Proof of Concept

Tim 3

June 14, 2020

1 Dizajn šeme baze podataka

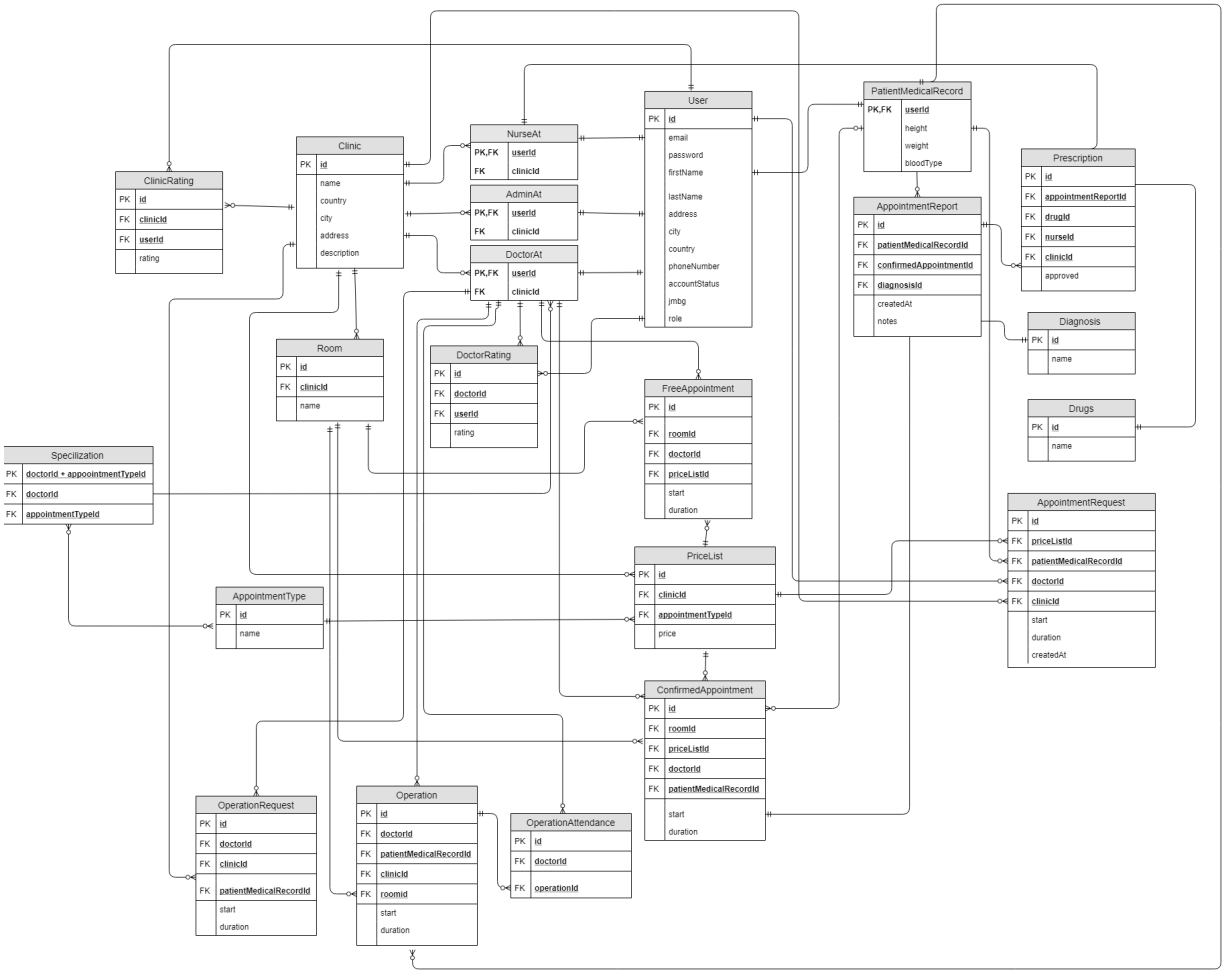

Figure 1: Šema baze podataka

2 Particionisanje podataka

U sistemi sa velikim količinama podataka, podaci se dele u particije. Ukoliko se dobro iskoristi particionisanje može povećati performanse, skalabilnost, kao i dostupnost sistema. Dve strategije particionisanje koje predlažemo su:

- **Horizontalno particionisanje** - particija sadrži jedan deo redova iz tabele
- **Vertikalno particionisanje** - particija sadrži jedan deo polja iz redova tabele

2.1 Particionisanje pojedinačnih tabela

2.1.1 Klinike

Klinika predstavlja možda najvažniji entitet u ovom sistemu, ali pošto ne očekujemo preveliki broj klinika nema potrebe za njihovim particionisanjem. Trenutno najveći problem specificirane baze na slici 1 je prevelika centralizovanost. Tabele su zajedničke svim klinikama, iako je udeo podataka jedne klinike u svim podacima izuzetno mali. Klinike rade samo sa svojim podacima, i iako bi particionisanje tabela prema identifikatoru klinike pomoglo, performanse i dalje ne bi bile optimalne. Idealno rešenje bi bilo kada bismo imali ovakav sistem enkapsuliran u svaku od klinika. Takav sada decentralizovan sistem bi odlikovale klinike koju rukuju mnogo manjim brojem samo svojih podataka.

Međutim, negativna strana ovog pristupa bi bila smanjena dimaničnost sistema, jer više dodavanje klinike ne predstavlja samo jedan insert u tabelu sa svim klinikama. Rast bi takođe bio usporen jer bi trebalo određeno vreme i resursi da se doda nova klinika, ali svakako smatram da bi ovo bio ispravan pristup za jednu ozbiljnu world-wide platformu.

2.1.2 Korisnici

Sa 200 miliona redova, ova tabela bi mogla predstavljati ozbiljan problem propusnosti sistema, te se njenom particionisanju treba posvetiti dosta vremena i analize. Prva očigledna stvar je to što nas pri upitima nad ovom tabelom skoro nikada ne zanimaju svi korisnici, već samo jedan određeni tip korisnika. Mislim da je situacija kada nam trebaju i doktori i pacijenti prilično retka, te bi dobra ideja bila horizontalno parcionisati tabelu prema ulozi korisnika.

Od svih korisnika aplikacije, sa ovom tabelom najviše interaguju administratori klinike, i razumno je pretpostaviti da će većina korisnika u njihovim upitima imati isto ili slično mesto/zemlju porekla pošto su klinike fizičke lokacije u koje dolaze pacijenti i medicinski radnici. Shodno tome, korisnici se dalje mogu particionisati i po zemlji porekla tako da bi ovi upiti sada radili sa još manjim brojem korisnika.

2.1.3 Pregledi i operacije

Trenutno, tabele sa pregledima i operacijama sadrže redove budućih kao i završenih operacija i pregleda. Očigledno je da bi sa vremenom ove tabele bile prepunjene završenim pregledima naspram onih nezavršenih. To se može rešiti horizontalnim particionisanjem, ili podelom na dve tabele.

2.1.4 Ostale tabele

Sve ostale tabele su direktno povezane sa gore navedenim, te bi za njih sve analogno važi.

3 Replikacija baze i otpornost na greške

Predlaže se inkrementalna replikacija bazirana na logovima sistema za upravljanje bazom podataka (eng. log-based incremental replication). Struktura tabela u relacionoj bazi podataka korišćenoj u ovom sistemu se neće često menjati i možemo bezbedno zaključiti da je dovoljno statična da opravda ovaj pristup.

Dakle, svaka serverska mašina mora da poseduje istu kopiju podataka, tj mora da se očuva konzistentnost na nivou čitavog distribuiranog sistema. Predložena strategija implicira da čvor u mreži propagira izmene ostalim čvorovima samo kada se one dese. Na ovaj način mreža nije bespotrebno zagušena propagiranjem podataka koji nisu menjani. Takođe, važno je napomenuti da se propagiraju operacije (upiti) potrebni da se naprave izmene, a ne sami podaci. Predlaže se da se izmene čuvaju u neki lokalni bafer i propagiraju kroz mrežu kada pređu maksimalan broj ili kada istekne vremenski rok. Ove parametre je najbolje podešavati empirijskim putem i dinamički ažurirati tokom produkcije poštujući skaliranje. Predlaže se odgovarajući balans između frekvencije propagiranja izmena i minimizacije zagušenja, tako da sistem pod opterećenjem ne uvećava isto pukom sinhronizacijom podataka.

Predlaže se primena celokupne replikacije baze podataka (eng. full database replication), tj kopiranje svih tabela i redova u bazi. Kako se radi o medicinskom sektoru ovaj sistem može da se smatra kritičnim, te je potrebno čuvati više kopija svih podataka na različitim geografskim lokacijama kako bi podaci mogli da prežive potpun otkaz bilo kojeg broja čvorova bilo gde, dokle god je funkcionalan barem jedan čvor distribuiranog sistema servera. Kao bezbednosna mera prilikom propagacije se predlaže korišćenje postojeće kolone version. Čvor u mreži ne treba da primeni operacije koje rade nad podacima starije verzije nego što on trenutno poseduje. Na ovaj način bi se sprečila dupla sinhronizacija.

4 Keširanje podataka

Predlaže se da keš memorija serverskog čvora sadrži samo redove koji su nedavno traženi od baze. Preciznije, zahtev koji namerava da čita prvo proverava da li se traženi red nalazi u kešu i komunicira s bazom samo ako ga nije pronašao. Zahtev koji namerava da piše čini to nad keš memorijom koja se kasnije sinhronizuje s bazom. Engleski termini za ove strategije su Read through / Lazy loading i Write through.

Keš memorija se sinhronizuje s bazom u redovom intervalu koji je potrebno odrediti empirijskim putem i redovno štelovati u toku produkcije za najbolje rezultate.

Memorija čvora u distribuiranom sistemu je konačna, te je potrebno odlučiti se za strategiju iseljenja podataka (eng. Eviction). Predlaže se kombinacija strategije najmanje nedavno korišćenog (eng. Least recently used) i strategije najređe korišćenog (eng. Least frequently used). Dakle, kada nestane memorije, primenom ovog pristupa se prvo izbacuju redovi koji su ređe korišćeni u poslednje vreme (recimo u poslednjih 7 dana), ali se ipak poštuje ukupan broj njihovog korišćenja. Na ovaj način dobija veće šanse red koji se nije koristio u poslednje vreme, ali se generalno često koristi. Odnos ove dve metode u rangiranju nekog reda je potrebno pronaći empirijskim putem i redovno štelovati u toku produkcije za najbolje rezultate.

Što se tiče keširanja na klijentskoj strani predlaže se čuvanje statičnih podataka preko više korisničkih sesija. Misli se na podatke koje samo može taj određeni klijent da promeni na nivou celog sistema te ih je potrebno sinhronizovati s bazom samo onda kada to i učini. Primer ovakvih podataka su lični podaci svakog korisnika kao što su ime, prezime, datum rođenja... Kako je razmatrani broj korisnika 200 miliona, samo čuvanje ličnih podataka na lokalnom nivou znatno oslobađa opterećenje cele mreže.

5 Okvirna procena hardverskih resursa

CPU: Tehnologija Node.js korišćena za realizaciju ovog sistema je jednonitna i kao takva ne profitira od umnožavanja procesorskih jezgara. Takođe, zahtevi sistema ne nameću procesorski intenzivne operacije te se može sa sigurnošću reći da procesor nije odlučujuća komponenta hardverske potpore sistema Covid19Clinic. Međutim, iako je Node.js jednonitna aplikacija ništa nas ne sprečava da pokrenemo više instanci iste na jednoj mašini.

RAM: Sistem rukuje s velikim brojem podataka ako se uzme u obzir da treba da opsluži 200 miliona korisnika. Ovakvo opterećenje može znatno da olakša keš memorija kojoj je potrebno prostora da diše. Predlaže se fokusiranje na ovu komponentu sistema jer će znatno doprineti brzini rada.

HDD: Kako su podaci od kritične važnosti, izbor kvalitetne, brze i dugotrajne masovne memorije treba da bude imperativ. Ipak, blagi primat i dalje treba da ima RAM. Imajući ovo u vidu, uzmimo praktičan primer. Jedan sistem izgrađen u Node.js tehnologiji pre 5 godina za upravljanje fakultetom je uspešno i zadovoljavajuće brzo podneo 500.000 studenata tokom ispitnog roka koji proveravaju rezultate svojih ispita.

Ovaj sistem je radio na Prescott 1.8GHz procesoru s 3GB RAM memorije. Ako modernizujemo ovu specifikaciju i budemo na skromnijoj strani, možemo postaviti početnu liniju na:
CPU: 2.4GHz, 8 jezgara
RAM: 128GB DDR3 266Mhz, 17GB/s
HDD: 10TB
Broj Node.js instanci: 8

Ako verujemo praktičnom primeru jedna Node.js instanca na 1.8GHz podnosi 500.000 približno istovremenih zahteva, te bi u našem slučaju podnela oko 650.000 zahteva. Dakle, za 8 instanci to iznosi 5.200.000 zahteva. Olakšavajuća okolnost u ovom sistemu predstavlja nedostatak velikih binarnih datoteka. Svi podaci su tekstualni, te ne zauzimaju mnogo mesta u memoriji. 128GB RAM memorije bi trebalo da bude sasvim dovoljno da opsluži 5 miliona zahteva ako pretpostavimo da svaki od njih neće rukovati sa više od 25kB podataka u jednom zahtevu. Jednostavnom aritmetikom zaključujemo da nam je potrebno 40 ovakvih servera za opsluživanje 200 miliona korisnika. Dakle, 40 ovakvih servera ukupno iziskuje:
CPU: 768GHz
RAM: 5120GB
HDD: 400TB
Broj Node.js instanci: 320
Procenjujem da su ove specifikacije uvećane za 10% radi dodatnog lufta sasvim dovoljne za narednih 5 godina funkcionisanja sistema.

6 Load balancer

Ukoliko naša aplikacija ima 200 miliona korisnika, možemo pretpostaviti da imamo korisnike širom sveta. Takođe, naše potrebe su odavno prevazišle performanse jednog servera, i sada se serverska infrastruktura sastoji od klastera servera koji su geografski raširenи tako da minimizuju put koji svaki zahtev treba da predje. Kada korisnik uputi zahtev ka serveru, on se preusmerava na njemu najbliži serverski klaster. Zadatak load balancera je da taj zahtev dalje preusmeri na pojedinačan server koji će ga obraditi, tako da opterećenje svih pojedinačnih servera bude što sličnije.

Za razliku od nekih tehnologija kao što je Java Spark, NodeJS sa JWT u svojoj memoriji ne čuva nikakve podatke o korisničkoj sesiji. Zahtevi se obradjuju dekodovanjem JWT tokena, što znači da različiti serveri ukoliko poseduju isti secret mogu dekodovati bilo koji korisnički token i obraditi njegov zahtev. To nas može dovesti do naivnog zaključka da je najbolji način za rasporedjivanje zahteva na servere heš funkcija koja mapira token na prirodni broj, čiji bi ostatak pri deljenju sa ukupnim brojem servera n predstavljao redni broj servera na koji šaljemo zahtev.

Problem kod ovog pristupa je to što pri dodavanju ili uklanjanju servera, što je prilično čest događaj zbog mogućih otkaza kao i potreba skaliranja, dolazi do velikih promena u mapiranju. Značajan deo zahteva će se sada mapirati na drugi server, što uzrokuje invalidiranje keš memorije i pad performansi.

Tako da ono što želimo je mapiranje zahteva na isti fizički server uz minimalne promene pri dodavanju ili uklanjanju servera iz klastera. To se može postići konzistentnim heširanjem.

6.1 Konzistentno heširanje

Implementacija konzistentnog heširanja se zasniva na prstenu sa $K >> n$ lokacija. Prvo će se svaki od n server mapirati na jednu od K lokacija. Zatim, svaki zahtev koji stigne se takodje preslikava na jednu od ovih lokacija, i server koji će obraditi zahtev će biti najbliži clock-wise server preslikanoj lokaciji (slika 2).

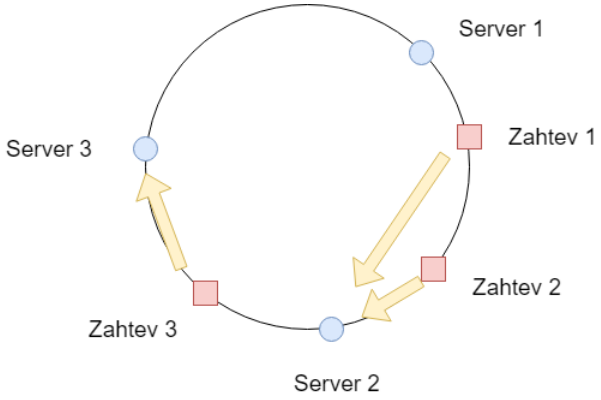

Figure 2: Konzistentno hešovanje

Sada će se pri dodavanju ili oduzimanju servera menjati mapiranja samo njemu susednog servera u prstenu, i tako omogućiti našem sistemu da efikasno koristi keširanje podataka kao i da se dinamično skalira u zavisnosti sa trenutnim potrebama.

7 Analiza korisničkih akcijia

Navedene strategije keširanja, particionisanja itd. se dalje mogu optimizovani analizom korisnika sistema. Na velikoj količini podataka se lako mogu uočiti obrasci koji nam mogu pomoći u tome. Neke od ovih akcija bi bile:

- analiza podataka koji se najviše dobavljaju iz baze sa ciljem povećanja keširanja datih podataka
- analiza podataka koji se najmanje dobavljaju koji su kandidati za vertikalno skaliranje ili smeštanje na jeftinije servere
- analiza lokacija odakle zahtevi dolaze radi blagovremenog geografskog skaliranja
- identifikacija vremenskih perioda kada su serveri najviše i najmanje opterećeni

8 Dizajn arhitekture

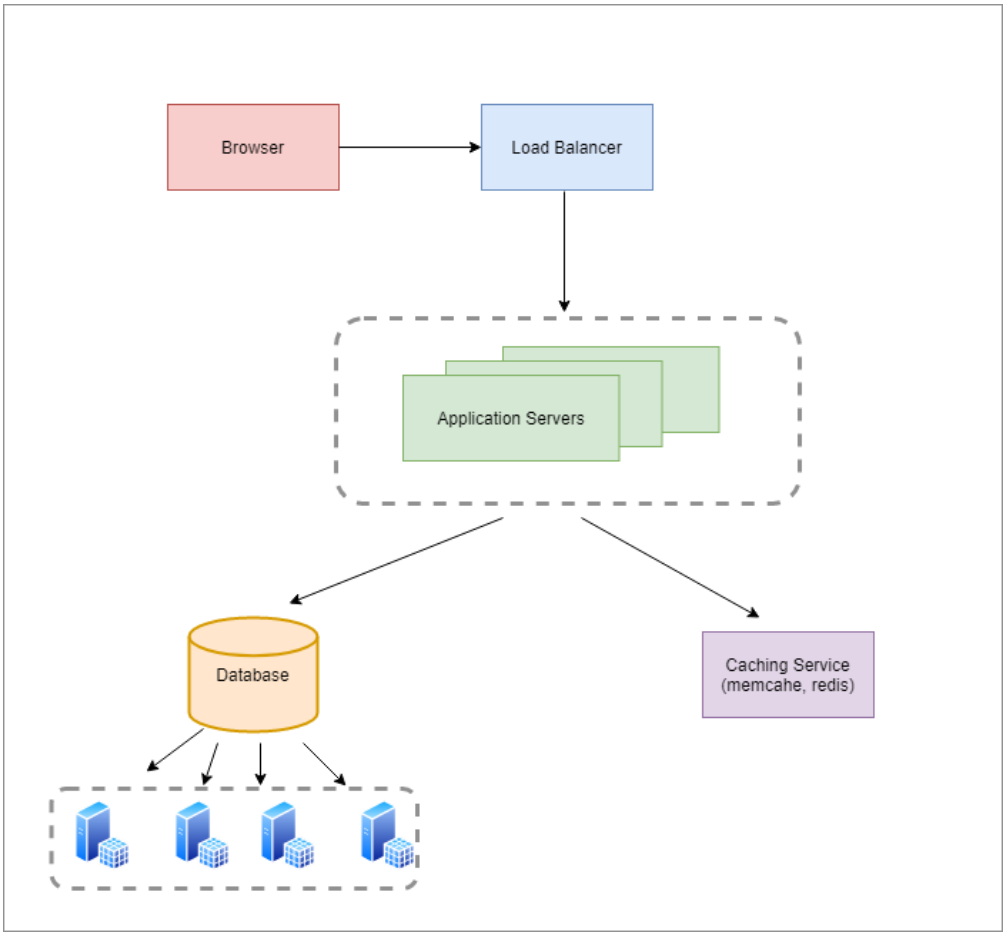

Figure 3: Apstrakcija arhitekture sistema