI

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

## diplomskega dela

Spodaj podpisani Škerjanc Nejc,

z vpisno številko 63090274,

sem avtor diplomskega dela z naslovom:

**Primerjava učinkovitosti NoSQL in relacijske podatkovne baze**

S svojim podpisom zagotavljam, da:

- sem diplomsko delo izdelal/-a samostojno pod mentorstvom (naziv, ime in priimek)

  **viš. pred. dr. Aljaža Zrnca**

- 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 _____ Podpis avtorja: _____

## Zahvala

Zahvaljujem se mentorju viš. pred. dr. Aljažu Zrncu za potrpežljivost, nasvete in podporo pri izdelavi diplomske naloge.

Prav tako gre zahvala tudi mojim staršem in vsem domačim, ki so me podpirali, spodbujali in verjeli vame v času mojega študija.

## Kazalo slik

## Povzetek

Cilj diplomske naloge je primerjava relacijskih podatkovnih baz z NoSQL podatkovnimi bazami. Relacijske podatkovne baze so v uporabi že več kot 30 let. Večina strokovnjakov je bila prepričanih, da se ne bo razvilo na tem področju nič novega. Z razvojem socialnih omrežij, računalništva v oblaku ter zahtev pri procesiranju tokov podatkov so se razvile NoSQL podatkovne baze. Te so lahko sledile potrebi po visoki razpoložljivosti in skalabilnosti. V diplomski nalogi je narejena predstavitev najbolj znanih NoSQL podatkovnih baz ter opis in predstavitev lastnosti le-teh. V praktičnem delu je predstavljena performančna analiza med NoSQL bazo Cassandra in klasično relacijsko bazo MySQL. Nameščeni sta na več vozliščih. Predstavljeno je tudi rokovanje ter upravljanje s Cassandro ter primerjava s stališča varnosti v primeru poskusov vdora v sistem oziroma tako imenovanih »višjih sil« (potresi, poplave itd.). Cilj diplomske naloge je podati konkretne odgovore glede izbire relacijskih baz oziroma NoSQL podatkovnih baz, ki temeljijo na praktičnih performančnih testih.

## Abstract

The aim of this bachelor thesis is a comparison between relation databases and NoSQL databases. The first ones are in use for more than 30 years. The majority of professionals were convinced that they achieved top of technology on this field. The NoSQL databases were developed with the development of social network, computing in cloud and processing data stream. Those databases could easily follow the need of high availability and scalability. In this bachelor thesis the most known NoSQL databases are presented with description and presentation features. The practical part introduces performance analysis between NoSQL database Cassandra and classical relation database MySQL. They are both installed on more nodes. We described handling and managing with Cassandra, comparison from the perspective of safety in case of hacking into system or the possibility of natural disasters (earthquake, floods, etc.)With this bachelor thesis the aim is to offer a specific answer about choice between relation databases or NoSQL databases, which are based on practical performance tests.

# Poglavje 1

# Uvod

Relacijske podatkovne baze so bile razvite pred več kot tremi desetletji. Zelo veliko znanstvenikov, inženirjev in ekspertov na tem področju je bilo prepričanih, da so podatkovne baze doživele svoj zaključek evolucije z odkritjem relacijskih podatkovnih baz. Menili so namreč, da ne more več priti do konkretnih sprememb.

## 1.1 Uvod v podatkovne baze ter motiv

Relacijske podatkovne baze so relativno hitre, enostavne za izvajanje transakcij, človeku razumljive zaradi dvodimenzionalnega prikaza.

Relacijske podatkovne baze je možno povezati tudi v skupino ali gručo z arhitekturo gospodar-suženj (angl. master-slave). Prav arhitektura gospodar-suženj pa ni ravno najboljša. Zaradi različnih razlogov lahko strežnik, ki ga določimo kot gospodarja, ne deluje več in celoten sistem postane neuporaben.

Razlogov, zaradi katerih lahko strežnik ne deluje več, je zelo veliko. Delijo se na človeške dejavnike ter naravne dejavnike. Med človeške dejavnike spadajo predvsem vdori v strežnik, napake v programski kodi, napake v administraciji strežnika, namerna fizična prekinitev dostopa strežnika do svetovnega spleta itd.

Med naravne dejavnike spadajo predvsem naravne nesreče oziroma tako imenovane »višje sile«. Sem spadajo potresi, poplave, cunamiji, udari strele itd. Po nekaterih podatkih naj bi veliko naravno grožnjo predstavljale tudi Sončeve nevihte. Te v električnih prevodnikih povzročijo inducirano napetost in posledično ogromen presežek energije.

Pri klasičnih relacijskih bazah z arhitekturo gospodar-suženj je problem ozko grlo, ki nastane pri gospodarju. Vse transakcije morajo biti naslovljene na gospodarja. Ta potem razporedi proces izvajanja transakcij po podrejenih strežnikih. Zaradi ozkega grla se celoten sistem lahko upočasni.

Ob prelomu tisočletja so se začele pojavljati potrebe po decentraliziranosti strežnikov po celem svetu. Potrebovali so tudi sistem, ki bo lokalno orientiran, ki omogoča serviranje in procesiranje podatkov iz lokalnega okolja. Strežniki morajo biti med seboj neodvisni, da ob izpadu kateregakoli strežnika preostali delujejo naprej. Transakcije, ki bi se izvajale na izpadlem strežniku, se morajo prerazporediti na ostale strežnike. Sistem mora biti skalabilen

in ga velika količina podatkov ne sme preobremeniti. Vse zgoraj naštete lastnosti nam ponujajo NoSQL baze.

Prednost NoSQL podatkovnih baz je zmožnost sprotnega procesiranja podatkovnih tokov in ogromnih količin podatkov.

Kljub temu imajo NoSQL podatkovne baze svoje slabosti. Največja od njih je dejstvo, da podatki med seboj lahko niso usklajeni. To lahko odpravimo z izbiro pravilne ravni usklajenosti. Posledično se zmanjša hitrost procesiranja zahtev oziroma transakcij. Pomemben podatek je, da NoSQL baze pridobijo na svoji veljavi šele, ko delujejo v gruči. En sam strežnik v gruči praktično izgubi vse prednosti, ki nam jih ponujajo NoSQL baze.

Očitna razlika med klasičnimi relacijskimi bazami in NoSQL bazami je nivo normalizacije podatkov. Pri relacijskih bazah so v večini primerov podatki normalizirani. Podatki v NoSQL bazi so denormalizirani. To nam omogoča hitrejše izvajanje kompleksnejših transakcij. Na splošno so NoSQL podatkovne baze veliko nasprotje klasičnim relacijskim bazam.

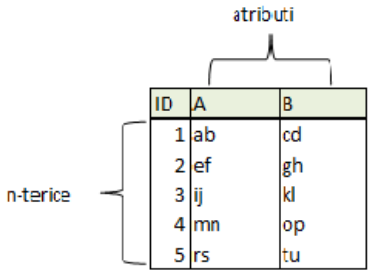
Motiv diplomske naloge sta raziskovanje ter primerjava teh dveh zelo nasprotujočih si pristopov v realizaciji podatkovnih baz. Želja je podati korektne in praktično dokazane trditve, ki bodo nazorno pokazale prednosti in slabosti obeh sistemov hrambe podatkov.

## 1.2 Relacijski podatkovni model

Relacijski podatkovni model je sistem smiselno poimenovanih in povezanih tabel. Vsaka tabela je osnovana na matematičnih funkcijah in se imenuje relacija. Ena relacija je podmnožica kartezijskega produkta zalog vrednosti atributov, ki sestavljajo relacijo. Relacijski podatkovni model uvrščamo med površinske modele, saj z njim predstavimo baze na logičnem nivoju.

Vsaka od relacij ima relacijsko shemo, v kateri so opisani atributi. Pod njo se nahajajo vrstice, ki vsebujejo podatke, in se imenujejo n-terice. Vsaka n-terica predstavlja entiteto ali povezavo entitet.

Vsakemu atributu pripada določena domena, ki opredeljuje njegovo zalogo vrednosti, ki jih lahko zavzame atribut. Atribut je preslikava množice objektov v pripadajočo domeno.




Slika 1: Primer relacije

Pri vsaki relaciji lahko govorimo o stopnji relacije in o njeni moči. Stopnja je število atributov, ki jih hrani relacija. Moč relacije določa število n-teric. Slednja se v aktivnih bazah zelo spreminja.

Pri relacijskih modelih govorimo o funkcionalnih odvisnostih. Te nam služijo za opredelitev odvisnosti med atributi relacije. Ob dejstvu, da poznamo ključ, lahko pridobimo ostale vrednosti. Ključi so lahko sestavljeni iz enega ali več atributov. Pomembno je, da ključ enolično določa eno n-terico. Enoličnost pomeni, da se ena kombinacija ne more pojaviti v nobeni drugi n-terici. Zraven imamo lahko tuji ključ. Ta nam omogoča lažje iskanje n-teric in ni nujno, da je enoličen.

Za dostop, dodajanje, brisanje, preverjanje in spreminjanje podatkov se v relacijskem podatkovnem modelu uporabljajo transakcije. Transakcija je vsak zunanji dostop do podatkovne baze. Vsaka transakcija mora upoštevati načela, ki skrbijo za dostop do podatkovnih baz:

- atomarnost,
- konsistentnost,
- izolirana,
- trajnost.

Atomarnost transakcije poskrbi, da se cela transakcija izvede v celoti. Transakcija se ne sme prekiniti med izvajanjem ter pustiti svoje naloge nedokončane. Če se v izrednih primerih zgodi, da se izvajanje transakcije prekine, je treba podatke vrniti na stanje, kot je bilo pred začetkom izvajanja transakcije.

V podatkovni bazi moramo poskrbeti za konsistentnost podatkov. Vsaka transakcija pretvori bazo iz enega veljavnega stanja v drugo veljavno stanje. Če transakcija upošteva integritetne omejitve, se izvede, sicer je zavrnjena.

Na primer imamo relacijo »Račun« ter relacijo »artikliZaRacun«. Povezani sta s povezavo ena proti mnogo (1, ∞) prek primarnega ključa »racunID«. Če pobrišemo zapis v prvi relaciji, moramo posledično pobrisati tudi zapise v drugi relaciji, sicer lahko pride do napake, ko želimo prebrati podatke o računih za točno določen artikel.

Transakcije morajo biti izolirane. Pri vzporednem izvajanju večjih transakcij ne sme priti do popačenja podatkov. To se zgodi, ko želi ena ali več transakcij spreminjati podatke. Lahko rešujemo že nastali konflikt, ali pa uvedemo algoritem, ki preprečuje konflikt. Poleg tega ostalim transakcijam ne sme biti viden rezultat, dokler le-ta ne zaključi z izvajanjem.

Pomembna lastnost transakcij v podatkovni zbirki je tudi trajnost. O tem govorimo takrat, ko se neka transakcija izvede in podatkovne baze ne moremo več vrniti v prejšnje stanje, oziroma je to zelo težko narediti. Po uspešno izvedeni transakciji so učinki transakcije na podatke trajno shranjeni.

Za zagotavljanje vseh štirih lastnosti transakcij skrbi tako SUPB kot tudi programer.

Najbolj razširjen poizvedovalni jezik za relacijske podatkovne baze je SQL.

### 1.2.1 Kaj je SQL?

SQL (Structured query language) se je razvil z namenom upravljanja podatkovnih baz. Razvil ga je IBM leta 1974 z namenom upravljanja podatkovnih zbirk v podjetju. Na začetku so ga poimenovali SEQUEL (Structured English query language). Kasneje so ga preimenovali v SQL, ker je SEQUEL že bilo tovarniško zaščiteno ime v drugem podjetju. Prva organizacija, ki je SQL priznala kot standard, je bila ANSI (American national standards institute). Leto kasneje (1987) je SQL priznala še ISO organizacija (International organization for standardization). Dandanes za standardizacijo skrbi ISO/IE JTC (Joint technical committee). To je posebna skupina v organizaciji ISO, ki skrbi za standarde v elektronskih in digitalnih tehnologijah. Na začetku je bil SQL zelo osiromašen in z malo funkcijami. Z novimi različicami so dodajali nove funkcionalnosti:

- rekurzivne poizvedbe,
- sprožilce,

- avtomatsko generirane vrednosti,
- definirani dostopi s pomočjo XML zapisov,
- definirani sortirani stavki,
- boljša manipulacija nad nizi
- itd.

Iz osnovnega jezika SQL so nastale različne implementacije:

- PostreSQL,
- MySQL
- Microsoft SQL Server,
- Oracle,
- DB2,
- Informix.

Danes imajo podatkovne baze SQL ter vse njegove implementacije še vedno vodilno vlogo pri upravljanju s podatki.

### 1.3 Zakaj NoSQL?

Prvo vprašanje, ki se poraja, je, kaj bi sploh lahko bilo narobe z relacijskimi podatkovnimi bazami, ki jih uporabljamo več kot tri desetletja. Pred 10 leti so celo govorili, da so podatkovne baze doživele svoj zaključek razvoja. Relacijske podatkovne baze so veljale za najboljšo tehnologijo na področju hranjenja podatkov. Imele so najhitrejši dostop do podatkovnih baz, relativno enostavne poizvedbe itd.

Ker očitno ni nič narobe z relacijskimi bazami, se poraja vprašanje, zakaj preiti na NoSQL podatkovno bazo. Odgovor se skriva v potrebi po obvladovanju zelo velikih količin podatkov. To so sistemi, ki jih uporabljajo podjetja, ki ponujajo storitve, ki morajo obvladovati ogromne količine podatkov:

- Google,

- Amazon,
- Facebook,
- eBay.

Omenjena podjetja v vsaki sekundi izvedejo več milijonov poizvedb. Že to predstavlja velik problem pri dostopu do podatkovne baze oziroma procesiranju zahtev. Stanje poslabša tudi dejstvo, da želimo izvajati sočasno več poizvedb nad istimi podatki. To pa seveda poslabša hitrost delovanja. V procesorju lahko naredimo navidezno vzporednost in s tem navzven ustvarimo vtis vzporednega delovanja procesov. A to ni rešitev, saj se v procesorju poizvedbe izvajajo v časovnih rezinah. Vsak ukaz se izvaja določen čas ter potem preklopi na drug ukaz. Vse to se dogaja ciklično. Ker vseskozi prihajajo nove poizvedbe, procesor ne more izvesti vseh operacij. Čakalne vrste se povečujejo in odzivnost se zmanjša.

Ne smemo pozabiti, da je poleg tega treba skrbeti za atomarnost, konsistentnost, izoliranost in trajnost v podatkovni bazi. To še upočasnjuje izvajanje transakcij nad podatkovno bazo. Več transakcij se izvaja v sistemu, večja je verjetnost, da bo treba skrbeti za podatkovno bazo in razreševati zgornja štiri pravila. To vse še povečuje čas izvajanja transakcij.

Delovanje celotnega sistema lahko izboljšamo s posodobitvijo strojne opreme podatkovnega strežnika. Nadgradimo lahko:

- procesor,
- trdi disk,
- pomnilnik,
- vodila med enotami
- itd.

Procesor lahko nadgradimo tako, da uporabimo večjedrno verzijo oziroma več procesorjev. Novejši procesorji imajo danes vgrajene različne strojne dodatke, ki poskrbijo za hitrejše izvajanje ukazov. Poleg tega je razširitev na večjedrno različico, boljša z več stališč. Največja prednost večjedrnih procesorjev je, da lahko sistemski ukazi tečejo na enem jedru, programska oprema pa na preostalih jedrih. Hitrost delovanja procesorja je odvisna od takta, v katerem se izvajajo ukazi, in od hitrosti prenosa.

Pri trdih diskih lahko nadgradimo kapaciteto ali zmanjšamo čas dostopa do podatkov. Pri zapisovanju v podatkovno bazo je bolje, da imamo večjo kapaciteto, ker algoritmi laže najdejo prazen prostor, v katerega lahko zapišejo podatke. Pomembna stvar, za katero je treba skrbeti na disku, je tudi ureditev podatkov. Najbolje je, da so podatki urejeni na način, kjer so podatki, ki jih velikokrat iščemo v sklopu kratke časovne rezine, zapisani skupaj. Za čim hitrejšo obdelavo podatkov je pomemben čas dostopa. To je povprečen čas, ki ga porabi disk, da priskrbi podatke za obdelavo. Pri trdih diskih je to seštevek:

- iskalnega časa,
- vrtilne zakasnitve (latenca),
- časa prenosa podatkov.

Iskalni čas je podatek, ki nam pove, koliko časa potrebuje bralno-pisalna glava, da pride nad zahtevani valj. Vrtilna zakasnitev je čas, ki ga potrebuje glava, da se postavi nad izbrani izsek ter prične z branjem ali pisanjem. Čas prenosa podatkov je čas, ki ga potrebujemo, da se prebrane podatke dekodira in jih spravi na vodilo.

Pri diskih je pomembna hitrost branja ter pisanja. Običajno je hitrost branja mnogo hitrejša od hitrosti pisanja. V današnjem času so diski najpočasnejša enota v računalniku. Zaradi tega se danes vse bolj uveljavljajo SSD (Solid state disc) diski. Zaradi flash tehnologije se hitrost pisanja in branja občutno izboljša. SSD diski so tudi mnogo bolj energetsko varčni.

Slabost SSD diskov je visoka cena, a se ta vztrajno zmanjšuje. Kmalu bodo postali dostopni za večje podatkovne baze.

Pri izvajanju posameznih transakcij sta pomembna tudi hitrost in velikost pomnilnika. Če imamo velik pomnilnik, lahko v njem hranimo veliko podatkov. Največkrat uporabljene podatke imamo shranjene v pomnilniku. Ker je čas dostopa do pomnilnika mnogo manjši kot do diska, lahko posledično hitreje obdelujemo podatke. Glede na to, kaj želimo početi s podatki, lahko v pomnilniku ustvarimo strukturo:

- sklada,
- vrste,
- povezovalnega seznama,
- razpršene tabele,

- drevesa
- itd.

Poleg posodabljanja posameznih komponent je priporočljivo posodobiti tudi vodila. Ta lahko izboljšamo tako, da postavimo več vodil vzporedno med komponente oziroma uporabimo materiale, ki izboljšajo hitrost prenosa. V zadnjih letih se vse bolj uveljavljajo serijske povezave. Primer take povezave je PCI Express 16. Ta ima 16 neodvisnih serijskih poti.

A kljub veliko možnostim pohitritev strojne opreme obstaja še vedno problem nedohajanja današnjih trendov podatkovnih baz (želimo hraniti vse večje količine podatkov in želimo izvajati vse več transakcij). Ena od možnosti pohitritve delovanje podatkovne baze je razdelitev na več strežnikov. Arhitektura pri povezavi več strežnikov v gručo je običajno gospodar-suženj. Pri razdelitvi podatkovne baze na več strežnikov nastane problem pri:

- pisanju,
- spreminjanju,
- brisanju.

Zaradi dejstva, da se podatkovna baza nahaja na več strežnikih, je treba vse skupaj sinhronizirati. Pri tem pa nastaja še večja obremenitev sistema. Za vse to poskrbi gospodar. Ko v bazi pišemo, spreminjamo ali brišemo podatke, želimo, da bodo vse naslednje transakcije imele na voljo zadnjo različico. Pri sinhronizaciji med več diski je treba paziti, da se izvede čim hitreje, oziroma se zapiše zaznamek, ki pove, da je prišlo do spremembe.

Celoten sistem lahko zelo hitro deluje, če ne kličemo kompleksnih transakcij. Pod te transakcije se štejejo na primer transakcije, ki imajo veliko stikov (naravnih, levih, desnih) zapisanih v veliko n-tericah. Takih problemov se rešimo s pomočjo denormalizacije. Pri tej metodi se soočimo s konsistenco podatkov. En podatek je treba spreminjati na več različnih lokacijah podatkovne baze. Problem je tudi, ker lahko pri brisanju izgubljamo podatke, ki si jih ne želimo izgubiti. Pri spreminjanju se lahko zgodi, da moramo podatke spreminjati na več lokacijah.

Za reševanje problemov, ki se pojavljajo v relacijskih podatkovnih bazah, so že leta 1998 razvili NoSQL baze.

### 1.3.1 Zakaj NoSQL še vedno ni v množični uporabi?

Prvi razlog, zaradi katerega NoSQL še vedno ni v množični uporabi, je sama filozofija, na kateri temeljijo NoSQL podatkovne baze. Programerji, informatiki ter vsi, ki se srečujejo s hrambo podatkov, imajo raje klasične relacijske podatkovne baze, ker:

- so se jih učili,
- znajo z njimi delati,
- poznajo trike,
- so enostavne za razumevanje,
- v ozadju temeljijo na matematičnih osnovah
- itd.

Drugi razlog, zaradi katerega NoSQL podatkovne baze še vedno niso v množični uporabi, so tudi interesi velikih podjetij, ki se ukvarjajo z razvojem relacijskih podatkovnih baz, v katere so vložila veliko sredstev. Ta podjetja beležijo zelo velike zaslužke in jim ni v interesu, da bi jih kdo drug izpodrinil s trga ter jim vzel zaslužek. Primeri takih podjetij so:

- Oracle,
- Microsoft,
- IBM
- itd.

Tretji razlog, zaradi katerega NoSQL podatkovne baze niso naredile revolucije v podatkovnih bazah je dejstvo, da so običajne relacijske podatkovne baze še vedno zelo učinkovite v primerih, ko:

- nimamo relacij z veliko n-tericami,
- ne izvajamo kompleksnih operacij,
- ne izvajamo več milijonov poizvedb v sekundi,
- želimo konsistentne podatke ves čas

- itd.

## 1.4 Opis NoSQL podatkovnih baz

NoSQL podatkovne baze imajo nekaj zakonitosti v svojem bistvu, kot so:

- decentraliziranost,
- prožnost,
- razdeljenost,
- tolerantnost do napak,
- razpoložljivost,
- orientiranost v vrstice,
- konsistentnost.

O decentraliziranosti in razdeljenosti govorimo v primeru, ko imamo podatke razdeljene na več strežnikih. NoSQL podatkovne baze lahko tečejo samo na enem strežniku, a potem ne izrabljajo vseh možnosti, kot jih ponujajo.

Prožnost sistema nam pove, koliko se sistem upočasni, ko na njem izvajamo večje število transakcij. Pri tem poznamo vertikalno in horizontalno povečevanje strojnega sistema. Vertikalno povečujemo strojni sistem takrat, ko nadgrajujemo obstoječi strežnik (dodajamo več spomina, nadgradimo procesor itd.). Horizontalna povečava je nadgrajevanje sistema in poleg obstoječega strežnika dodamo še nove strežnike. Ta povečava je mnogo bolj kompleksna, ker moramo potem skrbeti za sinhronizacijo med posameznimi členi. NoSQL baze imajo veliko zmožnost prožnosti, ker enostavno dodajamo in odstranjujemo nove strežnike. Ko želimo povečati zmožnosti sistema, dodamo nov strežnik (horizontalna povezava). Programska oprema poskrbi, da se naredi še ena identična kopija na novem strežniku. Takoj ko je kopija narejena, že lahko uporabljamo nov člen (strežnik) in tako razbremenimo ostale. Pri tem ni treba bistveno spreminjati celotnega vozlišča.

NoSQL baze so tudi zelo razpoložljive. Imajo zelo veliko zmožnost nadaljevanja izvajanja transakcij kljub nesrečam. Nesreče so na primer udar strele, poplave v območju strežnika, prekinitev glavne podatkovne povezave, požari itd. Podatkovni sistemi morajo biti v današnjem času na voljo vsak trenutek. Nesreče ne smejo biti izgovor za nedelovanje. Vse te

katastrofe finančno zelo prizadenejo podjetja, razen tega pa sam sistem izgubi na zaupanju uporabnikov. NoSQL baze ob nedelovanju enega strežnika avtomatsko začnejo izvajati ukaze na drugem strežniku. Na ta način uporabniki ne občutijo in se ne zavedajo, da del gruče ne deluje.

Cel sistem NoSQL baz je zelo toleranten do napak. Ko se določen strežnik pokvari, ga lahko enostavno zamenjamo z novim brez ponovnega zagona. Na novi člen prenesemo pripadajoče podatke ter omogočimo delovanje strežnika v gruči.

NoSQL baze imajo tudi lastnost, da so orientirane v vrstice. Po nekaterih trditvah naj bi bile sicer orientirane v stolpce, odvisno, kako gledamo na razporeditev. Na splošno velja, da so NoSQL baze nekje med vrstično in stolpčno orientacijo. NoSQL baze imajo podatke urejene tako, da ima vsaka vrstica svoje stolpce. Pri tem ni treba, da so v eni tabeli povsod isti stolpci. To je zelo velika prednost glede na relacijske podatkovne baze. Seveda mora imeti vsaka vrstica unikaten ključ, da lahko natančno določimo podatke. Dejansko so podatki shranjeni v večdimenzionalni tabeli, za katero nikoli ne vemo, kateri atributi so v posamezni vrstici. Atribute lahko brez problemov dodajamo in brišemo.

Konsistentnost v podatkovni bazi nam pove, kako hitro so vidne spremembe. Če uporabnik naredi spremembo v podatkovni bazi, moramo vedeti, kako hitro bo ta sprememba vidna tudi na ostalih strežnikih. Večina NoSQL baz ponuja možnost nastavitve ravni konsistentnosti. Tako lahko kadarkoli brez večjih problemov spremenimo konsistentnost. V splošnem poznamo 3 najbolj znane vrste konsistentnosti:

- stroga konsistenca,
- vzorčna konsistenca,
- slučajna konsistenca.

Strogo konsistentnost uporabimo, kadar želimo, da je podatek zagotovo zadnji in pravilen. Ta konsistenca je z vidika uporabnika najboljša, a se je treba vprašati, ali jo res nujno potrebujemo. Če želimo to doseči, moramo imeti vse strežnike sinhronizirane na isti čas. Ob vsaki transakciji moramo pogledati, če je bila narejena kakšna sprememba na katerem od strežnikov. Druga možnost je, da ob spreminjanju podatke zaklenemo na vseh strežnikih in počakamo, da se sprememba izvede. Potem pošljemo spremembo na vse strežnike ter nato strežnike odklenemo za ponoven dostop do podatkov. Ta način porabi veliko časa, česar si ne želimo.

Nekje vmes med strogo in slučajno konsistenco se nahaja vzorčna konsistenca. Kaj je ta konsistenca, je prikazano skozi primer:

- proces 1 piše vrednost v spremenljivko 'x',
- sočasno proces 2 bere vrednost spremenljivke 'x' in piše vrednost v spremenljivko 'y'.

Tu lahko pride do problema, ker je lahko vrednost, ki jo želimo pisati v spremenljivko 'y', odvisna od vrednosti 'x', katero pa smo prebrali z napačno vrednostjo.

Pri slučajni konsistenci vemo, da bodo vse spremembe prišle do ostalih strežnikov po določenem času. Tega časa ne poznamo. Edino, kar vemo, je, da bodo vsi strežniki dobili spremembo.

Seveda je stopnja izbire konsistence zelo odvisna od drugih dejavnikov – števila strežnikov ali želje po aktivnosti poizvedb ves čas. O tem govori CAP teorem, ki je bolj natančno razložen v enem od poglavij.

### 1.4.1 Pregled NoSQL podatkovnih baz

Tako kot imamo pri relacijskih podatkovnih bazah več podjetij, ki skrbijo za razvoj ter trženje, imamo pri NoSQL podatkovnih bazah vse večjo izbiro podatkovnih baz. Primeri teh so:

- Dynamo,
- SimpleDB,
- BigTable,
- CouchDB,
- Neo4J,
- Cassandra.

#### 1.4.1.1 Dynamo

Dynamo so razvili v Amazonu in predstavlja enega od največjih njihovih projektov. Amazon ob največji obremenjenosti streže nekaj milijonov uporabnikom na nekaj deset tisoč strežnikih s potrebnimi podatki. Da lahko to obdelajo brez težav, potrebujejo tehnološko dovršen sistem

hrambe podatkov. Vse to nakazuje, da potrebujejo sistem, ki je kar najbolj decentraliziran in je zmožen obdelave velikih količin podatkov. Ob tako velikem številu strežnikov je zelo velika verjetnost, da pride do napake na katerem od njih. V primeru napake ali odpovedi strežnika uporabniki tega ne smejo opaziti oziroma se podatki ne smejo izgubiti.

Dynamo je najbolj tipičen predstavnik koncepta ključ-vrednost. Vsak ključ ima podano vrednost. Pri algoritmih poznamo to kot podatkovno strukturo »slovar«.

Vsi strežniki v gruči imajo popolnoma identične podatke. Z vidika konsistentnosti je vseeno, na kateri strežnik se bo odjemalec priključil. Algoritem, ki določi, na kateri strežnik se bo odjemalec priključil, je zelo kompleksen in učinkovit.

#### 1.4.1.2 SimpleDB

SimpleDB je razvil Amazon tudi za lastne potrebe. Želji razvijalcev sta bili decentraliziranost zbirk po celotnem svetu ter neprestana dostopnost.

Velika prednost SimpleDB je nudenje možnosti razvijalcu, da se osredotoči na aplikacijo in mu ni treba skrbeti za administracijo podatkov ter celotnega sistema. Vse, kar se dogaja v ozadju procesiranja, ni pomembno za razvijalca. Prednost SimpleDB je tudi dejstvo, da se podatki sami indeksirajo ter tako poskrbijo za hitrejše izvajanje zahtev.

#### 1.4.1.3 BigTable

BigTable je razvil Google leta 2004. Razvil ga je z namenom lastne uporabe, zato ni odprte narave. Lahko ga uporabljamo samo v sklopu Googlovih storitev, kot so:

- Maps,
- Reader,
- Gmail,
- Youtube,
- Earth,
- zgodovina iskanja
- itd.

Do podatkov dostopamo prek dveh tipov podatkov in se imenujeta vrstični in stolpčni podatek. Vsak od teh dveh podatkov ima še označbo časovne značke, ki nam pove, kdaj je bil podatek zapisan v bazo, oziroma, kdaj je bila narejena zadnja sprememba.

#### 1.4.1.4 CouchDB

CouchDB je odprtokodni sistem, ki za delovanje uporablja JSON in JavaScript. Razvit je bil pod okriljem Apache. S podatkovno bazo komunicira s pomočjo HTML ukazov »get«, »post«, »put« in »delete«.

#### 1.4.1.5 Cassandra

Cassandra je bila razvita v Facebooku. Ime je dobila po junakinji iz grške mitologije, ki naj bi imela zmožnost gledanja v prihodnost. Razvila sta jo Prashant Malik in Avinash Lakshman. Slednji je bil tudi član ekipe, ki je razvila NoSQL podatkovni sistem Dynamo. Uradno je bila ideja razvita leta 2008 v projektu, imenovanem »Google code«. Leto kasneje je projekt prišel pod okrilje Apache. Tam je bil zapisan kot potencialno zelo zanimiv projekt in vanj so vložili veliko truda in denarja. Leta 2010 je bil projekt končan in začeli so ga uporabljati v socialnem omrežju Facebook.

Cassandra ima osnovo logiko podobno kot Dynamo.

Cassandra je zelo primerna za:

- velike projekte, katerim se vseskozi dodajajo funkcionalnosti,
- velike projekte, na katerih dela zelo veliko ljudi,
- izdelavo statističnih podatkov,
- izvajanje velikega števila sprememb v podatkovni bazi,
- sistem, v katerem morajo biti strežniki razporejeni po širšem geografskem območju,
- itd.

Cassandro zaradi njenih prednosti oziroma značilnosti kljub njeni kratki zgodovini uporablja kar nekaj podjetij:

- Twitter (za analize),

- Facebook (za iskanje),
- Cisco/platform64 (televizijska slika na mobilnih napravah),
- Digg (skladiščenje zadnjih sprememb),
- Rackspace (računalništvo v oblaku, prijavljanje v sistem),
- Reddit (za trajno skladiščenje podatkov),
- SimpleGeo (skladiščenje podatkov o trenutni lokaciji)
- itd.

### 1.5 CAP teorem

CAP(consisteny, availability, partiton tolerance) je bil leta 2000 predstavljen na simpoziju, dve leti kasneje pa so ga označili kot teorem. Včasih CAP teoremu pravimo tudi Brewerjev teorem (avtor je Eric Brewer). Teorem govori o treh različnih lastnostih, ki jih želimo imeti v gruči:

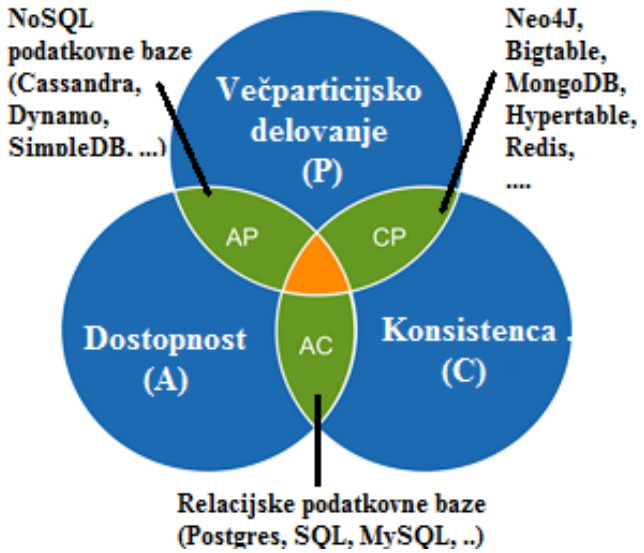
- konsistenca,
- dostopnost,
- večparticijsko delovanje.

Konsistenca nam pove, v kolikšni meri lahko pričujemo točnost oziroma aktualnost prebranega podatka. Z različnimi strategijami konsistence lahko točnost oziroma aktualnost podatka spreminjamo.

Dostopnost nam pove, ali so podatki vseskozi dostopni za branje in pisanje.

Večparticijsko delovanje je razdeljevanje podatkovne baze na več vozlišč.

Teorem govori o tem, da sta v enem sistemu implementirani le dve od treh lastnosti. Tiste dve lastnosti, ki ju želimo imeti, sta si med seboj vzajemno izključujoči.




Slika 2: Grafična predstavitev CAP teorema s podanimi najbolj znanimi predstavniki

V prvi skupini so relacijske podatkovne baze na enem strežniku. Te so vedno dostopne za branje in pisanje ter vedno konsistentne. Njihova največja napaka je nezmožnost razdelitve v gručo, pri kateri so vsi strežniki med seboj neodvisni. Vedno mora biti vsaj en glavni strežnik.

Druga skupina so v večini podatkovne baze, ki jih je razvil Google za svoje potrebe. Te podatkovne baze so vedno konsistentne ter imajo zmožnost večparticijskega delovanja. Ima pa ta skupina podatkovnih baz problem, ker ni vedno dostopna. Če pride do problema v katerem od strežnikov, se lahko zgodi, da del podatkov postane nedostopnih.

Tretja skupina so podatkovne baze, v katere spada tudi Cassandra. V tej skupini je najbolj pomembno, da so podatki vedno na voljo. Celoten sistem omogoča večparticijsko delovanje. Lahko se zgodi, da podatek, ki nam ga vrne podatkovna baza, ni pravilen – nima pravilne vrednosti. Če poskrbimo za hitre internetne povezave med posameznimi členi, nam podatkovna baza zelo malokrat vrne nepravilno vrednost – neaktualno vrednost.

### 1.6 Konsistentni nivoji v Cassandri

Pri konsistentnih nivojih v vrstičnih družinah (angl. column family) moramo v prvi vrsti razlikovati med branjem in pisanjem. Konsistentni nivo podaja razmerje med konsistenco podatkov ter hitrost delovanja celotnega sistema.

Pri branju imamo naslednje konsistentne nivoje:

- nič (angl. zero),
- katerikoli (angl. any),
- eden (angl. one),
- sklepčen (angl. quorum),
- sklepčen - nadgrajen (angl. dcquorum),
- vsi (angl. all).

Konsistentni nivo »nič« vse podatke zapisuje v ozadju. Je najhitrejši, a nikoli se ne moremo zanesti na pravilnost podatkov. Ta nivo je praktično skoraj neuporaben.

Konsistentni nivo »katerikoli«, je bil predstavljen premierno v Cassandri verzije 0.6. Deluje po principu, pri katerem podatke zapiše za katerikoli strežnik.

Konsistentni nivo »eden« zagotovi, da se zahteva zapiše v vse strežnike v spominsko tabelo in naredi zapis v datoteko s spremembami (angl. commit log).

Konsistentni nivo »sklepčen« deluje na principu, da najprej pogleda, na koliko strežnikih se pojavlja trenutni imenski prostor. To število potem deli s številom 2 ter prišteje 1. Enačba za izračun števila:

$$limit = \frac{število\_strežnikov}{2} + 1$$

Število, ki ga strežnik dobi, je limit, ki nam pove, po koliko uspešno obdelanih strežnikih nam operacija vrne podatek o uspešnosti. Strategija »sklepčen-nadgrajen« deluje zelo podobno, le da tu vzame v izračun več parametrov.

Zadnji nivo pri pisanju je »vsi«. Ta nivo je najbolj zanesljiv, ko želimo imeti v vsakem trenutku točne podatke. Če slučajno eden od strežnikov v gruči ni zmožen izvesti operacije, nam vrne podatek o neuspešnosti in cel ukaz se razveljavi. V realnosti je ta sistem skoraj neuporaben, če imamo v gruči zelo veliko strežnikov.

Pri branju pa imamo naslednje nivoje konsistence:

- eden (angl. one),
- sklepčen (angl. quorum),
- vsi (angl. all).

Konsistentni nivo »eden« vrne podatek s prvega strežnika, ki odgovori na zahtevo. Pri tem imamo lahko velike probleme, ker podatek ni nujno pravilen oziroma aktualen.

Konsistentni nivo »sklepčen« vpraša za rezultat vse strežnike v gruči. Kot rezultat pa nam vrne rezultat, ki ima časovni žig najmlajši od tega trenutka – je najbolj aktualen.

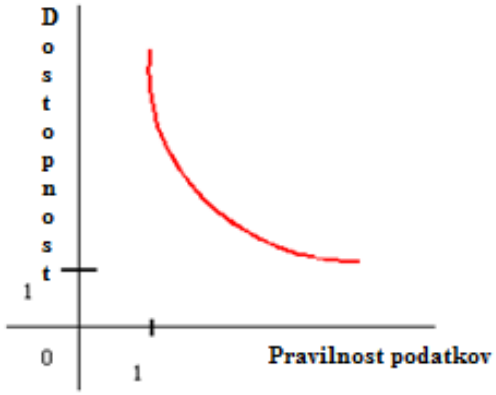
Konsistentni nivo »vsi« deluje po istem principu kot »sklepčen«, le da v primeru, ko se en strežnik ne odziva na zahtevo, celotna operacija propade. V realnih sistemih je ta nivo neuporaben, saj bi velikokrat operacija propadla.

Konsistentni nivo zbiramo glede na dejstvo, kakšno pravilnost oziroma natančnost podatkov želimo doseči. Glede na dejstvo, ali nam je bolj prioritetna točnost ali hitrost izvajanja operacij, lahko izberemo konsistentni nivo.

Na primer v socialnem omrežju je vseeno, če bo pridobljen podatek z malo zamude, oziroma podatek ne bo stoodstotno pravilen. Pri bančnih storitvah pa je situacija obratna. Zelo je pomembno, da vedno dobimo pravilen rezultat in da se celotna transakcija v celoti izvede.

Pri nivojih, v katerih moramo čakati, da vsi odgovorijo, moramo postaviti razumem čas čakanja. To lahko določimo v konfiguracijski datoteki »cassandra.yaml« zapisano v spremenljivki »rpc_timeout_in_ms«. Glede na trenutne hitrosti interneta in obremenjenosti omrežij se vrednost običajno nastavi na 10000. To je čas čakanja 10 sekund.

Hitrost operacij in pravilnost podatkov sta si obratno sorazmerna. Ob povečevanju enega se drugi zmanjšuje, ter obratno.




Slika 3: Grafični prikaz obratne sorazmernosti

Za potrebe testiranja podatkov je bil pri branju in pisanju uporabljen konsistenčni nivo »eden«. Izbran je bil zaradi dejstva, da večina aplikacij, ki tečejo na Cassandri, uporablja ta niv

# Poglavje 2

# Primerjava MySQL in NoSQL podatkovnih baz

## 2.1 Primerjava hrambe podatkov med Mysql in Cassandro

V MySQL in ostalih relacijskih podatkovnih bazah se hranijo podatki na strežnikih v podatkovnih bazah. Vsaka podatkovna baza je naprej razdeljena na relacije, vsaka relacija vsebuje n-terice, v katerih so zapisani podatki, ki imajo svoj ime glede na atribut. V relacijskih podatkovnih bazah je v primeru pravilne normalizacije vse zapisano v dvodimenzionalnem prostoru. Človek tako lahko zelo hitro dojame in razbere potrebne podatke. Prav zaradi tega je pisanje poizvedb v relacijskih podatkovnih bazah zelo enostavno:

- »select« vrstica nam pove, katere atribute želimo izpisati,
- »from« vrstica nam pove, iz katere relacije želimo pridobiti podatke,
- »where« vrstica nam pove, katere n-terice želimo izpisati.

Zaradi enostavnosti si razvijalec lahko še pred pisanjem poizvedbe predstavlja, kaj mu bo podatkovna baza vrnila za rezultat. Zelo podobno je tudi pri stavkih, ki brišejo, urejajo ter dodajajo podatke.

V Cassandri imamo namesto podatkovne baze na posameznem strežniku imenske prostore (angl. keyspaces). Ti se nahajajo na enem ali več strežnikih. Na koliko strežnikih se bo nahajal imenski prostor, se določi že pri ustvarjanju imenskega prostora. Potrebna je pozornost, da ni nastavljeno večje število imenskih prostorov, kot je strežnikov, priključenih v gručo. V primeru kasnejšega dodajanja novih strežnikov v gručo lahko še vedno popravljamo in povečujemo število kopij na strežnikih.

Naslednja stvar, ki jo je treba določiti je strategija pridobivanja podatkov. Obstajajo naslednje strategije:

- enostavna strategija,
- stara internetna topologija,
- internetna topologija.

Enostavna strategija je najhitrejša, saj podatke naenkrat obdeluje samo na enem strežniku. Eno operacijo lahko naenkrat obdeluje le en strežnik.

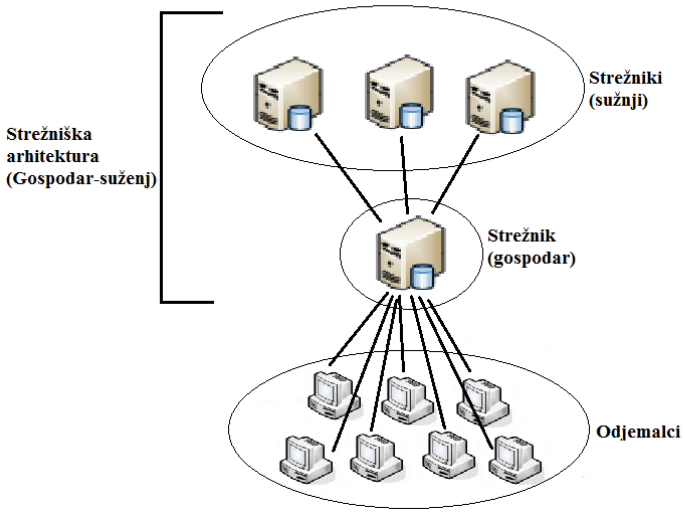
Interneta topologija priskrbi podatke iz drugega strežnika, medtem ko lahko trenutni strežnik nemoteno procesira svoje ukaze. Ta tehnologija je v uporabi v Cassandri šele od različice 0.7 naprej in je vključena že v osnovno konfiguracijo.

V vsakem imenskem prostoru so v Cassandri tako imenovane vrstične družine (angl. column family). Osnovni namen le-teh je enak kot relacije v relacijskih podatkovnih bazah. V njih so dejansko shranjeni vsi podatki.

## 2.2 Primerjava MySQL in Cassandra gruče

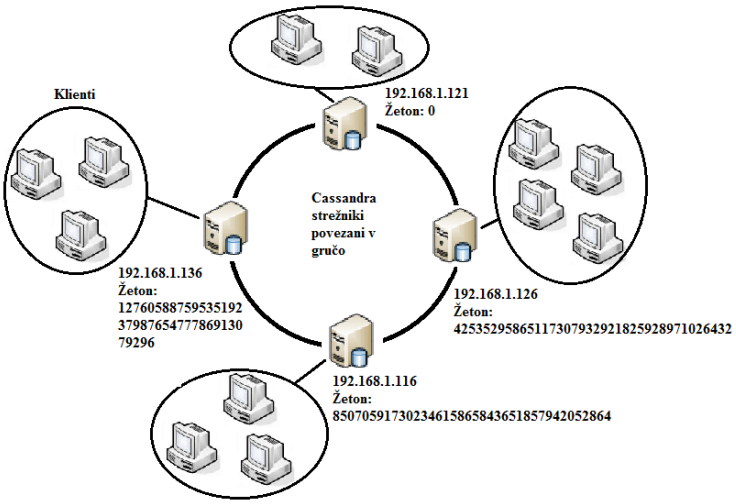
Gruči MySQL in Cassandra imata že v osnovi drugačno strežniško arhitekturo. To vpliva na uporabljene algoritme in protokole.

MySQL ima strežniško arhitekturo gospodar-suženj.




Slika 4: Strežniška arhitektura MySQL strežnika

Cassandra ima strežnike povezane v tako imenovani krog, v katerem so si vsi strežniki enaki ter si pomagajo pri opravljanju opravil. Vsi strežniki v gruči so si enakovredni.




Slika 5: Strežniška arhitektura Cassandra

Algoritmi, ki so vgrajeni v Cassandro, poskrbijo za razporejanje opravil med strežniki. Odjemalci lahko dostopajo do kateregakoli izmed strežnikov.

Velika prednost NoSQL baz glede na relacijske podatkovne baze je zmožnost delovanja brez uporabljene arhitekture gospodar-suženj. Pri arhitekturi gospodar-suženj nastane problem zaradi ozkega grla pri gospodarju. Vse poizvedbe morajo iti prek gospodarja. To ga lahko zelo obremeni. Pri NoSQL bazah so si vsa vozlišča (strežniki) identična. Tako imamo strežnike porazdeljene po širokem območju. Poizvedbe se izvajajo direktno samo na najbližjem oziroma na najmanj obremenjenem strežniku. S protokolom »peer to peer« (krajšano PTP) poskrbimo za prenos sprememb do ostalih strežnikov. Decentraliziranost ima prednost, saj ni treba skrbeti, kaj se dogaja z oddaljenim strežnikom. Pri NoSQL bazah so vsi strežniki v gruči identični in želeno spremembo lahko uveljavimo na vseh vozliščih.

# Poglavje 3

# Varnost

Pri vsaki napravi, ki jo priključimo na svetovno omrežje, je naprej treba poskrbeti za varnost. Nič drugače ni z NoSQL podatkovnimi bazami.

## 3.1 Varnost v NoSQL podatkovnih bazah

Varnost ni bila izpopolnjena vse od začetka tovrstnih podatkovnih baz. Na začetku je dostop do podatkovnih baz potekal brez avtentikacije. To je bilo zelo nevarno z vidika varnosti in praktično neuporabno za priklop na svetovno omrežje.

Kasneje so za varnost poskrbeli z mehanizmi avtentikacije z uporabniškimi imeni in gesli. Poskrbljeno je bilo zelo podobno, kot to poznamo v relacijskih podatkovnih bazah. Ko želi uporabnik dostopati do podatkov, mora vpisati uporabniško ime in geslo. Ta način ni najbolj varen, če ne kriptiramo gesla. V tem primeru lahko napadalec s pregledovanjem podatkov v omrežju pride do našega uporabniškega imena in gesla. Tega si ne želimo, zato so v NoSQL podatkovne baze vnesli tudi tako imenovano MD5 kriptiranje. To kriptiranje poskrbi, da niz znakov ne glede na dolžino pretvori v 128-bitno zgoščeno vrednost. Ta vrednost ima to lastnost, da najmanjša sprememba v nizu poskrbi za drugačno kriptirno vrednost. Iz te vrednosti je praktično nemogoče dobiti nazaj primarni niz znakov ob dejstvu, da je izbrano varno geslo, ki vsebuje:

- številke,
- posebne znake,
- velike črke,
- male črke.

Do primarnega niza lahko pridemo s tako imenovano »surovo močjo«. V tem trenutku to ni mogoče, ker bi računalniki (tudi najmočnejši) potrebovali za izračun nerealno veliko časa. Kljub temu je priporočljivo pogosto menjati gesla.

Ker večina uporabnikov uporablja zelo slaba gesla z vidika varnosti, so se v zadnjem času pojavile tako imenovane mavrične (angl. Rainbow) tabele. V tej tabeli so shranjeni nizi znakov ter njihov izračun MD5 vrednosti. To naredijo na način, da za vse možne različice

nizov izračunajo MD5 vrednost. Te potem shranijo v bazo, po kateri lahko kasneje iščejo. Tako pridejo do primarne vrednosti. Ker ne morejo izračunati vrednosti za dobra gesla, je zelo pomembno, da poskrbimo za izbor varnega gesla.

Pri NoSQL podatkovnih bazah, ki jih imamo razdeljene na več strežnikov, nastane problem pri izmenjavi podatkov med samimi strežniki. Pri tem je treba poskrbeti, da napadalec ne bo prišel do podatkov, ki se prenašajo med strežniki. Tovrsten napad lahko izvede napadalec z načinom mož v sredini (angl. man in the middle). To je eden najbolj znanih napadov za dostop do podatkov. Pred tem napadom so bile NoSQL podatkovne baze na začetku razvoja nemočne. Kasneje so razvili kriptirne algoritme, ki vse podatke kriptirajo. Če se napadalec vrine med strežnike, ne more ničesar početi s podatki, ker so kriptirani. Na oddajni strani se podatki kriptirajo, na sprejemni pa dekriptirajo. Za ta postopek potrebujemo ključe, ki jih poznata obe strani, napadalec pa ne.

Nekaj najbolj znanih kriptirnih algoritmov:

- RSA,
- SHA,
- AES.

## 3.2 Varnost v NoSQL podatkovni bazi Cassandra

Za varnost je odlično poskrbljeno tudi v Cassandri. Za implementacijo je potrebno v nastavitveno datoteko vpisati podatke, ki bodo omogočali uspešno avtentikacijo uporabnikov. Za vsak »keyspace« oziroma imenski prostor lahko določimo, kateri uporabniki lahko dostopajo do njega, ter njihova gesla. Cassandra omogoča tudi avtentikacijo s pomočjo MD5 kriptirnega gesla.

Za dostop do zaščitenih podatkov z zunanjimi tehnologijami je treba vključiti posebne knjižnice, ki skrbijo za pravilno avtentikacijo. Te knjižnice so praviloma že spisane v večini najbolj znanih programskih jezikov oziroma tehnologij.

Za zaščito pred napadom mož v sredini je bila Cassandra do pred kratkim zelo ranljiva. A z novimi različicami so tudi to uredili s kriptiranjem AES 128/256 in SHA algoritmom.

Cassandra je tudi zelo primerna za uporabnike, ki želijo implementirati svoje kriptirne algoritme. Omogoča enostavno implementacijo v samo programsko arhitekturo. Najlaže se to naredi z že vgrajenim programskim vmesnikom v programskem jeziku Java. To opcijo

predvsem veliko uporabljajo velika podjetja, ki so razvila svoje algoritme. Ti algoritmi niso splošno znani in jih je mnogo teže dešifrirati.

# Poglavje 4

# Priprava na testiranje

Za performančno analizo MySQL in Cassandre je bilo potrebno obe bazi primerjati z realnimi podatki v gruči.

## 4.1 Testno okolje:

Pred samim začetkom testiranja in primerjavo rezultatov med relacijsko podatkovno bazo in NoSQL podatkovnimi bazami je bilo treba namestiti 4 strežnike. Na vsak strežnik smo namestili operacijski sistem Linux Ubuntu, različice 12.4 (zadnja stabilna verzija). Ta operacijski sistem je dober predvsem zaradi odprtokodnosti. Odporokodnost nam omogoča vpogled in spreminjanje izvorne kode, ter posledničnо lažje popravljanje napak.

Na vse strežnike je bilo treba namestiti tudi naslednjo programsko opremo in pakete:

- Apache strežniški program,
- MySQL strežnik,
- Apache Cassandra 1.1.0,
- Java JDK.

Na strežnik, ki je bil namenjen izvajanju testnih scenarijev, je bilo poleg teh treba namestiti še:

- Mysql cluster(gruča),
- MySQL odjemalec,
- Hector-core 1.0,
- Apache Thrift 0.6.0,
- PHP,
- PHPCassa.

Apache strežniški program potrebujemo, ker na njem tečeta Apache strežnik in prevajalnik. Prevajalnik skrbi za prevajanje PHP kode. Njegova prednost je odprtokodnost, kar pripomore k hitrejšemu odpravljanju morebitnih napak. Na Apache strežniku teče več kot polovica vseh strežnikov v celotnem svetovnem omrežju in je posledično najbolj razširjen strežniški program.

MySQL strežnik je bil nameščen na vse strežnike zaradi namena izvajanja performančnih testov. Predstavnik relacijskih podatkovnih baz v testu je bil MySQL. Prednost MySQL je predvsem odprtokodnost ter velika razširjenost v spletu.

Apache Cassandra 1.1.0 se namesti na Apache strežnik. Trenutno je Apache edini strežnik, na katerega lahko namestimo Cassandro.

Da se Cassandri omogoči delovanje na strežniku, potrebujemo Javo JDK (angl. Java development kit). Paket je potreben, ker je skoraj celotna programska koda Cassandre napisana v Javi.

Poleg tega je za uspešno in učinkovito poganjanje testov na enega od strežnikov treba namestiti še dodatne programske pakete. S tega strežnika bomo poganjali teste oziroma scenarije. Imenovali ga bomo glavni strežnik.

Na glavni strežnik je poleg MySQL strežnika treba namestiti še programske pakete, ki skrbijo za razporejanje procesov po gruči.

Da je bil glavnemu strežniku omogočen dostop do podatkov v MySQL podatkovni bazi, je bilo treba namestiti MySQL odjemalec.

Za dostop do Cassandre se uporablja Hector-core 1.0. To je programski paket, ki ima že implementirane vse pomembnejše funkcije za dostopanje do podatkovne baze.. Poleg tega je treba za potrebe poganjanja testov namestiti tudi PHP. Tudi v PHP-ju je že napisan programski paket oziroma knjižnica, ki olajša dostop do podatkovne baze. Imenuje se PHPCassa in jo lahko na spletu dobimo brezplačno v odprtokodni različici.

Za prevajanje programske kode je treba namestiti še vmesnik Apache Thrift 0.6.0. Novejšo verzijo Thrift-a ni priporočljivo uporabljati, ker se v programski kodi nahaja tako imenovani »hrošč«, ki ga je zelo težko razrešiti.

## 4.2 Postavitev Cassandra strežnikov v gručo ter konfiguracija

Ko je nameščeno vse potrebno za postavitev strežnikov, je treba posamezne Cassandra strežnike med seboj povezati. To je najbolje narediti v konfiguracijski datoteki »cassandra.yaml« na vseh vozliščih. V datoteki so v parametru »seeds« vpisani IP naslovi vseh strežnikov, ki so povezani v gručo. V atributih »listen_address« in »rpc_address« je zapisan IP naslov naprave, na kateri teče določen strežnik. Cassandra prednastavljeno uporablja vrata (angl. port) številka 9160. Tega ni treba spreminjati, ker običajno teh vrat ne uporablja nobena druga aplikacija. Treba je določiti tudi tako imenovani žeton (angl. token). Cassandra podpira žetone od 0 pa vse do $2^{127}$. Vrednost žetona nam pove, kolikšen del gruče bo imel strežnik v lasti. V našem primeru smo žetone izračunali tako, da bo vsak od štirih strežnikov imel 25 % obremenjenosti. Najlaže je to izračunati s pomočjo enostavne funkcije, napisane v programskem jeziku Python:

```
MAX_RING_VALUE = 2**127

def tokens(stStreznikov):

tokens = []

for i in xrange(n):

 tokens.append(MAX_RING_VALUE/stStreznikov*i)

return tokens
```

Arhitekturo celotne gruče lahko pogledamo z ukazom: ./nodetool ring. Primer izpisa v naši gruči (zaslonska slika):

```
Address         DC          Rack        Status State   Load            Owns                Token
                                                                                           127605887595351923798765477786913079296
192.168.1.121   datacenter1 rack1       Up     Normal  165.91 MB       25.00%              0
192.168.1.126   datacenter1 rack1       Up     Normal  362.32 KB       25.00%              42535295865117307932921825928971026432
192.168.1.116   datacenter1 rack1       Up     Normal  100.41 MB       25.00%              85070591730234615865843651857942052864
192.168.1.136   datacenter1 rack1       Up     Normal  44.08 MB        25.00%              127605887595351923798765477786913079296
```

Slika 6: Prikaz aktualnih podatkov o gruči

S pomočjo teh podatkov se vidi IP naslov vsakega strežnika. V našem primeru smo imeli vse strežnike povezane v lokalnem omrežju. Pomembna stvar na tem izpisu je tudi status strežnika. Če je karkoli narobe na strežniku, oziroma v internetni povezavi do strežnika, se v

nekaj sekundah napiše »down« in tako hitro ugotovimo, na katerem strežniku se je pojavila napaka ter jo popravimo. Ostali strežniki v gruči med tem nemoteno delujejo naprej.

### 4.3 Priprava testnih podatkov

Ob predpostavljeni hipotezi, da se Cassansdra dobro obnese pri ogromni količini podatkov, smo na spletu poiskali podatkovno bazo, ki ima vnesene zapise za potrebe trgovine. Baza vsebuje podatke o:

- proizvajalcih,
- izdelkih,
- uvoznikih,
- ponudbi,
- osebah,
- kritikah,
- tipu produktov,
- opisu produktov.

Vseh zapisov je 25 milijonov, kar predstavlja zelo obsežno množico podatkov. S tako številnimi podatki bodo testi dobili smisel in bodo primerljivi tudi z realnimi problemi, ki jih srečujemo na področju podatkovnih baz.

#### 4.3.1 Priprava testnih podatkov za uvoz v MySQL

Za potrebe uvoza podatkov v MySQL podatkovno bazo smo imeli na voljo datoteke s končnico »sql«. Te je v bash-u oziroma ukazni vrstici relativno enostavno vnesti v podatkovno bazo z ukazom:

```
mysql -u <uporabnik> -p <podatkovna baza> < datotekaZaUvoz.sql.
```

Ko so vse relacije na tak način uvožene v podatkovno bazo, je treba pogledati, v kateri normalni obliki se nahaja celotna podatkovna baza. Podatkovna baza mora biti v 1. normalni obliki. Za prvo normalno obliko je značilno:

- da vsak atribut v relacijski shemi mora biti enovrednostni atribut,

- elementi vrednosti množic morajo biti le atomarne vrednosti.

Ob dejstvu, da so opisi artiklov zapisani kot celote (niz znakov), se baza nahaja v 1. normalni obliki.

Baza se nahaja v 2. normalni obliki, kadar:

- se nahaja v 1. normalni obliki,
- ne vsebuje parcialnih odvisnosti

Poleg tega se baza nahaja v 3. normalni obliki, kadar:

- se nahaja v 2. normalni obliki,
- ne vsebuje tranzitivnih odvisnosti.

Baza, ki sem jo uporabljal za namene testiranja, je bila normalizirana v 3. normalni obliki.

### 4.3.2 Priprava testnih podatkov za uvoz v Cassandro

Za uvoz podatkov v Cassandro je treba bazo denormalizirati. Cassandra ne pozna stikov. Stiki so največji problem pri hitrosti izvajanja transakcij v relacijskih podatkovnih bazah. Pri zelo kompleksnih transakcijah lahko traja zelo dolgo časa, ko s stikom povezujemo obsežne relacije.

Podatke smo prenesli v Cassandro s pomočjo datoteke s končnico »csv«. Tudi v MySQL je vgrajena funkcija, ki poskrbi za zapis podatkov v zunanjo datoteko. Treba je določiti:

- kako bodo posamezni zapisi ločeni med seboj,
- kako bodo ločene med seboj n-terice,
- s pomočjo katerih znakov bomo združevali celotne atribute.

Primer stavka za izvoz:

```
SELECT *
INTO OUTFILE '/pot/do/datoteke/datoteka.csv'
FIELDS TERMINATED BY ','
ENCLOSED BY '"'
ESCAPED BY '\\'
```

```
LINES TERMINATED BY '\n'
FROM rwpr;
```

Preden pa se naredi izvoz, je bilo treba stavke denomalizirati. To naredimo s pomočjo stikov med posameznimi relacijami v MySQL. S stališča računalniškega procesiranja je ta korak zelo dolgotrajen. Po procesiranju dobimo zelo obseženo množico združenih relacij.

Ko so denormalizirani podatki shranjeni v datoteki, jih je treba uvoziti v Cassandro.

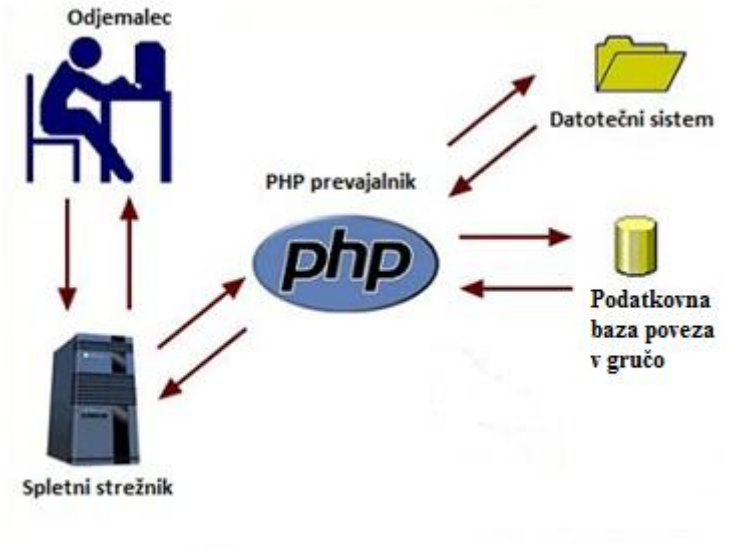
V našem primeru smo naredili imenski prostor z imenom »testSpeedMoreData126«, nato določili, da naj se razdeli na 4 strežnike. V tem prostoru je treba kreirati vrstično družino »test«, ki ima vnose zapisane v obliki »UTF8Type«. Če tip zapisa ni določen, vse vrednosti tipa »varchar« postanejo nizi števil.

Ko je vse pripravljeno, je treba pognati skripto, ki skrbi za prenos podatkov iz datoteke v Cassandro. Zaradi številčno obsežnih podatkov je operacija trajala več kot 25 ur.

### 4.4 Programski jeziki za testiranje

Še preden smo pričeli z dejanskim testiranjem, se je porajalo vprašanje, v katerem programskem jeziku je najbolj primerno narediti performančni test. Zaradi primerljivosti podatkov ni najboljša izbira uporabiti drug programski jezik za dostop do relacijskih podatkovnih baz kot za dostop do NoSQL podatkovne baze. Na voljo imamo Javo, Python in PHP. Odločitev je padla za slednjega zaradi dejstva, da je zelo razširjen na svetovnem spletu.

Vsakega od scenarijev smo poganjali vsaj trikrat na obeh bazah. S pridobljenimi podatki smo izračunali povprečno vrednost. Če je prihajalo do prevelikega odstopanja pri času (varianca, večja od 10 %), je treba pogledati, zakaj se podatki razlikujejo. V večini primerov je razlog za razliko obremenjenost omrežja, na katerega nimamo vpliva. 10 % odstopanja, smo dovolili zaradi dejstva, da so se v tem območju gibali izmerjeni rezultati kateri niso imeli odstopanja zaradi obremenjenosti omrežja. Vsi scenariji, katere smo opredelili na začetku, si sledijo od lažjih do bolj kompleksnih.




Slika 7: Arhitektura PHP ter napisanih testnih skript

Za testiranje se uporabljajo skripte, ki simulirajo izvajanje transakcij v realnih sistemih. Zaradi narave testov je treba v MySQL strežniku pobrisati pomnilnik. Če je vključeno shranjevanje v pomnilnik, bi testi vračali zelo nerelevantne podatke, saj bi bila večina najbolj uporabljenih podatkov že shranjenih v medpomnilniku. V realnem svetu, pri realnih podatkovnih bazah, se v večini primerov ne pojavlja nekaj tisočkrat podobna transakcija nad eno relacijo.

### 4.5 Konfiguracija Apache strežnika in PHP ter nastavitve MYSQL strežnika

Za izvajanje testov performančne zmogljivosti s pomočjo PHP-ja, je potrebna nastavitev Apache strežnika. Prva stvar, ki jo je je priporočljivo nastaviti, je prikazovanje morebitnih napak v brskalniku. To se popravi tako, da spremenljivki »display_errors« nastavimo vrednost 1. Ob pojavitvi napake se bo le-ta zapisala na zaslon v brskalniku.

Pomembna stvar, ki jo je treba nastaviti v konfiguraciji, je največji dovoljeni čas izvajanja skripte. Če pustimo osnovne nastavitve, se po 30 sekundah preneha izvajati, kar pa je za potrebe testa premalo. Ta čas se lahko poveča v konfiguracijski datoteki v spremenljivki »set_time_limit«. Za potrebe testa je najbolje vrednost nastaviti na 86400 sekund, kar je enako 24 ur

#### 4.5.1 Priprava testne skripte za dostop do MySQL baze

Skripta, ki skrbi za dostop do podatkovne baze na MySQL gruči, je napisana v PHP-ju.

Napisan je osnovni razred, ki skrbi za poganjanje testov. V njem so funkcije, ki implementirajo vsak test/scenarij posebej. V vsako funkcijo je dodana koda za štetje časa.

V primeru je viden konstruktor, ki poskrbi za povezavo v MySQL gručo. Vidimo tudi poizvedbo za prvi scenarij, v katerem želimo pridobiti artikel s točno določenimi enoličnimi identifikatorji (primarnimi ključi), ki si sledijo od 0 do vrednosti »n«.

```
class Test{
function konstruktor(){
$con = mysql_connect('192.168.1.126','uporIme','geslo');
if (!$con){ die('Napaka pri povezavi na strežnik: ' . mysql_error()); }
mysql_select_db("benchmark") or die(mysql_error());
}

function scenarij1($number){
$mtime = microtime();
$mtime = explode(" ",$mtime);
$mtime = $mtime[1] + $mtime[0];
$starttime = $mtime;
for($i=0;$i<$number; $i++){
//nr 7089
$query= "SELECT *
FROM product
WHERE nr='$i'; ";
$result = mysql_query($query) or die(mysql_error());
}
$mtime = microtime();
$mtime = explode(" ",$mtime);
$mtime = $mtime[1] + $mtime[0];
$endtime = $mtime;
$totaltime = ($endtime - $starttime);
echo "<br>Stevilo operacij: ".$number." Cas trajanja: ".$totaltime." sekund";
}
function scenarij2(...) { ... }
function scenarij3(...) { ... }
...
}
```

### 4.5.2 Priprava testne skripte za dostop do Cassandre

Za dostop do Cassandre s PHP skripto je treba uporabiti programski paket PHPCassa. Za delovanje je bilo treba na začetku uvoziti knjižnice, ki omogočajo nadaljnje delo.

```
require_once(__DIR__.'/../lib/autoload.php');

use phpcassa\Connection\ConnectionPool;
use phpcassa\ColumnFamily;
use phpcassa\SystemManager;
use phpcassa\Schema\StrategyClass;
use phpcassa\Iterator\ColumnFamilyIterator;
use phpcassa\Iterator\RangeColumnFamilyIterator;
use phpcassa\Index\IndexExpression;
use phpcassa\Index\IndexClause;
use phpcassa\Schema\DataType\LongType;
```

Za dostop do gruče je treba vzpostaviti povezavo z imenskim prostorom. To naredimo z inicializacijo razreda ConnectionPool. Kot atribute se vstavi poimenovanje imenskega prostora ter IP naslove strežnikov, povezane v gručo.

Ko je povezava v imenski prostor vzpostavljena, je treba določiti, v kateri vrstični družini želimo manipulirali s podatki. To naredimo z inicializacijo razreda ColumnFamily, v katerega je treba kot atribute podati objekt imenskega prostora ter ime vrstične družine.

```
$pool=new ConnectionPool('testSpeedMoreData126',
array('192.168.1.116',
        '192.168.1.121',
        '192.168.1.126',
        '192.168.1.136', ));
$users = new ColumnFamily($pool, 'rwpr');
```

V nadaljevanju je treba spisati programsko kodo, ki bo skrbela za izvajanje testov.

V nadaljevanju je podana koda za izvajanje testa prvega scenarija. V seznamu so že vnaprej določene vrednosti testnih primerov. Izvajale se bodo od vrednosti 1 pa do vrednosti, ki je zapisana v seznamu. V vsakem testu je zraven pripeta koda za merjenje časa izvajanja.

```
$primeri=array(1,10,50,100,500,1000,5000,10000);
for($k=0;$k<sizeof($primeri);$k++){
$mtime = microtime();
$mtime = explode(" ",$mtime);
$mtime = $mtime[1] + $mtime[0];
```

```
$starttime = $mtime;

for($i=1;$i<=$primeri[$k];$i++){
$user = $users->get(con($i));
$name = $user[con('title')];
}
$mtime = microtime();
$mtime = explode(" ",$mtime);
$mtime = $mtime[1] + $mtime[0];
$endtime = $mtime;
$totaltime = ($endtime - $starttime);
echo "<br>Stevilo operacij: ".$primeri[$k]." Cas trajanja: ".$totaltime." sekund";
}
```

# Poglavje 5

# Testiranje na podatkih

## 5.1 Uporabniški scenariji

Scenariji, ki so zapisani, so izdelani na temelju verjetnih realnih dogodkov. Ti so številčno potencirani, da se pokažejo prednosti ter slabosti posamezne baze. Napisani scenariji poskrbijo, da bodo na koncu trditve pravilne in bodo držale ob vseh možnih situacijah. V prvi vrsti bodo scenariji razdeljeni za 3 različne operacije:

- branje,
- pisanje,
- brisanje.

Scenariji bodo poskušali podati oceno performančnih testov za vsako od teh operacij. Na koncu bodo napisani tudi scenariji, ki bodo izmenjujoče najprej brali, potem pisali ter na koncu brisali. Tako bo poskrbljeno za najboljšo primerjavo klasične relacijske MySQL baze ter NoSQL baze Cassandra.

Število testnih primerov v vsakem scenariju se bo povečevalo do razumne meje glede na kompleksnost scenarija. Razumna meja je določena glede na časovno potratnost ukazov v scenariju, kjer je čas izvajanja do nekaj ur.

### 5.1.1 Scenarij branja podatkov (1)

Prvi scenarij je najbolj enostaven s stališča podatkovnih baz. V njem želimo prebrati vse podatke o artiklih. Vsak artikel bo pridobljen s svojim ukazom. En ukaz za en artikel. Na primer:

- želimo pridobiti podatke o 1. artiklu,
- želimo pridobiti podatke o 2. artiklu,
- ...
- želimo pridobiti podatke o »n« artiklu.

V relacijski podatkovni bazi so artikli shranjeni v eni relaciji. Ni jih treba povezovati s stiki, da pridemo do želenega podatka. V NoSQL podatkovni bazi je treba poiskati podatek ter potem o njem pridobiti vse želene informacije.

V spodnji tabeli so prikazani časi izvajanja operacije glede na število testnih primerov. Vsi časi so podani v sekundah.

| baza / testni primeri(n) | MySQL | Cassandra |
|---|---|---|
| **1** | 0,0007 | 0,0185 |
| **10** | 0,005 | 0,1668 |
| **50** | 0,0247 | 0,6815 |
| **100** | 0,0289 | 1,3314 |
| **500** | 0,0355 | 6,4889 |
| **1.000** | 0,1592 | 12,5131 |
| **5.000** | 0,5866 | 71,2045 |
| **10.000** | 1,0526 | 151,532 |

Slika 8: Časi, podani v sekundah, za enostavno branje podatkov

Na podlagi te table lahko narišemo graf, ki bo nazorno pokazal čas izvajanja operacij v posamezni bazi.


Slika 9: Grafični prikaz časa izvajanja operacij branja podatkov

Iz danega vidimo presenetljiv rezultat. Cassandra se je odrezala mnogo slabše pri branju enostavnih podatkov. Medtem ko je MySQL baza porabila zanemarljivo malo časa tudi pri 10

tisoč operacijah, je Cassandra porabila več kot 2 minuti in pol. To nam da misliti, da NoSQL podatkovne baze niso koristne na vseh področjih in za vsak problem. Razlika je namreč več kot očitna,v prid relacijskim podatkovnim bazam.

Glede na ta scenarij je priporočljivo v primeru enostavnih poizvedb uporabiti relacijsko podatkovno bazo.

### 5.1.2 Scenarij branja množice podatkov (2)

Scenarij, ki sledi, je zelo podoben prvem scenariju. V tem scenariju želimo pridobiti podatke o artiklih od 0 do določene vrednosti. Na primer:

- pridobiti želimo prvih 5 podatkov o artiklih,
- pridobiti prebrati želimo prvih 10 podatkov o artiklih,
- …
- pridobiti in prebrati želimo prve »n« podatke o artiklih.

V bazi MySQL so podatki shranjeni v eni relaciji, kar naj bil veljalo za podatek, ki je enostavno pridobljiv.

V spodnji tabeli so prikazani časi izvajanja operacij glede na število testnih primerov, ki jih želimo pridobiti iz baze.

| baza / testni primeri(n) | **MySQL** | **Cassandra** |
|---|---|---|
| **1** | 0,0007 | 0,0185 |
| **10** | 0,055 | 0,0645 |
| **50** | 0,0583 | 0,3323 |
| **100** | 0,1497 | 0,8583 |
| **500** | 4,0507 | 5,8536 |
| **1.000** | 11,8882 | 9,0815 |
| **5.000** | 121,7104 | 29,7771 |

Slika 10: Časi, podani v sekundah, za branje množice podatkov

Za lepši prikaz in lažjo predstavo je spodaj podan graf, narejen na osnovi izmerjenih časov za oba tipa podatkovnih baz.


Slika 11: Grafični prikaz izvajanja operacij pri scenariju branja množice podatkov

Pri tem scenariju opazimo, da pri majhnem številu operacij podatkovna baza MySQL dohiteva in celo prehiteva Cassandro. Ko želimo iz podatkovne baze pridobiti večjo skupino podatkov, se Cassandra mnogo bolje izkaže. Razmerje pri 5000 podatkih je:

$$\frac{121{,}7104}{29{,}7771} = 4{,}09$$

Izračunano razmerje je več kot 4, kar je lahko v današnjem času zelo veliko. Iz grafa in tudi računsko je opaziti, da se razlika ne povečuje enakomerno.

Rezultat nam pokaže, da je v primeru, ko želimo pridobiti veliko množico podatkov, bolje uporabiti NoSQL podatkovno bazo. Ravno zaradi tega, se NoSQL baze zelo veliko uporablja za pridobivanje množice podatkov skozi časovno obdobje.

### 5.1.3 Scenarij bolj kompleksnega branja podatkov (3)

Naslednji scenarij je zamišljen v primeru, ko imamo internacionalno trgovino in želimo pogledati, iz katere države prihaja oseba, ki je podala opombo oziroma kritiko. To je pomembno predvsem pri trgovinah, ki prodajajo produkte, ki so vezani na podnebje, običaje v tisti državi itd.

Da bo omogočen dostop do jezika, je treba v relacijski podatkovni bazi povezati med seboj relaciji z uporabo stika. To velja za bolj kompleksno poizvedbo. V Cassandri zelo težko govorimo o tovrstni kompleksnosti, saj so podatki predstavljeni v drugačni obliki.

S scenarijem smo želeli ugotoviti, kako učinkovito se testi razporejajo po gruči. Pognali smo najprej samo en stavek in šele kasneje več stavkov. Ko smo pognali en stavek, smo dobili pričakovane rezultate.

| baza / testni primeri(n) | **MySQL** | **Cassandra** |
|---|---|---|
| **1** | 25 | 2 |

Slika 12: Čas izvajanja kompleksne poizvedbe, podan v sekundah (en ukaz)

V tem primeru je MySQL baza mnogo slabša kot Cassandra. Razlika med njima je v razmerju 12,5 v prid Cassandre.

Naslednji korak je pridobivanje večje količine podatkov. Pri tem bomo pridobili informacijo, ali se v gruči bolje obnese Cassandra ali MySQL pri razporejanju opravil. V ta namen smo pognali v gruči 5 poizvedb, s katerimi bomo lahko določili hitrost in učinkovitost razporejanja opravil na strežnikih.

Poizvedba je enaka kot prejšnja. Poganjali smo 5 različnih poizvedb, ki se ciklično ponavljajo, 100-krat.

V vsaki poizvedbi želimo pridobiti podatke o vseh piscih komentarjev oziroma opomb iz določene države:

- Združene države Amerike,
- Španija,
- Kitajska,
- Nemčija,
- Rusija.

Ko se poženejo stavki za vsako državo, je to 5 poizvedb, ki jih je treba ciklično pognati 100-krat, kar skupaj znese 500 poizvedb. Rezultati, ki so si sledili, so bili presenetljivi.

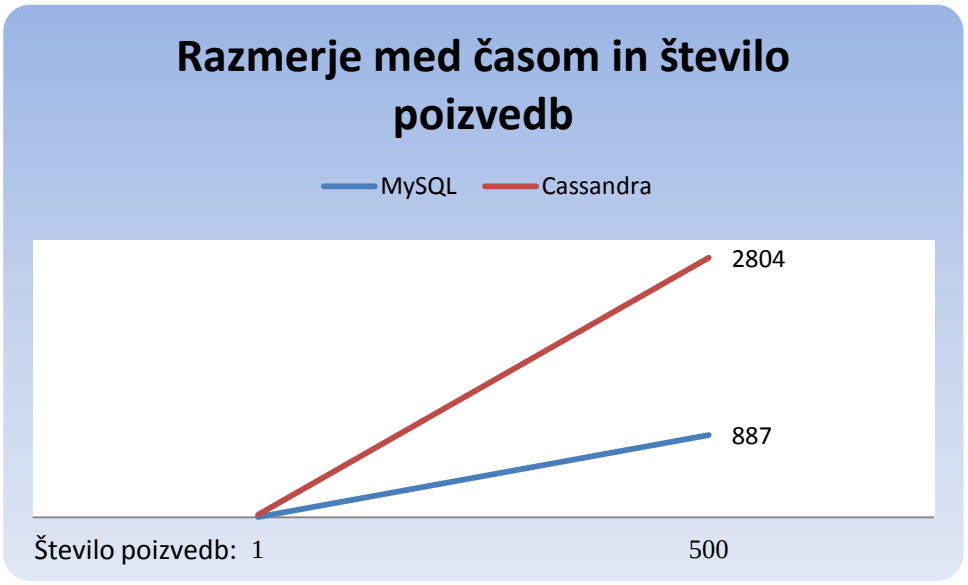
| baza / testni primeri(n) | MySQL | Cassandra |
|---|---|---|
| **5x100 poizvedb** | 2804 | 887 |

Slika 13: Čas izvajanja 500 poizvedb, podan v sekundah

MySQL podatkovna baza je poizvedbe izvedla v 2804 sekundah, medtem ko Cassandra to izvede v 887 sekundah. Razmerje, ki je pri tem, je 3,16.

Pri samo eni poizvedbi je razmerje 12,5. Pri 500 poizvedbah se je razmerje zmanjšalo na 3,16. Še vedno je v veliki prednosti Cassandra, a očitno je, da MySQL z arhitekturo gospodar-suženj mnogo bolje razporeja opravila po svoji gruči. To je laže, saj ima en strežnik vse podatke o obremenjenosti sistema in lahko optimalno razporedi na novo prispele procese oziroma transakcije.

Boljšo učinkovitost vidimo tudi v grafičnem diagramu. Črta, ki nakazuje MySQL podatkovno bazo, ima mnogo manjši naklon. To pomeni, da je veliko manj časa uporabila za množico poizvedb glede na izvedeno eno samo poizvedbo.




Slika 14: Razmerje pri eni oziroma 500 poizvedbah

### 5.1.4 Scenarij pisanja, branja ter brisanja podatkov pri osebah (4)

Scenarij, ki sledi, je vpis podatkov o novih osebah. Naslednji korak je branje teh podatkov in nato še izbris le-teh. Še pred testom je v tej tabeli shranjeno več kot 35.000 zapisov. Za vpis podatkov smo uporabili skripto, ki je vpisovala naključne vrednosti atributom:

- ID osebe,
- ime,
- država od koder prihaja,
- datum objave,
- 40-mestna identifikacijska koda.

ID osebe je vedno naslednja prosta številka. Atribut imena je sestavljen iz naključne dolžine in z naključnimi znaki angleške abecede. Država, od koder prihaja, je pridobljena v seznamu in naključno izbrana med vnosi. Datum objave je naključni dan v mesecu marcu. 40-mestna identifikacijska koda je sestavljena iz niza znakov naključnih števil in črk angleške abecede.

V podatkovno bazo smo s pomočjo skripte vnesli »n« število primerov. Sledi branje vseh na novo vnesenih podatkov o osebah. Zadnji korak je brisanje vnesenih podatkov z namenom vračanja baze v prvotno stanje.

V tabelah so predstavljeni časi izvajanja podani v sekundah posameznih vrst transakcij glede na število testnih primerov. V prvi tabeli so prikazani časi za relacijsko podatkovno bazo.

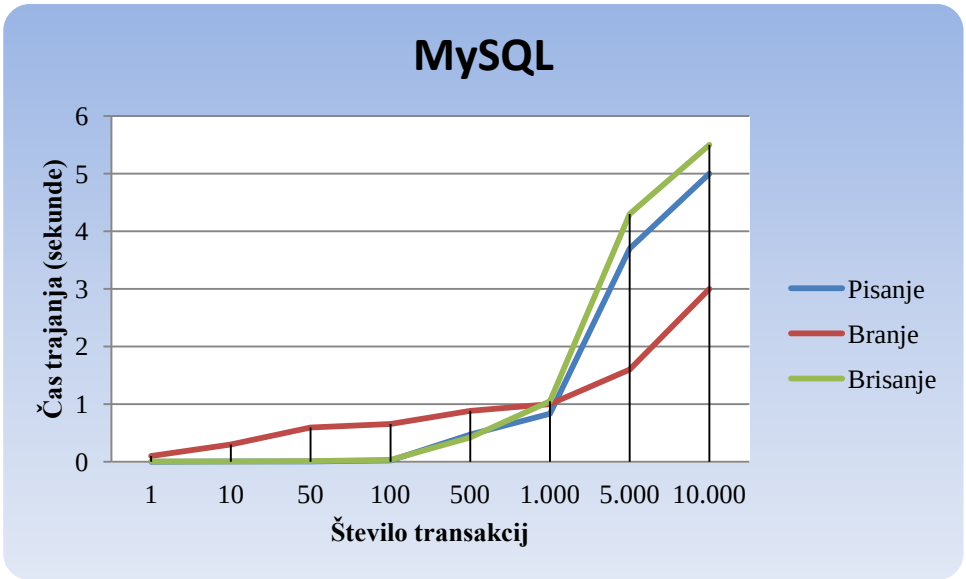
| Transkcija / testni primeri(n) | **Pisanje** | **Branje** | **Brisanje** |
|---|---|---|---|
| **1** | 0,0008 | 0,1 | 0,001 |
| 10 | 0,0011 | 0,3 | 0,002 |
| 50 | 0,0063 | 0,59 | 0,008 |
| 100 | 0,02 | 0,652 | 0,029 |
| 500 | 0,47 | 0,883 | 0,42 |
| 1.000 | 0,83 | 1,001 | 1,05 |
| 5.000 | 3,7 | 1,6 | 4,3 |
| 10.000 | 5 | 3 | 5,5 |
| 100.000 | 71 | 34 | 75 |

Slika 15: Čas izvajanja transakcij v sekundah v MySQL bazi

Podan je graf, ki je narejen na podlagi narejenih meritev MySQL podatkovne baze. Za boljši pregled je graf narisan do 10.000 (če želimo narisati do 100.000 primerov, postane graf v manjših vrednostih nepregleden). Število transakcij nam pove, koliko teh je izvedeno v posameznem tipu transakcij. Na primer, če imamo 1.000 transakcij, to pomeni:

- 1.000 transakcij pisanja,
- 1.000 transakcij branja,
- 1.000 transakcij brisanja.

To skupaj znese kar 3.000 transakcij.




Slika 16: Graf časov izvajanj transakcij v MsSQL bazi

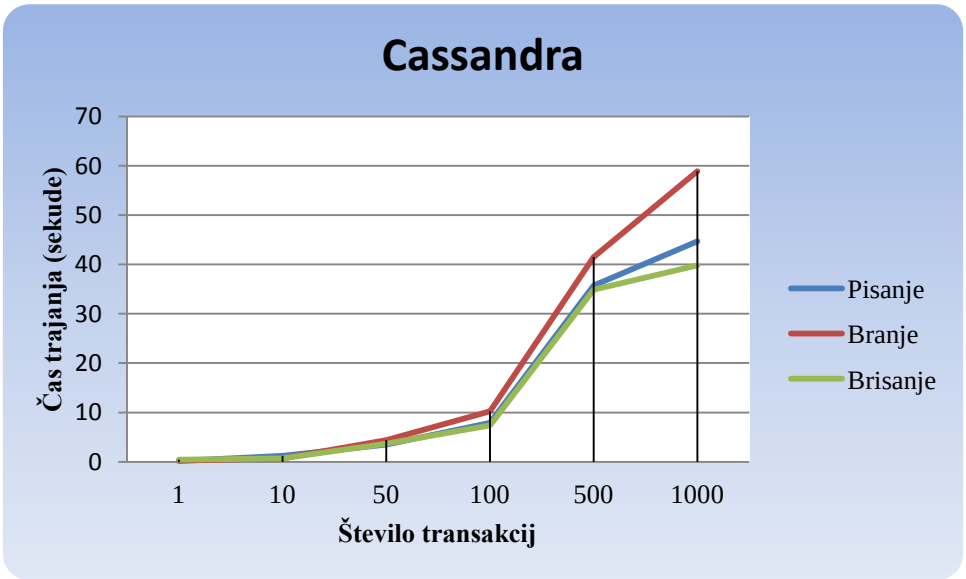
Pri MySQL podatkovni bazi se vidi zanimivo obnašanje grafa. Do približno 1.000 transakcij se graf enakomerno povečuje glede na število transakcij. Pri številu 1.000 pa transakcije potrebujejo vse več potrebnega časa, da se izvedejo. Obrne se trend in najbolj »požrešen« tip transakcije ni več branje, ampak postaneta pisanje in brisanje. Vse to nakazuje na dejstvo, da branje podatka z enim ID ni problematično. Pri veliki količini podatkov postaneta zelo problematična pisanje in brisanje.

Isti scenarij smo ponovili tudi na Cassandri ter dobili rezultate, izražene v sekundah.

| Transkcija / testni primeri(n) | Pisanje | Branje | Brisanje |
|---|---|---|---|
| **1** | 0,204 | 0,211 | 0,4023 |
| 10 | 1,23 | 0,722 | 0,711 |
| 50 | 3,5 | 4,39 | 3,69 |
| 100 | 7,877 | 10,24 | 7,37 |
| 500 | 35,72 | 41,5 | 34,91 |
| 1.000 | 44,7 | 58,86 | 39,83 |
| 5.000 | 183,29 | 229,2 | 164,7 |

Slika 17: Čas izvajanja ukazov v sekundah v Cassandri

Na podlagi pridobljenih podatkov je narisan graf, ki nazorno prikazuje čas izvajanja ukazov. Graf je zaradi lepše preglednosti narisan samo do izvedbe 1000 transakcij.

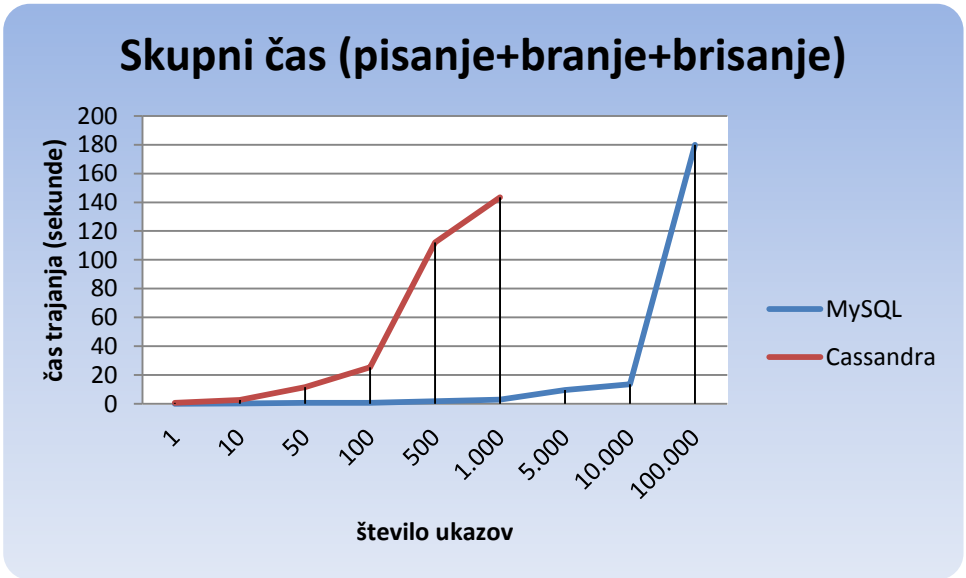



Slika 18: Graf časov izvajanja transakcij v Cassandri

Iz grafa lahko razberemo, da Cassandra porabi zelo veliko časa za samo administracijo in delovanje podatkovne baze. Pri zelo enostavnih ukazih, ki jih je malo, porabi zelo veliko časa glede na MySQL. Pri izbranem scenariju je tudi časovno Cassandra veliko slabša kot MySQL. Vse to kaže na dejstvo, da Cassandra ni najboljša izbira v situacijah, ko izvajamo enostavne poizvedbe.

Pri Cassandri je opaziti dejstvo, da je branje vseskozi najbolj potratna operacija, in ni tako kot pri MySQL, ko je branje samo na začetku zelo potratno, kasneje pa se stabilizira glede na pisanje in brisanje.

Za lažje razumevanje in primerjavo je narejen seštevek za vse čase skupaj (pisanje + branje + brisanje) ter narisan graf časov izvajanja skupine ukazov za MySQL in Cassandro. Za lepši pregled je graf narisan do 1000 izvedenih ukazov pri Cassandri.




Slika 19: Skupni čas trajanja transakcij za oba tipa podatkovnih baz

Na našem strežniškem omrežju so bile omejitve izvajanja ukazov zaradi pomnilnika in hitrosti procesorskih enot. Bilo bi zanimivo pogledati situacijo enostavnih ukazov na primeru pol milijona operacij. Kot lahko razberemo iz obeh grafov, je pri Cassandri strmina grafa začela padati pri večjem številu ukazov, medtem ko je pri MySQL začel graf strmo naraščati. Po predvidevanjih bi se morala ta dva grafa srečati ter nadaljevati v prid Cassandre.

### 5.1.5 Scenarij pisanja, branja ter brisanja podatkov pri opombah/kritikah (5)

Naslednji scenarij, ki je bil narejen, je podoben prejšnjemu scenariju. Razlika je le v tem, da so podatki, v katere se vstavlja, bere ter briše, vstavljeni v večjo relacijo. Tu je zapisanih okoli 714.500 zapisov, katerega atributi so mnogo bolj obsežni. S stališča Cassandre je to zelo podoben primer kot pri prejšnjem scenariju. V relaciji so shranjeni atributi:

- ID opombe/kritike,
- ID proizvajalca,

- ID osebe,
- datum vnosa opombe/kritike,
- naslov opombe/kritike,
- opis opombe/kritike,
- jezik,
- 4 atributi različnih ocen.

ID opombe, ID proizvajalca in ID osebe so tuji ključi, ki skrbijo, da lahko naredimo stik z drugo relacijo. Najbolj pomemben pri tem scenariju je opis opombe/kritike. Ta je v nekaterih primerih dolg skoraj 200 znakov. Opis opombe je primeren atribut za preverjanje zmožnost obdelovanja dolgih nizov znakov v bazi, ki ima veliko zapisov.

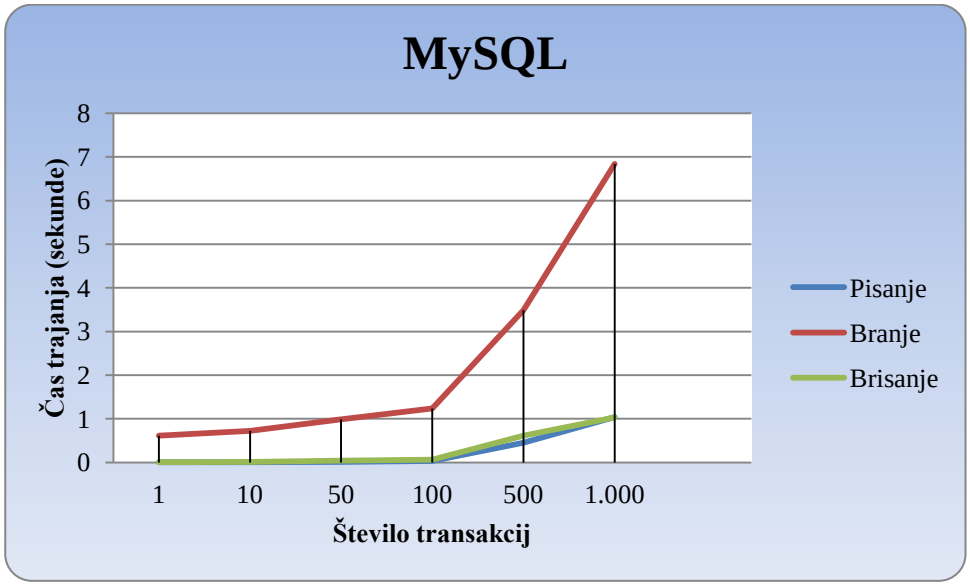
V fazi pisanja se uporablja naslednjo prosto vrednost za primarni ključ ID opombe/kritike. Za tuje ključe so vpisane naključne vrednosti med 0 in 1000. Pri datumu je izbran naključni dan v mesecu aprilu. Naslov in opis opombe/kritike je vpisan kot niz naključnih znakov. Za kasnejše preverjanje podatkovnih baz z dolgimi nizi znakov je v približno 30 % primerih dodan vnaprej določen niz znakov, katerega bomo kasneje iskali v opciji branja. Jezik je izbran naključno z vnaprej določenega seznama. Ocene so naključne vrednosti med 1 in 5.

Po izmerjenih časih smo dobili tabelo, ki nam prikaže trajanje transakcij s časovno enoto sekunde.

| Transkcija / testni primeri(n) | **Pisanje** | **Branje** | **Brisanje** |
|---|---|---|---|
| **1** | 0,0013 | 0,611 | 0,0022 |
| 10 | 0,0025 | 0,727 | 0,011 |
| 50 | 0,0087 | 0,986 | 0,0423 |
| 100 | 0,035 | 1,236 | 0,0629 |
| 500 | 0,45 | 3,489 | 0,611 |
| 1000 | 1,04 | 6,84 | 1,03 |
| 5000 | 3,7 | 81 | 4,94 |

Slika 20: Izvajanje transakcij v sekundah v MySQL bazi

Za lažjo predstavo je podana tudi grafična predstavitev podatkov, ki je narisana do 1000 transakcij zaradi lepšega pregleda.



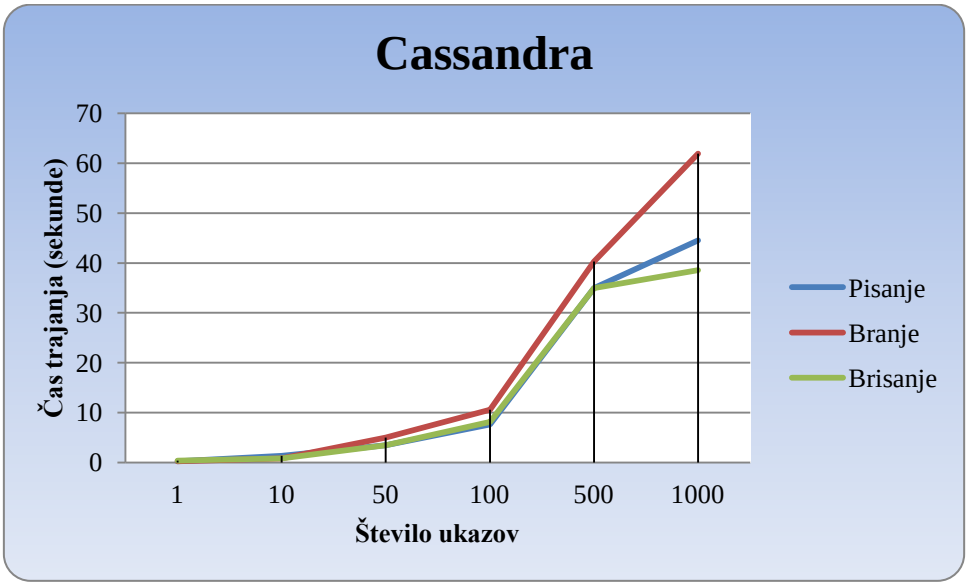

Slika 21: MySQL grafični prikaz trajanje transakcij v sekundah

Iz podatkov je razvidno, da je zaradi mnogo večjega števila primerov v tabeli prišlo do velikega odstopanja pri branju podatkov. Po predvidevanjih bi z večanjem števila testnih primerov krivulja postala še bolj strma in bi nakazovala, da je MySQL časovno zelo potraten z veliko količino podatkov. Operaciji pisanja ter brisanja pa sta ostali skoraj na istem nivoju, kot je bilo pri manj številčni relaciji.

S Cassandro smo dobili naslednje izmerjene čase ter podani graf:

| Transkcija / testni primeri(n) | **Pisanje** | **Branje** | **Brisanje** |
|---|---|---|---|
| **1** | 0,215 | 0,22 | 0,35 |
| 10 | 1,356 | 0,81 | 0,823 |
| 50 | 3,46 | 4,963 | 3,42 |
| 100 | 7,632 | 10,56 | 8,1 |
| 500 | 34,961 | 40,3 | 34,95 |
| 1000 | 44,55 | 61,9 | 38,552 |
| 5000 | 189,46 | 236,1 | 169,5 |

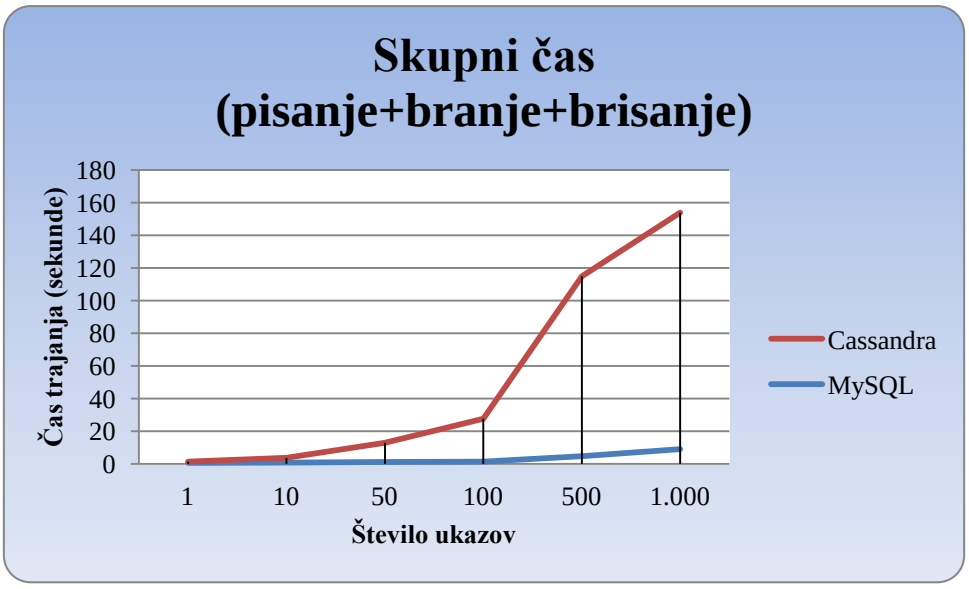
Slika 22: Cassandra izvajanje ukazov v sekundah




Slika 23: Čas izvajanja ukazov v Cassandri, podan v sekundah

Podatki v NoSQL bazi Cassandra so skoraj identični prejšnjemu scenariju. Cassandra očitno ni občutljiva, ali iščemo po ključu, ki je numeričnega tipa, ali po nizu znakov.

Lahko izrišemo skupne čase (pisanje + branje + brisanje) obeh baz za pregled dogajanja s povečevanjem števila testnih primerov.




Slika 24: Skupni čas trajanja transakcij za obe skupine podatkovnih baz

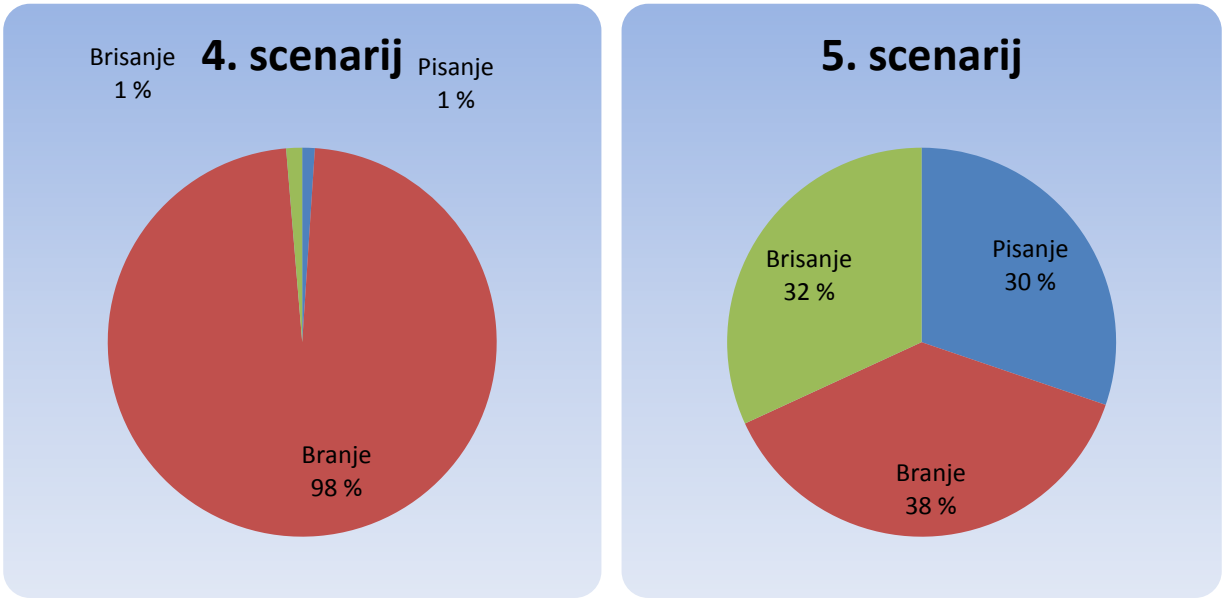
#### 5.1.5.1 Primerjava scenarija pisanja, branja ter brisanja podatkov oseb in opomb (4. in 5. scenarij)

Pri relacijski podatkovni bazi sta dve različni relaciji:

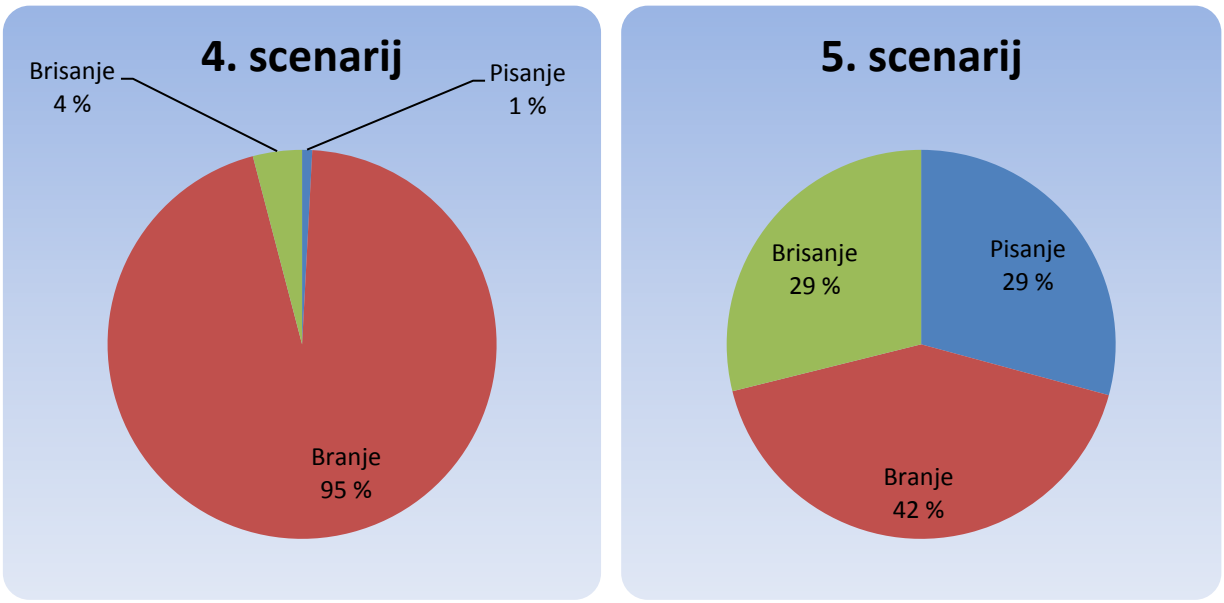
- opombe/kritike,
- osebe.

Prva relacija ima vpisanih podatkov pred izvajanjem scenarija malo več kot 35.000 zapisov, medtem ko ima druga relacija pred izvajanjem scenarija vpisanih več kot 714.500 zapisov.

V primerjavi med scenarijema, bi radi pogledali, kolikšen odstotek zaseda določena skupina ukazov v 4. in 5. scenariju pri 50 ukazih.



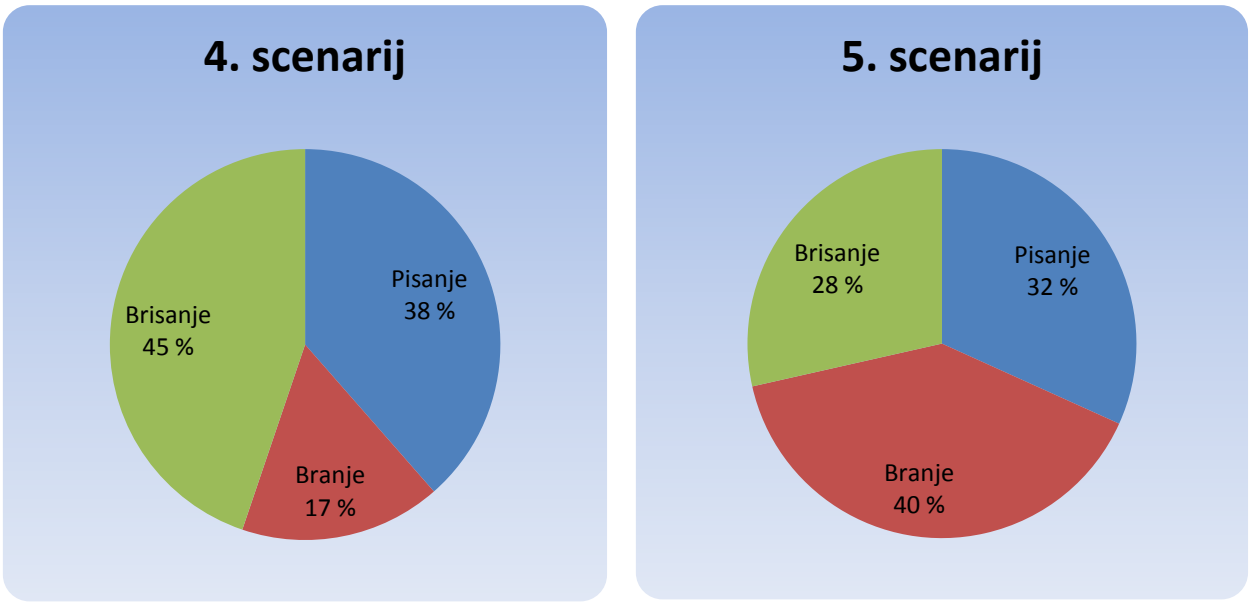

Slika 25: Primerjava časov izvajanja za 50 testnih primerov v Mysql bazi



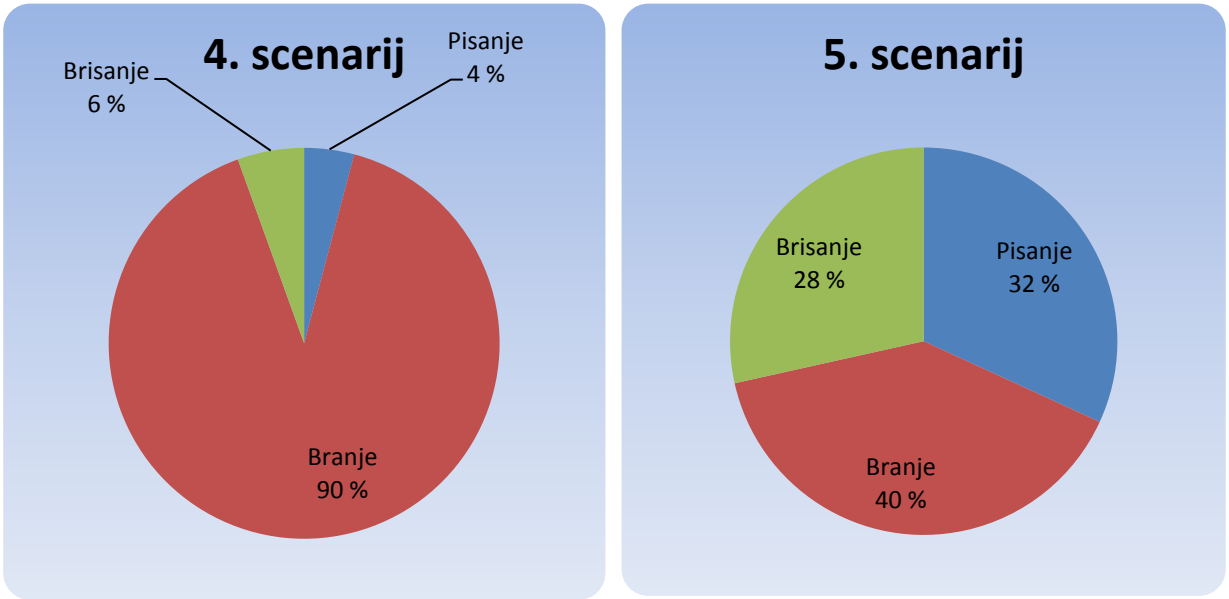

Slika 26: Primerjava časov izvajanja za 50 testnih primerov v Cassandri

Iz teh dveh krožnih diagramov se vidi, da obe podatkovni bazi porabita za enak tip operacije približno enako časa. To pomeni, da pri majhni količini podatkov ni nobena baza boljša ali slabša v primeru pisanja, branja ali brisanja podatkov.

Enako bi radi naredili za 5.000 primerov, da bomo lahko videli, če se bazi enako obnašata tudi na veliki količini podatkov.




Slika 27: Primerjava časov izvajanja za 5.000 testnih primerov v Mysql bazi




Slika 28: Primerjava časov izvajanja za 5.000 testnih primerov v Cassandri

Pri operiranju z večjimi količinami podatkov se iz krožnega diagrama razbere, da je pomembno, kateri tip baze uporabimo. Če imamo malo podatkov ter podatke običajno samo beremo, Cassandra ni najboljša izbira.

Če imamo majhne relacije, kar v našem primeru pomeni okoli 35.000 zapisov, je v primeru veliko sprememb (brisanja, dodajanja) bolje vzeti Cassandro. Cassandra pa je občutno slabša pri branju podatkov iz baze. Zanimivo, da je odstotkovno razmerje izvajanja skupin ukazov med Cassandra in Mysql popolnoma identičeno.

### 5.1.6 Scenarij zahtevnega pisanja branja ter brisanja (6)

Scenarij, ki sledi, simulira zahtevno pisanje, branje ter brisanje podatkov. Zahtevno je predvsem zaradi dejstva, da želimo operirati z veliko količino podatkov. S stališča relacijskih podatkovnih baz je to ena najbolj časovno potratnih operacij, saj želimo s stikom povezati med seboj 3 relacije. Od tega sta dve relaciji zelo številčni:

- relacija opombe kritike (okoli 714.500 zapisov),
- relacija produktov (okoli 72.760 zapisov),
- relacija proizvajalcev (okoli 1420 zapisov).

Scenarij, katerega želimo izvesti, temelji na zgodbi, v kateri želimo pridobiti za vsako kritiko podatek, na kateri artikel oziroma več artiklov se nanaša. Poleg tega nas zanima podatek, kdo je proizvajalec tega artikla. Neposredno želimo pridobiti informacijo o tem, na kateri produkt določenega proizvajalca se nanaša kritika.

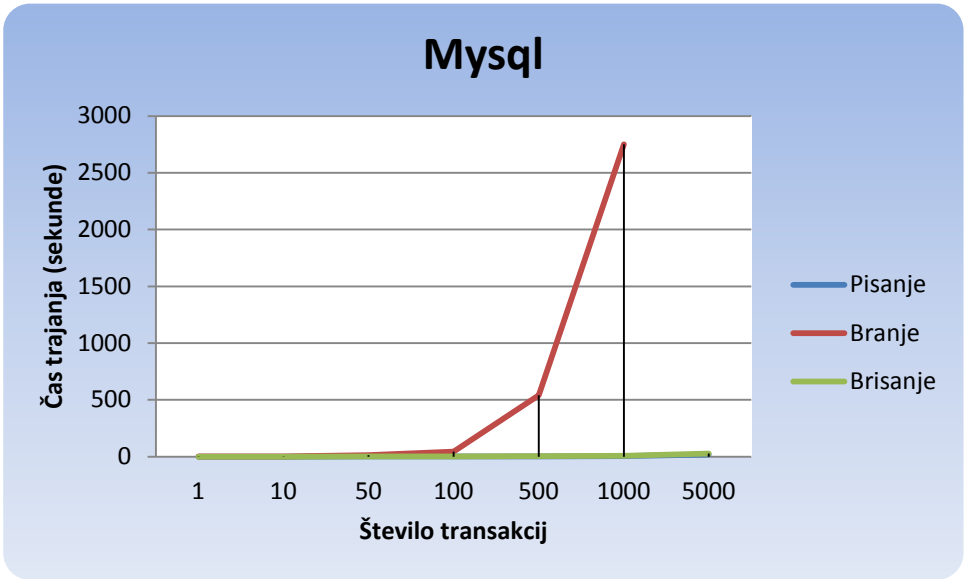
Vse te podatke je treba pred tem zapisati v bazo ter na koncu pobrisati.

Da bodo strežniki lahko vse teste izvedli, bodo vneseni podatki omejeni na »n« primerov in tudi kasneje v iskanju ter brisanju bodo testi omejeni le na te primere. Vsi podatki, ki bodo zapisani v bazo, so izmišljeni. V relacijski podatkovni bazi bo treba skrbeti, da so zapisani pravilni primarni in tuji ključi.

Tabela in pripadajoči graf za MySQL podatkovno bazo

| Transkcija / testni primeri(n) | **Pisanje** | **Branje** | **Brisanje** |
|---|---|---|---|
| **1** | 0,003 | 0,2 | 0,02 |
| 10 | 0,008 | 0,9 | 0,073 |
| 50 | 0,213 | 13,8 | 0,245 |
| 100 | 0,54 | 44,71 | 0,73 |
| 500 | 2,41 | 541 | 3,23 |
| 1000 | 4,12 | 2750 | 6,21 |
| 5000 | 19,35 | ??? | 26,7 |
| ?? = več kot 1 ura | | | |

Slika 29: MySQL izvajanje ukazov v sekundah



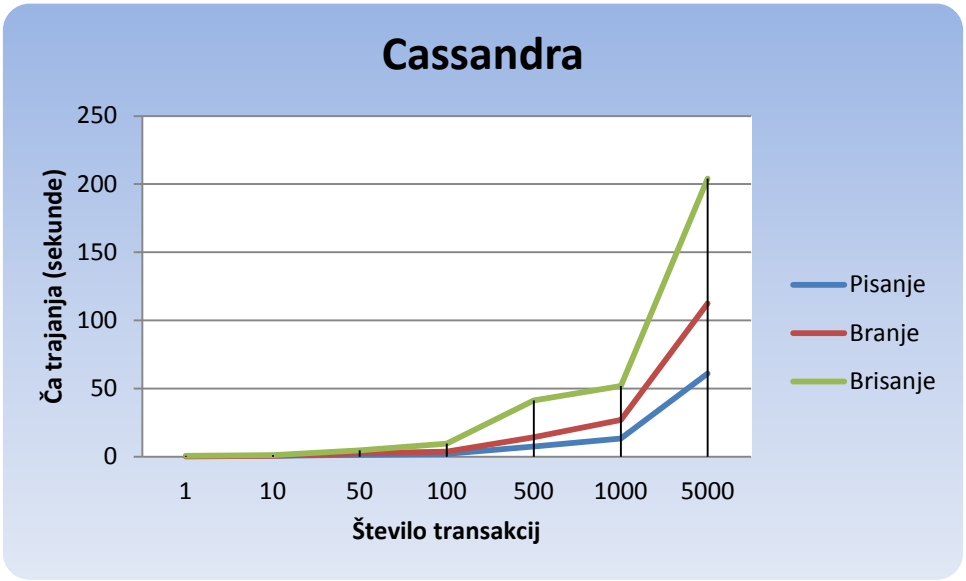

Slika 30: Graf časov izvajanja transakcij v MySQL

V podatkovni bazi MySQL je opaziti zelo veliko časovno potratnost, ko želimo dostopati do velikega števila podatkov v različnih relacijah. Dodajanje in brisanje delujeta brez težav zaradi dejstva, da tu ni stikov. Sicer bi lahko pri brisanju delali stik in vse skupaj pobrisali, a tega zaradi optimalnega delovanja ne želimo. Bolje je pobrisati zapis v vsaki relaciji posebej. Vidimo, da v grafu črta, ki prikazuje čas branja s številom transakcij in velikostjo baze, zelo narašča. Očitno je ravno to slabost relacijskih podatkovnih baz.

Enake operacije smo naredili tudi z bazo Cassandra, ter dobili naslednje rezultate.

| Transkcija / testni primeri(n) | **Pisanje** | **Branje** | **Brisanje** |
|---|---|---|---|
| 1 | 0,305 | 0,11 | 0,5693 |
| 10 | 0,683 | 0,53 | 0,9532 |
| 50 | 1,3 | 2,36 | 4,58 |
| 100 | 1,98 | 3,75 | 9,52 |
| 500 | 7,32 | 14,21 | 41,2 |
| 1000 | 13,4 | 26,91 | 51,9 |
| 5000 | 60,87 | 112,5 | 204 |

Slika 31: Cassandra izvajanje ukazov v sekundah




Slika 32: Graf časov izvajanja transakcij v Cassandri

Pri Cassandri je opaziti, da se kljub povečevanju podatkov, časi ne začnejo povečevati v nerazumne meje.

Če želimo obe bazi primerjati kot celotno operacijo, moramo sešteti čase pisanja, branja in brisanja ter te čase zapisati v tabelo za lažjo primerjavo.

| Podatkovna baza / testni primeri(n) | MySQL | Cassandra |
|---|---|---|
| 1 | 0,223 | 0,9843 |
| 10 | 0,981 | 2,1662 |
| 50 | 14,258 | 8,24 |
| 100 | 45,98 | 15,25 |
| 500 | 546,64 | 62,73 |
| 1000 | 2760,33 | 92,21 |
| 5000 | ?? | 377,37 |
| ?? = več kot 1 ura | | |

Slika 33: Seštevek časov obeh baz

Za boljšo predstavo, kako se povečujejo časi, je narisan graf.


Slika 34: Grafična primerjava seštevkov časov

Na grafu seštevkov časov se vidi, kako relacijska podatkovna baza potrebuje vse več časa za izvajanje ukazov. To je posledica vse večjih podatkov v relacijah, nad katerimi izvaja operacijo stika. V tem primeru je uporabljen naravni stik, ki je časovno manj potraten. Ta je bil izbran zaradi dejstva, da vemo, da so v podatkovni bazi zapisani podatki v obeh relacijah. Če tega ne bi vedeli zagotovo, bi se morali lotiti združevati relacije z levim ali desnim stikom, ki je časovno bolj potraten in časi bi bili še mnogo slabši.

# Poglavje 6

# Sklepne ugotovitve

## 6.1 Kdaj izbrati MySQL oziroma Cassandra gručo

Na koncu vseh izvedenih scenarijev in izmerjenih časov v različnih situacijah je treba podati ugotovitve, ki bodo olajšale izbiro tipa podatkovne baze.

V prvi vrsti je pomembno, kakšen tip sistema želimo imeti za hranjenje posameznih podatkov. Ali je pomembna skalabilnost, konsistentnost, centralizacija, delovanje brez izpada sistema itd.

Za MySQL bazo se je bolje odločiti, ko:

- imamo ACID transakcije,
- želimo imeti konsistentne podatke,
- izvajamo enostavne transakcije,
- nimamo ogromnih količin podatkov,
- ni potrebe po decentralizaciji.

Za Cassandro se je bolje odločiti, ko:

- imamo ogromne količine podatkov,
- želimo, da je sistem več čas delujoč,
- ni potrebe po strogi konsistentnosti,
- imamo podatke decentralizirane,
- želimo visoko zmogljivost,
- želimo shranjevati in procesirati podatkovne tokove.

V performančnih testih so scenariji pokazali, da Cassandra ni primerna v vseh primerih. Če imamo enostavne transakcije, je MySQL še vedno boljša rešitev. Pri zelo kompleksnih poizvedbah, v katerih želimo pridobiti in operirati z velikimi količinami podatkov, so mnogo boljša izbira NoSQL podatkovne baze.

Če želimo, da je sistem vseskozi delujoč, so boljša izbira NoSQL baze. Primer, ki se je zgodil tudi meni med testiranjem, je bil naslednji: strežniku, na katerem sem imel nameščen člen Cassandra gruče ter »gospodar« strežnik za MySQL gručo, se mi je pokvaril (zaradi preobremenjenosti je odpovedal napajalnik). MySQL gruča je postala v tem primeru neuporabna, medtem ko je Cassandra gruča delovala naprej.

Da sem preveril delovanje Cassandre v primeru nedelovanja enega strežnika od štirih, sem na gruči ponovil 1. scenarij. Podatki so bili pričakovani. Za majhno število ukazov, so bili časi skoraj identični kot pri polnem delovanju gruče. Pri povečevanju števila ukazov pa so časi vse bolj odstopali. Pri 10.000 ukazih je bil performančni test slabši za skoraj 20 %.

Pri testnih podatkih, ki so zelo obsežni, se je izkazalo, da je mnogo bolje vzeti NoSQL podatkovno bazo. Faktor razlike je bil v nekaterih primerih tudi več kot 10, v prid NoSQL podatkovnim bazam.

Primerjali smo tudi čase izvajanja skupin ukazov:

- pisanja,
- branja,
- brisanja.

S tega stališča sta se obe bazi pokazali za zelo podobne. Edino odstopanje je v prid relacijskim podatkovnim bazam, ko želimo pridobiti podatke v maloštevilnih zapisih.

Pri testu, v katerem smo ugotavljali učinkovitost delovanje gruče, se je mnogo bolje izkazala MySQL gruča. Očitno je, da je arhitektura gospodar-suženj, mnogo bolj učinkovita za enakomerno izkoriščanje strojnih virov v gruči

## Zaključek

V diplomski nalogi sem predstavil NoSQL podatkovne baze. Ker smo še vedno bolj navajeni klasičnih relacijski podatkovnih baz, sem Cassandro primerjal prav z njimi. Podatkovne baze sem primerjal z različnih stališč, kot so hramba podatkov, varnost, razlika v izvajanju ukazov, obnašanje v omrežju, arhitektura, dostopi do baze itd.

Za podano konkretno končno oceno smo naredili performančne teste za bazo Cassandra in MySQL. Obe bazi smo povezali v gručo štirih strežnikov. V obe bazi smo uvozili podatke ter pričeli z izvajanjem scenarijev. Narejenih je bilo šest scenarijev, ki so bili raznolični. Pokazali so prednosti in slabosti baz v različnih situacijah.

V zadnjem času se vse več govori o NoSQL podatkovnih bazah. Kadarkoli zasledimo diskusijo ali članek o NoSQL podatkovnih bazah, povsod poveličujejo njene zmogljivosti ter prednosti. V testu sem dokazal, da NoSQL podatkovne baze niso v vseh primerih najboljša izbira. Za odločitev je treba pogledati karakteristike posameznega tipa podatkovnih baz. Na podlagi tega se lahko odločimo za tip podatkovne baze. Gotovo pa velja, da ni vedno najbolje uporabiti NoSQL bazo.

Zaradi razvoja socialnih omrežij ter računalništva v oblaku se NoSQL baze vse bolj razvijajo. Posebej računalništvo v oblaku bo v nekaj letih postalo zelo pomembno, saj se bo pri tem hranjenje podatkov zelo spremenilo. Ljudje bodo imeli vse več podatkov shranjenih v oblakih pri ponudnikih hranjenja podatkov. Ti ponudniki bodo uporabljali NoSQL. S tem se bodo izognili problemom ob nedelovanju kakšnega strežnika ter skrbeli za hiter dostop do podatkov.

Diplomska naloga bo v pomoč vsem inženirjem, administratorjem podatkovnih baz, razvijalcem programske opreme, da se bodo laže odločili, kateri tip podatkove baze je bolj primeren za želeni tip aplikacije oziroma storitve. Diplomska naloga je tudi v pomoč pri spoznavanju z NoSQL podatkovnimi bazami, ki so po logiki nasprotne, glede na klasične relacijske baze. Poleg tega so opisani ter razloženi pojmi, ki se uporabljajo pri operiranju s podatkovnimi bazami.

Če bi imeli na voljo dovolj strežnikov, bi bilo zanimivo preizkusiti, kako bi podatkovne baze delovale na 100 ali več strežnikih. Tu bi se po vsej verjetnosti Cassandra še bolje izkazala.

Za zaključek naj še enkrat poudarim, da NoSQL podatkovne baze niso naredile revolucije na področju hranjenja podatkov. Le bolj primerne so za nekatere vrste podatkov in problemov, ki

se pojavljajo v zadnjem času. Preden se odločimo za NoSQL podatkovne baze, je priporočljivo preveriti, ali je res boljša rešitev za naš problem.

## Viri


[1] P. DuBois, MySQL, New Riders Publishing, dec 1999, pogl. 5, 6, 7.

[2] E. Hewitt, Cassandra: The Definitive Guide, nov. 2010.

[3] B. Laurie, Apache: The Definitive Guide, O'Reilly Media; feb. 1999, str. 9–45.

[4] J. Pollock, JavaScript, A Beginner's Guide(Third Edition), McGraw-Hill Osborne Media,

sep. 2009, pogl. 2.

[5] E. Redmond in J. R. Wilson, Seven Databases in Seven Weeks: A Guide to Modern Databases and the NoSQL Movement, maj 2012.

[6] J. Wait, PHP 5 Power Programming, Maryland, okt. 2004, pogl. 4, 8.


## Spletni viri


[7] Apache strežnik. Dostopno 2012 na:

http://en.wikipedia.org/wiki/Apache_HTTP_Server.

[8] BigTable. Dostopno 2012 na:

http://static.googleusercontent.com/external_content/untrusted_dlcp/research.google.com/sl//archive/bigtable-osdi06.pdf.

[9] Brewerjev CAP teorem. Dostopno 2012 na:

http://www.julianbrowne.com/article/viewer/brewers-cap-theorem/.

[10] CouchDB. Dostopno 2012 na:

http://www.ibm.com/developerworks/opensource/library/os-couchdb/index.html.

[11] Dynamo. Dostopno 2012 na:

http://www.allthingsdistributed.com/2007/10/amazons_dynamo.html.

[12] Hector vmesnik. Dostopno 2012 na:

https://github.com/rantav/hector/downloads.

[13] Kodirni algoritem MD5. Dostopno 2012 na:

http://en.wikipedia.org/wiki/MD5.

[14] Kriptografski protokol SSL. Dostopno 2012 na:

http://en.wikipedia.org/wiki/Secure_Sockets_Layer.

[15] Matematični zapis relacijskih baz. Dostopno 2012 na:

ttp://colos1.fri.uni-lj.si/ERI/RACUNALNISTVO/PODATKOVNE_BAZE/arhitektura_supb.html.

[16] Napad mož v sredini. Dostopno 2012 na:

https://www.owasp.org/index.php/Man-in-the-middle_attack.

[17] Odjemalec – strežnik arhitektura. Dostopno 2012 na:

http://www.s-sers.mb.edus.si/gradiva/rac/moduli/podatkovne_baze/08_aplikacije/

01_datoteka.html.

[18] Operacije v Cassandri. Dostopno 2012 na:

http://wiki.apache.org/cassandra/Operations.

[19] PHPCassa vmesnik. Dostopno 2012 na:

https://github.com/hoan/phpcassa.

[20] Pojmi v relacijski bazi. Dostopno 2012 na:

http://drenovec.tsckr.si/model/relac.htm.

[21] Predstavitev NoSQL baz. Dostopno 2012 na:

http://newtech.about.com/od/databasemanagement/a/Nosql.htm.

[22] SimpleDB. Dostopno 2012 na:

http://docs.amazonwebservices.com/AmazonSimpleDB/latest/DeveloperGuide/Welcome.html.

[23] SQL baza. Dostopno 2012 na:

http://en.wikipedia.org/wiki/SQL.

[24] Strategije pri branju ter pisanju. Dostopno 2012 na:

http://answers.oreilly.com/topic/2408-replica-placement-strategies-when-using-cassandra/.

[25] Testni podatki. Dostopno 2012 na:

www4.wiwiss.fu-berlin.de/bizer/BerlinSPARQLBenchmark/results/index.htmlb.