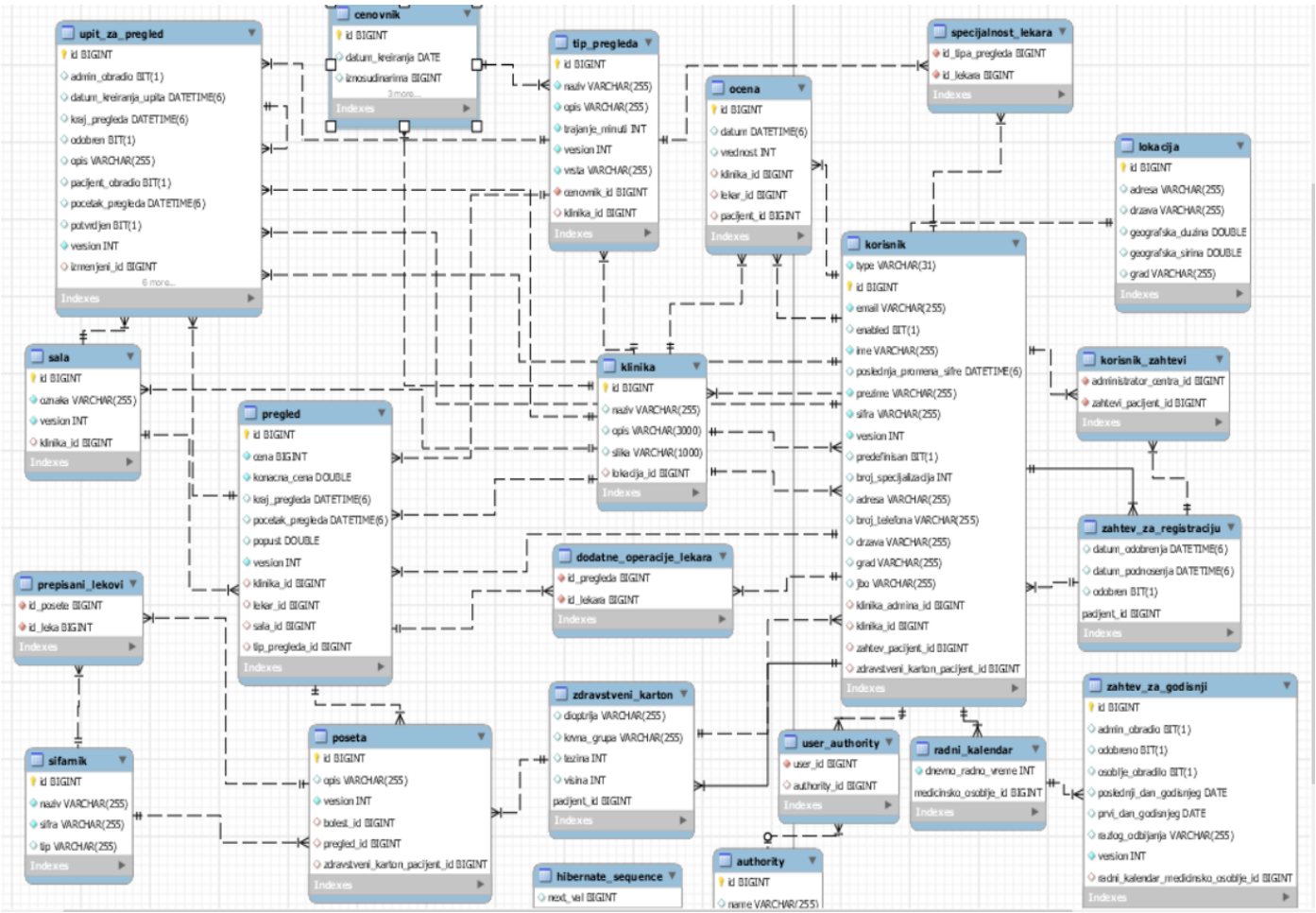
Arhitektektura web aplikacije, proof of concept, tim 21

1. Trenutna šema baze podataka – EER dijagram:



2. Strategija particionisanja podataka:

1. 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.

2. 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).

3. 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.

3. 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 propusnost (*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 algoritam za izbor 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.

4. Predlog strategije keširanja podataka

4.1. Keširanje podataka

I pored primenjenih particionisanja baze podataka i njene replikacije, sistem je potrebno dodatno optimizovati uvođenjem keširanja podataka. Keširanje podataka podrazumeva efikasan dodatan sloj pristupa podacima koji čuva samo podskup podataka kako bi pristup tim podacima bio brži od standardnog načina na koji im se pristupa (BP upiti). Keširanje se može izvesti na više nivoa.

4.2. Keširanje na nivou baze podataka

MySQL kao izabrani SUBP poseduje sopstveni in-memory sistem za keširanje u sklopu storage engine-a InnoDB. Nakon odrađenog *profiling*-a performansi baze (koristeći jmeter ili slično) konfigurisali bismo InnoDB-ov „buffer pool“ (buffer_pool_size, buffer_pool_chunk_size, buffer_pool_instances itd.). Ukoliko se, pak, budemo želeli veću fleksibilnost keširanja, to možemo postići korišćenjem nekog spoljašnjeg sistema za in-memory keširanje poput **memcached** ili **Redis.**

4.3. Keširanje na nivou Hibernate-a

Sam Hibernate kao implementacija JPA-a nudi različite nivoe keširanja, ali i vrste keširanja (entitet/upit). Prvi nivo keširanja, koji je zaslužan za keširanje objekata iz baze na nivou sesije, se već koristi u aplikaciji. Drugi nivo keširanja, koji je zaslužan za keširanje objekata na nivou „fabrike sesija“ na određeni način (READ_ONLY, READ_WRITE itd.) bi bilo potrebno dodatno podesiti. Na kraju, Hibernate nudi i keširanje upita što bismo isto podesili na odgovarajući način.

4.4. CDN keširanje

CDN (Content delivery network) je distribuirana mreža proksi servera koji obično služe da učine statičke podatke bliže korisniku te da se oni brže učitaju kod korisnika. Postoje razni provajderi koji nude CDN usluge. U našem slučaju, CDN bi „servirao“ frontend statički sadržaj (html/js/css).

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

5.1. Skladištenje podataka o korisnicima

Skladištenje podataka o korisnicima podrazumeva skladištenje korisničkog profila i zdravstvenog kartona. Podaci o jednom korisničkom profilu zauzimaju otprilike 1500 bajtova, dok povezani zdravstveni karton zauzima 500. Ukoliko imamo 200 miliona korisnika aplikacije (u ovom slučaju samo pacijenata) to je 200 M * 2 KB = 400 TB prostora.

5.2. Skladištenje podataka o klinikama

Skladištenje podataka o klinikama podrazumeva skladištenje osnovnih podataka o klinici (uključujući slike), njenim salama, lekarima, medicinskim osobljem i ocenama. Osnovni podaci o klinici zauzimaju otpilike 4000 B bez slika, a sa 3 slike 3,004 MB. Ukoliko svaka klinika ima 50 sala, 80 lekara i 160 ljudi pripadnika medicinskog osoblja, prostor koji klinika zauzima raste na otpilike: 3,004 MB + 50 * 300 B + 80 * 1500 B + 160 * 1500 B = 3,350 MB. Ukoliko u svetu imamo 500 klinika, jedna klinika u proseku ima 400000 pacijenata od kojih će 20% njih imati ocenu klinike i lekara što je dodatnih 1/5 * 400.000 * 50 B = 40 GB. Naposletku, ukupan prostor koji zauzimaju podaci o klinikama je: 11 TB.

5.3. Skladištenje podataka o pregledima

Skladištenje podataka o pregledima podrazumeva skladištenje osnovnih podatka o pregled (datum, lekar, sala itd.), upitima za taj pregled, poseti vezanoj za taj pregled. Osnovni podaci o pregledu zazimaju 100 B. Ukoliko upita po pregledu ima prosečno 2, prostor koji jedan pregled konceptualno zauzima je 100 B + 2 * 350 B = 800 B. Poseta vezana za pregled zauzima 300 B. Ukupan prostor koji jedan pregled zauzima je otpilike 1100 B. Ukoliko mesečno imamo 1.000.000 pregleda, potrebno nam je 1,1 TB za skladištenje svih pregled. U periodu od 5 godina, taj prostor raste na 66 TB.

5.4. Krajnje procene

Na osnovu procenjenih veličina skladišta za podatke o korisnicima, klinikama i pregledima, **ukupna procenjena veličina skladišta je oko 500 TB za rok od 5 godina**. Pri kreiranju ove procene u obzir nije uzeto skladištenje: tipova pregleda (koje se relativno retko menja), cenovnika, baze lekova i drugih ne toliko ključnih entiteta. Takođe, ukoliko pacijenti i/ili dijagnoze ili npr. operacije imaju slike vezane za njih, veličina skladišta bi se višestruko uvećala.

6. Predlog strategije za postavljanje load balansera

6.1. Pojam load balansera

Load balancer je softver ili hardver (dosta skupo) koji prosleđuje dolazeće zahteve klijenata određenim resursima poput servera aplikacije ili bazama podataka. Load balanser na ovaj način sprečava to da zahtevi budu upućeni serverima koji su otkazali, efikasno raspoređuje opterećenje na sve instance i pruža fleksibilnost za dodavanje ili gašenje servera u zavisnosti od potrebe itd.

6.2. Naš setup

S obzirom na to da koristimo HTTP-bazirani stateless protokol za komunikaciju – REST – vertikalno skaliranje same aplikacije je veoma jednostavno, tj. ne moramo voditi brigu o korisničkim sesijama. Dovoljno je samo pokrenuti više instanci aplikativnog servera, a „ispred“ postaviti najobičniji **round-robin** load balanser, npr. nginx-ov. Na ovaj način, ukoliko je sezona gripa ili je, pak, letnji period sa umanjenim brojem pregleda, možemo povećati ili smanjiti broj instanci aplikativnih servera, a load balanser će nam omogućiti da iskoristimo te instance i rasteretimo sistem.

Druga opcija je da automatizujemo skaliranje i load-balancing koristeći tehnike kontejnerizacije aplikacije i automatske inicijalizacije i orkestracije kontejnera. Tehnologije koje bi nam to omogućile su npr. **Docker** i **kubernetes.**

7. Predlog arhitekture: