Predlog skalabilne arhitketure aplikacije

Predlog strategije za particionisanje podataka

U mnogim skalabilnim aplikacijama, podaci su podeljeni u particije kojima se moze nezavisno upravljati I pristupati. Particionisanje moze poboljsati skalabilnost I optimizovati performanse. Takodje, pruza opciju podele podataka po sablonima koriscenja, te je zbog toga moguce 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 skladistenje podataka, a sve particije koriste istu semu baze podataka. Svaka particija cuva odredjeni podskup podataka.

2)Vertikalno particionisanje(particionisanje po y-osi)

Svaka particija sadrzi podskup polja podatka koji se skladisti. Polja su podeljena po frekvenciji koriscenja. Neki od nacina realizacije verikalnog particionisanja je da se polja kojima se cesce pristupa smeste u jednu particiju, dok se redje koriscena polja smestaju u drugu.

3)Funkcionalno particionisanje(particionisanje po z-osi)

Podaci su pridruzeni po tome kako su korisceni od strane sistema. Kod klinickog centra bi se podaci koji sadrze medicinsko osoblje mogli prebaciti u jednu paprticiju, a podaci o pregledima I operacijama u drugu. Takodje, 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 pruzi zadovoljavajuce rezultate, iskoristilo bi se I horizontalno particionisanje. Ovo bi uvelo novi nivo kompleksnosti u celokupan sistem zbog toga sto bi pri svakoj izmeni podataka bilo neophodno replicirati podatke na vise 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 nece izvrsiti istom brzinom. Bez obzira na brzinu izvrsavanja upita, ova cinjenica moze mnogo usporiti rad aplikacije.

Replikacija predstavlja proces kopiranja podataka I njihovog skladistenja na vise lokacija. Prvi razlog za primenu replikacije je cinjenica da podaci koji su blize korisniku putuju krace. Drugi je taj sto vise servera moze da opsluzi veci broj korisnika, sto predstavlja horizontalno skaliranje. Jos jedna od prednosti replikacije baze se ogleda u cinjenici da kada su podaci smesteni na vise mesta postoji *backup* na vise mesta. Ukoliko jedan server prestane da funkcionise, ostali ce moci neometano da nastave sa radom I sistem ce biti operativan.

U nasem sistemi bi primenili potpuno replikacionu semu. Prednosti koje se dobijaju potpunom replikaciju su velika dostupnost podataka, poboljsavaju se performanse upita sirom sveta I brze izvrsavanje upita. Negativne strane potpuno replicirane seme su sporija azuriranja podataka I poteskoce da se ostvari konkurentnost.

Jedan od nacina za resenje problema konkurentnog pristupa je primena sinhrone replikacije. Ideja je da se podaci prvo repliciraju I izmene na svim lokacijama I tek nakon toga se salje odgovor klijentu. Iako ovo malo usporava rad sistema, garantuje se konkurentnost pristupa I baza postaje otporna na greske pomocu optimistickog zakljucavanja resursa. Ukoliko dobijene performanse idalje nisu zadovoljavajuce primenila bi se strategija *Single leader replication*. Ova strategija je bazirana na principu latice I tucak. Ukoliko se citaju podaci iz baze to je moguce uraditi iz bilo koje latice(ona koja je najbliza), dok se pisanje u bazu uvek vrsi u tucku, master bazi, koja dalje replicira podatke na sve ostale latice(baze). Glavnni benefit ovog pristupa je izbegavanje konflikta jer svi klijenti pisu na isti server.

Predlog strategije za kesiranje podataka

Kesiranje je jedan od najlaksih nacina da se poboljsaju performanse sistema. Ukoliko je primenjena odgovarajuca strategija kesiranje moze smanjiti vreme cekanja na odgovor, smanjiti opterecenje baze I cene sacuvavanja podataka. Odabir strategije zavisi od podataka I sablonima pristupa podacima. Nacin kesiranja najvise zavisi od odnosa broja citanja I pisanja. Pretpostavka je da ce u nasoj aplikaciji biti mnogo vise citanja nego pisanja jer ce broj pacijenata biti veci od broja medicnisnkog osoblja za nekoliko redova velicine. Zbog toga bi se primenila strategija za kesiranje pod nazivom *Read-Through Cache* I *Write-Through Cache.*

Za kes se najcesce koristi *Memcached* ili *Redis*. *Read-Through Cache* stoji u istoj liniji sa bazom. Tok je takav da baza prvo proveri kes. Ukoliko se trazeni podatak nalazi u kesu, dodadja se *cache hit.* Podatak je procitan I vraca se klijentu. Kada se dogodi slucaj da podatak nije pronadjen u kesu, *cache miss,* tada je neophodno obaviti dodatni posao. Vrsi se upit na bazom za trazene podatke, klijentu se vraca odgovor, a vraceni podaci se stavljaju u kes da bi bili kesirani za sledeci takav upit. *Read-Through* strategija se koristi za aplikacije u kojima je veoma veliki broj citanja, znatno veci od broja pisanja. Jedina mana je sto ne garantuje konzistentnost podataka. Ovaj problem konzistentnosti je neophodno resiti za pravilno funkcionisanje aplikacije. Jedno od resenja bi bilo koriscenje *Read-Through Cache* u kombinaciji sa *Write-Through Cache*-om. *Write-Through Cache* sam po sebi ne radni mnogo, cak uvodi I dodatnu latentnost jer se podaci prvo pisu u kes pa tek onda u bazu, ali u kombinaciji sa *Read-Through Cache*-om obezbedjuje konzistentnost. *DynamoDB Accelerator(DAX)* je primer implementacije *Read-Through/Write/Through Cache*-a.

Okvirna procena za hadverske resurse za skladistenje podataka u narednih 5 godina

Posto se vecina danasnjih resenja *deoply*-uje na *cloud* servise poput *Azure* I *AWS(Amazon Web Services),* to nam pruza mogucnost da se automatizuje procena hardverskih resursa. U praksi je najbolje imati malo vise resursa nego sto 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 racunanja I logiku, savet je da ukoliko je to moguce bude upotreblenj zasebni hosting. Upotreba CPU-a ne bi trebalo nikad da predje 80% jer bi u tom slucaju server mnogo usporio sa radom.

Pod pretpostavkom da ce aplikaciju koristiti 200 miliona korisnika I da ce svakom korisniku za smestanje podataka trebati xyz mb memorije, bice potrebno yzx memorije.
Broj zahteva za pregledima u toku jednog meseca iznosi 1 milion, sto je ~30 hiljada po danu, ~1250 po satu, ~20 po minutu.Bez uzimanja u obzir peak momenata. Ukoliko bi pretposstavili da se korisnik zadrzava 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 hipotetickog 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 skladistenje podataka u bazi na 200 miliona korisnika celokupna potreba memorije bi bila 200gb.

Bitno je napomenuti da nije pozeljno samo skalirati po X ili Y osi, vec 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 razlicitih strategija za postavljanje *load balansra*. Jedan od najjednostavnijih nacina je *Round Robin* tehnika. Pre svega je neophodno konfigurisati vise identicnih server koji su konfigurisani da pruze iste usluge. Svi su podeseni da koriste isti dome ali sa unikatnim IP adresama. DNS server ima listu svih unikatnih IP adresa koje su uparene sa zeljenim domenom. Kada stigne zahtev, prosledjuje se jednom od servera rotacionim prinicipom. Problem kod Round Robin strategije je sto se zahtevi prosledjuju tako da se ne uzima u obzir opterecenje servera. Zbog toga bi se preslo na *Least Connection* ili alternativno *Weighted Least Connection* gde se zahtev prosledjuje onom serveru koji je trenutno najmanje opterecen. *Weighted Least Connection* unapredjuje *Least Connection strategiju tako sto svakom serveru dodeljuje tezinu. U slucaju da su dva servera u trenutku pristizanja zahteva isto opterecena onaj sa vecom tezinom ce preuzeti taj zahtev I obraditi ga.*

Predlog koje operacije korisnika treba nadgledati u cilju poboljsanja sistema

Operacije koje su od vitalnog znacaja za funkcionisanje sistema su zakazivanje pregleda I operacija. Zbog toga bi za proveru optimalnog funkcionisanja citavog sistema bilo najznacajnije posmatrati operacije kao sto su definisanje brzih pregleda, zakazivanje brzih pregleda kao I zakazivanje pregleda I operacija.