MATEMATIČKI FAKULTET
UNIVERZITET U BEOGRADU

Baza podataka Redis

MASTER RAD

Autor:

Ana Simijonović

Mentor:

dr Aleksandar Kartelj

Beograd, 2019.

Mentor: **dr Aleksandar Kartelj**

Matematički fakultet
Univerzitet u Beogradu

Članovi komisije: **prof. dr Nenad Mitić**

Matematički fakultet
Univerzitet u Beogradu

dr Milan Banković

Matematički fakultet
Univerzitet u Beogradu

Datum odbrane: _____

Sadržaj

1 Uvod

Redis sistem baza podataka je projekat otvorenog koda koji je nastao na osnovu ideje programera Salvatorea Sanfilippa 2009. godine. Trenutno se razvija u kompaniji RedisLabs i napisan je u programskom jeziku C.

Redis je akronim od **Re**mote **Di**ctionary **S**erver (srp. udaljeni rečnički server). Redis aplikacija se sastoji iz Redis servera i Redis klijenta. Redis server je zadužen za skladištenje podataka i čitavu logiku njihove obrade, dok Redis klijent predstavlja biblioteku implementiranu u programskom jeziku u kom je pisana aplikacija.

Redis server čuva podatke u primarnoj memoriji, što omogućava veoma brze operacije čitanja i pisanja u bazu podataka. S obzirom na to da se podaci koji se nalaze u primarnoj memoriji gube pri gašenju servera, Redis sadrži mehanizme za čuvanje podataka u sekundarnoj, trajnoj memoriji na neki od dva načina – čuvanjem svih podataka iz baze kada se zadati uslovi ispune (broj upisa u bazu, zakazano vreme itd.) ili čuvanjem svake od izvršenih naredbi nad bazom.

Redis je NoSQL sistem baza podataka, ili kako je često nazivaju skladište (engl. data store), koji organizuje podatke u ključ - vrednost parove. Ključ je niska koji jedinstveno određuje jedan zapis, dok vrednost može biti tipa: String, Lista, Skup, Uređeni skup, Heš i HyperLogLog.

Redis sistem baza podataka trenutno koristi veliki broj kompanija kao što su Twitter, GitHub, StackOverflow, Pinterest, Instagram, Snapchat. Može se koristiti kao baza podataka, kao sistem za keširanje podskupa podataka iz primarne baze podataka, keširanje sesije, kao implementacija redova zadataka, kao posrednik u slanju poruka itd.

1.1 Redis tipovi podataka

Redis sistem za upravljanje bazama podataka podržava nekoliko tipova podataka koji liče na standardne strukture podataka u programskim jezicima. Prilikom rešavanja različitih problema veliku pažnju treba posvetiti odabiru pravog tipa podatka.

Redis ključ je niska i uobičajeno je dizajnirati ga tako da ne bude predugačak, ne samo zbog opterećenja memorije, već i zbog pada performansi usled operacija poređenja ključeva prilikom pretrage podataka. Maksimalna dužina ključa je 512 MB.
Redis vrednost je jedinstveno određena ključem koji se u nju preslikava. Redis vrednost može biti tipa: String, Lista, Skup, Uređeni skup, Heš i HyperLogLog. Mapa bitova se ponekad navodi kao poseban tip podatka u Redisu, iako je to vrsta Stringa koji sadrži samo nule i jedinice.

Redis string

Redis string se sastoji iz sekvence karaktera i može sadržati tri vrste vrednosti: tekst (xml, json, html ili sirovi tekst), brojeve (cele brojeve i brojeve u pokretnom zarezu) i binarne vrednosti. Maksimalna veličina je 512 MB. Neke od svrha korišćenja su:

- Keš mehanizmi - za čuvanje teksta ili binarnih podataka u Redisu;
- Brojanje - ako vrednost tipa String može da se interpretira kao ceo broj u dekadnom sistemu ili kao razlomljeni broj, Redis će to detektovati i biće moguća njhova obrada pomoću naredbi poput inkrementacije i dekrementacije.

Takođe je omogućeno čitanje i pisanje niski u druge niske.
Tipične naredbe za rad sa tipom podatka Redis String date su u tabeli 1.1.

Ime naredbe	Opis naredbe
SET	Postavlja vrednost na dati ključ.
GET	Pribavlja vrednost sa datog ključa.

Tabela 1.1 Naredbe za rad sa Redis stringom

Redis lista

Lista u Redisu je implementirana kao jednostruko povezana lista elemenata. Elementi liste su tipa String, i sortirani su prema redosledu dodavanja u listu. Budući da je ona implementirana kao jednostruko povezana lista, a ne kao indeksirani niz, operacija dodavanja elemenata u veoma dugu listu je veoma brza, što je njena glavna prednost. Operacije dodavanja i čitanja prvog i poslednjeg elementa iz liste izvršavaju se u konstantnom vremenu *O(1)*, dok je složenost operacije pristupanja *n*-tom elementu *O(n).* Stoga, ukoliko je operacija pristupanja elementima u sredini frekventnija, postoje pogodniji tipovi podataka od liste. Tipične primene Redis liste su:

- Kod društvenih mreža za čuvanje poslednjih *n* poruka/komentara/slika;

- Kod međuprocesne komunikacije za smeštanje poruka.

Tipične naredbe za rad sa listama date su u tabeli 1.2.

Ime naredbe	Opis naredbe
RPUSH	Postavlja vrednost u element na kraju liste.
LPUSH	Postavlja vrednost u element na početku liste.
RPOP	Uklanja i vraća element sa kraja liste.
LPOP	Uklanja i vraća element sa početka liste.
LINDEX	Vraća element sa zadate pozicije.
LRANGE	Vraća elemente iz liste u zadatom intervalu, uključujući i elemente sa početne i krajnje pozicije intervala.
LTRIM	Uklanja elemente iz liste koji se ne nalaze u intervalu zadatom u argumentima.

Tabela 1.2 Naredbe za rad sa Redis listom

Redis skup

Redis skup predstavlja neuređenu kolekciju niski. Elementi u skupu moraju biti jedinstveni, odnosno nisu dozvoljeni duplikati. Skup je implementiran kao heš tabela, što implicira da se dodavanje, uklanjanje i pretraga vrednosti vrši u konstantnom vremenu. S obzirom na to da su elementi skupa neuređeni, elementi se dodaju u skup i uklanjaju iz njega po vrednosti. Tipične primene skupova su:

- Filtriranje ili grupisanje podataka - koristeći operacije preseka, razlike i unije skupova. Na primer pronaći sve letove koji kreću iz jednog, a završavaju u drugom mestu;
- Proveravanje pripadnosti grupi.

Tipične naredbe za rad sa skupovima date su u tabeli 1.3.

Ime naredbe	Opis naredbe
SADD	Dodaje elemente u skup.
SCARD	Vraća broj elemenata u skupu.
SDIFF	Vraća razliku skupova.
SINTER	Vraća presek skupova.
SISMEMBER	Vraća 0 ukoliko je dati element član skupa ili 1 ukoliko dati element nije član skupa.
SMEMBERS	Vraća sve elemente skupa.
SUNION	Vraća uniju skupova.

Tabela 1.3 Naredbe za rad sa Redis skupom

Redis uređeni skup

Uređeni skup (engl. *sorted set*, *zset*) predstavlja uređenu kolekciju jedinstvenih vrednosti tipa String. Svakom elementu skupa je dodeljen realan broj - rang (engl. *score*). Elementi u uređenom skupu su sortirani na sledeći način:

- ako elementi A i B imaju različiti rang onda: A > B ako je A.rang > B.rang, A < B ako je A. rang < B. rang,
- ako elementi A i B imaju isti rang onda se na njih primenjuje leksikografsko poređenje.

Operacije su sporije nego kod neuređenih skupova zbog prisustva ranga koga je potrebno upoređivati, i logaritamske su složenosti. Ovaj tip podatka je posebno pogodan kada je potrebno pristupati elementima čiji je rang poznat. Tipične primene su:

- Prikaz liste čekanja u realnom vremenu;
- Prikaz najboljih rezultata u igri, prikaz korisnika sa rezultatom sličnim zadatom korisniku;
- Automatsko završavanje reči.

Tipične naredbe za rad sa uređenim skupovima su date u tabeli 1.4.

Ime naredbe	Opis naredbe
ZADD	Dodaje elemente u uređeni skup.
ZCARD	Vraća kardinalnost uređenog skupa.
ZCOUNT	Vraća broj elemenata uređenog skupa u datom intervalu rangova.
ZINCRBY	Uvećava rang elementa u uređenom skupu za zadati inkrement.
ZRANGE	Vraća elemente uređenog skupa u zadatom intervalu pozicija. Ukoliko je zadat opcioni parametar *WITHSCORES,* naredba vraća elemente sa njihovim rangovima.
ZRANK	Vraća poziciju elementa u uređenom skupu.
ZSCORE	Vraća rang elementa u uređenom skupu.

Tabela 1.4 Naredbe za rad sa Redis uređenim skupom

Redis heš

Heš u Redisu predstavlja kolekciju atribut - vrednost parova. I atribut i vrednost u Redis hešu su tipa String pa je heš zapravo preslikavanje elemenata tipa String u elemente tipa String. Stoga, za njih važe osobine koje važe za Redis string - mogu se interpetirati na razne načine u zavisnosti od toga šta sadrži. Heševi su veoma pogodni za grupisanje elemenata koji semantički pripadaju zajedno, pa se često koriste za skladištenje kompleksnih objekata. Na primer, za objekat tipa auto, heš atributi bi mogli biti “boja”, “snaga motora” itd.
Redis heš se može porediti sa redom u relacionoj bazi ili sa dokumentom u dokument-orijentisanoj bazi, i moguće je baratati pojedinačnim ili višestrukim atributima odjednom.
Tipične operacije za rad sa heševima date su u tabeli 1.5.

Ime naredbe	Opis naredbe
HSET	Postavlja atribut na zadatu vrednost.
HGET	Vraća vrednost atributa.
HGETALL	Vraća sve atribute i njihove vrednosti.
HDEL	Briše atribute iz heša.

Tabela 1.5 Naredbe za rad sa Redis hešom

Mapa bitova

Mapa bitova nije novi tip podatka, već je posebna vrsta niski koja sadrži sekvencu nula i jedinica. Indeksi bitova u mapi bitova nazivaju se često pozicijama i koriste se na različite načine u zavisnosti od aplikacije. Nule se posmatraju kao "isključeni" bitovi, dok su jedinice "uključeni" bitovi.
Mape bitova su efikasne za analitiku u realnom vremenu - da li se neki događaj desio, da li je korisnik izvršio određenu akciju u određenom vremenskom intervalu ili broj poseta veb sajtu određenog datuma.
Memorijska efikasnost Mape bitova se može uvideti poređenjem sa skupom, na primeru broja poseta sajtu određenog datuma. Poziciju u Mapi bitova predstavlja identifikatorkorisnika. Pretpostavimo da naša aplikacija ima 5 miliona korisnika, a da je na zadati datum koristilo 2 miliona korisnika, i da se svaki identifikator predstavlja uz pomoć 4 bajta. Kod Redis skupa potrebno je čuvati identifikator svakog korisnika koji je posetio sajt kao poseban element skupa. Kod Mape bitova potrebno je čuvati jedinice na onim pozicijama jednakim identifikatoru korisnika koji su posetili sajt određenog datuma. U najgorem slučaju u implementaciji pomoću Mape bitova, moraće se alocirati memorija za čitav opseg vrednosti polja ID, ukoliko je korisnik sa najvišom vrednošću polja ID (ID od 5000000) posetio sajt tog datuma. Međutim, Mapa bitova nije efikasna kada je broj posetilaca mnogo manji od broja korisnika, na primer 100, i tada će ona opet iskoristiti 5 miliona bitova u najgorem slučaju (tabela 1.6).

Broj posetilaca sajta	Struktura podatka	Broj bitova po korisniku	Broj identifikatora korisnika koje čuvamo	Iskorišćena memorija
2 miliona	Mapa bitova	1	5 miliona	1b x 5000000 = 625kB
2 miliona	Skup	32	2 miliona	32b x 2000000 = 8MB
100	Mapa bitova	1	5 miliona	1b x 5000000 = 625kB
100	Skup	32	100	32b x 100 = 0.4kB

Tabela 1.6 Primer poređenja Mape bitova i Skupa

Postoje posebne naredbe koje operišu njima i dele se u dve grupe: operacija nad pojedinačnim bitovima i operacija nad grupom bitova.

Ime naredbe	Opis naredbe
SETBIT	Postavlja vrednost bita na zadatoj poziciji.
GETBIT	Vraća vrednost bita na zadatoj poziciji.

BITOP	Izvršava bitovsku operaciju između zadatih Mapa bitova.
BITPOS	Vraća poziciju prvog bita postavljenog na zadati bit (0 ili 1).

Tabela 1.7 Naredbe za rad sa Mapom bitova

HyperLogLog

HyperLogLog je struktura podataka koja se koristi za procenu broja jedinstvenih elemenata unutar zadate liste vrednosti. Koristi probabilistički algoritam koji procenjuje kardinalnost skupa različitih elemenata sa greškom od oko 1%. Veoma je koristan kada potpuna preciznost nije jedan od zahteva pri rešavanju problema, dok ušteda vremena i prostora jeste. Za deterministički algoritam brojanja jedinstvenih elemenata potrebno je izdvojiti onoliko memorije koliko vrednosti treba izbrojati, zato što je potrebno zapamtiti elemente koji se ponavljaju, kako se ne bi brojali više puta. Kod HyperLogLog strukture podataka potrebno je rezervisati 12kB po ključu u najgorem slučaju. Kodirani su kao Redis string, ali postoje posebne naredbe za njih - za dodavanje, brojanje i spajanje stringova. Česti slučajevi korišćenja:

- Brojanje različitih upita izvršenih od strane korisnika;
- Brojanje različitih tagova koje korisnik prati;
- Brojanje različitih reči koji se pojavljaju u knjizi.

Komande za rad sa strukturom podataka HyperLogLog su date u tabeli 1.8.

Ime naredbe	**Opis naredbe**
PFADD	Dodaje elemente u HyperLogLog.
PFCOUNT	Vraća približnu vrednost jedinstvenih elemenata unutar HyperLogLog vrednosti.
PFMERGE	Pripaja HyperLogLog vrednosti.

Tabela 1.8 Naredbe za rad sa HyperLogLog tipom podatka

1.2 Postojanost podataka

Jedna od glavnih karakteristika koja Redis izdvaja u odnosu na druge baze podataka je to što podatke čuva u RAM memoriji. Pristup RAM memoriji je znatno brži od pristupa sekundarnoj memoriji, pa su zato i operacije nad podacima znatno brže.

RAM memorija je nestalna i svi podaci bi nestali ukoliko bi server morao ponovo da se pokrene, ukoliko dođe do pada sistema i slično. Redis nudi dva načina trajnog čuvanja podataka u sekundarnoj memoriji: pomoću kreiranja i čuvanja „snimka“ baze na disk (engl. snapshotting) i pomoću upisivanja svih izvršenih naredbi pisanja u posebnu datoteku na disku (engl. append-only file, *AOF*). Može se koristiti jedan mehanizam, kombinacija oba ili nijedan od njih, zavisno od slučaja korišćenja.

Osim što ti mehanizmi imaju za posledicu trajno čuvanje podataka, takođe se dobija mogućnost replikacije kopija baze na druge mašine, čime bi se dobilo i na performansama i na još većoj pouzdanosti u odnosu na onu koju dobijamo čuvanjem na samo jednoj mašini.

Snimak baze

Snimkom baze u Redisu pravi se datoteka koji predstavlja podatke smeštene u bazi do tog trenutka. Postoji više načina izvođenja snimka baze:

- Pomoću naredbe SAVE koja je blokirajuća naredba i stopira sve upite ka Redisu dok se snimanje ne završi (fragment kôda 1.1);
- Pomoću naredbe BGSAVE (background save) koja ne zaustavlja rad servera prilikom snimanja podataka. Ona kreira dete proces (fork) koji se bavi snimanjem podataka, dok roditelj proces sve vreme prima naredbe od klijenata i izvršava ih.

```
SAVE broj_sekundi broj_promena
```

Fragment kôda 1.1 Oblik naredbe SAVE

Naredba SAVE, sa argumentima broj_sekundi i broj_promena, kaže da će se čuvanje trenutnog stanja baze pokrenuti automatski na svakih broj_sekundi ukoliko se u tom intervalu desio broj_promena operacija pisanja od poslednje uspešno izvršene naredbe SAVE. U datoteci koja sadrži podešavanja Redisa može biti navedeno više takvih naredbi istovremeno, i čim se prva od njih ispuni, pokreće se proces pisanja datoteke.

Koliko god često podešavali iniciranje čuvanja snimka skladišta, nije zagarantovano da neće biti izgubljenih podataka. Ukoliko dođe do stopiranja Redis servera, sve promene nad postojećim, i svi novokreirani podaci nakon poslednjeg snimka će biti izgubljeni. Još jedna mana ovog mehanizma je što prilikom BGSAVE naredbe kreiranje procesa deteta može za cenu imati potpuno stopiranje sistema ukoliko imamo posla sa velikom količinom podataka koja ne ostavlja puno memorije za dodatno kreiranje procesa deteta. U tom slučaju, zbog bitke za resurse, sâm proces čuvanja snimka trenutnog stanja Redis skladišta traje duže, i dolazi do značajnog pogoršavanja performansi sistema. Kako bi se to izbeglo, može se koristiti varijanta sa SAVE blokirajućom naredbom, ili se može potpuno isključiti automatsko iniciranje snimka i pokretati se ručno onda kada je najmanje frekventno korišćenje baze.

AOF

AOF predstavlja mehanizam upisivanja u datoteku svake od naredbi pisanja u bazu, redosledom kojim se izvršavaju. Time se kompletna baza može rekreirati tako što se izvrše naredbe iz datoteke. Prilikom pisanja na disk, podaci se prvo upisuju u tzv. *bafer*, a operativni sistem zatim u nekom trenutku uzima podatke i upisuje ih na disk. Pošto postoji vremenski razmak između te dve operacije, postoji i šansa da dođe do gubljenja podataka ukoliko u međuvremenu dođe do gašenja servera. Zato postoje opcije sinhronizacije koje podešavaju koliko često će se vršiti sinhronizacija bafera i diska: prilikom svake naredbe, svake sekunde ili bez podešavanja (operativni sistem bi na svoj način vršio sinhronizaciju).

Najsigurnija opcija vršenja sinhronizacije je prilikom svake naredbe, ali je ona najsporija i zavisi od performansi diska (npr. za SSD je kobno često upisivanje malog broja informacija). Optimalna opcija koja daje kompromis onoga što nudi i onoga što odnosi je opcija sinhronizovanja bafera i diska svake sekunde – ne opterećuje se disk, a u slučaju pada servera su izgubljeni podaci pristigli u toku jedne sekunde.

Međutim, *AOF* datoteka može značajno narasti, s obzirom da se u nju dodaje na kraj svaka izvršena naredba. Takođe, prilikom ponovnog pokretanja Redis servera, rekreiranje podataka iz datoteke *AOF* može potrajati ukoliko je datoteka dugačka. To se rešava podešavanjem opcije za prepisivanje datoteke u drugu, spajajući više srodnih naredbi u jednu. Tu se javlja isti problem kao kod snimka baze i kreiranja procesa deteta koji radi prepisivanje.

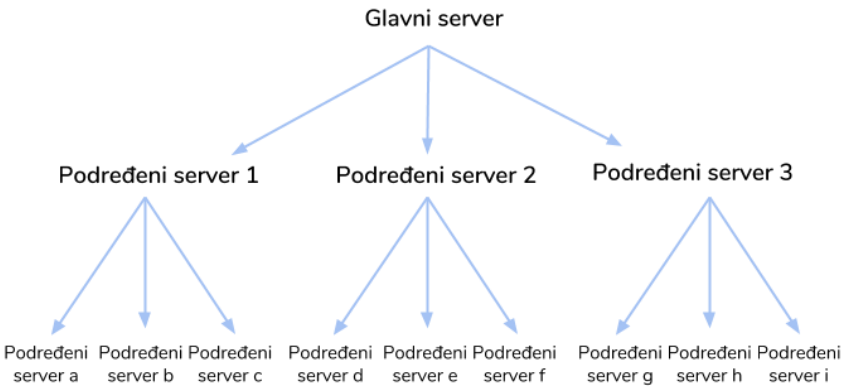
1.3 Replikacija Redisa

Replikacija baze podataka je metoda kojom se postiže njeno skladištenje na više lokacija. Arhitektura sistema u kom se koristi replikacija Redisa predstavlja arhitekturu glavni/podređeni (engl. master/slave). Služi u svrhe skalabilnosti, tako što sve operacije čitanja obavljaju replike, dok master obavlja operacije pisanja.

Kada se podređeni Redis server poveže na glavni Redis server, glavni server započinje BGSAVE operaciju. Kada završi, počinje sa slanjem snimka baze podređenoj instanci Redis servera. Podređena instanca tada briše sve podatke koje je sadržala i počinje sa učitavanjem pristiglog snimka baze. Ukoliko je u intervalu od startovanja BGSAVE naredbe, do primanja snimka baze na replici, bilo nekih operacija pisanja, glavna instanca ih čuva u datoteci i šalje podređenoj instanci, koja ih kasnije izvršava. Kao rezlultat, glavna i podređena instanca servera sadrže iste podatke.

Nakon što podređeni server primi prvi put snimak baze podataka, on se održava ažurnim pri svakom pisanju na glavni server. Klijent koji ima potrebu za čitanjem, povezuje se na podređeni server, i tako smanjuje opterećenje na glavnom serveru. Opterećenje glavnog servera se može umanjiti i isključivanjem opcije za trajno čuvanje podataka, kako ne bi trošio vreme na ulazno/izlazne operacije.

Kada su zahtevi za čitanjem znatno veći od zahteva za pisanjem, preporučuje se dodavanje novih podređenih Redis instanci, čime se smanjuje opterećenje postojećih servera. Međutim, tako se može doći u situaciju gde jedan glavni server ne može da izvršava brzo pisanja na podređene servere. Tada se preporučuje ubacivanje novog sloja čvorova glavni/podređeni, koji pomaže u podeli dužnosti pri replikaciji (slika 1.1).



Slika 1.1 Primer drvolike replikacije Redis servera

Replikacija, u kombinaciji sa *AOF* metodom trajnog čuvanja podataka, se pokazala kao stabilan model otporan na pad sistema i gubljenje podataka.

Topologija koja je inicijalno dizajnirana za Redis sistem sastojala se iz Redis instanci u kom glavna instanca servera prima sve naredbe pisanja i replicira ih na podređene instance servera. Takva topologija radi dobro u slučajevima kada:

- Master ima dovoljno memorije da skladišti sve podatke;
- Prihvatljivo je zaustaviti izvršavanje aplikacije koja koristi Redis, kada je potrebno zaustaviti glavni Redis server.

Ali ne i u slučajevima kada:

- Količina podataka premašuje memoriju dostupnu na glavnom serveru;
- Izvršavanje aplikacije koja koristi Redis ne sme biti zaustavljeno;
- Potrebno je distribuirati podatke;
- Nije prihvatljivo otkazivanje čitavog sistema usled otkazivanja Redis servera.

Da bi odgovorili na takve izazove, razvijeni su projekti Redis Klaster i Redis Sentinel. Redis Klaster ima za cilj distribuiranje podataka i automatsko rešavanje problema sa otkazivanjem rada glavnog servera. Redis Sentinel ima za cilj da pruži automatsko rešavanje problema otkazivanja glavnog servera u glavno/podređenom sistemu Redis instanci.

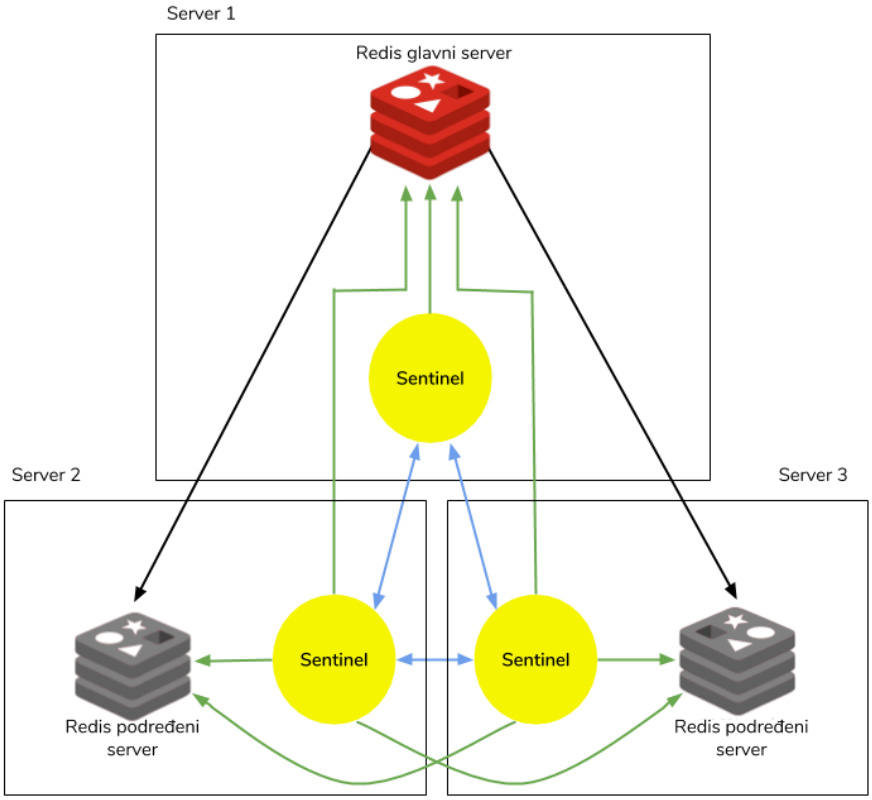
Redis Sentinel

Redis Sentinel je distribuirani sistem čija je namena olakšavanje upravljanja Redis instancama i pružanje visoke dostupnosti. Mogućnosti Sentinela su:

- Nadgledanje (engl. monitoring) – Sentinel konstantno proverava da li sve instance Redis servera rade kao što je očekivano;
- Obaveštenja – Sentinel može da obaveštava drugi program ili administratora sistema, pomoću nekog programskog interfejsa, da je došlo do neželjenog stanja sa nadgledanim Redis instancama;

- Automatsko rešavanje otkazivanja glavnog servera – Sentinel može pokrenuti proces rešavanja problema otkazivanja rada glavnog servera, tako što podređeni server unapredi u glavni;
- Podešavanje – Sentinel se ponaša kao izvor informacije o tome koji je glavni server zadužen za određeni servis potreban klijentu.

Ukoliko sistem sadrži jednu glavnu instancu i dve podređene instance servera, Sentinel se može koristiti na način predstavljen na slici 1.2.



Slika 1.2 Svaka Redis instanca ima po jedan Sentinel, Sentineli komuniciraju međusobno, kao i sa Redis instancama

Automatsko rešavanje otkazivanja glavnog servera

Automatsko rešavanje otkazivanja glavnog servera radi tako što posmatra Redis servere – glavne i podređene – i pomoću *publish/subscribe* mehanizma proverava da li su oni ispravni. Svaki Sentinel je zadužen za posmatranje određenog broja glavnih i podređenih instanci. Kada glavni server otkaže, prvo će biti izabran jedan Sentinel, koji će potom od podređenih instanci izabrati novu glavnu instancu. Nakon toga, podređene instance će se preusmeriti na novu glavnu instancu.

Sentinel je praktično specijalni režim izvršavanja Redis servera i može se pokrenuti na dva načina koja su semantički ista (fragmenti kôda 1.2 i 1.3).

```
redis-sentinel sentinel.conf
```

Fragment kôda 1.2 Pokretanje Sentinela pokretanjem redis-sentinel procesa

```
redis-server sentinel.conf –sentinel
```

Fragment kôda 1.3 Pokretanje Sentinela pokretanjem redis-server procesa

Redis Sentinel za nadgledanje sistema

Sentinel naredba za nadgledanje Redis sistema (fragment kôda 1.4) sastoji se iz parametara koji govore o serveru za nadgledanje - njegova IP adresa i port, i celobrojnog parametra *kforum*. Kforum je broj Sentinela potrebnih da se slože da glavni server nije dostupan. Kforum se koristi samo za detektovanje takvog stanja, a ne i za izvršavanje procedure oporavka (engl. failover).

```
sentinel monitor server IP port kforum
```

Fragment kôda 1.4 Naredba za nadgledanje Redis servera

Za sâm oporavak, jedan od Sentinela mora biti izabran za lidera, i to ukoliko ima glasove većine Sentinel procesa. Na primer, ukoliko postoje 5 Sentinel procesa, a kforum je vrednosti 2:

- Ukoliko se 2 Sentinela u isto vreme slože da je glavni server nedostupan, jedan od njih će pokušati da izvede oporavak;
- Ukoliko su najmanje 3 Sentinela dostupna, oporavak će se izvršiti.

Podešavanja Sentinela za nadgledanje navode se u sentinel.conf datoteci (fragment kôda 1.5). Podređene instance se automatski pronalaze, i nema potrebe navoditi ih.

```
sentinel monitor master 127.0.0.1 6379 2
sentinel down-after-milliseconds master 60000
sentinel failover-timeout master 900000
sentinel can-failover master yes
sentinel parallel-syncs master 1
```

Fragment kôda 1.5 Primer sadržaja sentinel.conf datoteke, naredbe su obično oblika: sentinel ime_opcije ime_servera vrednost_opcije

Redis Sentinel nije konzistentan u situacijama kada dođe do gubljenja veze između podređenih instanci i glavne instance. Pretpostavimo da svaka Redis instanca ima svoj Sentinel, da u sistemu postoji jedan glavni i dva podređena servera i da je klijent povezan na glavni server na koji piše. Ukoliko dođe do problema sa vezom između glavne i podređene instance Redis servera, a podređene instance i dalje mogu da komuniciraju, jedna od podređenih instanci će pomoću Sentinela biti promovisana u glavni server. Ali klijent je i dalje povezan na stari, izolovani glavni server. Kada se Redis serveri opet povežu, Sentineli će za glavnu instancu, takvog „novog“ sistema izabrati već izabrani glavni server od podređene instance, a stari,

izolovani glavni server, će biti njegov podređeni. Iz tog razloga, sve promene nastale u tom periodu biće izgubljene.

Redis Klaster

Redis klaster rešava prethodno navedeni problem. Redis klaster je distribuirana implementacija Redisa sa sledećim ciljevima:

- Visoke performanse i skalabilnost;
- Visok stepen sigurnosti operacije pisanja;
- Visoka dostupnost.

Čvorovi distribuiranog sistema su zaduženi za čuvanje podataka i stanja o Klasteru, kao i za mapiranje ključeva ka čvorovima. Čvorovi Klastera su u mogućnosti da automatski otkrivaju prisustvo drugih čvorova, da detektuju one koji ne rade, i unapređuju podređene čvorove u glavne kada je to potrebno. Da bi uspeli da izvrše ove zadatke, svi čvorovi su međusobno povezani TCP magistralom i binarnim protokolom koji se naziva *Redis Klaster Magistrala* (engl. Redis Cluster Bus). Čvorovi koriste tzv. trač protokol (engl. gossip protocol) za razmenjivanje informacija.

U distribuiranju podataka kod Redis klastera, svaki ključ je deo *heš slota*. Postoji 16384 heš slota u Redis klasteru. Da bi se odredilo kom heš slotu pripada određeni ključ, računa se CRC16[1] ključa, po modulu 16384. Svaki čvor u Klasteru je zadužen za svoj podskup heš slotova. To omogućava lako dodavanje čvorova u sistem, kao i njihovo lako uklanjanje. Na primer, ako u Klasteru postoje 3 čvora, čvor *A* sadrži heš slotove od 0 do 5500, čvor *B* sadrži slotove od 5501 do 11000, i čvor *C* sadrži heš slotove od 11001 do 16383. Da bi se dodao novi čvor *D* u sistem, potrebno je pomeriti neke heš slotove sa čvorova *A*, *B* i *C* na njega. Takođe, pri brisanju čvora *A*, potrebno je njegove heš slotove premestiti na čvorove *B* i *C.* S obzirom na to da pomeranje heš slotova sa čvora na čvor ne zahteva stopiranje operacija sa klijenata, ne postoji nedostupnost sistema ni u jednom trenutku.

Redis klaster podržava operacije nad više ključeva, sve dok ključevi pripadaju istom heš slotu. Postoji mogućnost ručnog postavljanja ključeva u isti slot, koristeći koncept *heš tagova.* Da bi se implementirali heš tagovi, metoda računanja heš slota za ključ je drugačija. Ukoliko ključ sadrži podstring oblika „{…}“, samo će podstring između `{` i `}` biti korišćen za određivanje pripadnosti heš slotu. Ukoliko postoji više takvih podstringova, koristi se sledeći algoritam:

1. Ako ključ sadrži karakter `{`,
2. I ako sadrži `}` nakon karaktera `{`,
3. I ako postoji barem jedan karakter između prvog pojavljivanja karaktera `{` i prvog pojavljivanja karaktera `}`,

onda se za heširanje ključa koristi podstring između prvog pojavljivanja `{` i prvog pojavljivanja `}`.

[1] Metoda cikličke provere redundanci (engl. Cyclic Redundancy Checking, CRC) počiva na aritmetici po modulu 2 i deljenju polinoma.

Postoje dve vrste čvorova Klastera: glavni i podređeni čvorovi. Kako bi sistem ostao dostupan kada glavne instance otkažu ili izgube vezu ka većini čvorova, Redis klaster koristi glavno/podređeni replikaciju, tako da svaki heš slot ima bar jednu svoju repliku. U korišćenom primeru sa Klasterom koji sadrži čvorove *A*, *B* i *C*, ukoliko server *B* otkaže, otkazuje i ceo Klaster, jer ključevi sa vrednostima smešteni na heš slotovima od 5501 do 11000 više nisu dostupni. Međutim, ukoliko se svaki glavni čvor replicira na jedan podređeni čvor, Klaster će moći da nastavi da radi u konzistentom stanju, i ključeve sa heš slotova smeštenih na čvoru *B*, će sadržati njegova replika.

1.4 Primeri upotrebe Redis platforme

Redis platforma može služiti u razne svrhe. Redis pripada grupi nerelacionih baza podataka, i može se koristiti kao tradicionalna baza podataka, mada joj to nije primarna namena. Najčešći primeri korišćenja Redisa su u vidu sekundarnog skladišta za podatke, gde Redis služi kao keš, ili za komunikaciju između mikroservisa ili procesa.

1.4.1 Redovi zadataka i slanje poruka

Jedan od načina za međuprocesnu komunikaciju je *publish/subscribe* obrazac dizajna u kom pošiljaoci (eng. publisher) šalju poruke na imenovani kanal, dok se slušaoci (engl. subscriber) pretplaćuju na kanale. Svi klijenti koji slušaju određeni kanal, primaju sve poruke poslate na kanal dok su povezani na njega. Naredbe koje se koriste za *publish/subscribe* u Redisu su date u tabeli 1.9.

Ime naredbe	Opis naredbe
SUBSCRIBE	Prijavljuje klijenta na zadate kanale.
UNSUBSCRIBE	Odjavljuje klijenta sa zadatih kanala.
PUBLISH	Šalje poruku na zadati kanal.
PSUBSCRIBE	Prijavljuje klijenta na kanale koji zadovoljavaju zadati šablon.
PUNSUBSCRIBE	Odjavljuje klijenta sa kanala koji zadovoljavaju zadati šablon.

Tabela 1.9 Naredbe za rad sa redovima zadataka

Jednom prijavljen na kanal, klijent prelazi u poseban režim rada i nijedna naredba neće biti obrađena, osim naredbi čitanja, dok klijenti koji samo emituju poruke nemaju takvo ograničenje. Na jedan kanal se može prijaviti više klijenata u istom trenutku.

Ovakva implementacija *publish/subscribe* mehanizma može biti korisna, ali ima svoje mane. Jedna od njih je nepouzdanost. Ukoliko dođe do problema sa mrežom kod klijenta koji sluša poruke, u trenutku stizanja poruke na kanal, poruka će zauvek biti izgubljena za njega. Alternativu ovom obrascu daju nam metode navedene u sledećem odeljku.

Redis kao posrednik u slanju poruka pomoću *pull* mehanizma

Postoje dva najčešća načina komunikacije između klijenata iz perspektive kako se prosleđuje poruka. Jedan on njih je već naveden (*publish/subscribe* mehanizam). Drugi način se zove povlačenje poruke (engl. *pull*) i on zahteva da primaoci poruke dohvate poruku.

Tu razlikujemo dva scenarija – kada postoji tačno jedan primalac i kada postoji više primalaca poruke.

Kada tačno jedan klijent prima poruku rešenje je jednostavnije, i za baratanje porukama najbolje je koristiti Redis listu. Lista je pogodna zato što podržava naredbe za dohvatanje i umetanje elemenata na početak i kraj i naredbu BLPOP koja je blokirajuća. Stavljanje poruke u listu i čitanje iz nje se najčešće radi po principu *first in-first out*, pa se stoga za pisanje koristi RPUSH, a za čitanje BLPOP. Klijent ne mora biti dostupan u trenutku slanja poruke jer ga poruke čekaju skladištene u listi.

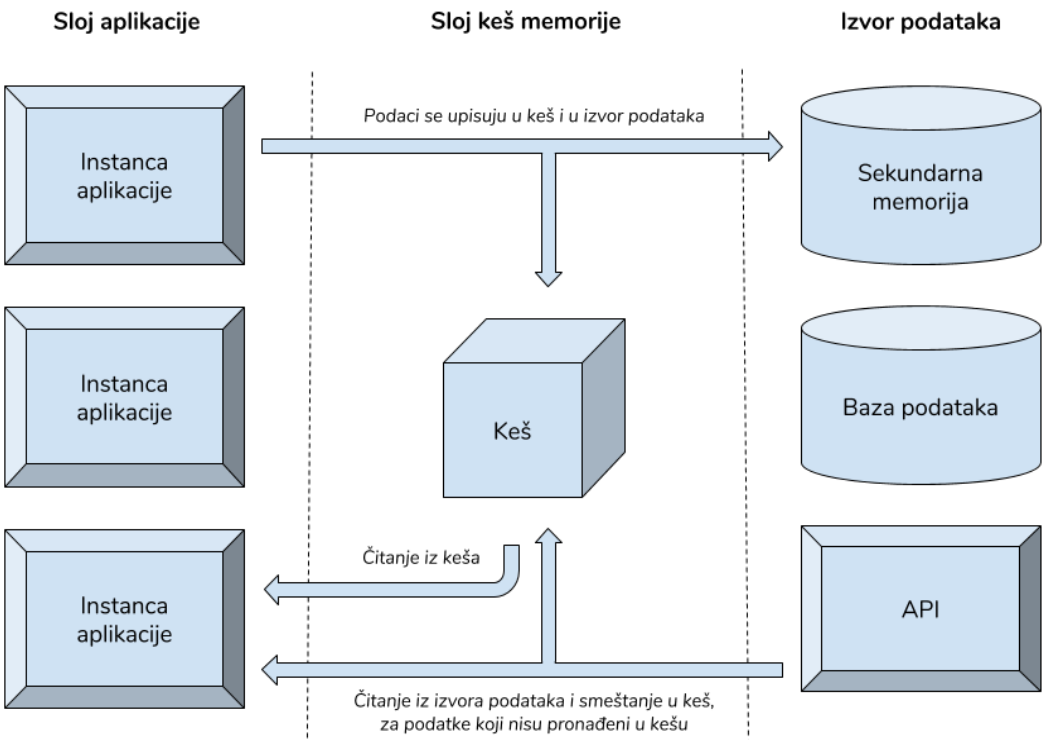
Problem kada više klijenata treba da primi poruku može se posmatrati kao grupna prepiska – svaka grupna prepiska treba da ima skup primaoca grupnih poruka, korisnici (klijenti) mogu da izlaze i ulaze u prepisku kada žele, a ne moraju stalno imati pristup mreži. Sada je najpogodnije korisiti ZSET strukturu podataka gde će vrednost činiti ime korisnika (klijenta) - primaoca poruke, a poziciju će činiti najviši identifikator poruke koju je korisnik primio u prepisci. Pored njega potrebna su nam još dva sortirana skupa kako bi podaci bili potpuni. Prvi čini informacije o konkretnom korisniku – grupe kojima korisnik pripada, a kao poziciju identifikator poslednje poruke koje je korisnik primio u toj prepisci. Drugi sadrži samu poruku i njen identifikator kao poziciju. Akcija slanja poruke sastoji se iz sledećih koraka – generiše se novi identifikator poruke i poruka se dodaje u ZSET grupne prepiske. Pribavljanje nepročitanih poruka nekog korisnika vrši se tako što se iz njegovog prvog skupa dohvate identifikatori grupa i poziciju poslednje pribavljene poruke, kojim se zatim iz odgovarajućih sortiranih skupova koji sadrže poruke dohvate one nepročitane, pomoću naredbe ZRANGEBYSCORE. Zatim se ZSET korisnika ažurira tako da pokazuje na novu najvišu poziciju, a poruka se briše iz grupne prepiske ukoliko su je svi članovi primili. Akcija pristupanja grupnoj prepisci sastoji se iz postavljanja potrebnih referenci – u korisnikov ZSET se dodaje identifikator prepiske i najvišu poziciju poruke iz nje, a u ZSET prepiske dodaje se identifikator korisnika kao člana. Kada korisnik izlazi iz grupne prepiske, potrebno je izvršiti nekoliko akcija:

- Uklanja se korisnikov identifikator iz ZSET skupa prepiske;
- Uklanja se identifikator prepiske iz korisnikovog ZSET skupa;
- Ukoliko nema više korisnika u prepisci, ZSET koji predstavlja prepisku se briše;
- Ukoliko ima korisnika u prepisci, uklanjaju se poruke pročitane od strane svih korisnika.

1.4.2 Redis kao keš

Keširanje u kontekstu veb aplikacija je tehnika najčešće korišćena da ubrza vreme odgovora aplikacije. Tipičan serverski proces sastoji se od primanja klijentskog zahteva, procesiranja zahteva, pripremanja odgovora i vraćanja odgovora. Pripremanje odgovora, koje je iole

komplikovanije od samog korišćenja ulaznih podataka kojima proces odmah ima pristup, zahteva generisanje dodatnih podataka koji dovode do krajnjih izlaznih podataka u odgovoru. Primarna memorija Redis servera (RAM memorija) se koristi za skladištenje podataka ali prima samo konačnu i relativno malu količinu podataka, nestalna je, često nedeljiva i skupa. Stoga se podaci skladište u eksternim memorijama radi skalabilnosti, trajnosti, konkurentnosti i štednje resursa.



Slika 1.3 Arhitektura aplikacije koja koristi keš

Bez upotrebe keša, aplikacija interaguje sa eksternim izvorima za svaki zahtev, dok sa upotrebom keša samo je jedan zahtev ka eksternom izvoru potreban, a za svaki sledeći interaguje sa kešom (kao što je prikazano na slici 1.3 – aplikacija pokušava da pročita podatke iz keša, ako ga tu ne nađe obraća se eksternom izvoru podataka koji može biti sekundarna memorija, baza podataka, neki aplikativni interfejs itd.).

Podaci koji se često keširaju su:

- Podaci vezani za internacionalizaciju i lokalizaciju, potrebni svakom zahtevu tj. korisniku;
- Šabloni (engl. *template*) koji su nepromenljive prirode i zahteva ih svaki upit ka aplikaciji kao što su na primer HTML fragmenti strane;
- Ponovno iskoristivi rezultati niza komplikovanih operacija (koje zahtevaju više poziva ka eksternim izvorima), tako da se ti rezultati ne moraju iznova izračunavati već se mogu naći u kešu;

- Podaci o sesiji koji su potrebni u obradi svakog zahteva, budući da sadrže podatke o korisniku. U tom slučaju, ne gubi se fleksibilnost da bilo koji aplikacioni server može obraditi zahtev;
- Podaci iz baze podataka: podskupovi tabela/dokumenata ili rezultati kompleksnih upita;
- Podaci potrebni za unutar-procesnu komunikaciju. Aplikacije čija je arhitektura mikroservisna, ili aplikacije koje zovu interfejse drugih aplikacija (engl. *application programming interface, API*), mogu odgovore od aplikativnog interfejsa keširati, pa makar i ukoliko su oni potreban kratko, čime se performanse aplikacije značajno mogu poboljšati;
- Objekti iz aplikacije čiji se podaci nalaze u raznim izvorima podataka.

Poništavanje keševa je proces u kome se podaci iz keša brišu ili zamenjuju drugim podacima. U slučaju da je primarni izvor podataka neki drugi medijum, npr. relaciona baza podataka, a da Redis sadrži jedan njihov deo, potrebno je imati odgovarajući sistem osvežavanja podataka u Redisu ukoliko dođe do problema sa Redis serverom. Takav problem može dovesti do nekonzistentnosti podataka, recimo ukoliko je pri nekoj operaciji pisanja u Redis server nedostupan, i time izgubi neke od promena i sadrži netačne podatke.

Kod upotrebe keš tehnologija, važno je odabrati strategiju poništavanja keševa, povoljnu za svrhe aplikacije. Postoje dva osnovna principa poništavanja keševa:

- Ažuriranje podatka u Redisu kada god se ažurira vrednost u primarnom izvoru podataka;
- Brisanje podatka iz Redisa kada god se ažurira vrednost u primarnom izvoru podataka, i ažuriranje vrednosti u Redisu pri prvom čitanju podatka.

1.4.2.1 Redis kao LRU keš, ključevi sa rokom trajanja

Kada se Redis koristi kao keš, korisno je automatski brisati stare podatke pri dodavanju novih. LRU (engl. *last recently used*) algoritam je jedna od podržanih od strane Redisa. Naredbom EXPIRE se ključu postavlja rok trajanja, tako da se nakon isteka roka trajanja ključ automatski briše.

Politike brisanja koje su dostupne su:

- Noeviction: vraća grešku kada je granica dostupne memorije dostignuta, a klijent pokušava da izvrši naredbe koje bi rezultirale povećanjem korišćene memorije;
- Allkeys-lru: briše ključeve počevši od najdavnije pristupanog ključa;
- Volatile-lru: briše ključeve počevši od najdavnije pristupanog ključa, ali samo u okviru ključeva koji imaju postavljenu osobinu *expire*;
- Allkeys-random: briše ključeve nasumičnim izborom;
- Volatile-random: briše ključeve nasumičnim izborom, ali samo u okviru ključeva koji imaju *expire* postavljen;
- Volatile-ttl: briše ključeve sa postavljenim atributom *expire*, redosledom dobijenim po principu kraćeg postavljenog životnog veka (engl. *time to live, TTL*).

Politike brisanja *volatile-lru*, *volatile-random* i *volatile-ttl* se ponašaju kao politika *noeviction* kad nema ključeva sa postavljenom *expire* osobinom.

Politika brisanja se bira u zavisnosti od aplikacije i može se menjati u toku izvršavanja aplikacije. Ukoliko se očekuje da neki ključevi budu posećeniji od drugih, dobar izbor je *allkeys-lru*, a ukoliko se ključevima ciklično pristupa tj. očekuje se uniformna distribucija pristupa, onda je dobar izbor *allkeys-random*. Postavljanje *expire* osobine na ključu troši memoriju, pa je politika brisanja *allkeys-lru* efikasna u pogledu memorije jer za odabir ključeva za brisanje ne koristi *expire*.
Proces brisanja odvija se na sledeći način:

1. Klijent pošalje naredbu Redis serveru, koja rezultira dodavanjem novih podataka.
2. Redis proverava da li je iskorišćena memorija veća od *maxmemory* limita, i u tom slučaju briše ključeve po izabranoj politici brisanja.
3. Nova naredba se izvršava.

To znači da se željena granica memorije pređe, a zatim se brisanjem ključeva vrati. Redisov LRU algoritam je probabilistički i ne pronalazi najboljeg kandidata za brisanje (ključ sa najstarijim vremenom poslednjeg pristupa). On uzima mali uzorak ključeva i među njima pokreće tačan LRU algoritam - vrši pronalazak, a zatim i brisanje LRU ključa. Od verzije Redisa 3.0 algoritam je poboljšan tako da bira uzorak dobrih kandidata za brisanje uz pomoć naredbe *maxmemory-samples*. Od verzije Redisa 4.0 dostupan je i LFU (eng. *least frequently used*) algoritam, koji pronalazi frekvenciju korišćenja ključeva. Ključevi koji su korišćeni ređe imaju veće šanse da budu obrisani. Pri korišćenju LRU algoritma, moguće je da se najskorije koristio ključ koji se skoro nikada više neće koristiti, i takav ključ će imati manje šanse za brisanje od nekih pogodnijih. Kod LFU takav problem ne postoji.
Za podešavanje LFU algoritma postoje 2 politike brisanja ključeva:

- *Volatile-LFU* koji briše ključeve sa postavljenim *expire* atributom;
- *Allkeys-LFU* koji briše sve ključeve.

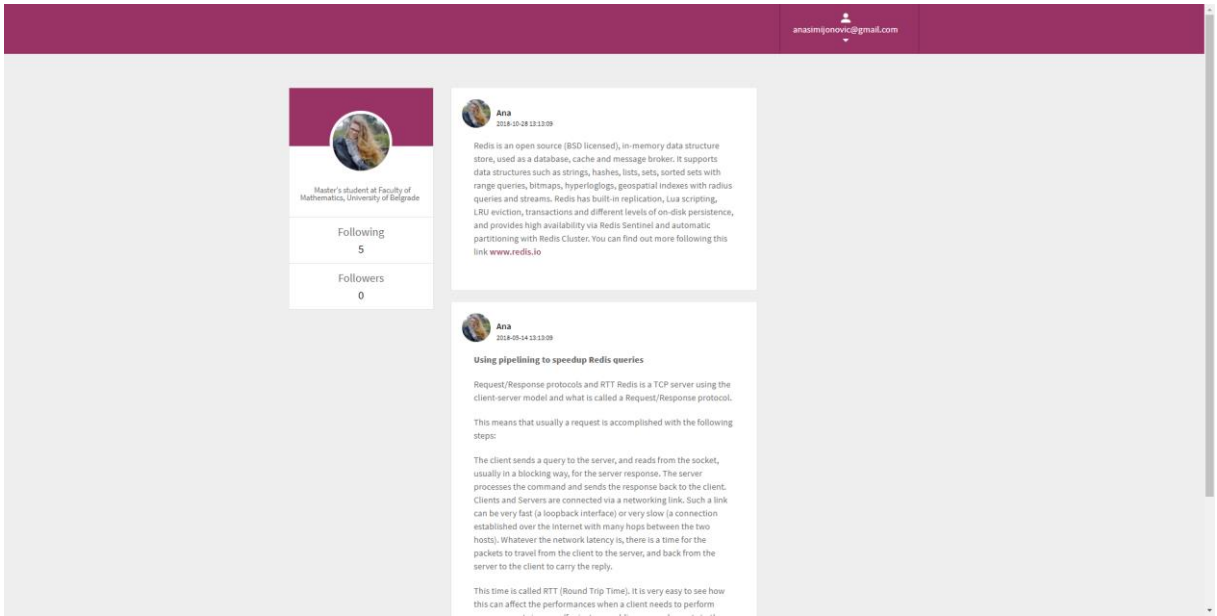
LFU algoritam je takođe probabilistički. On koristi brojač koji procenjuje frekvenciju pristupa ključu, koristeći samo par bitova po objektu. Algoritam se može prilagoditi promenama u šablonu pristupa podacima jer za računanje koristi i vreme poslednjeg korišćenja.

2 Dizajn sistema za keširanje uz pomoć Redis platforme

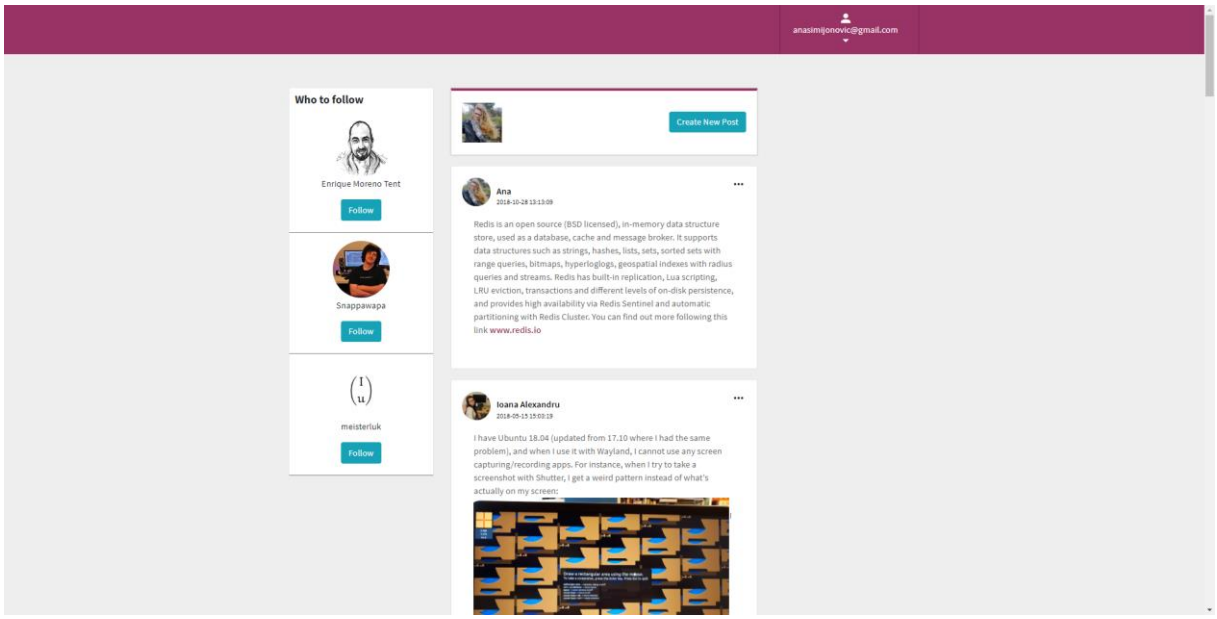
U ovom poglavlju biće opisana veb aplikacija čiji je glavni cilj predstavljanje primera upotrebe Redis platforme. Aplikacija koristi Redis platformu kao sekundarni medijum za smeštanje podataka, i to jednog njihovog dela, najčešće identifikatora objekata, kako bi aplikacija vrlo brzo mogla da dođe do njih. Kao primarni izvor podataka, aplikacija koristi relacionu bazu podataka. Svrha aplikacije je da demonstrira koliko umetanje Redis skladišta u ekosistem aplikacije može doprineti brzini i celokupnom poboljšanju performansi aplikacije.

Razvijena je aplikacija koja nalikuje na društvenu mrežu. Izabrana je takva namena aplikacije, zato što danas društvene mreže koristi veliki broj korisnika pa je potrebno sačuvati i obraditi veliku količinu podataka, tako da krajnjem korisniku upotreba aplikacije bude udobna i brza za korišćenje. Cilj je da se performanse ne pogoršavaju sa porastom broja korisnika koje pratimo na društvenoj mreži, a samim tim i sa porastom količine informacija koju je potrebno obraditi da bi se one prikazale u nekom smislenom redosledu. Uz pomoć Redis platforme moguće je odgovoriti na takve zahteve.

Korisnik aplikacije se registruje pomoću email adrese i lozinke. Ima mogućnost da prati druge korisnike i da objavljuje svoje poruke. Korisnik ima svoju profilnu stranu, gde može videti sve svoje poruke (slika 2.1), i svoju vremensku liniju (slika 2.2), gde vidi poruke korisnika koje prati. Poruke su sortirane po vremenu pravljenja, pa su na vrhu linije najnovije poruke. Postoji koncept paginacije – odjednom se učitava određeni (podesivi) broj poruka, kako na vremenskoj liniji, tako i na profilnoj strani.

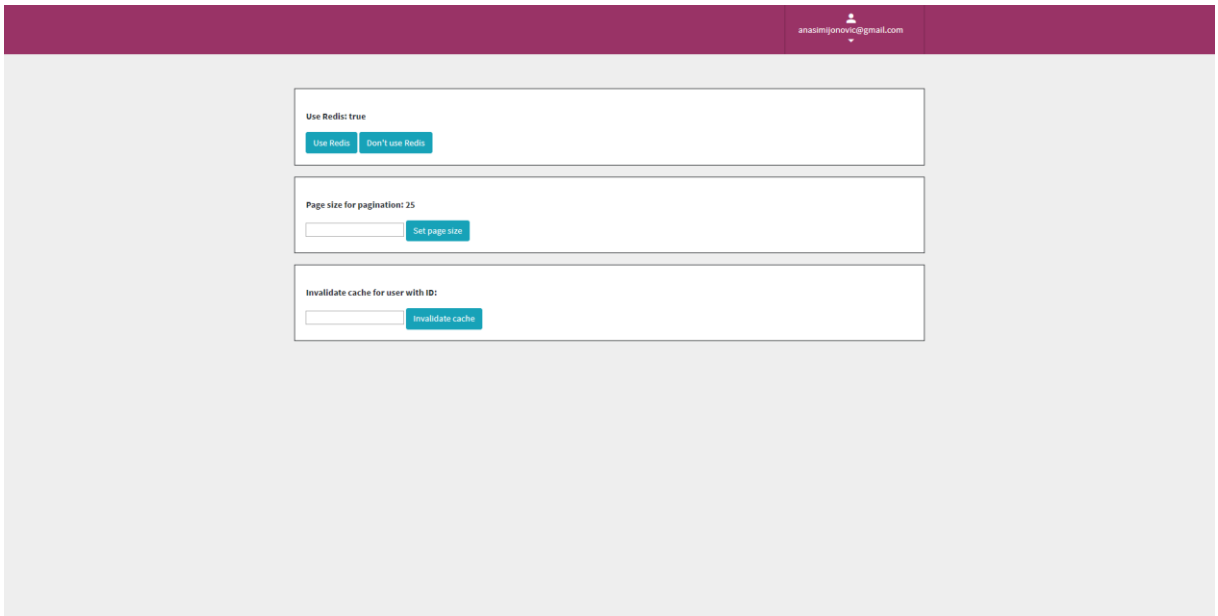


Slika 2.1 Profilna strana



Slika 2.2 Vremenska linija

Aplikacija sadrži stranu za administraciju (slika 2.3), predviđenu za korišćenje od strane programera ili administratora aplikacije. Ona nudi nekoliko opcija – dugme za podešavanje da li aplikacija treba da koristi Redis ili ne, polje za unos dužine naslovne i profilne strane i polje za unos identifikatora korisnika za koga želimo da poništimo keš. Prva opcija je ključna za merenje poboljšanja/pogoršanja brzine aplikacije. Različitim odabirom tog parametra dolazimo do različitih zaključaka oko korišćenja Redis servera, o čemu će biti reči u sekciji sa rezultatima (sekcija 4). Opcija za unos paginacije daje nam veću fleksibilnost u smislu da je moguće menjati je u toku izvršavanja aplikacije. Opcijom za poništavanje keševa postiže se osvežavanje podataka iz Redisa koji se tiču jednog korisnika. To podrazumeva brisanje onih podataka iz Redisa, koji se na njega odnose, kao i njihovo ponovno umetanje u Redis, čitanjem rezultata koje čuvamo u relacionoj bazi podataka.



Slika 2.3 Strana za administraciju

3 Implementacija sistema za keširanje uz pomoć Redis platforme

Kao što je pomenuto u prethodnom poglavlju, poruke na naslovnoj i profilnoj strani su sortirane po vremenu pravljenja, pa su na vrhu stranice najnovije poruke. Takav zahtev, iako izgleda trivijalno, može biti teško ispunjen. Trajanje izvršavanja upita koji dobavlja zapise iz baze sa određenim sortiranjem može trajati veoma dugo, u zavisnosti od količine podataka koju baza sadrži. S druge strane, takav upit može stvoriti problem sa skalabilnošću. Tu Redis pokazuje svoju moć.

3.1 Korišćene tehnologije

Razvijena je klijent – server aplikacija. Serverski deo aplikacije implementiran je u programskom jeziku Java, korišćenjem programskog okvira Spring. Upotrebljen je model-izgled-kontroler obrazac dizajna. Za izgled komponentu korišćena je JSP tehnologija (eng. *Java Server Pages*). Konekcija na Redis skladište i komunikacija sa njim postignuta je pomoću Redis klijenta Jedis, verzija 2.9.0.

Kao primarna baza podataka korišćena je MySQL relaciona baza podataka. Aplikacija joj pristupa pomoću JDBC (Java Database Connectivity) aplikativnog interfejsa. Budući da je za demonstraciju performansi aplikacije potrebna velika količina podataka, kako bi se simuliralo realno okruženje sa puno korisnika, iskorišćen je skup podataka sa adrese https://archive.org/details/stackexchange generisane na sajtu askubuntu.com. Konkretnije, uzete su tabele korisnik i poruke. Tabela korisnik ima oko 560.000 redova, a tabela sa porukama ima oko 570.000 redova.

Za autentifikaciju korisnika korišćena je Firebase baza podataka. Povezivanje sa Firebase bazom podataka dešava se na klijentskoj strani aplikacije, a zatim se sa njegovim odgovorom, klijentski deo aplikacije obraća serverskoj strani aplikacije za dalju obradu podataka.

Aplikacija je razvijena u okviru Windows operativnog sistema, u Intellij IDEA razvojnom okruženju. Korišćen je MySQL Workbench za kreiranje i održavanje MySQL baze podataka. Za Redis je korišćen FastoRedis – alat za upravljanje Redis skladištem sa grafičkim korisničkim interfejsom.

Zvaničan Redis server, razvijan od strane RedisLabs kompanije, može da se kompajlira i koristi na Linux, OSX, OpenBSD, NetBSD, FreeBSD operativnim sistemima, kao i na sistemima izvedenim iz Solaris operativnog sistema. Trenutna stabilna verzija Redisa je 4.0, objavljena jula 2017. godine. Redis server za Windows operativni sistem razvija se u okviru Microsoft Open technologes (MS OpenTech) kompanije. Trenutna verzija Redis servera za Windows OS je 3.2, objavljena u julu 2016. godine, kompatibilna sa zvaničnom verzijom Redisa 3.2.1. Većinski deo koda napisanog za Windows operativni sistem je isti, međutim postoje funkcionalnosti koje su učaurene u platformski specifične biblioteke. Najbitniji segmenti u kojima se razlikuju su mrežni aplikativni interfejs, deskriptori datoteka i fork funkcija. Najveća prepreka pri enkapsuliranju ovih karakteristika bilo je simuliranje *fork()* aplikativnog interfejsa

koji koriste POSIX verzije Redisa, a koji Redis koristi pri trajnom čuvanju skladišta na disk. Umesto kreiranja dete procesa, Microsoft koristi niti pri upisivanju na disk. U svrhe demostriranja rada Redis skladišta u ovom radu, navedene razlike nisu od ključnog značaja.

U svrhe testiranja performansi aplikacije pri opeterećenju, upotrebljen je Apache JMeter alat, čija je osnovna gradivna jedinica plan testa. Tačan plan testa biće opisan u sekciji 4.2.

3.2 Arhitektura aplikacije

Serverski deo aplikacije implementiran je u programskom okviru Spring MVC. Kada korisnik uputi zahtev ka aplikaciji, prvu ulaznu tačku čine kontrolerske klase i njihove metode. Sledeći sloj aplikacije predstavlja sloj servisa, koga direktno koriste kontrolerske metode. Sloj servisa predstavlja srednji sloj između kontrolera i sloja za pristup podacima. I na poslednjem sloju se nalaze klase koje pristupaju podacima.

Postoje 3 kontrolera: *WelcomeController*, *FeedController* i *AdminController*. *WelcomeController* sadrži metode za obradu zahteva vezanih za registrovanje, prijavljivanje i odjavljivanje korisnika, kao i za obradu zahteva vezanih za profilnu stranu korisnika. *FeedController* je zaslužan za obradu zahteva vezanih za naslovnu stranu korisnika i sadrži obradu zahteva, i njihovo dalje usmeravanje, poput prikazivanja naslovne strane, dodavanja nove poruke, dodavanja/uklanjanja korisnika za praćenje. *AdminController* obrađuje zahteve koji dolaze sa stranice za administraciju.

Sloj servisa sadrži logiku vezanu za odluke odakle i kako dopremiti podatke. Koristeći parametre koje zadaje administrator na nivou aplikacije (koja je veličina stranice, da li koristiti Redis ili samo MySQL), servis komunicira sa bazom podataka i sa Redis serverom. Klase na ovom sloju zovu se *PostService* i *UserService.*

Sloj pristupa bazi podataka sadrži klase *PostDAO* i *UserDAO*. One učauruju konekciju ka MySQL bazi podataka, i metode i upite potrebne za pribavljanje podataka iz nje.

Na istom sloju, nalazi se klasa *RedisManager*, koja sadrži metode pristupa Redis serveru i njegovim podacima. Za potrebe ove aplikacije, i prikazivanje Redis platforme u jednom od najčešćih primera upotrebe, Redis je korišćen u svrhe keširanja. Dakle, u Redis skladištu se ne čuvaju svi podaci, već jedan njihov deo. S obzirom na to da kod Redisa nema tabela i shema koje treba dizajnirati, kao kod relacionih baza podataka, treba dizajnirati koji ključevi su potrebni da predstavljaju objekte i koji tip vrednosti treba da sadrže. Paradigma kojom se treba voditi je da podaci treba da budu organizovani tako da im se može direktno pristupiti, jer u Redisu nije moguće tražiti ključ koji sadrži određenu vrednost.

Konvencija korišćena za nazivanje ključeva je: tip objekta koga predstavlja, dvotačka i identifikator objekta koga predstavlja (npr. tipObjekta:ID).

Jedan od centralnih entiteta u aplikaciji je korisnik. Korisnik može da prati druge korisnike (engl. following), korisnika mogu pratiti drugi korisnici (engl. followers). Struktura podataka prikladna za predstavljanje entiteta korisnika je skup. Elementi skupa su jedistveni, proveravanje pripadnosti elementa skupu vrši se u konstantnom vremenu. Takođe, korisna je

operacija u Redisu kojom se računa presek dva skupa. Tako se presekom *following* skupova dva korisnika nalaze njihovi zajednički prijatelji. Definicije ključeva za skupove koji sadrže informaciju o tome *ko prati korisnika (followers)* i *koga prati korisnik (following)* su oblika datog u fragmentu kôda 3.1.

```
followers:korisnikovID

following:korisnikovID
```

Fragment kôda 3.1 Definicije ključeva followers i following

Primer jednog preslikavanja ključa u vrednost, gde npr. korisnika sa identifikatorom 100 prate korisnici sa identifikatorima 50, 120 i 150, dok on prati korisnike sa identifikatorima 50 i 200, dat je u fragmentu kôda 3.2.

```
followers:100 -> 50 120 150

following:100 -> 50 200.
```

Fragment kôda 3.2 Primer mapiranja ključ -> vrednost u Redisu

Kada bi aplikacija koristila samo MySQL bazu podataka, identifikatori o *followers* i *following* bi optimalno morali da stoje denormalizovani u jednoj koloni, i onda bi dalje bili deo kompleksnog, često ugnježdenog upita, što bi značajno ugrozilo performanse. Sa korišćenjem Redisa, imamo skup identifikatora korisnika koji su nam potrebni za kreiranje upita ka MySQL bazi podataka, i uz pomoć indeksa na primarnom ključu tabele korisnik, upit izvršava veoma brzo.

Pored čuvanja informacija o korisniku, u Redisu čuvamo i informacije o porukama. Poruke koje čine vremensku liniju jednog korisnika, pripadaju korisnicima koje prati, i samom korisniku. Sortirane su rastuće po vremenu pravljenja. Kada korisnik otvori svoju vremensku liniju, prvo se učitava jedan deo svih poruka, recimo 25 poruka, u skladu sa podešenom paginacijom. Kada korisnik dođe do kraja strane, pojavi se dugme *Show more*, čijim klikom se učitava još toliko poruka itd. Upit koji dobavlja ove poruke zadat je u fragmentu kôda 3.3.

```
SELECT p.* FROM posts p JOIN users u ON p.ownerUserId = u.id
WHERE …
ORDER BY p.createdTime DESC LIMIT 25
```

Fragment kôda 3.3 Upit za bazu podataka koji dobavlja poruke za vremensku liniju

Dakle, da bi se prikazale poruke na naslovnoj strani, potrebno je sortirati ih sve da bi se izlistale po vremenu kreiranja. Bitno je napomenuti da upravo iz tog razloga vrednost paginacije nije ključna za testiranje performansi, jer se najviše vremena troši na sortiranje poruka, a daleko manje na njihovu dalju obradu.

Korišćenjem Redisa, problem se pojednostavljuje. Za učitavanje naslovne strane potrebno je znati koje poruke treba prikazati, u pomenutom redosledu. Zato je pogodno za svakog korisnika čuvati upravo identifikatore poruka, koje treba da sačinjavaju njegovu naslovnu

stranu, ulančane tako da im se lako pristupa, i da su u redosledu kreiranja. Struktura podataka najpogodnija za modelovanje ovog problema je lista. Ključ koji koristimo je *userFeed:userId*.

Svaki put kada korisnik *K* objavi poruku *P*, ažuriraju se zapisi u Redisu tako što se za svakog korisnika K_i koji prati korisnika *K*, ažurira lista *userFeed:*K_i*Id* dodavanjem identifikatora kreirane poruke *Pid* na njen levi kraj.

Svaki put kada korisnik *K* obriše poruku *P,* ažuriraju se zapisi u Redisu tako što se za svakog korisnika K_i koji prati korisnika *K*, ažurira lista *userFeed:*K_i*Id* uklanjanjem identifikatora poruke *Pid.*

Kôd koji se bavi kreiranjem nove poruke unutar klase *PostService* nalazi se u fragmentu kôda 3.4.

```
public void addPost(Post post) {
   // dodavanje u bazu podataka
   int id = PostDAO.getInstance().createPost(post);
   post.setId(id);

   // dodavanje u Redis
   try (Jedis jedis = JedisFactory.getResource()) {
      // 1. pribavljanje svih korisnika koji su pratioci
      List<Integer> followersIds = redisManager.getIdsFromRedisSet(jedis, KEY_USER_FOLLOWERS,
post.getOwnerUserId());

      // 2. dodavanje identifikatora nove poruke na ključeve koji predstavljaju korisnike koji ga prate
      for (Integer followerId : followersIds) {
         redisManager.addIdsToRedisList(jedis, KEY_USER_FEED, followerId,
Collections.singletonList(post.getId()));
      }

      // 3. dodavanje identifikatora nove poruke na ključ koji predstavlja svoju vremensku liniju
      redisManager.addIdsToRedisList(jedis, KEY_USER_FEED, post.getOwnerUserId(),
Collections.singletonList(post.getId()));
   }
}
```

Fragment kôda 3.4 Dodavanje poruke

Kada korisnik K_1 počne da prati korisnika K_2, potrebno je ažurirati skupove *following* i *followers* korisnika K_1 i K_2, tako što se identifikator K_2 doda u skup *following:*K_1*Id*, a identifikator K_1 doda u skup *followers:*K_2*Id*. Takođe, potrebno je ažurirati listu *userFeed* korisnika K_1, tako da sadrži identifikatore poruka korisnika K_2. Tim postupkom će korisnik K_1 odmah videti poruke korisnika K_2. Postupak je analogan u slučaju kada korisnik prestane da prati drugog korisnika.

```
public void follow(int userId, int userIdToFollow) {
   // dodavanje u bazu podataka
   UserDAO.getInstance().follow(userId, userIdToFollow);

   // dodavanje u Redis
   try (Jedis jedis = JedisFactory.getResource()) {
      // 1. dodavanje atributa ID korisnika K2, koga počinje da prati korisnik K1, u following skup
```

```
korisnika K1
        redisManager.addIdsToRedisSet(jedis, KEY_USER_FOLLOWING, userId, new
ArrayList<>(userIdToFollow));

        // 2. dodavanje atributa ID korisnika K1, koji počinje da prati korisnik K2, u followers skup korisnika
K2
        redisManager.addIdsToRedisSet(jedis, KEY_USER_FOLLOWERS, userIdToFollow, new
ArrayList<>(userId));

        // 3. invalidiranje userFeed liste korisnika K1
        ArrayList<Integer> followingIds = UserDAO.getInstance().getFollowingIds(userId);
        List<Integer> postIds = PostDAO.getInstance().getFeedPosts(followingIds, 0, -1)
                .stream().map(Post::getId).collect(Collectors.toList());
        redisManager.deleteKeyFromRedis(jedis, KEY_USER_FEED, userId);
        redisManager.addIdsToRedisList(jedis, KEY_USER_FEED, userId, postIds);
    }
}
```

Fragment kôda 3.5 Praćenje drugog korisnika

Kod takvih situacija, gde je potrebno izvršiti više naredbi jednu za drugom u Redisu, poželjno je smanjiti broj pozivanja Redis servera (engl. round trip). Jedan koncept je korišćenje naredbi koje primaju više parametara, kao što je dohvatanje vrednosti više ključeva odjednom, ili dodavanje/brisanje više vrednosti unutar liste (skupa) jednog ključa. Drugi koncept je korišćenje protočne obrade (engl. pipeline). Kada klijent, u ovom slučaju naša aplikacija, pošalje zahtev Redisu, on čeka odgovor pre nego što pošalje sledeći. Sa protočnom obradom, moguće je poslati više naredbi odjednom, i čekati odgovor svih naredbi kao jedan korak. Redis to omogućava tako što pravi redove poslatih naredbi u memoriji, i zato treba voditi računa da jedna serija poslatih naredbi ne bude veće veličine od raspoložive memorije. Protočna obrada je iskorišćena u metodama za dodavanje/uklanjanje elemenata u Redis listu/skup (fragment kôda 3.6).

```
public void addIdsToRedisSet(Jedis jedisClient, String prefix, int userId, List<Integer> ids) {
    final String redisKey = generateRedisKey(prefix, userId);

    Pipeline pipeline = jedisClient.pipelined();
    ids.forEach(id -> pipeline.sadd(redisKey, String.valueOf(id)));

    pipeline.sync();
}

public void removeIdsFromRedisSet(Jedis jedisClient, String prefix, int userId, List<Integer> ids) {
    final String redisKey = generateRedisKey(prefix, userId);

    Pipeline pipeline = jedisClient.pipelined();
    ids.forEach(id -> pipeline.srem(redisKey, String.valueOf(id)));

    pipeline.sync();
}
```

Fragment kôda 3.6 Dodavanje i uklanjanje identifikatora iz Redis skupa

Za dohvatanje identifikatora objava potrebnih za učitavanje naslovne strane korisnika, koristimo naredbu LRANGE, koja nam odgovara za potrebe paginacije. Parametri koje prima su ključ, početni i krajnji indeks u Listi.

```
public List<Integer> getUserFeedFromRedisWithPagination(Jedis jedisClient, int userId, int page, int pageSize) {
   String redisKeyForFeed = generateRedisKey(KEY_USER_FEED, userId);

   // bez paginacije
   if (pageSize == -1) {
      return getUserFeedFromRedis(jedisClient, userId);
   }

   return jedisClient.lrange(redisKeyForFeed, page * pageSize, page * pageSize + pageSize - 1)
         .stream()
         .map(Integer::valueOf)
         .collect(Collectors.toList());
}

private List<Integer> getUserFeedFromRedis(Jedis jedisClient, int userId) {
   String redisKeyForFeed = generateRedisKey(KEY_USER_FEED, userId);

   return jedisClient.lrange(redisKeyForFeed, 0, -1)
         .stream()
         .map(Integer::valueOf)
         .collect(Collectors.toList());
}
```

Fragment kôda 3.7 Izračunavanje identifikatora poruka koje čine vremensku liniju korisnika

Prvobitno postavljeni problem sortiranja velikog broja zapisa unutar MySQL baze podataka, da bi se prikazao samo jedan njihov mali deo na naslovnoj strani, sveli smo na problem dohvatanja konačnog i vrlo malog broja zapisa po primarnom ključu, uz pomoć Redisa. Dakle, početno postavljeni upit pojednostavljuje se uz pomoć Redis naredbe (fragment kôda 3.8) koja vraća sve potrebne identifikatore poruka.

```
LRANGE userFeed:userId pageNumber*pageSize pageNumber*pageSize+pageSize-1,
```

Fragment kôda 3.8 Naredba ka redisu koja dobavlja identifikatore poruka vremenske linije

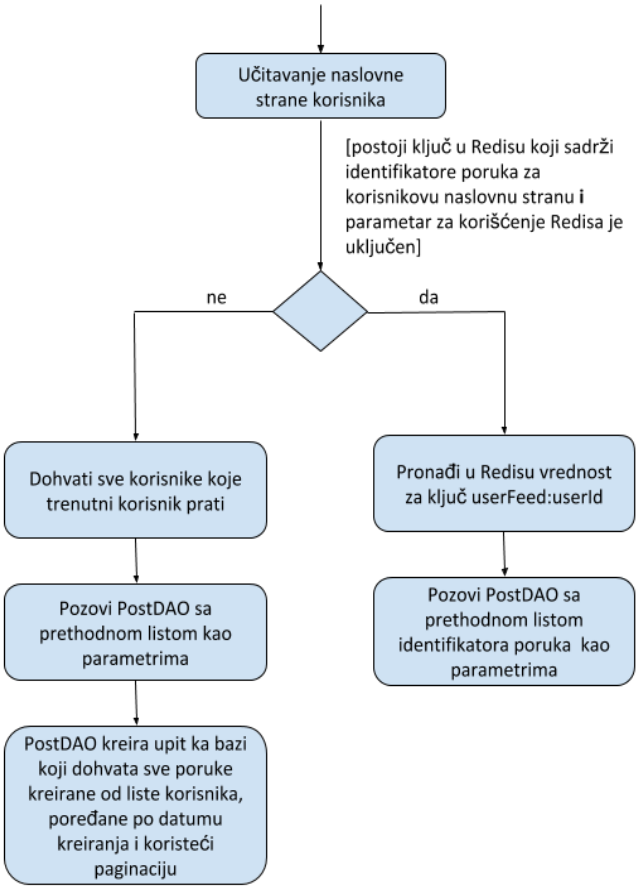
Naredba iz fragmenta kôda 3.8, za pageSize 25, kao rezultat vraća 25 identifikatora poruka, p_0id, p_2id,…, $p_{24}id$, uz pomoć kojih kreiramo upit ka primarnoj bazi (fragment kôda 3.9).

```
SELECT p.* FROM posts p JOIN users u ON p.ownerUserId = u.id
WHERE p.id IN (p0id, p2id,…, p24id)
ORDER BY p.createdTime DESC
```

Fragment kôda 3.9 Pojednostavljeni upit za bazu podataka

Dakle, Redis skladište koristimo kao keš memoriju koja ima uvek najažurnije podatke. Kada zahtev korisnika stigne do aplikacije, prvo se proverava da li je parametar za korišćenje Redisa uključen, ako jeste, zatim se proverava da li odgovarajući ključ postoji u Redisu, dohvata se vrednost ključa, i uz pomoć njega vrši se upit ka bazi podataka.

Učitavanje vremenske linije vrši se na način zadat na slici 3.1.



Slika 3.1 Algoritam učitavanja vremenske linije

Baza podataka MySQL sadrži oko 560 hiljada korisnika. S obzirom na to da u Redis skladištimo tri ključa po svakom korisniku (*userFeed*, *userFollowers* i *userFollowing*), Redis sadrži oko 1.6 miliona ključeva. Bitno je napomenuti da baza podataka nije sadržala informaciju o praćenju korisnika inicijalno, već da je ta veza naknadno uspostavljena, tako da svaki korisnik ima između 200 i 800 nasumično izabranih korisnika koja prati. Time se ostvaruje realističnost okruženja aplikacije i testa.

4 Eksperimentalni rezultati i diskusija

Jedan od ciljeva ovog rada bio je praktična primena Redis platforme u svrhe keširanja. Implementirana je veb aplikacija koja bi trebalo da demonstrira kako se u velikim sistemima, sa velikim brojem korisnika i opterećenjem, korišćenjem Redisa može znatno dobiti na performansama.

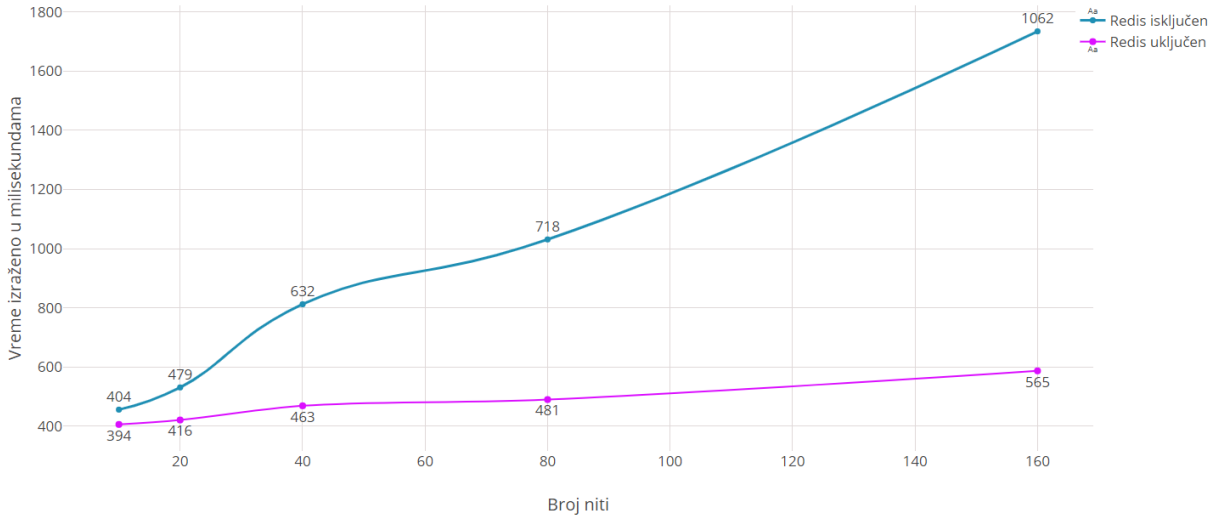
Kako bi se to eksperimentalno dokazalo, aplikacija ima dva režima korišćenja. Jedan se odnosi na podešavanje aplikacije takvo da se Redis koristi kao keš, dok drugi podrazumeva podešavanje takvo da se Redis ne koristi. Poređenjem performansi aplikacije u ta dva režima može se pokazati u kojoj meri Redis doprinosi brzini izvršavanja zahteva.

Hardverske karakeristike računara na kojem je testirana aplikacija su: procesor Intel i7-8700K 3.70GHz, sa 12 MB L3 keš memorije i 6 jezgara, i DDR4 memorija od 16GB.

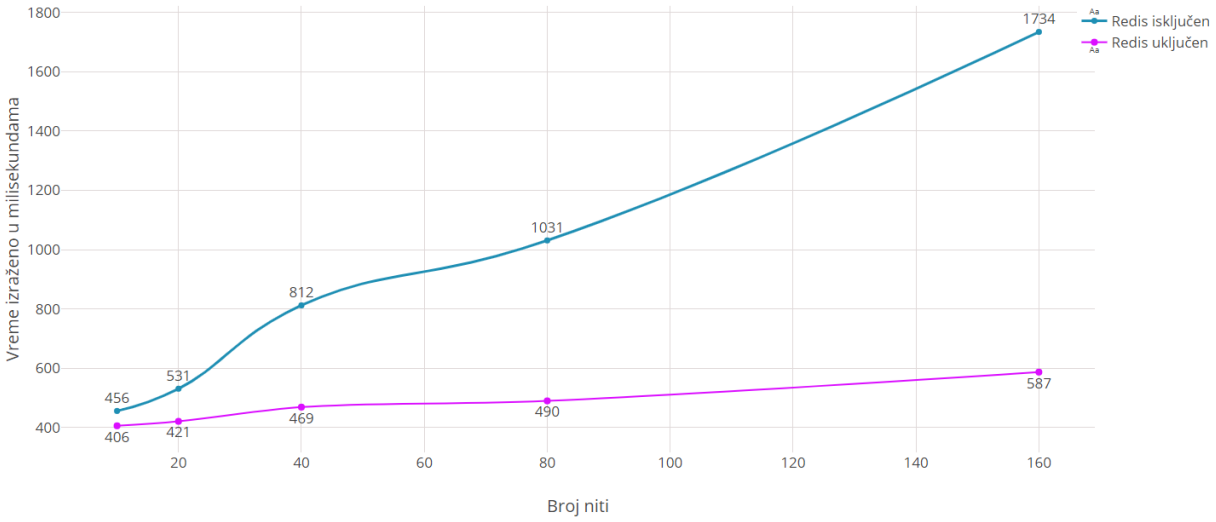
Testiranje performansi aplikacije pod opterećenjem postignuto je na dva načina opisana u sekcijama 4.1 i 4.2.

4.1 Primarni test

Primarno testiranje vršeno je pomoću konzolne Javine aplikacije. Aplikacija pokreće simultano određeni broj niti. U svakoj niti, uzima se po jedna konekcija ka MySQL bazi podataka. Zatim se nasumično bira korisnik, za kog se zatim dopremaju podaci potrebni za učitavanje vremenske linije. Informacija od ključnog značaja, dobijena iz testa, je vreme izvršavanja dopremanja podataka. Testove razlikuje broj niti koje se kreiraju, kao i parametar za uključenost Redisa. Ideja je da niti simuliraju scenario korišćenja aplikacije od strane više korisnika istovremeno, i kakav je odziv aplikacije u različitim slučajevima. Test je pokrenut za 10, 20, 40, 80 i 160 niti. Paginacija je postavljena na 25 poruka. Na slici 4.1 i slici 4.2 prikazani su grafici zavisnosti vremena izvršavanja (izraženog u milisekundama) od broja niti. Na slici 4.1 vreme predstavlja prosečno vreme izvršavanja niti, dok na slici 4.2 vreme predstavlja maksimalno vreme izvršavanja niti.



Slika 4.1 Prosečno vreme izvršavanja niti



Slika 4.2 Maksimalno vreme izvršavanja niti

4.2 Sekundarni test

Sekundarni test ostvaren je uz pomoć JMeter alata. JMeter alat se bazira na kreiranju test plana, u našem slučaju za veb aplikaciju. Test plan se sastoji iz određenih podešavanja i koraka koji JMeter izvršava. Pre svega, podešava se broj korisnika koji se simulira, koliko često oni treba da simuliraju slanje zahteva aplikaciji, i koliko dugo to treba da ponavljaju. Nakon definisanja korisnika, definišu se koraci koje oni treba da izvrše – URL adrese i parametri zahteva koje šalju serveru. JMeter pruža i mogućnost generisanja izveštaja i statistike izvršenog test plana u vidu grafika i tabela.

Na stranici za administraciju vršimo modifikacije parametara, za pokretanje istog test plana, kako bismo dobili željeno poređenje. Parametar koji modifikujemo je parametar za uključivanje/isključivanje Redis skladišta. Podešavanje parametara može se postići i kao deo test plana – to bi bio dodatni veb zahtev.

Kao u primarnom testu, testove razlikuje broj niti koje se kreiraju, kao i parametar za uključenost Redisa. Svaki test se sastoji iz kreiranja određenog broja niti, u razmaku od 0 sekundi (u isto vreme se kreiraju), i njihovi koraci će se ponavljati tri puta. Korak koji svaka nit simulira je zahtev ka aplikaciji: nasumično biranje korisnika i učitavanje početne strane. U tabeli 4.1 su dati rezultati testiranja, koja se sastoji od osam kolona. Prve tri kolone pokazuju prosečno, minimalno i maksimalno vreme potrebno za generisanje odgovora sa servera. Slede dve kolone koje su statističke mere (percentili) koje pokazuju vrednost ispod koje određeni procenat posmatranja upada: 90% uzoraka i 95% uzoraka. One su bitne jer najbolje opisuju korisničko iskustvo, za najveći broj korisnika. Vrednosti u koloni propusnost (engl. throughput) se računaju prema formuli (broj zahteva) / (ukupno vreme trajanja testa). Poslednje dve

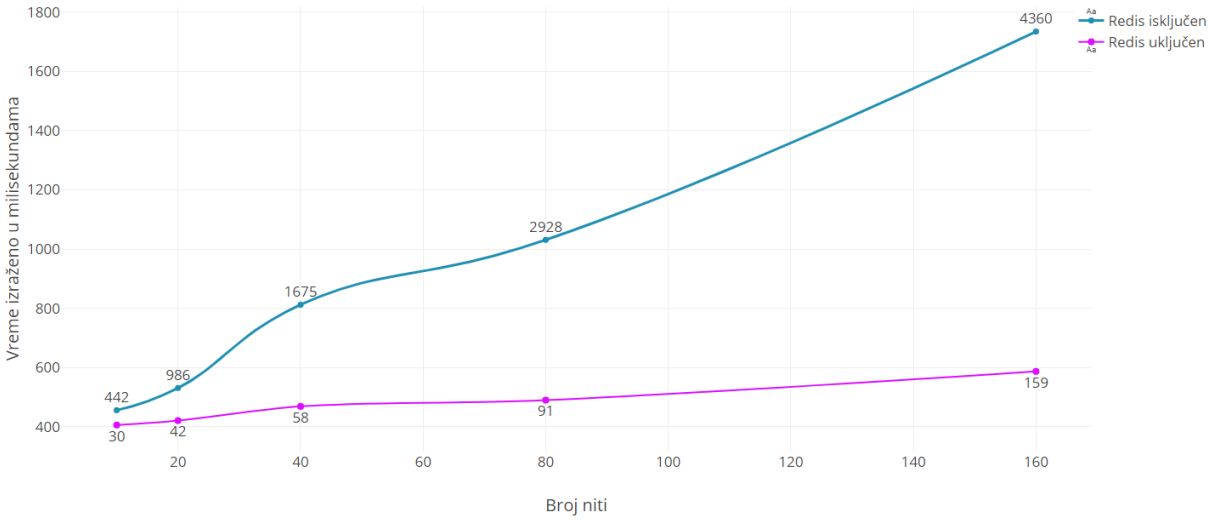
kolone su deo podešavanja unutar aplikacije i govore o tome da li je Redis uključen i sa koliko je niti opterećena aplikacija.

Testovi se pokreću za 10, 20, 40, 80 i 160 niti, sa istog računara na kom je aplikacija. Paginacija je postavljena na 25 poruka.

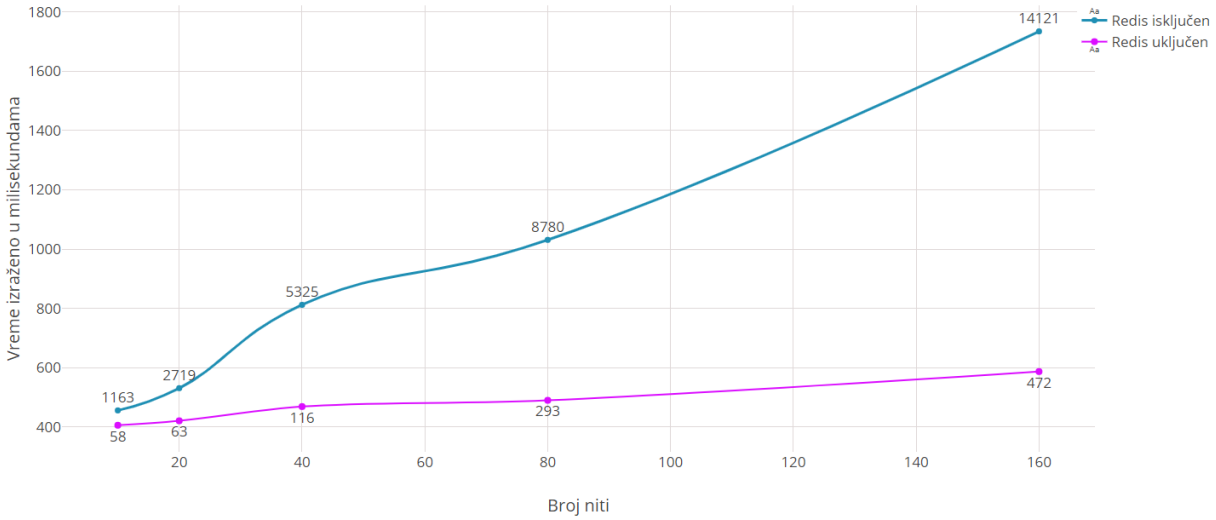
Prosečno vreme odgovora	Minimalno vreme odgovora	Maksimalno vreme odgovora	90. percentil	95. percentil	Propusnost	Redis uključen	Broj niti
442.77	52	1163	768.50	972.70	18.79	Ne	10
29.77	9	58	58	58	280.37	Da	10
985.88	7	2719	2086	2423.10	17.17	Ne	20
41.77	23	63	63	63	444.44	Da	20
1674.60	5	5323	3656.40	3776.45	19.83	Ne	40
57.99	6	116	87.90	98.80	618.56	Da	40
2927.84	30	8780	6409.20	7096.70	22.90	Ne	80
90.86	3	293	144	176	774.19	Da	80
4360.20	5	14121	9253.90	10915.65	30.07	Ne	160
159.20	5	472	312.90	373.95	867.99	Da	160

Tabela 4.1 Rezultati testiranja uz pomoć JMeter alata

Navedene rezultate objedinjujemo na slikama 4.3 i 4.4.



Slika 4.3 Prosečno vreme izvršavanja niti



Slika 4.4 Maksimalno vreme izvršavanja niti

Poređenjem prethodnih rezultata dolazimo do zaključaka. Kod primarnog, kao i kod sekundarnog testiranja, za slučaj kada je parametar za korišćenje Redisa *uključen*, evidentno je da se vreme odgovora aplikacije ne menja značajno sa povećanjem broja korisnika. Razlog tome je taj što smo podatke tako modelovali, da se do informacija potrebnih za učitavanje vremenske linije, dolazi tako što se iz Redis liste dohvati 25 identifikatora iz primarne memorije, zatim se oni potraže iz baze podataka, iz sekundarne memorije, i sortiraju po vremenu pravljenja. Dakle, ne predstavlja razliku to koliko korisnik prati drugih korisnika tj. kolika mu je teorijski vremenska linija, upravo zbog posredovanja Redisa koji sadrži tačno one podatke, koji su potrebni. Takođe, povećanjem broja niti, vreme odziva aplikacije ne raste značajno, što implicira da performanse aplikacije nisu narušene sa porastom broja istovremeno aktivnih korisnika.

Sa druge strane, kada je parametar za korišćenje Redisa *isključen,* vreme odgovora aplikacije zavisi od broja niti. Tada se podaci dopremaju isključivo iz relacione baze, bez posredstva Redis skladišta. U tom slučaju, vreme potrebno za simultano učitavanje vremenske linije nasumično odabranih korisnika, raste sa porastom broja korisnika (niti). Razlog tome je sortiranje svih poruka koje teorijski čine korisnikovu vremensku liniju, kako bi se dobilo 25 najnovije kreiranih. Takva operacija je zahtevna, jer poruka može biti u velikom broju.

Kod primarnog testa, prosečno vreme učitavanja vremenske linije korišćenjem Redisa brže je za 1,88 puta, dok je maksimalno vreme učitavanja vremenske linije korišćenjem Redisa brže za 2,95 puta. Kod rezultata dobijenih uz pomoć JMeter alata, prosečno vreme učitavanja vremenske linije brže je za 27,42 puta, korišćenjem Redisa, dok je maksimalno vreme učitavanja vremenske linije brže za 29,91 puta.

5 Zaključak

Redis skladište je različito od ostalih baza podataka u mnogo segmenata - koristi primarnu memoriju servera kao primarno skladište, a disk za trajno čuvanje podataka, radi u jednoj niti sa stanovišta izvršavanja naredbi, operacije pristupa i ažuriranja podataka su atomične, model podataka je jedinstven. Podržava kompleksne strukture podataka koje predstavljaju vrednost, dok je ključ uvek tipa string. S druge strane, svi podaci moraju stati u memoriju, nema mogućnosti kreiranja upita i podacima se uvek pristupa preko jedinstevnog ključa (skladište se ne može pretraživati po vrednosti).

Redis skladište se dobro pokazalo u raznim primenama u softverskom inženjerstvu. Može služiti u svrhe kreiranja redova zadataka, u svrhe održavanja tabele s rezultatima i rangiranja, *publish/subscribe* mehanizma, keširanja stranica, sesije i podataka. Redis takođe može služiti kao primarna baza podataka, s ograničenjem da svi podaci moraju stati u primarnu memoriju. U poređenju sa svojim konkurentima na tržištu, Redis daje dobre rezultate u terminima brzine pristupa podacima. U poređenju sa keš tehnologijama, on ima podršku perzistencije podataka i kompleksnije strukture podataka kojima se lakše i prirodnije modeluju podaci.

U ovom radu opisane su ključne osobine Redis platforme, njegove strukture podataka, zatim na koji način on vrši trajno čuvanje podataka, koje su najčešće njegove primene u praksi i na kraju, izabran je jedan od tih primera upotrebe pri implementaciji veb aplikacije. Pokazalo se da je Redis veoma koristan alat u terminima performansi softvera. Osim toga, veoma je jednostavno Redis platformu uvrsititi u ekosistem postojeće aplikacije. Tome svedoče i brojni primeri reinženjeringa velikih veb aplikacija, poput Instagrama, Twittera, StackOverflow i druge.

Glavni doprinos ovog rada je prikaz glavnih osobina Redis platforme i demonstracija istih kroz implementaciju veb aplikacije. Odabran je jedan od čestih primera upotrebe za keširanje podataka. Kao primarni medijum za smeštanje podataka korišćena je relaciona baza podataka, dok je u Redis smeštan njihov mali deo, ali dovoljan za kreiranje glavne stranice u aplikaciji – vremenske linije. Početni problem, koji se odnosio na učitavanje malog broja poruka, za koji je potrebno sortirati i obraditi mnogo veći deo podataka unutar relacione baze, sveo se na sortiranje tačno onog podskupa baze, potrebnog za učitavanje vidljivog dela vremenske linije. Time se brzina učitavanja vremenska linije, koristeći Redis, povećala značajan broj puta.

6 Literatura

[1] Victor Zakhary, Divyakant Agrawal, Amr El Abbadi, *Caching at the Web Scale,* Santa Barbara, California, 2017, https://cs.ucsb.edu/~victorzakhary/assets/papers/caching-at-the-web-scale.pdf

[2] Redis dokumentacija, https://redis.io/

[3] Josiah L. Carlson, *Redis In Action*, Manning Publications Co, Shelter Island, New York, 2013. ISBN: 9781617290855

[4] Hugo Lopes Tavares, Maxwell Dayvson Da Silva, *Redis Essentials,* Packt Publishing, Birmingham, United Kingdom, 2015. ISBN: 9781784392451

[5] RedisLabs white papers, *15 reasons to use Redis as an Application Cache,* https://redislabs.com/docs/15-reasons-caching-is-best-done-with-redis/

[6] Salvatore Sanfilippo, *Take advantage of Redis by adding it to your stack,* http://oldblog.antirez.com/post/take-advantage-of-redis-adding-it-to-your-stack.html

[7] Redis na Windows operativnom sistemu, https://github.com/MicrosoftArchive/redis/tree/win-3.2.100

[8] FastoRedis dokumentacija, https://fastoredis.com/

[9] Apache JMeter dokumentacija, http://jmeter.apache.org/

[10] Spring dokumentacija, https://spring.io/

[11] Font ikonice, https://material.io/tools/icons/

[12] MySQL dokumentacija, https://www.mysql.com/

[13] Jedis dokumentacija, https://github.com/xetorthio/jedis

7 Prilog

U prilogu navodimo ključne klase (ili neke njihove segmente) i metode za rad aplikacije, detaljnije opisane u sekcijama 2 i 3.

Klase modela su *Post* i *User*, koje opisuju centralne entitete u aplikaciji.

Klasa *Post* data je u fragmentu kôda 7.1. Ona predstavlja jednu poruku koju kreira korisnik.

```
public class Post {
   private int id;
   private String title;
   private String message;
   private int ownerUserId;
   private String displayName;
   private String profilePictureUrl;
   private String creationDate;

   …
}
```

Fragment kôda 7.1 Klasa Post sa svojim atributima

Klasa *User* opisuje entitet korisnika (fragment kôda 7.2). Za autentifikaciju korisnika koristi se Firebase baza podataka koja identifikuje zapise u bazi pomoću jedinstvenog stringa (engl. unified identifier, *uid*). Zbog toga klasa *User* sadrži atribut *uid,* koga nasleđuje iz klase *BaseUID* (fragment kôda 7.3). Atribut *id* odnosi se na identifikator u bazi podataka MySQL.

```
public class User extends BaseUID {
   private int id;
   private String token;
   private String email;
   private String displayName;
   private String aboutMe;
   private String profileImageUrl;
   private List<Integer> followingIds;
   private int numberOfFollowers;
   private int numberOfFollowing;

   …
}
```

Fragment kôda 7.2 Klasa User sa svojim atributima

```
public class BaseUID {
   protected String uid = UUID.randomUUID().toString();

   public String getUid() {
      return uid;
   }

   public void setUid(String uid) {
      this.uid = uid;
```

```
    }
}
```

Fragment kôda 7.3 Klasa BaseUID, natklasa klase User

Klasa *Settings* (fragment kôda 7.4) sadrži podešavanja vezana za rad aplikacije u više režima. Sadrži atribute da li aplikacija treba da koristi Redis i kolika je paginacija. Parametri se mogu podešavati na strani za administraciju, a postoje i podrazumevane vrednosti – da aplikacija koristi Redis i da je paginacija 25. Opseg važenja ovih parametara je cela aplikacija.

```
@ApplicationScope
public class Settings {
    private boolean useRedis = true;
    private int pageSize = 25;

    public boolean useRedis() {
        return useRedis;
    }

    public void setUseRedis(boolean useRedis) {
        this.useRedis = useRedis;
    }

    public int getPageSize() {
        return pageSize;
    }

    public void setPageSize(int pageSize) {
        this.pageSize = pageSize;
    }
}
```

Fragment kôda 7.4 Klasa Settings koja sadrži podešavanja za aplikaciju

Na nivou servisa nalaze se klase *UserService* i *PostService.* One se nalaze na sloju između kontrolerskih metoda i sloja pristupa bazi podataka i Redis skladištu.

Najbitnije metode unutar klase *UserService* date su u fragmentu kôda 7.5 i sadrže logiku vezanu za početak praćenja drugog korisnika (engl. follow), i prestanak praćenja drugog korisnika (engl. unfollow).

```
@Service
public class UserService {
    private RedisManager redisManager;
    private Settings settings;

    public static final String KEY_USER_FOLLOWING = "userFollowing";
    public static final String KEY_USER_FOLLOWERS = "userFollowers";

    // praćenje korisnika B sa identifikatorom userIdToFollow, od strane korisnika A sa identifikatorom
userId
    public void follow(int userId, int userIdToFollow) {
        // dodavanje u bazu podataka
```

```
        UserDAO.getInstance().follow(userId, userIdToFollow);

        // dodavanje u Redis
        try (Jedis jedis = JedisFactory.getResource()) {
            // 1. dodavanje atributa ID korisnika B, koga počinje da prati korisnik A, u following skup
korisnika A
            redisManager.addIdsToRedisSet(jedis, KEY_USER_FOLLOWING, userId, new
ArrayList<>(userIdToFollow));

            // 2. dodavanje atributa ID korisnika A, koji počinje da prati korisnik B, u followers skup
korisnika B
            redisManager.addIdsToRedisSet(jedis, KEY_USER_FOLLOWERS, userIdToFollow, new
ArrayList<>(userId));

            // 3. invalidiranje userFeed liste korisnika A
            ArrayList<Integer> followingIds = UserDAO.getInstance().getFollowingIds(userId);
            List<Integer> postIds = PostDAO.getInstance().getFeedPosts(followingIds, 0, -1)
                    .stream().map(Post::getId).collect(Collectors.toList());
            redisManager.deleteKeyFromRedis(jedis, KEY_USER_FEED, userId);
            redisManager.addIdsToRedisList(jedis, KEY_USER_FEED, userId, postIds);
        }
    }

    // prestanak praćenja korisnika B sa identifikatorom userIdToUnfollow, od strane korisnika A sa
identifikatorom userId
    public void unfollow(int userId, int userIdToUnfollow) {
        // uklanjanje iz baze podataka
        UserDAO.getInstance().unfollow(userId, userIdToUnfollow);

        // uklanjanje iz Redisa
        try (Jedis jedis = JedisFactory.getResource()) {
            //  1. uklanjanje atributa ID korisnika B, koga počinje da prati korisnik A, iz  following skupa
korisnika A
            redisManager.removeIdsFromRedisSet(jedis, KEY_USER_FOLLOWING, userId, new
ArrayList<>(userIdToUnfollow));

            // 2. uklanjanje atributa ID korisnika A, koji počinje da prati korisnik B, iz followers skupa
korisnika B
            redisManager.removeIdsFromRedisSet(jedis, KEY_USER_FOLLOWERS, userIdToUnfollow, new
ArrayList<>(userId));

            // 3. pribavljanje atributa ID svih poruka korisnika B
            List<Integer> postIdsToRemoveFromRedis = postService.getPostsByUserId(userIdToUnfollow)
                    .stream()
                    .map(Post::getId)
                    .collect(Collectors.toList());

            // 4. uklanjanje atributa ID svih poruka korisnika B iz userFeed liste korisnika A
            redisManager.removeIdsFromRedisList(jedis, KEY_USER_FEED, userId,
postIdsToRemoveFromRedis);
        }
    }
```

```
}
```

Fragment kôda 7.5 Isešak klase UserService i metode za praćenje i prestanak praćenja drugog korisnika

Najbitnije metode unutar klase *PostService* sadrže logiku vezanu za prikaz vremenske linije u aplikaciji (fragment kôda 7.6), kao i za dodavanje i brisanje poruke (fragment kôda 7.7).

```
@Service
public class PostService {
   private RedisManager redisManager;
   private UserService userService;
   private Settings settings;

   public static final String KEY_USER_FEED = "userFeed";

   public List<Post> getFeedByUserIdWithPagination(int userId, int page, int pageSize) {
      try (Jedis jedis = JedisFactory.getResource()) {
         List<Post> result = new ArrayList<>();

          // proveranje da li se u Redisu nalazi lista sa ključem userFeed
         boolean inRedis = redisManager.checkForKeyExistence(jedis, userId, KEY_USER_FEED);

         // ukoliko ključ postoji, i paramater za korišćenje Redisa je uključen, podaci se dopremaju iz
Redisa
         if (settings.useRedis() && inRedis) {
            List<Integer> postIds = redisManager.getUserFeedFromRedisWithPagination(jedis, userId,
page, pageSize);

            if (postIds.size() > 0) {
               result.addAll(PostDAO.getInstance().getPostsByIds(postIds));
            }

         } else {
            // ukoliko ne postoji ključ u Redisu, poruke za vremensku liniju se pribavljaju iz baze podataka
            List<Integer> followingIds = userService.getFollowingIds(userId);

            result.addAll(PostDAO.getInstance().getFeedPosts(followingIds, page, pageSize));

            // punjenje Redisa atributom ID poruka koje čine vremensku liniju korisnika, s obzirom na to
da Redis nije sadržao listu userFeed za korisnika
            if (result.size() > 0) {
               ArrayList<Integer> postIds = new ArrayList<>();
               result.forEach(r -> postIds.add(r.getId()));

               redisManager.deleteKeyFromRedis(jedis, KEY_USER_FEED, userId);
               redisManager.addIdsToRedisList(jedis, KEY_USER_FEED, userId, postIds);
            }
         }

         return result;
      }
```

```
    }
}
```

Fragment kôda 7.6 Isešak klase PostService i metoda koja doprema poruke koje čine vremensku liniju korisnika

```
// dodavanje nove poruke
public void addPost(Post post) {
    // dodavanje u bazu podataka
    int id = PostDAO.getInstance().createPost(post);
    post.setId(id);

    // dodavanje u Redis
    try (Jedis jedis = JedisFactory.getResource()) {
        // 1. dohvatanje svih korisnika koji su pratioci
        List<Integer> followersIds = redisManager.getIdsFromRedisSet(jedis, KEY_USER_FOLLOWERS,
post.getOwnerUserId());

        // 2. dodavanje identifikatora nove poruke na ključeve koji predstavljaju korisnike koji ga prate
        for (Integer followerId : followersIds) {
            redisManager.addIdsToRedisList(jedis, KEY_USER_FEED, followerId,
Collections.singletonList(post.getId()));
        }

        // 3. dodavanje identifikatora nove poruke na ključ koji predstavlja svoju vremensku liniju
        redisManager.addIdsToRedisList(jedis, KEY_USER_FEED, post.getOwnerUserId(),
Collections.singletonList(post.getId()));
    }
}

// uklanjanje poruke
public void removePost(Post post) {
    // uklanjanje iz baze podataka
    PostDAO.getInstance().deletePost(post.getId());

    // uklanjanje iz Redisa
    try (Jedis jedis = JedisFactory.getResource()) {
        // 1. pribavljanje svih korisnika koji su pratioci
        List<Integer> followersIds = redisManager.getIdsFromRedisSet(jedis, KEY_USER_FOLLOWERS,
post.getOwnerUserId());

        // 2. uklanjanje atributa ID nove poruke sa ključeva koji predstavljaju korisnike koji ga prate
        for (Integer followerId : followersIds) {
            redisManager.removeIdsFromRedisList(jedis, KEY_USER_FEED, followerId,
Collections.singletonList(post.getId()));
        }

        // 3. uklanjanje atributa ID nove poruke sa ključa koji predstavlja svoju vremensku liniju
        redisManager.removeIdsFromRedisList(jedis, KEY_USER_FEED, post.getOwnerUserId(),
Collections.singletonList(post.getId()));
    }
}
```

```
// dobavljanje svih poruka korisnika iz baze podataka
public List<Post> getPostsByUserId(Integer userId) {
   return PostDAO.getInstance().getPostsByOwnerUserId(userId);
}
```

Fragment kôda 7.7 Metode unutar PostService klase koje dodaju/brišu poruku i metoda koja dobavlja sve poruke korisnika iz baze podataka

Klase *PostDAO* i *UserDAO* sadrže metode pristupa bazi podataka MySQL i njih nećemo prikazivati. Klasa *RedisManager* sadrži metode pristupa Redis skladištu i enkapsulira operacije dodavanja, brisanja i dopremanja podataka iz Redis skladišta na najnižem nivou (fragment kôda 7.8).

```
public class RedisManager {
   // provera da li postoji ključ u Redisu
   public boolean checkForKeyExistence(Jedis jedisClient, Integer userId, String prefix) {
      return jedisClient.exists(generateRedisKey(prefix, userId));
   }

   // dopremanje vremenske linije sa paginacijom
   public List<Integer> getUserFeedFromRedisWithPagination(Jedis jedisClient, int userId, int page, int
pageSize) {
      final String redisKeyForFeed = generateRedisKey(KEY_USER_FEED, userId);

      // bez paginacije – pribavlja sve poruke iz Redisa
      if (pageSize == -1) {
         return getUserFeedFromRedis(jedisClient, userId);
      }

      return jedisClient.lrange(redisKeyForFeed, page * pageSize, page * pageSize + pageSize - 1)
            .stream()
            .map(Integer::valueOf)
            .collect(Collectors.toList());
   }

   // dopremanje vremenske linije bez paginacije
   private List<Integer> getUserFeedFromRedis(Jedis jedisClient, int userId) {
      final String redisKeyForFeed = generateRedisKey(KEY_USER_FEED, userId);

      return jedisClient.lrange(redisKeyForFeed, 0, -1)
            .stream()
            .map(Integer::valueOf)
            .collect(Collectors.toList());
   }

   // dopremanje vrednosti iz skupa sa zadatog ključa
   public List<Integer> getIdsFromRedisSet(Jedis jedisClient, String prefix, int userId) {
      return jedisClient.smembers(generateRedisKey(prefix,
userId)).stream().map(Integer::valueOf).collect(Collectors.toList());
   }
```

```
    // dodavanje vrednosti u Redis listu
    public void addIdsToRedisList(Jedis jedisClient, String prefix, int userId, List<Integer> ids) {
        final String redisKey = generateRedisKey(prefix, userId);

        String[] arrayOfIds = ids.stream()
                .map(String::valueOf)
                .collect(Collectors.toList())
                .toArray(new String[ids.size()]);

        jedisClient.lpush(redisKey, arrayOfIds);
    }

    // uklanjanje vrednosti iz Redis liste
    public void removeIdsFromRedisList(Jedis jedisClient, String prefix, int userId, List<Integer> ids) {
        final String redisKey = generateRedisKey(prefix, userId);

        ids.forEach(id -> jedisClient.lrem(redisKey, 1, String.valueOf(id)));
    }

    // dodavanje vrednosti u Redis skup
    public void addIdsToRedisSet(Jedis jedisClient, String prefix, int userId, List<Integer> ids) {
        final String redisKey = generateRedisKey(prefix, userId);

        Pipeline pipeline = jedisClient.pipelined();
        ids.forEach(id -> pipeline.sadd(redisKey, String.valueOf(id)));

        pipeline.sync();
    }

    // uklanjanje vrednosti iz Redis skupa
    public void removeIdsFromRedisSet(Jedis jedisClient, String prefix, int userId, List<Integer> ids) {
        final String redisKey = generateRedisKey(prefix, userId);

        Pipeline pipeline = jedisClient.pipelined();
        ids.forEach(id -> pipeline.srem(redisKey, String.valueOf(id)));

        pipeline.sync();
    }

    // brisanje ključa iz Redisa
    public void deleteKeyFromRedis(Jedis jedisClient, String prefix, int userId) {
        final String key = generateRedisKey(prefix, userId);

        jedisClient.del(key);
    }

    // generisanje ključa pomoću prefiska i identifikatora
    private String generateRedisKey(String prefix, Integer id) {
        return prefix + ":" + id;
    }
}
```

Fragment kôda 7.8 RedisManager klasa i njene metode