Distribuirane baze podataka
CockroachDB

Seminarski rad

Strahinja Laktović
br. indeksa 1089

Sistemi za upravljanje bazama podataka
Elektronski fakultet u Nišu

1. Uvod

Distribuirani sistem baza je sistem baza gde su one smeštene na više fizičkih medijuma na različitim lokacijama i njima mogu upravljati različiti sistemi za upravljanje bazama podataka. Sistemi trebaju biti projektovani tako da obezbede transparentnu distribuciju podataka za klijente, što znači da korisnici mogu slati upite sistemu bez znanja o tome gde se podaci nalaze fizički, ali i da atomičnost transakcija treba biti ista kao i pri radu sa lokalnim podacima.

U ovom radu prvo ćemo obraditi vrste distribuiranih baza, odnosno tipove njihovih arhitektura a zatim i način čuvanja podataka kao i osnove horizontalnog i vertikalnog fragmentisanja i replikacije, u drugom i trećem poglavlju.

Četvrto poglavlje bavi se katalozima imena, odnosno načinima globalnog imenovanja objekata u distribuiranom okruženju, dok se u petom poglavlju govori o načinima obrade upita i ažuriranja podataka.

Sedmo i osmo poglavlje govore o distribuiranim transakcijama i načinima distribuiranog oporavka. Transakcije su jedne od ključnih pojmova u relacionim bazama podataka i distribuirane baze trebaju zadržati ACID svojstva transakcija, krijući od korisnika fizičku odvojenost podataka i njihovo repliciranje.

U poglavljima devet i deset obradjuje se CockroachDB kao primer distribuirane relacione baze podataka koja je veoma skalabilna i ima visoku dostupnost podataka. Neki od algoritama koje CockroachDB koristi pri transakcijama i repliciranju podataka jesu optimizacije postojećih, standardnih algoritama, opisanim u prethodnim poglavljima. Na kraju rada mogu se videti primeri rada baze u distribuiranom okruženju, sa tri čvora zajedno sa metrikama performansi koje obezbedjuje sam CockroachDB pri instalaciji.

2. Tipovi distribuiranih baza podataka

U zavisnosti od toga da li serveri, na kojima se sistem baza prostire, koriste isti sistem za upravljanje bazom podataka, distribuirane baze mogu biti heterogene ili homogene. Problem koji heterogene baze trebaju rešiti jeste prihvatanje standardnog protokola za komunikaciju, odnosno API-ja koji pruža usluge sistema za upravljanje bazom drugim servisima. Standardni API omogućava prikrivanje razlika izmedju različitih sistema, ali takodje i unosi dodatnu kompleksnost u vidu preprocesiranja podataka koji stizu ili se traze iz sistema.

Nezavisno od stepena heterogenosti sistema, upravljanje distribuiranim bazama unosi dodatnu kompleksnost, smanjuje performanse i usložnjava administriranje.

U odnosu na tip arhitekture koji distribuirani sistemi baza imamo podelu na tri tipa:

- *Klijent-server sistemi* - sistemi za jasnim razdvajanjem funkcionalnosti izmedju servera i klijenta. Klijenti su jeftinije mašine, zaduženi za korisnički interfejs, najčešće grafički i pojednostavljen za korisnike. Oni šalju upite serverima, skupljim mašinama koje bi trebale da služe samo za rad sa podacima. Često se u pri korišćenju ovog tipa arhitekture podaci keširaju na klijentu da bi se smanjila učestalost komunikacije izmedju klijenta i servera i poboljšale performanse.
- *Serveri koji medjusobno saradjuju* - Ovaj tip rešava problem klijent-server arhitekture gde se jedan upit ne može prostirati na više servera jer bi klijent uvodjenjem takve podrške bio previše složen i odvajanje zaduženja izmedju klijenta i servera bi bilo narušeno. Kod servera koji medjusobno saradjuju svaki server može dobiti upit koji zahteva podatke i sa drugih servera. Tada on za te podatke generiše dodatne podupite. Pri računanja cene ovakvih upita sada se uvodi i faktor komunikacije medju serverima.
- Sistemi sa posrednicima (eng. *middleware systems*) - dozvoljavaju prostiranje upita na više servera bez postavljanja logike koja će vršiti organizaciju takvih upita kod samih servera. Umesto toga postoji samo jedan server, posrednik, koji upravlja upitima i transakcijama koji se protežu po različitim serverima, dok su oni zaduženi samo za lokalne upite i transakcije. Posrednik je zatim zadužen za spajanje podataka dobijenih sa više servera ali ne i da održava te podatke.

3. Čuvanje podataka u distribuiranim bazama

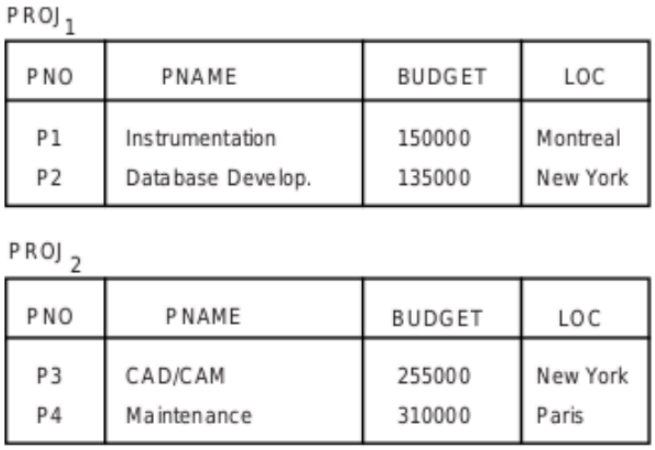
Kod distribuiranih podataka relacije se prostiru na više fizičkih lokacija. Zbog dodatne dužine trajanja upita usled slanja poruka za spajanje tih podataka oni moraju biti fragmentirani tako da se odredjeni fragmenti čuvaju na lokacijama odakle se najviše zahtevaju.

Fragmentacija predstavlja razdvajanje relacije na manje relacije ili fragmente koji mogu biti smešteni na različitim lokacijama. Postoji horizontalna fragmentacija koja deli relacije na podskupove redova i vertikalne koje dele relacije na podskupove kolona. Horizontalne fragmente možemo dobiti selekcijom a vertikalne projekcijom relacije.

Kada se vrši fragmentacija moraju biti ispoštovana odredjena pravila da bi podaci mogli da budu vraćeni u originalni oblik usled takvog zahteva u nekom upitu. To znači da unija horizontalnih fragmenata treba biti jednaka originalnoj relaciji. Takodje, podela na vertikalne

fragmente mora biti takva da prilikom ponovnog spajanja ne bude gubitaka. Fragmentiranje je operacija koja se može rekurzivno izvršavati.

Sa druge strane, u procesu replikacije se čuvaju više kopija iste relacije ili nekog njenog fragmenta na jednoj ili više mašina. Replikacija se vrši da bi se povećala dostupnost podataka,



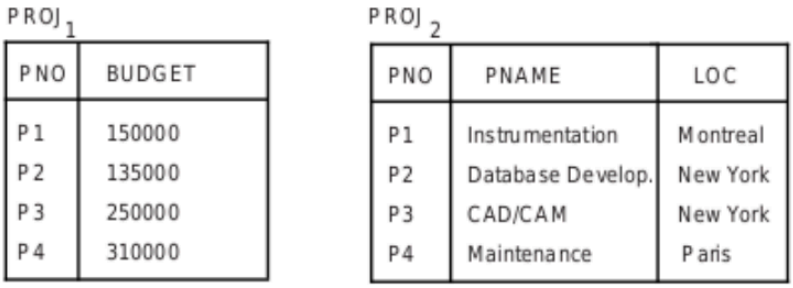
PROJ 1

PNO	PNAME	BUDGET	LOC
P1	Instrumentation	150000	Montreal
P2	Database Develop.	135000	New York

PROJ 2

PNO	PNAME	BUDGET	LOC
P3	CAD/CAM	255000	New York
P4	Maintenance	310000	Paris

Fig. 3.4 Example of Horizontal Partitioning



PROJ 1

PNO	BUDGET
P1	150000
P2	135000
P3	250000
P4	310000

PROJ 2

PNO	PNAME	LOC
P1	Instrumentation	Montreal
P2	Database Develop.	New York
P3	CAD/CAM	New York
P4	Maintenance	Paris

Fig. 3.5 Example of Vertical Partitioning

Slika 3.1 *Primer horizontalnog i vertikalnog particionisanja* [3]

odnosno u slučaju pada jedne mašine kopija na drugoj se može koristiti dalje prilikom upita, ali i da bi se upiti brže izvršavali, odnosno ukoliko postoji više kopija neke relacije ili fragmenta koji se koristi u upitu, server će pre iskoristiti svoju kopiju umesto tudje. Replikacije mogu biti sinhrone i asinhrone, o čemu će biti više reči u narednim poglavljima

4. Upravljanje katalozima imena

Praćenje podataka u distribuiranim bazama podataka može postati veoma složeno usled fragmentacije i replikacije, te se moraju voditi katalozi imena koji jedinstveno opisuju fragmente u globalnom sistemu distribuiranih baza.

Jedan unos nekog objekta obično sadrži lokalno ime relacije i ime lokacije na kom je kreiran, za jedinstvenu identifikaciju. Ukoliko je taj objekat repliciran onda se na ime dodaje i identifikator replike.

Postojanje jednog centralizovanog kataloga imena dovodi do problema jedinstvene tačke otkaza gde ukoliko on otkaže svi podaci bivaju izgubljeni. Zbog toga svaki server bi trebalo da ima lokalni katalog sa imenima svojih objekata, odnosno podataka. Dodatno, server na kome je kreirana neka relacija takodje sadrži i podatke o njenim replikama. Zbog toga se pri pristupanju nekoj replici mora kontaktirati i taj server. Serveri često keširaju ove podatke koji se vremenom menjaju, te probaju da pristupe serveru koji imaju u kešu, i ukoliko se replika tamo ne nalazi dodatno kontaktiraju server sa podacima o replikama i ažuriraju keš.

Korisnici ne bi trebali da vode računa, kada pišu upite, o tome gde se podaci nalaze i da navode njihova puna imena. Zbog toga, lokalno ime relacije u katalogu jeste, zapravo, kombinacija imena korisničkog imena korisnika koji je kreirao i imena same relacije. Kada korisnik napiše upit nad relacijom, imenu relacije se doda njegovo ime, zatim ime servera i dobije globalno ime relacije. Moguće je i kreiranje sinonima za globalna imena.

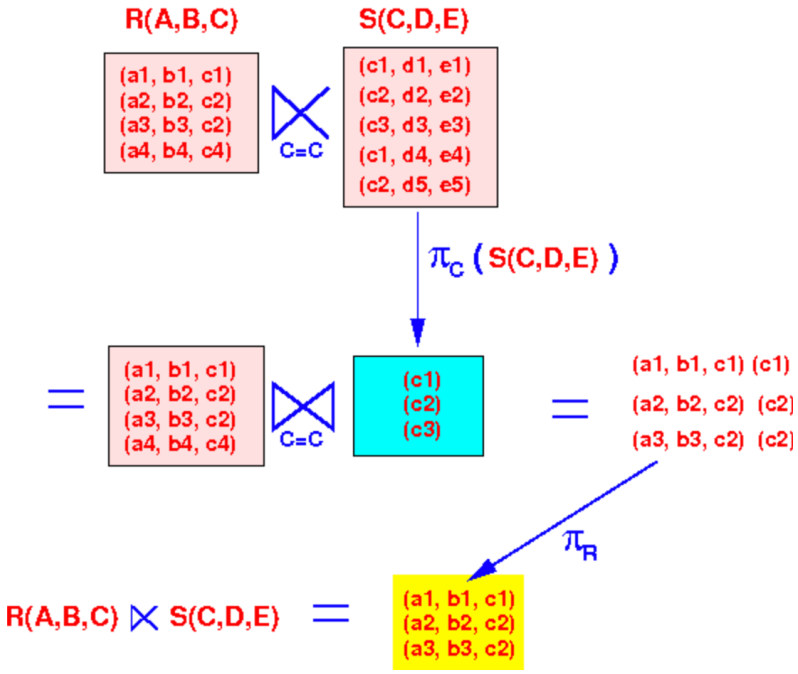
5. Obrada distribuiranih upita

Obrada distribuiranih upita u cenu upita mora da uračuna i cenu prenosa odredjenih relacija ili njihovih fragemenata preko mreže, čak i kada su u pitanju najosnovniji upiti kao što su selekcija i projekcija jedne relacije. Kod upita sa *joinovima* problemi distribuiranih upita postaju još očigledniji kada se relacije koje trebaju biti joinovane nalaze na fizički distribuiranim serverima. Tehnike za ublažavanje problema sa joinovima i smanjivanje broja torki koje moraju biti kopirane sa jednog servera na drugi su:

- *Semijoin* - tehnika kod koje se prvo izračunava projekcija jedne relacije na drugu i šalje na lokaciju druge. Zatim se na drugom serveru izračuna prirodni join projekcije sa prvog servera sa lokalnom relacijom. Ovakav join se zove redukcija druge relakcije u odnosu na prvu. Konačno, redukcija se šalje natrag na prvi server gde se vrši join sa prvom relacijom.
- *Bloomjoin* - razlika u odnosu na semijoin je ta da se, u slučaju bloomjoina, u prvom koraku šalje *bit vektor* umesto projekcije. Bit vektor, veličine *k* računa se heširanjem svake torke u rasponu od 0 do k - 1 i postavljanjem bita na 1 ako se torka i hešira u broj i, ili, u suprotnoom na 0. U drugom koraku , redukcija druge relacije se računa heširanjem torki korišćenjem stranog ključa, na raspon od 0 do k - 1, korišćenjem iste heš funkcije za kreiranje bit vektora i izbacivanjem torki čija heširana vrednosot oodgovara bitu vrednosti 0. Cena slanja bit vektora i redukovanja druge relacije uz pomoć vektora je manja od cena kod semijoina. Sa druge strane, veličina redukcije će biti verovatno veća nego kod semijoina, te će cena zadnjeg koraka biti skuplja.

Globalna optimizacija upita kod distribuiranih baza mora uzimati u obzir troškove komunikacije i autonomiju različitih sistema za upravljanje bazama podataka na različitim serverima. Optimizacija inicijalno kreće kao kod centralizovanih sistema, prikupljanjem informacija o različitim relacijama iz kataloga. Zatim se uzimaju u obzir lokalni planovi gde se smeštaju podaci u relacije koje služe za slanje preko mreže. Lokalni planovi bi trebalo da realistično računaju cenu svojih podupita, s tim što globalni optimizator zadržava pravo

nalaženja jeftiniji rešenja i ignorisanja lokalnih planova ukoliko takva rešenja ne narušavaju autonomiju servera na koje se odnose.



Slika 5.1 *Primer semijoin-a* [4]

6. Ažuriranje podataka

Distribuirane baze podataka se trebaju ponašati kao centralizovane baze iz ugla korisnika. Što se tiče upita, korisnik ne treba da zna na kojoj lokaciji se nalaze odredjene relacije. Što se tiče ažuriranja, transakcije treba da ostanu atomične, uprkos fragmentaciji i replikaciji. Konkretno, pri ažuriranju jedne kopije podataka sve ostale trebaju biti takodje ažurirane. Replikacije sa ovom semantikom zovu se sinhrone replikacije. Kod asinhronih replikacija, koje se koriste kod komercijalnih sistema za upravljanje bazama podataka, kopije se periodično ažuriraju, i transakcija koja čita više različitih kopija jednog podatka može pročitati različite vrednosti, čime se ugrožava nezavisnost podataka kod distriburanih podataka. Medjutim, asinhrone replikacije se implementiraju efikasnije od sinhronih.

Sinhrona replikacija koristi dve osnovne tehnike za sinhronizaciju podataka.

Prva tehnika, *glasanje,* je tehnika gde transakcija mora da ažurira vrednost većine kopija da bi modifikovala objekat i da pročita dovoljan broj kopija da bi ustanovila da je vrednost koju čita prava vrednost. Svaka kopija ima broj verzije, kopija sa najvišim brojem je trenutna vrednost. Ova tehnika je vrlo nepopularna zato što je čitanje operacija koja je mnogo češća od ažuriranje a ovde je čitanje usporeno.

Druga tehnika, čitaj-bilo-koji upiši-sve (eng. *read-any write-all*) je tehnika gde je potrebno pročitati samo jednu kopiju podatka, što čitanje čini jako brzim, pogotovo ako je kopija lokalna,

ali si priliikom upisivanja moraju ažurirati sve kopije. Ova tehnika se najčešće koristi kod sinhrone replikacije.

Problem sa sinhronom replikacijom, konnkretno ukoliko se koristi čitaj-bilo-koji upiši-sve je taj što je neophodno obaviti ekskluzivna zaključavanja svih kopija podataka koji se menja pre potvrdjivanja transakcije. Ukoliko ima problema u komunikaciji i neki od servera bude pao za vreme potvrdjivaja, on se mora sačekati pre nastavljanja. Čak i u slučaju uspešnog zaključavanja svih objekata, velika je cena razmene poruka kod protokola potvrdjivanja. Zbog toga se često bira asinhrona replikacija, sa cenom da korisnici znaju kojim replikama pristupaju i znaju da kopije bivaju ažurirane povremeno.

Postoje dve tehnike asinhronog ažuriranja. Prva tehnika, primarno mesto (eng. *primary site*) je tehnika gde postoji samo jedna primarna kopija relacije. Replike relacije ili fragmenti mogu biti kreiranni na drugim mestima, ali su to sekundarne kopije i ne mogu biti ažurirane. Korisnici prvo objave replikaciju na primarnom mestu a zatim se pretplate na fragmente ili relacije na drugom mestu.

Glavni problem kod primarnog mesta je odluka kako propagirati promene nad prvom kopijom drugim kopijama. Rešavanje ovog problema najčešće se rešava kroz dva koraka - zauzeće (eng. *capture*) i primena (eng. *apply*). U koraku zauzeća promene potvrdjenih transakcija su detektovane i distribuirane u koraku primene.

Zauzeće može biti implementirano na dva načina. Zauzeće baziano na logu pravi zapis promena. Kada taj zapis dospe na stabilni medijum čuvanja tada se zapisi koji sadrže ažuriranja za sve kopije čuvaju u tabelu promene podataka (CDT). Kada se zapisi dodaju u CDT transakcija možda nije potvrdjena još uvek. Zbog toga se pri pribavljanju zvaničnih podataka za ažuriranje moraju obrisati podaci povezani poništenih transakcija ili da se u CDT dodaju i podaci o potvrdjenim transakcijama da bi se u sledećem koraku pribavili zvanični validni podaci.

Proceduralno zauzeće je osmišljeno tako da sistem za upravljanje bazom pdoataka zove proceduru koja inicijalizuje proces zauzeća koji čuva *snapshot* primarne kopije. Snapshot je vrednost kopije u odredjenom trenutku u vremenu. Obično zauzeće baziranog na logu ima manji vremenski razmak od ažuriranja jedne vrednosti do propagiranja tih promena drugim kopijama.

Korak pimene koristi podatke iz CDT-a ili snapshotove i propagira ih drugim kopijama. Primarno mesto može povremeno da šalje CDT ili snapshotove drugima, ali je češći slučaj gde sekundarna mesta povremeno traže ove podatke da bi ažurirali svoje replike.

Kombinacija zauzeća baziranog na logovima i kontinuirane primene maksimalno umanjuje dužinu propagacije promena i koristi se u situacijama kada sve kopije moraju biti stalno ažurirane. Zauzeće bazirano na logu i kontinuirana primena su u kombinaciji jeftiniji od sinhrone replikacije. Sa druge strane proceduralno zauzeće i primena bazirana na tajmeru u nekoj aplikaciji omogućavaju preprocesiranje podataka pre nego što ažuriraju kopije, što više odgovara tipovima baza koje se koriste kao skladište podataka, gde je bitnije imati repliku koja je pravilno obradjenja od imanja replike koja je maksimalno ažurna.

Druga tehnika asinhrone replikacije, jednak-jednakom (eng. *peer-to-peer*). Više kopija može biti ažurirano. Ali kada se promene propagiraju na ostala mesta moraju se rešavati konflikti ukoliko je više od jedne ažurirano u isto vreme. Postoje situacije gde ne dolazi do ovakvih konflikata i gde se ova tehnika uspešno koristi. Prvo, svaki *master* server može ažurirati samo fragment (obično horizontalni) neke relacije i fragmenti koje različiti masteri ažuriraju nemaju

preklapanja. U drugom slučaju ažuriranje može da vrši samo jedan master. Jedan server je glavni a ostali služe kao rezerva u slučaju kvara prvog i bivaju ažurirane naknadno.

7. Distrbuirane transakcije

Prilikom izvršavanja distribuiranih transakcija, transakcije započete na jednom serveru mogu imati pristup i podacima sa drugih servera, te ih transakcioni menadžer deli na podtransakcije koje se izvršavaju tamo gde se nalaze podaci za njih i koordinišu njihov rad. Da bi ovakve transakcije bile uspešne, potrebno je obezbediti efikasan način upravljanja zaključavanjem podataka i način razrešavanja potpunih zastoja (eng. *deadlock*). Takodje, proces distribuiranog oporavka je znatno složeniji od centralizovanog, zato što se mora osigurati da promene potvrdjene transakcije budu primenjene na svim različitim mestima gde su se odvijale podtransakcije, čak i u slučaju njihovog pada.

Tehnike za implementaciju sinhrone i asinhrone replikacije diktiraju koje će replike nekog podatka biti zaključane. Tehnike za upravljanje konkurencijom odredjuju kada će biti zaključane i otključane. Upravljanje konkurentnšću obavlja se na jedan od tri načina:

- *Centralizovano* - jedno mesto je zaduženo za upravljanje zahtevima za zaključavanje i otključavanje za sve objekte.
- Primarna kopija - jedna kopija svakog objekta je proglašena za primarnu kopiju. Svim zahtevima da se neka kopija zaključa ili otključa upravlja menadžer zaključavanja na mestu gde se nalazi primarna kopija, bez obzira gde se ta kopija nalazi.
- Potpuno distriburano - zahtevima za zaključavanje i otključavanje neke kopije upravljaju menadžeri na mestima gde se te kopije nalaze.

Centralizovana šema je ranjiva na otkaz u jednoj tački i zbog toga je manje privlačna od ostale dve. Primarna kopija rešava ovaj problem ali uvodi komunikaciju sa dva mesta, sa mestom gde se nalazi kopija i sa mestom primarne kopije, što nije vrlo poželjno usporavanje kada je u pitanju obično čitanje podataka. To rešavaju potpuno distriburane šeme, ali za upisivanje je kod njih potrebno zaključavanje kod svake od njih, za razliku od prethodne dve. Medjutim, zato što je upisivanje mnogo manje često u odnosu na čitanje, distribuirana šema i dalje ostaje kao šema sa najviše benefita.

Kada je u pitnaju rešavanje potpunih zastoja, svaki server ima svoj lokalni graf čekanja i ciklus u grafu označava potpuni zastoj. Ali u distribuiranom okruženju ovaj graf nije dovoljan zato što je moguće imati situaciju potpunog zastoja čak i kad nema ciklusa u lokalnim grafovima, zato što se mogu slati zahtevi za zaključavanje i drugim serverima. Da bismo detektovali ovakve grafove moramo imati algoritam za detekciju distribuiranih potpunih zastoja.

Kod c*entralizovanog algoritma* za detekciju distribuiranih globalnih zastoja, serveri povremeno šalju svoje lokalne grafove centralnom serveru koji generiše globalni graf koji se sastoji od unije svih čvorova.

Hijerarhijski algoritam grupiše servere u hijerarhije. Hijerarhije su geografski odredjene, npr. u najmanjoj grupi su serveri koji se nalaze u istom okrugu, zatim se na višem nivou nalaze oni u istoj državi itd. Motivacija za ovakvo grupisanje je bazirano na pretpostavci da se potpuni zastoji češće dešavaju u situacijama sa serverima koji su geografski blizu. Svaki čvor u ovakvoj postavci konstruiše lokalni graf za svoje podstablo. Zatim se grafovi sa jednog nivoa periodično

šalju višem nivou i tako sve dok se ne konstruiše globalni graf. Svi potpuni zastoji se detektuju ali zastoji geografski udaljenih čvorova se dosta sporije detektuju od ,,lokalnih".

Treći algoritam je dosta jednostavniji u odnosu na prva dva. Ukoliko neka transakcija čeka duže od nekog predvidjenog vremena ona se poništava. Ovaj način detekcije može dovesti do nepotrebnih poništavanja, ali je nekad jedini izboru heterogenim distribuiranim bazama.

Prilikom detekcije potpunih zastoja kod prva dva algoritma može doći do detekcije i takozvanih *fantomskih potpunih zastoja*, zastoja koji više ne postoje ali se zbog kašnjenja poruka u mreži detektovani, što može dovesti do nepotrebnih poništavanja transakcija.

8. Distribuiran oporavak

Oporavak u distribuiranim sistemima je složeniji u odnosu na centralne zato što može doći i do novih vrsta problema, kao što su problemi u komunikaciji i zato što svaki server koji učestvuje u transakciji mora potvrditi svoju lokalnu podtransakciju. Kao i u centralizovanim sistemima, održavaju se logovi i prate odredjeni protokoli da bi se transakcije uspešno potvrdile. Najčešće korišćen protokol je dvo-fazna potvrda (eng. *two-phase commit*) i njegova verijanta sa pretpostavljenim obustavljanjem (eng. *presumed abort*) prihvaćena je kao industrijski standard.

Za vreme normalnog izvršenja, svaki server održava log gde se upisuju akcije transakcije. Kod protokola dvo-fazonog potvrdjivanja, menadžer transakcija koji se nalazi na serveru gde je transakcija otpočela zove se koordinator, a ostali menadžeri podredjeni (eng. *subordinate*).

Kada korisnik odluči da potvrdi transakciju, akcija potvrdjivanja se šalje koordinatoru. Zatim on šalje poruke pripreme svojim podredjenima. Kada podredjeni primi poruku on odlučuje da li će obustaviti ili potvrditi transakciju. Svoju odluku upisuje u log i šalje odgovor koordinatoru. Ako koordinator primi potvrdan odgovor od svih podredjenih onda upisuje potvrdan odgovor u log i šalje poruku svim podredjenima. Ako primi bar jedan negativan odgovor onda obustavlja transakciju i šalje svim podredjenima poruku obustavljanja. Kada podredjeni dobiju ove poruke onda oni upisuju podatak u lokalni log i šalju poruku razumevanja zahteva (*ack* poruka). Koordinator, nakon primanja ack poruka od svih podredjenih, konačno upisuje podatke o kraju transakcije u log.

Kada se transakcija zvanično potvrdi, u trenutku kada log koordinatora dospe na stabilni medijum za čuvanje, ne postoji opcija opovrgavanja te transakcije, čak i ako neki od podredjenih padne nakon toga jer log potvrde sadrži i imena podredjenih.

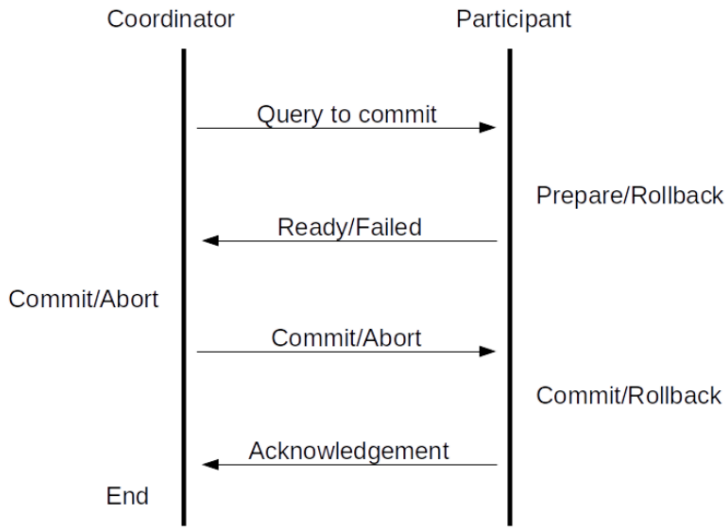
Ukoliko dodje do pada servera i on se restartuje, ulazi se u proces oporavka koji čita log i procesuira sve ttransakcije koje su izvršavale potvrdjivanje u trenutku pada.

Ako server ima log potvrde ili obustavljanja za trannsakciju onda se izvršava akcija na osnovu tog podatka. Ako je server koordinator, što se može zaključiti iz liste učesnika u logovima, on mora povremeno slati poruke obustave ili potvrde drugim sajtovima koji su možda takodje pali u trenutku završavanja transakcije, dok ne dobije poruku ack od svih.

Ako server ima log pripreme ali nema log potvrde ili obustave, to znači da je server podredjeni i da koordinator može biti odredjen iz loga. Podredjeni mora stalno da kontaktira koordinatora da odredi status transakcije. Kada koordinator odgovori potvrdom ili obustavom transakcije tada se može to i odraditi.

U slučaju da nema nijedne vrste loga za transakciju, o transakciji se sigurno nije izglasalo do kraja pre kraha. Stoga se ne može odrediti da li je server koji je pao koordinator ili ne, ali je možda pre pada poslao poruku pripreme nekom drugom serveru koji će ga sad kontaktirati i onda će znati da je koordinator. Ukoliko koordinator padne pre potvrdjivanja transakcije, podredjeni koji su glasali potvrdno ne mogu sami da odluče da potvrde transakciju dok se koordinator ne oporavi.

Ako server komunicira sa drugim serverom koji je nedostupan u trenutku komuniciranja, a server je koordinator, onda treba da poništi transakciju. Ako je dati server podredjeni i nije još odgovorio na koordinatorovu poruku pripreme onda treba takodje da poništi transakciju. Ako je odgovorio i glasao potvrdno, onda se mora čekati koordinatorov oporavak i treba ga povremeno kontaktirati.



Slika 8.1 *Dvo-fazno potvrdjivanje* [5]

Verzija dvo-faznog potvrdjivanja sa pretpostavljenim poništavanjem uvodi optimizacije kod dvo-faznog potvrdjivanja u smanjivanja komunikacije. Ukoliko podredjeni dobije poruku obustave ne mora da šalje odgovor potvrde. Koordinator, u procesu oporavka posle pada, ukoliko nema informacije u logovima o statusu transakcije on je obustavlja i šalje odgovor obustave svakom ko ga dodatno kontaktira. Uvodi se i poruka čitanja koju podredjeni može poslati koordinator umesto potvrdnog ili negativnog odgovora za potvrdjivanje transakcije zato što neke transakcije uopšte ne modifikuju podatke. Koordinator smatra ove poruke potvrdnim odgovorom ali briše transakciju iz svojih logova jer nema upisivanja zbog kojih mora da koordiniše.

Tro-fazno potvrdjivanje je protokol koji se ne koristi u praksi jer u normalnom izvršavanju uvodi preveliku složenost i odlaganje konačnog povrdjivanja ili obustave zbog komunikacije, ali rešava problem blokiranja čak i kada koordinator padne u toku oporavka. Ideja je da koordinator nakon potvrdnih odgovora podredjenima pošalje prepotvrdnu poruku (eng. *precommit*) umesto potvrdne. Kada sakupi dovoljan broj ack odgovora on upisuje potvrdnu poruku u log i onda je

šalje podredjenima. Time se odlaže potvrdjivanje transakcije taman toliko da dovoljan broj podredjenih zna odluku pre nego što se ona izvrši, da bi se u slučaju pada koordinatora znalo šta da se radi, putem medjusobne komunikacije.

9. Koncepti distribuiranih baza kod CockroachDB-a

Kako organizacije prelaze na korišćenje cloud-a za sve svoje potrebe, tako baze podataka koje se koriste za organizacije sa takvim potrebama trebaju da prate trendove. Stare relacione baze u legacy sistemima ne iskorišćavaju cloud potencijal do kraja i veoma ih je teško uspešno skalirati. NoSQL baze podataka donekle rešavaju ove probleme, omogućavajući veliku skalabilnost, ali bez ACID transakcija koje su ključne za odredjene sisteme.

CockroachDB je jedna od baza novije generacije koja spadau distribuirane SQL baze. Da bi to postigla potrebno je ispuniti nekoliko uslova:

- *skaliranje* - sposobnost skaliranje baze i ide ukorak sa mogućnostima cloud okruženja, podržavajući velike workload-e.
- *konzistentnost* - distribuirane baze moraju imati visok stepen izolacije u distribuiranom okruženju. U svetu distribuiranih sistema i mikroservisa transakcionalna konzistentnost postaje izuzetno teža nego kod centralizovanih, ali ove potrebe moraju biti ispunjene.
- *otpornost* - distribuirane baze trebaju imati otpornost na najvišem nivou, omogućavajući oporavak delova sistema bez prestanka rada cele baze koristeći replikaciju.
- *geo-replikacija* - baze trebaju imati mogućnost distribucije podataka kroz kompleksnu, geografski razudjenu mrežu.
- SQL - za veoma mali broj distribuiranih baza, kao što je CockroachDB, može se reći da imaju stari, dobro poznat, SQL dijalekt

Nakon otpremanja CockroachDB baze u produkciju, programerima je dovoljan connection string i SQL da bi radili s bazom. Čvorovi CockroachDB-a se ponašaju simetrično, zahtevi mogu biti poslati bilo kom čvoru koji će se kasnije ponašati kao gateway ka klasteru. Kada SQL naredbe stignu do klastera one se pretvaraju u planove ključ/vrednost parova koji se prosledjuju transakcionom sloju.

Da bi se održala konzistentnost, CockroachDB implementira potpunu podršku za ACID transakcione semantike u transakcionom sloju. Kod CockroachDB-a sve naredbe se posmatraju kao transakcije, odnosno ponašanje baze je takvo kao da posle svake naredbe ide naredba *potvrdi* (eng. *commit*).

Prva faza transakcije je faza čitanja i upisivanja koje kreira nove objekte potrebne za dalje faze transakcije. Pri upisivanju novih podataka, CockroachDB obezbedjuje nekoliko nivoa zaključavanja. I to:

- *nereplicirana zaključavanja* - čuvaju se u memoriji, u tabeli zaključavanja na nivou čvora i ne repliciraju se
- *replicirana zaključavanja* - drugačije zvani namere zaključavanja (eng. *write intents*) se repliciraju i deluju kao kombinacija probne vrednosti objekta i ekskluzivnog zaključavanja i sadrže pokazivač na *podatke o transakciji* koji se čuvaju u klasteru

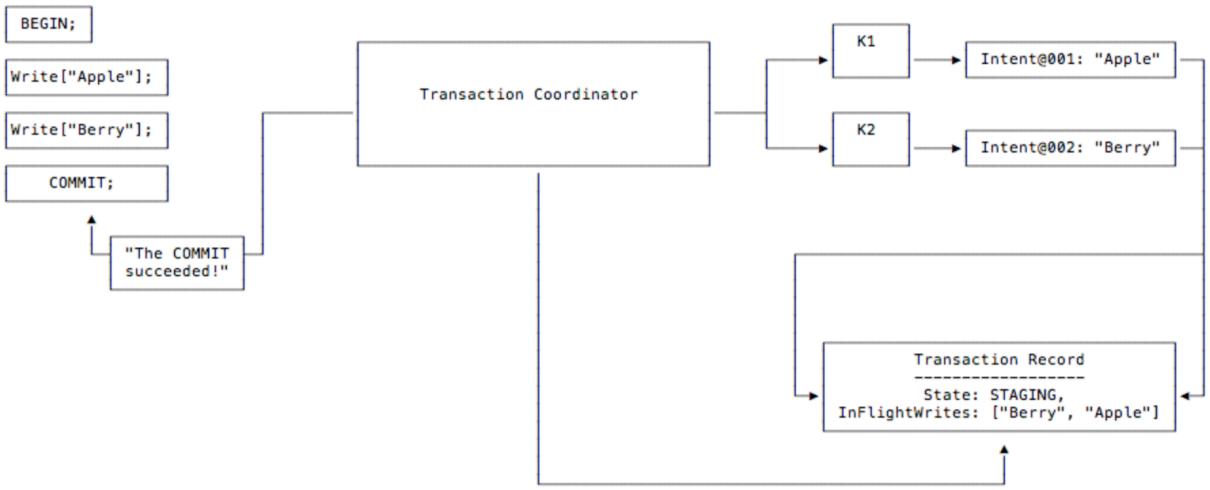
Podaci o transakciji sadrže i status transakcije koji može biti *čekanje* (eng. *pending*), *priprema* (eng. *staging*), *potvrdjena* (eng. *commited*), *poništena* (eng. *aborted*).

Kada se namere upisivanja kreiraju, CockroachDB proverava da li ima novijih vrednosti koje su potvrdjene. Ako ima onda se transakcija restartuje. Ako ima drugih namera upisivanja onda se ulazi u fazu *rešavanja konflikta*.

Druga faza, faza potvrdjivanja, proverava da li je transakcija poništena, ako jeste onda je restartuje. U opštem slučaju postavlja transakciju na stanje priprema i proverava da li su namere upisivanja uspele da se repliciraju. Kada transakcija prodje sve provere onda se klijentu javlja da je uspela a transakcija prelazi u sledeće stanje. U tom trenutku transakcija je potvrdjena, a klijent može da šalje nove zahteve.

Faza potvrdjivanja oslanja se protokol koji se zove paralelno potvrdjivanje (eng. *Parallel Commits*) koji kašnjenje transakcije prepolovljava, sa dve runde koncenzusa na

Nakon što je transakcija potvrdjena sve namere upisivanja trebaju biti razrešene, u *fazi tri*, a o tome se brine čvor koordinator koji menja stanje transakcije, pretvara namere upisa u vrednosti sa datumom koji govori o njihovoj ažurnosti i brise namere.



Slika 9.1 *Paralelno potvrdjivanje* [6]

Da bi CockroachDB obezbedio dostupnost svih podataka sa bilo kog čvora, podaci se čuvaju u monolitski sortiranu mapu parova ključ vrednost. Ključevi su organizovane u raspone (eng. *ranges*), kontinuirane delove prostora ključeva i time se omogućava pronalazak ključa u tačno jednom rasponu. Mape se implementiraju ovako da bi obezbedile:

- *jednostavna pretraživanja* - zato što ključevi, izmedju ostalog, sadrže i lokaciju podataka, kada se šalju upiti bazi oni brzo mogu da lociraju potrebne podatke
- *efikasna skeniranja* - definisanjem uredjenja podataka, lako je naći podatke unutar nekog raspona ključeva u toku skeniranja.

Čvorovi medjusobno komuniciraju korišćenjem *gRPC* protokola.

Visoka dostupnost zahteva da baza može istolerisati da neki čvorovi budu u kvaru neko kratko vreme bez prekidanja pružanja usluga aplikacijama. Zbog toga se podaci repliciraju.

U procesu repliciranja CockroachDB koristi *Raft konsenzus protokol*, algoritam koji se stara o tome da su podaci bezbedno čuvaju na više mašina i da se te mašine slažu šta je trenutno stanje podataka iako su možda neke od njih offline.

Raft organizuje sve čvorove koji sadrže repliku raspona parova ključ/podatak u Raft grupe. Svaka replika u Raft grupi je vodja (eng. *leader*) ili pratilac (eng. *follower*). Vodja, kog izabere Raft, koordiniše sve upise u Raft grupu. Povremeno proverava da li su pratioci dostupni i održava njihove logove repliciranim. Ukoliko odredjen vremenski period niko ne proverava pratioce, pratioci postaju kandidati i održavaju izbore za novog vodju. Kada čvor primi *BatchRequest* (gRPC zahtev) za raspon koji sadrži, pretvara parove ključeva/podataka u Raft komande. Komande se predlažu vodji Raft grupe koji ih šalje ostalim čvorovima i replicira podatke. Nakon dostizanja konsenzusa o novoj vrednosti, odnosno kad vodja dobije od dovoljnog broja čvorova potvrdu da je nova vrednost replicirana i kod njih, vodja potvrdjuje upisivanje i upisuju se u Raft log. Ovim dobijamo uredjeni set komandi za koje se sve replike slažu da predstavljaju izvor istine za konzistentnu replikaciju. Logovi su serijalizabilni.

Svaki od čvorova baze ima svoj *store,* mesto na kom *cockroach* proces upisuje i čita podatke sa diska. Podaci se upisuju kao ključ-vrednost parovi korišćenjem *RocksDB-a* koji se tretira kao black box API. Interno, svaki store sadrži dve instance RocksDB-a, jednu za čuvanje privremeno distribuiranih SQL podataka, drugu za čuvanje svih ostalih podataka. Dodatno, na čvoru postoji keš blok koji dele svi store-ovi na čvoru. Store-ovi na smenu sadrže kolekciju range replika. Više od jedne replike nekog range-a nikad neće biti smeštena samo u jednom store-u, čak i na istom čvoru.

10. Primeri rada distribuirane baze kod CockroachDB-a

Primeri rada demonstrirani u nastavku koriste docker image cockroach db-a *cockroachdb/cockroach:20.1.1.* Nakon skidanja docker image-a kreirana docker network bridge mreža i klaster od tri čvora. Primer pokretanja jednog od čvorova:

```
$ docker run -d \
--name=roach1 \
--hostname=roach1 \
--net=roachnet \
-p 26257:26257 -p 8080:8080  \
-v "${PWD}/cockroach-data/roach1:/cockroach/cockroach-data"  \
cockroachdb/cockroach:v20.1.1 start \
--insecure \
--join=roach1,roach2,roach3
```

Slika 10.1 *Pokretanje CockroachDB čvora putem docker-a* [6]

Nakon pokretanja tri čvora lokalnog klastera demonstrirana je distribuiranost podataka ulaskom na prvi čvor, kreiranjem baze, tabele i selektovanjem rezultata iz tabele. Zatim se ulaskom na drugi čvor i izvršavanjem istog upita dobijaju isti podaci.

```
$ docker exec -it roach1 ./cockroach sql --insecure
```

```
> CREATE DATABASE bank;
```

```
> CREATE TABLE bank.accounts (id INT PRIMARY KEY, balance DECIMAL);
```

```
> INSERT INTO bank.accounts VALUES (1, 1000.50);
```

```
> SELECT * FROM bank.accounts;
```

```
  id | balance
+----+---------+
   1 | 1000.50
(1 row)
```

Slika 10.2 *Ulazak na čvor koji se izvršava u kontejneru i kreiranje baze, tabele i izvršavanje upita* [6]

```
$ docker exec -it roach2 ./cockroach sql --insecure
```

```
> SELECT * FROM bank.accounts;
```

```
  id | balance
+----+---------+
   1 | 1000.50
(1 row)
```

Slika 10.3 *Ulazak na drugi čvor koji se izvršava u kontejneru i izvršavanje istog upita* [6]

U daljem testiranju vršimo testove sa jednim od workloadova koje je unapred pripremio CockroachDB za testiranje različitih scenarija kod klastera. Izabran je *tpcc* workload koji simulira procesiranje transakcija u šemi sa većem broja tabela. Naredbe za inicijalizaciju i početak rada workloada prikazane su na slikama *10.4 i 10.5* u nastavku:

Slika 10.4 *Inicijalizacija tpcc workloada i tabela sa podacima* [6]

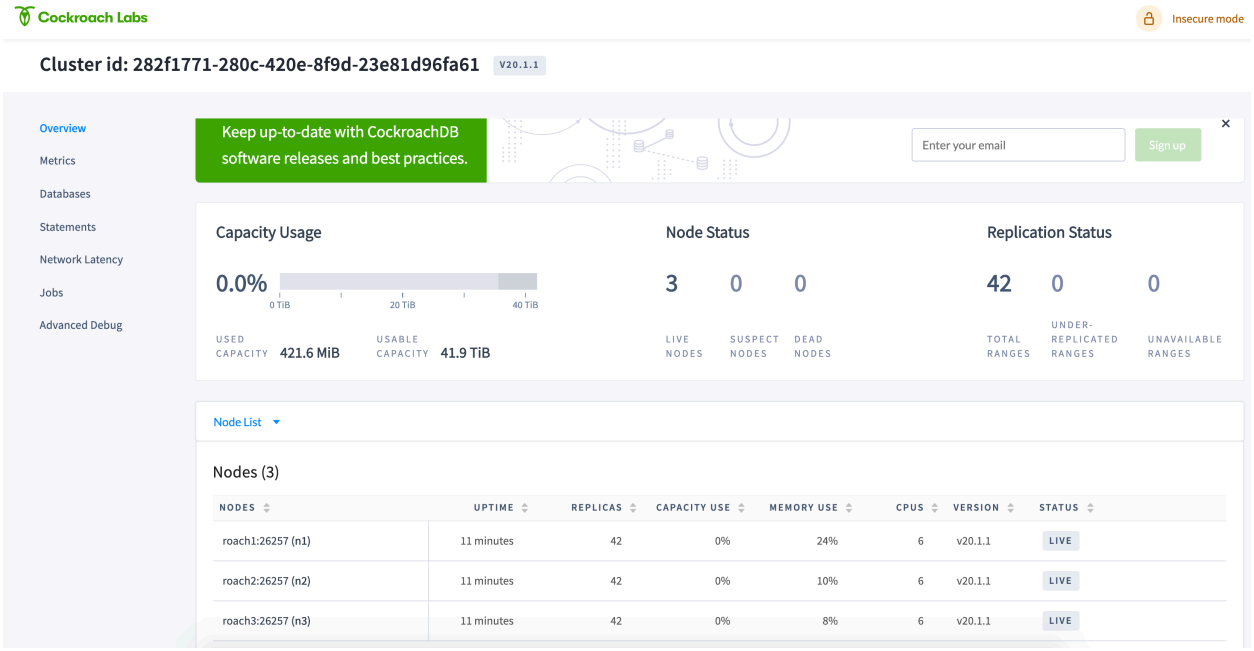
```
➜  Projects docker exec -it roach1 ./cockroach workload init tpcc 'postgresql://root@roach1:26257?sslmode=disable'
I200607 12:32:57.467385 1 workload/workloadsql/dataload.go:140  imported warehouse (0s, 1 rows)
I200607 12:32:57.499097 1 workload/workloadsql/dataload.go:140  imported district (0s, 10 rows)
I200607 12:32:59.829644 1 workload/workloadsql/dataload.go:140  imported customer (2s, 30000 rows)
I200607 12:33:00.868556 1 workload/workloadsql/dataload.go:140  imported history (1s, 30000 rows)
I200607 12:33:01.572267 1 workload/workloadsql/dataload.go:140  imported order (1s, 30000 rows)
I200607 12:33:01.670414 1 workload/workloadsql/dataload.go:140  imported new_order (0s, 9000 rows)
I200607 12:33:03.142824 1 workload/workloadsql/dataload.go:140  imported item (1s, 100000 rows)
I200607 12:33:07.496433 1 workload/workloadsql/dataload.go:140  imported stock (4s, 100000 rows)
I200607 12:33:15.454388 1 workload/workloadsql/dataload.go:140  imported order_line (8s, 300343 rows)
```

Na slici *10.5* ispod vidimo da je workload inicijalizovan da traje pet minuta i u logovima kontejnera biće prikazane metrike koje se menjaju nakon svakog koraka u workloadu i pokazuju broj proteklih sekundi od početka rada workload-a, broj dosadašnjih greški, tabele na koje se odnose naredbe i druge.

```
→  Projects docker exec -it roach1 ./cockroach workload run tpcc --duration=5m 'postgresql://root@roach1:26257?sslmode=disable'
Initializing 2 connections...
Initializing 10 workers and preparing statements...
_elapsed___errors__ops/sec(inst)___ops/sec(cum)__p50(ms)__p95(ms)__p99(ms)_pMax(ms)
    1.0s        0            0.0            0.0      0.0      0.0      0.0      0.0 delivery
    1.0s        0            0.0            0.0      0.0      0.0      0.0      0.0 newOrder
    1.0s        0            0.0            0.0      0.0      0.0      0.0      0.0 orderStatus
    1.0s        0            0.0            0.0      0.0      0.0      0.0      0.0 payment
    1.0s        0            0.0            0.0      0.0      0.0      0.0      0.0 stockLevel
    2.0s        0            0.0            0.0      0.0      0.0      0.0      0.0 delivery
    2.0s        0            0.0            0.0      0.0      0.0      0.0      0.0 newOrder
    2.0s        0            0.0            0.0      0.0      0.0      0.0      0.0 orderStatus
    2.0s        0            0.0            0.0      0.0      0.0      0.0      0.0 payment
    2.0s        0            0.0            0.0      0.0      0.0      0.0      0.0 stockLevel
    3.0s        0            1.0            0.3    100.7    100.7    100.7    100.7 delivery
    3.0s        0            0.0            0.0      0.0      0.0      0.0      0.0 newOrder
    3.0s        0            3.0            1.0     15.2     22.0     22.0     22.0 orderStatus
    3.0s        0            0.0            0.0      0.0      0.0      0.0      0.0 payment
    3.0s        0            0.0            0.0      0.0      0.0      0.0      0.0 stockLevel
    4.0s        0            0.0            0.2      0.0      0.0      0.0      0.0 delivery
    4.0s        0            0.0            0.0      0.0      0.0      0.0      0.0 newOrder
    4.0s        0            0.0            0.7      0.0      0.0      0.0      0.0 orderStatus
    4.0s        0            3.0            0.7     83.9    113.2    113.2    113.2 payment
    4.0s        0            0.0            0.0      0.0      0.0      0.0      0.0 stockLevel
_elapsed___errors__ops/sec(inst)___ops/sec(cum)__p50(ms)__p95(ms)__p99(ms)_pMax(ms)
```

Slika 10.5 *Početak rada tpcc workloada i metrike* [6]

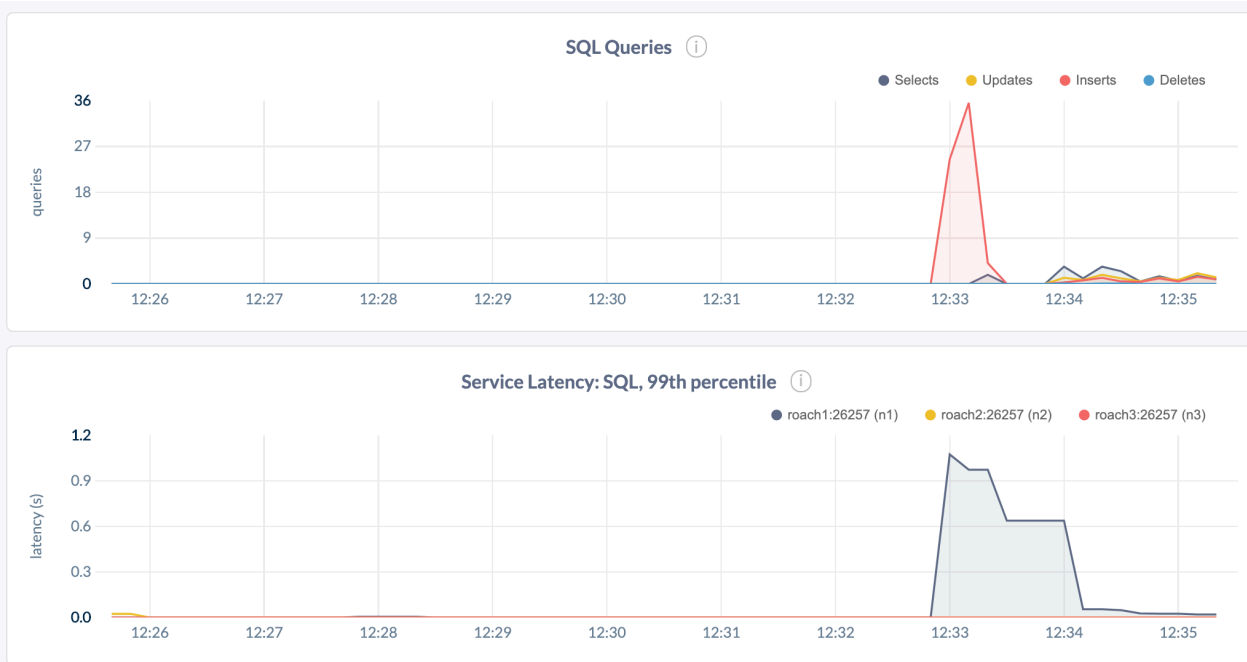
CockroachDB takodje ima dashboard posvećen praćenju svih ovih dogadjaja koji se instalira instaliranjem baze i ažurira u realnom vremenu. Dashboard je dostupan na *localhost:8080:*



Slika 10.6 *Početna strana CockroachDB dashboard-a sa prikazom infromacija o čvorovima u klasteru*

U nastavku slede slike koje demonstriraju neke od mogućnosti koje dashboard pruža. Moguće je videti broj konkurentnih upita i kašnjenje servisa.

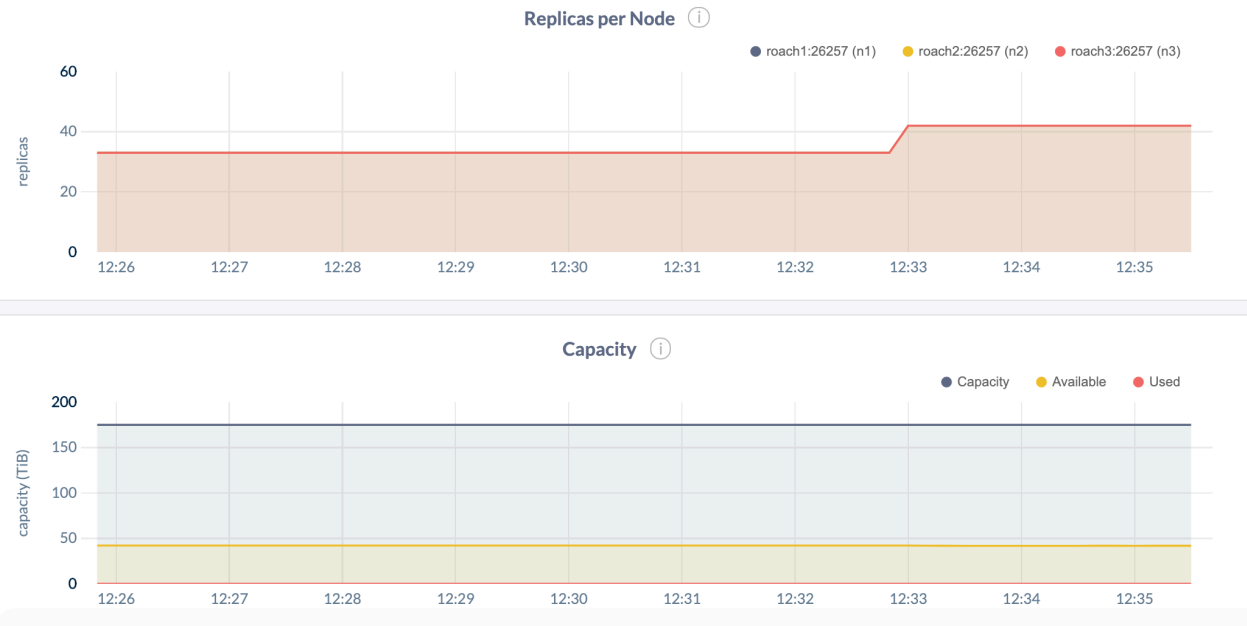
Na slici *10.7* uočavamo korelaciju izmedju broja konkurentnih upita, pogotovo upita tipa INSERT na početku workloada koji znatno povećavaju kašnjenje servisa izmedju ostalog i zbog repliciranja, dok kasnije upiti tipa SELECT, UPDATE i DELETE ne povećavaju kašnjenje u velikoj meri.



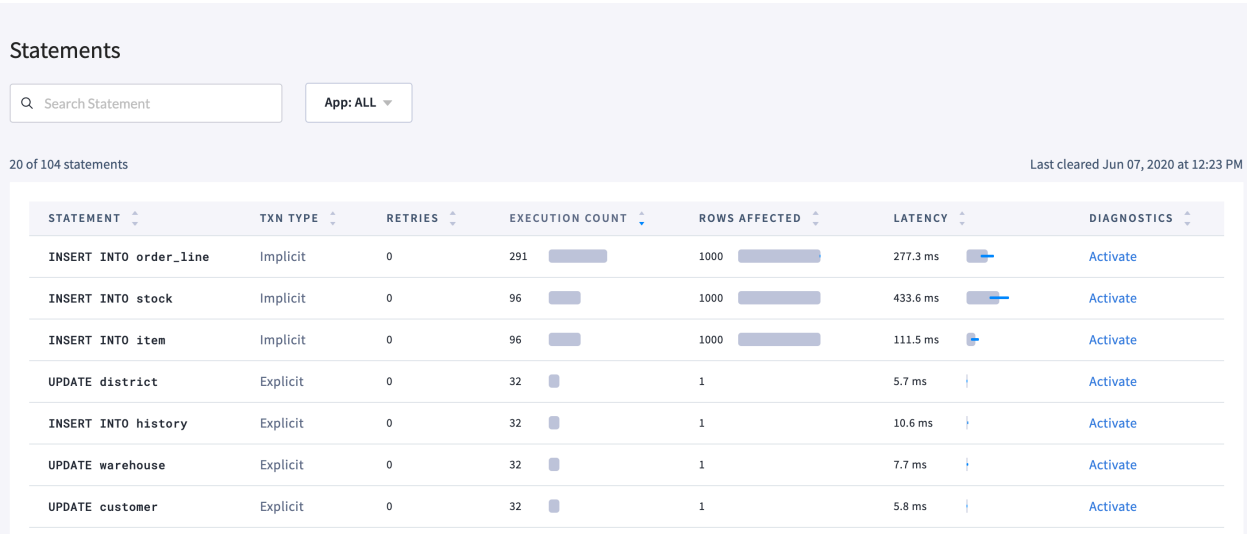
Slika 10.7 *Prikaz grafika upita i kašnjenja servisa sa podacima s početka tpcc workload-a*

Broj replika podataka za vreme rada tpcc workload-a prelazi četrdeset, što govori o izuzetno velikoj dostupnosti podataka. Sa druge strane na grafikonu ispod, na slici *10.8* vidimo da je kapacitet klastera nekoliko redova veličine veći od onog što su postojeći podaci zauzeli.

Na slici *10.9* vidimo i upite koji su izvršavani nad bazom sa detaljima njihovog izvršenja i mogućnošću ispitivanja dijagnostike koja detaljno pokazuje vreme izvršavanja svih delova upita.



Slika 10.8 *Prikaz grafika replika po čvoru i kapaciteta*



Slika 10.9 *Lista upita izvršenih tokom tpcc workload-a*

11. Zaključak

Razvoj tehnologije i ekonomije utiče na to kakvi će sistemi baza biti korišćeni. Sistemi koji imaju data warehousing, data mining, geografski i drugi informacioni sistemi čuvaju mnoštvo podataka koji trebaju biti obradjeni. Stoga javlja se potreba za distribuiranim bazama podataka, velikim jedinicama čuvanja podataka kojima se može pristupiti jednako brzo i efikasno iz raznih geografski razudjenih regiona koji su visoko dostupni i veoma skalabilni. Podaci nisu nestruktuirani i biznis modeli ovih sistema zahtevaju ACID svojstva transakcija, te distribuirane baze podataka često trebaju biti relacione.

Na početku rada obradjena su tri tipa arhitekture distribuiranih baza podataka. Da bi podaci bili efikasno smešteni na fizički odvojenim medijumima uvode se pojmovi vertikalnog i horizontalnog fragmentisanja relacija kao i njihova replikacija na jednoj ili više mašina da bi se obezbedila velika dostupnost.

Usled postojanja više replika jedne iste relacije ili nekog njenog fragmenta, čvorovi distribuiranih baza moraju održavati kataloge s jedinstvenim imenima objekata iz baze. Svaki čvor treba imati odredjene lokalne kataloge umesto jednog centralizovanog jer centralizovani predstavlja jedinstvenu tačku pada.

Da bi se upiti efikasno evaluirali i vršili u distribuiranom okruženju potrebno je voditi računa o planu upita na globalnom nivou kao i o raznim tehnikama prenošenja minimalnog broja podataka kroz mrežu. Kao rešenja kod joinovanja tabela predloženi su semijoin i bloomjoin.

Replikacija podataka može biti sinhrona i asinhrona. Danas baze mahom koriste asinhronu zato što je kašnjenje usled mrežne razudjenosti čvorova kod sinhrone preveliko.

Kao glavni protokol u distribuiranim transakcijama, detaljno je obradjen protokol dvo-faznog potvrdjivanja, njegova optimizacija sa pretpostavljenim poništavanjem najčešće se koristi pri implementaciji transakcija. Tro-fazno potvrdjivanje je pristup koji u teoriji rešava probleme sa padom koordinatora kod dvo-faznog potvrdjivanja i dugog oporavka, ali zahteva previše komunikacije izmedju čvorova koja se u praksi pokazala kao prevelika cena.

Kao primer distribuirane relacione baze obradjen je CockroachDB, baza koja se pokazala kao baza sa visokim performansama, skalabilnošću i dostupnošću. CockroachDB koristi paralelno potvrdjivanje, jedan od mehanizama zaslužnih za minimalno vreme kašnjenja kod distribuiranih transakcija koje svejedno imaju najviši nivo izolacije i Raft konsenzuz protokol za repliciranje.

U zadnjem delu demonstriran je rad CockroachDB lokalnog klastera sa tri čvora, pokrenutog putem Docker-a i podvrgnutom workload-u čiji se uticaj na bazu mogao videti kroz dashboard koji sam CockroachDB pruža.

LITERATURA

[1] Kurs *Sistemi za upravljanje bazama podataka* na Elektronskom fakultetu u Nišu - https://cs.elfak.ni.ac.rs/nastava/course/view.php?id=57

[2] „Database Management Systems“ - Raghu Ramakrishnan & Johannes Gehrke

[3] „[Notes]Distributed Database Management System, Advanced“ - https://medium.com/@harshityadav95/notes-distributed-database-management-system-advanced-database-management-systems-655036ab4c77

[4] „www.mathcs.emory.edu/~cheung/Courses/554/Syllabus/9-Parallel/semi-join1.html“ - http://www.mathcs.emory.edu/~cheung/Courses/554/Syllabus/9-parallel/semi-join1.html

[5] „What is a two phase commit“ - https://www.xenovation.com/blog/development/java/java-professional-developer/what-is-a-two-phase-commit-2pc-xa-transaction

[6] „CockroachDB Docs“ - https://www.cockroachlabs.com/docs/stable/