Proof of Concept

Namjena sisema

Sistem je namjenjen za podršku rada kliničkog centra koji može da ima više klinika. Svaka klinika ima doktore koji su određenih specijalnosti. Korisnici koji su se registrovali i čija je registracija prihvaćena od strane administratora klinike su pacijenti koji mogu online da zakazuju preglede. Takođe, doktori tokom pregleda mogu da zakazuju naredni pregled ili operaciju za pacijente koje pregledaju.

Ovaj sistem, takođe, omogućava radnicima (doktorima i medicinarima) da šalju zahtijeve za godišnji odmor koje prihvaćaju ili odbijaju administratori klinike.

Takođe administratori klinike imaju uvid u prihode klinike u željenom vremenskom periodu, kako numerički tako i grafički.

Particionisanje podataka

Cilj samog particionisanja je razbiti ekstremno velike tabele na manje i to na dobro osmišljen način kako bismo omogućili planeru upita da pristupi podacima što je brže moguće, mnogo brže nego što bi to bilo u originalnoj tabeli. PostgreSQL podržava tzv. *deklarativnu particiju*. To je lakši način za podešavanje particija, tako što se za svaki unijeti red u tabelu biraju kolone čiji će atributi ulaziti u sastav ključa za particiju u novoj tabeli.

Prijedlog particionisanja u ovom sistemu sveo bi se na particionisanje po UserID. Možemo da čuvamo podatke jednog korisnika na jednom serveru i UserID se može proslijediti hash funkciji koja će odrediti koji server baze podataka će čuvati podatke o istoriji pregleda, zakazanih pregleda,..

Kada hoćemo da učitamo podatke za korisnika, hash funkcija će reći odakle je potrebno čitati te podatke.

Replikacija i otpornosti na greške

PostgreSQL uključuje ugrađenu binarnu replikaciju koja se zasniva na slanju promjena u *slave* servere. To omogućava efikasno dijeljenje prometa za čitanje na više čvorova.

Pošto je sistem pretežno okrenut ka čitanju podataka, možemo imati više sekundarnih servera koji će služiti samo za čitanje podataka. Svako pisanje će ići na primarni server baze podataka i repliciraće se na sekundarne, a kada primarni server nije dostupan možemo izabrati neki od sekundarnih servera za novi primarni server (odratiti failover).

Svaka instanca server PostgreSQL-a se sastoji od jednog postmaster server procesa koji je povezan sa inicijalizovanim PostgreSQL direktorijumom podataka koji sadrži nekoliko baza podataka. Svaki trenutno aktivan server ima transakcijski log koji se sastoji od binarnih zapisa podataka koji se upisuju svaki put kada se desi neka promjena, a taj log se koristi usljed neočekivanog gašenja servera baze podataka.

Dostupno je nekoliko asinhronih paketa za replikaciju na bazi okidača. One ostaju korisne čak i nakon uvođenja proširenih jezgrenih sposobnosti, za situacije u kojima je binarna replikacija cjelokupne baze podataka neprimjerena:

- Slony-I
- Londiste, dio of SkyTools (developed by Skype)
- Bucardo multi-master replication (developed by Backcountry.com)[30]
- SymmetricDS multi-master, multi-tier replication

Keširanje

Kada se podnese zahtjev vraćeni podaci mogu se sačuvati u kešu na takav način da im se u budućnosti lakše/brže pristupa. Upit će u početku pokušati keš memorija, ali ako ne uspije proslijeđuje se bazi.

Pretpostavimo da svaki pojedinačni korisnik u našoj aplikaciji ima statiči skup podataka koji mora prenijeti svaki put kada pređe na novu stranicu – počevši od sloja podataka i završavajući na prezentacijskom sloju. Ako se u tom slučaju sloj podataka stalno ispituje, to dovodi do velikog naprezanja i lošeg korisničkog iskustva uzrokovanog kašnjenjem. Uvođenjem keš memorije, podaci kojima se često pristupa mogu se sačuvati u pripremenoj memoriji, omogućujući tako da se brzo servira u prezentacijski sloj.

Keširanje izvan procesa: predmet stavljamo u memoriju aplikacije (heap) ili u memoriju koja nije u našoj aplikaciji. Prednosti ovog keširanja su jer se memorija keša ne dijeli sa memorijom naših servera, nismo ograničeni na memoriju servisiranja, imaćemo jedno mjesto ili klaster za čitanje i upisivanje podataka iz i u keš memoriju. Dakle, dijelimo keš memoriju između naših servera a i keš se ne briše kada se aplikacija ponovo pokreće.

Možemo koristiti gotova rješenja poput Memcached ili Redis da čuvamo cijele objekte. Ako je keš pun, želimo da zamijenimo stare preglede sa novijim. Možemo koristiti Leat Recently Used (LRU) strategiju. **Ovom strategijom prvo odbacujemo preglede koji su se ranije obavili**. Podatke možemo keširati na samo jednom serveru ali takođe podatke možemo replicirati na više servera kako bismo distribuirali saobraćaj bolje i izbjegli opterećenje samo jednog keš servera.

Druga strategija bi se odnosila na parametre pretrage pregleda (princip Leat Frequently Used) gdje bi se u kešu čuvale najćešći birani tipovi pregleda ali i doktori sa najvećim brojem pregleda na sedmičnom nivou.

Okvirna procjena za hardverske resurse

Pretpostavka je da imamo 200 miliona korisnika sa milion pregleda u toku jednog mjeseca i da izveštaj nekog pregleda u prosjeku ima 500 karaktera. (potrebna su nam dva bajta da sačuvamo karakter bez kompresije).

Ukupno za skadištenje treba:

Potrebno je 40 bajta za čuvanje metapodataka za svaki pregled u kome se nalazi izvještaj (npr. userID, checkupID, datum i vrijeme pregleda, kao i soba, klinika i doktor).

Mjesečni nivo:

- 1 000 000 pregleda/operacija * (2 bajta * 500 karaktera + 40 bajta) = 0.96858 GB/mjesec

U toku pet godina za skladištenje je potrebno:

- Još prostora za korisnikove osnovne podatke (lični podaci, ime, prezime, adresa,..), radni kalendar za doktore, zdravstveni kartoni za pacijente, informacije o klinikama,..
- Za dijagnoze i lijekove: pretpostavka da ima ukupno 1000 dijagnoza * 100 karaktera * 2 bajta i 10000 lijekova * 100 karaktera * 2 bajta = 0,2 GB
- Za svakog korisnika objezbjeđujemo 1000 karaktera * 2 bajta = 2000 bajta.

- 0.96858 GB/mjesecu * 12 mjeseci * 5 godina + 200 miliona korisnika*2000 bajta = = 430,84356 GB

Ako po mjesecu imamo milion pregleda to je dolazni saobraćaj od 0,16MB/sekunda.

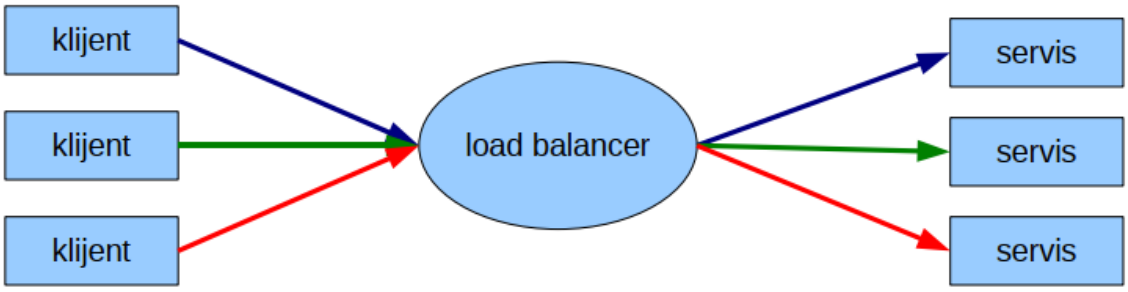
Prijedlog strategije postavljanja load balancer-a

Load balancer (podjela opterećenja) je uređaj kome je posao da prima http zahtijeve na svoju javnu adresu i da ih proslijedi na neki od servera, balansira opterećenje servera kako bi svi serveri bili jednako opterećeni, otkriva kada je neki server prestao sa radom, idealno je da nijedan korisnik ne osjeti da neki server ne radi. Load balancer prosljeđuje zahtijeve serveru koji je ispravan.

Load balancer predstavlja odličan način skaliranja aplikacije i povećavanje performansi. Ideja je da postoji više servera koji hostuju aplikaciju i da postoji jedan raspoređivač koji će, u zavisnoti od opterećenosti servera da prosljeđuje zahtijeve. Korisnika aplikacije ne zanima kako to funkcioniše i ne smije postojati razlika u izvršavanju zahtijeva u odnosu na server na koji se poziv prosljeđuje.

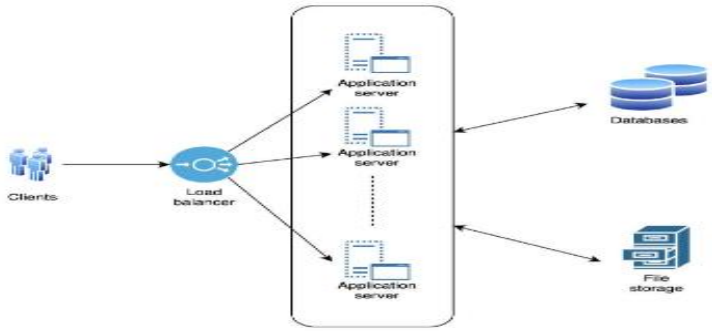
Ako jedan server ne može da zadovolji potrebe svih klijenata, možemo da kupimo veći server ili da kupimo još običnih malih servera ali onda moramo da ih dodamo u klaster kako bi radili zajedno. Ako imamo jedan server – single point, kada dođe do greške kod njega pokvari se i ništa ne radi, njegova cijena ne raste linearno i jaku su skupi. Ako imamo više manjih svaki server bi mogao da opsluži više korisnika odjednom, cijena bi rasla linearno, dodavanje propusne moći u ovaj sistem bi koštao mnogo manje nego u varijanti sa jednim serverom. Problem sa više servera je što traba da omogućimo i dobro organizujemo saradnju svih servera zajedno.

Round Robin je jedan od jednostavnijih koji zahtjev usmjerava na prvi sljedeći server u rotaciji, a ako neki od servera nije dostupan, izbaciće ga iz razmatranja pri usmjeravanju zahtijeva. Problem koji se može javiti je ako se neki server preoptereti ili ako je spor, load balancer neće prestati da šalje zahtjeve na njega pa neka naprednija strategija raspodjele zahtjeva više opcija.



Radi povećanja performansi i pouzdanosti jedan od koraka u toku skaliranja bio bi klasterovanje servisa. Za povećanje pouzdanosti bilo bi neophodno uvesti redundante servise, tako da se izbegne Single Point of Failure problem. Smatramo da je ovakav korak neophodan, s obzirom da aplikacija rukuje sa informacijama značajnim za korisnika. Za povećanje performansi bilo bi neophodno uvesti redundante servise i load balancer-a koji raspoređuje korisničke zahtijeve tako da ravnomjerno optereti servise.

Jedan od prvih koraka koji bi se morali primjeniti u slučaju povećanja broja korisnika je regulacija upotrebe vanjskih servisa. Da bi se aplikacija skalirala za upotrebu od strane više korisnika bilo bi neophodno izvršiti zamjenu trenutno upotrebljenih besplatnih email naloga, mapa i geodecodera, baze podataka i svih ostalih vanjskih servisa sa plaćenim verzijama.



Skalabilnost

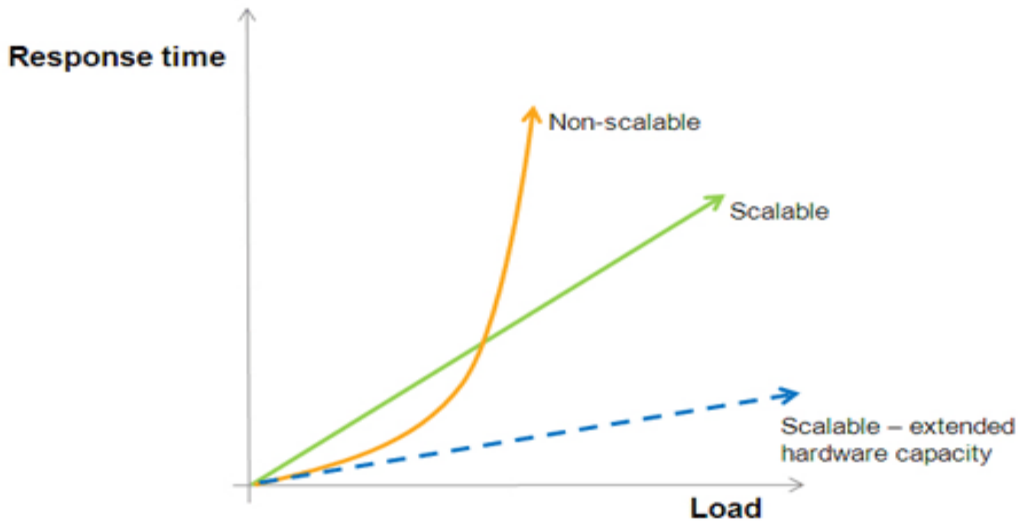
Skalabilnost je mogućnost aplikacije da podnese povećanje zahtjeva I broja korisnika a da sama aplikacija ne mora da se mijenja. Što je aplikacija skalabilnija ona će lakše podnijeti povećan protok podataka. Cilj kojim teže svi projektanti sistema jested a se postigne linearnosti u brzini odgovora na zahtjev I količine podataka kojima se manipuliše.

Baza koju smo koristili je PostgreSQL, besplatan i open-source sistem za upravljanje ralacijskim bazama podataka koji naglašava proširivost i usklađenost sa tehničkim standardima. Razvoj Interneta i socijalnih mreža koje okupljaju ogroman broj korisnika, među kojima prednjači Facebook sa preko 500 miliona korisnika, uticao je da se potraže alternativna i inovativna rješenja. Sam Facebook je doprinio razvoju i unapređenju InnoDB endžina, ali kako bi postigli skalabilnost morali su da pokrenu razvoj sopstvenih rješenja (Cassandra).

Kada web aplikacija dođe do stadijuma da povećan broj podataka sa kojima se manipuliše utiče na brzinu odgovora na zahtev, tj na učitavanje stranica, možemo reagovati na više načina:

1. Uložiti gomilu novca u kupovinu hardvera koji će moći da se nosi sa novonastalom situacijom.
2. Misliti na vrijeme i dizajnirati samu aplikaciju tako da bude skalabilna, a to ćemo postići tako što ćemo na probleme odgovarati rješenjima koja podižu performanse. Ne postoji univerzalan odgovor već svaki scenario i svaka situacija zahtijevaju posebno rješenje.

Ukoliko sama aplikacija nije skalabila, treba pronaći usko grlo i na za njega odgovarajuće rješenje. U relacionim bazama podataka čest odgovor na probleme jeste denormalizacija



Vertikalna skalabilnost je kada se je aplikacija smještena na jedom serveru, a na povećan protok reagujemo tako što serveru dodajemo memoriju, jači procesor, nova jezgra ili dodatni hard disk.

Horizontalna skalabilnost je idealnije rešenje, posebno za velike sisteme. Dodavanjem novih čvorova sistem nastavlja da radi kao do sada samo sa novim igračem(čvorom) u timu. Čvor predstavlja jedan server.

Za veliki saobracaj podataka relacione baze podataka ne daju baš najbolje performanse i ne mogu da održe tempo skaliranja kakav može da izdrži Spring dio aplikacije. U slučaju velikog porasta broja korisnika bilo bi poželjno preći na neku distribuiranu bazu poput Cassandra ili Redis što bi vjerovatno zahtijevalo restruktuiranje modela podataka. Prelaskom na Cassandru dobili bismo na skalabilnosti i robusnosti. Međutim, u trenutnom modelu su iskorišteni vjestacki ključevi koji u određenoj mjeri poboljašavju performanse realacione baze u odnosu na upotrebu prirodnih ključeva.

Cassandra predstavlja projekat Apache korporacije, napisan u Java programskom jeziku pod Apache 2.0 licencom koja se može smatrati open source licencom. Cassandra se koristi u kompanijama kao što su Netflix, eBay, Twitter, Cisco i raznim drugim koje imaju velike skupove podataka. Najveći klaster gde je Cassandra u upotrebi sadrži preko 300 TB podataka na preko 400 mašina. Replikacija podataka je automatska na više različitih mesta radi potpune kontrole nad greškama. U slučaju pada nekog čvora, on može biti zamenjen bez prekida rada. **Cassandra je pogodna za aplikacije koje ne dozvoljavaju gubitak podataka čak i kad ceo centar podataka iz nekog razloga prekine sa radom**. Nudi kontrolu nad sinhronom ili asinhronom replikacijom za svaku operaciju ažuriranja. Propusnost čitanja i pisanja raste linearno sa dodavanjem novih mašina, bez prekida ili usporavanja rada aplikacija.

Rest servisi

Za autentifikaciju korisnika nakon prvobitne provere njihovog korisničkog imena i šifre, koristili smo JWT tokene, koji su u svakom pozivu ka nasem web serveru bili poslati u vidu Autorizacionog header-a.

JWT (JSON Web Token) je javno dostupni industrijski standardizovani način za dijeljenje klejmova (eng. claims) bezbjedno između neka dva entiteta. Podatke koji se nalaze unutar tih klejmova je moguće verifkovati da li su nepromjenjeni jer postoji potpis koji je pridružen svakom tokenu. Potpis se generiše korišćenjem nekog hashing algoritma, podataka iz claimova i ključa koji je dostupan samo serveru. Na ovaj način, kada JWT token stigne od klijenta, server može ponovo da proba da kreira potpis, i ako se potpisi podudaraju, znamo da je validnost i integritet tokena nepromjenjena i možemo vjerovati podacima unutar klejmova. U svojoj kompaktnoj formi JWT token se sastoji iz tri dela razdvojenim tačkama:

- Header
- Payload
- Signature

Prijedlozi poboljšanja sistema

- Aplikacija bi se mogla poboljšati dodavanjem paging-a na odgovarajuća mjesta ako bi sistem imao veliki broj korisnika, ovo je svakako neophodno zbog prostora na kome se nalaze komponente ali i zbog lakšeg i bržeg dobavljanja podataka iz baze.
- Sređivanje koda i optimizacija. Greške koje se mogu lako ukloniti u kodu mogu kasnije koštati mnogo resursa.
- Upotreba upita nad bazom, a što manje proceduralno da rešavanje problema. Potrebno je i omogućiti keširanje čestih upita poput zakazivanja pregleda

- Još jedna mogućnost bi bila da se pacijenti podijele u pulove i da se tako raspoređuju resursi servera. Prilikom zakazivanja pregleda te preglede bismo stavili u redove čekanja u slučaju preopterećenja servera.
- Zbog velike količine mejlova koje bi trebali da šaljemo sa 2000.000.000 korisnika, neophodno je implementirati sopstveni smtp server.
- Način na koji trenutno predstavljamo lokacije pomoću Google mapa (developer mode) je još jedno ograničenje koje bi mogli da zaobiđemo pretplatom na Google Maps.
- Uvođenjem Dynamic Content Caching-a smanjila bi se dodatna komunikacija sa serverom.
- Relacione baze nisu najbolje rešenje za horizontalno skaliranje aplikacija, umesto njih bi prešli na NoSQL baze kao što su MongoDB. MongoDB baza podataka je veoma povoljna za horizontalno skaliranje uz pomoć Sharding mehanizma. Sharding je način distribucije podataka na više servera tako što se baza aplikacije podeli na shard-ove i korisniku se omogući pristup pomoću mongosa.
- Što se tiče same baze, uveli bi indekse za najčešće tražene stvari poput tipova pregleda, datuma, doktora itd. Pored toga, keširanje čestih upita upita, kao što su zakazivanje brzih pregleda, bi takođe poboljšalo performanse aplikacije.
- Način na koji možemo da poboljšamo aplikaciju je upotreba message queue-a. To je način komunikacije između aplikacija ili komponenti slanjem poruka. Najbolju primjenu imaju kod asinhrone obrade velike količine podataka.

Realizovane transakcije objašnjenje su u posebnom dokumentu.

Dizajn predložene arhitekture