Predlog skalabilne arhitekture aplikacije

Uvod

San svake internet aplikacije ☺ je da bude skalabilna, odn. da može da opslužuje veliki broj korisnika istovremeno. Može se reći da je ovo veliki izazov, jer ne postoji univerzalno rešenje za postizanje ovog cilja. Pored toga što je potrebno domensko znanje, da bi se razvila skalabilna internet aplikacija očekuje se da tim bude agilan i spreman da arhitekturu svoje aplikacije oblikuje prema „potražnji“ korisnika. Na primer, naša aplikacija trenutno koristi mali broj korisnika – što znači da će odziv sistema sigurno biti pristojan (ma koliko loše naše rešenje bilo). Međutim, kako vreme prolazi, Lotus Clinic postaje uvažena i poštovana klinika i veliki broj korisnika svakodnevno pristupa našoj aplikaciji. Vrlo verovatno je da će doći do problema, jer odziv aplikacije sigurno više neće biti realizovan u nekom pristojnom vremenu. Ono što se prvo pokušava je najčešće vertikalno skaliranje, to jest unapređenje performansi jednog servera koji opslužuje korisnike našeg servisa. Međutim, sa velikim brojem koristnika, vrlo brzo ne možemo više da poboljšamo ove performanse te moramo da primenimo klasterizaciju, tj horizontalno skaliranje. Ovo znači da će korisnik i dalje da interaguje sa sistemom kao da je sistem jedinstven, ali sistem će se zapravo sastojati od više međusobno povezanih računara koji će zajedno raditi da pruže mnogo bolje performanse i biće mnogo pouzdaniji.

Dizajn šeme baze podataka

Prvi korak ka razvijanju skalabilne aplikacije je uočavanje uskih grla u performansama. Najčešće usko grlo je baza podataka. Njoj se često pristupa i potrebno je što bolje dizajnirati arhitekturu baze kako bi što bolje podržavala najčešće korišćenje zahteve.

Mi smo se u našem projektu odlučili za bazu PostgreSQL. Ona podržava indeksovanje podataka, koje bismo iskoristili za brz pristup i pored ogromne količine podataka. Za tip indeksiranja bismo iskoristili B-tree indexove (Balanced tree index), jer postižu pristup podacima u logaritamskom vremenu i dovoljno su fleksibilni da se implementiraju za sve naše najkorišćenije tabele.

Za tabelu sa korisnicima bismo uveli Multicolumn index za brzo pronalaženje putem korisničkog imena i passworda, kako bismo omogućili brzo logovanje. Za tabelu sa stavkama kalendara (koja se verovatno najviše koristi u aplikaciji) bismo uveli Multicolumn index po početnom datumu i medicinskom osoblju, kako bi brzo mogli doći do informacije da li je doktor slobodan. Ove kolone tabela su dobre za indeksiranje jer se jako retko menjaju i nikada nisu null.

Predlog strategije za particionisanje podataka

Particionisanje podataka razdvaja tabele u setove podataka. Svaki set se može modifikovati samo od strane jednog servera. Implementirali bismo particionisanje po klinikama za tabele u koje se najviše piše, konkretno, stavke kalendara za doktore, i preglede. Usko grlo je proveravanje dostupnosti doktora za neki termin, zbog potencijalnog ogromnog broja pregleda u bazi. Pacijenti bi, nakon što su odabrali doktora i kliniku, slali zahteve koje brzo možemo da obradimo jer imamo posebne particije po klinikama, a znamo kojoj klinici doktor pripada. Kombinacijom particija sa indeksima, dobili bismo odlične performanse čak i ako se jako velik broj novih zahteva i pregleda obrađuje svaki dan.

Ovakav vid particionisanja se naziva List Partitioning (particionisanje u odnosu na vrednost jedne kolone u tabeli). Da bismo ga omogućili, proširili bismo CalendarEntry sa poljem Clinic id. Appointments (pregledi) već imaju polje Clinic, pa ovde ne bismo morali ništa da menjamo u šemi baze. Appointments tabela predstavlja drugu tabelu kojoj se mnogo pristupa i čita, uz to da njoj najviše pristupaju doktori i administratori. Zbog ovoga, ima smisla da se i ona particioniše po Clinic id, posebno ako pretpostavimo da svaka klinika ima približan broj zaposlenog osoblja.

Predlog strategije za replikaciju baze i obezbeđivanje otpornosti na greške

Za repliciranje baze koristili bismo strategiju koja se zasniva na Master/Slave mehanizmu. Master je server koji prima čitanje i pisanje, dok je slave server koji je sa svakom izmenom koja se obavi na masteru obavešten i izmenjen da bude konzistentan sa masterom. Hot standby server su slave serveri koji mogu da primaju i konekcije od servera i obrađuju read-only zahteve. Ukoliko se desi problem sa master serverom, jedan od slave servera se unapređuje u master server i time se obezbeđuje visok nivo dostupnosti. Koristićemo WAL (Write Ahead Logging – log fajl u kojem se nalaze izmene koje trebaju da se primene na bazu) koje je omogućeno u PostgreSQL bazi, preko streaming mehanizma, koji podrazumeva asinhrono slanje delova WAL datoteka. Čim se izgenerišu, poslati su slave serverima i tako oni imaju visok stepen konzistentnosti. Ovim se postiže vrlo dobra otpornost na greške, jer u slučaju otkazivanja master servera, imamo log fajl koji sadrži poslednje izmene koje su bile zahtevane, i tako se brzo možemo oporaviti od greške i proglasiti slave server za novi master.

Predlog strategije za keširanje podataka

Za keširanje bismo koristili gotov servis poput Amazon ElastiCache-a ili AzureCache-a. Keširali bismo podatke za stranice koje bi pacijenti najčešće posećivati, a to su zdravstveni karton, u kojem se nalazi istorija bolesti kao i svi prepisani recepti. Takođe bismo mogli da keširamo klinike i doktore kojima se najčešće pristupa, metodom npr. most-recently-used (u keš se ubacuju podaci kojima se najčešće pristupa, i rangiraju se po broju pristupa).

Okvirna procena za harverske resurse potrebne za skladištenje svih podataka u narednih 5 godina

Gledajući veličine testnih podataka u Statistics tabu naše baze, došli smo do podataka da jedan pregled zauzima otprilike 6kB (pV), jedna stavka kalendara otprilike 4kB (sV), jedan zahtev otprilike 6kB (zV) a jedan korisnik otprilike 4kB (kV). Sa pretpostavkom da nam je broj korisnika 200 miliona (total), to je, samo za korisnike: total * kV = 800 GB, i ako pretpostavimo da svaki od njih napravi jedan zahtev nedeljno (ima 260 nedelja u 5 godina) (valjda nisu toliko jako lošeg zdravlja da moraju češće), to je: total * 260 * pV * sV * zV = 832 TB. Dakle, kada se dodaju još i dijagnoze i recepti (koji se ne menjaju tako često ali se prave joinovane tabele za svaki appointment), došli bismo sigurno do 1 PB podataka.

Predlog strategije za postavljanje load balansera

Load balanser predstavlja mehanizam kojim se teret postavljen na jedan server raspoređuje na sve servere u klasteru. On omoguće da klijenti vide jednu ulaznu tačku u sistem i da zatim rasporedi zahteve ka unutrašnjim serverima, omogućujući više performanse i veći nivo zauzetostosti svakog servera. Također

osigurava da, ukoliko dođe do otkaza jednog od servera, klijent će i dalje dobiti odgovor jer će load balanser proslediti zahtev nekom drugom serveru.

Predlažemo sledeću strategiju za load balanser – on će da radi u round-robin režimu, što podrazumeva da će prvi zahtev da se prosledi prvom serveru, drugi zahtev sledećem serveru, i tako u krug, dok ne dođemo opet do prvog zahteva. Ovo rešenje je prihvatljivo za naš scenario jer nemamo rukovanje sa velikom količinom podataka u sesijama (nemamo slike ili video sadržaj na klinici, i kada se zatraže liste omogućena je paginacija za sav sadržaj kojem se često pristupa, te količina podataka u transferu nije velika), te nije strašno da svaki server opslužuje veći broj sesija.

Prelog koje operacije korisnika treba nadgledati u cilju poboljšanja sistema

Za poboljšanje sistema posmatraćemo broj zahteva za preglede i operacije u cilju ocene performansi našeg sistema. Takođe nam je važno da pratimo broj novih korisnika da bismo bolje mogli da predividimo potrebne količine memorije za skladištenja podataka u budućnosti, kao i da bismo mogli da unapredimo delove sistema koje smo prevideli i koji se takođe mnogo koriste. Za ocenjivanje pregleda mogli bismo dodati i način da se dostavi feedback o zadovoljnosti korisnika sa sistemom, kao i bilo kojih problema do kojih je možda došlo u korišćenju istog.

Skica dizajna predložene arhitekture