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 [1].

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 [5] . 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 [4].

2.2 Kolonski-orijentisani model

Opšte karakteristike

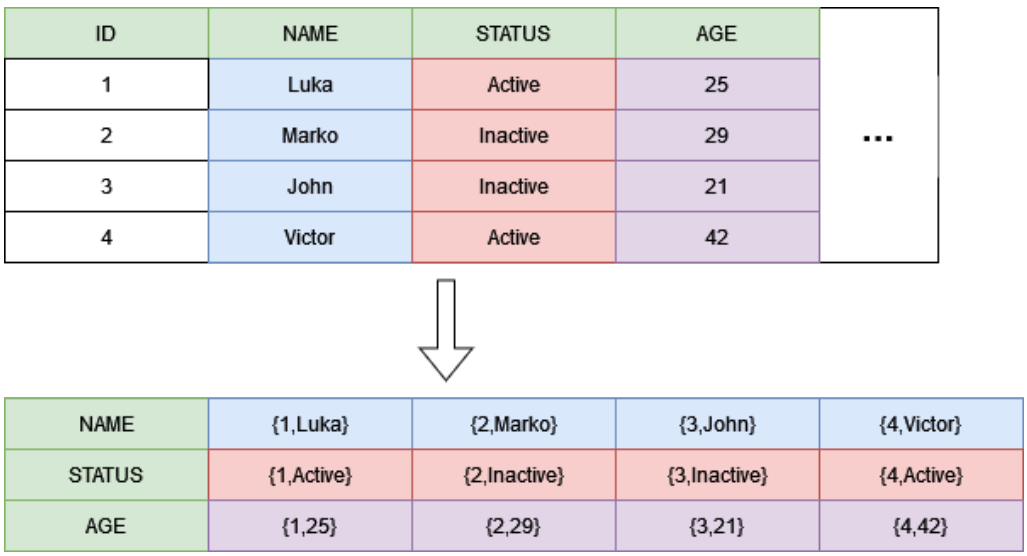
Susret sa Big Data problemima doveo do potrebe za tabelama koje imaju ogroman broj kolona, i ogroman broj redova u okviru tih tabela. Novonastali zahtevi ukazali su na problem kod postojećih relacionih modela. Svaki upit nad tabelom podrazumevao je dohvatanje svih kolona jednog reda, gde bi se filtriranje 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.

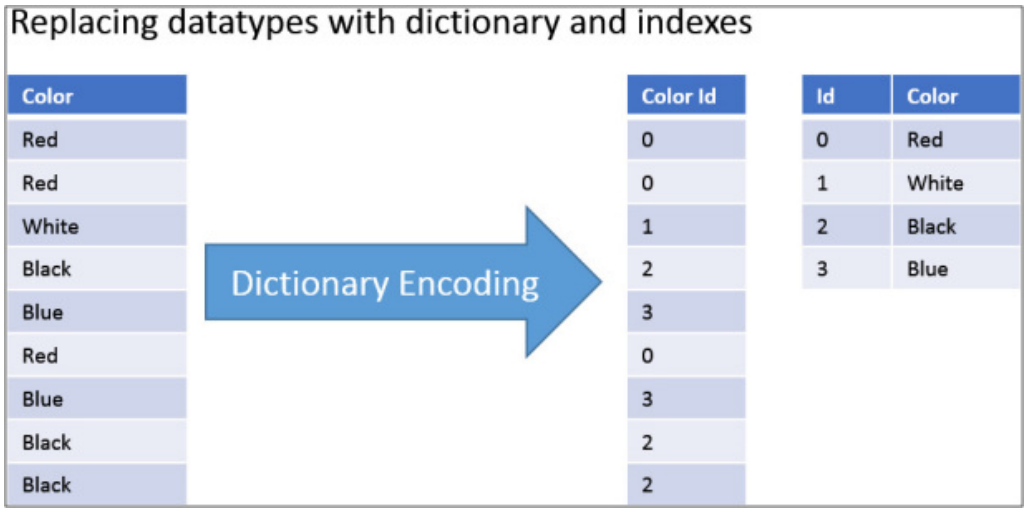


Slika 2.1: Kolonski orijentisan format

Popularni algoritmi kompresije kolonski orijentisanog modela

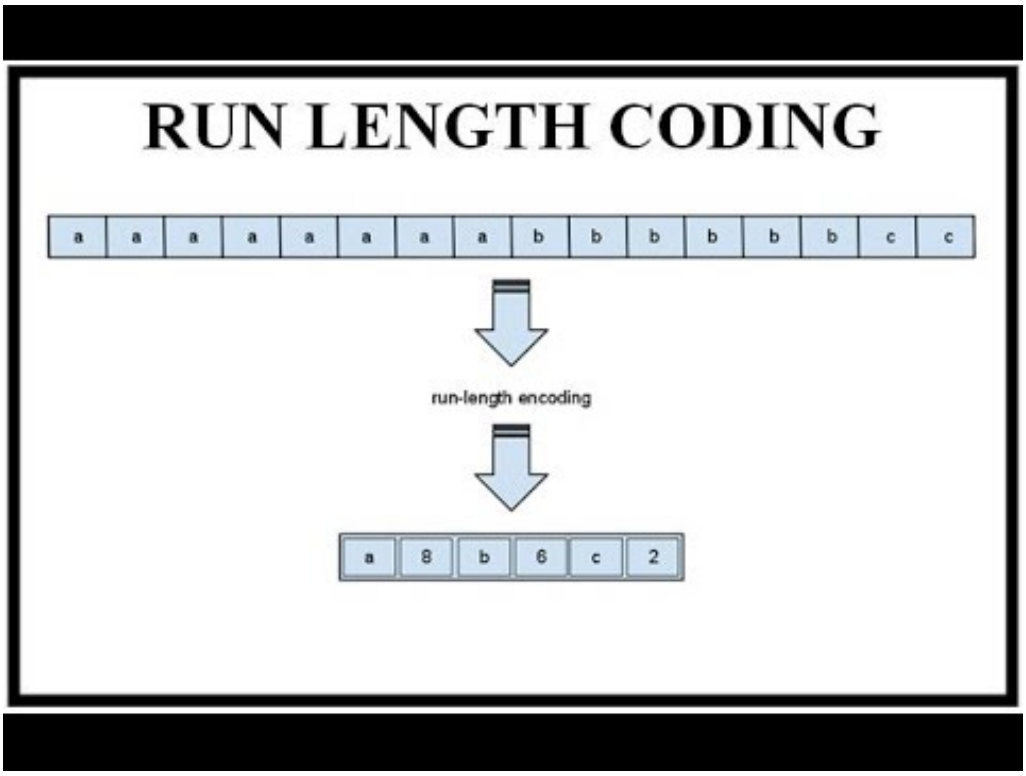
Neke od najpoznatijih algoritama kompresije koje kolonski orijentisan model koristi i koji će biti opisani u nastavku jesu: enkodiranje zasnovano na rečniku, enkodiranje po broju ponavljanja i delta enkoding. Važno je napomenuti da relacioni modeli imaju svoje mehanizme kompresije podataka koji u ovom 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 na disku. 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.



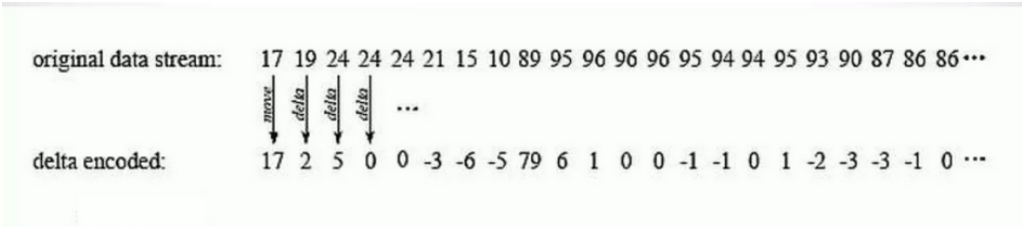
Slika 2.2: Enkodiranje zasnovano na rečniku

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.



Slika 2.3: Enkodiranje po broju ponavljanja

Delta enkoding, 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 enkoding

HBase

HBase je distribuirana kolonski orijentisana nerelaciona baza podataka nastala 2007. godine kao prototip *BigTable* baze koja je modelovana u okviru Google-ovog

članka 2006 [2].

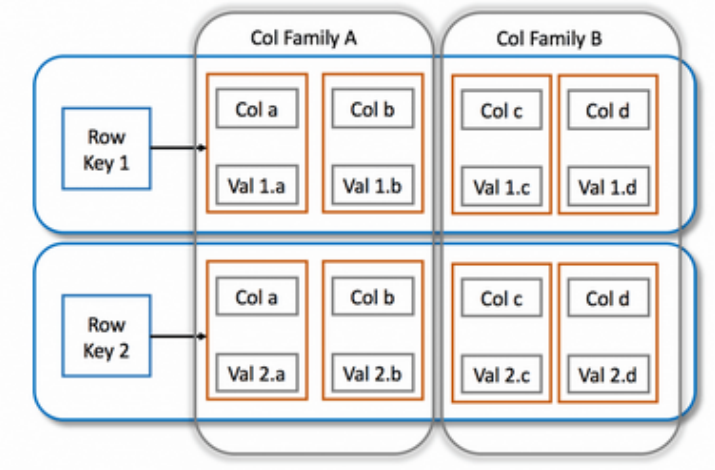
Model podataka koji HBase koristi podrazumeva da se svaka tabela sastoji iz familije kolona, a da svaka familija kolona sadrži određeni skup kolona. S obzirom da će se kolone koje pripadaju jednoj familiji sladištiti blizu na disku, cilj je da atributi, odnosno kolone koje su po prirodi slične pripadaju 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 [3]:

- 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.

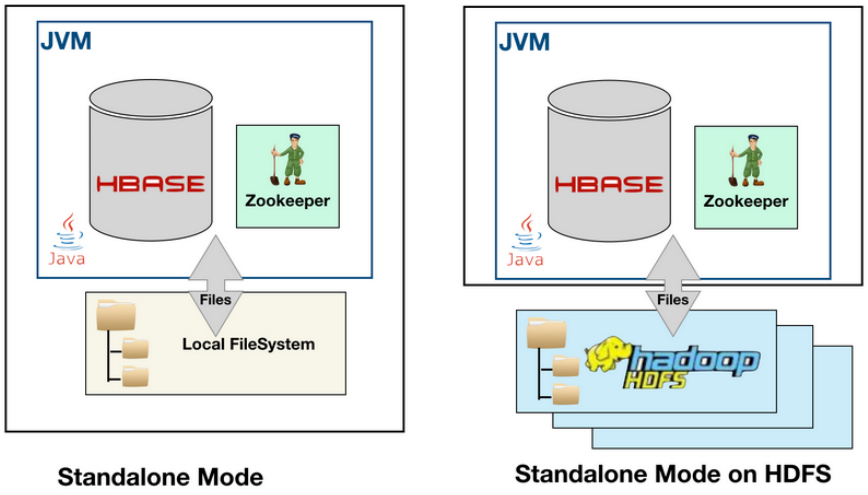
Master server sadrži servise koji rade sa metainformacijama o tabelama i familijama kolona. Uloga Master servera jeste da zahtev za podacima za konkretan ključ reda može da delegira na odgovarajući region server. Master server u te svrhe koristi *Zookeeper* [2] servis. Po potrebi master server takođe 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.

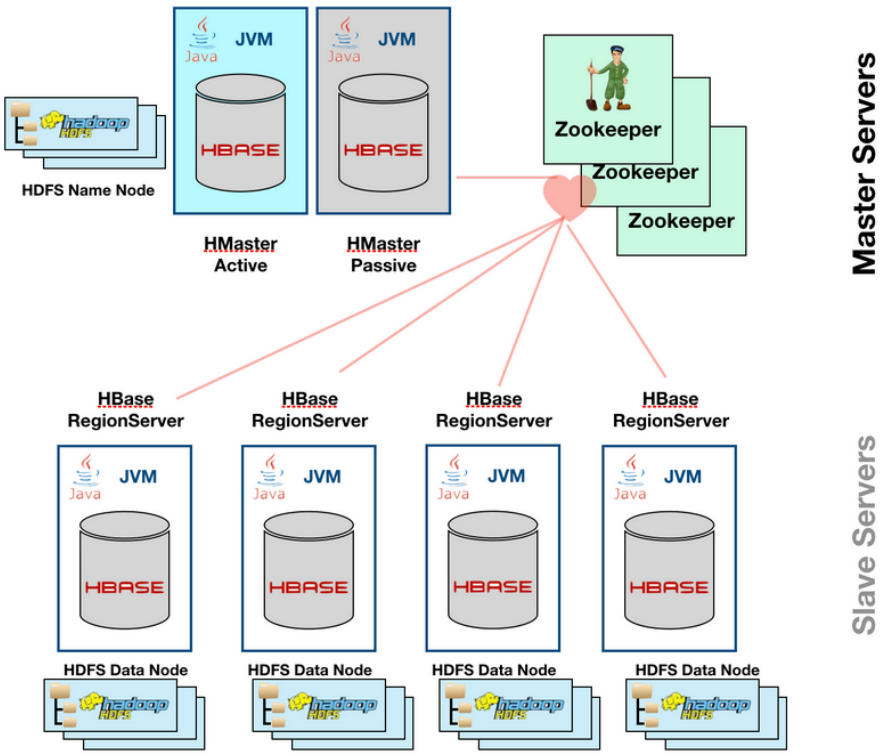
HBase dozvoljava dva režima: samostalan i distribuirani režim.

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

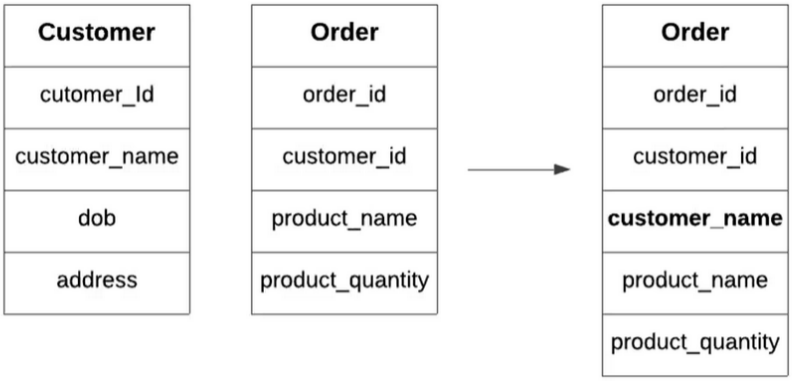
U relacionim modelima često se radi na izbegavanju *redudantnosti* u podacima. Redudantni podaci zauzimaju višak prostora na disku i otežavaju kasnije održavanje sistema. Kako bi se izbegla redudantnost postoje postupci koji nam pomažu da organizujemo podatke tako da redudantnost umanjimo. Proces izmene 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.

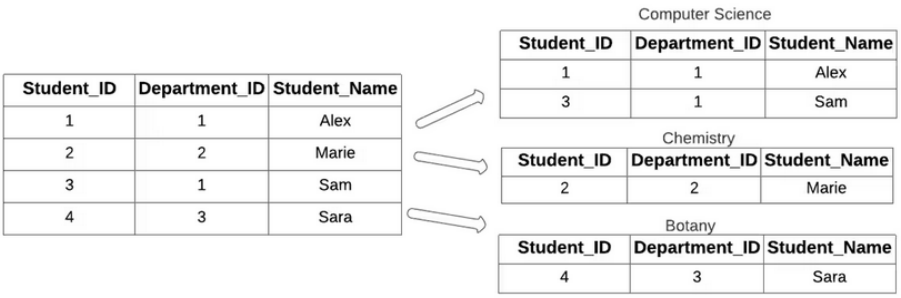
Neke od tehnika denormalizacije su: spajanje kolone, horizontalna podela tabele, vertikalna podela tabele i uvođenje izvedene kolone.

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.

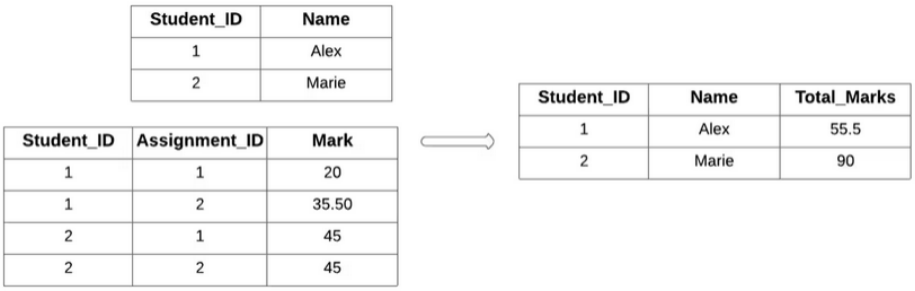


Slika 2.9: Horizontalna podela tabele

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

Slika 2.10: Vertikalna podela tabele

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 2.11: Uvođenje izvedene kolone

ACID i BASE

Transakcija je logička jedinica posla pri radu sa podacima [5]. 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.

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 .

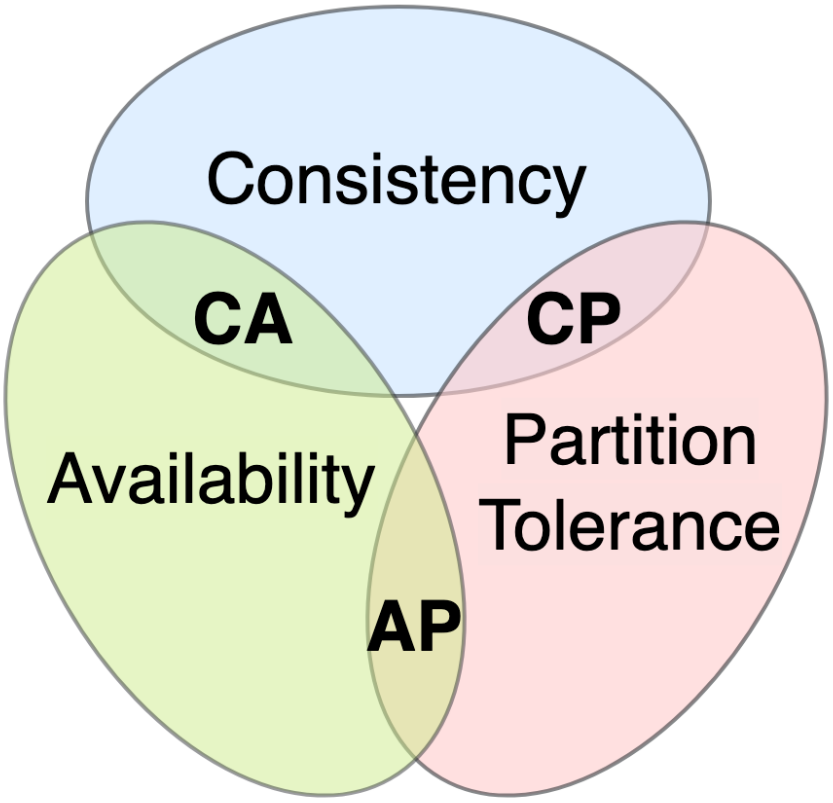
Raspoloživost je svojstvo koje obezbeđuje da će baza uvek vratiti neki odgovor.

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

CAP teorema koju je formulisao Eric Brewer [1], 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.

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



Slika 2.12: CAP teorema

Glava 3

Analiza slučajeva upotrebe

3.1 Opis i sadržaj eksperimenta

Analiza i upoređivanje slučajeva upotrebe biće realizovani na osnovu teorijskih i praktičnih izvora i istraživanja. Svaki primer će biti praćen eksperimentom koji će se sastojati od izvršavanja različitih vrsta postupaka.

U radu će biti analizirani sledeći slučajevi upotrebe:

1. Onlajn transakciono procesiranje (OLTP)
2. Onlajn analiticko procesiranje (OLAP)
3. Primena u distribuiranom okruženju

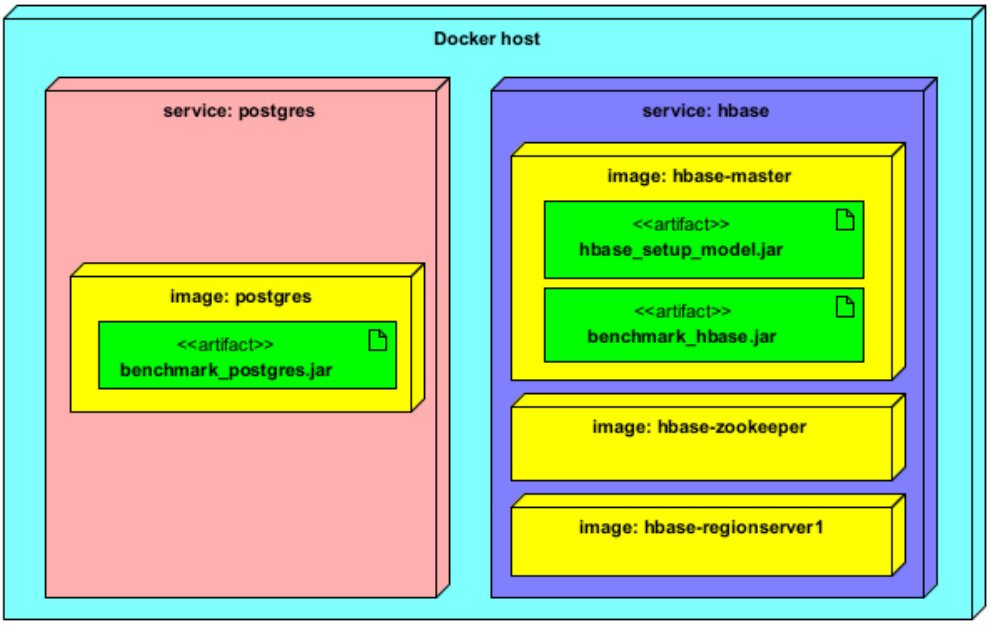
Svakom slučaju upotrebe u okviru pripreme eksperimenata dodeljen je model podataka nad kojim se izvršavaju testovi.

Pored definisanja konteksta, analiza slučaja upotrebe sadržaće opis implementacije testova nad predstavnicima, uputstvo za pripremu okruženja kao i analizu rezultata.

Analiza rezultata eksperimenta će pored prikaza rezultata samih merenja, obuhvatiti i upoređivanje složenosti realizacije konkretnog slučaja upotrebe, kao i eventualna unapređenja i primedbe koje treba imati u vidu kada se radi sa datim tehnologijama.

Platforma testiranja

Za predstavnike baza podataka čiji se modeli upoređuju i analiziraju izabrani su HBase i Postgres [1]. Kao okruženje za izvršavanje eksperimenata korišćen je docker.Oba predstavnika biće pokrenuta u okviru nezavisnih kontejnera na jednom računaru sa instaliranim docker-om. Svaki test predstavlja jedan java program koji se izvršava na konkretnom kontejneru, kako bi se smanjila potreba za evenutalnim saobraćajem kroz mrežu, a time i stekla što objektivnija slika na osnovu rezultata merenja. Dijagram sa opisom strukture platforme dat je na slici 3.1.



Slika 3.1: 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.

3.2 Primena u online transakcionom procesiranju (OLTP)

Onlajn transakciono procesiranje obuhvata kratke, učestale invazivne operacije na relativno malom skupu podataka. Da bismo simulirali ovakvo okruženje korišćen je primer onlajn transakcija i prenosa sredstava sa jednog računa na drugi. Parametri testa će biti broj biznis transakcija koje treba izvršiti, kao i broj klijenata, odnoso konekcija ka bazi, koji će paralelno obavljati svoj deo posla u okviru ovog testa. Jedna biznis transakcija sastoji se iz kreiranja transakcije, izvršavanja transakcije, odnosno prenos sredstava sa jednog računa na drugi, i na kraju provera statusa transakcije.

Opis modela

Model koji ćemo koristiti sastojaće se iz 4 tabele:

- **fxrates**: Sadrži informacije o kursu valutnih parova
- **fxuser**: Podaci o korisnicima
- **fxaccount**: Podaci o računima korisnika.
- **fxtransaction**: Transakcije prenosa 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';
```

Implementacija testa

Implementacija testa podrazumeva da ćemo imati dva parametra pri pokretanju: broj biznis transakcija koje će biti obrađene i broj klijenata koji će paralelno izvršavati svoj deo posla. Svakom klijentu će biti dodeljen određeni broj biznis

transakcija koje treba da obradi. Svaka biznis transakcija sastoji se iz kreiranja transakcije (**createFXTransaction**), izvršavanja uplate (**executePayment**), provere statusa transakcije (**checkTransactionStatus**).

CreateFXTransaction prvo pročita stanje naloga sa kojeg treba preneti sredstva, nakon toga čita odgovarajući kurs, a ukoliko ima dovoljno sredstava na računu u tabelu sa transakcijama upisuje transakciju 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 stanje sa naloga koji učestvuju u transakciji, menja im balans u skladu sa transakcijom, nakon toga transakciji 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 odgovarajucu transakciju čita status.

CheckTransactionStatus

```
1 select status
2 from fxtransaction
3 where id = ?
```

Svaki klijent u zasebnoj niti obrađuje njemu dodeljen broj biznis transakcija, a svaka od njih se sastoji iz gore navedenih delova.

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               ExecutePaymentInfo executePaymentInfo =
17              oltpUtil.createFXTransaction(connection);
18              oltpUtil.executePayment(,executePaymentInfo);
19              oltpUtil.checkTransactionStatus(fxT);
20            }
21            endSignal.countDown();
22          } catch (Exception e) {
23              throw new IllegalStateException(e);
24          }
25     }
26 }
```

Priprema okruženja

S obzirom da se testovi izvršavaju na posebnim docker kontejnerima, potrebno je da napravimo docker slike i pokrenemo kontejnere koristeći docker alat. Pored toga, neophodno je da kompajliramo java testove koristeći maven i nakon toga prosledimo odgovarajućim kontejnerima.

prepareEnv.sh

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

rm -f ./hbase_setup_model.jar
rm -f ./benchmark_hbase.jar
rm -f ./benchmark_postgres.jar

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";
```

Za pokretanje docker kontejnera korišćen je docker-compose sa sledećim sadržajem:

docker-compose.yml

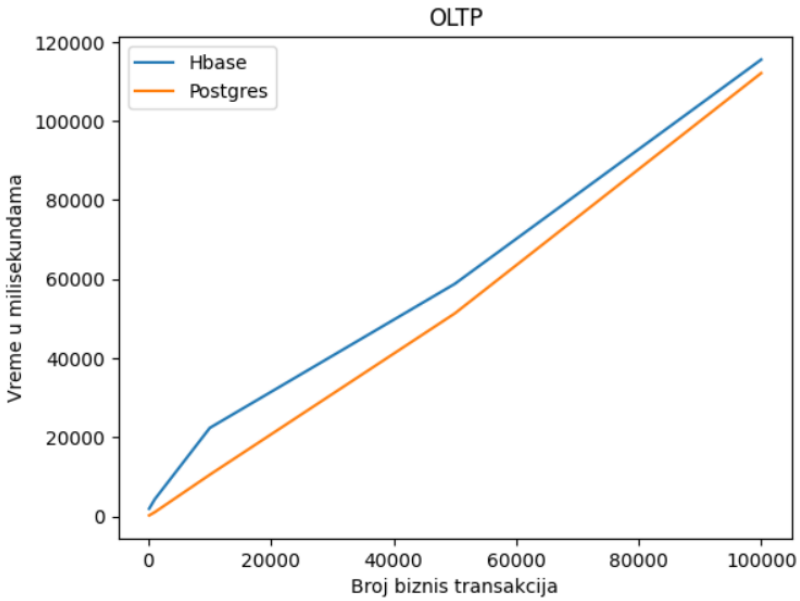
```
services:
  postgres:
    container_name: postgres
    ports:
      - "5433:5432"
    volumes:
      - ./setup_model.sql:/docker-entrypoint-initdb.d/create_script.sql
      - ./benchmark_postgres.jar:/benchmark_postgres.jar
    environment:
      - POSTGRES_PASSWORD=postgres
      - POSTGRES_USER=postgres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

```
13      build:
14        context: .
15        dockerfile: ./Dockerfile_postgres
16
17    hbase:
18      image: bde2020/hbase-standalone:1.0.0-hbase1.2.6
19      container_name: hbase
20      volumes:
21        - hbase_data:/hbase-data
22        - hbase_zookeeper_data:/zookeeper-data
23        - ./hbase_setup_model.jar:/hbase_setup_model.jar
24        - ./benchmark_hbase.jar:/benchmark_hbase.jar
25      ports:
26        - 16000:16000
27        - 16010:16010
28        - 16020:16020
29        - 16030:16030
30        - 2888:2888
31        - 3888:3888
32        - 2181:2181
33
34  volumes:
35    hbase_data:
36    hbase_zookeeper_data:
```

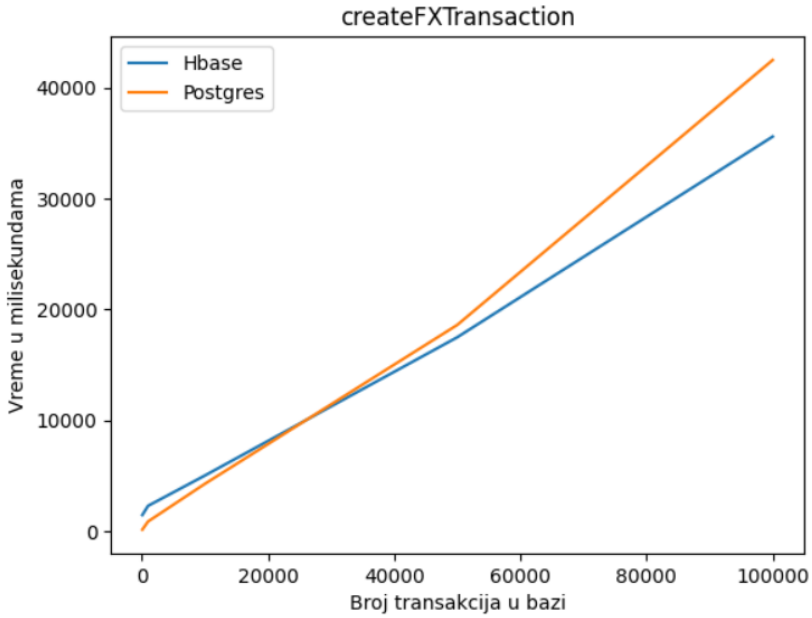
Analiza rezultata

Kada se radi o OLTP okruženju, važno je naglasiti da HBase ne garantuje konzistentost nad svim podacima kakvu nudi Postgres. HBase ipak garantuje neki vid konzistentosti, tj konzistentost nad jednim redom tabele. 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. Postgres sa druge strane kao ACID baza sama garantuje da izmene neke transakcije koje se rade nad nekim skupom podataka neće biti vidljive za druge klijente dok se ta transakcija ne komituje, pa samim tim i konzistento stanje podataka.

Merenje efikasnosti predstavnika u OLTP okruženju uključivalo je pokretanje testova u više iteracija. Svaki test je podrazumevao 5 klijenata koji paralelno obaljaju svoj deo posla, a pokrenuti su testovi za obradu 100,1 000, 10 000,50 000,100 000



Slika 3.2: Rezultati merenja u OLTP okruženju



Slika 3.3: Rezultati merenja createFXTransaction dela biznis transakcije

biznis transakcija, a rezultati merenja prikazani su na slici 3.2. Primećuje se da sa porastom podataka koji se nalaze u tabeli sa čijim se podacima radi, Postgres ima određena usporenja, relativno gledano u odnosu na HBase. Konkretno kada se pogleda vreme delova biznis transakcije odvojeno, primećuje se da HBase u

testovima sa 50 000 i 100 000 iteracija prednost stiče u createFXTransaction delu cele biznis transakcije, što se može videti na slici 3.3.

Tendencija merenja ukazuje da bi sa porastom broja podataka (na milione i desetine miliona redova) HBase davao bolje rezultate u odnosu na Postgres, međutim za testiranje takvog okruženja neophodno je koristiti računar koji je sposoban da sprovede tako zahtevan test.

3.3 Primena u online analitičkom procesiranju (OLAP)

OLAP okruženje sačinjeno je od velikog broja čitanja podataka i vrlo malog broja izmena podataka.. Upiti koji se koriste obično imaju parametre, visok nivo kompleksnosti i visok procenat podataka kojima pristupaju. Primer koji emo koristiti jeste uopšten primer održavanja trgovinskog lanca koji ima skup mušterija, proizvoda, dobavljača, narudžbina. Ovaj OLAP eksperiment će se sastojati iz dohvatanja izveštaja iz tabele koja sadrži stavke narudžbina. Taj izveštaj će sadržati rezultate agregiranih operacija nad kolonama, a parametri će biti status i datum slanja stavke.

Modeli podataka

Model koji ćemo koristiti sastojaće se iz sledećih tabela:

- **product**: Sadrži informacije o proizvodima. Predviđeno je da sadrži 3000000 redova.
- **supplier**: Podaci o dovaljačima. Predviđeno je da sadrži 1000000 redova.
- **productsupplier**: Vezna tabela između dobavljača i proizvoda. Predviđeno je da sadrži 5000000 redova.
- **customer**: Informacije o mušterijama. Predviđeno je da sadrži 1500000 redova.
- **order**: Informacije o narudžbinama. Predviđeno je da sadrži 1500000 redova.
- **orderitem**: Informacije o pojedinim stavkama narudžbine. Predviđeno je da sadrži 6000000 redova.

setup-postgres-model.sql

```
1
2
3  create table product (
4      id  integer not null primary key,
5      name varchar(50) not null,
6      brand varchar(50) not null,
7      type varchar(50) not null,
8      size integer not null,
9      container varchar(50) not null,
10     price varchar(50) not null,
11     comment varchar(50)
12 );
13
14 create table supplier (
15     id  integer primary key,
16     name varchar(50) not null,
17     address varchar(200) not null,
18     phone varchar(50) not null
19 );
20
21 create table productsupplier (
22     id integer not null primary key,
23     product integer not null,
24     supplier integer not null,
25     available integer not null,
26     supply_cost real not null,
27     comment varchar(200),
28     constraint fk_product
29     foreign key(product) references  postgresdb.product(id),
30     constraint fk_supplier
31     foreign key(supplier) references  postgresdb.supplier(id)
32
33 );
34
35 create table customer(
36     id integer primary key,
37     name varchar(50) not null,
38     address varchar(200) not null,
39     phone varchar(50) not null,
40     comment varchar(200)
41 );
```

```
42
43 create table order(
44     id integer primary key,
45     customer integer not null,
46     status varchar(20) not null,
47     total_price real not null,
48     entry_date date not null,
49     priority varchar(20) not null,
50     comment varchar(200),
51     constraint fk_customer
52     foreign key(customer) references  postgresdb.customer(id)
53 );
54
55 create table order_item(
56     order_id integer not null,
57     product integer not null,
58     supplier integer not null,
59     order_no integer not null,
60     quantity integer not null,
61     base_price real not null,
62     discount real not null,
63     tax real not null,
64     status varchar(20) not null,
65     ship_date date not null,
66     commit_date date not null,
67     comment varchar(200),
68     primary key(order_id,product,supplier),
69     constraint fk_order
70     foreign key(order_id) references  postgresdb.order(id),
71     constraint fk_product
72     foreign key(product) references  postgresdb.product(id),
73     constraint fk_supplier
74     foreign key(supplier) references  postgresdb.supplier(id)
75 );
```

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';
```

Implementacija testa

Test će se sastojati iz dve faze. Bulk upunjavanje tabela na osnovu csv fajlova, kao i izvršavanje složenog OLAP upita.

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";
```

BenchmarkExecutor.java

```
1
2     void bulkLoad(BenchmarkUtility util,BenchmarkOLAPUtility oUtil) {
3         long start = System.currentTimeMillis();
4         olapUtility.bulkLoad(benchmarkUtility.connect());
5         long end = System.currentTimeMillis();
6         System.out.println("Bulk load duration: "+(end-start));
7     }
```

OLAP upit sadrži informacije o agregiranim vrednostima kolona za određene vrednosti kolona. Upit će se izvršavati nad orderitem tabelom, a kao parametre uzimaće status i datum slanja stavke narudžbine.

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 = ?
```

Parametri eksperimenta će biti broj iteracija izvršavanja OLAP upita kao i broj klijenata koji će paralelno izvršavati svoj deo posla u vidu određenog broja izvršavanja OLAP upita.

BenchmarkSingleClientExecutor.java

```
1
2  public class BenchmarkSingleClientExecutor implements Runnable{
3
4      private final CountDownLatch endSignal;
5      private final BenchmarkOLAPUtility olapUtil;
6      private final int numOfT;
7      private final int start;
8
9      private final Object connection;
10
11     @Override
12     public void run() {
13         try {
14             for (int i = this.start; i < this.start+this.numOfT; i++) {
15                 olapUtil.executeOLAPQuery(connection);
16             }
17             endSignal.countDown();
18         } catch (Throwable e) {
19             throw new IllegalStateException(e);
20         }
21     }
22 }
```

Priprema okruženja

Priprema okruženja je većinom identična kao priprema okruženja za OLTP slučaj upotrebe. Dodatan korak jedino je kreiranje csv fajlova sa podacima koji treba da budu bulk učitani u odgovarajuće tabele Postgres-a i HBase-a.

prepareEnv.sh

```
#!/bin/bash
echo 'PREPARING ENVIRONMENT...';
rm -f ./olap_benchmark_postgres.jar
rm -f ./olap_benchmark_hbase.jar
rm -f ./productHB.csv;
rm -f ./supplierHB.csv;
rm -f ./productsupplierHB.csv;
rm -f ./customerHB.csv;
rm -f ./orderHB.csv;
rm -f ./orderitemHB.csv;
rm -f ./productPG.csv;
rm -f ./supplierPG.csv;
rm -f ./productsupplierPG.csv;
rm -f ./customerPG.csv;
rm -f ./orderPG.csv;
rm -f ./orderitemPG.csv;


echo 'PREPARING HBASE BENCHMARK JARS...';
export JAVA_HOME="$JAVA_8";
mvn -f olap_benchmark_hbase clean compile assembly:single;
mvn -f hbase_setup_olap_model clean compile assembly:single;
mvn -f hbase_bulk_load_setup clean compile assembly:single;

echo 'PREPARING HBASE BULK LOAD RESOURCES..';
java -jar ./hbase_bulk_load_setup.jar;


echo 'PREPARING POSTGRES BENCHMARK JARS...';
export JAVA_HOME="$JAVA_17";
mvn -f olap_benchmark_postgres clean compile assembly:single;
mvn -f postgres_bulk_load_setup clean compile assembly:single;

echo 'PREPARING POSTGRES BULK LOAD RESOURCES..';
java -jar postgres_bulk_load_setup.jar;

docker-compose -f docker-compose.yml up --build -d;
winpty docker exec -it hbase sh -c "java -jar setup_olap_model.jar";
```

Za pokretanje docker kontejnera korišćen je docker-compose sa sledećim sadržajem:

docker-compose.yml

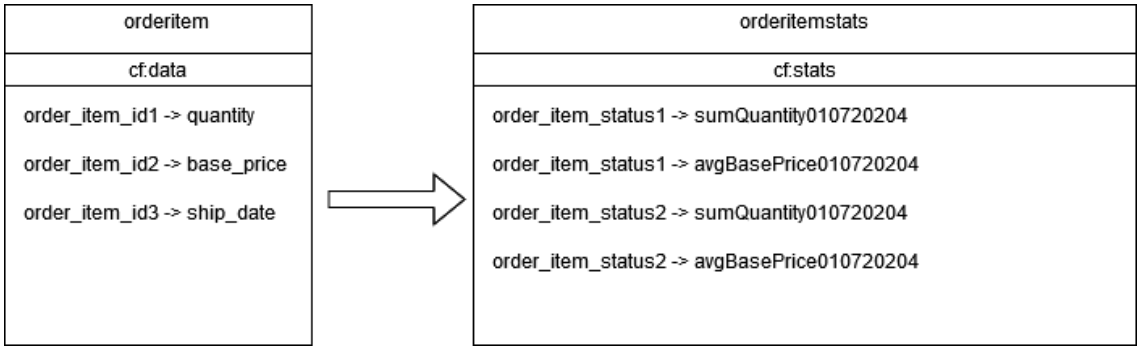
```
1
2  services:
3    postgres:
4      container_name: postgres
5      ports:
6        - "5433:5432"
7      volumes:
8        - ./setup_model.sql:/docker-entrypoint-initdb.d/create_script.sql
9        - ./benchmark_postgres.jar:/benchmark_postgres.jar
10     environment:
11       - POSTGRES_PASSWORD=postgres
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:
```

Analiza rezultata

Rezultati merenja u OLAP okruženju ,kada se koriste ravnopravni modeli pokazuju jasnu prednost Postgres-a u odnosu na HBase. Sa druge strane, ovakav pristup gde se čitaju svi redovi filtrirani po statusu i datumu, pa se nad vrednostima izvršavaju odgovarajuće aritmetičke operacije, može se nazvati *„naivnim"* u slučaju HBase-a.

Jednostavna denormalizacija modela, slika 3.4 može doneti dramatična poboljšanja u performansama HBase-a čak i u odnosu na Postgres.

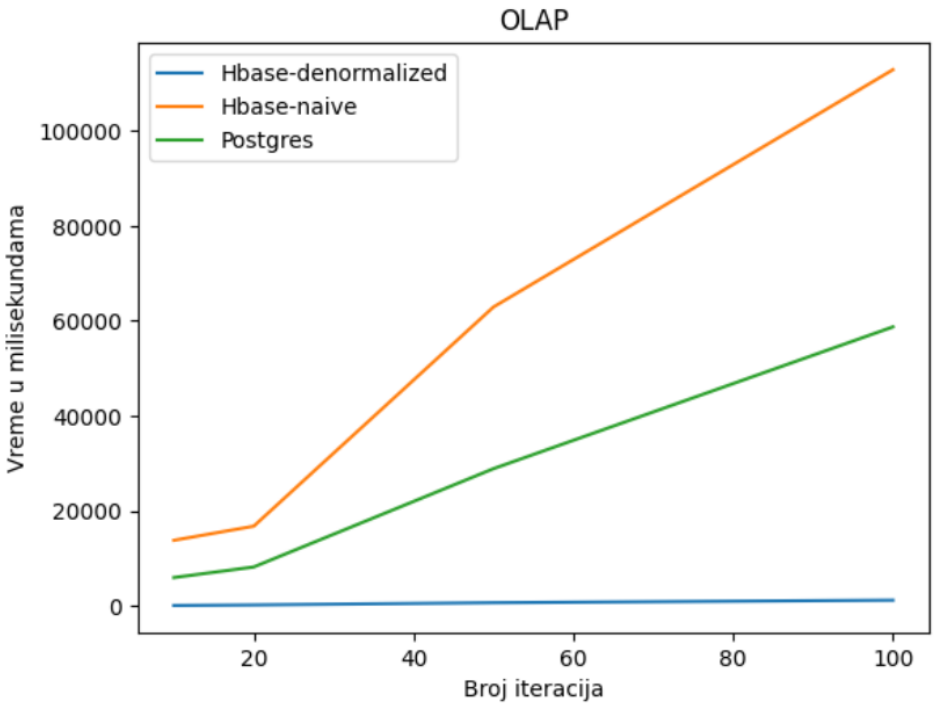


Slika 3.4: Denormalizacija na primeru tabele orderitem

Izdvajanje tabele **orderitemstats**, koja će za ključ imati status stavke, a kao kolone imati rezultate već ranije izračunatih agregatnih vrednosti nad kolonama za odgovarajuče dane, dovodi do toga da čitanje izveštaja sada predstavlja jednostavno čitanje reda po ključu i skupa odgovarajućih kolona koje su nam potrebne za izveštaj.

Uticaj denormalizacije modela na performanse mogu se videti na slici 3.5.

Ovakav pristup prilikom kreiranja modela, omogućila nam je fleksibilnost strukture koju HBase nudi, kao i nepostojanje limita broja kolona.



Slika 3.5: Uporedna analiza performansi postgresa i hbase-a

3.4 Primena u distribuiranom okruženju

Distribuirano okruženje odlikuje

Glava 4

Zaključak

Bibliografija

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

[2] 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`.

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

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

[5] 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.