

Predlog skalabilne arhitekture aplikacije

Predlog strategije za particionisanje podataka

U mnogim skalabilnim aplikacijama, podaci su podeljeni u particije kojima se može nezavisno upravljati i pristupati. Particionisanje može poboljšati skalabilnost i optimizovati performanse. Takođe, pruža opciju podele podataka po sablonima koriscenja, te je zbog toga moguće arhivirati starije podatke u jeftiniji prostor.

U praksi se najcesce koristi jedna od dole navedene tri strategije za particionisanje podataka ili neka njihova kombinacija.

1)Horizontalno particionisanje(particionisanje po x-osi)

Kod ove strategije, svaka particija predstavlja zaseban prostor za skladištenje podataka, a sve particije koriste istu semu baze podataka. Svaka particija čuva određeni podskup podataka.

2)Vertikalno particionisanje(particionisanje po y-osi)

Svaka particija sadrži podskup polja podatka koji se skladišti. Polja su podeljena po frekvenciji koriscenja. Neki od načina realizacije vertikalnog particionisanja je da se polja kojima se često pristupa smeste u jednu particiju, dok se redje koriscena polja smestaju u drugu.

3)Funkcionalno particionisanje(particionisanje po z-osi)

Podaci su pridruženi po tome kako su korisceni od strane sistema. Kod klinickog centra bi se podaci koji sadrže medicinsko osoblje mogli prebaciti u jednu paprticiju, a podaci o pregledima i operacijama u drugu. Takođe, podaci bi mogli da se podele po kriterijumu *read only* i *write only*.

Ukoliko bi nas sistem dostigao broj korisnika gde baza podataka predstavlja usko grlo(*eng. Bottleneck*) sistema, onda bi bilo potrebno primeniti strategije za particionisanje podataka. Prvo bi se primenilo skaliranje po z-osi. Kada bi zahtevi prerasli ovo skaliranje bilo bi potrebno implementirati strategiju za vertikalno skaliranje gde bi se tabele particionisale. Na kraju, ukoliko kombinacija predhodno dve navedene strategije ne pruži zadovoljavajuće rezultate, iskoristilo bi se i horizontalno particionisanje. Ovo bi uvelo novi nivo kompleksnosti u celokupan sistem zbog toga što bi pri svakoj izmeni podataka bilo neophodno replicirati podatke na više servera zbog ocuvanja konzistentnosti.

Predlog strategije za replikaciju baze i obezbedjivanje otpornosti na greske

Potreba za replikacijom baze se javlja zbog cinjenice da podaci mogu da se prenose konacnom brzinom. Dakle, ukoliko se baza podataka nalazi u Evropi, upiti iz Evrope i Severne Amerike se neće izvršiti istom brzinom. Bez obzira na brzinu izvršavanja upita, ova cinjenica može mnogo usporiti rad aplikacije.

Replikacija predstavlja proces kopiranja podataka i njihovog skladištenja na više lokacija. Prvi razlog za primenu replikacije je cinjenica da podaci koji su blizu korisniku putuju krace. Drugi je taj što više servera može da opsluži veći broj korisnika, što predstavlja horizontalno skaliranje. Još jedna od prednosti replikacije baze se ogleda u cinjenici da kada su podaci smesteni na više mesta postoji *backup* na više mesta. Ukoliko jedan server prestane da funkcioniše, ostali će moći neometano da nastavljaju sa radom i sistem će biti operativan.

U našem sistemu bi primenili potpuno replikacionu semu. Prednosti koje se dobijaju potpunom replikacijom su velika dostupnost podataka, poboljšavaju se performanse upita sirom sveta i brže izvršavanje upita. Negativne strane potpuno replicirane seme su sporija azuriranja podataka i poteskoće da se ostvari konkurentnost.

Jedan od načina za rešenje problema konkurentnog pristupa je primena sinhrona replikacije. Ideja je da se podaci prvo repliciraju i izmene na svim lokacijama i tek nakon toga se šalje odgovor klijentu. Iako ovo malo usporava rad sistema, garantuje se konkurentnost pristupa i baza postaje otporna na greske pomoću optimistickog zaključavanja resursa. Ukoliko dobijene performanse idalje nisu zadovoljavajuće primenila bi se strategija *Single leader replication*. Ova strategija je bazirana na principu latice i tucak. Ukoliko se citaju podaci iz baze to je moguće uraditi iz bilo koje latice (ona koja je najbliža), dok se pisanje u bazu uvek vrši u tucku, master bazi, koja dalje replicira podatke na sve ostale latice (baze). Glavni benefit ovog pristupa je izbegavanje konflikta jer svi klijenti pišu na isti server.

Predlog strategije za kesiranje podataka

Kesiranje je jedan od najlakših načina da se poboljšaju performanse sistema. Ukoliko je primenjena odgovarajuća strategija kesiranje može smanjiti vreme čekanja na odgovor, smanjiti opterećenje baze i cene sacuvavanja podataka. Odabir strategije zavisi od podataka i sablonima pristupa podacima. Način kesiranja najviše zavisi od odnosa broja čitanja i pisanja. Pretpostavka je da će u našoj aplikaciji biti mnogo više čitanja nego pisanja jer će broj pacijenata biti veći od broja medicinskog osoblja za nekoliko redova veličine. Zbog toga bi se primenila strategija za kesiranje pod nazivom *Read-Through Cache* i *Write-Through Cache*.

Za kes se najčešće koristi *Memcached* ili *Redis*. *Read-Through Cache* stoji u istoj liniji sa bazom. Tok je takav da baza prvo proverí kes. Ukoliko se traženi podatak nalazi u kesu, dodaje se *cache hit*. Podatak je pročitán i vraća se klijentu. Kada se dogodi slučaj da podatak nije pronađen u kesu, *cache miss*, tada je neophodno obaviti dodatni posao. Vrsi se upit na bazom za tražene podatke, klijentu se vraća odgovor, a vraćeni podaci se stavljaju u kes da bi bili kesirani za sledeći takav upit. *Read-Through* strategija se koristi za aplikacije u kojima je veoma veliki broj čitanja, znatno veći od broja pisanja. Jedina mana je što ne garantuje konzistentnost podataka. Ovaj problem konzistentnosti je neophodno rešiti za pravilno funkcionisanje aplikacije. Jedno od rešenja bi bilo korišćenje *Read-Through Cache* u kombinaciji sa *Write-Through Cache*-om. *Write-Through Cache* sam po sebi ne radi mnogo, čak uvodi i dodatnu latentnost jer se podaci prvo pišu u kes pa tek onda u bazu, ali u kombinaciji sa *Read-Through Cache*-om obezbeđuje konzistentnost. *DynamoDB Accelerator(DAX)* je primer implementacije *Read-Through/Write-Through Cache*-a.

Okvirna procena za hardverske resurse za skladištenje podataka u narednih 5 godina

Postoje većina današnjih rešenja *deoply*-uje na *cloud* servise poput *Azure* i *AWS(Amazon Web Services)*, to nam pruža mogućnost da se automatizuje procena hardverskih resursa. U praksi je najbolje imati malo više resursa nego što je neophodno da bi se obezbedilo optimalno funkcionisanje aplikacije. Tri osnovne komponente kod odabira servera su CPU, memorija(RAM) i *hard drive*. Procesor se koristi za sva računanja i logiku, savet je da ukoliko je to moguće bude upotrebljen zasebni hosting. Upotreba CPU-a ne bi trebalo nikad da predje 80% jer bi u tom slučaju server mnogo usporio sa radom.

Pod pretpostavkom da će aplikaciju koristiti 200 miliona korisnika i da će svakom korisniku za smestanje podataka trebati xyz mb memorije, biće potrebno yzx memorije.

Broj zahteva za pregledima u toku jednog meseca iznosi 1 milion, što je ~30 hiljada po danu, ~1250 po satu, ~20 po minutu. Bez uzimanja u obzir peak momenata. Ukoliko bi pretpostavili da se korisnik zadržava na aplikaciji 3minuta u kontinuitetu to dovodi do ukupnog broja od 60 konkurentnih korisnika. Kada bi se pretpostavilo da postoje trenuci u danu kada su korisnici najaktivniji doslo bi se do hipotetičkog broja od 120 konkurentnih korisnika. Predlog iz prakse za ovaj broj korisnika, sa dodatim prostorom i procenom za rast je 64gb RAM memorije, Intel Xeon 2.5GHz E5 sa 6 jezgara i 12 tredova za 60 konkurentnih korisnika i dva servera sa load balanserom sa istim procesorom za 120 konkurentnih korisnika, takodje ukoliko bi za svakog korisnika bilo potrebno 0.2mb memorije za skladištenje podataka u bazi na 200 miliona korisnika celokupna potreba memorije bi bila 40tb.

Bitno je napomenuti da nije poželjno samo skalirati po X ili Y osi, već napraviti kombinaciju dva pristupa. Gde sklairanje po X-osi predstavlja broj trenutno pokrenutih instanci, a skaliranje po Y-osi predstavlja skaliranje svakog servera dodavanjem dodatne procesorske moci ili RAM memorije koju poseduje.

Predlog strategije za postavljanje load balansera

Postoji veliki broj različitih strategija za postavljanje *load balansera*. Jedan od najjednostavnijih načina je *Round Robin* tehnika. Pre svega je neophodno konfigurisati više identičnih servera koji su konfigurisani da pruže iste usluge. Svi su podešeni da koriste isti domaćin ali sa unikatnim IP adresama. DNS server ima listu svih unikatnih IP adresa koje su uparene sa željenim domenom. Kada stigne zahtev, prosledjuje se jednom od servera rotacionim principom. Problem kod *Round Robin* strategije je što se zahtevi prosledjuju tako da se ne uzima u obzir opterećenje servera. Zbog toga bi se preslo na *Least Connection* ili alternativno *Weighted Least Connection* gde se zahtev prosledjuje onom serveru koji je trenutno najmanje opterećen. *Weighted Least Connection* unapređjuje *Least Connection* strategiju tako što svakom serveru dodeljuje težinu. U slučaju da su dva servera u trenutku pristizanja zahteva isto opterećena onaj sa većom težinom će preuzeti taj zahtev i obraditi ga.

Predlog koje operacije korisnika treba nadgledati u cilju poboljšanja sistema

Operacije koje su od vitalnog značaja za funkcionisanje sistema su zakazivanje pregleda i operacija. Zbog toga bi za proveru optimalnog funkcionisanja celog sistema bilo najznačajnije posmatrati operacije kao što su definisanje brzih pregleda, zakazivanje brzih pregleda kao i zakazivanje pregleda i operacija.