# Proof of Concept

Tim 8

## 1. Dizajn Šema baze podataka

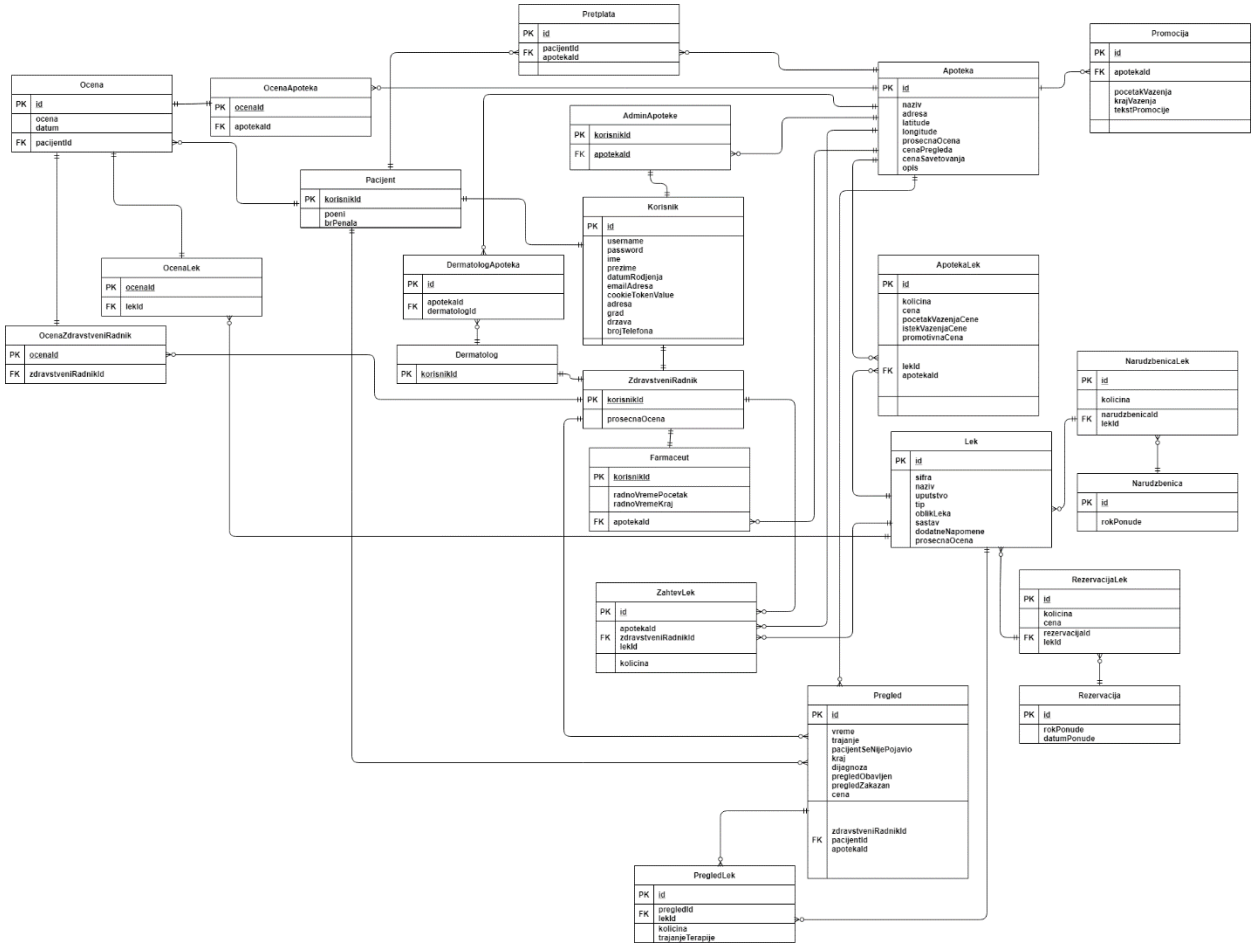



Figure 1: Šema baze podataka

## 2. Particionisanje podataka

Podela podataka u particije je efikasan način da se u sistemu sa velikom količinom podataka povećaju performance i dostupnost samog sistema. Načine podele podataka koje smo izdvojili su:

- **Horizontalno particionisanje** – podela različitih redova u različite tabele
- **Funkcionalno particionisanje** – pri podeli svaki modul koji se izdvoji ima samo podatke koji služe nekom cilju (read-only entities, read-write entities)

### 2.1. Particionisanje tabela

#### 2.1.1 Korisnici - Pacijent

Naš sistem treba da podrži 200 miliona korisnika, samim tim tabela za pacijente će imati veliku količinu podataka u sebi i otežano raditi. Moramo uraditi neko particionisanje podataka za ovu tabelu. Predlažemo horizontalno particionisanje pacijenata pre svega po lokaciji stanovanja. Svaki pacijent će najčešće otići u neku od oklonih apoteka ili neku u obližnjim gradovima. Podela tabele po državama ili čak po regijama unutar same države (Vojvodina) će nam pružiti željeno poboljšanje rada same baze.

#### 2.1.2 Rezervacije lekova

Na osnovu broja korisnika, očekivano da će biti velik broj rezervacija lekova. Svi rezervisani lekovi se trenutno čuvaju u jednoj tabeli bez obzira na to da li jos uvek nisu preuzeti ili su preuzeti. Možemo primetiti da će broj redova u samoj tabeli biti sve veći i veći. Ovo je posebno problematično pošto će sve funkcionalnosti za trenutno rezervisane lekove biti usporene. Rešenje koje predlažemo za ovaj problem je funkcionalno particionisanje po statusu rezervacije. Ovime bi smo sve *read-only* redove odvojili od *read-write* redova i time bi smo poboljšali rad same baze.

#### 2.1.3 Zakazivanje pregleda

Za zakazivanje pregleda, kao i za rezervaciju lekova isto predlažemo funkcionalno particionisanje po statusu pregleda.

#### 2.1.4 Zdravstveni radnici i ostale tabele

U našem sistemu zdravstveni radnici (farmaceuti i dermatolozi) se čuvaju u posebnim tabelama. Kao i za pacijente i za njih se isto može uraditi horizontalno particionisanje na osnovu mesta stanovanja. Ostalim tabelama se mogu primeniti slični koncepti poput ovih navedenih u prethodnim pasusima.

## 3. Replikacije baze i otpornost na greške

U našem sistemu je korišćen PostgreSQL za bazu podataka, tako da nam je omogućeno korišćenje *Log-based Increment Replication* za replikaciju baze. Ovaj princip

replikacije je baziran na čuvanju svih promena baze u vidu *log message* u *log file*. Replikacija se vrši pomoću sačuvanih *log message*-a u *log file*-u. Ova metoda najbolje radi ako je sama struktura baze relativno statična. Ovo je nešto što očekujemo da naša baza može da ispoštuje tako da bi smo se opredelili za ovaj pristup pri replikaciji baze.

Sa 200 miliona korisnika, očekivano je da postojati veći broj serverskih mašina rasprostranjenih po celom svetu. Svaka serverska mašina mora da poseduje kopiju podataka kako bi se održala konzistentnost samog sistema. Strategija koju predlažemo se bazira na propagaciji izmena čvora u mreži na sve ostale kada se izmena na tom čvoru desi. Ovim pristupom se smanjuje opterećenje mreže pošto se samo pri promeni propagira izmena. Pošto je ovo sistem od većeg broja servera koji su međusobno povezani, pri otkazivanju jednog treba obezbediti da sistem i dalje normalno funkcioniše uz lošije performanse.

## 4. Keširanje podataka

Keširanje podataka nam omogućava smanjenu potrebu za pozivima baze. Za keš memoriju predlaže se ***Least Frequently Used*** strategija. Prema tome vodićemo evidenciju o tome koliko je puta svaki red bio referenciran i na osnovu toga kada se keš memorija napuni izbacivati najmanje korišćene. Kod *Least Frequently Used* strategije postoji problem da se određeni redovi u bazi u kratkom vremenskom periodu referenciraju mnogo puta i potom određen druži period ne budu referencirane. Zbog toga predlažemo konstrukciju nekog algoritma koji će primetiti nagli rast referenciranja u toku vremena i pri tome normalizovati ovaj broj u odnosu na druge redove. Pretpostavljeni broj korisnika od oko 200 miliona stvara ogromnu količinu podataka i prema tome predlažemo da se čuvanje tih ličnih podataka korisnika dešava na lokalnom nivou. Pod ličnim podacima misli se na podatke koje može samo korisnik da promeni u celom sistemu. Te podatke je potrebno sinhronizovati sa sistemom samo pri njihovoj izmeni, što smatramo da neće biti često.

## 5. Okvirna procena hardverskih resursa

**CPU**: Pošto *Spring* platforma omogućava višenitnu obradu podataka smatramo da će se naša aplikacija znatno bolje ponašati sa procesorima koji imaju veliki broj jezgara manje brzine. Na primer procesor od 16 jezgara i 2.0GHz.

**RAM**: Zbog same prirode sistema korisnici kao što su pacijenti (koji će činiti većinu korisnika sistema) će samo povremeno praviti zahteve prema sistemu. Prema tome mislimo da je 32GB DDR3 memorije dovoljno za jednu računarsku jedinicu.

**HDD**: Potrebna nam stabilna, i dugotrajna memorija. Procenjuje se da memorija koju svaki korisnik bude zauzimao nije velikog kapaciteta, svega par megabajta prema tome smatramo da je HDD od 10 terabajta memorije dovoljno za jednu instancu, tj oko 200 TB za ceo sistem.

Prema tome jedna računarska jedinica bi imala sledeće specifikacije:

CPU: 2.0GHz sa 16 jezgara
RAM: 32GB DDR3 memorije
HDD: 10TB
Ideja je da imamo klaster od ovakvih računara i širimo sistem dodavanjem novih po potrebi.

## 6. Load balancer

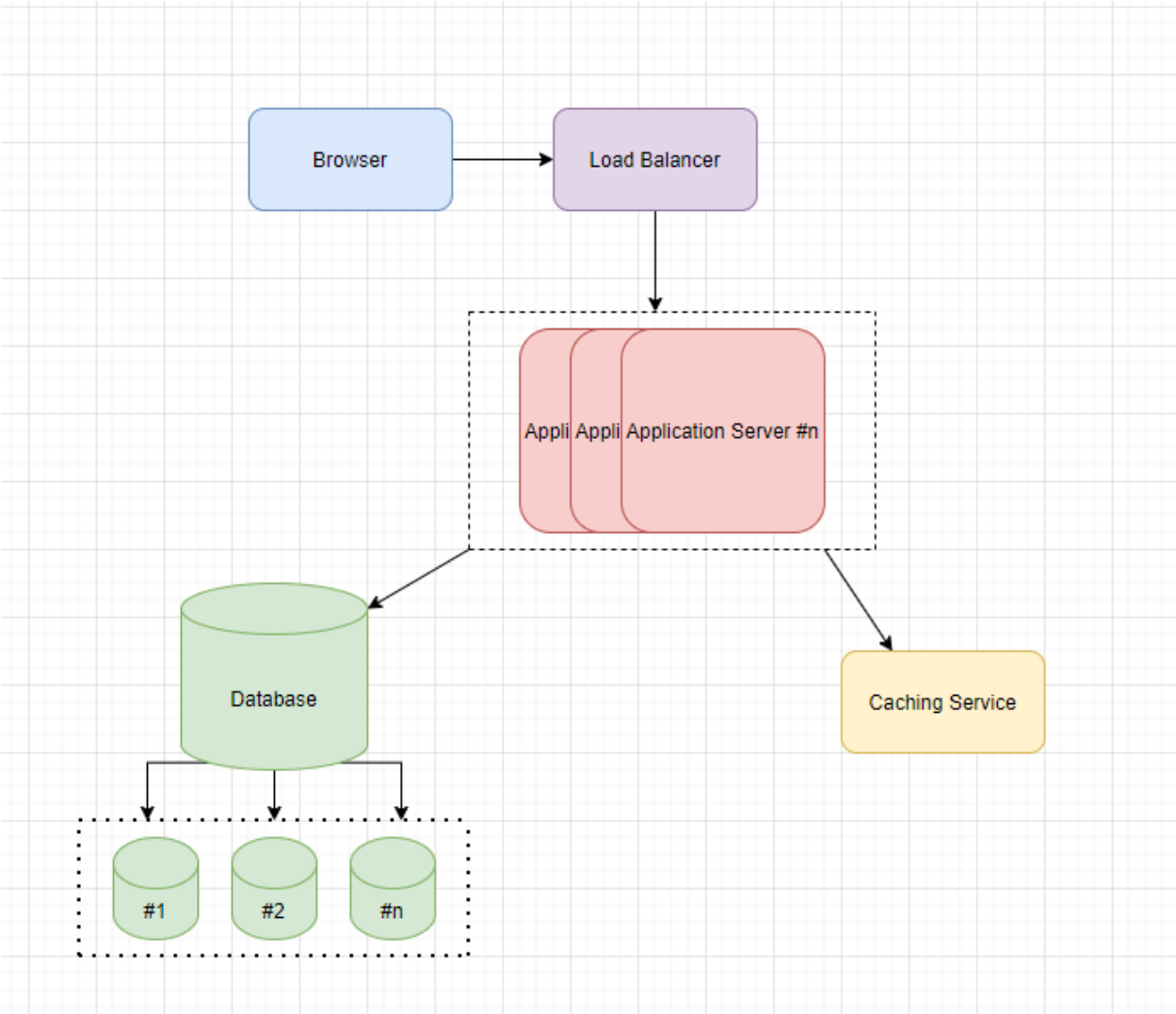
U našem sistem sa 200 miliona korisnika imaćemo veći broj servera. Pored toga sami serveri će biti grupisani na osnovu geograskih lokacija radi load balancinga. Kada korisnik pošalje zahtev sistemu, njegov zahtev će se preusmeriti najbližem skupu servera, tj. njihovom load balanceru. Nakon toga load balancer će preusmeriti zahtev na jedan od servera iz grupe. Algoritam koji mi mislimo da bi bio dobar za ovaj pristup je Least Connections. Least Connections (round robin) prebacuje korisnikov zahtev na server sa najmanje aktivnih veza. Smatramo da nema potrebe uzeti u obzir ostale faktore razmatranja pri balansiranju kao što su: opterećenje servera, vreme odziva, up-down odnos servera, broj aktivnih konekcija, kao i da li je server u skorije vreme bio u većoj meri opterećen. Pojedinačni korisnik naše aplikacije ne troši puno resursa servera, tako da smatramo da je bolje zbog jednostavnosti ne obraćati pažnju na njih.

## 7. Analiza korisničkih akcija

Na osnovu ponašanja i načina korišćenja našeg sistema od strane korisnika, možemo jos više poboljšati gore navede strategije. Stvari poput:

- Podaci koji se najviše dobavljaju iz baze se mogu smestiti na brže server, keširati radi bržeg pristupa ili pomoću vertikalnog particionisanja smestiti u tabele sa manje podataka
- Podaci kojima se retko pristupa od strane korisnika se takođe mogu vertikalno particionisati ili smeštati na sporije servere
- Vremenski periodi u kojima su serveri pod opterećenjem i obrnuto
- Dodavanje dodatnih server ili unapređenje postojećih servera na lokacijama gde je aktivnost korisnika naručito velika

## 8. Dizajn arhitekture


Browser
Load Balancer
Appli
Appli
Application Server #n
Database
Caching Service
#1
#2
#n