UNIVERZITET U BEOGRADU
MATEMATIČKI FAKULTET

Luka B. Đorović

ANALIZA SLUČAJEVA UPOTREBE RELACIONIH I KOLONSKI ORIJENTISANIH NERELACIONIH BAZA PODATAKA

master rad

Beograd, 2024.

Mentor:

dr Saša Malkov, vanredni profesor
Univerzitet u Beogradu, Matematički fakultet

Članovi komisije:

dr Nenad Mitić, redovni profesor
Univerzitet u Beogradu, Matematički fakultet

dr Ivana Tanasijević, docent
Univerzitet u Beogradu, Matematički fakultet

Datum odbrane: 15. januar 2016.

Ovaj rad posvećujem...

Naslov master rada: Analiza slučajeva upotrebe relacionih i kolonski orijentisanih nerelacionih baza podataka

Rezime:

Ključne reči: analiza, geometrija, algebra, logika, računarstvo, astronomija

Sadržaj

Glava 1

Uvod

Podaci su najstabilniji deo svakog sistema. Oni su reprezentacija činjenica i instrukcija u formalizovanom stanju spremnom za dalju interakciju, interpretaciju ili obradu od strane korisnika ili mašine. Iako kroz svoju istoriju računarstvo važi za oblast koja uvodi nove tehnologije i alate neverovatnom brzinom, to nije slučaj za svaku njenu granu. Postoje oblasti koje se kroz istoriju nisu menjale, ili su se slabo menjale i proširivale. Primera za to ima puno i oni su uglavnom usko vezani za funkcionalne principe koji se prožimaju kroz računarske mreže, kompilatore, operativne sisteme, sisteme za upravljanje podacima itd.

Kada je reč o istoriji sistema za upravljanje podacima, mogu se izdvojiti tri faze: period pre relacionih sistema, vreme neprikosnovene vladavine relacionih sistema i nastanak alternativa relacionim sistemima pod grupnim nazivom *NoSQL*.

Do nastanka relacionih sistema za upravljanje podacima, rukovanje podacima izvodilo se kroz pisanje i čitanje iz datoteka operativnog sistema. Rukovanje većim količinama podataka nije bilo standardizovano ni na koji način, već su se konvencije uvodile na nivou organizacija. Apolo sletanje na Mesec realizovano je koristeći ovakav vid rada sa podacima, što ovaj poduhvat čini utoliko neverovatnim [4].

S obzirom da je ovaj vid rada sa podacima imao mnogobrojne mane, među kojima je jedna od glavnih bila komplikovan pristup podacima, javile su se potrebe za unapređenjem. Najuspešniji je bio Edgar F. Codd [1] koji je 1970. godine objavio rad pod imenom „*A Relational Model of Data for Large Shared Data Banks*" kao rezultat sopstvenih istraživanja i teorija o organizaciji podataka. Kao dokaz da je njegov model moguće implementirati pokrenut je System R [2], čiji je rezultat bio

[1] Edgar Frank „Ted" Codd (19 Avugst 1923 – 18 April 2003) Američki računarski naučnik

[2] System R - sistem za rad sa podacima napravljen kao deo istraživačkog projekta IBM-a

i pojava SQL-a (*Structured Query Language*) kao standardizovanog jezika za rad sa podacima. Nakon toga pojavili su se Oracle i IBM sa svojim komercijalnim proizvodima za upravljanje relacionih baza podataka. Naredni period obeležio je rad sa podacima koristeći relacioni model.

Ubrzanu digitalizacija, povećana dostupnost interneta, donela je sa sobom potrebu za obradom veće količine podataka. Sve ovo je pokazalo pojedine slabosti dosadašnjih sistema zasnovanih na relacionim modelima, koji nisu mogli u svim segmentima da odgovore na zahteve modernog doba. Ovi problemi obično poznati pod imenom: problemi velikih podataka (engl *BigData problems*), doveli su do pojave niza novih modela i principa za čuvanje podataka, kojima je dodeljen grupni naziv: nerelacione baze podataka (engl. *NoSQL*). Sistematizovanje ogromne količine fizičkog prostora na disku na kojem se podaci mogu čuvati i kasnije koristiti, kao i fleksibilnost strukture podataka sa kojima se radi, glavni su problemi tog vremena na koje su se fokusirale tehnologije nastale u *NoSQL* pokretu. Decenije vladavine relacionih sistema za čuvanje podataka ostavile su dubok trag u praksama rada sa podacima, i sa razlogom predstavljaju standard i dan danas, te je eventualno usvajanje tehnologija nastalih u ovoj fazi i danas česta dilema mnogih stručnjaka.

Kao važna grupa nerelacionih baza podataka izdvajaju se kolonski-orijentisane baze podataka. One su uvele tada nekonvencionalne koncepte čuvanja podataka po kolonama. To podrazumeva sekvencialno skladištenje vrednosti jedne kolone na disku, sa referencom na red kojem pripadaju. To sa sobom vuče razne mogućnosti za optimizaciju ali i nove pristupe modelovanja i organizacije podataka. Ovakav način skladištenja ispitavan je još davnih sedamdesetih godina XX veka, međutim u ranim godinama XXI veka došlo je do obnove interesovanja u akademskim ali i industrijskim krugovima.

Nijedan od navedenih koncepata nije univerzalno rešenje, zato je bitno postojanje sadržaja koji se bave analizom slučajeva upotrebe tih tehnologija. Pored teorijske analize koja se može pronaći u relevantnim javnim dokumentacijama korisno je imati i konkretne implementacije testova čiji se rezultati mogu iskoristiti kako bi se povukle paralele u skladu sa potrebama realnih sistema.

Cilj ovog rada je analiza i upoređivanje slučajeva upotrebe relacionih i kolonski orijentisanih baza podataka. Rad će se sastojati iz teorijskog opisa navedenih tehnologija kao i opisa konkretnih predstavnika baza podataka koji će biti korišćeni. Na osnovu teorijskih izbora i istraživanja biće analizirani različiti slučajevi

upotrebe.

Glava 2

Modeli podataka

2.1 Relacioni model

Opšte karakteristike

Relacioni model je najpopularniji model za rad sa podacima. On podatke kao i veze izmedju njih predstavlja kroz skup relacija koje predstaljaju skupove torki. Da bi jedan skup torki ili vrsta bila validna relacija u relacionom modelu, ona mora ispunjavati sledeće uslove:

- Presek kolone i vrste jedinstveno određuje vrednosnu ćeliju.
- Sve vrednosne ćelije jedne kolone pripadaju nekom zajedničkom skupu.
- Svaka kolona ima jedinstveno ime.
- Ne postoje dve identične vrste jedne tabele.

Iako ovakva formalizacija relacije jeste intuitivna (usled istorijskog uticaja koji je relacioni model ostavio na ideju organizacije podataka) ona je neophodna za definisanje složenijih pojmova.

Koncept ključa relacionog modela

Skup kolona relacije za koji važi da dva reda te relacije nemaju identične vrednosti za svaku kolonu iz tog skupa naziva se *natključ* relacije. Svaki minimalan natključ naziva se *ključ kandidat*. Svaka relacija može imati više ključeva kandidata, a jedan on njih se bira za *primarni ključ* koji mora imati definisanu vrednost

za svaku njegovu kolonu. *Strani ključ* je kolona ili skup kolona čije vrednosti predstavljaju referencu na određeni red neke druge relacije. On uzima vrednost primarnog ključa torke na koju pokazuje.

Primarni i strani ključ igraju veliku ulogu u očuvanju integriteta baze podataka o čemu će biti reči u nastavku.

Integritet relacionog modela

Integritet relacionog modela predstavlja uslove koje podaci treba da zadovolje kako bi stanje u bazi ostalo konzistentno [8] . On se drugačije naziva i „unutrašnja konzistentost" s obzirom da predstavlja aspekte koji mogu da se provere bez konsultovanja domena (npr. ne može se proveriti da li je ime studenta u tabeli ispravno bez konsultovanja domena, ali može se garantovati da će neophodni podaci biti prisutni, uzimati vrednosti iz predviđenog skupa vrednosti i sl.) . Provera integriteta se izvršava implicitno ili eksplicitno prilikom svakog ažuriranja baze podataka. Postoji više vrsta integriteta u relacionom modelu: *integritet entiteta*, *integritet domena*, *integritet neposojeće vrednosti* i *referencijalni integritet*.

Integritet entiteta kaže da svaka torka mora imati definisan primarni ključ bez nedostajućih vrednosti.

Integritet domena predstavlja uslov da za svaku kolonu postoji unapred poznati skup vrednosti koje ona može uzimati.

Integritet nepostojeće vrednosti dodeljuje se kolonama koje ne smeju uzimati nedostajuću vrednost.

Referencijalni integritet nalaže da za svaki strani ključ relacije mora postojati torka u tabeli na koju pokazuje čiji se primarni ključ poklapa sa njim.

PostgreSQL

PostgreSQL je objektno-relacioni sistem za upravljanje bazama podataka koji je nastao, a kasnije i bio razvijan na Berkliju, Univerzitet Kalifornija. PostgreSQL je otvorenog koda sa velikom SQL podrškom kao i modernim funkcionalnostima poput: kompleksnih upita, okidača, izmenjivih pogleda, transakcionog integriteta i mnogih drugih. Postgres nudi širok spektar proširenja od strane korisnika poput dodavanja novih tipova podataka, funkcija, operatora, agregatnih funkcija itd.

Kao takav, PostgreSQL je pogodan sistem za čuvanje najkompleksnijih podataka i veza između njih. Mogućnost kreiranja procedura na samoj bazi u integrisanoj

SQL sintaksi, daje široke mogućnosti optimizacije aplikacija.

PostgresSQL koristi server-klijent model funkcionisanja. Sastoji se iz serverskog i klijentskog dela procesa. Serverski deo rukuje fajlovima baze podataka, prihvata konekcije, izvršava konkretne operacije nad bazom. Klijentski deo predstavlja aplikaciju kojom korisnik može da komunicira i rukuje podacima na serverskom delu. Klijent i server komunkciraju preko TCP/IP protokola. Serverski deo može raditi sa više konekcija istovremeno tako što svaka klijentska konekcija radi kao zaseban proces.

Postgres iza sebe ima razvijenu društvenu zajednicu, pa samim tim ima dosta izvora i dokumentacije koje mogu olakšati učenje ovog sistema [7].

2.2 Kolonski-orijentisani model

Opšte karakteristike

Susret sa Big Data problemima doveo do potrebe za tabelama koje imaju ogroman broj kolona, i ogroman broj redova u okviru tih tabela. Novonastali zahtevi ukazali su na problem kod postojećih relacionih modela. Svaki upit nad tabelom podrazumevao je dohvatanje svih kolona jednog reda, gde bi se filtriranje nepotrebnih kolona izvršavalo nakon što su se sve kolone učitale u memoriju. Ovo je bila samo jedna od motivacija za implemetanciju sistema zasnovanih na kolonski orijentisanom modelu koji je dizajniran tako da ovakav problem izbegne i uz to donese i druga poboljšanja o kojima će biti reči u nastavku.

Kolonski orijentisan model podatke na disku skladišti po kolonama, a ne po redovima kao što je to slučaj kod relacionh modela, slika 2.1. Sve vrednosti kolone svih redova skladište se jedna do druge, a na konkretnu vrednosnu ćeliju referiše se pomoću ključa konkretnog reda kao i kolone čiju vrednost želimo da pročitamo. Ovakav dizajn doveo je do toga da za dohvatanje određenog skupa kolona nema potrebe da čitamo sve vrednosti tog reda, već je dovoljno da znamo konkretan ključ tog reda kao i imena kolona čije vrednosti želimo da pročitamo i tako izbegnemo višak operacija čitanja sa diska.

Ovakav vid skladištenja podataka sa sobom nosi veliki potencijal za primenu raznih algoritama za kompresiju podataka. Kompresija nad sličnim podacima koji se nalaze na uzastopnim adresama u memoriji, omogućava izbegavanje čuvanja složenih meta informacija u okviru struktura koje se koriste za tu kompresiju, što

ovaj model čini posebno pogodnim za njihovu primenu.

Kolonski orijentisan model kao i većina ostalih nerelacionih modela, nudi fleksibilnost strukture podataka koja se ogleda u neograničenom broju kolona, što daje dosta prostora za eksperimentisanje sa dizajnom baze podataka. Primer toga biće prikazan u okviru analize OLAP slučaja upotrebe.

ID	IME	STATUS	STAROST	
1	Luka	Aktivan	25	
2	Marko	Neaktivan	29	...
3	Jovan	Neaktivan	21	
4	Viktor	Aktivan	42	

ID	{1,Luka}	{2, Marko}	{3, Jovan}	{4, Viktor}
STATUS	{1, Aktivan}	{2, Neaktivan}	{3, Neaktivan}	{4, Aktivan}
STAROST	{1, 25}	{2, 29}	{3, 21}	{4, 42}

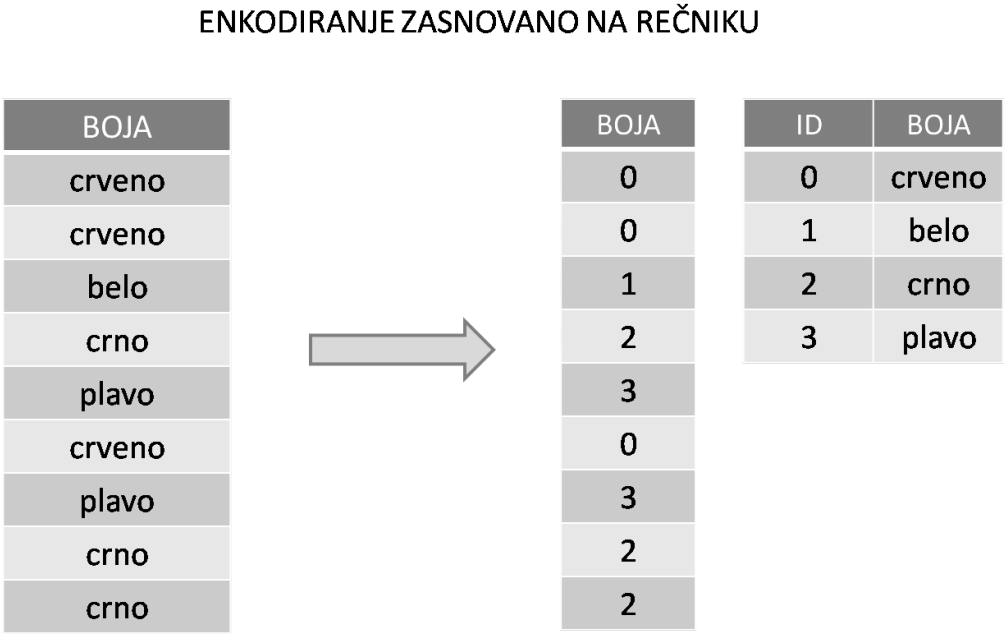
Slika 2.1: Kolonski orijentisan format

Popularni algoritmi kompresije kolonski orijentisanog modela

Neke od najpoznatijih algoritama kompresije koje kolonski orijentisan model koristi i koji će biti opisani u nastavku jesu: enkodiranje zasnovano na rečniku, enkodiranje po broju ponavljanja i delta enkoding. [1]

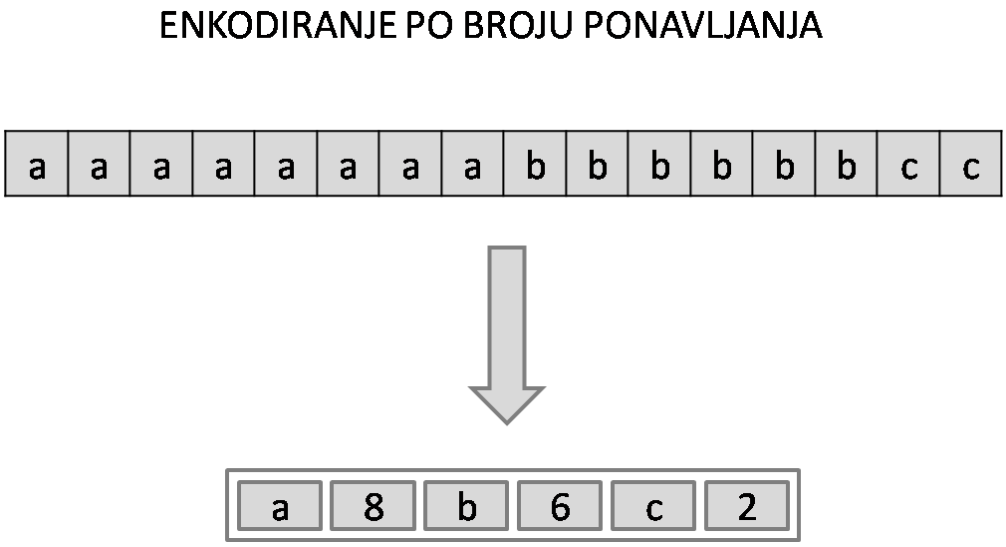
[1]Važno je napomenuti da relacioni modeli imaju svoje mehanizme kompresije podataka koji u ovom radu nisu analizirani.

Enkodiranje zasnovano na rečniku (engl. *Dictionary based encoding*), slika 2.2., funkcioniše tako što se napravi mapa vrednosti koja sadrži svaku vrednost kolone koja je prisutna među podacima. Kao vrednost kolone tada se ne upisuje konkretna vrednost, već ključ iz rečnika koji je mapiran na tu vrednost. Veličina ključa je srazmerna veličini mape, te je ovaj vid kompresije najpogodniji za kolone koje imaju mali broj vrednosti koje se ponavljaju.



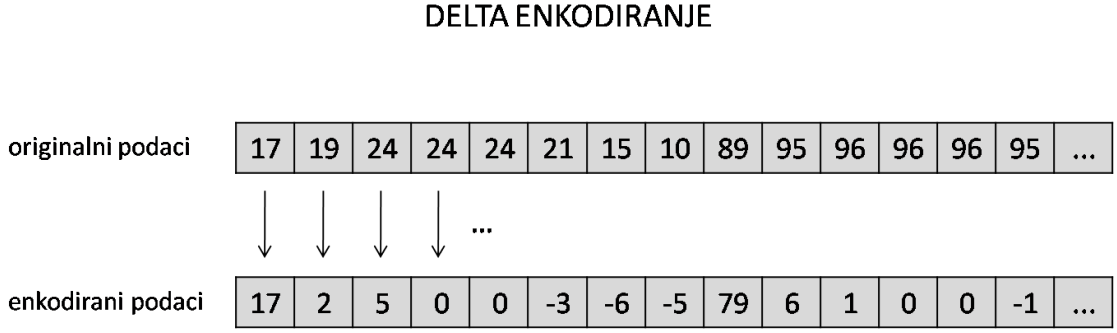
Slika 2.2: Enkodiranje zasnovano na rečniku

Enkodiranje po broju ponavljanja (engl. *Run Length Encoding*) , slika 2.3., funkcioniše tako što se uz svaku vrednost koja se ponavlja čuva i broj ponavljanja te vrednosti. Na taj način se izbegava pojava duplikata na uzastopnim adresama u memoriji. Ovaj vid kompresije najpogodniji je za kolone koje su sortirane i imaju ponavljajuće vrednosti.



Slika 2.3: Enkodiranje po broju ponavljanja

Delta enkodiranje algoritam kompresije, slika 2.4. funkcioniše tako što se u kolonama ne čuvaju same vrednosti već razlike između uzastopnih vrednosti. Očigledan primer primene ove kompresije je datumska kolona koja za vrednosti uzima uzastupne datume. U tom slučaju je dovoljno da izaberemo neki referentni datum i da za ostale kolone čuvamo razliku u odnosu na taj datum.



Slika 2.4: Delta enkodiranje

HBase

HBase je kolonski orijentisana nerelaciona baza podataka nastala 2007. godine kao prototip *BigTable* baze koja je modelovana u okviru Google-ovog članka 2006 [5].

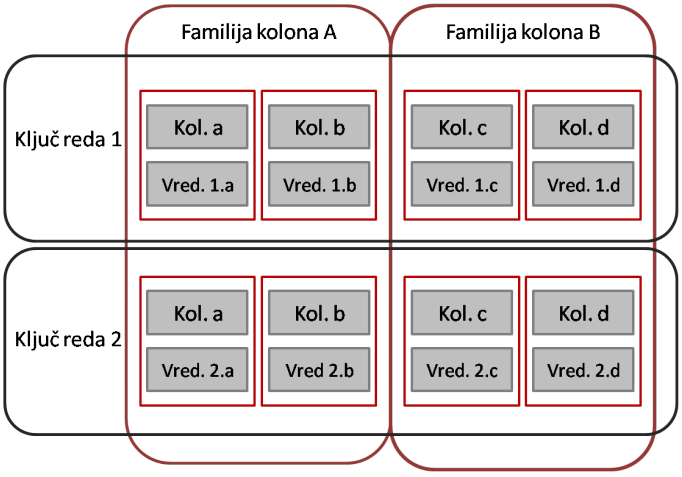
Model podataka koji HBase koristi podrazumeva da se svaka tabela sastoji iz familije kolona, a da svaka familija kolona sadrži određeni skup kolona. S obzirom da će se kolone koje pripadaju jednoj familiji sladištiti blizu na disku, cilj je da atributi, odnosno kolone koje su po prirodi slične pripadaju istoj familiji kolona, kako bi se nad njima mogli primeniti algoritmi kompresije. Hbase nudi fleksibilnost strukture podataka, što znači da da bismo neki podatak skladištili ne moramo unapred da definišemo skup kolona koji pripada nekoj tabeli , ali moramo definisati skup familija kolona te tabele.

HBase ne podržava indekse, ali su ključevi svih redova sortirani rastuće, tako da se koristi binarna pretraga za pretragu vrednosti po ključu reda kojem pripada. Ovakvo ponašanje daje na važnosti dizajniranju ključa kako bi svako čitanje podataka išlo preko ključa ili njegovog prefiksa.

Vrednosnu ćeliju u HBase tabeli određuje ključ reda, ime kolone te vrednosne ćelije, kao i familija kojoj kolona pripada, slika 2.5.

HBase nije ACID baza podataka, ali nudi sledeće garancije [6]:

- Atomičnost pri radu sa jednim redom tabele koja se ogleda što će svaka svaka izmena reda u potpunosti uspeti, ili u potpunosti propasti.
- Svako čitanje reda iz tabele vratiće stanje reda koje je bilo aktuelno najranije u trenutku kada je čitanje započeto.



Slika 2.5: Hbase model

Arhitektura HBase klastera sastoji se iz dve glavne komponente: master server i region server.

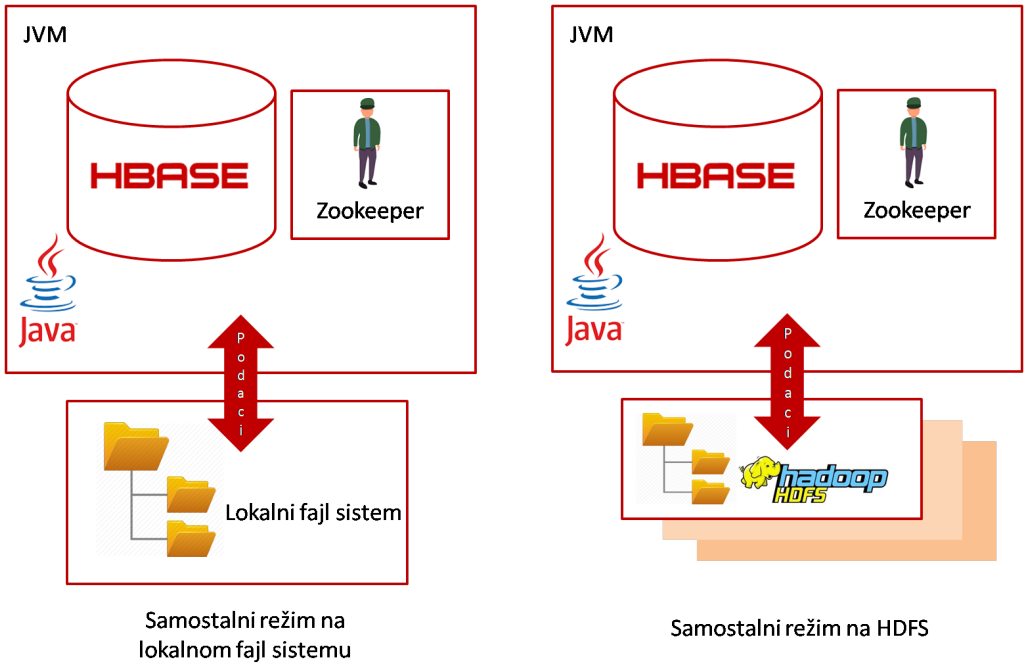
Master server sadrži servise koji rade sa metainformacijama o tabelama i familijama kolona. Uloga Master servera jeste da zahtev za podacima za konkretan ključ reda može da delegira na odgovarajući region server. Master server u te svrhe koristi *Zookeeper* [5] servis. Po potrebi master server takođe radi balansiranje opterećenja klastera.

Region server implementira servise koji direktno rade sa podacima. On organizuje regione (jedan region čine redovi u nekom rasponu vrednosti ključeva), čita i upisuje u HFile fajlove (HFile je fajl koji je rezultat kolonski orijentisanog formata, a koji se skladišti na disku). Svaka izmena koja treba da se izvrši na disku prvo se upisuje u WAL (engl *Write ahead log*), a nova izmenjena vrednost se upisuje u MemStore fajl. Kada se MemStore fajl upuni tek tada se izmene reflektuju u jedan HFile koji dalje ide na HDFS ili lokalni fajl sistem u zavisnosti od režima rada.

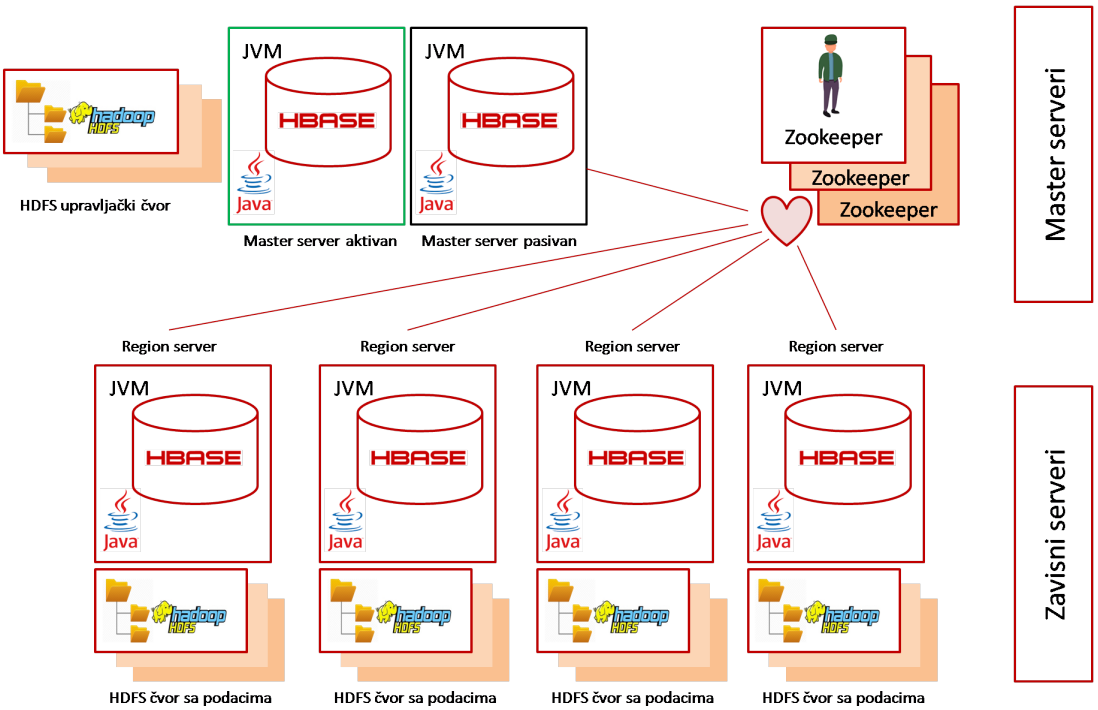
HBase dozvoljava dva režima: samostalan i distribuirani režim.

U samostalnom režimu HFile fajlove može čuvati na lokalnom fajl sistemu i korišćenje HDFS-a je opciono, slika 2.6.

Sa druge strane distribuiranom režimu neophodno je korišćenje HDFS-a za skladištenje, slika 2.7.



Slika 2.6: HBase standalone



Slika 2.7: HBase distributed

2.3 Glavne razlike između relacionog i kolonski-orijentisanog modela

Normalizacija i denormalizacija

U relacionim modelima često se radi na izbegavanju *redudantnosti* u podacima. Redudantni podaci zauzimaju višak prostora na disku i otežavaju kasnije održavanje sistema. Kako bi se izbegla redudantnost postoje postupci koji nam pomažu da organizujemo podatke tako da redudantnost umanjimo. Proces izmene logičkog modela baze podataka u cilju rešavanja problea redundantosti podataka naziva se normalizacija. U zavisnosti od toga koja pravila zadovoljava određena relacija, dodeljuje joj se odgovarajuća normalna forma. Neke od normalnih formi relacionog modela su: 1. normalna forma, 2. normalna forma, 3. normalna froma, normalna forma elementarnog ključa, Bojs-Kodova normalna forma, 4. normalna forma, normalna forma esencijalnih torki, normalna forma bez redundansi, normalna forma superključeva, 5. normalna forma, normalna forma domena i ključa.

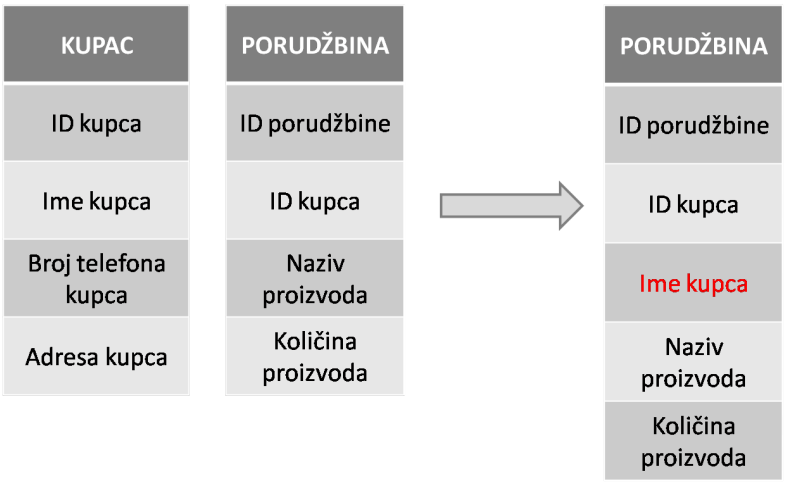
Normalizovani modeli obično raspolažu velikim brojem stranih ključeva što dovodi do povećanja broja tabela kojima se pristupa, a time i povećanja broja

operacija čitanja sa diska što može uticati na performanse čitanja.

Denormalizacija je strategija koja se koristi kod modela kod kojih je neophodno ubrzati operacije čitanja podataka, odnosno umanjiti broj tabela kojima je neophodno pristupiti kako bi se neki skup podataka pročitao iz baze podataka.

Neke od tehnika denormalizacije su: spajanje kolone, horizontalna podela tabele, vertikalna podela tabele i uvođenje izvedene kolone.

Spajanje kolone, slika 2.8. je dodavanje kolone kojoj bi se često pristupalo preko stranog ključa.

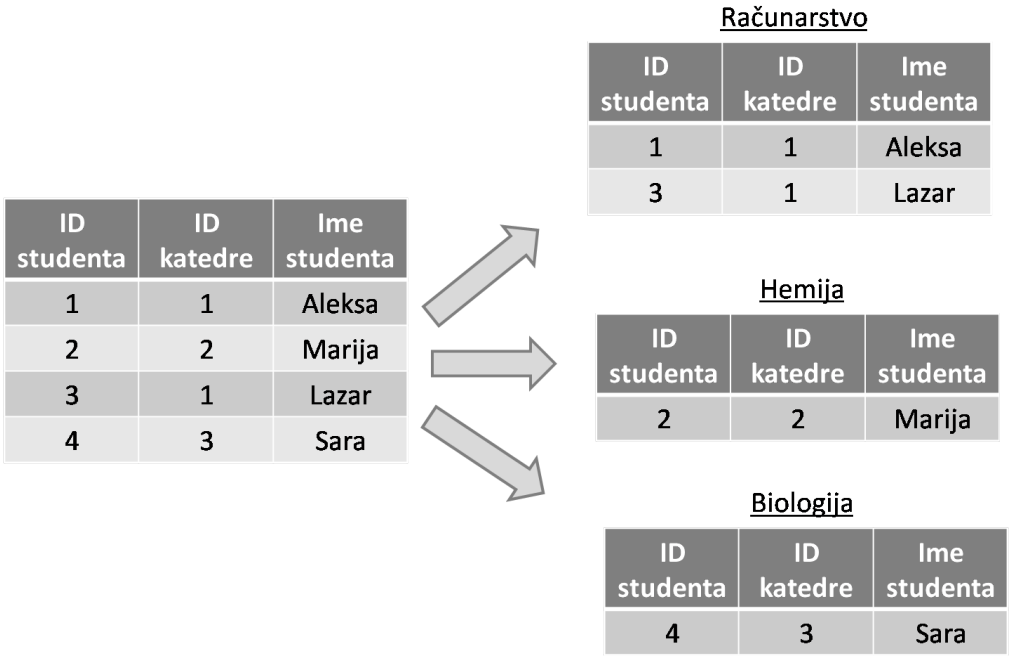


Slika 2.8: Spajanje kolone

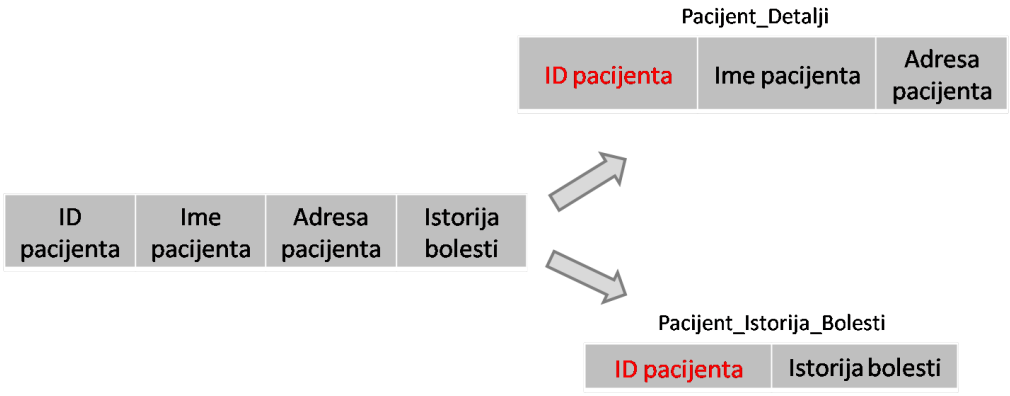
Horizontalno deljenje tabele, slika 2.9. podrazumeva da se na osnovu prirode podataka jedna tabela podeli na više tabela tako da se čitanje svede samo na čitanje grupe redova.

Vertikalno deljenje tabele, slika 2.10. podrazumeva da se tabela podeli grupisanjem kolona koje se često čitaju zajedno.

Uvođenje izvedene kolone, slika 2.11. predstavlja dodavanje kolone koja čuva rezultat neke agregatne funkcije. Time se izbegava da se pri svakom čitanju ta agregatna funkcija izvršava, već se pri ažuriranju stanja ta vrednost ažurira, da bi se rezultat kasnije mogao samo pročitati.



Slika 2.9: Horizontalna podela tabele

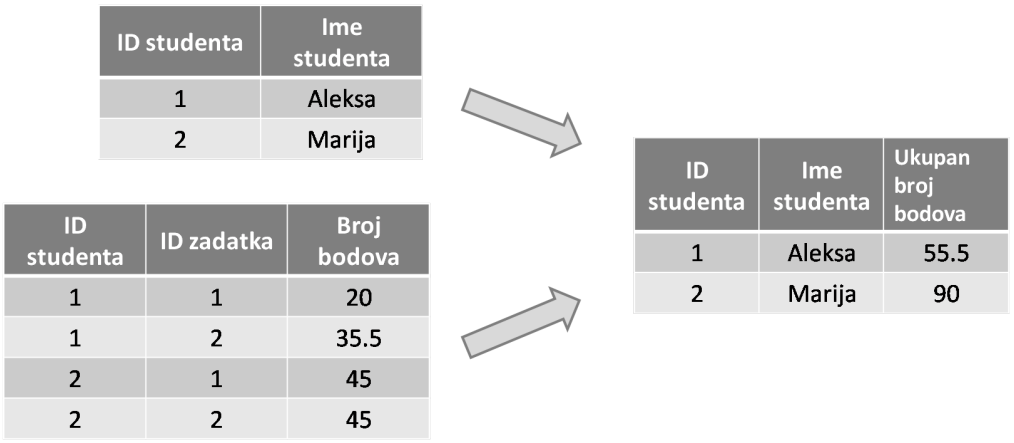


Slika 2.10: Vertikalna podela tabele

ACID i BASE

Transakcija je logička jedinica posla pri radu sa podacima [8]. Kod relacionih modela jednu transakciju karakteriše: atomičnost, konzistentost, izolovanost i trajnost (ACID).

Atomičnost transakcije se može objasniti pravilom: Jedna transakcija se izvršava u celini ili se ne izvršava nijedan njen deo, odnosno dejstvo transakcije je nedeljivo.



Slika 2.11: Uvođenje izvedene kolone

Konzistentost transakcije znači da dejstvo transakcije ne može ostaviti stanje koje narušava integritet baze podataka.

Izolovanost transakcije čini da transakcije ne mogu uticati međusobno jedna na drugu, odnosno, kada se jedna transakcija pokrene, pa sve dok se ne završi, za nju, izmena neke druge transakcije neće biti vidljiva.

Trajnost transakcije garantuje da će kompletirana transakcija u slučaju prekida rada sistema pre nego što su izmene upisane na disk, biti upamćena i izvršena nakon restarta sistema. Svaka izmena se upisjue u log fajl pre nego što je upisana na disk, kako bi se operacije mogle poništiti u slučaju poništavanja transakcije.

Ova svojstva obično karakterišu transakcije u relacionom modelu, kada konzistentost baze ima nešto veći prioritet od brzine i dostupnosti. Kod kolonski orijentisanih baza podataka, posebno u distribuiranom režimu, dostupnost i brzina često imaju veći prioritet od stalne konzistentosti. Kod njih se koristi alternativni pristup karakterizacije transakcije - BASE. BASE transakcije zadovoljavaju sledeća svojstva: suštinska raspoloživost, postojanje mekog stanja i konvergentna konzistentnost.

Suštinska raspoloživost omogućava maksimalnu dostupnost čitanja i pisanja, ali bez garancije konzistentosti.

Meko stanje daje mogućnost da se stanje baze podataka menja čak i kada nijedna transakcija nije u toku, i to u periodu postizanja konzistetnosti.

Konvergentna konzistentnost obezbeđuje da će baza podataka u slučaju da nema novih upisa, postati konzistentna kroz neki vremenski period.

CAP teorema

Kvalitet distribuiranih baza podataka ogleda se kroz tri svojstva: konzistentost, raspoloživost i tolerancija razdvojenosti.

Konzistenost u ovom kontekstu razlikuje se od konzistentosti transakcije. U kontekstu distribuiranih sistema, konzistetnost je svojstvo baze podataka, koje garantuje da će odgovor koji se šalje sa bilo kog čvora klastera (ukoliko odgovora ima) biti konzistentan .

Raspoloživost je svojstvo koje obezbeđuje da će baza uvek vratiti neki odgovor.

Tolerancija razdvojenosti znači da će u slučaju delimičniih otkaza unutar klastera, sistem i dalje vraćati ispravne odgovore.

CAP teorema koju je formulisao Eric Brewer [2], navodi da distribuirana baza podatka ne može istovremeno ispunjavati sva tri svojstva, slika 2.12.

Obzirom da je konzistentost i dostupnost u praksi skoro nemoguće dostići, distribuirane baze podataka organizuju se u skladu sa tim da li se veći prioritet daje dostupnosti ili stalnoj konzistentosti. Priroda sistema koji koriste relacioni model obično je takva da daju prioritet konzistetnosti, pa se od relacionog modela očekuje da u distribuiranom okruženju osim konzistetnosti nudi i toleranciju razdvojenosti, za razliku od kolonski orijentisane nerelacione baze koja konzistetnost ne garantuje (garantuje konvergentnu konzistenciju), ali uz toleranciju razdvojenosti nudi stalnu dostupnost.

[2]Eric Allen Brewer profesor računarskih nauka na Univerzitetu Kalifornija, Berkli

Slika 2.12: CAP teorema

Glava 3

Slučajevi upotrebe

Kako bi se postigao dovoljan dokaz koncepta (engl. *proof of concept*) kreirano je više scenarija koji na različit način pristupaju radu sa podacima. Scenariji, odnosno slučajevi upotrebe koji su testirani u ovom radu jesu:

1. Onlajn transakciono procesiranje (OLTP)
2. Onlajn analitičko procesiranje (OLAP)
3. Distribuirano okruženje

3.1 Onlajn transakciono procesiranje (OLTP)

Onlajn transakciono procesiranje predstavlja procesiranje podataka koje se sastoji iz velikog broja transakcija koje se mogu paralelno izvršavati. Svaka pojedinačna transakcija obično deluje na manjem broju podataka i obično uključuje njihovu izmenu, a kada se radi čitanje podataka, ono je obično po ključu [3]. Kod ovakvog vida procesiranja, gde su izmene podataka česte, čuvanje integriteta podataka prilikom tih upisa je prioritet, pa je često sastavni deo finanskijskih sistema, sistema za rezervacije i sl. Konkretan scenario koji će u radu simulirati OLTP okruženje je prebacivanje sredstava sa jednog računa na drugi. Svako prebacivanje novca predstavljaće jednu poslovnu transakciju koja se sastoji iz više transakcija baze podataka. Definicija poslovne transakcije koja se koristi za testiranje delom je preuzeta iz TPC-C specifikacije [1].

Jedna poslovna transakcija sastoji se iz tri transakcije baze podataka: kreiranje transfera sredstava (*createFXTransaction*), izvršavanje uplate (*executePayment*) i provera statusa transfera (*checkTransactionStatus*).

CreateFXTransaction sastoji se iz provere sredstava na računu korisnika i unosa novog transfera u tabelu.

ExecutePayment radi ažuriranje računa korisnika i menja status transfera

CheckTransaction dohvata status transfera.

3.2 Onlajn analitičko procesiranje (OLAP)

Onlajn analitičko procesiranje sačinjeno je od velikog broja čitanja podataka i manjeg broja izmena podataka (koje su obično masivne - *bulk*). Upiti koji se koriste obično imaju parametre, visok nivo kompleksnosti i visok procenat podataka kojima pristupaju. OLAP je obično sastavni deo sistema koji nude podršku za donošenje poslovnih odluka, generisanje složenih izveštaja kao i bilo koji vid korišćenja velike količine podataka koristeći složene upite i operacije. Scenario koji se simulira u ovom radu u svrhu testiranja, jeste primer funkcionisanja trgovinskog lanca sa skupom svojih mušterija, proizvoda, dobavljača i narudžbina. Korišćen model delom je preuzet iz TPC-H specifikacije [2]. Test će se sastojati iz paralelnog izvršavanja složenog upita koji iz tabele sa stavkama narudžbina izvlači vrednosti agregatnih funkcija određenih kolona grupisanih po statusu. Parametri upita su dan slanja narudžbine i status po kojima filtriramo rezultat.

3.3 Distribuirano okruženje

Distribuirani sistemi su sistemi kod kojih se podaci nalaze na više povezanih jedinica, odnosno čvorova kako bi se postigla skalabilnost, umanjilo kašnjenje odgovora (engl *latency*) i povećala dostupnost sistema u slučaju problema. Dva načina distribuiranja podataka su: **replikacija** i **particionisanje**.

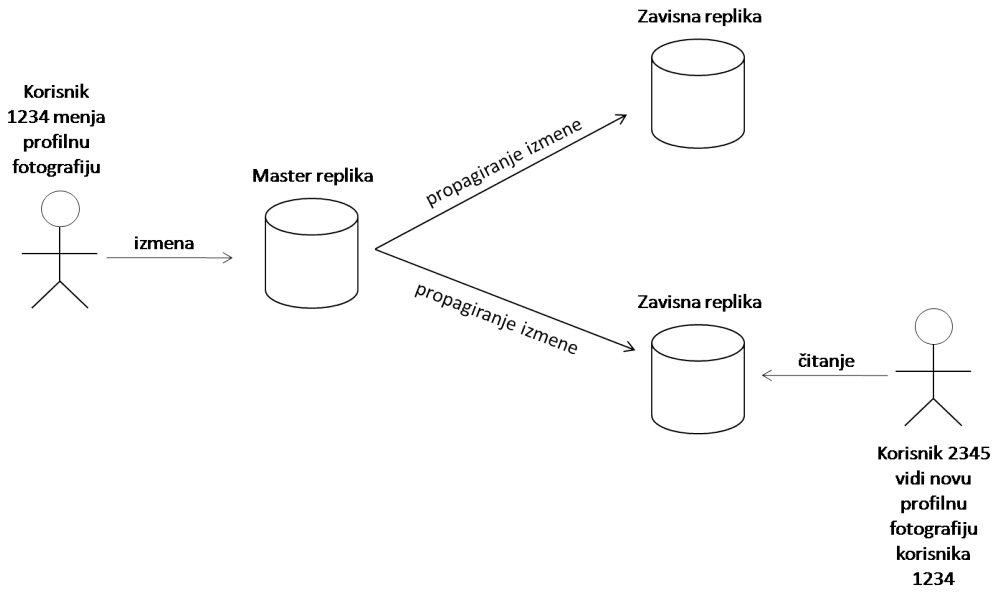
Particionisanje distribuira podatke tako što se veliki skupovi podataka podele na manje celine i čuvaju na različitim čvorovima.

Replikacija podrazumeva da se više kopija podataka čuva na različitim čvorovima. Ukoliko neki od čvorova postane nedostupan, klijentski zahtevi mogu biti obrađeni na nekom od preostalih čvorova. Kako bi svaka replika bila ažurna neophodno je da se vaka izmena podataka propagira do svakog čvora. Najpoznatiji model po kojem se to realizuje je *master-slave*, slika 3.1. (postoje i alternative: *multi-master* model i *masterless* model, ali usled njihove složenosti oni se koriste samo u nekim specifičnim slučajevima). Po tom modelu jedna od replika izabrana

je da bude master replika, i svaka izmena podataka mora ići kroz nju, a onda ona izmenu propagira do ostalih zavisnih (*slave*) replika. Čitanje podataka se sa druge strane, može raditi sa bilo kog čvora.

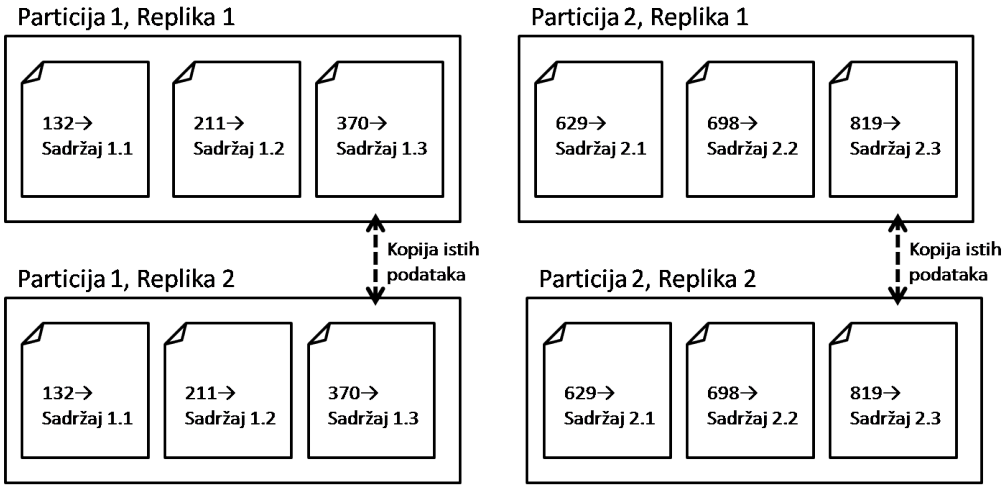
Replikacija može biti sinhrona i asinhrona. Kod sinhrone replikacije kada izmena dođe do master čvora, on pre nego što klijentu vrati odgovor da je izmena uspešno sačuvana, sačeka da mu svaki zavisni čvor potvrdi da je izmena uspešno upisana kod njega. Sinhrona replikacija je karakteristična za sisteme koji garantuju da će klijent pri čitanju podatka sa bilo koje replike imati ažurno stanje, međutim, cena toga je umanjena dostupnost, s obzirom da ukoliko neki od zavisnih čvorova postane nedostupan, ne može master čvoru potvrditi da je izmena upisana na njega, pa samim tim ni master ne može klijentu potvrditi da je izmena uspešno upisana. Kod asinhrone replikacije, kada izmena podataka dođe do mastera, i on tu izmenu propagira ka ostalim zavisnim čvorovima, master ne čeka nikakav odgovor od zavisnih čvorova, već ukoliko je izmena uspešno upisana na njega on klijentu vraća potvrdu o sačuvanoj izmeni. Asinhrona replikacija obično nudi bolje performanse i veću dostupnost servisa, po cenu toga da čitanje sa neke od replika mogu vraćati zastarele podatke.

Kako relacione baze podataka daju prioritet konzistentosti (CP iz CAP teoreme) najčešće se kod njih koristi sinhrona replikacija, dok je kod modela kojima je dostupnost (AP iz CAP teoreme) prioritet, asinhrona replikacija model po kojem se podaci distribuiraju.



Slika 3.1: Master-slave replikacija

Replikacija i particionisanje obično idu zajedno, tako što se podaci prvo podele na particije, a svaka particija replikuje na više čvorova [4], slika 3.1.



Slika 3.2: Replikacija i particionisanje

Primer koji će biti analiziran jeste rad jednostavnog distribuiranog DNS servera[1].

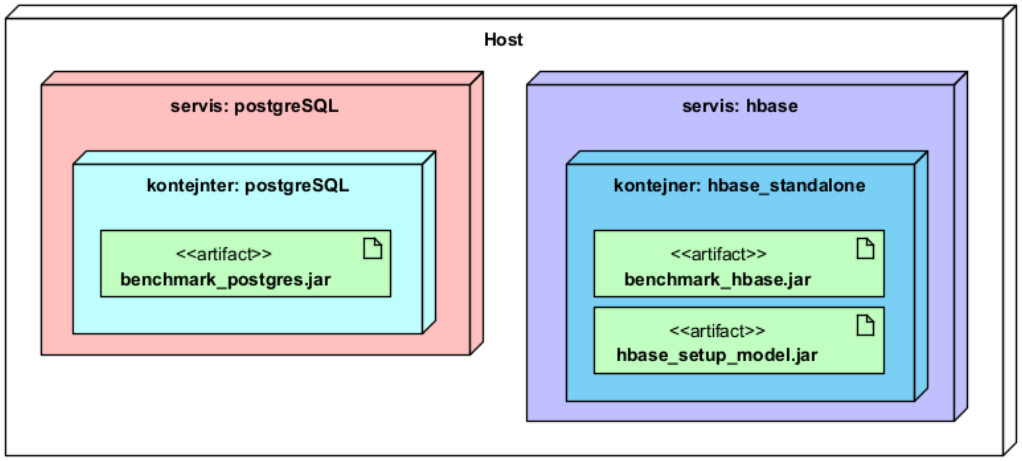
[1]Usled složenosti implementacije distribuiranog PostgreSQL sistema, za distribuirano okruženje primer je dat samo za HBase, a upoređivanje je dato na osnovu teorijskih istraživanja

Glava 4

Merenje performansi po modelima

4.1 Opis platforme za testiranje

Za predstavnike baza podataka izabrani su HBase (kao predstavnik kolonski orijentisane nerelacione baze podataka) i PostgreSQL (kao predstavnik relacione baze podataka) [1]. Kao okruženje za izvršavanje testova korišćen je *docker*. Serveri oba predstavnika biće pokrenuti kao nezavisni kontejneri.



Slika 4.1: Platforma testiranja

[1]Napomena: Kako su za predstavnike izabrani PostgresSQL i HBase, rezultati dobijeni u nastavku su rezultati poređenja konkretnih predstavnika, i nisu opšti za sve relacione i kolonski orijenisane nerelacione baze podataka.

4.2 Merenje performansi u OLTP okruženju

Model baze podataka za testiranje u OLTP okruženju koji se koristu u radu, sastoji se iz četiri tabele:

- **fxrates**: Sadrži informacije o kursu valutnih parova. Sadrži 16 redova.
- **fxuser**: Podaci o korisnicima. Sadrži 30 000 redova.
- **fxaccount**: Podaci o računima korisnika. Sadrži 120 000 redova.
- **fxtransaction**: Sadrži informacije o transferima, odnosno transakcijama sredstava.

setup-postgres-model.sql

```
1
2  create table postgresdb.fxrates (
3          currency_from  varchar(50) not null,
4          currency_to varchar(50) not null,
5          rate real not null,
6          primary key(currency_from,currency_to)
7  );
8
9  create table postgresdb.fxuser (
10         id  integer primary key,
11         username varchar (50) unique not null,
12         password varchar (50) not null,
13         start_balancecurrency varchar (10) not null,
14         start_balance real not null,
15         firstname varchar(100) not null,
16         lastname varchar(100) not null,
17         street varchar(100) not null,
18         city varchar(100) not null,
19         state varchar(100) not null,
20         zip varchar(50) not null,
21         phone varchar(30) not null,
22         mobile varchar(30) not null,
23         email varchar(50) unique not null,
24         created timestamp not null
25 );
26
27 create table postgresdb.fxaccount (
```

```
28          id  integer primary key,
29          fxuser integer not null  references postgresdb.fxuser(id),
30          currency_code varchar(10) not null,
31          balance real not null,
32          created timestamp not null,
33          unique (fxuser, currency_code)
34  );
35
36  create table postgresdb.fxtransaction (
37          id integer primary key,
38          fxaccount_from  integer references postgresdb.fxaccount(id),
39          fxaccount_to  integer references postgresdb.fxaccount(id),
40          amount numeric(15,2) not null,
41          status varchar(50) not null,
42          entry_date timestamp not null
43  );
```

setup-hbase-model.sh

```
1  create fxrates, 'data';
2  create fxuser, 'data';
3  create fxaccount, 'data';
4  create fxtransaction, 'data';
```

Kao što je ranije spomenuto, poslovna transakcija koja se testira jeste prebacivanje sredstava sa jednog računa na drugi. Ona se sastoji iz tri dela: kreiranje transfera sredstava (*createFxTransaction*), izvršavanja uplate(*executePayment*) i provera statusa transfera (*checkTransactionStatus*);

CreateFXTransaction prvo pročita stanje naloga sa kojeg treba preneti sredstva, nakon toga čita kurs odgovarajućeg valutnog para i ukoliko ima dovoljno sredstava na računu, u tabelu sa transferima upisuje transfer u statusu *NEW*.

CreateFXTransaction

```
1
2  select fa.balance
3  from fxaccount fa
4  where fa.id = ?;
5
6  select fr.rate
7  from fxrates fr
8  where fr.currency_to = ? and fr.currency_from = ?;
9
10 insert into fxtransaction
11 (id,fxaccount_from, fxaccount_to, amount, status, entry_date)
12 values(?,?,?,?,?,?)
```

ExecutePayment prvo pročita stanja sa naloga koji učestvuju u transferu, menja im balans i nakon toga transferu menja status.

ExecutePayment

```
1
2  select balance
3  from fxaccount
4  where id = ?
5
6  update fxaccount
7  set balance = ?
8  where id = ?
9
10 select balance
11 from fxaccount
12 where id = ?
13
14 update fxaccount
15 set balance = ?
16 where id = ?
17
18 update fxtransaction
19 set status = ?
20 where id = ?
```

CheckTransactionStatus za odgovarajuću transakciju čita status po primarnom ključu.

CheckTransactionStatus

```
select status
from fxtransaction
where id = ?
```

Test ima podršku za paralelno izvršavanje poslovnih transakcija. Parametri testa su **broj klijenata** (*numOfClients*) i **broj poslovnih transakcija koje treba izvršiti** (*totalTransactions*). Oba parametra postavljaju se prilikom pokretanja testa, kroz standardni ulaz. Politika dodeljivanja poslovnih transakcija svakom od klijenata je da se ukupan broj poslovnih transakcija ravnomerno podeli svim klijentima, a eventualni ostatak pri podeli dodeljuje se nekom od njih.

Implementacija politike podele posla klijentima

```
List<Integer> transClientList = new ArrayList<>();
int transToAssign = totalTransactions;
int transPerClient = transToAssign / numOfClients;

for(int i = 0; i<numOfClients;i++){
   transClientList.add(transPerClient);
   transToAssign-=transPerClient;
}
if (transToAssign > 0) {
   int transForLast = transPerClientList.get(numOfClients - 1);
   transClientList.set(numOfClients - 1, transForLast + transToAssign);
}

assert numOfClients==transClientList.size();
Thread[] threads = new Thread[numOfClients];
for(int i = 0;i<numOfClients;i++){
   threads[i] = new Thread(
      new BenchmarkSingleClientExecutor(
         i*transClientList.get(i),transClientList.get(i)
         )
      );
 }
```

Nakon što se klijentu dodeli skup poslovnih transakcija koje treba da obradi, on krene da izvršava sve tri faze svake poslovne transakcije koja mu je dodeljena. Svaki transfer koji učestvuje u poslovnoj transakciji generisan je na osnovu rednog broja poslovne transakcije koju klijent procesira. To je garancija da se ne može

desiti da se isti transfer obrađuje više puta, kao i to da dva klijenta ne mogu obrađivati isti transfer.

BenchmarkSingleClientExecutor.java

```
1
2  public class BenchmarkSingleClientExecutor implements Runnable {
3
4  private final CountDownLatch endSignal;
5  private final BenchmarkOLTPUtility oltpUtil;
6  private final int numOfT;
7  private final int startFrom;
8
9  private final Object connection;
10
11 @Override
12 public void run() {
13
14 try {
15    for (int i = this.start; i < this.start + this.numOfT; i++) {
16      FXTransaction fxTransaction = DataGenerator.seedTransacton(i);
17      ExecutePaymentInfo executePaymentInfo =
18          oltpUtil.createFXTransaction(connection);
19      oltpUtil.executePayment(,executePaymentInfo);
20      oltpUtil.checkTransactionStatus(fxT);
21     }
22     endSignal.countDown();
23 } catch (Exception e) {
24      throw new IllegalStateException(e);
25  }
26 }
27 }
```

Priprema okruženja za testiranje podrazumeva kompilaciju java testova, pokretanje docker kontejnera i prebacivanje kompiliranih testova na odgovarajuće kontejnere. Dodatan korak za HBase jeste da se na HBase kontejneru kreira struktura baze podataka. Skripta u nastavku sadrži sve neophodne komande za pokretanje okruženja.

prepare-env.sh

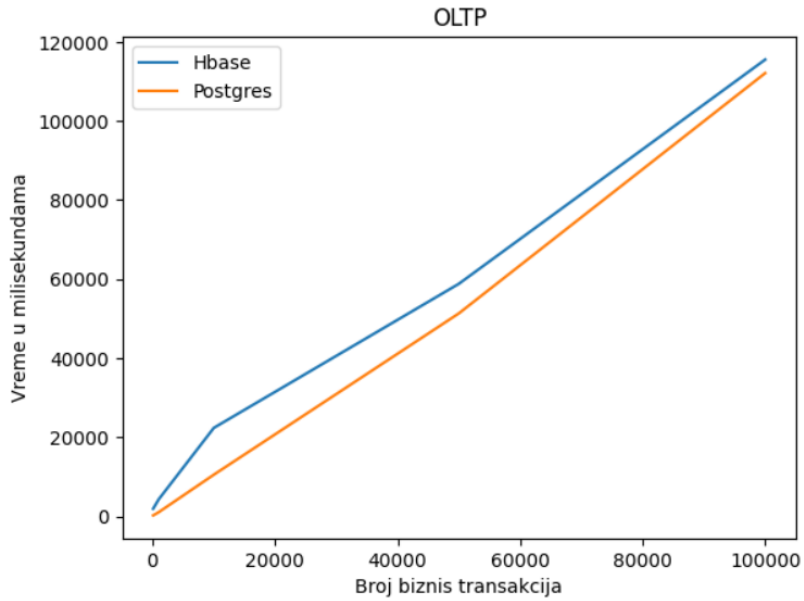
```
#!/bin/bash
echo 'PREPARING ENVIRONMENT...';

export JAVA_HOME="$JAVA_8";
mvn -f ./hbase_setup_model clean compile assembly:single;
mvn -f ./benchmark_hbase clean compile assembly:single;
export JAVA_HOME="$JAVA_17";
mvn -f ./benchmark_postgres clean compile assembly:single;

docker-compose -f docker-compose.yml up --build -d;
docker exec -it hbase-master-1 sh -c "java -jar hbase_setup_model.jar";
```

docker-compose.yml

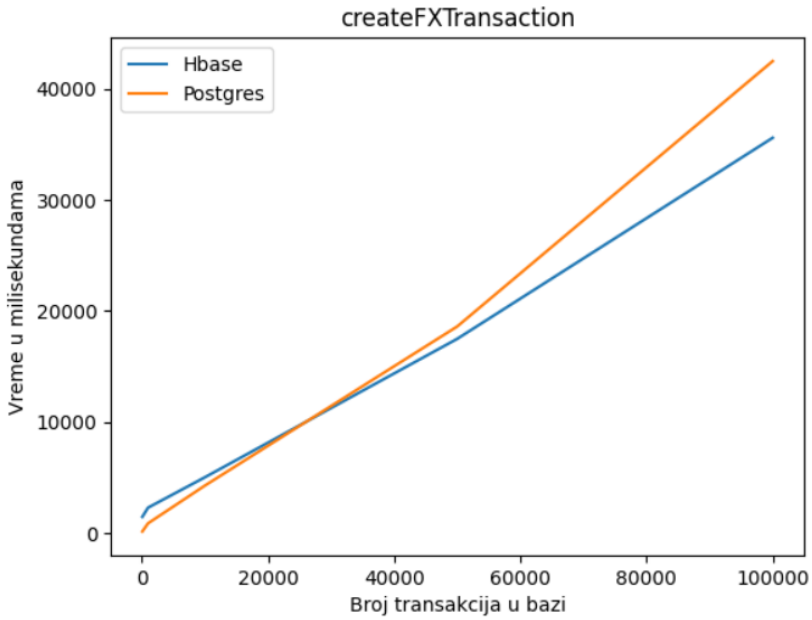
```
services:
  postgres:
    container_name: postgres
    ports:
      - "5433:5432"
    volumes:
      - ./setup_model.sql:/docker-entrypoint-initdb.d/create_script.sql
      - ./benchmark_postgres.jar:/benchmark_postgres.jar
    environment:
      - POSTGRES_PASSWORD=postgres
      - POSTGRES_USER=postgres
      - POSTGRES_DB=postgresdb
    build:
      context: .
      dockerfile: ./Dockerfile_postgres

  hbase:
    image: bde2020/hbase-standalone:1.0.0-hbase1.2.6
    container_name: hbase
    volumes:
      - hbase_data:/hbase-data
      - hbase_zookeeper_data:/zookeeper-data
      - ./hbase_setup_model.jar:/hbase_setup_model.jar
      - ./benchmark_hbase.jar:/benchmark_hbase.jar
    ports:
      - 16000:16000
      - 16010:16010
      - 16020:16020
```

```
29        - 16030:16030
30        - 2888:2888
31        - 3888:3888
32        - 2181:2181
33
34  volumes:
35    hbase_data:
36    hbase_zookeeper_data:
```

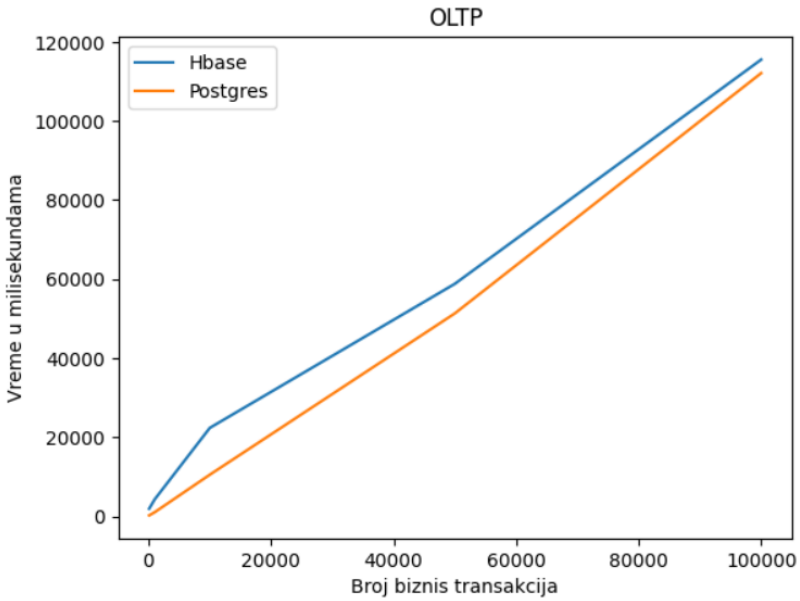
Rezultati merenja prikazani u nastavku nastali kao rezultat pokretanja testova sa 100, 1 000, 10 000, 50 000, 100 000 poslovnih transakcija i pet klijenata koji te poslovne transakcije paralelno obrađuju.



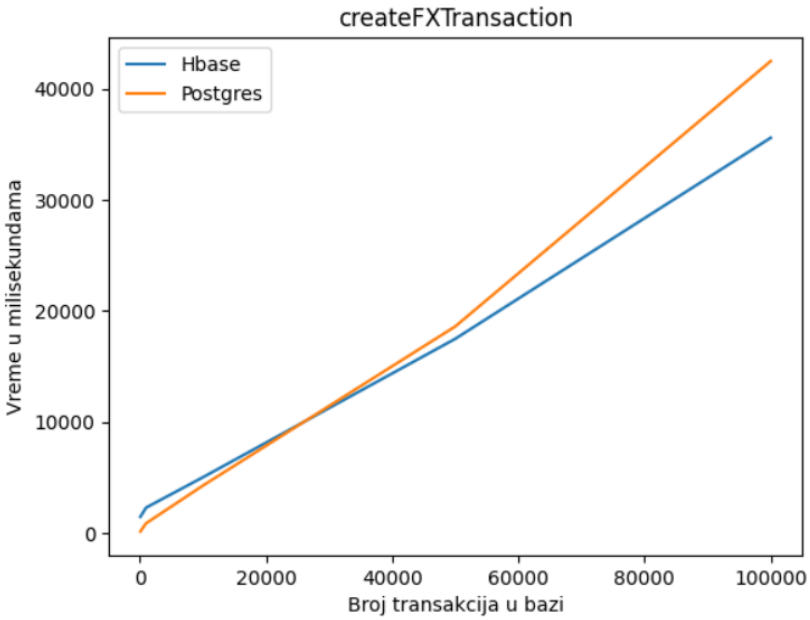
Slika 4.2: Rezultati merenja u OLTP okruženju



Slika 4.3: Rezultati merenja createFXTransaction dela biznis transakcije



Slika 4.4: Rezultati merenja u OLTP okruženju



Slika 4.5: Rezultati merenja createFXTransaction dela biznis transakcije

4.3 Merenje performansi u OLAP okruženju

Model baze podataka koji se koristi za testiranje u OLAP okruženju sastoji se iz sledećih tabela:

- **product**: Sadrži informacije o proizvodima. Sadrži 3 000 000 redova.
- **supplier**: Podaci o dovaljačima. Sadrži 1 000 000 redova.
- **productsupplier**: Vezna tabela između dobavljača i proizvoda. Sadrži 5 000 000 redova.
- **customer**: Informacije o mušterijama. Sadrži 1 500 000 redova.
- **order**: Informacije o narudžbinama. Sadrži 1 500 000 redova.
- **orderitem**: Informacije o pojedinim stavkama narudžbine. Sadrži 6 000 000 redova.

setup-postgres-model.sql

```
1 create table product (
2     id  integer not null primary key,
3     name varchar(50) not null,
```

```
4       brand varchar(50) not null,
5       type varchar(50) not null,
6       size integer not null,
7       container varchar(50) not null,
8       price varchar(50) not null,
9       comment varchar(50)
10  );
11
12  create table supplier (
13      id  integer primary key,
14      name varchar(50) not null,
15      address varchar(200) not null,
16      phone varchar(50) not null
17  );
18
19  create table productsupplier (
20      id integer not null primary key,
21      product integer not null,
22      supplier integer not null,
23      available integer not null,
24      supply_cost real not null,
25      comment varchar(200),
26      constraint fk_product
27      foreign key(product) references  postgresdb.product(id),
28      constraint fk_supplier
29      foreign key(supplier) references  postgresdb.supplier(id)
30
31  );
32
33  create table customer(
34      id integer primary key,
35      name varchar(50) not null,
36      address varchar(200) not null,
37      phone varchar(50) not null,
38      comment varchar(200)
39  );
40
41  create table order(
42      id integer primary key,
43      customer integer not null,
44      status varchar(20) not null,
45      total_price real not null,
```

```
46      entry_date date not null,
47      priority varchar(20) not null,
48      comment varchar(200),
49      constraint fk_customer
50      foreign key(customer) references  postgresdb.customer(id)
51  );
52
53  create table order_item(
54      order_id integer not null,
55      product integer not null,
56      supplier integer not null,
57      order_no integer not null,
58      quantity integer not null,
59      base_price real not null,
60      discount real not null,
61      tax real not null,
62      status varchar(20) not null,
63      ship_date date not null,
64      commit_date date not null,
65      comment varchar(200),
66      primary key(order_id,product,supplier),
67      constraint fk_order
68      foreign key(order_id) references  postgresdb.order(id),
69      constraint fk_product
70      foreign key(product) references  postgresdb.product(id),
71      constraint fk_supplier
72      foreign key(supplier) references  postgresdb.supplier(id)
73  );
```

hbase-setup-model

```
1
2  create product, 'data';
3  create supplier, 'data';
4  create productsupplier, 'data';
5  create customer, 'data';
6  create order, 'data';
7  create orderitem, 'data';
```

OLAP test će obuhvatiti upunjavanje tabela iz csv fajlova (*bulk load*), kao i iz izvršavanja upita koji će klijenti paralelno izvršavati više puta. Upit će biti parametrizovan i uzimaće dva parametra. Prvi parametar je dan narudžbine, a

drugi je njen status. Na osnovu tih parametara dohvataju se agregirane vrednosti kolona iz tabele *orderitem*, grupisane po statusu.

bulkLoad

```
1 copy product from "./product.csv";
2 copy supplier from "./supplier.csv";
3 copy productsupplier "./productsupplier.csv";
4 copy customer from "./customer.csv";
5 copy order from "./order.csv";
6 copy order_item from "./order_item.csv";
```

executeOLAPQuery

```
1
2 select
3         oi.status status,
4         sum(oi.quantity) as sum_qty,
5         sum(oi.base_price) as sum_base_price,
6         sum(oi.base_price*(1-oi.discount)) as sum_disc_price,
7         sum(oi.base_price*(1-oi.discount)*(1+oi.tax)) as sum_charge,
8         avg(oi.quantity) as avg_qty,
9         avg(oi.base_price) as avg_price,
10        avg(oi.discount) as avg_disc,
11        count(*) as count_order
12 from
13         postgresdb.order_item oi
14 where
15        oi.ship_date = to_date(?,'dd.mm.yyyy') and
16         oi.status = ?
17 group by oi.status;
```

Parametri testa su **broj klijenata** (*numOfClients*) koji će paralelno izvršavati olap upit i **ukupan broj izvršavanja upita** (*totalIterations*). Oba parametra postavljaju se prilikom pokretanja testa, kroz standardni ulaz. Politika dodeljivanja broja izvršavanja upita svakom od klijenata ista je kao i dodeljivanja broja poslovnih transakcija klijentima, kod testiranja OLTP okruženja.

Implementacija politike podele posla klijentima

```
1 List<Integer> iterClientList = new ArrayList<>();
2 int itersToAssign = totalIterations;
3 int itersPerClient = itersToAssign / numOfClients;
4
```

```
5  for(int i = 0; i<numOfClients;i++){
6     iterClientList.add(itersPerClient);
7     itersToAssign-=itersPerClient;
8  }
9  if (itersToAssign > 0) {
10    int itersForLast = iterClientList.get(numOfClients - 1);
11    iterClientList.set(numOfClients - 1, itersForLast + itersToAssign);
12 }
13
14 assert numOfClients==iterClientList.size();
15 Thread[] threads = new Thread[numOfClients];
16 for(int i = 0;i<numOfClients;i++){
17    threads[i] = new Thread(
18       new BenchmarkSingleClientExecutor(
19          i*iterClientList.get(i),iterClientList.get(i)
20          )
21       );
22  }
```

Svaki klijent nakon što mu je dodeljen broj iteracija, kreće da izvršava upit onoliko puta koliko mu je iteracija dodeljeno. Svako izvršavanje OLAP upita ima iste parametre.

BenchmarkSingleClientExecutor.java

```
1  public class BenchmarkSingleClientExecutor implements Runnable {
2
3  private final CountDownLatch endSignal;
4  private final BenchmarkOLAPUtility olapUtil;
5  private final int numOfIters;
6  private final int startFrom;
7
8  private final Object connection;
9  @Override
10 public void run() {
11
12   try {
13     for (int i = this.start; i < this.start + this.numOfIters; i++) {
14        olapUtil.executeOLAPQuery(connection);
15      }
16      endSignal.countDown();
17     } catch (Throwable e) {
18       throw new IllegalStateException(e);
```

```
19      }
20    }
21  }
```

Priprema okruženja za testiranje podrazumeva generisanje csv fajlova koji kasnije treba da budu učitani u tabele koristeći podršku za *bulk load*, zatim kompilaciju java testova, pokretanje docker kontejnera i prebacivanje kompiliranih testova kao i izgenerisanih csv fajlova na odgovarajuće kontejnere. Dodatan korak za HBase jeste da se na HBase kontejneru kreira struktura baze podataka. Skripta u nastavku sadrži sve neophodne komande za pokretanje okruženja.

prepareEnv.sh

```
1  #!/bin/bash
2  echo 'PREPARING ENVIRONMENT...';
3  echo 'PREPARING HBASE BENCHMARK JARS...';
4  export JAVA_HOME="$JAVA_8";
5  mvn -f olap_benchmark_hbase clean compile assembly:single;
6  mvn -f hbase_setup_olap_model clean compile assembly:single;
7  mvn -f hbase_bulk_load_setup clean compile assembly:single;
8
9  echo 'PREPARING HBASE BULK LOAD RESOURCES..';
10 java -jar ./hbase_bulk_load_setup.jar;
11
12
13 echo 'PREPARING POSTGRES BENCHMARK JARS...';
14 export JAVA_HOME="$JAVA_17";
15 mvn -f olap_benchmark_postgres clean compile assembly:single;
16 mvn -f postgres_bulk_load_setup clean compile assembly:single;
17
18 echo 'PREPARING POSTGRES BULK LOAD RESOURCES..';
19 java -jar postgres_bulk_load_setup.jar;
20
21 docker-compose -f docker-compose.yml up --build -d;
22 winpty docker exec -it hbase sh -c "java -jar setup_olap_model.jar";
```

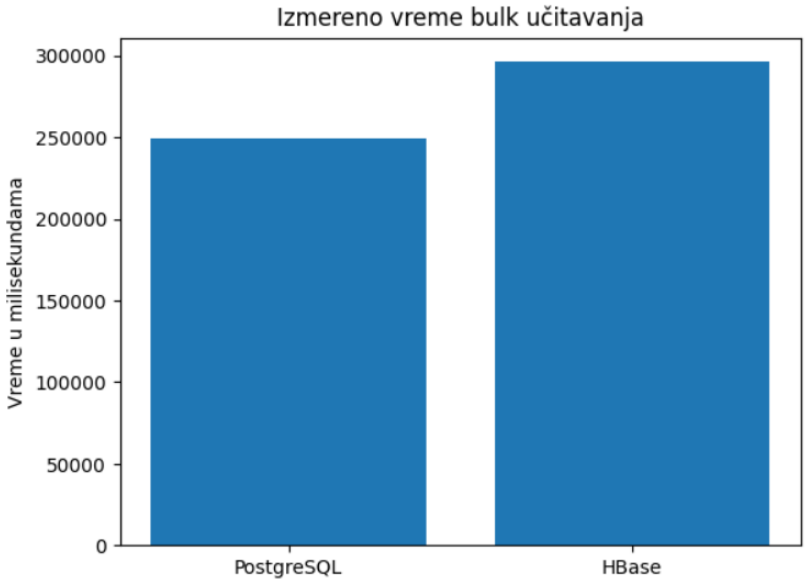
docker-compose.yml

```
1
2  services:
3    postgres:
4      container_name: postgres
5      ports:
```

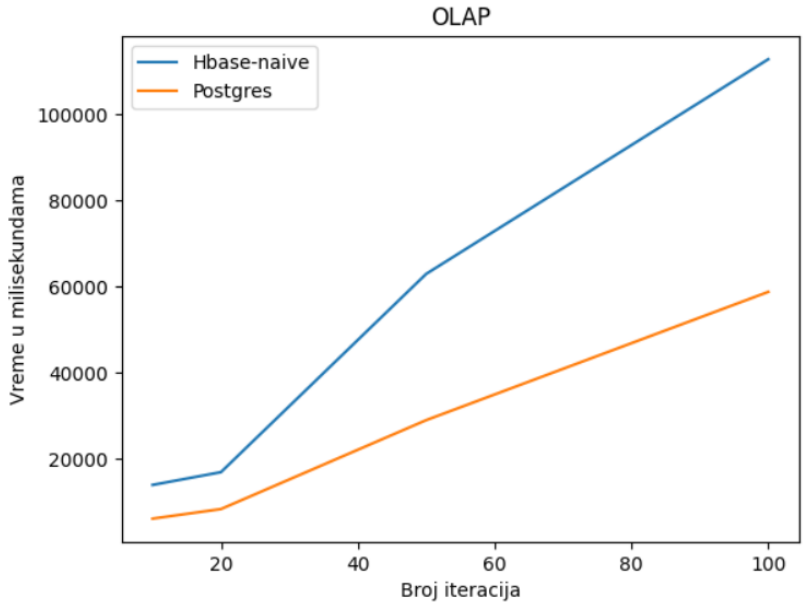
```
6       - "5433:5432"
7     volumes:
8       - ./setup_model.sql:/docker-entrypoint-initdb.d/create_script.sql
9       - ./benchmark_postgres.jar:/benchmark_postgres.jar
10    environment:
11      - POSTGRES_PASSWORD=postgres
12      - POSTGRES_USER=postgres
13      - POSTGRES_DB=postgresdb
14    build:
15      context: .
16      dockerfile: ./Dockerfile_postgres
17
18  hbase:
19    image: bde2020/hbase-standalone:1.0.0-hbase1.2.6
20    container_name: hbase
21    volumes:
22      - ./productsupplierHB.csv:/productsupplier.csv
23      - ./productHB.csv:/product.csv
24      - ./supplierHB.csv:/supplier.csv
25      - ./customerHB.csv:/customer.csv
26      - ./orderHB.csv:/order.csv
27      - ./orderitemHB.csv:/orderitem.csv
28      - ./orderitemStatsHB.csv:/orderitemstats.csv
29      - hbase_data:/hbase-data
30      - hbase_zookeeper_data:/zookeeper-data
31      - ./hbase_setup_model.jar:/hbase_setup_model.jar
32      - ./benchmark_hbase.jar:/benchmark_hbase.jar
33    ports:
34      - 16000:16000
35      - 16010:16010
36      - 16020:16020
37      - 16030:16030
38      - 2888:2888
39      - 3888:3888
40      - 2181:2181
41
42 volumes:
43   hbase_data:
44   hbase_zookeeper_data:
```

Rezultati merenja prikazani u nastavku nastali kao rezultat pokretanja testova sa 100, 1 000, 10 000, 50 000, 100 000 iteracija izvršavanja OLAP upita i pet klije-

nata koji te upite paralelno izvršavaju. Na slici se može primetiti da u legendi uz HBase stoji reč naivni (*naive*). To se odnosi na model koji se koristi, vše reči o razlogu zašto se ovako napravljen model za HBase u ovom okruženju može smatrati naivnim, biće u sledećem poglavlju.



Slika 4.6: Bulk load podataka



Slika 4.7: OLAP

Glava 5

Analiza rezultata

Analiza rezultata obuhvata interpretaciju rezultata koji su dobijeni merenjima performansi, kao i navođenje eventualnih unapređenja i primedbi koje treba imati u vidu kada se radi sa datim tehnologijama.

5.1 Analiza rezultata kod testiranja u OLTP okruženju

Rezultati merenja ukazuju da kako raste broj poslovnih transakcija koje treba obraditi u testu, inicijalna prednost PostgreSQL-a u odnosu na HBase opada. Konkretno po fazama poslovne transakcije, izdvaja se createFXTransaction transakcija baze podataka, kod koje na postavljenih 50000 i 100000 poslovnih transakcija, HBase ima prednost. Tendencija rezultata ukazuje da bi sa porastom broja poslovnih transakcija (na milion ili deset miliona) PostgreSQL imao veća usporenja, relativno u odnosu na HBase, međutim za testiranje takvog okruženja neophodno je koristiti računar koji je sposoban da sprovede tako zahtevan test.

Važno je naglasiti da HBase ne garantuje konzistentost nad svim podacima kakvu nudi PostgreSQL. HBase nudi jedan vid konzistenosti, i to konzistetnost u radu sa jednom redom tabele (više o tome u 2.2). To dovodi do zaključka da ukoliko je neophodno implementirati transakciju koja uključuje rad sa podacima više tabela ili više redova jedne tabele, ukoliko koristimo HBase, moramo na aplikativnom sloju voditi računa o očuvanju eventualne konzistetnosti. PostgreSQL sa druge strane kao ACID baza podataka, sama garantuje konzistentost, te ne zahteva dodatan napor kao u slučaju HBase-a.

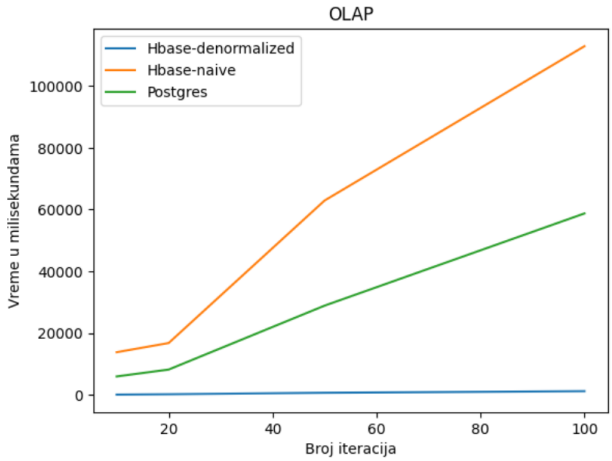
5.2 Analiza rezultata kod testiranja u OLAP okruženju

Rezultati merenja u okviru OLAP rezultata jasno ističu prednost PostgreSQL-a u performansama. Prednost HBase-a u odnosu na PostgreSQL jeste fleksiblnost sheme koja u ovakvom slučaju može doći do izražaja. Vid jednostavne denormalizacije koja se u slučaju HBase-a može primeniti može doneti dramatična poboljšanja u performansama čitanja HBase-a čak i u odnosu na Postgres.

Kreiranje tabele (u ovom slučaju orderitemstats) koja za vrednost ključa ima status, a za kolone ima vrednosti svake potrebne agregirane funkcije po danima, uticalo bi na to da se čitanje izveštaja svodi na čitanje po ključu sa izdvajanjem potrebnih koloni[1]. Primer dodate tabele može se videti na slici 5.1 ,a uticaj na performanse na slici 5.2.

status	sum_qty01072021	sum_base01072021	sum_disc01072021	sum_chg01072021	...	sum_qty12092024	sum_base12092024	avg_qty12092024	...
Status_01	1818.00	234 921.00	32 921.00	32 921.00		219.00	89 321.00	32.50	
Status_02	1829.00	242 021.00	34 331.00	32 987.00		230.00	75 012.00	65.20	
...									

Slika 5.1: Tabela orderitemstats



Slika 5.2: Uporedna analiza performansi postgresa i hbase-a

[1]Razlog zašto se ovakva modifikacija ne može primeniti u slučaju PostgreSQL-a jeste ograničen broj kolona, kao i to što skup kolona mora biti unapred definisan pri kreiranju modela. Kod HBase-a takvi limiti ne postoje.

Glava 6

Zaključak

Bibliografija

[1] Tpc-c. on-line at: https://www.tpc.org/tpcc/.

[2] Tpc-h. on-line at: https://www.tpc.org/tpch/.

[3] What is oltp? on-line at: https://www.oracle.com/database/what-is-oltp/.

[4] Designing data-intensive applications. the big ideas behind reliable, scalable and maintainable systems. 2024.

[5] Sanjay Ghemawat Fay Chang, Jeffrey Dean. Bigtable: A Distributed Storage System for Structured Data. *SIAM Journal on Computing*, 16:486–502, 2006. on-line at: https://static.googleusercontent.com/media/research.google.com/fr//archive/bigtable-osdi06.pdf.

[6] Free Software Foundation. ApacheHBase, 2013. on-line at: https://hbase.apache.org/acid-semantics.html.

[7] Regina O. Obe and Leo S. Hsu. PostgreSQL: Up and Running.

[8] Gordana Pavlović-Lažetić. Uvod u relacione baze podataka. 1999.

Biografija autora

Vuk Stefanović Karadžić (*Tršić, 26. oktobar/6. novembar 1787. — Beč, 7. februar 1864.*) bio je srpski filolog, reformator srpskog jezika, sakupljač narodnih umotvorina i pisac prvog rečnika srpskog jezika. Vuk je najznačajnija ličnost srpske književnosti prve polovine XIX veka. Stekao je i nekoliko počasnih doktorata. Učestvovao je u Prvom srpskom ustanku kao pisar i činovnik u Negotinskoj krajini, a nakon sloma ustanka preselio se u Beč, 1813. godine. Tu je upoznao Jerneja Kopitara, cenzora slovenskih knjiga, na čiji je podsticaj krenuo u prikupljanje srpskih narodnih pesama, reformu ćirilice i borbu za uvođenje narodnog jezika u srpsku književnost. Vukovim reformama u srpski jezik je uveden fonetski pravopis, a srpski jezik je potisnuo slavenosrpski jezik koji je u to vreme bio jezik obrazovanih ljudi. Tako se kao najvažnije godine Vukove reforme ističu 1818., 1836., 1839., 1847. i 1852.