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, vandredni profesor
Univerzitet u Beogradu, Matematički fakultet

Članovi komisije:

dr Ana Anić, vanredni profesor
University of Disneyland, Nedođija

dr Laza Lazić, 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, koncepata jednog sistema kao i instrukcija u formalizovanom stanju spremnom za dalju interakciju, interpretaciju ili obradu od strane korisnika ili mašine. Iako kroz svoju istoriju računatstvo 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 menjali, ili su se slabo menjali i proširivali. 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, izdvojio bih tri glavne faze: vreme pre relacionih sistema, vreme neprikosnovene vladavine relacionih sistema i nastanak alternativa relacionim sistemima pod grupnim nazivom NoSQL.

Do 70ih i nastanka relacionih sistema za upravljanje podacima, rukovanje podacima izvodilo se kroz pisanje i čitanje sa fajl sistema 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.

S obzirom da je ovaj vid rada sa podacima imao mnogbrojne mane, među kojima je jedan od glavnih bio težak 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 istraživanja i sopstvenih teorija o organizaciji podataka. Kao dokaz da je

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

njegov model moguće implementirati pokrenut je System R, čiji je rezultat bio 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 koristecći relacioni model.

XXI vek doneo je sa sobom ubrzanu digitalizaciju, povećanu dostpunost interneta, samim tim pojavila se potreba za obradom veće količine podataka. Sve ovo je pokazalo pojedine slabosti dosadašnjih sistema zasnovane na relacionim modelima, koji nisu mogli u svim segmentima da odgovore na zahteve modernog doba. Ovi problemi obično su poznati pod gruonim imenom „problemi velikih podataka" (BigData problems). Došlo je do pojave niza novih modela i principa za čuvanje podataka, a svi pod grupnim imenom „NoSQL" (ili nerelacione) baze podataka. 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 , jesu glavni aktuelni 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 defakto standard i dan danas, te je zaočekivati postojanje doze skepse pri korišćenju tehnologija nastalih u ovoj fazi.

Kao važna grupa nerelacionih izdvajaju se kolonski-orijentisane baze podataka. One su uvele tada nekonvencionalne koncepte čuvanja podataka po kolonama. Dakle fizički na disku skladištile su se vrednosti jedne kolone jedna do druge, sa referencom na red kojem pripadaju. To sa sobom vuče razne mogućnosti za optimizaciju ali i novih pristupa modelovanja i organizacije podataka. Ovakav način skladištenja ispitavan je još davnih 70ih 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 materijala koji se bave analizom slučajeva upotrebe tih tehnologija. Pored teorijske analize koja se može pronaći u relevantnim javnim dokumnetacijama 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 opsia navedenih tehnologija kao i opisa konkretnih predstavnika baza podataka koji će biti kori-

šćeni. Na osnovu teorijskih izbora i istraživanja biće analizirani različiti slučajevi upotrebe.

Glava 2

Modeli za upravljanje podacima

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. Kao fundamentalna ideja iza relacionog modela stoji tabelarni prikaz podataka, sto uvecava njegovu intuitivnost. Da bi jedna tabela 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 deluje intuitivno (usled istorijskog uticaja koji je relacioni model ostavio na vizualizaciju organizacije podataka) ona je neophodna za definisanje složenijih pojmova.

Koncept ključa relacionog modela

Usled jedinstvenosti svake vrste relacije, jasno je da mora postojati skup kolona za koji važi da nikoja dva reda te relaciju nemaju identične vrednosti za svaku kolonu iz tog skupa. Takav skup se naziva *superključ* relacije. Minimalan superključ

naziva se *ključ kandidat*. Svaka relacija može imati više ključeva kandidata, ali samo jedan od njih je *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. Primarni i strani ključ igraju veliku ulogu u očuvanju integriteta baze podataka o čemu ce biti reči u nastavku.

Integritet relacionog modela

Integritet relacionog modela predstavlja „istinitost" podataka koji se čuvaju u bazi, što je njena i najveća vrednost. Čuvanje integriteta baze posebno je važno prilikom invazivnih operacija kao što su dodavanje reda, izmena reda ili brisanje reda u tabeli. Postoji više vrsta integriteta u relacionom modelu: *integritet entiteta*, *integritet domena*, *integritet neposojece vrednosti* i *referencijalni integritet*.

Integritet entiteta kaže da svaka vrsta jedne relacije predstavlja jedan entitet i da kao takva ne može u okviru primarnog ključa, koji taj entitet identifikuje, imati nedefinisanu ili nepostojeću vrednost.

Integritet domena nameće shemu po kojoj svaka kolona može uzimati vrednost iz unapred dodeljenih skupova vrednosti.

Integritet nepostojece vrednosti govori o eventulnim kolonama cije vrednosti ne mogu kao vrednost imatu nepostojecu vrednost kako se ne bi narušila uspostavljena biznis logika.

Referencijalni integritet nalaže da se svaki strani kljuc jedne tabele, ukoliko je definsan, mora poklapati sa nekim od primarnih ključeva uparene relacije.

PostgreSQL

PostgreSQL je objektno-relacioni sistem za upravljanje bazama podatak nastao kao potomak POSTGRES-a, proizvoda koji je nastao, a kasnije i 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, agregagtnih 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

[1] Susret sa Big Data problemima dovelo je do potreba za tabelama koje imaju ogroman broj kolona, i ogroman broj redova u okviru tih tabela. Jasno je da nam je za potrebe različitih analitika potreban različit skup kolona. 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.

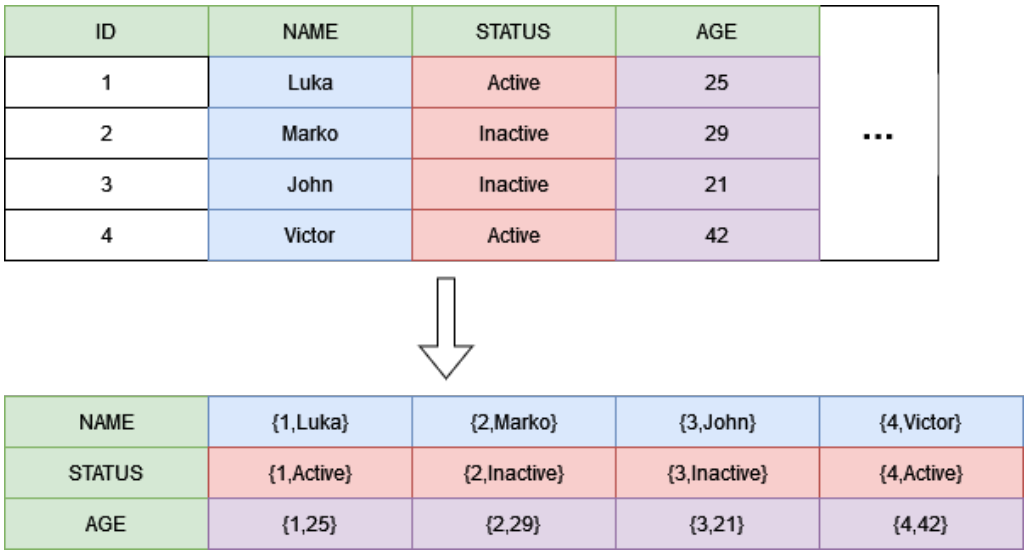
Kao što mu ime govori kolonski orijentisan model podatke na disku skladišti po kolonama a ne po redovima kao što je to slučaj kod relacionh modela. U prevodu, sve vrednosti kolone svih redova skladište se jedna do druge, a na konkretnu vrednosnu ćeliju referiše se pomoću ključa konkretnog reda kao i 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 sloga, već je dovoljno da znamo konkretan ključ tog reda kao i imena kolona čije vrednosti želimo da pročitamo.

Ovakav vid skladištenja 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.

Nedostajanje unapred definisanog skupa kolona, ukida smisao čuvanja nepostojeće vrednosti (null) s obzirom da ukoliko red nema vrednost neke kolone, nema potrebe za upisivanjem bilo koje vrednosi te kolone za taj red.

Kolonski orijentisan model kao i većina ostalih nerelacionih modela, nudi fleksibilnost sheme. To kao posledicu ima da eventualna promena strukture podataka neće bitno uticati na unapred definisanu shemu, kao ni iziskivati dodatnu migraciju podataka, kao sto bi to bio slucaj kod relacionog modela. Osim toga fleksibilnost sheme se ogleda i u tome sto je broj kolona jednog vektora neogranicen, sto daje dosta prostora za eksperimentisanje sa dizajnom baze. Primer toga kako ovakvo svojstvo modela može doprineti dostizanju prednosti pri analitičkom sistemu možete videti na slici SLIKA 1.



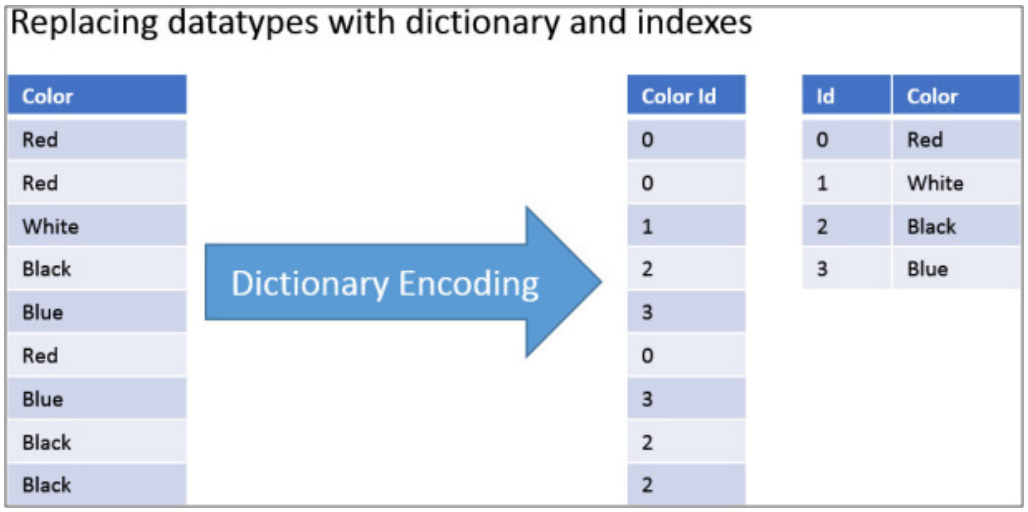
Slika 2.1: Kolonski orijentisan format

Popularni algoritmi kompresije kolonski orijentisanog modela

[2]

Enkodiranje zasnovano na rečniku

Enkodiranje zasnovano na rečniku (Dictionary based encoding) jeste tehnika kompresije podataka koja se može primeniti na vrednosti jedne kolone ili skupa kolona. Najefikasnija je nad kolonama koje imaju mali skup mogućih vrednosti. Rade tako što se u memoriji sačuvaju sve moguće vrednosti te kolone i svakoj od njih se dodeli ključ. Veličina ključa je direktno zavisna od kardinalnosti skupa vrednosti koje se mapiraju. Svaki unos ili izmena vrednosti kolone u konsultaciji sa postojećim rečnikom radi enkodiranje pristigle vrednosti, a svako dohvatanje vrednosti radi dekodiranje sačuvane vrednosti. Ovim se izbegava ponavljanje velikih podataka tako smanjujući potrebnu memoriju na disku. Uglavnom su pogodne primene nad kolonama sa statičkim i opisnim podacima koji se ponavljaju.

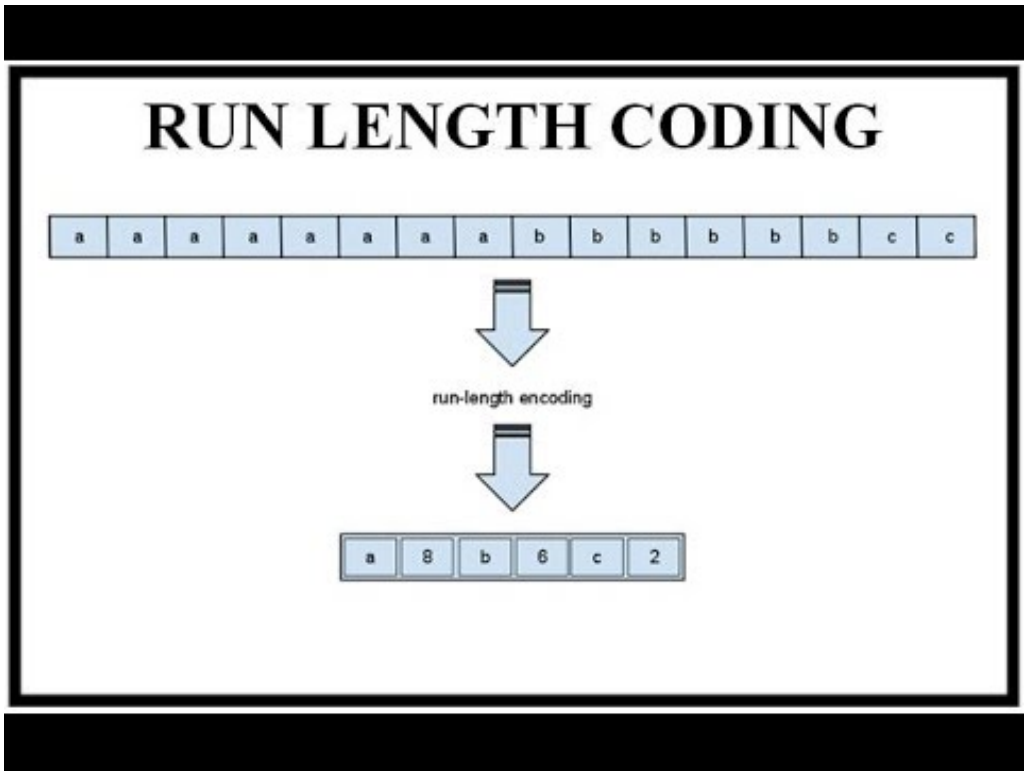


Slika 2.2: Enkodiranje zasnovano na recniku

Enkodiranje po broju ponavljanja

Enkodiranje po broju ponavljanja (Run Length Encoding) je jednostavan mehanizam za kompresiju podataka pogodan kompresiju kolona kod koje se vrednosti ponavljaju. Funkcioniše tako što kada se naiđe na vrednost koja se ponavlja, ne

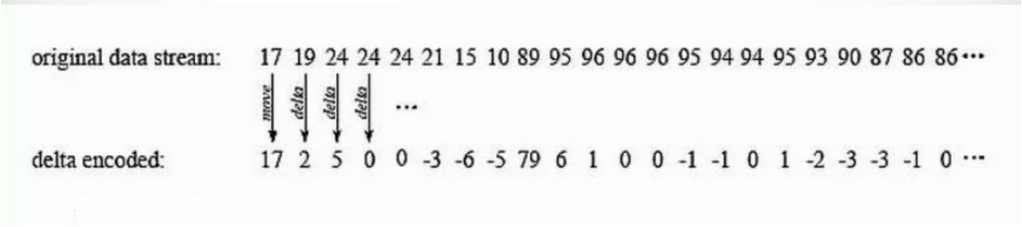
skladišti duplikate već sačuva tu vrednost jednom a dodatno kao meta informaciju prosledi koliko puta se ta vrednost ponavlja. Takav vid optimizacije najkorisnije je očigledno kada su vrednosti sortirane, a s obzirom da su vrednosti kolona u kolonski orijentisanim bazama jedna do druge to otvara prostor za ovaj vid kompresije podataka kako bi se umanjilo zauzeće prostora.



Slika 2.3: Enkodiranje po broju ponavljanja

Delta enkoding

Delta enkoding je mehanizam za optimizaciju prostora baze podataka koji se zasniva na čuvanju razlike između objekata a ne celih vrednosti. Primera za upotrebu ima dosta a jedan od najčešćih je slučaj datumskih kolona, gde će nam referentna vrednost biti neki konkretan datum, a vrednosti ostalih kolona će biti čuvane kao razlika u odnosu na njega. Kao što je rečeno pogodna je za datumske vrednosti ali postoje i drugi tipovi, kao što su numerički, kod kojih ovaj vid kompresije može doneti unapređenja u vidu slobodnog prostora na disku.



Slika 2.4: Delta enkoding

HBase

HBase je distribuirana kolonski orijentisana nerelaciona baza podataka pisana u javi. Nastala je 2007 kao prototip BigTable baze koja je modelovana u okviru Google-ovog članka 2006 [3].

Model podataka koji HBase koristi podrazumeva da se svaka tabela sastoji iz familije kolona, a da svaka familija kolona sadrži određeni broj atributa nekog reda. Cilj je da atributi, odnosno kolone koje su po prirodi slične pripadaju istoj familiji kolona, kako bi se nad njima mogli primeniti algoritmi kompresije, s obizrom da će se kolone koje pripadaju jednog familije sladištiti blizu na disku. Hbase nudi fleksibilnost sheme, pa s toga da bismo neki podatak skladištili ne moramo unapred da definišemo skup kolona koji pripada nekoj tabeli , ali moramo definisati skup familija kolona te tabele kako bi bilo moguće odrediti pogodnu lokaciju skladištenja podatka.

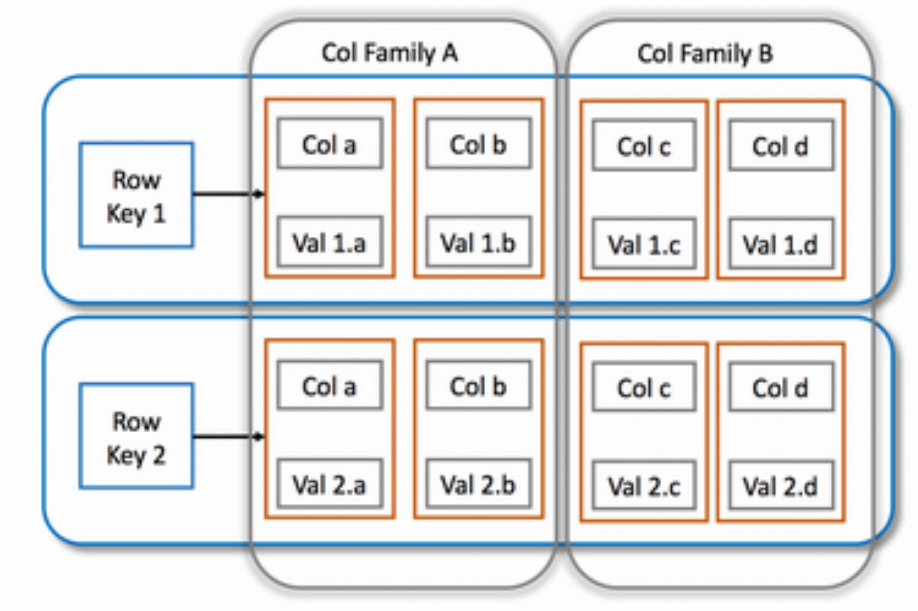
HBase ne podržava postojanje indeksa, međutim podrazumevano ponašanje je da su ključevi svih redova sortirani rastuće, tako da se koristi binarna pretraga za pretragu vredosnti po ključu reda kojem pripada. Ovo daje na važnosti strategiji pro dizajniranju ključa. Poželjno je da svaka pretraga ide direktno preko ključa ili njegovog prefiksa.

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

HBase nije ACID baza podataka, ali garantuje konzistentnost u radu sa jednim redom.

Arhitektura HBase klastera sastoji se iz dve glavne komponente: master server i region server. Ove komponente je najbolje opisati kroz interfejse koje oni implementiraju.

Region server implementira HRegionInterface, odnosno implementira servise koji se bave operacijama nad podacima i održavanjem i upravljanjem regiona.



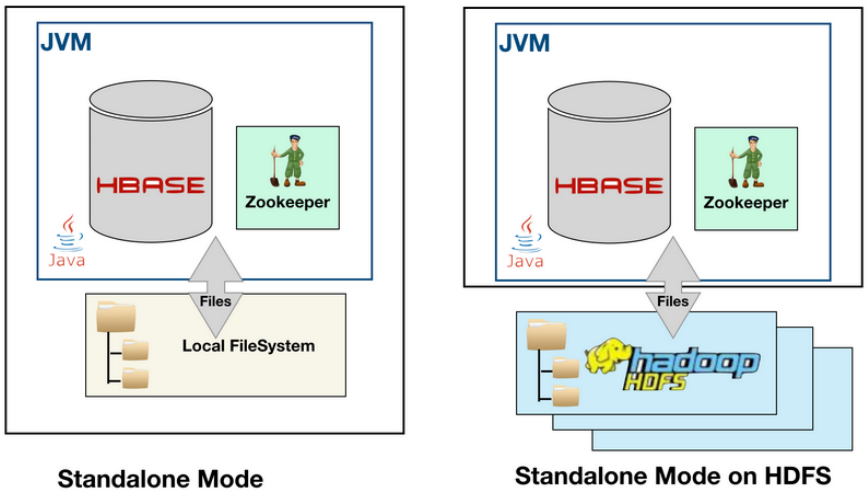
Slika 2.5: Hbase model

Jedan region čine podaci za redove u nekom rasponu vrednosti ključeva. Region server kreira i dohvata HFile fajlove koji se čuvaju na disku. HBase dozvoljava dva režima: samostalan i distribuirani režim.

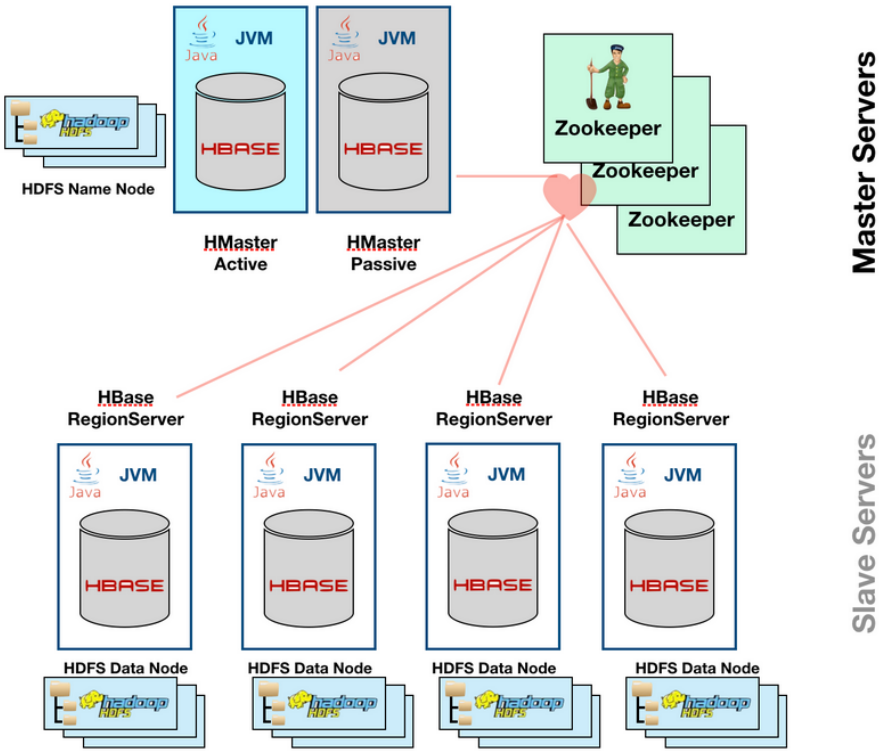
Samostalan režim HFile fajlove može čuvati na lokalnom fajl sistemu i korišćenje HDFS-a je opciono.

U distribuiranom režimu sa druge strane neophodno je korišćenje HDFS-a za skladištenje. Region server možemo zamisliti kao indeks za dohvatanje HFile-ova sa sa diska. Svaka izmena koja treba da se izvrši na disku prvo se upisuje u WAL (Write ahead log), a nova izmenjena vrednost se upisuje u MemStore fajl. Kada se MemStore fajl upuni tek tada se izmene flush-uju u jedan HFile koji dalje ide na HDFS.

HMaster implementira HMasterInterface koji sadrži servise koji rade sa metainformacijama o tabelama, familijama kolona, kao i regiona. Uloga Mastera je da za svaki konkretan row id može da odredi koji region odnostno region server treba da bude pročitan kako bi se izvršila odgovarajuća operacija. Za postizanje navedene funkcionalnosti HMaster koristi Zookeeper [3] servis. Pored toga master servis ima pozadinske procese koji regulišu rad load balansera i sadržaj hbase:meta tabele.



Slika 2.6: HBase standalone



Slika 2.7: HBase distributed

2.3 Glavne razlike između relacionog i kolonski-orijentisanog modela

Normalizacija i denormalizacija

Da bi se stanje u bazi olakšalo čuvanje podataka konzitentnim 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 postoji jasno definisani postupci koji nam pomažu da organizujemo podatke tako da redudantnost umanjimo. Proces unapređivanja logičkog dizajna baze tako da rešava problem redundantosti podataka, ali ne po cenu očuvanja integriteta, naziva se normalizacija. Teorija o normalizaciji se zasniva nad konceptima normalnih formi iz matematičke logike. U zavisnosti od toga koja pravila zadovoljava određena relacija, dodeljuje joj se normana forma. Trenutno postoji 5 defnisanih normalnih formi.

1. Prva normalna forma (1NF)
    - Svaka relacija mora imati primarni ključ koja jedinstveno određuje svaku vrste.
    - Svaka kolona mora saržati atomičnu (nedeljivu) vrednost.
    - Sve vrednosti jedne kolone pripadaju nekom zajedničkom skupu.
2. Druga normalna forma (2NF)
    - Relacija mora biti 1NF
    - Sve vrednsti kolone koje ne pripadaju superključu relacije, direktno su određene celim primarnim ključem .
3. Treća normalna forma (3NF)
    - Relacija mora biti 2NF
    - Nijedna kolona van ključa kandidata nije jedinstveno određena drugim kolonama koje ne pripadaju ključu kandidatu.
4. Boyce-Codd normalna forma (BCNF)
    - Relacija mora biti 3NF.

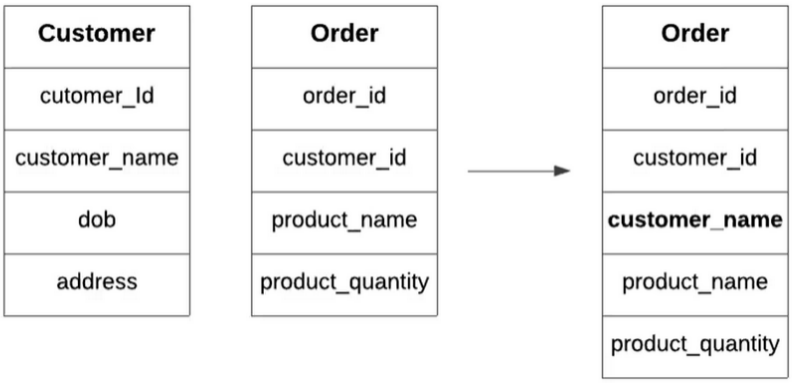
- Svaka kolona koja ne pripada ključu mora biti jednistveno određena vrednostima superključa relacije.

5. Četvrta normalna forma (4NF)
   - Relacija mora biti BCNF.
   - Regulisanje zavisnostni među kolonama tako da nema ponavljanja podataka.
6. Peta normalna forma (5NF)
   - Relacija mora biti 4NF.
   - Reguliše zavisnosti između relacija tako da se izbegne potreba za komplikovanim upitima za dohvatanje podataka.

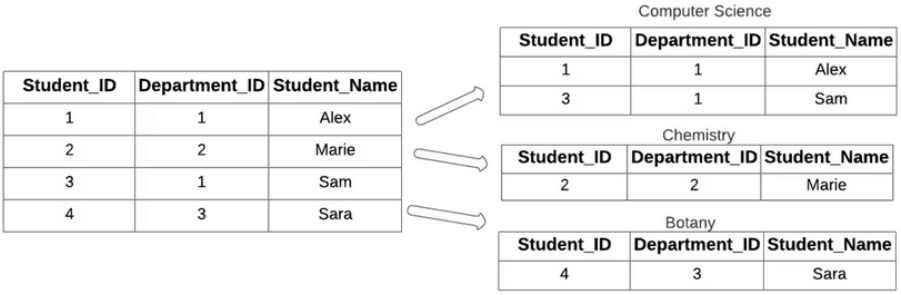
Modeli koji su bili podloženi normalizaciji obično raspolažu velikim brojem referencijalnih ključeva u tabelama koji predstavlaju referencu ka originalnim podacima iz matične tabele u kojoj se nalaze. Ukoliko želimo da pristupimo celom podatku iz neke tabele koja referiše na njega, podrazumeva se da ćemo pročitati celu tabelu na koju ta referenca pokazuje, što povećava broj operacija čitanja sa diska samim timmože uticati na performanse.

Denormalizacija je strategija kod modela gde 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 dodavanje redundantne kolone, horizontalna podelea tabele, vertikalna podela tabele, uvođenje izvedene kolone. Pre join kolone je dodavanje kolone tabeli čija se referenca na vrednost te kolone često koristi. Horizontalno deljenje tabele 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 podrazumeva da se tabela podeli grupisanjem kolona koje se često čitaju zajedno. Uvođenje izvedene kolone predstavlja dodavanje kolone koja predstavlja vrednost neke agregatne funkcije, time se izbegava da se pri čitanju ta operacija izvršava, već se radi prosto čitanje.



Slika 2.8: Pre join kolone



Slika 2.9: Horizontalna podela tabele

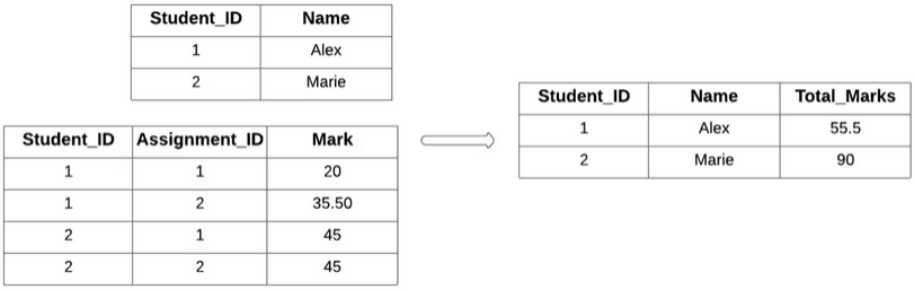
Slika 2.10: Vertikalna podela tabele

ACID i BASE

ACID (Atomicity, Consistency, Isolation, Durability) svojstva služe kao garancija tačnosti i konzistetnosti podataka prilikom konkurentnom pristupu.

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

Durability garantuje da će kompletirana transakcija u slučaju prekida rada sistema pre nego što su izmene reflektovane na disk, biti upamćena i izvršena nakon



Slika 2.11: Uvođenje izvedene kolone

restarta sistema. Svaka izmena se upisjue u log fajl pre nego što je reflektovana na disk, kako bi se operacije mogle poništiti u slučaju poništavanja transakcije.

Consistency se čuva od strane korisnika. Bitno je da korisnik koji pokreće transakciju vodi računa o tome da stanje podataka ostane u konzistentom stanju.

Isolation svojstvo nalaže da se transakcije međusobno izolovane tako da izvršavanje jedne transakcije ne može uticati na izvršavanje druge. Ovo se obezbeđuje pomoću scheduler-a od strane samog sistema za upravljanje bazom podataka.

ACID svojstva obično su karakteristika relacionog modela.

BASE sa druge strane je skup svojstava koje je definisao Eric Brewer , a koja su nastala usled zelje da se formalizuju svojstva koja u Big Data svetu garantuju da je baza pogodna za horizontalno skaliranje a ujedno daje vid konzistentnosti koji je neophodan.

Suštinska rasoloživost svojstvo kaze da ukoliko imamo klaster sa vise pojedinačnih skladišta baze, problem sa jednim od njih nece spreciti da ostala skladišta procesiraju zahteve i salju odgovore.

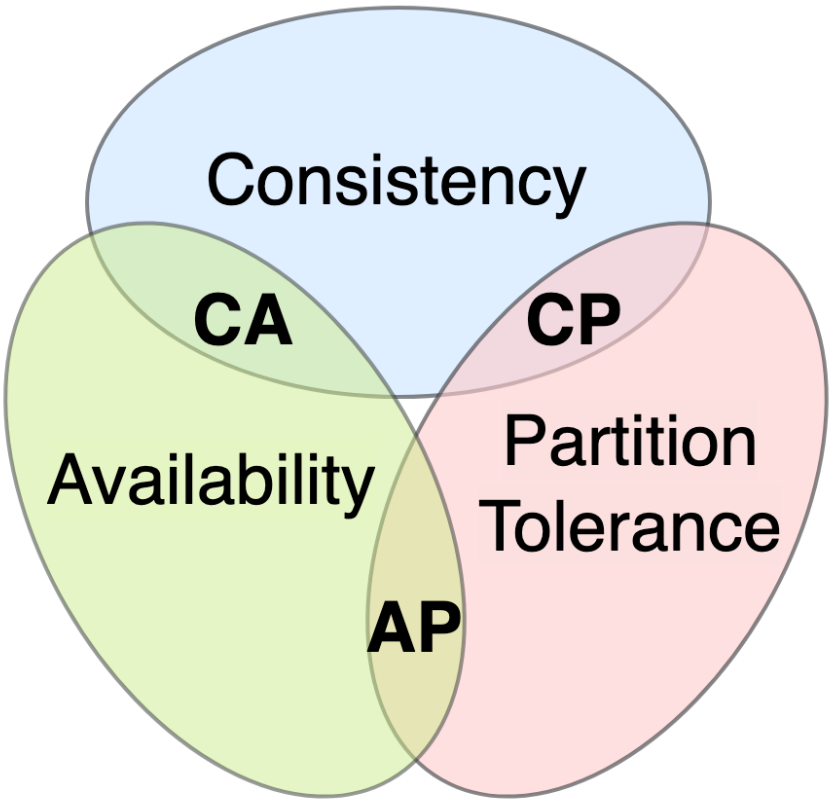
Trenutno nekonzistento stanje - svojstvo kaze da se stanje podataka moze menjati, cak i u trenucima kada nema spoljnih komunikacija sa bazom podataka.

Konvergentna konzistencija garantuje da ce baza podataka u periodima kada nema spoljne komunikacije sa klijentima, postati konzistentna kroz neki vremenski period.

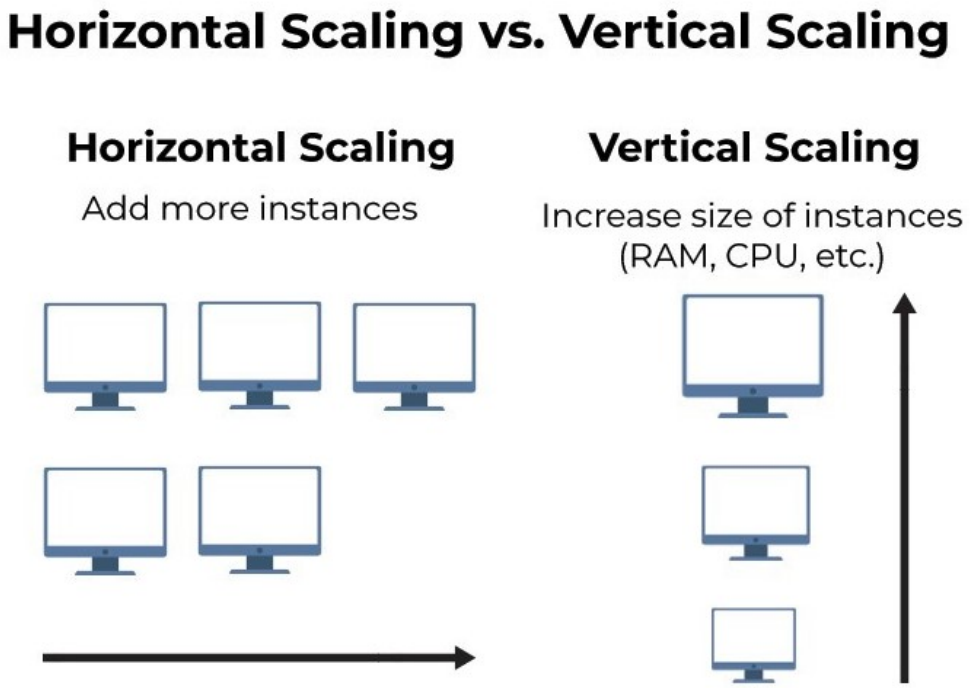
Kolonski orijentisan model načelno ne mora da ispunjava BASE svojstva, ali to obicno jeste slucaj upravo zbog horizontalnog skaliranja koja ima tendenciju da ponudi.

CAP teorema

U teoriji sistema za čuvanje podataka, CAP teorema koju je definsiao Eric Brewer, navodi da baza podatka koja skladišti deljene (distribuirane podatke) ne može istovremeno ispunjavati konzistentnost, raspoloživost, toleranciju razdvojenosti. Tolerancija razdvojenosti je svojstvo koje se obično pordazumeva i njegovo odsustvo sistem čini ne prihvatljivim u praksi. Ovo navodi da se od relaciono modela načelno očekuje da u distribuiranom okruženju osim tolerancije razdvojenosti nudi i konzistentost, za razliku od kolonski orijentisane nerelacione baze koja konzistetnost ne garantuje (ali garantuje konvergentnu konzistenciju) pa samim tim fleksibilnije radi nad deljenim podacima.



Slika 2.12: CAP teorema



Slika 2.13: Vertikalna i horizontalna skalabilnost

Glava 3

Slučajevi upotrebe

3.1 Opis i sadržaj eksperimenta

Analiza i uporedjivanje slucajeva upotrebe bice realizovani na osnovu teorijskih i prakticnih izvora i istrazivanja. Svaki primer ce biti pracen eksperimentom koji ce se sastojati od izvrsavanja razlicitih vrsta postupaka.

Kako bi se postigao dovoljan dokaz koncepta (eng. proof of concept), ali i doslednost modernom vremenu, kategorije slucajeva upotrebe koji ce biti obuhvaceni su:

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

Svakom slučaju upotrebe u okviru pripreme eksperimenata dodeljen je kontekst u skladu sa slučajem upotrebe koji se testira. Kontekst predstavlja konkretan model podataka nad kojim se izvršavaju testovi.

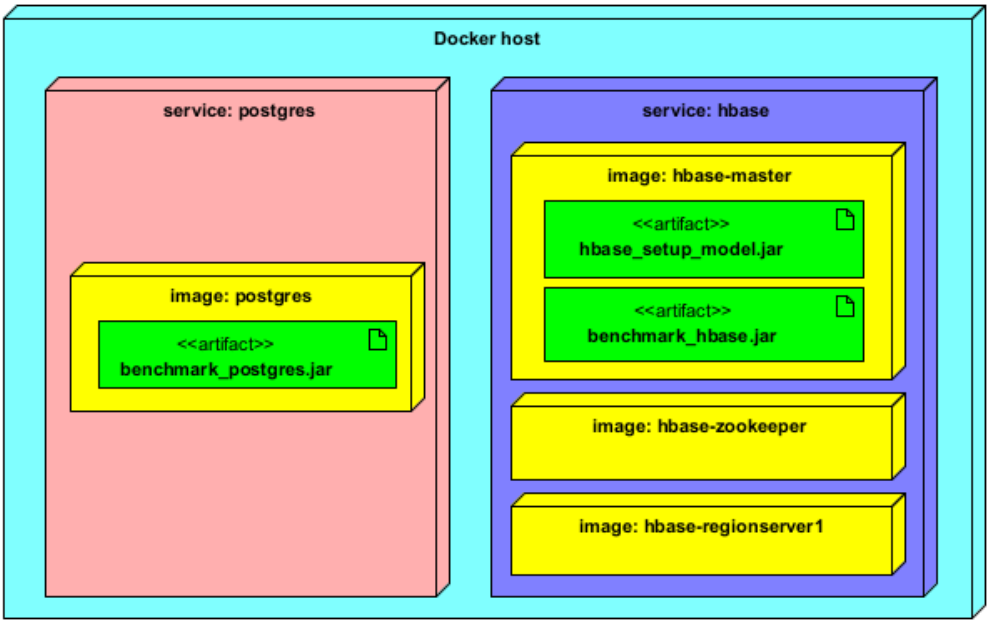
Pored definisanja konteksta, analiza slučaja upotrebe sadržaće opis java implementaicije testova nad predstavnicima, kao i analizu rezultata na kraju.

Analiza rezultata eksperimenta sprovodi se kroz vise faza. Prva faza je uporedjivanje slozenosti realizacije konkretnog slucaja upotrebe kao i to da li je konkretan slucaj upotrebe moguce realizovati sa postojecom tehnologijom. Druga faza je uporedjivanje efikasnosti, koja podrazumeva uporedjivanje vremena izvrsavanja programa. Svaka od faza ce ukljucivati tekstualnu diskusiju, slike kao i druge graficke prikaze ukoliko su pogodni.

Platforma testiranja

Kao okruženje za izvršavanje eksperimenata korišćen je docker. Oba predstavnika bice pokrenuta u okviru nezavisnih kontejnera na jednom host-u sa docker engine-om. Svaki test predstavlja jedan java program koji se izvršava na konkretnom kontejneru, kako bi se smanjila potreba za evenutalnim saobraćajem kroz mrežu kako bi se tako stekla što objektivinija slika na osnovu rezultata merenja.

Napomena: Kako su za predstavnike izabrani PostgresSQL i HBase, za rezultate merenja u nastavke treba uzeti u obzir da implementacija navedenih koncepata nije opšta za sve relacione sisteme kao ni za sve kolonski orijentisane baze podataka.



Slika 3.1: Enkodiranje po broju ponavljanja

3.2 Primena u online transakcionom procesiranju (OLTP)

Online transakciono procesiranje obično obuhvata kratke, jednostavne, učestale promene na relativno malom skupu podataka. Njih odlikuje velik broj invazivnih operacija nad bazom ali i veliki broj čitanja individualnih redova po nekom skupu atributa. Da bismo simulirali ovakvo okruženje korišćen je primer online transakcija i prenosa sredstava sa jednog računa na drugi. Parametri testa će biti broj biznis transakcija koje treba izvršiti u tom konkretnom testu, 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, izvršavanje transakcije, odnosno prenos sredstava sa jednog računa na drugi, i na kraju provera statusa transakcije.

Modeli podataka

Specifikacije modela su u nastavku:

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

U nastavku se nalazi java klasa zadužena za izvršavanje testova nad bazama:

OLTPBenchmarkExecutor.java

```
1  default void executeOLTPWorkload
2      (BenchmarkUtility util, BenchmarkOLTPUtility oltpUtil,
3       int transNum, int clNum)  {
4
5          List<Integer> transPerClientList = new ArrayList<>();
6          int transactionsToAssign = totalTransactions;
7          int transactionsPerClient = transactionsToAssign / clNum;
```

```
8
9           for(int i = 0; i<clNum;i++){
10              transPerClientList.add(transactionsPerClient);
11              transactionsToAssign-=transactionsPerClient;
12          }
13          if (transactionsToAssign > 0) {
14              int transactionForLastClient = transPerClientList.get(
15                                                          clNum - 1);
16              transPerClientList.set(clNum - 1,
17                          transactionForLastClient + transactionsToAssign
18                                                  );
19          }
20
21          assert clNum==transPerClientList.size();
22
23          Thread[] threads = new Thread[clNum];
24          CountDownLatch latch = new CountDownLatch(numOfClients);
25          for(int i = 0;i<clNum;i++){
26              threads[i] = new Thread(
27                                  new BenchmarkSingleClientExecutor(
28                                          util,oltpUtil,
29                                          i*transPerClientList.get(i),
30                                          transPerClientList.get(i),
31                                          latch)
32                          );
33          }
34
35
36          long startTimestamp = System.currentTimeMillis();
37          for(int i = 0;i<clNum;i++){
38              threads[i].start();
39          }
40          latch.await();
41          long endTimestamp = System.currentTimeMillis();
42          System.err.println("Total benchmark duration: " +
43                  (endTimestamp-startTimestamp));
44      }
```

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

Priprema okruženja

Priprema okruženja predstavlja pokretanje Postgres i Hbase standalone docker kontejnera, kompilacija java testova, podmetanje testova u vidu jar fajlova pod odgovarajuce kontejnere.

prepareEnv.sh

```
1  #!/bin/bash
2  echo 'PREPARING ENVIRONMENT...';
3
4  rm -f ./hbase_setup_model.jar
5  rm -f ./benchmark_hbase.jar
6  rm -f ./benchmark_postgres.jar
7
8  export JAVA_HOME="$JAVA_8";
9  mvn -f ./hbase_setup_model clean compile assembly:single;
10 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_postgres_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
      - 16030:16030
      - 2888:2888
      - 3888:3888
      - 2181:2181
    env_file:
```

```
34        - ./standalone.env
35
36  volumes:
37    hbase_data:
38    hbase_zookeeper_data:
```

Analiza rezultata

3.3 Primena u online analitičkom procesiranju (OLAP)

OLAP procesiranje sacinjeno je od skoro iskljucivo citanja podataka. Upiti koji se koriste obicno imaju parametre, imaju visok nivo kompleksnosti i visok procenat podataka kojima pristupa. Primer koji cemo koristiti jeste uopsten primer odrzavanja trgovinskog lanca koji ima skup musterija, proizvoda, dobavljaca, narudzbina. Nas OLAP eksperiment ce se sastojati iz dohvatanja izvestaja o ukupnom kvanitetu, ceni nakon odbijanja poreza, prosecnom popustu za dati status stavke narudzbine.

Modeli podataka

setup-postgres-model.sql

```
1
2
3   CREATE TABLE postgresdb.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 postgresdb.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 postgresdb.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 postgresdb.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 postgresdb.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 postgresdb.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','stats';
```

Implementacija testa

hbase-setup-model

```
1
2      default void executeBulkLoad(BenchmarkUtility benchmarkUtility,BenchmarkOLAPUtili
3          long bulkLoadStart = System.currentTimeMillis();
4          olapUtility.bulkLoad(benchmarkUtility.connect());
5          long bulkLoadEnd = System.currentTimeMillis();
6          System.out.println("Bulk load duration: "+(bulkLoadEnd-bulkLoadStart));
7      }
8      default void executeOLAPWorkload(BenchmarkUtility benchmarkUtility,BenchmarkOLAP
9          List<Integer> iterationsPerClientList = new ArrayList<>();
10         int iterationsToAssign = totalIterations;
```

```
11          int iterationsPerClient = iterationsToAssign /numOfClients;
12          for (int i = 0 ; i<numOfClients;i++){
13              iterationsPerClientList.add(iterationsPerClient);
14              iterationsToAssign-=iterationsPerClient;
15          }
16
17          if(iterationsToAssign>0){
18              int iterationsForLastClient = iterationsPerClientList.get(numOfClients-1)
19              iterationsPerClientList.set(numOfClients-1,iterationsForLastClient+iterati
20          }
21
22          assert numOfClients==iterationsPerClientList.size();
23
24          Thread[] threads = new Thread[numOfClients];
25          CountDownLatch latch = new CountDownLatch(numOfClients);
26
27          for(int i=0;i<numOfClients;i++){
28              threads[i] = new Thread(new BenchmarkSingleClientExecutor(benchmarkUtility
29          }
30
31          long startTimestamp = System.currentTimeMillis();
32          for(int i = 0;i<numOfClients;i++){
33              threads[i].start();
34          }
35          latch.await();
36          long endTimestamp = System.currentTimeMillis();
37          System.err.println("Total benchmark duration: " + (endTimestamp-startTimestamp
38      }
```

hbase-setup-model

```
1
2   public class BenchmarkSingleClientExecutor implements Runnable{
3
4       private final CountDownLatch endSignal;
5       private final BenchmarkOLAPUtility benchmarkOLAPUtility;
6       private final int numOfTransactions;
7       private final int startFrom;
8
9       private final Object connection;
10
11      @Override
12      public void run() {
```

```
13          try {
14              for (int i = this.startFrom; i < this.startFrom + this.numOfTransactions;
15                  benchmarkOLAPUtility.executeQuery1(connection);
16                  benchmarkOLAPUtility.executeQuery2(connection);
17                  benchmarkOLAPUtility.executeQuery3(connection);
18              }
19              endSignal.countDown();
20          } catch (Throwable e) {
21              throw new IllegalStateException(e);
22          }
23      }
24  }
```

Priprema okruženja

prepareEnv.sh

```
1  #!/bin/bash
2  echo 'PREPARING ENVIRONMENT...';
3  rm -f ./olap_benchmark_postgres.jar
4  rm -f ./olap_benchmark_hbase.jar
5  rm -f ./productHB.csv;
6  rm -f ./supplierHB.csv;
7  rm -f ./productsupplierHB.csv;
8  rm -f ./customerHB.csv;
9  rm -f ./orderHB.csv;
10 rm -f ./orderitemHB.csv;
11 rm -f ./productPG.csv;
12 rm -f ./supplierPG.csv;
13 rm -f ./productsupplierPG.csv;
14 rm -f ./customerPG.csv;
15 rm -f ./orderPG.csv;
16 rm -f ./orderitemPG.csv;
17
18
19 echo 'PREPARING HBASE BENCHMARK JARS...';
20 export JAVA_HOME="$JAVA_8";
21 mvn -f ../../benchmarking/OLAP/olap_benchmark_hbase clean compile assembly:single;
22 mvn -f ./hbase_model_utilities/hbase_setup_olap_model clean compile assembly:single;
23 mvn -f ./hbase_bulk_load_setup clean compile assembly:single;
24 cp ./hbase_model_utilities/hbase_setup_olap_model/target/hbase_setup_olap_model-1.0-SN
25 cp ../../benchmarking/OLAP/olap_benchmark_hbase/target/olap_benchmark_hbase-1.0-SNAPSH
```

```
26
27
28 echo 'PREPARING HBASE BULK LOAD RESOURCES..';
29 java -jar ./hbase_bulk_load_setup/target/hbase_bulk_load_setup-1.0-SNAPSHOT-jar-with-
30
31
32 echo 'PREPARING POSTGRES BENCHMARK JARS...';
33 export JAVA_HOME="$JAVA_17";
34 mvn -f ../../benchmarking/OLAP/olap_benchmark_postgres clean compile assembly:single;
35 mvn -f ./postgres_bulk_load_setup clean compile assembly:single;
36 cp ../../benchmarking/OLAP/olap_benchmark_postgres/target/olap_benchmark_postgres-1.0-
37
38 echo 'PREPARING POSTGRES BULK LOAD RESOURCES..';
39 java -jar ./postgres_bulk_load_setup/target/postgres_bulk_load_setup-1.0-SNAPSHOT-jar-
40
41 docker-compose -f docker-compose.yml up --build -d;
42 winpty docker exec -it olap-hbase-master-1 sh -c "java -jar hbase_setup_olap_model.jar
```

prepareEnv.sh

```
1
2 services:
3   postgres:
4     container_name: postgres
5     ports:
6       - "5433:5432"
7     volumes:
8       - ./setup_postgres_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      - hbase_data:/hbase-data
23      - hbase_zookeeper_data:/zookeeper-data
```

```
24        - ./hbase_setup_model.jar:/hbase_setup_model.jar
25        - ./benchmark_hbase.jar:/benchmark_hbase.jar
26      ports:
27        - 16000:16000
28        - 16010:16010
29        - 16020:16020
30        - 16030:16030
31        - 2888:2888
32        - 3888:3888
33        - 2181:2181
34      env_file:
35        - ./standalone.env
36
37  volumes:
38    hbase_data:
39    hbase_zookeeper_data:
```

Analiza rezultata

3.4 Primena u distribuiranom okruženju

Skalabilnost

Vertikalna skalabilnost

Horizontalna skalabilnost

CAP teorema

Glava 4

Zaključak

Bibliografija

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

[2] Optimizacija kod kolonski orijentisanih. 2024. on-line at: `https://chistadata.com/compression-techniques-column-oriented-databases/`.

[3] asdsadsada. Expected computation time for Hamiltonian path problem. *SIAM Journal on Computing*, 16:486–502, 1987. on-line at: `https://static.googleusercontent.com/media/research.google.com/fr//archive/bigtable-osdi06.pdf`.

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

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.