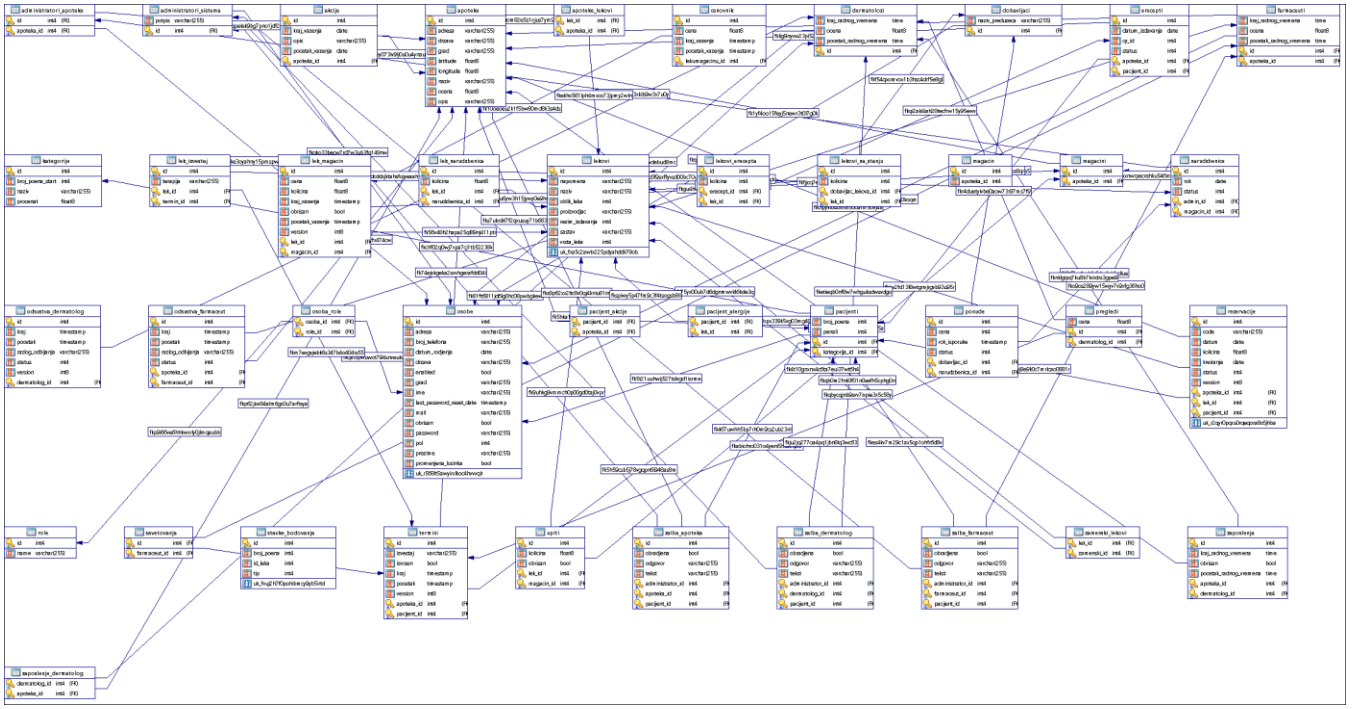
Proof of Concept

Tim 14

1. Dizajn šeme baze podataka



2. Particionisanje podataka

Particionisanje podatka predstavlja razdvajanje postojećih grupisanih podataka na zasebne particije, radi poboljšanja skalabilnosti, performansi, sigurnosti, fleksibilnosti i dostupnosti podataka u aplikaciji. Načini kojima se može pristupiti particionisanju podataka su horizontalni, verikalni i funkcionalni.

Načini na koje se particionisnaje podataka može primeniti na našu aplikaciju su sledeći.

Ogroman broj korisnika dovodi do ogromnog broja održanih termina(svaetovanja i pregleda) samim tim potrebno je optimizovati pristup tabeli baze koja sadrži ove podatke. Pre svega, moguće je kreirati zasebne particije na osnovu vremena izvršavanja termina. Češće će se vršiti pristup terminima koji su bliži trenutnom trenutku i u slučaju provera koje vrši aplikacija, ali i samim farmaceutima i dermatolozima su svakako bitnije svežije informacije. Pored ovoga, informacije o starijim terminima se mogu skladištiti na jeftinijim skladištima podataka, jer potencijalan gubitak ovih informacija nije katastrofalan. Na ovaj način smanjujemo trošak održavanja podataka o terminima. Takođe, poželjno bi bilo vršiti i particionisanje na osnovu sličnosti podataka koje termini nose. Npr. termini održani u određenom gradu, državi ili na manjem nivou apoteci ili skupini apoteka mogu biti smešteni u zasebne particije. Pored toga, ukoliko određene mreže apoteka žele da svoje podatke dodatno zaštite može se napraviti particija koja sadrži samo njihove termine radi povećanja sigurnosti tih podatka.

Osim termina, naša aplikacija radi sa velikim brojem korisnika i apoteka. Moguće su određene particije tabele apoteka na osnovu lokacije(veća šansa da će korisnici imati interakciju sa apotekama koje su im u neposrednoj blizini). Što se tiče kornisnika, određeno particionisanje podataka već postoji, na osnovu njihove uloge u sistemu. Dodatne particije koje se mogu uvesti su: particija koja sadrži samo username, password i role korisnika i particije na osnovu aktivnosti korisnika. Username, password i role su atributi korisnika kojima će se najčešće pristupati zbog prijave na sistem, međutim moguće je napraviti dodatnu particiju koja sadrži ime, prezime i lokaciju, jer su ovo podaci koji bi trebalo prvo prikazati, a onda po potrebi dobaviti ostale podatke. Što se tiče particionisanja podatka na osnovu aktivnosti korisnika, na ovaj način informacije o slabo aktivnim korisnicima možemo čuvati u jeftinijim skladištima podataka i smanjiti trošak održavanja podataka. Takođe, korisnici koji su slabo aktivni će mnogo ređe dolaziti u interakciju sa drugim korisnicima i samim tim njihovi podaci neće biti često potrebni.

Na kraju, treba i spomenuti rad sa lekovima. Velika popularnost aplikacije sa sobom donsi i više zainteresovanih farmaceutskih kuća koje žele da prodaju svoje lekove u apotekama na našem sistemu. Usled ovoga raznovrsnot lekova može eksponencijano rasti. Neke od particija koje možemo uvesti usled ovoga particije na osnovu potražnje za lekom i particija koja sadrži najbitnije informacije o leku. Svakako neće svi lekovi biti podjenako preporučivani i kupovani(što se može posmatrati na osnu rada aplikacije i izveštaja) samim tim lekove za koja je velika šansa da će njihovim informacija biti pristupano tokom termina i kupovine leka, možemo staviti u zasebnu particiju od onih za kojima je potražnja manja. Pored ovoga, često nam nisu potrebne sve informacije o leku. Farmaceuti i Dermatolozi iskustvom često napamet znaju informacije o većini lekova tako da im je potreban samo naziv, eventualno proizvođač leka. Takođe, bitan podatak je i količina leka na stanju jer će ovo biti proveravano u skoro svim akcijama sa lekovima.

3. Replikacija baze i obezbeđivanje otpornosti na greške

Replikacija baze koja bi mogla da se primeni na našu aplikaciju je tipa transakcione replikacije. Pored ovoga prilikom dodavanja novog pomoćnog skladišta(subscriber) potrebno je kirstiti snapshot primarne baze.

Pošto je struktura naše baze statična i neće biti ponovnog dodavanja i brisanja kolona moguće je korišćenje replikacije bazirane na logovima. Ovaj pristup je malo komplikovaniji od ostalih jer zahteva postojanje log file, ali je najefikasniji pristup. Potrebno je posedovati log file u primarnoj bazi koji će sadržati ili sve izmenjenje redove ili same akcije izmene. Kada se započne proces replikacije, na svim pomoćnim bazama se prolazi kroz log file primarne baze i izvršavaju se izmene redom kojim su dadate u log file. Prednost ovog pristupa je to što se nikad ne šalju redovi koji nisu menjani, samim tim komunikacija između baza i proces replikacije su znatno efikasniji.

Sama frekvencija replikacije bi trebala da bude što veća, ali isto tako potrebno je naći balans sa performansama aplikacije. Čestom replikacijom povećavamo otpornost sistema na greške, jer u slučaju da primarni server padne, ostali serveri imaju veoma svežu kopiju podataka. Ovo nam je bitno jer naša aplikacija radi sa medicinskim podacima, koje se mogu smatrati veoma bitnim. Zbog ovog, preporučuje se i korišćenje pune replikacije.

4. Predlog strategije za keširanje podataka

Keširanje podataka je još jedan koncept koji može doprineti skalabilnosti naše aplikacije. Ovo nam dodatno pomaže da smanjimo opterećenje nad samom bazom tako što ćemo imati učitan podskup

podataka. Keširanje se može implementirati na više različitih nivoa, od kojih su dva najčešća L1 (Level 1) i L2 (Level 2). Hibernate podržava i jedan, i drugi nivo keširanja, s tim što se uz L2 mora koristiti i eksterni provajder, kao što su: EhCache, Infinispan, Redis...

L1 keširanje predstavlja keširanje na nivou sesije, tj. ono nam omogućava izbegavanje konfliktnih podataka i učitavanja duplikata na taj način što će zahtev za objektom iz baze uvek vratiti istu instancu. Takođe, memorija se ne opterećuje, jer životni vek keširanog objekta traje koliko i Hibernate sesija.

L2 keširanje operiše na nivou „fabrika sesije“, i za njega postoje strategije adaptirane za različite situacije rada sa aplikacijom: Read Only, Read Write, Nonrestricted Read Write i Transactional. Read Only služi za podatke koji se samo čitaju i nikada se ne ažuriraju, pa je samim tim i najjednostavnija strategija sa optimalnim performansama. Read Write, pored čitanja, može vršiti i ažuriranje podataka, a Nonrestricted Read Write funkcioniše po istom principu, s tim što se podaci ažuriraju veoma retko. Transactional strategija se koristi kod konkuretnih sistema, gde je od velike važnosti konzistentnost podataka. Konkretno u našem sistemu, Read Only bi se moglo primeti nad raznim prikazima podataka, gde nisu moguće CRUD operacije. Nad većinom ostalih operacija i podataka bi bilo najbolje koristiti Read Write strategiju, kako bi se izbegli problemi do kojih može dovesti Nonrestricted Read Write. Takođe, Transactional se može upotrebiti za onemogućavanje konfliktnih situacija operisanja nad podacima, što je i implementirano u pojedinim delovima naše aplikacije.

Kako bi keš memorija bila u toku sa izmenama u sistemu, a da pri tome ne utiče na opterećenje baze, poželjno bi bilo podešavati njen životni vek (TTL - Time To Live). Takođe, ukoliko dođe do popunjenja memorije, poželjno bi bilo koristiti Evicted, koji po principu LRU (Least Recently Used) izbacuje najmanje nedavno korišćeni objekat.

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

Java procesi su dokazano teški za analizu performansi. Zahtevi sistema ne nameću procesorski intenzivne operacije, što nam daje prostora za paralelizaciju i usmerava nas na korišćenje konkurentnosti u programiranju.

Postavljeni slučaj broja korisnika jeste 200 miliona korisnika kao ukupan broj, pored čega se uzima u obzir i 1 milion redovnih mesečnih korisnika. Iz ovoga zaključujemo da bi sistem radio sa ogromnim količinama podataka. Masovna memorija je ključna komponenta sistema koji bi podržao navedene brojke korisnika. Keširanje podataka ovde predstavlja neophodan pristup.

Java Spring Boot aplikacija može da testira svoje opterećenje pod zadatim brojem istovremenih zahteva. Dozvoljava nam da sami biramo koliko ćemo niti koristiti u okviru tomcat servera koji radi u pozadini.

```
server.tomcat.max-threads: 4
```

Jedna studija slučaja je jednostavnu Spring Boot aplikaciju pokrenula u 40 instanci na prosečnom laptopu.

Kada se uzme u obzir RAM memorija kojom je raspolago autor, stiže se do sledećeg rezultata:

3.6GB/40 = 90MB Zauzeće jednog od tih 40 procesa jeste 90 MB.

Ako bismo postavili 8 niti na spring boot aplikaciju, to je 320 niti na 90 MB.

Da bismo bili sigurni u performanse sa opisanim brojem korisnika, reći ćemo da je potrebna sledeća konfiguracija sistema:

CPU: 2.4GHz, 8 jezgara
RAM: 128GB 266Mhz
HDD: 8TB

Opisani sistem je ojačan za 15% svoje procenjene potrebne jačine, iz razloga što sistem treba da ostane stabilan i nakon 5 godina korišćenja.

6. Load Balancing

Veliki broj korisnika aplikacije podrazumeva potrebu za većim brojem servera, kao i njihov raspored na veoma širokom geografskom području. U cilju upravljanja saobraćajem, odnosno komunikacijom sa serverom, preporučuje se uvođenje load balancer-a. Load balancer preusmerava zahtev klijenta na server koji će obraditi taj zahtev. Cilj preusmeravanja zahteva je jednako raspoređivanje posla među serverima kako nijedan server ne bi bio preopterećen, a samim tim i nepouzdan.

Za load balancing postoje različiti algoritmi. Round Robin pristup je najmanje kompleksan za implementaciju. Glavna ideja ovog algoritma je da se zahtevi prosleđuju servirima jednom po jednom, do svakog servera. Mana algoritma je to što nema provere opterećenosti servera. Zahtevi koje klijent naše aplikacije može da uputi serveru se među sobom veoma razlikuju po pitanju potrebnog procesorskog vremena za obradu. Iz tog razloga, koristeći Round Robin pristup može doći do preopterećenja nekih servera. Takođe, uz veliki broj servera ne možemo očekivati da svi serveri imaju iste performanse, te je to još jedan razlog za odbacivanje Round Robin strategije.

Sa druge strane, metod najkaćeg vremena odziva je dobro rešenje za našu aplikaciju. Glavni razlog upotrebe ovog metoda su zahtevi različite kompleksnosti. Pomenuti metod se zasniva na proveri brzine odgovora na zahtev provere servera. Brzina odgovora servera je indikator koji pokazuje koliko je server opterećen. Uzimajući u obzir opterećenost servera, rešava se problem Round Robin algoritma nastao usled zahteva različite kompleksnosti. Dakle, serveri će biti opterećeni proporcionalno svojim performansama, te će opterećenje servera biti približno jednako raspoređeno. Ono što informacija o brzini odgovora servera sa sobom nosi je i informacija o očekivanom korisničkom iskustvu. S obzirom da aplikaciju koristi 200 miliona korisnika, interakcija sa korisnikom i njegovo iskustvo prilikom upotrebe aplikacije su važni, te je ovo još jedan razlog za upotrebu metoda najkaćeg vremena odziva.

7. Predlog koje operacije korisnika treba nadgledati u cilju poboljšanja sistema

Akcije korisnika koje bismo mogli nadgledati variraju u odnosu na tip korisnika. Razmotrimo prvo slučaj klijenta, odnosno pacijenta. Kada bude pretraživao sistem u potrazi za željenim lekom, šanse su da bi njegovu pažnju pre privukle apoteke koje se nalaze u njegovoj okolini. Čak ukoliko je traženi lek nađen u apoteci koja je lokalizovana daleko od njegovog mesta, klijentu se ne isplati da kupuje u njoj, jer ga prevoz do destinacije košta, možda i više neo što bi uštedeo. Ovde pronalazimo mogućnost optimizacije preko lokacijskog servisa. Brža pretraga i veća šansa da će se pretraga završiti kupovinom predstavljaju profitabilne posledice koje su od velikog značaja. Ako gledamo lokacije iz ugla apoteke, takođe bismo videli poboljšanja u korisnosti sistema. Naime, kada administrator apoteke naručuje pošiljku lekova, ponude koje dobija zasigurno su srazmerne sa udaljenošću od apoteke, te bi bliži dobavljači bili efikasniji u svom poslu.

Sledeća optimizacija koja je moguća i vrlo lako ostvariva bi bila izdvajanje popularnih lekova. Nadgledanjem istorije kupovine pacijenta možemo doći do zaključka šta je to što njemu često treba. Lekovi za srce, antibiotici ili dermalne kreme? Statističkom obradom ovih podataka dolazimo do optimizacije u vidu boljeg poretka lekova prilikom pretrage. Prvi rezultati koji mu se pojavljuju treba da budu lekovi za koje znamo da bi ga interesovale, jer smo odradili analizu njegovih prethodnih rezervacija.

8. Dizajn predložene arhitekture