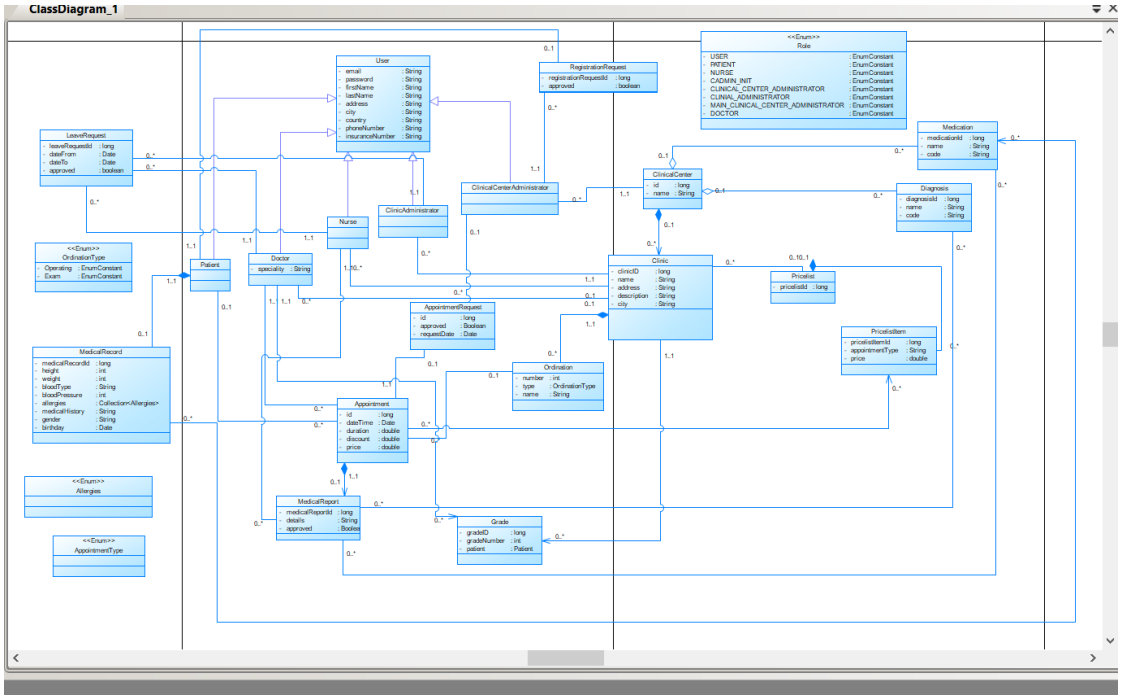
# *Skalabilnost aplikacije*

-Pretpostavke:

- ukupan broj korisnika aplikacije je 200 miliona,
- broj zakazanih novih pregleda i operacija na mesečnom nivou je milion,
- sistem moram biti skalabilan i visoko dostupan.

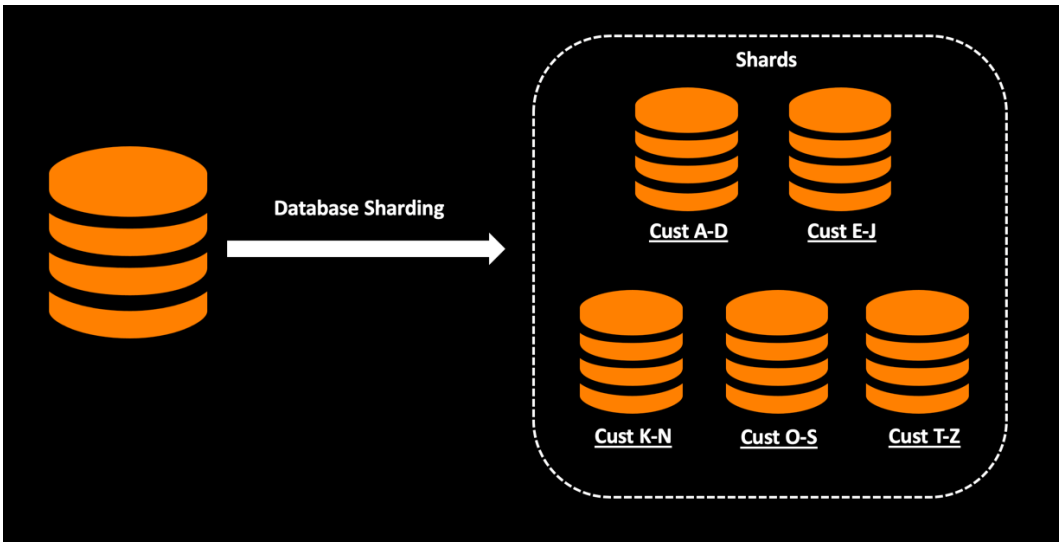
## *1. Dizajn šeme baze podataka (konceptualni, logički ili fizički)*



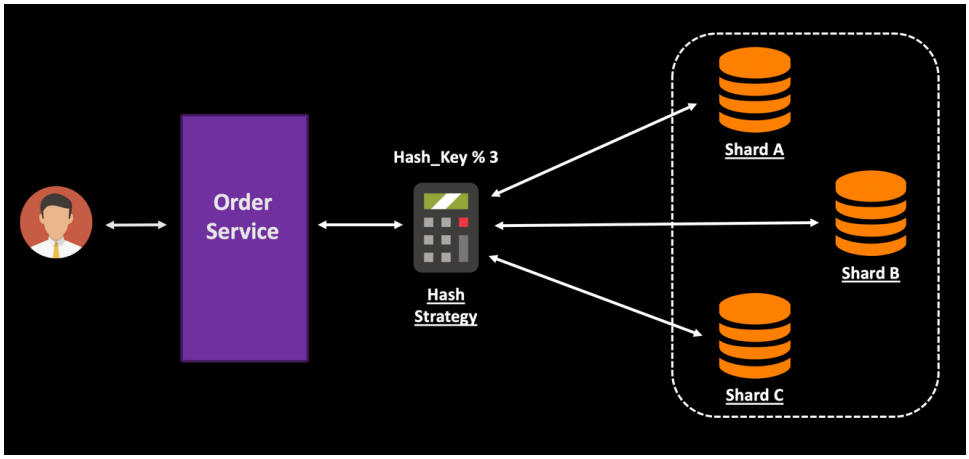
## *2. Predlog strategije za particionisanje podataka*

-Kada broj zahteva aplikacije počne drastično da raste, mora se razmišljati o načinima optimizacije performansi baze. *Sharding* je jedno od potencijalnih rešenja problema. *Sharding* se može definisati kao particionisanje podataka na različite servere kako bi se postiglo ispunjavanje velikog broja zahteva i time povećala efikasnost. Ova metoda poboljšava performanse kako kod *read*, tako i kod *write* zahteva jer sada svaka baza radi sa manjim obimom podataka, odnosno svaki *Shard* se može zamisliti kao zasebna baza podataka.

Kod ovakvog načina particionisanja, podaci se dele u određen broj čvorova na osnovu *shard key*-a. Svaki *shard* će sadržati podskup podataka, što će povećati brzinu upravljanja podacima. Svaki upit koji će se izvršavati nad bazom se pokreće paralelno nad svim *shard*-ovima. Bolje rečeno, *sharding* se može posmatrati kao horizontalno particionasanje gde se podaci distribuiraju na različita mesta kako bi se povećala horizontalna skalabilnost.



Na primeru naše aplikacije, *shard key* može biti, na primer, ID korisnika, ili njegova mail adresa. Međutim, ovde se javlja problem nebalansirane veličine *shard*-ova. Particionisanje na osnovu ID-a ili mail-a može rezultovati nejednakim obimima podataka u različitim *shard*-ovima. Zbog toga bi najbolje bilo koristiti *hash* strategiju za kreiranje *shard*-ova. Koristi se *hash* algoritam za heširanje polja entiteta. Ruter koji sadrži informacije za mapiranje se može onda iskoristiti da distribuira zahteve na određen *shard* na osnovu tog *hash* ključa.




### *3. Predlog strategije za replikaciju baze i obezbeđivanje otpornosti na greške*

-Replikacija baze predstavlja često kopiranje podataka iz baze podataka na jedom serveru u bazu na drugom serveru. Rezultat ovoga je distribuirana baza podataka gde korisnici mogu brzo pristupiti podacima.

U zavisnosti od veličine aplikacije možemo imati *read replicas*, odnosno replike baze koje služe samo za čitanje podataka, dok i dalje, samo glavna baza služi za pisanje. Međutim, ovde se ne rešava kompletno problem, jer aplikacija može dosta narasti, pa bi se ovaj princip koristio tek u početku. Kasnije se mora koristiti repliciranje baze, gde će se svi serveri koristiti i za *read* i za *write*.

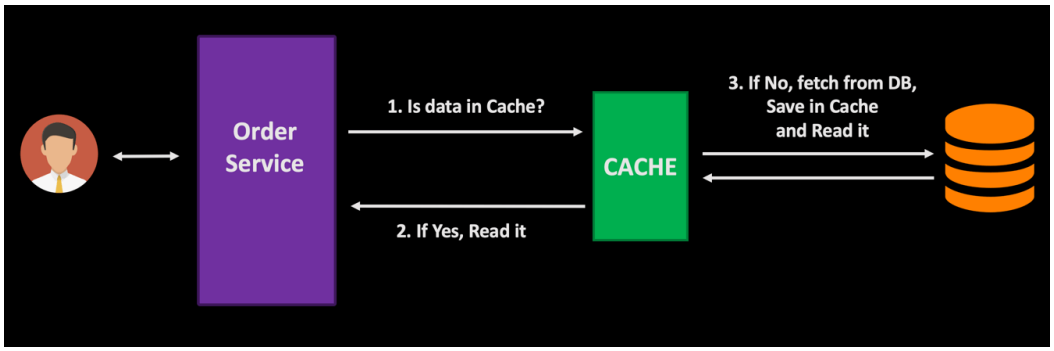
Samo repliciranje baze može biti jednom ili proces koji se stalno ponavlja. Za repliciranje bi se koristili *Distributed database management systems (DDBMS)* koji će osigurati da se bilo kakva izmena u jednoj bazi (dodavanje, brisanje ili promena podataka) automatski reflektuje na baze koje se nalaze na drugim serverima.

Tehnika repliciranja koja bi se koristila bi bila *Synchronous replication*, odnosno sinhronizovano repliciranje. Iako je asinhrono repliciranje dosta lakše za korišćenje jer se replikacije dešavaju u pozadini, korisnik nema garanciju da je baza u konzistentnom stanju, dok se kod sinhronizovanog repliciranja klijenti obaveštavaju da li je repliciranje uspešno ili ne. Ovo oduzima više vremena, i nije jednostavno kao asinhroni pristup, ali osigurava da su svi podaci kopirani pre nego što se nastavi sa radom.

Arhitektura koja, po našem mišljenju, zadovoljava potrebe našeg sistema, a pre svega osigurava konzistentnost baze, jer sama namena aplikacije nam govori da podaci moraju u svakom trenutku biti tačni, je *Multi-leader arhitecture*, koja se zasniva na tome da više servera prima zahteve za izmenu baze i izvršavaju ih, i ova arhitektura je jako korisna kada su replike raspoređene na većoj geografskoj lokaciji, jer je onda moguće omogućiti da uvek imaju neki server u blizini kako bi podaci uvek bili ažurni.

### *4. Predlog strategije za keširanje podataka*

Kada su potrebne dobre performanse sistema koji radi sa podacima, odnosno kada imamo često čitanje podataka iz baze, jako je bitno imati efektivnu tehniku keširanja podataka kojima se često pristupa. Na taj način se smanjuje vreme utrošeno pri pristupu bazi što dosta povećava performanse.




Na primeru naše aplikacije, informacije o korisnicima i o klinikama se mogu keširati, i na taj način smanjiti broj puta pristupa bazi, što povećava performanse, a i ostavlja bazu "slobodnu" za zahteve koji se ne mogu keširati.

Neki od programa koji se mogu iskoristiti za keširanje podataka su NCache ili memecached.

Samo keširanje predstavlja jedan od delova vertikalnog skaliranja, a bilo šta što omogućava brži pristup podacima je svakako potrebno aplikaciji kao što je naša.

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

Ukoliko se iskoristi pretpostavka da je ukupan broj korisnika aplikacije 200 miliona, i da će, opet pod pretpostavkom, za smeštanje podataka za jednog korisnika u bazu biti potrebno oko 70kB na dnevnom nivou jer se smeštaju primitivni tipovi podataka, odnosno nema slika, video zapisa ili audio zapisa, gruba procena potrebnih resursa iznosi:

70kB * 30dana * 12meseci * 5godina *200.000.000korisnika =

23.500 TB

Naravno, za ovu količinu podataka je potreban i veliki broj servera kako bi aplikacija što efikasnije funkcionisala.

### *6. Predlog strategije za postavljanje load balansera*

Balansiranje opterećenja tj. *load balancing* je tipična metoda koja se koristi za raspodelu dolaznog mrežnog saobraćaja i opterećenja na više servera kako bi se umanjili zastoji i povećala efikasnost. Iako je najčešće korišćena strategija balansiranja opterećenja hardverom, mi mislimo da bi za našu aplikaciju bilo efikasnije i sigurnije korišćenje balansiranje opterećenja softverom (*Load Balancing With Software*).

Ovaj način balansiranja se zasniva na tome da balanseri opterećenja evaluiraju zahteve klijenata, i da na osnovu određenih karakteristika (IP adresa, HTTP heder, sadržaj zahteva..) odlučuju kom serveru treba proslediti zahtev.

Za razliku od balansiranja zasnovanog na hardveru, nema uređaja koji to sam radi, pa je eliminisan mogući problem prestanka rada uređaja, a kako je već navedeno, sama primena aplikacije govori da je jako bitno da aplikacija efikasno radi u svakom trenutku.

Iako se *load balanser* može pokretati na običnim serverima, nudi se i druga mogućnost, a to su virtuelni serveri. Kako bi se, opet, smanjila mogućnost prestanka rada, mislimo da je sigurnije da se balanseri pokreću na našem serveru kako bismo u svakom trenutku imali uvid u način rada.

Takođe, dosta je bitno odabrati dobar algoritam raspodele, kako ne bi bilo previše čekanja. Jedan od boljih algoritama je *Least-time algorithm*. Zasniva se na tome da se traži server koji ima najveću brzinu obrade zahteva, a najmanje zahteva na čekanju. Za razliku od ovog algoritma, algoritam *Hash-based algorithm* vrši heširanje kako bi zahtevi istog klijenta bili obrađivani na istom serveru, zbog keširanih podataka, ali kako je efikasnost ključna za ovu aplikaciju, ipak bismo odabrali prvi algoritam, jer server na kome se obrađuju svi zahtevi određenog korisnika može imati ogroman broj zahteva, a to nam ne odgovara.

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

Bilo koja operacija koju korisnici često pozivaju treba biti nadgledana. Kao primer se nameće operacija zakazivanja pregleda. Kako bi sistem poboljšali, u smislu da je lakši za korišćenje, mogli bismo nadgledati u kojim klinikama i kod kojih konkretnih doktora pacijent zakazuje preglede, i ako primetimo da pacijent ima tendenciju da zakazuje preglede u tačno određenoj klinici ili kod tačno određenog doktora, aplikacija bi sama mogla ponuditi te konkretne podatke kako bismo klijentu olakšali rad.

Takođe, može se nadgledati koji tip pregleda pacijent najčešće zakazuje, pa da se na osnovu toga klijentu šalju notifikacije kada je taj tip pregleda na popustu, ili da jednostavno pacijentima koji često koriste naše usluge obezbedimo popuste kao stalnim korisnicima.

### *8. Kompletan crtež dizajna predložene arhitekture (aplikativni serveri, serveri baza, serveri za keširanje, itd)*