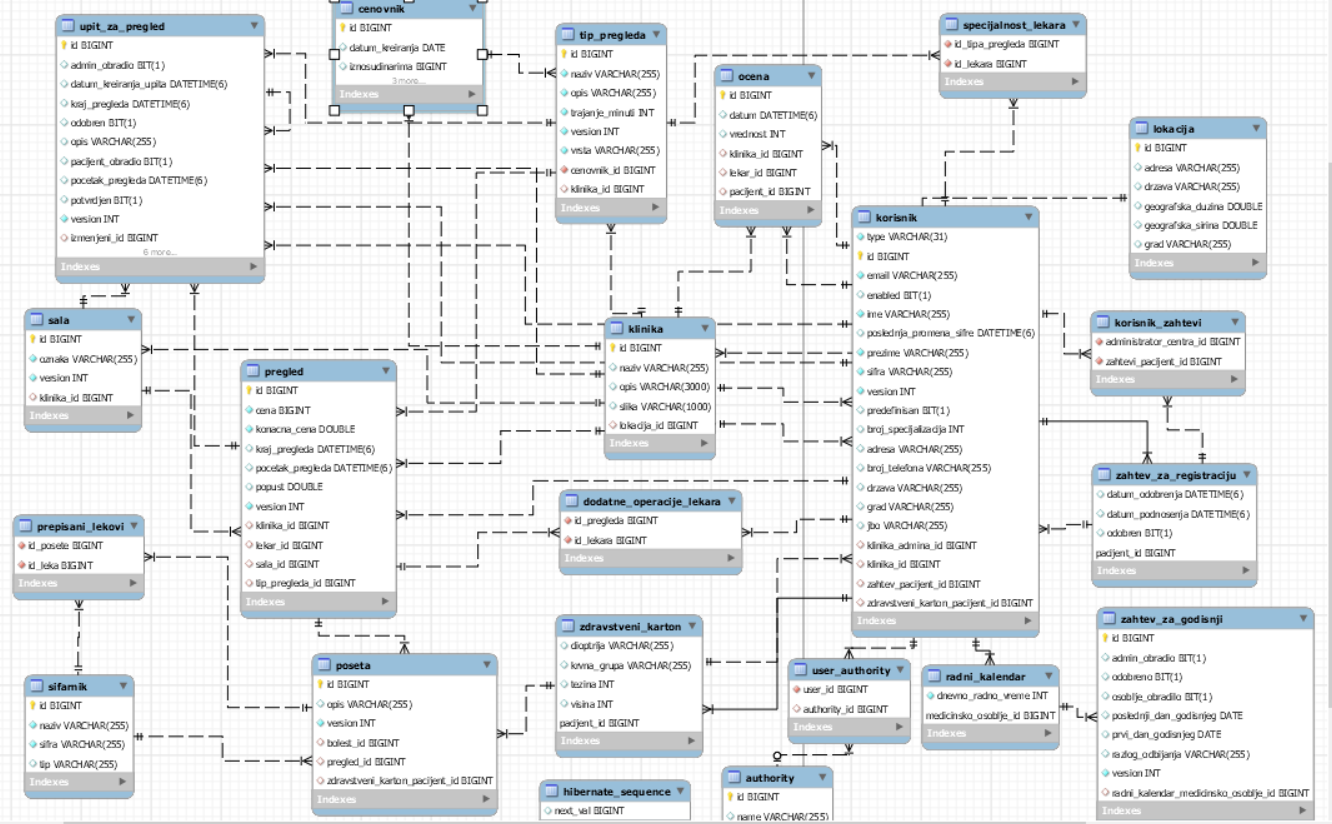
Arhitektektura web aplikacije, proof of concept, tim 21

# Trenutna šema baze podataka – EER dijagram:



# Strategija particionisanja podataka:

## Horizontalno particionisanje

Imajući u vidu da je okviran broj korisnika aplikacije oko 200 miliona, kao i da je okviran broj pregleda na mesečnom nivou oko 1 milion, postavlja se pitanje na koji način fizički distribuirati tabele baze podataka na fajl sistemu. Jedno od standardnih rešenja jeste da se izvrši horizontalno particionisanje tabela za koje se očekuje da jako narastu po broju perzistentnih torki. Horizontalno particionisanje, ukoliko se pametno iskoristi, nudi dosta benefita, a neki od značajnijih su:

* Podaci koji su zastareli i čija perzistencija više nije od značaja mogu jednostavno i brzo biti uklonjeni ukoliko se svi čuvaju u istoj particiji tabele
* Ukoliko postoji SELECT ili DML naredba sa WHERE klauzulom koja se često izvršava, onda izvršavanje te naredbe može biti drastično ubrzano ako se sve torke koje zadovoljavaju WHERE klauzulu nalaze u istoj particiji.

## Izbor sistema za upravljanje relacionom bazom podataka

Jedna od mana horizontalnog particionisanja je što njegova upotrebljivost za konkretan problem dosta zavisi od sistema za upravljanje relacionom bazom podataka (SUBP) koji se koristi, odnosno od njegovog storage engine-a (SE). Konkretno, za implementaciju ovog projekta korišćen je MySQL. Na žalost, iz MySQL dokumentacije vidimo da njegov podrazumevani *InnoDB* SE nameće sledeća ograničenja po pitanju particionisanja tabela:

* Nije moguće izvršiti korisnički definisano horizontalno particionisanje tabele koja ima strani ključ
* Nije moguće izvršiti korisnički definisano horizontalno particionisanje tabela koja se referencira preko stranog ključa iz neke druge tabele

S obzirom da nijedna tabela koja je od interesa iz naše šeme ne zadovoljava ova ograničenja, MySQL horizontalno particionisanje nam nije opcija. Iz tog razloga bi se prilikom skaliranja ove aplikacije gotovo sigurno prebacili na neki SUBP čiji SE ovo podržava, kao što je na primer PostgreSQL (od verzije 12 ovo je podržano).

## Predlog strategije particionisanja podataka

### Particionisanje upita za pregled i zahteva za odsustvo

Kako je očekivani broj rezervisanih pregleda i operacija na mesečnom nivou 1 milion, to znači da je minimalni broj novih upita za pregled 1 milion (minimalni, jer ne računamo i odbijene upite). U slučaju da je recimo očekivano ponašanje da se svaki drugi upit odbije, to znači da će naša tabela za upite za preglede veoma brzo narasti do veličine koja bi mogla negativno da utiče na performanse naredbi koje izvršavamo na njoj. Usled prirode operacija koje se izvršavaju nad ovom tabelom, smatramo da bi minimalno bilo potrebno izvršiti particionisanje nad kolonom *pacijent\_obradio*. Ovo je kolona boolean tipa koja ako je postavljena na *true,* znači da je pacijent video ishod svog upita za pregled i da taj upit za pregled nema više razloga da se skladišti. U tom smislu bi značajno ubrzali postupak održavanja ove tabele, jer bi za ove dimenzije aplikacije sigurno koristili neki stored procedure za brisanje ovakvih upita koji bismo pokretali na, recimo, nedeljnom nivou. Ova operacija bi umesto

*DELETE from upit\_za\_pregled u WHERE u.pacijent\_obradio=true;*

zapravo izgledala

*DELETE from particija\_pacijent\_obradio\_true;*

što je svakako drastično efikasnije. Ako bismo hteli da idemo korak dalje, mogli bi da nad tabelom upit\_za\_pregled izvršimo multi-level particionisanje, i to prvo nad kolonom *admin\_obradio* a zatim nad kolonom *pacijent\_obradio* (ukratko, tok logike je takav da admin prvo prihvati/odbije upit čime se *admin\_obradio* postavlja na true, a zatim pacijent potvrdi rezervaciju/odbijanje čime se *pacijent\_obradio* postavlja na true). Kako admin treba (veoma često) da dobavlja samo upite koje nije obradio, a pacijent samo upite koje admin jeste obradio, ovim dodatnim nivoom particionisanja bi sigurno ubrzali performanse dve veoma česte operacije. Particionisanje nad kolonom *pacijent\_obradio* nam je idalje neophodno zbog efikasnog uklanjanja „mrtvih“ upita. Ukoliko ni ovo nije dovoljno, može se ići korak dalje i dodati još jedan nivo particionisanja, ovoga puta nad stranim ključem koji reprezentuje identifikator klinike (ovo bi dobro leglo u naš sadašnji kod zato što se identifikator klinike uglavnom koristi u WHERE klauzuli upita nad tabelom upit\_za\_pregled).

Naravno, u svim ovim slučaju bi sve kolone na osnovu kojih se vrši particionisanje morale da uđu u sva UNIQUE ograničenja date tabele. Kako se jedino ovakvo ograničenje odnosi na primarni ključ, zaključujemo da bi primarni ključ tabele upita za pregled morao da bude kompozitni.

Za particionisanje zahteva za odsustvo bi mogli upotrebiti sličnu strategiju.

### Particionisanje tabele korisnika

Šema baze podataka koju smo predstavili čuva sve korisnike aplikacije u jednoj tabeli. Kako je očekivani broj korisnika aplikacije 200 miliona, horizontalno particionisanje tabele korisnika je neophodno. Smatramo da bi imalo smisla izvršiti multi-level particionisanje u ovom slučaju, konkretno po stranim ključevima authority\_id i klinika\_id. Razlog za to je taj što je čest slučaj korišćenja od strane pacijenta sledeći:

SELECT k from korisnik k where TYPE(k)=’LEKAR’ AND k.klinika\_id=:idKlinike;

Sa druge strane, lekar pri pretrazi pacijenta često izvršava upit:

SELECT k from korisnik k where TYPE(k)=’PACIJENT’ AND k.klinika\_id=:idKlinike;

Izvršavanje prethodna dva upita bi se značajno ubrzalo ukoliko bi se sve torke koje ih zadovoljavaju nalazile u istoj particiji.. Naravno, i u ovom slučaju bi se šema morala promeniti na taj način da je primarni ključ tabele korisnika kompozitni, odnosno (id, authority\_id, klinika\_id).

### Particionisanje tabele pregleda:

Pri dodavanju novog pregleda često treba proveriti da li su sala i lekar slobodni za dati termin pregleda. Pri toj proveri, moramo izvršiti sledeće upite:

SELECT p from pregled p WHERE p.lekar\_id=:idLekara;

SELECT p from pregled p WHERE p.sala\_id=idsale;

Iz ovog razloga mislim da bi se pregledi mogli particionisati po identifikatoru sale i lekara. To će naravno opet imati posledice po primarni ključ pregleda, koji bi onda izgledao (id, id\_sale, id\_lekara).

### Particionisnje ostalih tabela:

Usled prirode same aplikacije, smatramo da particionisanje ostalih tabela nije neophodno.

# Replikacija baze podataka

Aplikacija koja koristi samo jednu instancu servera baze podataka (jedan čvor) za perzistenciju svih svoji podataka ima sledeće probleme:

* Tolerancija na otkaze praktično ne postoji pošto je dati čvor takozvani „*single point of failure*“ čiji otkaz potpuno onemogućava dalje korišćenje aplikacije.
* Pri povećanju broja korisnika i intenziteta saobraćaja ka bazi podataka, vreme odziva (*transactional response time*) značajno raste dok proposunost (*throughput*) sa druge strane opada.

U cilju sprečavanja ovih problema vrši se replikacija baze podataka. Dve strategije koje se mogu koristiti pri replikaciji su master-slave i master-master. Master-slave replikacija je rešenje u kojem se sve write operacije usmeravaju ka jednoj instanci baze podataka (master). Nakon uspešnog commit-a, master propagira promene ostalim replikama (slaves) kako bi se očuvala konzistentnost. Prednosti ovog modela su sledeće:

* Usmeravanje svih readonly transakcija ka slave čvorovima. Ovo ima za posledicu uvećanje propusnosti celog sistema
* Konzistentnost podataka je lako održiva. Iako su svi slave čvorovi zapravo „eventually consistent“, ovo nije problem zato što se nad njima mogu izvršavati samo operacije čitanja tako da konzistentnost podataka ne može biti narušena

Upotrebom master-slave strategije replikacije bi svakako uspeli do neke mere da poboljšamo performanse sistema. Takođe, tolerancija na otkaze se postiže tako što se nametne **sinhrono** propagiranje WAL fajlova (write ahead logs - protokol za propagiranje promena između replika) između mastera i **jednog** slave čvora. Na ovaj način bi usporili write operaciju nad master-om (zato što master mora da se blokira dok sinhroni slave ne potvrdi da je primenio promene), ali bi se osigurali da u slučaju otkaza mastera imamo jednu repliku koja je potpuna ažurna, i u tom slučaju bi izbor algoritma za novog mastera bio trivijalan. Master bi komunicirao sa svim ostalim slave čvorovima **asinhrono**.

Sa druge strane, master-master replikacija nam nudi mnogo veću efikasnost na račun globalne konzistentnosti. U ovakvom modelu, svaki čvor je osposobljen za obradu write operacija. Ovo rešenje je svakako korisno za geografski distribuirane sisteme, ali nameće problem očuvanja konzistentnosti. Imajući u vidu da postoje elementi naše aplikacije, poput obrade upita za pregled i kreiranje pregleda, gde je konzistencija bitna, ipak smo se opredelili za master-slave strategiju replikacije koja će detaljnije biti predstavljena u poslednjem poglavlju.