Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Luka Šilovinac

Štetje dogodkov s porazdeljenimi podatkovnimi tipi za dosego močne zakasnjene konsistentnosti

diplomsko delo

univerzitetni program računalništvo in informatika

Ljubljana, 2016

Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Luka Šilovinac

Štetje dogodkov s porazdeljenimi podatkovnimi tipi za dosego močne zakasnjene konsistentnosti

diplomsko delo

univerzitetni program računalništvo in informatika

Mentor: doc. dr. Dejan Lavbič

Ljubljana, 2016

Besedilo je oblikovano z urejevalnikom besedil LaTeX.

Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Tematika naloge:

V času rasti obsega in količine podatkov, ki so na voljo na svetovnem spletu, se na področju obvladovanja podatkov v okviru podatkovnih baz pojavljajo številni novi izzivi. Relacijske podatkovne baze pogosto ne omogočajo ustrezne in učinkovite rešitve, zato se vedno bolj uveljavjajo namenske NoSQL podatkovne baze. Eden izmed aktualnih problemov je štetje dogodkov in zanesljivo shranjevanje števcev v porazdeljenih okoljih s poudarkom na skalabilnosti in visoki razpoložljivosti. V okviru diplomske naloge zato raziščite omenjen problem in empirično preverite raven podpore štetju dogodkov s CRDT-ji v okviru najbolj razširjenih NoSQL podatkovnih baz. Pri evaluaciji se osredotočite na metrike zakasnitve, delež zahtev, ki se uspešno obdelajo in delež uspešnih zahtev, za katere podatki ostanejo zapisani. Rezultate meritev kritično ovrednotite in jih ustrezno argumentirajte.

Izjava o avtorstvu diplomskega dela

Spodaj podpisani Luka Šilovinac, z vpisno številko 63070176, sem avtor diplomskega dela z naslovom:

Štetje dogodkov s porazdeljenimi podatkovnimi tipi za dosego močne zakasnjene konsistentnosti

S svojim podpisom zagotavljam, da:

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

V Ljubljani, 16. avgusta 2016 Podpis avtorja:

Za vse nasvete in strokovno pomoč se zahvaljujem mentorju doc. dr. Dejanu Lavbiču.

Zahvaljujem se Tomažu Kovačiču in mag. Dušanu Omerčeviću za nasvete, podporo in spodbudo pri pisanju diplomske naloge. Prav tako se zahvaljujem podjetju Zemanta za podporo pri opravljanju praktičnih preizkusov v delu.

Zahvaljujem se tudi svoji družini, mami Zlatki, očetu Mateu in sestri Nini, za vse potrpljenje in spodbudo tekom študija.

Kazalo

Seznam uporabljenih kratic

Kratica	angleško	slovensko
ACID	Atomic, Consistent, Isolated, Durable	Nedeljivo, konsistentno, izolirano in trajno
AWS	Amazon Web Services	Spletne storitve Amazon
BSON	Binary JSON	Dvojiški JSON
CAP	Consistency, Availability, Partition Tolerance	Konsistenca, razpoložljivost in odpornost na razdelitve v omrežju
CDH	Cloudera Distribution Including Apache Hadoop	Distribucija Cloudera, ki vsebuje Hadoop
CmRDT	Convergent Replicated Data Types	Konvergentni podvojeni podatkovni tipi
CRDT	Convergent or Commutative Replicated Data Types	Konvergentni ali komutativni podvojeni podatkovni tipi
CSV	Comma-Separated Values	Vrednosti ločene z vejico
CvRDT	Commutative Replicated Data Types	Komutativni podvojeni podatkovni tipi
DDL	Data definition language	Jezik za definiranje podatkov
EC2	Elastic Compute Cloud	Prožni računski oblak
GPG	GNU Privacy Guard	Varovanje zasebnosti GNU
HDFS	Hadoop Distributed File System	Porazdeljen datotečni sistem Hadoop

Kratica	angleško	slovensko
HTTP	HyperText Transfer Protocol	Protokol za prenos hiperteksta
JSON	Javascript Object Notation	Način zapisa Javascript objektov
LTS	Long Term Support	Dolgoročna podpora
SSH	Secure shell	Varna lupina
URL	Uniform Resource Locator	Enolični krajevnik vira
VPC	Virtual Private Cloud	Navidezni zasebni oblak
WAL	Write ahead log	Dnevnik z vnaprejšnjim pisanjem

Povzetek

Naslov: Štetje dogodkov s porazdeljenimi podatkovnimi tipi za dosego močne zakasnjene konsistentnosti

Sodobne spletne aplikacije se soočajo z vse večjim obsegom spletnega prometa. Ker morajo takšne aplikacije hraniti vedno več s tem povezanih podatkov, se je v zadnjih letih razvilo več pristopov k hranjenju podatkov za različne namene. V diplomskem delu raziščemo problem štetja dogodkov in zanesljivega shranjevanja števcev v okoljih, kjer sta pomembna visoka razpoložljivost in skalabilnost. Z uporabo števcev je mogoče rešiti različne probleme, ki se pojavljajo v spletnih aplikacijah. V delu s simulacijo prometa primerjamo tri različne podatkovne baze, ki jih je mogoče uporabiti za shranjevanje števcev, in ugotovimo, da lahko s tipi CRDT učinkovito rešimo problem štetja.

Ključne besede: porazdeljeni sistemi, podatkovne baze, štetje dogodkov, CRDT, MongoDB, HBase.

Abstract

Title: Counting events with distributed data types to achieve strong eventual consistency

Modern web applications face an ever increasing amount of web traffic. The requirement to store data of this traffic at an increasing rate lead to development of a large number of database solutions for different purposes. We explore the problem of counting events and reliably storing counters in environments where high availability and scalability are a requirement. Counters can be used to solve different problems in web applications. We evaluate three different distributed database solutions that can be used to store counters by simulating network traffic and conclude that CRDT data types can be an effective solution to the problem of counting.

Keywords: distributed systems, databases, counting events, CRDT, MongoDB, HBase.

Poglavje 1

Uvod

1.1 Namen dela

Namen tega diplomskega dela je raziskati področje pristopov podatkovnih baz k štetju dogodkov v porazdeljenih sistemih z velikim obsegom zahtev. Zanimajo nas zagotovila, ki nam jih takšne podatkovne baze ponujajo, in njihov vpliv na razpoložljivost sistema ter pravilnost rezultata.

V delu predstavimo hipotezo, da lahko z uporabo porazdeljenih podatkovnih tipov CRDT zagotovimo visoko razpoložljivo rešitev, ki prav tako zagotavlja pravilnost podatkov.

1.2 Motivacija

Pri načrtovanju spletnih aplikacij se pogosto srečujemo s problemom štetja. Zanima nas lahko število prikazov spletne strani, število sledilcev v socialnem omrežju ali količina porabljenih sredstev v spletnem oglaševanju. Poznavanje natančne količine dogodkov je pogosto ključnega pomena za uspeh podjetja na trgu. Članka [28, 27] omenjata sicer drugačne pristope od pristopa, opisanega v tem delu za problem štetja pri velikem obsegu zahtev.

V kolikor je sistem, v katerem ta problem rešujemo, majhen, je to mogoče storiti z uporabo relacijskih podatkovnih baz. Takšen pristop postane po-

manjkljiv, ko v sistemu naraste število zahtev in obseg podatkov [29]. Druga možnost je pisanje v dnevniške datoteke ali nestrukturirano v podatkovne baze tipa "key-value". S tem pristopom je pisanje hitro in skalabilno, za branje pa je podatke potrebno obdelati. Pogosto se v ta namen uporablja pristop MapReduce.

Če pa morajo biti podatki takoj dostopni za branje, je treba sproti vzdrževati stanje, kar v porazdeljenih sistemih pripelje do sklepanja določenih kompromisov. V takšnih sistemih posamezne enote komunicirajo med sabo preko računalniških omrežij, ki niso vedno zanesljiva in tako ne morejo zagotavljati enakih lastnosti pri branju in pisanju, kot jih lahko strojna oprema v samostojnem računalniku.

V tem diplomskem delu bomo raziskali področje rešitev, ki omogočajo skalabilno pisanje in hkrati branje v realnem času za podatkovni tip števcev.

1.3 Cilji

Cilji tega dela so ovrednotiti sisteme in pristope, ki so primerni za namen štetja dogodkov s simulacijo obremenitve v izbranih problemskih domenah. Pristope bomo ovrednotili glede na:

- zakasnitev (čas, ki preteče med začetkom in koncem zahteve na strežnik),
- razpoložljivost (delež zahtev, ki se uspešno obdelajo) in
- pravilnost (delež uspešnih zahtev, za katere podatki ostanejo zapisani, saj se lahko v nekaterih primerih zgodi, da pride do izgube podatkov).

1.4 Struktura diplomske naloge

Diplomska naloga je sestavljena iz sedmih poglavij. Po prvem, uvodnem poglavju je v poglavju 2 predstavljena problematika štetja v spletnih aplikacijah. V poglavju 3 obravnavamo lastnosti porazdeljenih sistemov, v poglavju 4 pa podatkovne tipe CRDT. V poglavju 5 sledi podroben opis podat-

kovnih baz (Dynamo in Riak, Bigtable in HBase ter MongoDB). Poglavje 6 je namenjeno obremenitvenim preizkusom s simulacijo zahtev na strežnike s podatkovnimi bazami. V tem poglavju je opisana vzpostavitev okolja za izvedbo obremenitvenih preizkusov in rezultati. V poglavju 7 so navedene sklepne ugotovitve.

Poglavje 2

Štetje v spletnih aplikacijah

V diplomskem delu nas zanimajo porazdeljene podatkovne rešitve, uporabne za namen štetja dogodkov. Štetje je pomembno v spletnih aplikacijah z velikim obsegom zahtev, še posebej ko je potrebno v realnem času zagotoviti dostop do analitičnih podatkov.

Štetje dogodkov je enostavno v aplikacijah, kjer je obremenitev nizka in je za shranjevanje podatkov mogoče uporabiti eno izmed relacijskih podatkovnih baz. Relacijske podatkovne baze ponujajo močno konsistentonst in atomične povečave števcev v poljih s številkami.

Problem pri relacijskih podatkovnih bazah nastopi, ko naraste količina zahtev na enoto časa in s tem obremenitev sistema. V tem primeru je potrebno zmogljivosti sistema povečati. Ta postopek se imenuje *skaliranje* [18]. Obstajata dva načina skaliranja — *vertikalno* in *horizontalno*.

Vertikalno skaliranje je povečevanje zmogljivosti strežnika. To je lahko uporaba bolj zmogljivega centralnega procesorja ali uporaba večje količine pomnilnika. Pogosto je to najprimernejši način za skaliranje relacijskih podatkovnih baz, njegova pomanjkljivost pa je, da je najvišja možna zmogoljivost posameznega strežnika navzgor omejena s trenutno stopnjo razvoja tehnologije. Prav tako z višjo zmogljivostjo narašča cena za povečanje zmogljivosti za enako stopnjo kot pri manj zmogljivih sistemih.

Pri horizontalnem skaliranju v primerjavi z vertikalnim zgornja meja zmo-

gljivosti ne obstaja. Pri tem načinu se zmogljivost sistema povečuje z dodajanjem strežnikov v sistem, tako da se delo razdeli med posameznimi strežniki. Ta način se pogosto uporablja v sodobnih podatkovnih centrih, kjer se zaradi najbolj ugodne cene na enoto v strežnikih uporablja običajna strojna oprema (angl. commodity hardware). Relacijske podatkovne baze omogočajo horizontalno skaliranje, ampak je to pogosto zelo omejeno zaradi zagotovil ACID [19]. NoSQL podatkovne rešitve, ki so postale zanimive z razvojem svetovnega spleta, namenoma olajšajo nekatera zagotovila, da lahko omogočijo horizontalno skalabilnost.

V delu smo se zato osredotočili na primerjavo porazdeljenih rešitev. V sodobnih spletnih aplikacijah sta pomembna *skalabilnost* in *zakasnitev*. Porazdeljene rešitve so za ta namen bolj primerne.

Za primerjavo smo izbrali podatkovne baze HBase [20], Riak [21] in MongoDB [22]. Te baze so med sabo po načinu delovanja različne in ponujajo različna zagotovila, kot je opisano v poglavjih 5.1, 5.2 in 5.3. Za njihovo primerjavo pa smo se odločili zaradi pogoste uporabe v industriji in zato, ker so primerne za namen štetja dogodkov.

Zanimala sta nas njihova konsistentnost in razpoložljivost, kot ju opredeljuje izrek CAP (podrobnosti so v poglavju 3.4). Za ovrednotenje teh lastnosti sistemov smo naredili tudi preizkuse z napakami v omrežju.

Poglavje 3

Porazdeljeni sistemi

3.1 Zmote pri porazdeljenem računalništvu

Načrtovanje in implementacija porazdeljenih sistemov sta zahtevni. Že programiranje enega računalnika postane zahtevnejše, če v programu želimo uporabiti več niti, ki si delijo pomnilnik. Pri porazdeljenih sistemih prihaja do podobnih problemov, komunikacija med računalniki pa poteka preko nezanesljivega omrežja, namesto preko deljenega pomnilnika. Peter Deutsch je zmote, povezane s porazdeljenimi sistemi, strnil v sedem opazk [13]:

1. Omrežje je zanesljivo.
2. Zakasnitev je nična.
3. Pasovna širina je neskončna.
4. Omrežje je varno.
5. Topologija se ne spreminja.
6. Obstaja samo en oskrbnik omrežja.
7. Omrežje je homogeno.

To so pogosto predpostavke nekoga, ki se s porazdeljenimi sistemi srečuje prvič.

3.2 Tipi napak v porazdeljenih sistemih

Načrtovalci porazdeljenih sistemov se morajo zavedati napak, ki se lahko pojavijo v takšnih sistemih. V literaturi [16, 26] se pogosto pojavaljajo naslednji tipi napak:

- napake v omrežju,
- zrušenje vozlišča,
- performančne težave,
- bizantinske napake.

Napake v omrežju se omenjajo v kontekstu razdelitev izreka CAP, poleg njih pa imajo lahko vozlišča performančne težave, lahko pa pride tudi do popolnega zrušenja posameznega vozlišča. Zrušenje predstavlja manjšo težavo za porazdeljene sisteme kot razdelitve v omrežju, saj ob njem vozlišče preneha odgovarjati na zahteve. *Bizantinske napake* so poseben tip napak, ki se pojavijo, ko se začnejo posamezna vozlišča obnašati v neskladju s pravilnim delovanjem sistema. Ponavadi porazdeljeni sistemi niso odporni na ta tip napak, ker so algoritmi, ki jih rešujejo kompleksni in dragi za implementacijo [16].

3.3 Tipi konsistentnosti

V porazdeljenih sistemih se pogosto srečujemo s pojmom konsistentnosti. V literaturi se ta beseda pogosto uporablja za sorodne, ampak ne enake pojme. V kontekstu zagotovil ACID je konsistentnost zagotovilo, da vsaka transakcija povzroči spremembo stanja iz enega veljavnega stanja v drugo [19]. V tem kontekstu je pomen vezan predvsem na relacijsko shemo.

Pomen besede konsistentnost, ki se pogosteje uporablja v kontekstu porazdeljenih sistemov in tudi v kontekstu izreka CAP, je opredeljen s pojmom modela konsistence. Model konsistence je zagotovilo sistema, da so operacije

na podatkovno shrambo predvidljive, če programer sledi določenim pravilom [16]. V tem kontekstu se obravnava tudi konsistentnost v večprocesorskih sistemih pri dostopu do pomnilnika.

Tipe modelov konsistence v porazdeljenih sistemih je mogoče razdeliti na *močne* in *šibke*. Avtorji v [12] opisujejo najpomembnejše modele močne konsistence. To sta *linearizabilnost* (angl. linearizability) in *sekvenčna konsistenca*. Modeli konsistence ponavadi na podlagi enostavnih modelov opišejo delovanje sistema. Tako se pogosto analizira pisanje sočasnih pisanj in branj v register. Register je mišljen kot bralno-pisalna lokacija v pomnilniku.

Model sekvenčne konsistentnosti zahteva, da z vidika procesov izgleda, da so se vsa pisanja zgodila atomično in da je zaporedje pisanj konsistentno s tem, kar vidijo posamezni procesi, ter da je zaporedje enako na vseh vozliščih [16].

Model linearizabilnosti še dodatno zahteva, da je zaporedje pisanj konsistentno z globalnim zaporedjem pisanj v resničnem času.

Če bi vozlišča poznala točen globalni čas, bi bilo mogoče zagotoviti močno konsistentnost na enostaven način. Ker to ni mogoče zaradi lastnosti omrežja in fizikalnih lastnosti (informacije potujejo s končno hitrostjo), je potrebno za zagotovitev te lastnosti uporabiti algoritme, ki za izpolnitev močne konsistentnosti žrtvujejo visoko razpoložljivost.

Modeli konsistence, ki niso močni, spadajo med šibke. V našem delu nas zanimata predvsem *model konsistence s stanjem mirovanja* (angl. quiescent consistency) in *zakasnjena konsistenca* (angl. eventual consistency).

Model konsistence s stanjem mirovanja zahteva, da se ohrani vrstni red pisanj, ko so ta ločena s poljubno dolgim stanjem mirovanja. Če na primer dva procesa zapišeta neko vrednost v register in sta pisanji sočasni, nato pa nastopi stanje mirovanja, po katerem tretji proces zapiše neko novo vrednost, mora biti ob naslednjem branju vrnjena ta vrednost ali vrednost, ki je bila zapisana kasneje. Vrstni red pri sočasnih zapisih ni določen. To je model konsistence, ki ga podpirajo tipi CRDT [11].

Zakasnjena konsistenca zahteva samo, da ob prenehanju pisanj v sistem

ta po določenem času konvergira k nekemu skupnemu stanju.

3.4 Izrek CAP

Pri ovrednotenju primernosti uporabe porazdeljenih sistemov predstavlja pomembno izhodišče izrek CAP. O njem je (takrat samo kot o domnevi) prvi govoril Eric A. Brewer [2], kasneje pa sta njegov dokaz objavila S. Gilbert in N. Lynch [3].

Izrek govori o visoki razpoložljivosti, konsistentnosti in odpornosti na razdelitve v omrežju; treh garancijah, ki jih spletne storitve lahko zagotavljajo. *Razpoložljivost* (angl. availability) je definirana kot sposobnost sistema, da odgovori na vsako prejeto zahtevo. Ta lastnost je v sodobnih sistemih zaželjena, ker zagotavlja dobro uporabniško izkušnjo.

Konsistentnost izrek CAP razume kot linearizabilnost. Sistem, ki izpolnjuje zahteve linearizabilnosti, zagotavlja, da bo po poljubni operaciji pisanja neke vrednosti vsako branje po zaključeni operaciji pisanja vrnilo to ali novejšo vrednost. Takšen sistem je intuitiven, zaradi česar je z njim lažje delati med programiranjem odjemalskih aplikacij.

Tretja lastnost je *odpornost na razdelitve* (angl. partition tolerance). To je sposobnost sistema, da se v primeru izpada omrežja obnaša vnaprej definirano. Je ena izmed lastnosti odpornosti sistema na okvare (angl. fault tolerance).

Izrek CAP pravi, da lahko v poljubnem porazdeljenem sistemu naenkrat zagotovimo samo dve od teh lastnosti. A ker moramo zagotoviti odpornost na razdelitve v sistemu, da bo delovanje sistema definirano, je potrebno skleniti kompromis med razpoložljivostjo in konsistentnostjo. Do tega pride, ker je za vzdrževanje nekega stanja potrebno najmanj kvorum vozlišč v gruči, če želimo, da je pri pisanju zagotovljena konsistentnost. V primeru razdelitve v omrežju bo moral del gruče, ki kvoruma ne bo imel, čakati, da se povezave med vozlišči obnovijo, preden bo lahko začel sprejemati zahteve za pisanje. Če izberemo visoko razpoložljivost kot pomembnejšo lastnost in dovolimo

Slika 3.1: Razmerje med visoko razpoložljivostjo, konsistentnostjo in odpornostjo na razdelitve (CAP).

pisanje tudi med razdelitvami v omrežju, bo na vozliščih v različnih delih gruče lahko prišlo do sočasnega pisanja v isto polje, s čimer lahko izgubimo konsistentnost, ker lahko pisanje, ki se je sicer zgodilo prej, ob obnovitvi povezave dobi prednost in prepiše novejšo (pravilno) vrednost v tem polju.

Na sliki 3.1 je prikazano razmerje med lastnostmi CAP v porazdeljenih sistemih. Lastnosti, ki jih je mogoče zagotoviti, so navedene v presečiščih.

3.5 Replikacija v podatkovnih bazah

Nekatere podatkovne baze imajo podporo za replikacijo podatkov na več vozliščih in s tem tudi podporo za skaliranje. Pri podatkovnih bazah obstajata dva pristopa k replikaciji, način *master-master* in *master-slave*.

Pristop master-master ima več glavnih strežnikov. Do vsakega od teh

strežnikov imajo uporabniške aplikacije dostop za pisanje. Za potrjevanje transakcij se v tem primeru med strežniki pogosto uporablja protokol dvosmerne potrditve. Ta protokol ima pomanjkljivost, da ni odporen na izpade v omrežju in lahko pride do zastoja, ko se transakcije ne uspejo prenesti med strežniki.

Pri pristopu master-slave v sistemu obstaja samo en glavni strežnik za pisanje, medtem ko se na ostale samo prenašajo posodobitve. Ta pristop je primeren, če je pisanje redka operacija, branje pa pogosta. Tako se lahko obremenitev uporabniških aplikacij izenačuje med več strežniki.

Poglavje 4

Podatkovni tipi CRDT

Podatkovni tipi CRDT [11] predstavljajo celovit pristop k *zakasnjeni konsistenci* v porazdeljenem okolju. Zasnova takšnih tipov temelji na preprostih formalnih pogojih, ki so zadostni za zagotavljanje zakasnjene konsistence.

Objekti tipa CRDT so porazdeljeni na več vozlišč. Vrednost objekta na enem vozlišču se imenuje replika. Uporabniške aplikacije spreminjajo stanje replike z izvajanjem operacij, ki jih tip ponuja na vmesniku. Pisanje se zgodi v dveh korakih. Uporabniška aplikacija najprej izvede operacijo na poljubni repliki, ki se imenuje izvorna replika. Nato ta replika asinhrono posreduje spremembo stanja ostalim replikam.

V članku [11] sta opisana dva pristopa k replikaciji pri tipih CRDT. Pri prvem se med replikami izmenjuje celotno stanje objekta. Izvorna replika pošlje prejemni repliki trenutno stanje. Prejemna replika nato izvede operacijo *združevanja*, ki sprejme kot vhodna argumenta stanje v izvorni repliki ter stanje v prejemni repliki. Rezultat te operacije je novo stanje te prejemne replike. Podatkovni tipi s takšnim načinom replikacije se imenujejo CvRDT, ker operacija združevanja konvergira k enotnemu stanju med replikami. Ko se izvedejo posodobitve med vsemi replikami, je v vseh zapisano enako stanje. Primer takšnega načina replikacije je pošiljanje najnovejše vrednosti števca pri podatkovnem tipu števca. Pogoj za delovanje tega načina replikacije je, da vsaka posodobitev doseže vzročno zgodovino vsake replike.

To je mogoče doseči s sistemom, ki ob nedoločenem času neskončno pogosto posodablja stanje med pari replik. Komunikacija med replikami mora tvoriti povezan graf. Zahteve za kanal so pri tem načinu replikacije nizke. Operacije združevanja so komutativne in idempotentne, kar zagotavlja pravilno končno stanje tudi ob izgubljenih in podvojenih sporočilih ali sporočilih v napačnem vrstnem redu.

Drug način replikacije temelji na izmenjevanju operacij, ki so se izvedle na podatkovnem tipu. Operacija posodabljanja vrednosti je tako razdeljena na dve fazi. Prva faza se zgodi na izvorni repliki in se ob izpolnjenem predpogoju izvede brez stranskih učinkov. Takšni tipi CRDT se imenujejo tudi CmRDT, ker morajo operacije biti komutativne. Primer je preverjanje, ali element obstaja v množici pri operaciji odstranjevanja elementa. V primeru, da ne obstaja, se druga faza ne bo zgodila. Druga faza je posodobitev stanja. Najprej se zgodi na izvorni repliki, nato pa še na vseh ostalih. Primer replikacije z izmenjevanjem operacij je števec, ki se lahko povečuje ali zamnjšuje, kjer se med vozlišči prenese samo operacija povečevanja ali zmanjšanje števca. Zahteve za kanal so močnejše kot pri tipih, temelječih na izmenjevanju stanja. Za ta način replikacije mora imeti kanal lastnosti zanesljivega oddajanja.

Načina replikacije sta si v nekaterih pogledih podobna, v nekaterih pogledih pa se razlikujeta. O načinu z izmenjevanjem stanja je lažje razmišljati in ima šibkejše zahteve za kanal. Pomanjkljivost je velikost prenešenega stanja, saj se ta lahko povečuje z večjih številom posodobitev za nek objekt. Nasprotno je pri načinu temelječem na izmenjevanju operacij potrebno ob replikaciji izmenjati manj informacij, ima pa močnejše zahteve za komunikacijski kanal.

Članek [11] opisuje več različnih možnih tipov CRDT. To so števci, ki so obravnavani v tem delu, množice, registri, grafi in zaporedja. Semantika, ki jo podpirajo posamezni tipi, je zaradi lastnosti asinhrone replikacije včasih pomanjkljiva v primerjavi s temi tipi v neporazdeljenem okolju. Primer takšne implementacije je porazdeljena množica, ki zaradi nekomutativnih operacij dodajanja in brisanja elementa ne more podpirati popolne semantike množice. Predstavljene implementacije tako dajejo prednost do-

dajanju ali brisanju elementa.

4.1 Števci CRDT

Števec je podatkovni tip, ki podpira operaciji *povečevanja* in *zmanjševanja*. V članku [11] je opisanih več možnih implementacij števcev.

4.1.1 Števec, temelječ na replikaciji z izmenjevanjem operacij

Števci, temelječi na replikaciji z izmenjevanjem operacij, so enostavni. V replikah se nahaja lokalni števec, ki se povečuje ali zmanjšuje glede na prihajajoče operacije preko omrežja. Ta način potrebuje kanal z lastnostjo zanesljivega oddajanja.

4.1.2 G-števec

G-števec je podatkovni tip, ki temelji na replikaciji z izmenjevanjem stanja. Ta tip zaradi poenostavitve podpira samo operacijo povečevanja. Pri tem števcu je stanje v repliki seznam celih števil. Vsaki repliki je dodeljena svoja celica v tem seznamu. Replika ob zahtevi za povečanje poveča samo število v svoji celici. Operacija združevanja stanj med replikami združi dva takšna seznama tako, da vzame maksimum v vsaki celici. Ob poizvedbi replika vrne seštevek vseh celic. Ta način predpostavlja, da je množica replik znana vnaprej, saj mora obstajati ena celica v seznamu za vsako repliko.

4.1.3 PN-števec

PN-števec doda podporo za zmanjševanje. Te ni mogoče dodati enostavno tako, da bi podprli zmanjševanje v G-števcu. Eden izmed razlogov je, da operacija združevanja stanj vzame maksimum vseh vrednosti. Če bi bila operacija zmanjševanja števcev dovoljena, ta ne bi imela vpliva ob posodabljanju replik, saj bi vedno obveljale vrednosti pred zmanjševanjem.

Zato je PN-števec sestavljen iz dveh seznamov. V seznamu P se nahajajo povečave, medtem ko so v seznamu N zmanjšanja števca. Operacija združevanja stanj replik vzame maksimum vsake celice obeh seznamov. Za streženje poizvedbam je potrebno sešteti seznama in nato od sešteveka seznama P odšteti seštevek seznama N.

4.2 Ostali podatkovni tipi

Poleg števcev so v članku [11] predstavljeni tudi podatkovni tipi registrov, množic, grafov in poseben tip za hranjenje zaporedja, primeren za skupno urejanje besedil. Za impelementacijo množic obstaja več načinov, omenjene so množice tipov G, 2P, U, LWW, PN in OR. Vsak od teh tipov ponuja nekoliko drugačno semantiko pri uporabi. V množico tipa G se lahko elementi samo dodajajo. V množico tipa 2P se lahko posamezen element enkrat doda in odstrani, po tem pa se ne more več odstraniti. Množica tipa U je primerna, ko so elementi edinstveni. Množica tipa LWW deluje na podlagi časovnih oznak operacij. Množica tipa PN za vsak element hrani števec, s katerim hrani informacijo o prisotnosti elementa v množici. Množica tipa OR za vsako operacijo določi edinstveno oznako, ki ni vidna odjemalskim aplikacijam.

Poglavje 5

Pregled NoSQL podatkovnih baz

V tem poglavju predstavimo NoSQL podatkovne baze, ki jih v delu raziskujemo in primerjamo. To so podatkovne baze Dynamo in Riak, ki je nastal na podlagi Dynamo, BigTable in HBase, ki je nastal na podlagi BigTable ter MongoDB. Od teh predstavimo Dynamo in BigTable samo za namen opisa ostalih podatkovnih baz.

5.1 Dynamo in Riak

Podatkovna baza Riak [21] je odprtokodna rešitev podjetja Basho. Nastala je kot implementacija principov sistema Dynamo, opisanega v članku podjetja Amazon [4]. Članek je osnova za več odprtokodnih implementacij podatkovnih baz. Poleg podatkovne baze Riak, sta to tudi Cassandra [23] in Voldemort [24]. Dynamo se je na začetku uporabljal kot interna rešitev v podjetju Amazon, kasneje pa je na tej podlagi nastala tudi komercialna storitev v okviru Amazon Web Services, DynamoDB.

Sistem Dynamo je bil načrtovan za aplikacije, ki potrebujejo zanesljivo delovanje in kratek odzivni čas. V članku [4] je ta opredeljen kot največ nekaj 100 milisekund za 99,9-ti percentil zahtev. Glede zanesljivosti je bila

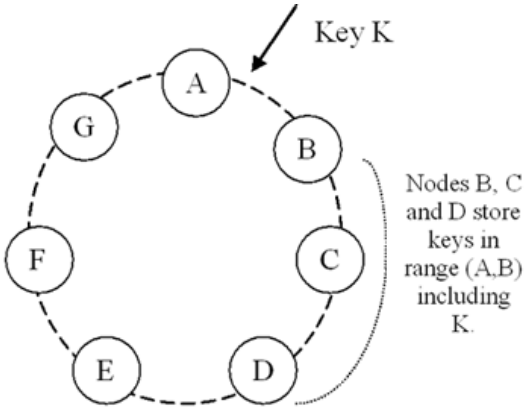
podana zahteva, da mora biti sistem v vsakem trenutku dostopen za pisanje in zahtev za pisanje ne sme zavračati. Na zasnovo pa je pomembno vplivala tudi enostavnost pri upravljanju, dodajanju in odstranjevanju vozlišč.

Dynamo je visoko razpoložljiv sistem, ki podpira poizvedbe po principu ključ-vrednost (angl. key-value). Vsaka poizvedba izvrši operacijo branja ali pisanja za posamezen ključ neodvisno od ostalih ključev. Tak način ne omogoča kompleksnih poizvedb, ki so mogoče v relacijskih podatkovnih bazah. Za sistem Dynamo sta vsak ključ in vrednost samo polje bajtov brez posebnega pomena. Ker je Dynamo načrtovan za aplikacije, ki takšnih poizvedb ne potrebujejo, je ta lastnost sprejemljiva.

Gre za popolnoma decentraliziran sistem, sestavljen iz avtonomnih vozlišč, ki lahko neodvisno eno od drugega obdelujejo zahteve. Podatki so med vozlišči razdeljeni po principu *konsistentnega razprševanja* (angl. consistent hashing). Zaloga vrednosti razpršitvene funkcije, ki se za to uporablja, se obravnava kot fiksen krožni prostor ali "obroč". Ob dodajanju novega vozlišča v sistem, se za le-to določi prostor na obroču. Mesto posameznega podatka v sistemu se na podlagi njegovega ključa določi tako, da se z razpršitveno funkcijo določi njegovo mesto na obroču. Iskano vozlišče je prvo v smeri urinega kazalca. Za namen replikacije je podatek shranjen še na naslednjih $N-1$ vozliščih, kjer je N nastavitev števila vozlišč, ki hranijo repliko posameznega podatka. Množica takšnih vozlišč sestavlja *preferenčni seznam* za posamezen ključ. Z uporabo konsistentnega razprševanja sta rešena tako *skalabilnost* kot *trajnost* podatkov. Primer razporejanja podatkov po obroču je prikazan na sliki 5.1.

Zahteve se, v primeru, da pridejo na vozlišče, ki ga ni na preferenčnem seznamu, posredujejo na vozlišče, ki je najvišje na tem seznamu. Če so vsa vozlišča na preferenčnem seznamu nedosegljiva, je zahteva posredovana naslednjemu vozlišču na obroču. Ker tako vozlišče ni na preferenčnem seznamu, vrednosti hrani samo dokler vozlišče, za katerega je bila zahteva namenjena, ne postane dosegljivo.

Prilagoditev želene stopnje konsistentnosti omogoča nastavitev parame-



Slika 5.1: Primer razporejanja podatkov po obroču.[1]

trov W in R. W je število vozlišč, na katere se sinhrono replicira podatek preden je zahteva za pisanje potrjena. R pa je število vozlišč, ki morajo vrniti zahtevan podatek, da je zahteva uspešna. Nastavitev $R = W = 1$ je s stališča aplikacije najhitrejša, ampak zagotavlja najnižjo stopnjo konsistentnosti, nastavitev $R = W = N$ pa na račun razpoložljivosti ob primeru okvar v sistemu zagotavlja najvišjo možno stopnjo konsistentnosti. Z uporabo nastavitve $R+W > N$ je mogoče doseči približek konsistentnosti kvoruma, kjer v primeru, ko v sistemu ni okvar, ta vrne zadnjo zapisano vrednost. Vseeno lahko v primeru okvar še vedno pride do dostopov do vozlišč, ki niso na preferenčnem seznamu, in v tem primeru ne more obstajati zagotovilo, da bo ob naslednjem branju vrnjena nazadnje zapisana ali novejša vrednost, kar bi sicer zagotavljal kvorum.

Ker se pisanje v Dynamo replicira asinhrono in ker vsa vozlišča hkrati sprejemajo zahteve za pisanje, je sistem *zakasnjeno konsistenten*. Za zagotavljanje konsistentnosti med vozlišči Dynamo uporablja sistem vodenja različic objektov z uporabo *vektorskih ur*. Vozlišče za vsak objekt vodi tre-

[1]Vir: www.allthingsdistributed.com `http://www.allthingsdistributed.com/2007/10/amazons_dynamo.html`

nutno številko različice. To omogoča, da sistem zazna spremembe, ki so se zgodile sočasno. V takšnem primeru se je zgodil konflikt pri pisanju in odločiti se je potrebno, katero od dveh ali več različic uporabiti. V kolikor je ena različica prednik druge, jih Dynamo lahko samodejno združi, sicer mora to odločitev prepustiti aplikaciji. Ta mora biti programirana na način, da to omogoča. Zaradi tega nekatere implementacije uporabljajo enostavnejši način "Last writer wins", ki izbere tisto posodobitev, ki ima najnovejšo časovno oznako. Nevarnost tega pristopa je, da lahko pripelje do stanja, ki ni zadnje, ima pa najnovejšo časovno oznako zaradi razlik med sistemskimi urami na vozliščih. Način usklajevanja s časovnimi oznakami je edini, ki ga ponuja Cassandra, Riak pa omogoča izbiro med zgoraj opisanim načinom z vektorskimi urami ter načinom s časovnimi oznakami.

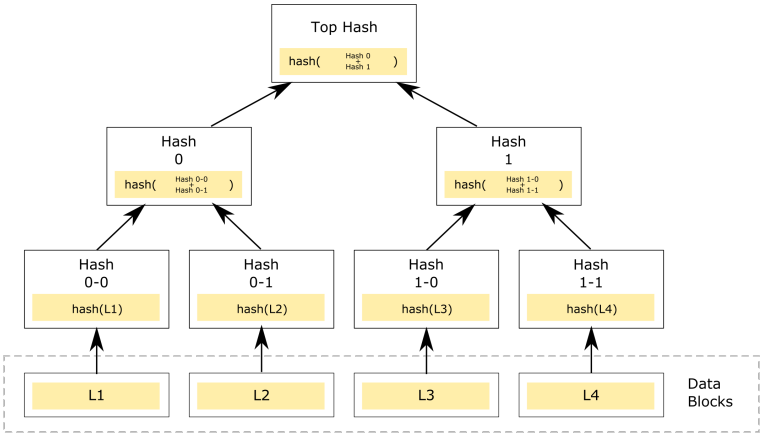
Pri branju iz večih vozlišč hkrati (odvisno od števila W) Dynamo popravlja zastarele vrednosti na vozliščih. Obenem obstaja tudi sistem proti entropiji, ki periodično posodablja podatke med različnimi vozlišči. Za delovanje uporablja drevesa Merkle, s katerimi je na učinkovit način mogoče ugotoviti razliko med podatki na vozliščih. V končnih vozliščih dreves Merkle se nahajajo sami podatki, vozlišča na višjih nivojih pa so zgoščene vrednosti njihovih poddreves. S primerjavo zgoščenih vrednosti je mogoče ugotoviti mesto, kjer se podatki dveh vozlišč v sistemu Dynamo razlikujejo. Primer drevesa Merkle je prikazan na sliki 5.2.

5.1.1 Tipi CRDT v podatkovni bazi Riak

Podatkovna baza Riak podpira podatkovne tipe CRDT pod imenom Data Types. Podprtih je pet različnih tipov [17]:

- zastavice,
- registri,
- števci,
- množice in

- preslikovalne tabele.



Slika 5.2: Primer drevesa Merkle.[2]

Zastavice so Boolove vrednosti, ki imajo lahko resnično ali neresnično vrednost. Registri so poimenovane dvojiške vrednosti. Zastavice in registri se ne morejo uporabljati samostojno, lahko so samo del preslikovalnih tabel. Števci so tipa PN, množice pa tipa 2P (opisano v poglavju 4.2). V preslikovalnih tabelah je mogoče uporabiti katerikoli tip kot vrednost.

V Riaku je potrebno za uporabo podatkovnih tipov ustvariti koš in mu nastaviti podatkovni tip. Vsak koš ima lahko samo en podatkovni tip.

5.2 Bigtable in HBase

5.2.1 Bigtable

Bigtable je porazdeljen sistem podjetja Google za upravljanje s strukturiranimi podatki [5]. Načrtovan je za podporo aplikacijam z zelo različnimi zahtevami glede podatkovne shrambe, od takšnih, ki potrebujejo nizek od-

[2]Vir: Wikipedia `https://en.wikipedia.org/wiki/Merkle_tree#/media/File:Hash_Tree.svg`

zivni čas, do takšnih, ki shranjujejo velike količine podatkov. Primeren ja za velikosti gruč od nekaj do več tisoč strežnikov.

Podatkovni model, ki ga podpira Bigtable, je enostavnejši od relacijskih. Zasnovan je kot razpršena, večdimenzionalna in urejena preslikovalna tabela. Podatki so razporejeni v vrstice in stolpce. Indeksirani so po imenih vrstic in stolpcev, ki so poljubna polja bajtov. Vsako branje in pisanje podatkov v eni vrstici je nedeljiva operacija ne glede na število stolpcev v zahtevi. Ta lastnost olajša razumevanje, kaj se v sistemu zgodi med sočasnimi posodobitvami.

Bigtable vzdržuje vrstice na disku v leksikografskem vrstnem redu. Na ta način zagotavlja boljšo lokalnost podatkov ob branju zaporednih vrstic. Vrstice so namreč razdeljene med različna vozlišča tako, da vsako od njih upravlja z obsegom zaporednih vrstic, ki ga avtorji članka [5] imenujejo *tablica* (angl. tablet). Razdeljevanje je dinamično — razporeditev med vozlišči se z naraščanjem velikosti tabele spreminja, kar omogoča učinkovito izenačevanje obremenitve. Za novo tabelo na začetku obstaja samo ena tablica, ki se po preseženi določeni velikosti (med 100 in 200 MB) začne deliti.

Stolpci so v tabelah združeni v množice imenovane *družine stolpcev* (angl. column families). Ob branju in pisanju je potrebno za vsak stolpec določiti družino. Te je potrebno ustvariti vnaprej in pričakovano je, da je njihovo število manjše (največ nekaj 100), medtem ko število samih stolpcev na delovanje sistema ne vpliva in je neomejeno.

Poleg ključa vrstice in stolpca imajo podatki še določeno *časovno oznako*. Na ta način Bigtable omogoča branje starejših podatkov. Da je upravljanje s podatki bolj obvladljivo, sta na voljo dve nastavitvi, ki upravljata samodejno čiščenje zastarelih podatkov. Z eno nastavitvijo je mogoče prilagoditi število preteklih različic, ki jih sistem ohrani, z drugo pa, koliko časa hrani posamezne različice. Na ta način je mogoče doseči, da se podatki na primer ohranjajo en teden. Obe nastavitvi je mogoče nastaviti za vsako družino stolpcev posebej.

Bigtable za shranjevanje podatkov uporablja porazdeljen datotečni sistem GFS [6], ki že podvaja podatke in je na ta način odporen na okvare v

sistemu. Za vzdrževanje informacij o razporeditvi tablic v gruči, za zagotavljanje, da je v vsakem trenutku aktiven samo en glavni strežnik, in za vzdrževanje informacij o shemi (družinah stolpcev) uporablja sistem Chubby [7]. Chubby je porazdeljen sistem za zaklepanje, ki uporablja algoritem Paxos [8] za zagotavljanje konsistentnosti med vozlišči. Paxos je algoritem za reševanje problema konsenza v porazdeljenem okolju.

Gruča Bigtable strežnikov je sestavljena iz glavnega strežnika in ostalih strežnikov tablic, ki so odgovorni za tablice in podatke v njih. Glavni strežnik je odgovoren za razporejanje tablic med strežniki, dodajanje in odvzemanje strežnikov iz gruče, izenačevanje obremenitev med strežniki in za čiščenje datotek v GFS. Strežniki tablic obravnavajo zahteve za branje in pisanje ter skrbijo za razdeljevanje tablic, ko te postanejo prevelike. Vsakemu takemu strežniku je dodeljenih med nekaj deset in nekaj tisoč tablic.

Informacija o lokaciji podatka je shranjena na treh nivojih. Prvi nivo je *korenska tablica*. Ta se nahaja v sistemu Chubby. Vsebuje podatek o tem, kje se nahajajo tablice posebne tabele `METADATA`. Vsaka od teh tablic vsebuje podatek o lokaciji tablice za podan ključ. Ker se podatki o lokacijah ne spreminjajo pogosto, jih odjemalske aplikacije lahko shranijo in tako zmanjšajo število omrežnih zahtev, ki jih morajo narediti za eno branje ali pisanje.

Podatki so trajno shranjeni v datotekah oblike SSTable. Takšne datoteke vsebujejo nespremenljivo leksikografsko urejeno preslikovalno tabelo, v kateri so ključi in vrednosti shranjeni kot poljubna polja bajtov. Za pisanje in branje sistem uporablja medpomnilnik imenovan "memtable" preden podatke trajno zapiše v datoteko. To stori šele po preseženi določeni količini podatkov v medpomnilniku. Hkrati se spremembe trajno zapišejo v dnevniško datoteko.

5.2.2 HBase

Apache HBase je odprtokodna rešitev, zasnovana na principih iz članka Bigtable [5]. Razvija se kot del projekta Apache Hadoop. Sledi povzetek lastnosti HBase iz članka [9].

HBase za trajno shrambo podatkov uporablja datotečni sistem HDFS, ki

je prav tako del ekosistema Hadoop. HDFS je porazdeljen datotečni sistem, prilagojen za navadno strojno opremo [10]. Ena izmed njegovih lastnosti je odpornost proti okvaram, ki jo zagotavlja tako, da podatke podvaja na več vozliščih. V sistemu HBase je tako že na tem nivoju poskrbljeno za podvajanje podatkov.

HBase je, podobno kot Bigtable, sestavljen iz več različnih tipov strežnikov. Za dostop do podatkov uporabniške aplikacije uporabljajo strežnike RegionServer. Ti upravljajo z zaporednimi deli vrstic tabel, ki se v HBase imenujejo *regije*. Strežniki RegionServer se ponavadi nahajajo zraven HDFS strežnikov DataNode, ki skrbijo za podatke v datotečnem sistemu. Ta lastnost omogoča lokalnost podatkov in zmanjša število omrežnih zahtev, potrebnih za obdelavo posamezne zahteve. Ob prerazporejanju regij se lahko še vedno zgodi, da je regija dodeljena strežniku, ki podatkov nima na voljo lokalno. Ob dostopu do podatkov v takšni regiji je potreben omrežni promet. Za reševanje tega problema obstaja postopek *zgoščevanja*, ki prenese regije na ustrezna vozlišča.

Za usklajevanje med strežniki regij in upravljanje z gručo je odgovoren strežnik HMaster. Ta strežnik dodeljuje regije ob zagonu strežnikov regij in jih prerazporeja ob njihovem izpadu. Prav tako prerazporedi regijo, kadar količina podatkov v njej naraste preko določene meje. Njegova naloga je tudi streženje zahtevam DDL, s katerimi je mogoče ustvariti nove tabele in posodobiti ali izbrisati že obstoječe. V gruči je lahko več strežnikov HMaster, ampak je aktiven lahko samo eden. Ostali so v pripravljenosti. Ob izpadu aktivnega strežnika pa sistem ZooKeeper [25] izbere enega izmed njih, da prevzame njegovo mesto.

Za koordinacijo se uporablja sistem ZooKeeper, ki vzdržuje stanje o strežnikih prisotnih v sistemu. Je porazdeljen sistem, ki z uporabo konsenza med vozlišči zagotavlja enotno skupno stanje. Uporablja se za hranjenje informacije o tem, kateri strežniki v gruči so trenutno dosegljivi, in posreduje obvestila o okvarah strežnikov. Odgovoren je tudi za izbiranje aktivnega strežnika HMaster.

V primeru, da sistem ZooKeeper ne more več dostopati do regijskega strežnika, strežnik HMaster sproži postopek prerazporeditve regij, za katere je ta regijski strežnik skrbel, med ostale regijske strežnike.

Podatek o lokaciji posameznih regij se nahaja v posebni tabeli `META`, njena lokacija je shranjena v sistemu ZooKeeper. Podobno kot v sistemu BigTable mora uporabniška aplikacija izvesti tri poizvedbe, da pride do željenega ključa. Prva izmed teh je poizvedba v sistem ZooKeeper o lokaciji tabele `META`, druga je poizvedba na to tabelo na regijskem strežniku, kjer se nahaja. Tam je podatek o lokaciji regije iskanega ključa v gruči. S tretjo poizvedbo aplikacija dostopa do samih podatkov. Uporabniške knjižnice so implementirane tako, da si ob prvem dostopu do ključa v regiji zapomnijo lokacijo regije in tabele `META`.

Sistem HBase zagotavlja močno konsistentnost in zaradi tega po izreku CAP ob prisotnosti razdelitev v omrežju ne more zagotavljati visoke razpoložljivosti. Močno konsistentnost lahko zagotavlja zaradi dejstva, da je za poljuben podatek v sistemu odgovoren samo en strežnik, ki ima popoln nadzor nad branji in pisanji v svojih regijah. To omogoča nedeljive spremembe na posameznih vrsticah in onemogoča branje zastarelih podatkov. Takšen sistem ne zagotavlja visoke razpoložljivosti v primeru izpada vozlišča ali omrežja, saj med prerazporejanjem regij podatki niso na voljo.

Strežniki regij hranijo podatke, preden jih zapišejo v datotečni sistem, v strukturi imenovani MemStore, ki se nahaja v glavnem pomnilniku vozlišča. Za vsako družino stolpcev v tabeli vzdržuje strežnik regij eno strukture MemStore. Podatki so tako kot v datotekah urejeni po ključu vrstice. Ko ena izmed struktur MemStore preseže vnaprej določeno velikost, se podatki iz nje in vseh ostalih za tabelo zapišejo v datoteko v sistem HDFS imenovano HFile. Zraven pisanja v MemStore se podatki ob zahtevah za pisanje zapišejo tudi v dnevniško datoteko WAL, ki se prav tako nahaja v sistemu HDFS. Ker se v to datoteko vrstice samo dodajo, je pisanje vanjo hitrejše od pisanja v ostale datoteke. Podatki se zapišejo najprej v datoteko WAL in šele nato v strukturo MemStore. Datoteka WAL obstaja kot zaščita pred okva-

rami vozlišč. Ob okvari vozlišča je tako mogoče obnoviti stanje v pomnilniku pred okvaro iz kateregakoli drugega vozlišča, ker je datoteka podvojena v datotečnem sistemu HDFS.

5.3 MongoDB

Podatkovna baza MongoDB je dokumentno usmerjena. Ideja za njeno zasnovo je bila uporabiti temelje, ki so jih postavile relacijske podatkovne baze, in na njih ustvariti rešitev, ki je primerna za uporabo v sodobnih sistemih, ki se spreminjajo hitro in podpirajo horizontalno skaliranje, ter pri tem uporabiti inovacije, ki so jih prinesle NoSQL podatkovne rešitve [14].

Podatkovni model je zasnovan na osnovi *dokumentov*, ki so shranjeni v binarni datoteki oblike BSON. Ta je razširitev oblike JSON z dodanimi tipi, kot so cela števila, števila s plavajočo vejico in datumi. Dokument sestavlja eno ali več polj. Vsako polje ima svojo vrednost, ki je lahko nov dokument. Struktura dokumenta je dinamična — obstoja polj ni potrebno definirati vnaprej.

MongoDB pozna tudi koncept *zbirke*, ki združujejo dokumente s podobno strukturo. Koncept zbirke je podoben konceptu tabele v relacijskih podatkovnih bazah, koncept dokumenta je podoben konceptu vrstice v tabeli, koncept polja pa konceptu stolpca v vrstici.

V podatkovni bazi MongoDB je v nasprotju z relacijskimi podatkovnimi bazami priporočeno shranjevati vse podatke o enem zapisu v enem dokumentu. V relacijskih podatkovnih bazah bi podatki v skladu z normalizacijo bili shranjeni v posameznih tabelah, poizvedbe pa bi jih združevale z ukazom `JOIN`. S shranjevanjem povezanih podatkov v en dokument so poizvedbe hitrejše, se pa lahko podatki podvajajo, ker so denormalizirani. Kljub temu MongoDB podpira semantiko leve zunanje združitve z operatorjem `$lookup`.

Podobno kot relacijske podatkovne baze, MongoDB poleg ključa dokumenta, ki je osnovno kazalno polje, omogoča tudi ustvarjanje več tipov dodatnih kazalnih polj. Ti vključujejo sestavljena, enolična in druga kazalna

polja. Ob poizvedbah se kazalna polja uporabijo, da je dostop do podatkov najučinkovitejši.

MongoDB podpira drobljenje podatkov (angl. sharding) na več vozlišč, ki omogoča horizontalno skaliranje. Podatke je mogoče razporediti na podlagi obsega ključa, razpršitvene funkcije ali po meri uporabnika. Za usmerjanje zahtev na prava vozlišča v gruči obstajajo namenski strežniki, preko katerih gre vsaka zahteva.

Visoko razpoložljivost zagotavljajo množice replik (angl. replica set). Množica replik je sestavljena iz več vozlišč, ki vsebujejo isto množico podatkov. V množici replik je v vsakem trenutku samo eno vozlišče, ki sprejema zahteve za pisanje. Ostala vozlišča sprejemajo samo zahteve za branje. Ob izpadu glavnega vozlišča, zaradi okvare v omrežju ali samem vozlišču, izbere množica replik najprimernejšega kandidata za novo glavno vozlišče. Pri tem uporabljajo razširjeno implementacijo algoritma Raft [15] za zagotavljanje konsenza. Vozlišča lahko tako izberejo novo glavno vozlišče samo v primeru, če pri izbiranju sodeluje več kot polovica vseh vozlišč. Branje iz glavnega vozlišča je močno konsistentno, medtem ko je branje iz ostalih vozlišč zakasnjeno konsistentno.

Poglavje 6

Obremenitveni preizkusi

Obremenitvene preizkuse smo izvedli, da na praktičnem primeru obremenitve sistema z veliko količino sočasnih zahtev prikažemo prednosti in slabosti posameznih podatkovnih baz. Preizkuse smo izvedli enkrat brez in enkrat z razdelitvami v omrežju. Pri tem želimo pokazati, katera podatkovna baza se z vidika povprečnih zakasnitev, uspešnosti in pravilnosti izkaže kot najboljša.

V obremenitvene preizkuse so vključene podatkovne baze Riak, MongoDB in HBase. V poglavju 5 je sicer za lažje razumevanje ostalih podatkovnih baz tudi opis delovanja podatkovnih baz BigTable in Dynamo, ki pa nista vključena v obremenitvene preizkuse.

6.1 Vzpostavitev okolja

Obremenitvene preizkuse smo izvedli s simuliranimi zahtevami na strežnike s podatkovnimi bazami. Vsaka zahteva v simulaciji je povečala števec pod naključno izbranim ključem, ki je bil sestavljen iz dveh delov. Prvi del je bil naključen niz znakov iz večje zaloge vrednosti, drugi del pa naključen niz iz manjše zaloge vrednosti.

Pri tem smo bili posebej osredotočeni na simulacijo napak v omrežju med posameznimi strežniki. V ta namen smo uporabili orodje *iptables*, da smo gručo strežnikov razdelili v več med seboj nepovezanih delov. Na ta način

smo v sistemu ustvarili pogoje, ki niso idealni in jih lahko pričakujemo tudi v sistemih izven simulacijskega okolja.

Simulacije in obremenitvene preizkuse smo izvedli z uporabo ene izmed storitev Amazon Web Services, Amazon EC2. Za vsak tip podatkovne baze je bila postavljena gruča s petimi strežniki. Da bi omejili vpliv na porabo virov na teh strežnikih, so bili obremenitveni programi naloženi na ločene strežnike.

Tako obremenitveni strežniki kot strežniki s podatkovnimi bazami so bili postavljeni v isto omrežje VPC. Na ta način je bila zagotovljena povezljivost med strežniki v privatnem omrežju.

6.1.1 Obremenitveni strežniki

Za izvedbo konstantne obremenitve na strežnike smo uporabili orodje Vegeta [1]. Izvršljiva datoteka tega programa sprejme seznam URL-jev kot tarče za obremenitev. Nastaviti je mogoče tudi ostale parametre, kot so število niti in količine zahtev na sekundo. Na standardni izhod je nato zapisan izid izvedbe vsake posamezne zahteve v binarni obliki. Ta vsebuje podatke o času začetka zahteve, njenega trajanja in končnega HTTP odgovora strežnika. V primeru izteka časovne omejitve, je zahteva označena kot neuspešna. Časovno omejitev je mogoče prilagoditi z zastavico `-timeout`. Izhod izvajanja programa je nato mogoče obdelati in analizirati z isto izvršljivo datoteko. Izpišemo lahko statistiko o uspešnih in neuspešnih zahtevah, izvozimo neobdelane podatke v datoteko za nadaljnjo obdelavo ali kot grafikon, ki ga lahko odpremo s spletnim brskalnikom.

Orodje Vegeta smo v simulacijah naložili skupaj na strežnike z obremenitvenimi programi z namenom, da bi omejili vpliv zakasnitev omrežja med tema deloma simulacije. V seznamu strežnikov za obremenitev je tako bil samo en naslov URL, ki je kazal na obremenitveni program na istem strežniku z naslovom `localhost`.

Program, ki je dobil zahtevo orodja Vegeta, je bil prilagojen za tip podatkovne baze, ki je bila del preizkusa, in je vseboval vse potrebne knjižnice za

povezavo s to podatkovno bazo. Ta program je po prejemu zahteve generiral naključni ključ v podatkovni bazi in izvedel povečavo števca.

Za vzpostavitev obremenitvenih strežnikov je bila uporabljena slika sistema Amazon Machine Image (AMI) Ubuntu 14.04 LTS. Vsaka instanca je zaradi lažje izvedbe pripadala istemu omrežju in podomrežju kot strežniki s podatkovnimi bazami. Njegova varnostna skupina je imela odprta vrata 22 za omogočanje povezave SSH.

Nato je sledila namestitev knjižnic, potrebnih za izvedbo preizkusov. V izpisu 6.1 je naveden nabor ukazov za namestitev vseh zahtev.

Izpis 6.1: Seznam ukazov za vzpostavitev okolja obremenitvenih strežnikov

```
$ sudo apt-get update
$ sudo apt-get install python-pip
$ sudo pip install docopt
$ sudo apt-get install openjdk-7-jre-headless
$ curl -L -O https://github.com/tsenart/\
    vegeta/releases/download/v6.0.0/\
    vegeta-v6.0.0-linux-amd64.tar.gz
$ tar -xvf vegeta-v6.0.0-linux-amd64.tar.gz
$ sudo mv vegeta /usr/bin
$ sudo apt-get install python-dev
$ sudo apt-get install python-lxml
$ sudo apt-get install libffi-dev
$ sudo apt-get install libssl-dev
$ sudo pip install cryptography
$ sudo pip install gevent
$ sudo pip install flask
$ sudo pip install riak
$ sudo pip install pymongo
```

Za zagotovitev izvedbe preizkusov brez prekinitev je bila nato spremenjena meja operacijskega sistema za največje število sočasno odprtih opisni-

kov datotek (angl. file descriptor) tekočih procesov. Razlog je v dejstvu, da pri obremenitvah z večjim številom povezav včasih pride do daljših zakasnitev pri odzivu podatkovnih baz in zaradi tega povezave ostajajo odprte dalj časa. Za vsako odprto povezavo med orodjem Vegeta in programom za obremenitev se odpre opisnik datoteke, kar je značilno za sisteme podobne Unixu ("Everything is a file.").

V sistemu Ubuntu je mejo mogoče spremeniti v datoteki `/etc/security/limits.conf`, kjer je potrebno dodati vrstici, kot je prikazano v izpisu 6.2.

Izpis 6.2: Nastavitev meje največjega števila sočasno odprtih datotek

```
*       hard    nofile  65536
*       soft    nofile  65536
```

Za vključitev novih nastavitev je potrebno v datoteki `/etc/pam.d/common-session` dodati še modul `pam_limits.so`, prikazano v izpisu 6.3.

Izpis 6.3: Dodajanje modula `pam_limits.so`

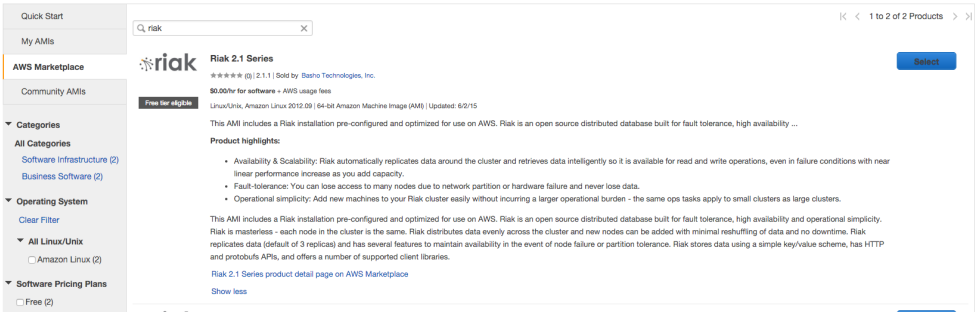
```
session required        pam_limits.so
```

Za uveljavitev sprememb je potreben ponovni zagon, nato pa je trenutno nastavitev mogoče preveriti z ukazom `ulimit -n`.

6.1.2 Riak

Za vzpostavitev gruče Riak strežnikov smo uporabili sliko sistema AMI z nameščenim okoljem Riak verzije 2.1, ki jo je mogoče dobiti na AWS Marketplace ob postavljanju nove instance v okolju EC2. Izbor slike AMI je prikazan na sliki 6.1.

Po zagonu je potrebno pravilno nastaviti nastavitve za komunikacijo med vozlišči gruče. V postavitveni datoteki *riak.conf* mora biti nastavljena ustrezna spodnja in zgornja meja številke vrat z nastavitvama prikazanima v iz-



Slika 6.1: Riak 2.1 AMI.

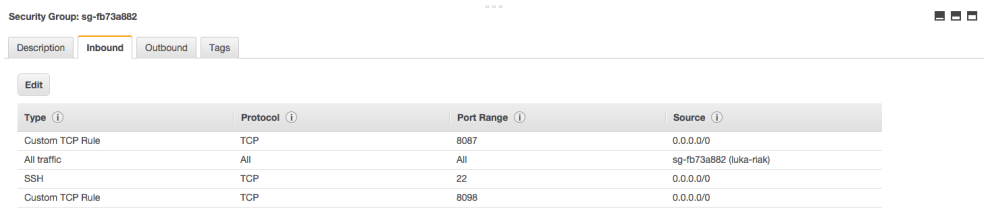
pisu 6.4.

Izpis 6.4: Nastavitev območja številke vrat za sistem Riak

```
erlang.distribution.port_range.minimum = 6000
erlang.distribution.port_range.maximum = 7999
```

V našem primeru smo ju nastavili na *6000* za spodnjo in *7999* za zgornjo mejo.

To območje številk vrat je nato za pravilno delovanje gruče potrebno omogočiti tudi v varnostni skupini EC2 instanc skupaj s številkama vrat *4369* in *8099* za notranjo komunikacijo. Med eksperimenti smo zaradi bolj preprostega upravljanja imeli vedno omogočeno komunikacijo med vsemi vozlišči, kot je prikazano na sliki 6.2.



Slika 6.2: Nastavitve varnostne skupine za gručo Riak.

Naslednji korak je bila konfiguracija povezav med vozlišči. To je mogoče

storiti z ukazom

```
$ riak-admin cluster join riak@<drugo vozlisce>.
```

Z ukazom se trenutno vozlišče poveže v gručo z nekim drugim vozliščem. Povezav ni potrebno ustvariti med vsakim posameznim parom vozlišč. Najlažje je obravnavati eno vozlišče kot osnovno in na njega povezati ostala vozlišča. Vsak tak ukaz se ne izvede takoj, ampak se doda v vmesno stopnjo za izvedbo (angl. stage). Trenutno stanje te stopnje je mogoče preveriti z ukazom

```
$ riak-admin cluster plan.
```

Ko je doseženo želeno stanje, ukaz

```
$ riak-admin cluster commit
```

izvede vse prejšnje ukaze in vzpostavi gručo.

Podatkovni tip

Za uporabo podatkovne baze je nato potrebno ustvari tip koša (angl. bucket type), ki je množica lastnosti, skupna vsem košem istega tipa. Sami koši se nato ustvarijo dinamično ob prvem dostopu.

Pri ustvarjanju tipa koša smo uporabili nastavitve lastnosti, opisane v tabeli 6.1.

Z lastnostjo `datatype` je mogoče določiti tip CRDT, ki ga uporabljajo koši s tem tipom koša. Privzeto Riak uporablja podatkovno shrambo ključ-vrednost, podprti pa so tipi CRDT `counter`, `set` in `map`.

Za faktor replikacije je bila eksperimentalno izbrana vrednost enaka številu vozlišč v gruči, saj je na ta način mogoče zagotoviti visoko razpoložljivost, kljub razdelitvam v omrežju. S tem se prepreči možnost zahtev, ko se med razdelitvijo na vozlišču ne bi nahajal ključ poizvedbe in bi nato zaradi nedosegljivosti ostalih vozlišč bila zahteva zavrnjena.

parameter	vrednost	opis
datatype	counter	podatkovni tip
r	1	št. vozlišč, ki se morajo odzvati na zahtevek za branje, da je to uspešno
w	1	št. vozlišč, ki se morajo odzvati na zahtevek za pisanje, da je to uspešno
dw	1	št. vozlišč, kjer se mora podatek zapisati na trdi disk
n_val	5	faktor replikacije

Tabela 6.1: Lastnosti tipa koša Riak, uporabljene v preizkusih

Željen tip koša je mogoče ustvariti z ukazom

```
$ riak-admin bucket-type create counters '
  {
    "props":
      {
        "datatype": "counter",
        "r": 1,
        "w": 1,
        "dw": 1,
        "n_val": 5
      }
  }'.
```

6.1.3 HBase

Za namestitev smo uporabili instance EC2 tipa *m4.xlarge*. Ta tip instance ima štiri navidezne centralne procesne enote in 16 GB prostora v pomnilniku. Prav tako smo posameznemu vozlišču dodelili 32 GB prostora na disku. Sistem ima namreč visoke zahteve za prostor v pomnilniku in na disku. Nastavitve varnostne skupine za gručno strežnikov HBase so prikazane na

Type	Protocol	Port Range	Source
Custom TCP Rule	TCP	9095	0.0.0.0/0
Custom TCP Rule	TCP	60030	0.0.0.0/0
Custom TCP Rule	TCP	8080	0.0.0.0/0
Custom TCP Rule	TCP	60020	0.0.0.0/0
All traffic	All	All	sg-c76f6ba1 (luka-cdh)
SSH	TCP	22	0.0.0.0/0
Custom TCP Rule	TCP	7180	0.0.0.0/0
Custom TCP Rule	TCP	60000	0.0.0.0/0
Custom TCP Rule	TCP	9090	0.0.0.0/0
Custom TCP Rule	TCP	8085	0.0.0.0/0
Custom TCP Rule	TCP	2181	0.0.0.0/0
Custom TCP Rule	TCP	60010	0.0.0.0/0

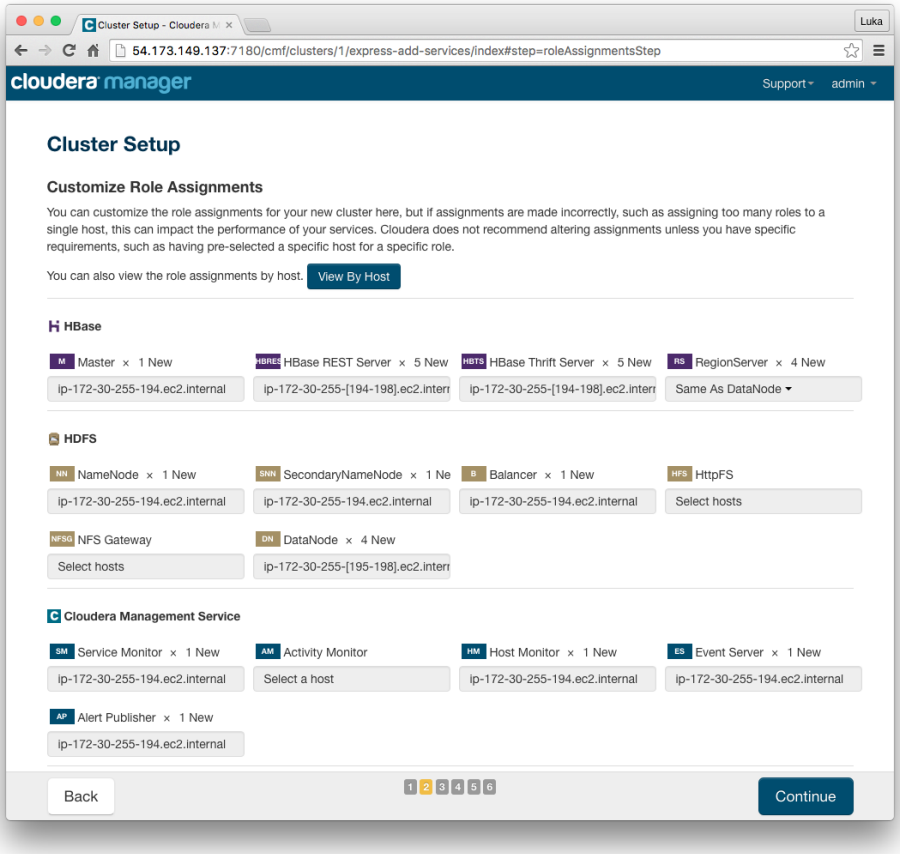
Slika 6.3: Nastavitve varnostne skupine za gručo HBase.

sliki 6.3.

Za vzpostavitev gruče HBase strežnikov smo uporabili odprtokodno distribucijo CDH podjetja Cloudera. Ta omogoča avtomatizirano namestitev gruče preko spletnega vmesnika. Na eno od vozlišč je potrebno namestiti namestitven program, ta pa zažene spletni strežnik, preko katerega je mogoče namestiti distribucijo v gručo. V čarovnik je potrebno vnesti spletne naslove ostalih strežnikov v gruči in način za dostop do njih, bodisi geslo bodisi ključ SSH.

Distribucija olajša namestitev vseh komponent, ki so potrebne za delovanje sistema HBase. To sta porazdeljen datotečni sistem HDFS in sistem za koordinacijo med vozlišči Zookeper. Tekom nastavitve je preko uporabniškega vmesnika mogoče nastaviti vloge vozlišč v gruči, kot je prikazano na sliki 6.4.

Gručo HBase v preizkusih je na petih strežnikih sestavljalo eno glavno vozlišče, štiri podatkovna vozlišča in dva vozlišča Zookeeper. Vsak strežnik je imel tudi svoje vozlišče HDFS.



Slika 6.4: Nastavitve vlog vozlišč v gruči HBase.

Po zaključeni namestitvi se je za kreiranje tabele in družine stolpcev potrebno povezati na poljuben strežnik v gruči preko protokola SSH in se postaviti v vlogo uporabnika `hdfs`, ki ga je ustvaril namestitveni program. Nato je možno zagnati lupinski vmesnik HBase:

```
$ sudo su hdfs
$ hbase shell.
```

Z ukazom `create` je mogoče ustvariti tabelo z družino stolpcev:

```
hbase(main):001:0> create 'table', 'counts'
0 row(s) in 2.4530 seconds


=> HBase::Table - table.
```

V tem primeru je tabela poimenovana *table*, družina stolpcev pa *counts*.

6.1.4 MongoDB

Za namestitev MongoDB strežnikov je bila, enako kot pri strežnikih za obremenitev, uporabljena slika sistema Ubuntu 14.04 LTS. Uporabljen tip instance za vseh pet strežnikov je bil *t2.medium*, ki ima dve navidezni centralni procesni enoti in 4 GB prostora v pomnilniku. Varnostna skupina je imela zraven vrat s številko 22 za vse TCP povezave odprta tudi vrata s številko 27017 za TCP povezave iz varnostne skupine za obremenitvene strežnike.

Sledila je namestitev MongoDB. Uporabljena različica programa je bila 3.2. Postopek namestitve je opisan v nadaljevanju.

Prvi korak je namestitev javnega GPG ključa za MongoDB. Z uporabo tega ključa orodje za upravljanje s paketi `apt` ob namestitvi preveri verodostojnost paketov. Namestiti ga je mogoče z ukazom

```
$ sudo apt-key adv --keyserver\
    hkp://keyserver.ubuntu.com:80 --recv EA312927.
```

Paketi se nahajajo v samostojnem repozitoriju, ki ga dodamo v sistem z ukazom

```
$ echo "deb http://repo.mongodb.org/apt/ubuntu\
  trusty/mongodb-org/3.2 multiverse" |
  sudo tee /etc/apt/sources.list.d/mongodb-org-3.2.list.
```

Z ukazoma

```
$ sudo apt-get update
$ sudo apt-get install -y mongodb-org
```

nato posodobimo podatke o paketih iz prej dodanega repozitorija in namestimo MongoDB.

Sledila je konfiguracija strežnikov. V datoteki `/etc/mongod.conf` je bilo potrebno popraviti nastavitev IP-naslova, na katerem posamezen strežnik `mongod` posluša za vhodni promet. Privzeta nastavitev `127.0.0.1` je bila spremenjena v omrežni naslov strežnika v omrežju VPC. V sledečih primerih je uporabljen naslov `172.30.255.1`.

V isti datoteki je treba nastaviti ime množice replik (replica set), ki se v MongoDB uporabljajo za združevanje posameznih `mongod` procesov v skupine, ki hranijo enako množico podatkov. Celotno množico podatkov je tako možno s tehniko drobljenja (angl. sharding) razporediti med več množic replik in na tak način zagotoviti večjo prepustnost sistema. Za namen preizkusov v tem delu smo uporabili samo eno množico replik za celotno množico podatkov.

Primeri nastavitev, ki jih je potrebno nastaviti v zgoraj omenjeni datoteki `/etc/mongodb.conf`, so prikazani v izpisu 6.5.

Izpis 6.5: Nastavitve v datoteki /etc/mongod.conf

```
net:
  bindIp: 172.30.255.1
replication:
  replSetName: rs0
```

Po ponovnem zagonu `mongod` procesa z ukazom

```
$ sudo service mongod restart
```

se je za dokončanje namestitve na enem izmed vozlišč potrebno povezati v ukazno lupino procesa z ukazoma

```
$ export LC_ALL=C
$ mongo --host 172.30.255.1
```

in vzpostaviti množico replik z ukazom

```
> rs.initiate()
```

ter dodati vsa ostala vozlišča z ukazom

```
> rs.add("172.30.255.2:27017").
```

Zadnji korak je kreiranje zbirke podatkov, ki bo uporabljena v obremenitvenih preizkusih z ukazom

```
> rs.createCollection("counters").
```

6.2 Rezultati

Rezultate preizkusov smo zbirali s predhodno omenjenim orodjem Vegeta, ki omogoča proženje zahtev na želene omrežne naslove in zbiranje podatkov o njih. Na voljo je podatek o času začetka posamezne zahteve, času trajanja in podatek o tem, ali je bila zahteva uspešna ali ne, skupaj z morebitno napako ob neuspešnih zahtevah.

Tekom testiranja orodje Vegeta zapisuje te podatke v datoteko v lastni binarni obliki, kasneje pa je takšno datoteko mogoče obdelati z isto izvršljivo datoteko in podatke z ukazom `vegeta dump` shraniti v CSV ali JSON obliki. Prav tako je mogoč izvoz v tip datoteke HTML. V tem primeru se generira izvorna koda Javascript, pripravljena za izris grafikona, ki se izvede ob odprtju datoteke. Vsaka od teh operacij ima možnost na vhodu sprejeti več binarnih datotek, kar je uporabno v primeru izvajanja preizkusov na večih strežnikih.

Za vsak sistem smo poskusili oceniti zgornjo mejo števila sočasnih zahtev med običajnim delovanjem omrežja pri omejenem naboru strežnikov.

Hkrati smo za preverjanje konsistentnosti posamezne podatkovne rešitve zapisovali ključe, pod katerimi so bili povišani števci ter podatek o tem, ali je bila zahteva potrjena (angl. acknowledged) ali ne. Na ta način smo po zaključku obremenitvenih preizkusov lahko preverili podatke v obremenjenih sistemih in prešteli prisotne ali manjkajoče povečave števcev. Le-te smo razporedili v štiri kategorije — *pravilno pozitivne*, *napačno pozitivne*, *pravilno negativne* in *napačno negativne*.

6.2.1 Obremenitveni preizkusi

V tabelah in grafikonih v nadaljevanju so prikazani rezultati obremenitvenih preizkusov za obravnavane podatkovne baze.

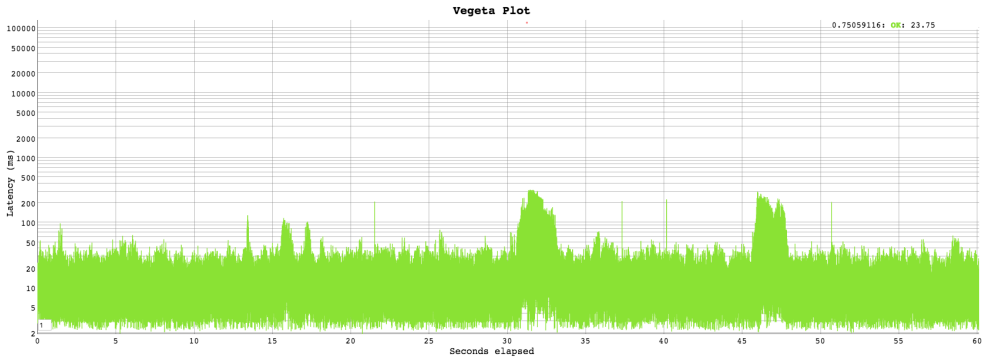
Riak

Statistika za primer obremenitvenega preizkusa gruče Riak brez razdelitev v omrežju je prikazana v tabeli 6.2. Grafikon s temi podatki je prikazan na sliki 6.5.

Rezultat preverjanja prisotnosti povečav števcev je prikazan v tabeli 6.3. Uspešna je bila večina povečav. Ena zahteva je bila neuspešna zaradi prekoračitve časovne omejitve.

čas trajanja	60 sekund
skupno število zahtev	300000
število uspešnih zahtev	299999
povprečna zakasnitev	13 ms
uspešnost	100 %

Tabela 6.2: Statistika obremenitvenega preizkusa sistema Riak brez razdelitev v omrežju



Slika 6.5: Grafikon preizkusa sistema Riak brez razdelitev v omrežju.

	Pozitivni	Negativni
Pravilno	299999	1
Napačno	0	0

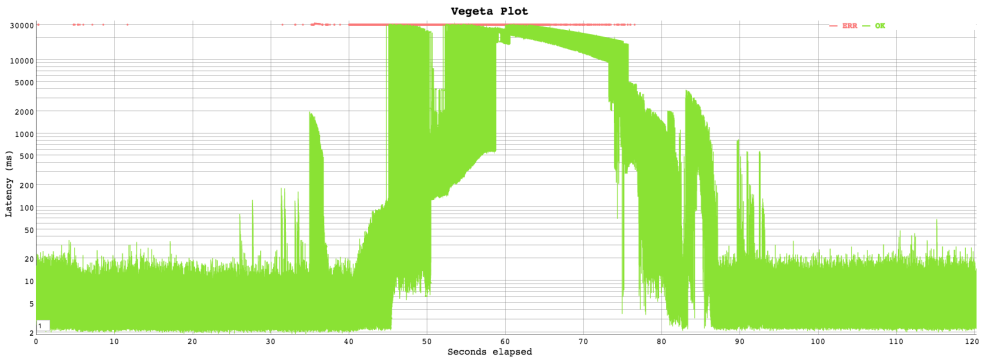
Tabela 6.3: Uspešnost povečav števcev v sistemu Riak brez razdelitev v omrežju

Statistika primera, ko smo povzročili razdelitev v omrežju med strežniki Riak, je prikazana v tabeli 6.4. Grafikon z istimi podatki je prikazan na sliki 6.6.

Rezultat preverjanja prisotnosti povečav števcev je prikazan v tabeli 6.5. Iz podatkov lahko razberemo, da je bil del zahtev neuspešen.

čas trajanja	120 sekund
skupno število zahtev	300000
število uspešnih zahtev	289295
povprečna zakasnitev	3963 ms
uspešnost	96,43 %

Tabela 6.4: Statistika obremenitvenega preizkusa sistema Riak z razdelitvami v omrežju



Slika 6.6: Grafikon preizkusa sistema Riak z razdelitvami v omrežju.

	Pozitivni	Negativni
Pravilno	298235	1765
Napačno	0	0

Tabela 6.5: Uspešnost povečav števcev v sistemu Riak z razdelitvami v omrežju

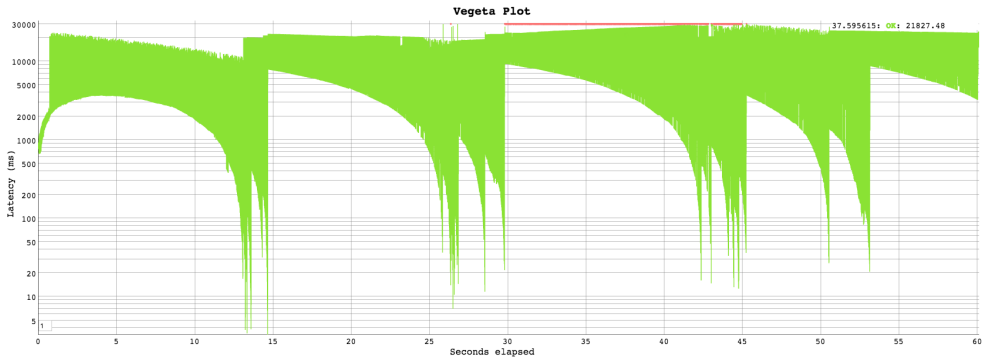
HBase

Preizkus sistema HBase brez razdelitev v omrežju je prikazan na sliki 6.7. Statistični podatki tega preizkusa se nahajajo v tabeli 6.6.

Velika večina zahtev v preizkusu je bila uspešnih. Vsem neuspešnim zahtevam je potekla časovna omejitev v sistemu Vegeta. Iz tabele 6.7 je razvidno, da so bile spremembe kljub temu uspešno zapisane.

čas trajanja	60 sekund
skupno število zahtev	225000
število uspešnih zahtev	222855
povprečna zakasnitev	13625 ms
uspešnost	99,05 %

Tabela 6.6: Statistika preizkusa sistema HBase brez razdelitev v omrežju



Slika 6.7: Grafikon preizkusa sistema HBase brez radelitev v omrežju.

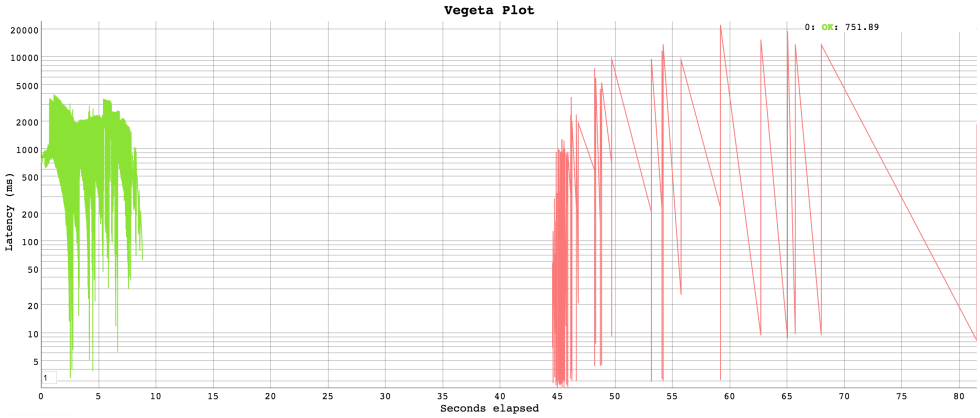
	Pozitivni	Negativni
Pravilno	225000	0
Napačno	0	0

Tabela 6.7: Uspešnost povečav števcev v sistemu HBase brez razdelitev v omrežju

Rezultati preizkusa z razdelitvami v omrežju so prikazani v tabeli 6.8. Kot je razvidno tudi iz slike 6.8, takoj po nastopu razdelitve HBase preneha potrjevati zahteve in čaka, da se spet vzpostavi povezava med strežniki.

čas trajanja	60 sekund
skupno število zahtev	83850
število uspešnih zahtev	26203
povprečna zakasnitev	4823 ms
uspešnost	31,25 %

Tabela 6.8: Statistika preizkusa sistema HBase z razdelitvami z omrežju



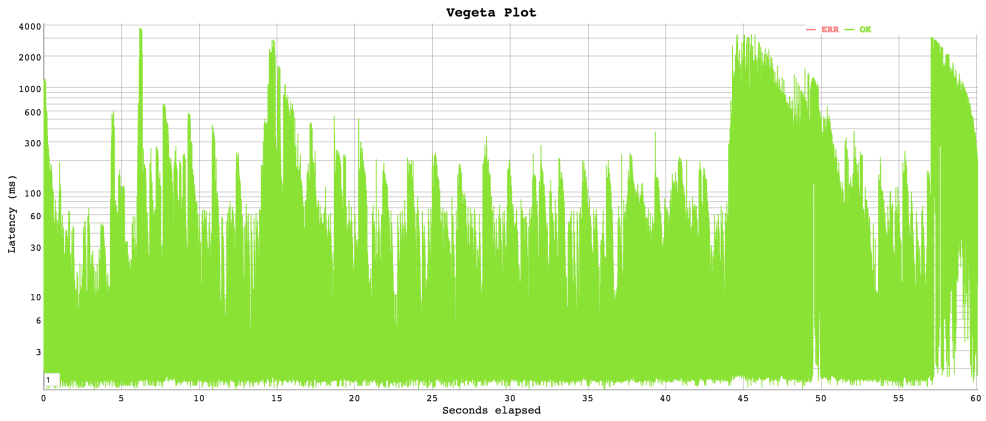
Slika 6.8: Grafikon preizkusa sistema HBase z razdelitvami v omrežju.

MongoDB

V tabeli 6.9 je prikazana statistika preizkusa sistema MongoDB brez porazdelitev v omrežju. Na sliki 6.9 je grafikon, ki prikazuje uspešnost zahtev. Iz rezultatov je razvidno, da ima sistem MongoDB visoko prepustnost zahtev, višjo tako od sistema Riak kot sistema HBase.

čas trajanja	60 sekund
skupno število zahtev	720000
število uspešnih zahtev	720000
povprečna zakasnitev	47 ms
uspešnost	100 %

Tabela 6.9: Statistika MongoDB brez razdelitev v omrežju

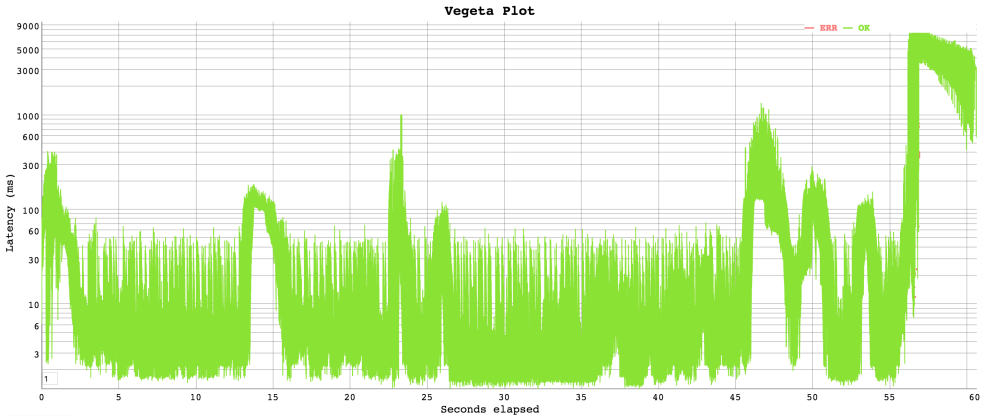


Slika 6.9: Grafikon preizkusa sistema MongoDB brez razdelitev v omrežju.

Statistika preizkusa z razdelitvami v omrežju je prikazana v tabeli 6.10. Na sliki 6.10 je prikazan grafikon z uspešnimi in neuspešnimi zahtevami. Velika večina zahtev je bila tudi v tem primeru uspešna. Ampak iz tabele 6.11 je razvidno, da po ponovni vzpostavitvi povezav manjka približno polovica potrjenih zapisov. Do tega pride, ker ob razdelitvi v omrežju v vsakem delu postane aktiven po en glavni strežnik. Ti strežniki sočasno sprejemajo in potrjujejo zahteve tudi medtem, ko so povezave prekinjene. Po ponovni vzpostavitvi povezav samo en strežnik ostane glavni, spremembe, ki so jih do takrat shranili ostali, pa so zavržene. Tako MongoDB na račun konsistentnosti omogoča visoko razpoložljivost.

čas trajanja	60 sekund
skupno število zahtev	720000
število uspešnih zahtev	719582
povprečna zakasnitev	299 ms
uspešnost	99,94 %

Tabela 6.10: Statistika MongoDB z razdelitvami v omrežju



Slika 6.10: Grafikon preizkusa sistema MongoDB z razdelitvami v omrežju.

	Pozitivni	Negativni
Pravilno	392539	0
Napačno	327058	0

Tabela 6.11: Uspešnost povečav števcev MongoDB z razdelitvami v omrežju

6.2.2 Diskusija

Obravnavane podatkovne baze se po načinu delovanja med seboj razlikujejo. Naš namen je bil primerjati posamezne pristope za problem štetja ob delovanju omrežja brez napak ter ob razdelitvah v omrežju.

Podatke obremenitvenih preizkusov brez in z razdelitvami v omrežju smo zbrali v skupnih tabelah in posamezne pristope ovrednotili glede na postavljene cilje:

- zakasnitev,
- delež uspešnih zahtev (razpoložljivost sistema),
- pravilnost podatkov po opravljenem preizkusu.

Podatki o delovanju sistemov ob običajnih pogojih v omrežju brez razdelitev so zbrani v tabeli 6.12.

Podatkovna baza	Obremenitev (zahteve na sekundo)	Povprečna zakasnitev (v ms)	Uspešnost (%)
Riak	5.000	13	100,00
HBase	3.750	13.625	99,05
MongoDB	12.000	47	100,00

Tabela 6.12: Rezultati obremenitvenih preizkusov brez razdelitev v omrežju

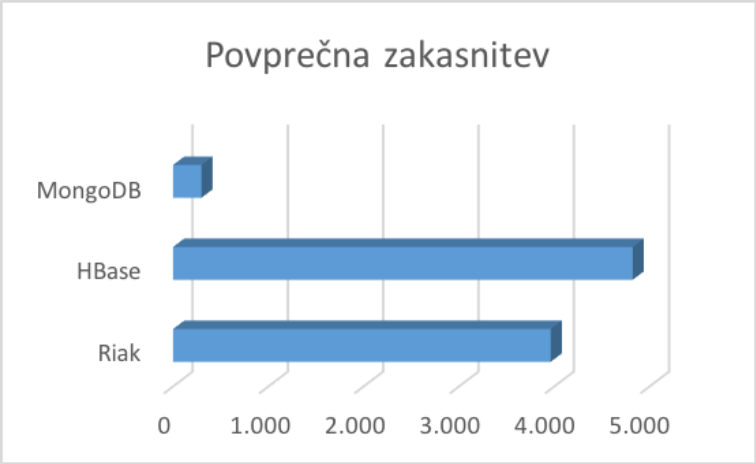
Iz podatkov je razvidno, da vsi sistemi ob delovanju omrežja brez napak delujejo brez večjih pomankljivosti. HBase ima visoko zakasnitev za odgovor na zahteve, kljub temu pa uspešno obdela vse zahteve. MongoDB se izkaže kot podatkovna baza z najvišjo propustnostjo.

Podatki o delovanju sistemov ob nastopu razdelitev v omrežju so zbrani v tabeli 6.13.

Podatkovna baza	Obremenitev (zahteve na sekundo)	Povprečna zakasnitev (v ms)	Uspešnost (%)	Pravilnost (%)
Riak	2.500	3.963	96,43	100,00
HBase	1.397,5	4.823	31,25	100,00
MongoDB	12.000	299	99,94	54,55

Tabela 6.13: Rezultati obremenitvenih preizkusov z razdelitvami v omrežju

Ob pojavu razdelitev v omrežju je pisanje v sistem Riak pravilno za vsako potrjeno zahtevo v sistem. Ob pojavu razdelitve se pojavi kratko obdobje, ko sistem ne uspe obravnavati nekaterih zahtev zaradi načina podvajanja. To se zgodi kljub temu, da je bil sistem konfiguriran tako, da se mora podatek zapisati samo v eno vozlišče, da se pisanje smatra kot uspešno. Po tem obdobju Riak kljub razdelitvi v omrežju sprejema zahteve in zapisuje podatke. MongoDB zahteve sprejema v obeh delih razdelitve, ampak po po-



Slika 6.11: Povprečne zakasnitve posameznih podatkovnih baz pri obremenitvenih preizkusih z razdelitvami v omrežju.

novni vzpostavitvi povezav izgubi vse podatke zapisane na vozlišče, ki izgubi status glavnega vozlišča.

Za boljšo preglednost sledi grafični prikaz doseganja posameznih ciljev za podatkovne baze.

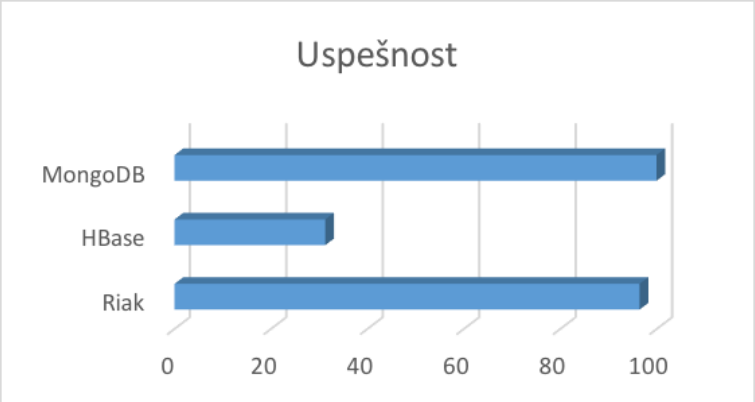
Slika 6.11 prikazuje povprečne zakasnitve sistema v milisekundah.

Slika 6.12 prikazuje delež uspešnih zahtev ob prisotnosti razdelitev v sistemu. Najvišji delež ima MongoDB, takoj za njim je Riak, HBase pa ima najnižji delež.

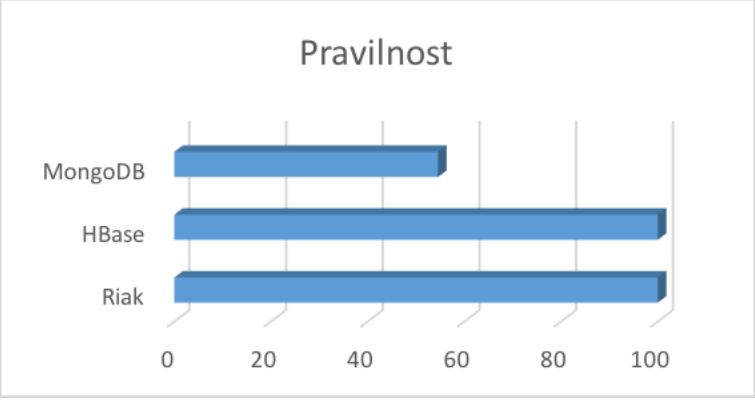
Tretji kriterij, na katerega smo se osredotočili v obremenitvenih preizkusih, je pravilnost potrjenih zahtev. To smo izmerili po ponovni vzpostavitvi povezav v omrežju. Rezultati so prikazani na sliki 6.13. Riak in HBase sta potrjene zahteve uspešno zapisala, medtem ko je MongoDB del podatkov kljub potrjenim zahtevam izgubil.

Iz opravljenih preizkusov je mogoče skleniti sledeče:

- Riak z ustrezno nastavitvijo replikacije ohrani potrjene spremembe tudi, ko se zgodijo razdelitve v omrežju, kar je lastnost tipov CRDT,
- HBase zaradi ohranjanja konsistentnosti čaka z obdelovanjem zahtev



Slika 6.12: Uspešnosti posameznih podatkovnih baz pri obremenitvenih preizkusih z razdelitvami v omrežju.



Slika 6.13: Pravilnosti posameznih podatkovnih baz pri obremenitvenih preizkusih z razdelitvami v omrežju.

na ponovno vzpostavitev povezav med vozlišči,

- MongoDB kljub razdelitvam v omrežju nadaljuje z obdelovanjem zahtev, ampak del potrjenih sprememb v sistemu zavrže po ponovni vzpostavitvi povezav, kljub temu da so bile potrjene kot uspešne.

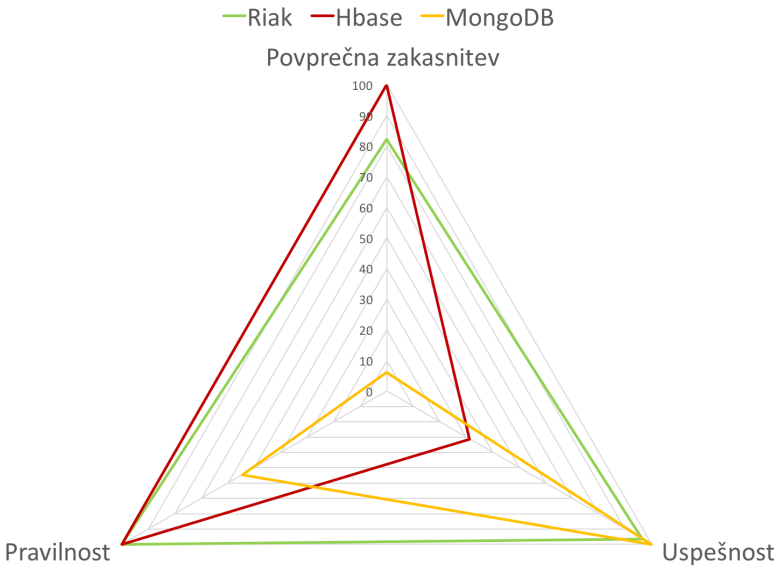
Poglavje 7

Sklepne ugotovitve

V diplomskem delu je raziskano področje pristopov k štetju dogodkov v porazdeljenih sistemih z velikim obsegom zahtev. V primerjavo so vključene podatkovna baza Riak, ki za hranjenje števcev ponuja podatkovni tip CRDT, ter podatkovni bazi HBase in MongoDB, ki se pogosto uporabljata v praksi.

Sisteme smo ovrednotili z analizo delovanja ob običajnih pogojih v omrežju ter ob prisotnosti prekinitev v omrežju. V preizkusih smo s simulacijo na vsak sistem sprožili večjo količino sočasnih zahtev, enkrat brez razdelitev v omrežju, drugič so razdelitve bile prisotne. Z razdelitvami smo želeli ustvariti razmere, ki niso idealne in se pojavijo tudi v praksi, sistemi pa se na njihovo prisotnost odzovejo različno. V obeh primerih smo za vse sisteme izmerili povprečno zakasnitev, delež uspešnih zahtev ter delež pravilnih zapisov, glede na uspešne (potrjene) zahteve.

Ob običajnih pogojih v omrežju se sistemi razlikujejo zgolj v številu zahtev, ki jih sistem lahko obdela na enoto časa (propustnost). Drugače je ob uvedbi prekinitev v omrežju. Takrat se sistemi razlikujejo tudi v deležu potrjenih zahtev in deležu pravilno zapisanih podatkov. Na sliki 7.1 je na podlagi merjenih količin prikazana primerjava med sistemi. Ob predpostavki, da je največja povprečna izmerjena zakasnitev 4.823 ms (prikazano v tabeli 6.13) enaka 100 %, pretvorimo vrednosti povprečne zakasnitve na interval $[0, 100]$. Skala pri tem atributu je obrnjena, kar pomeni, da največja vrednost pred-



Slika 7.1: Primerjava povprečnih zakasnitev, uspešnosti in pravilnosti posameznih podatkovnih baz pri obremenitvenih preizkusih z razdelitvami v omrežju.

stavlja največjo povprečno zakasnitev oziroma najnižjo propustnost.

Sistem Riak je v preizkusih z razdelitvami v omrežju imel najboljše rezultate. V primerjavi z MongoDB je povprečna zakasnitev sicer visoka, ampak sistem uspešno obdela večino zahtev, ki tudi ostanejo trajno dostopne. HBase v primeru razdelitev v skladu s svojimi zagotovili preneha obdelovati zahteve, medtem ko MongoDB obdeluje zahteve, ampak nekatere spremembe lahko ob ponovni vzpostavitvi povezav v omrežju zavrže.

Literatura

[1] Vegeta. [Online]. Dosegljivo na: `https://github.com/tsenart/vegeta`. [Dostopano 30. 7. 2016].

[2] Towards Robust Distributed Systems. [Online]. Dosegljivo na: `https://people.eecs.berkeley.edu/~brewer/cs262b-2004/PODC-keynote.pdf`. [Dostopano 30. 7. 2016].

[3] S. Gilbert, N. Lynch. "Brewer's Conjecture and the Feasability of Consistent, Available, Partition-Tolerant Web Services", ACM SIGACT News, št. 33, zv. 2, str. 51–59, 2002.

[4] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, W. Vogels. "Dynamo: Amazon's highly available key-value store", ACM SIGOPS Operating Systems Review, št. 41, zv. 6, str. 205–220, 2007.

[5] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, R. E. Gruber. "Bigtable: A Distributed Storage System for Structured Data", ACM Transactions on Computer Systems (TOCS), št. 26 zv. 2, članek št. 4, 2008.

[6] S. Ghemawat, H. Gobioff, S-T. Leung. "The Google file system", ACM SIGOPS Operating Systems Review, št. 37, zv. 5, str. 29–43, 2003.

[7] M. Burrows. "The Chubby lock service for loosely-coupled distributed systems", v zborniku OSDI'06: Seventh Symposium on Operating System Design and Implementation, str. 335–350, Seattle, 2006.

[8] L. Lamport. "The part-time parliament", ACM Transactions on Computer Systems (TOCS), št. 16, zv. 2, str. 133–169, 1998.

[9] An In-Depth Look at the HBase Architecture. [Online]. Dosegljivo na: `https://www.mapr.com/blog/in-depth-look-hbase-architecture`. [Dostopano 30. 7. 2016].

[10] HDFS Architecture Guide. [Online]. Dosegljivo na: `https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html`. [Dostopano 30. 7. 2016].

[11] M. Shapiro, N. Preguiça, C. Baquero, M. Zawirski. "A comprehensive study of Convergent and Commutative Replicated Data Types", INRIA, Pariz, 2011.

[12] M. Herlihy, N. Shavit. "The Art of Multiprocessor Programming", Morgan Kaufmann, 2008, pogl. 3.

[13] Fallacies of Distributed Computing Explained. [Online]. Dosegljivo na: `https://pages.cs.wisc.edu/~zuyu/files/fallacies.pdf`. [Dostopano 30. 7. 2016].

[14] MongoDB Architecture Guide. [Online]. Dosegljivo na: `https://www.mongodb.com/collateral/mongodb-architecture-guide`. [Dostopano 30. 7. 2016].

[15] D. Ongaro, J. Ousterhout. "In search of an understandable consensus algorithm", v zborniku USENIX ATC'14 Proceedings of the 2014 USENIX conference on USENIX Annual Technical Conference, str. 305–320, Philadelphia, 2014.

[16] Distributed systems for fun and profit. [Online]. Dosegljivo na: `http://book.mixu.net/distsys/single-page.html`. [Dostopano 31. 7. 2016].

[17] Developing with Riak KV. [Online]. Dosegljivo na: `https://docs.basho.com/riak/kv/2.1.1/developing/data-types/`. [Dostopano 31. 7. 2016].

[18] A. B. Bondi. "Characteristics of scalability and their impact on performance", v zborniku WOSP '00 Proceedings of the 2nd international workshop on Software and performance, str. 195–203, Ottawa, 2000.

[19] T. Haerder, A. Reuter, "Principles of transaction-oriented database recovery", ACM Computing Surverys, št. 15, zv. 4, str. 287–317, 1983.

[20] Apache HBase. [Online]. Dosegljivo na: `https://hbase.apache.org/`. [Dostopano 31. 7. 2016].

[21] Riak KV. [Online]. Dosegljivo na: `http://basho.com/products/riak-kv/`. [Dostopano 31. 7. 2016].

[22] MongoDB. [Online]. Dosegljivo na: `https://www.mongodb.com/`. [Dostopano 31. 7. 2016].

[23] Apache Cassandra. [Online]. Dosegljivo na: `http://cassandra.apache.org/`. [Dostopano 31. 7. 2016].

[24] Project Voldemort. [Online]. Dosegljivo na: `http://www.project-voldemort.com/voldemort/`. [Dostopano 31. 7. 2016].

[25] P. Hunt, M. Konar, F. P. Junqueira, B. Reed, "ZooKeeper: wait-free coordination for internet-scale systems", v zborniku USENIXATC'10 Proceedings of the 2010 USENIX conference on USENIX annual technical conference, Boston, Massachusetts, 2010, str. 11–11.

[26] A. S. Tanenbaum, M. Van Steen. "Distributed Systems: Principles and Paradigms". Pearson, 2. izdaja, pogl. 8, 2006.

[27] Redis bitmaps — fast, easy, realtime metrics. [Online]. Dosegljivo na: `http://blog.getspool.com/2011/11/29/`

fast-easy-realtime-metrics-using-redis-bitmaps/. [Dostopano 31. 7. 2016]

[28] Big Data Counting: How To Count A Billion Distinct Objects Using Only 1.5KB Of Memory. [Online]. Dosegljivo na: http://highscalability.com/blog/2012/4/5/big-data-counting-how-to-count-a-billion-distinct-objects-us.html. [Dostopano 31. 7. 2016]

[29] Relational Databases Are Not Designed For Scale. [Online]. Dosegljivo na: http://www.marklogic.com/blog/relational-databases-scale/. [Dostopano 31. 7. 2016]