

Univerzitet u Nišu

Elektronski fakultet

Seminarski rad

Sistemi za upravljanje bazama podataka

Tema:

**Distribuirane strategije keširanja korišćenjem Redis baze**

Mentor: Student:

Aleksandar Stanimirović Milan Lazarević 906

Niš, 2020

Sadržaj

[1. Uvod 3](#_Toc44025688)

[2. Izazovi prilikom projektovanja baze 4](#_Toc44025689)

[3. Tipovi keširanja baza podataka 5](#_Toc44025690)

[4. Obrasci za keširanje 7](#_Toc44025691)

[4.1 Cache-Aside (Lazy Loading) 8](#_Toc44025692)

[4.2 Write-Through 9](#_Toc44025693)

[5. Validnost keš memorije 11](#_Toc44025694)

[6. Amazon ElastiCache and Self-Managed Redis 12](#_Toc44025695)

[7. Tehnike keširanja relacionih baza 13](#_Toc44025696)

[7.1 Keširanje rezultata SQL upita 15](#_Toc44025697)

[7.2 Keširanje serializovanih objekata 18](#_Toc44025698)

[8. Zaključak 20](#_Toc44025699)

# Uvod

In-memory keširanje podataka predstavlja jedan od najefikasnijih strategija za ubrzanje vase aplikacije kao I redukuje broj upita prema glavnoj bazi I smanjuje cenu upita prema bazi.

Keširanje ce može primeniti nad bilo kojom bazom bazom, bez obzira da li se radi o relacionoj bazi podataka ili se radi o NoSQL bazi.

Najbolji deo keširanja je taj što ga je lako implementirati I drastično poboljšava brzinu I skalabilnost vase aplikacije.

U ovom seminarskom radu biće opisane neke od strategija keširanja I primene pristupa koji se bave ograničenjima I izazovima povezanim sa bazama podataka na disku.

# Izazovi prilikom projektovanja baze

Kada kreirate distribuirane aplikacije za koje je potrebna mala latencija i skalabilnost, baze podataka zasnovane na disku mogu predstavljati brojne izazove. Neke uobičajene uključuju sledeće:

* **Spora obrada upita**: Postoji veliki broj tehnika za optimizaciju upita i dizajna šema koji pomažu da se poboljšaju performanse upita. Međutim, brzina preuzimanja podataka s diska plus dodatna vremena obrade upita obično u najboljem slučaju stavljaju vaše odgovore na pitanja u dvocifrenim milisekundnim brzinama. Ovo pretpostavlja da vaša baza nema stalno opterećenje i da baza podataka radi optimalno.
* **Troškovi skaliranja**: bilo da se podaci distribuiraju na diskovnoj bazi podataka NoSQL ili vertikalno postavljenoj relacionoj bazi podataka, skaliranje za izuzetno visok broj upita čitanja, može biti skupo. Takođe može zahtevati nekoliko replika za čitanje baze podataka.
* **Potreba za pojednostavljivanjem podataka**: Iako relacione baze podataka pružaju odlično sredstvo za redukciju ponavljanja podataka, one nisu optimalne za pristup podacima. Postoje slučajevi u kojima vaše aplikacije možda žele da pristupe podacima u određenoj strukturi ili pogledu da bi pojednostavile pretraživanje i povećale performanse aplikacije.

Pre primene keširanja baza podataka, mnogi arhitekti i inženjeri troše veliki napor pokušavajući da izvuku što više performansi iz svojih baza podataka. Međutim, postoji ograničenje u performansama koje možete postići pomoću baze podataka skladištenoj na disku, a pokušaj rešavanja problema s pogrešnim alatima kontraproduktivn je. Na primer, veliki deo latencije upita vaše baze podataka diktiran je fizikom preuzimanja podataka s diska.

# Tipovi keširanja baza podataka

Keš baze podataka nadopunjuje vašu primarnu bazu podataka uklanjanjem nepotrebnog pritiska na nju, obično u obliku često pristupanih pročitanih podataka. Sam keš može da živi u nekoliko oblasti, uključujući vašu bazu podataka, u aplikaciji ili kao samostalan sloj.

Slede tri najčešće vrste keširanja baza podataka:

* **Database-integrated keš**: Neke baze podataka, poput **Amazon Aurora**, nude integrisanu keš memoriju kojom se upravlja u okviru samog endžina baze i ima ugrađene mogućnosti pisanja. Baza podataka automatski ažurira keš memoriju kada se osnovni podaci promene. Ništa u sloju aplikacije nije potrebno za upotrebu ove keš memorije. Slaba strana integrisanih keševa je njihova veličina i mogućnosti. Integrisane keš memorije su obično ograničeni na dostupnu memoriju koju baze podataka dodeljuje kešu i ne može se koristiti u druge svrhe, kao što je deljenje podataka sa drugim instancama.
* **Lokalna keš memorija**: Lokalna keš memorija čuva najčešće korišćene podatke u vašoj aplikaciji. Ovo omogućava brže pribavljanje podataka u odnosu na ostale arhitekture keširanja, jer uklanja mrežni saobraćaj koji je povezan sa preuzimanjem podataka. Glavni nedostatak je što među vašim aplikacijama svaki instanca ima svoju rezidualnu keš memoriju koja radi na nepovezan način. Informacije koje su prikupljene u pojedinačnom kešu instance aplikacije, bilo da je reč o podacima iz baze podataka, veb sadržaju ili podacima sesije, ne mogu se deliti s drugim lokalnim keš memorijama. Ovo stvara izazove u distributivnom okruženju gde je razmena informacija kritična za podršku skalabilnih dinamičkih okruženja. Budući da većina aplikacija koristi više aplikacionih servera, upravljanje vrednostima preko njih postaje glavni izazov ako svaka instance ima svoju keš memoriju. Pored toga, kada dođe do prekida rada, podaci u lokalnoj keš memoriji se gube. Većina ovih nedostataka nadomeštena je korišćenjem udaljene keš memorije.
* **Udaljena keš memorija**: Udaljeni keš (ili „bočna keš memorija“) je zasebna instanca (ili instance) namenjena za čuvanje keširanih podataka u memoriji. Udaljeni keš se čuva na namenskim serverima i obično se zasniva na key/value NoSQL bazama podataka, poput **Redis** i **Memcached**. Omogućuju stotine hiljada i do milion zahteva u sekundi po instance keš servera. Mnoga rešenja, poput **Amazon ElastiCache** za Redis, takođe obezbeđuju visoku dostupnost potrebnu za kritično radno opterećenje.

Prosečna kašnjenja zahteva za udaljenom keš memorijom je u milisekundama, što je u većini slučajeva brže od zahtev prema bazi podataka na disku. Pri ovim brzinama lokalni keš je retko kad potreban. Udaljene keš memorije idealne su za distribuirana okruženja jer funkcionišu kao povezani klaster koji mogu svi vaši različiti sistemi da koriste. Međutim, kada je latencija mreže velika, možete primieniti dvoslojnu strategiju keširanja koja koristi lokalni I udaljeni keš zajedno. Ova strategija obično se koristi samo kad je neophodno zbog složenost koju dodaje. Sa udaljenom keš memorijom, orkestracija između keširanja podataka I upravljanjem validnošću podataka upravljaju vaše aplikacije i / ili procesi koji ga koriste. Sama keš memorija nije direktno povezana sa baze podataka, ali se koristi pored nje.

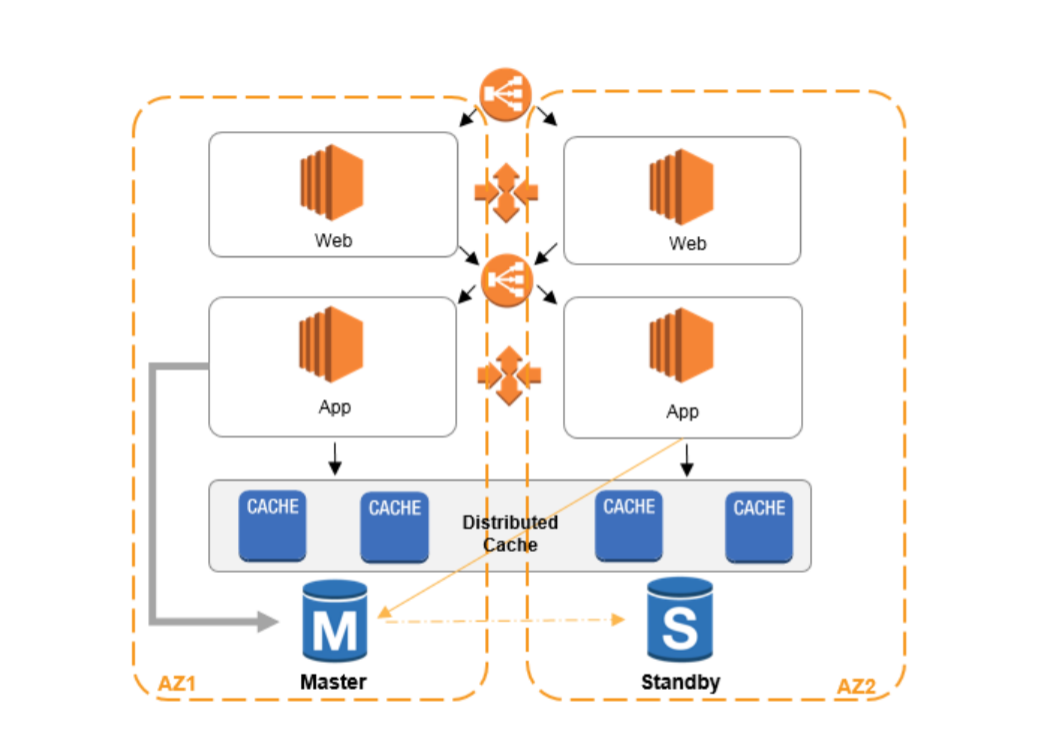
# Obrasci za keširanje

Kada keširate podatke iz svoje baze podataka, postoje obrasci za keširanje za **Redis** i **Memcached**

oje možete implementirati, uključujući proaktivni i reaktivni pristupi. Obrasc koji odlučite da primenite treba da budu direktno u vezi sa vašim ciljevima keširanja i aplikacije.

Dva uobičajena pristupa su cache-aside ili lazy loading (reaktivni pristup) i write-though (proaktivni pristup). Cache-aside keš se ažurira nakon pribavljanja traženih podataka. Write-though keš se ažurira odmah kada se ažurira i primarna baza podataka. Sa oba pristupa aplikacija upravlja kakvim se podaci upisuju u keš i koliko dugo traju u keš memoriji.

Sledeći dijagram je tipičan prikaz arhitekture koja koristi za rad sa distribuiranom keš memorijom.

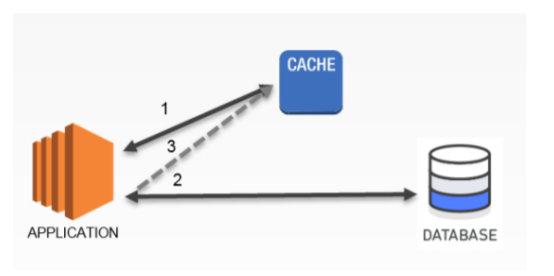


*Slika 1. Arhitektura koja koristi distribuiranu keš memoriju*

## Cache-Aside (Lazy Loading)

Cache-aside je najčešća dostupna strategija keširanja. Osnovna logika pretraživanja podataka može se prikazati na sledeći način:

1. Kada aplikacija mora da pročita podatke iz baze podataka, ona prvo proverava keš memoriju da bi utvrdila da li su podaci dostupni.
2. Ako su podaci dostupni (pogodak iz keširanja), keširani podaci se vraćaju, a odgovor se predaje pozivaocu.
3. Ako podaci nisu dostupni (promašaj u kešu), kontaktira se baza podataka za rezultat. Keš se zatim puni podacima koji su preuzeti iz baze podataka, a podaci se vraćaju pozivaocu.



*Slika 2. cache-aside strategija keširanja*

Ovaj pristup ima nekoliko prednosti:

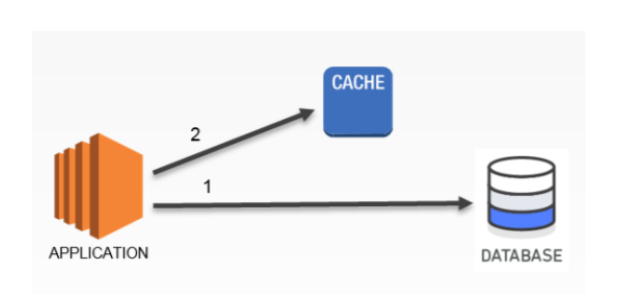
* Keš memorija sadrži samo podatke koje aplikacija zapravo zahtijeva, što pomaže u održavanju efikasnosti veličine keša.
* Implementacija ovog pristupa je jednostavna i donosi trenutne rezultate u performansama, bilo da koristite framework koji implementira lazy loading ili svoju vlastitu prilagođenu logiku aplikacije.

Nedostatak kada se koristi cache-aside kao jedini obrazac keširanja je taj što se podaci učitavaju u keš samo nakon promašaja, u tom slučaju imamo dodatno vreme izvršenja gde imamo proveru keša, promašaj, pribavljanje rezultata iz baze I ponovno snimanje u keš memoriju.

## Write-Through

Write-though strategija keširanja vrši popunjavanje keš memorije u obrnutom redosledu. Umesto lazy-loading podataka u keš memoriju nakon promašavanja keš memorije, keš je proaktivno ažurira odmah nakon primarnog ažuriranja baze podataka. Ta osnovna logika pretraživanja podataka može se prikazati na sledeći način:

1. Aplikacija ili pozadinski proces, ažurira primarnu bazu podataka.
2. Odmah nakon toga podaci se takođe ažuriraju u kešu.

*Slika 3. write-through strategija keširanja*

Write-though obrazac se gotovo uvek implementira zajedno sa lazy loading strategijom. Ako aplikacija dobije promašaj od keš memorije, jer podaci ne postoje ili je podatak istekao, lazy loading učitavanje vrši ažuriranje keš memorije.

Write-through strategija ima nekoliko prednosti:

* Pošto je keš ažuriran sa podacima iz primarne baze podataka, mnogo je veća verovatnoća da će se podaci naći u kešu. Ovo zauzvrat, rezultira boljim ukupnim performansama aplikacije i korisničkim iskustvom.
* Učinkovitost vaše baze podataka je optimalna, jer se izvršava manji broj čitanja iz nje.

Nedostatak write-through pristupa za keširanje je taj što se retko traženi podaci takođe upisuju u keš, što rezultira povećanju keš memorije I većem broju promašaja keš memorije.

Pravilna strategija keširanja uključuje efikasnu upotrebu write-through i lazy-loading pristupa i postavljanje odgovarajućeg roka validnosti podataka za njihovo čuvanje unutar keš memoriju.

# Validnost keš memorije

Možemo kontrolisati vreme trajanja podataka u keš memoriji, primenom vremen (TTL time to live) vreme trajanja podatka I ključa. Po isteku podešenog vremena, ključ se briše iz keš memorije, nakon toga u slučaju pristupa ovom ključu potrebno je ponovo kontaktirati glavnu bazu za pripremu podataka.

Dva principa vam mogu pomoći da odredite odgovarajuće TTL-ove za primenu I vrstu keširanja za implementaciju. Prvo, važno je da znamo stopu promena osnovnih podataka. Drugo, važno je da vi procienite rizik vraćanja zastarjelih podataka u aplikaciju umesto ažuriranog podataka.

Na primer, možda bi imalo smisla čuvati statičke ili referentne podatke (to jest, podatke koja se retko ažurira) važi duži vremenski periode sa strategijom write-through do keš memorije će doći ažurirani podaci kada se osnovni podaci ažuriraju.

Uz dinamičke podatke koji se često menjaju, možda biste želeli da primenite niže TTL-ove, u ovom slučaju je potrebno podesiti trajanja keša sa prosečnom brzinom izmene podataka u glavnoj bazi. To smanjuje rizik vraćanja zastarelih podataka, a istovremeno redukuje što veći broj upita prema glavnoj bazi.

Takođe je važno da prepoznate, čak i ako keširate samo podatke minut ili sekundu u odnosu na duže trajanje, odgovarajuća primena TTL-ova na vaše keširane podatke, može rezultovati ogromnim povećanjem performansi i ukupnim boljim korisničkim iskustvom sa vašom aplikacijom.

Još jedna dobra praksa kod primene TTL-ova na vaše keširane podatke je dodavanje nekih

vreme podrhtavanja (jitter) za TTL. To smanjuje mogućnost velikog opterećenja nad bazom podataka, nastaje kada keširani podaci isteknu. Uzmimo za primer scenario keširanje informacija o proizvodu Ako isovremeno isteknu svi podaci o vašem proizvodu a vaša aplikacija je pod velikim opterećenjem, tada vaša glavna baza podataka mora ispunite sve zahteve aplikacije. U zavisnosti od opterećenja, to može takođe da stvori veliki pritisak na vašu bazu podataka, što rezultira lošim performansama. Dodavanjem neznatno podrhtavanje TTL-ova, nasumično generisana vremenska vriednost (npr., TTL = vaša inicijalna vrednost TTL-a u sekundama + jitter) umanjila bi pritisak na glavnu bazu podataka i takođe smanjuju upotrebu procesora na vašem keš mehanizmu kao rezultat brisanje isteklih ključeva.

# Amazon ElastiCache and Self-Managed Redis

**Redis** predstavlja in-memory bazu podataka otvorenog koda, ova baza je postala najpopularnija ključa / vrednosti baza na tržištu. Veliki deo njegove popularnosti je zahvaljujući podršci raznim strukturama podataka kao i ostalim funkcijama, uključujući podršku za Lua skripte i mogućnost slanja poruka Pub / Sub komunikacija. Ostale dodatne prednosti uključuju topologije visoke dostupnosti sa podrškom za čitanje replika i mogućnost prezistentnog skladišta podataka.

**Amazon ElastiCache** nudi potpuno upravljiv servis za Redis. To znači da se upravlja svim administrativnim zadacima povezanim sa upravljanjem Redis klasterom, uključujući nadgledanje, azuriranje, kreiranje rezervnih kopija i automatsko prebacivanje, upravlja Amazonom. To vam omogućava da se fokusirate na posao i svoje podatke umesto na svoje poslovanje.

Ostale prednosti korišćenja Amazon ElastiCache za Redis nad samoupravljiviom keš memorijom uključuju sledeće stvari:

* Poboljšani Redisov engine koji je u potpunosti kompatibilan s otvorenim kodom verzija, ali to takođe pruža dodatnu stabilnost i robusnost.
* Parametri koji se lako mijenjaju, kao što su politike iseljenja, limiti skladišta itd.
* Sposobnost za skaliranje i proširenja skladišta klastera u terabajtima podataka.

# Tehnike keširanja relacionih baza

Mnoge tehnike keširanja opisane u ovom odeljku mogu biti primene na bilo koju vrstu baze podataka. Međutim, ovaj rad fokusira na relacione baze podataka jer su to najčešći slučajevi upotrebe keširanja baze podataka.

Osnovna paradigma kada tražite podatke iz relacione baze podataka uključuje izvršavanje SQL upita i ponavljanje nad vraćenim setom podataka. Postoji nekoliko tehnika koje možete primenite kada želite keširati vraćene podatke. Međutim, najbolje je odabrati metodu koja pojednostavljuje vaš način pristupa podacima ili optimizuje arhitekturne ciljeve koje imate za svoju aplikaciju.

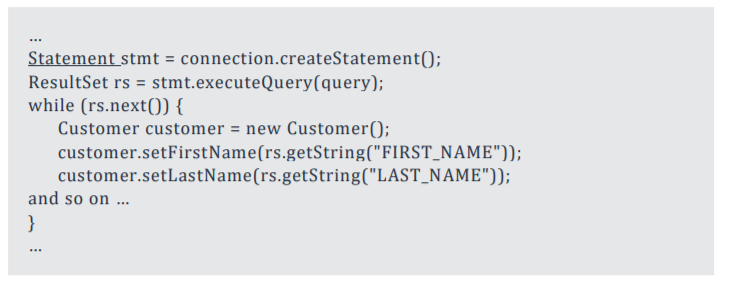
Da bismo to vizualno prikazali, pregledaćemo isečke Java koda da bismo objasnili logiku. Primeri koriste Jedis Redis klijentsku biblioteku za povezivanje sa Redisom, iako je možete koristiti bilo koju Java Redis biblioteku.

Pretpostavimo da ste izvršili sledeći SQL upit nad korisnikovom bazom podataka za CUSTOMER\_ID ​​1001. Istražićemo razne predmemorije strategije koje možete da koristite.

**SELECT FIRST\_NAME, LAST\_NAME, EMAIL, CITY, STATE, ADDRESS,**

**COUNTRY FROM CUSTOMERS WHERE CUSTOMER\_ID = “1001”;**

Ovaj upit vraća sledeći rezultat:



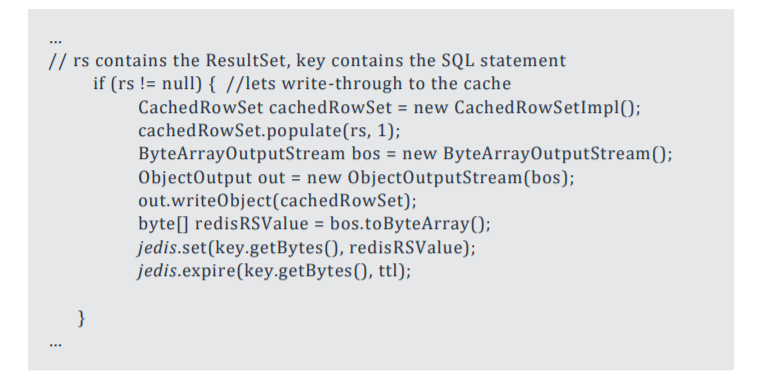
Iteracija nad ResultSet kursorom omogućava nam da preuzmemo polja i vrednosti iz redova baze podataka. Od tog trenutka aplikacija može birati gde, kada I kako koristi te podatke.

Pretpostavimo da se vaš framework aplikacije se ne može koristiti za implementaciju keširanja. S obzirom na ovaj scenario, imate puno mogućnosti. Sledeći primeri prikazuju neke opcije, sa fokusom na logiku keširanja.

## Keširanje rezultata SQL upita

Keširanje objekta ResultSet koji sadrži prikupljene podatek iz baze:

* Prednost: Kada je logika za preuzimanje podataka apstraktna (npr., Kao u pristupu Data Access Object ili DAO sloj), korišćeni kod očekuje samo ResultSet objekte i ne treba ga poznavati u potpunosti da bi se koristio. ResultSet objekti se možu ponavljati, bez obzira da li su nastali iz baze podataka ili su deserijalizovani iz keša, koji uvelike smanjuje logiku integracije. Ovaj obrazac se može primeniti na bilo kojoj relacionoj bazi podataka.
* Mana: Preuzimanje podataka i dalje zahteva vađenje vrednosti iz ResultSet-a kursor objekta i ne pojednostavljuje dalji pristup podacima; samo se smanjuje latencija za preuzimanje podataka.

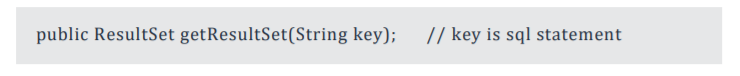
Napomena: Kada keširate red, važno je da je serializabilan. Sledeće primer koristi implementaciju CachedRowSet za ovu svrhu. U slučaju korišćenja Redisa, ovo se čuva kao vrednost bajt polja. Sledeći kod pretvara objekt CachedRowSet u niz bajtova I zatim čuva to bajt polje kao Redis vriednost. Stvarni SQL upit I rezultat se čuva kao ključ i pretvara se u bajtove.

Lepa stvar za čuvanje SQL upita kao ključ je ta što omogućava transparentan sloj apstrakcije keširanja koji skriva detalje implementacije. Druga dodatna prednost je ta što vam nije potrebno da pravite nikakvo dodatno preslikavanje između korisničkog ID-a ključa i izvršenog SQL upita.

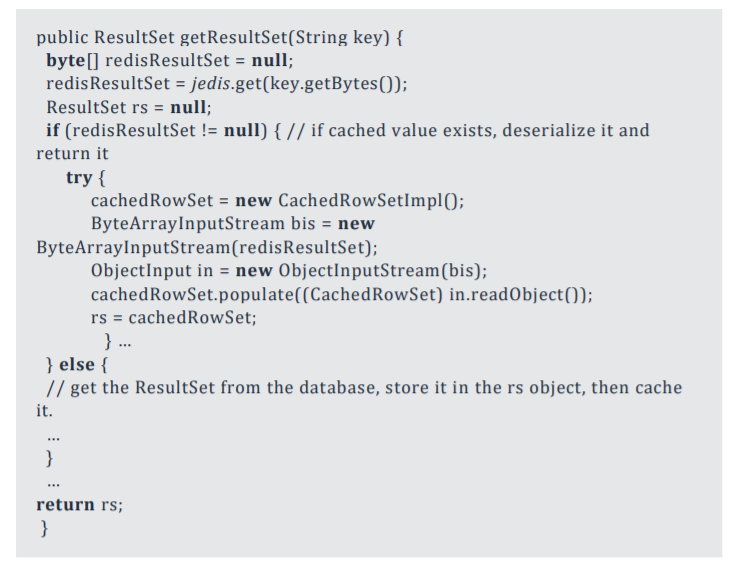
Poslednja linija koda izvršava naredbu za istek da primeni TTL na sačuvani ključ. Ovaj kod prati našu logiku write-though po kojoj kada se upit nad bazom izvrši, vrši se keširanje rezultata odmah nakon toga.

Za lazy loading, prvo biste upitali keš memoriju pre nego što izvršite upit nad bazom podataka. Da biste sakrili detalje implementacije, koristite DAO obrazac i iznesite generičku metodu za vašu aplikaciju za preuzimanje podataka.

Na primer, jer je naš ključ SQL naredba, potpis naše metode bi mogao izgledati ovako:



Kod koji poziva (troši) ovu metodu očekuje samo ResultSet objekat, bez obzira na detalje implementacije interfejsa. Pod haubom, metoda getResultSet izvršava GET naredbu za SQL ključ koji se, ako postoji, deserisališe i pretvara u ResultSet objekta.



Ako podaci ne postoje u kešu, upit se šalje bazi podataka i izvršite keširanje pre povratka.

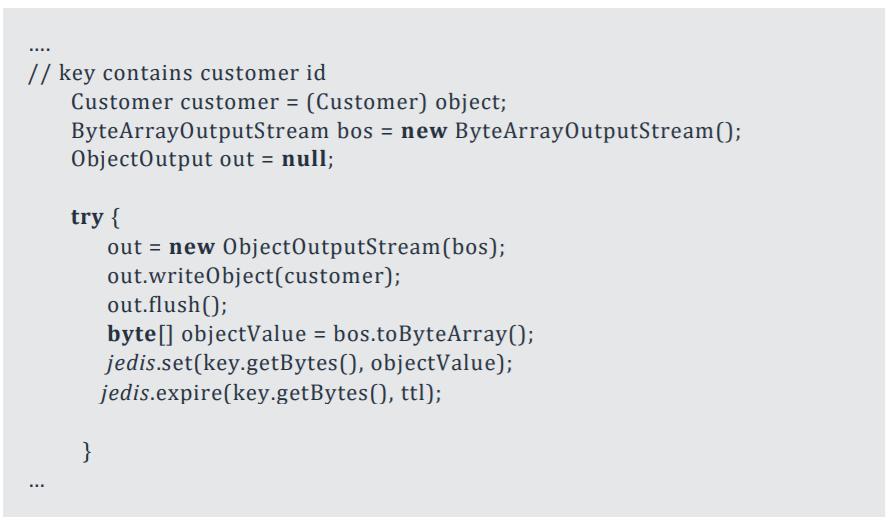
Kao što je ranije spomenuto, najbolja praksa bi bila primena odgovarajućeg TTL-a na i ključeve.

Za sve ostale tehnike keširanja koje ćemo pregledati, trebalo bi da ustanovimo konvenciju imenovanja za naše Redis ključeve. Dobra konvencija o imenovanju je ona koja je lako predvidljiva za aplikacije i programere. Hijerarhijska struktura razdvajanja kolona je uobičajena konvencija o imenovanju ključeva, kao što je primer: **object:type:id**

## Keširanje serializovanih objekata

Keširajte podskupa pribavljenih iz baze podataka u prilagođenoj strukturi koja može biti korišćena unutar aplikacije.

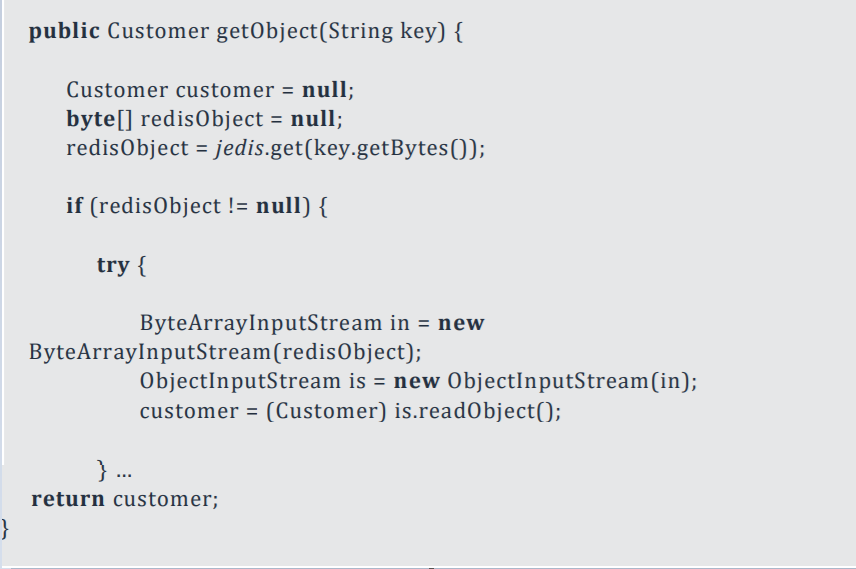
* Prednosti: Koristite objekte aplikacije u originalnom stanju aplikacije jednostavnim tehnikama serializacije i deserializacije. Ovim se možg ubrzati performanse aplikacije minimiziranjem logike transformacije podataka.
* Mane: Slučaj primene u naprednim aplikacija.

Sledeći kod pretvara objekat korisnika u niz bajtova, a zatim skladišti njegovu vrednos u Redisu:

Identifikator ključa se takođe čuva kao bajt reprezentacija i može biti predstavljen kao

customer: id: 1001 format.

Kao što pokazuju drugi primeri, možete stvoriti generičku metodu preko aplikacionog interfejsa koji sakriva detalje metode. U ovom primeru, kada se instancira objekt ili dobija jedan sa stanjem, metoda prihvata korisnički ID (ključ) i vraća objeat sa zadatim ključem iz keša ili kreira jedan nakon upita nad bazom podataka. Prvo, naša aplikacija pita keš memoriju za pribavljanje serializovanog objekta korićenjem njegovog identifikatora kao ljuč. Ako podaci nisu prisutni, izvršava se SQL upit i aplikacija pribavlja podatke entiteta kupc, a zatim lazy-loading učitava serializovane podatke u keš.



# Zaključak

Savremene aplikacije ne mogu posedovati loše performanse. Današnji korisnici imaju malo tolerancije za sporo pokretane aplikacije i loše korisničko iskustvo. Nisko kašnjenje i skaliranje baze podataka presudno je za uspeh vaših aplikacija, neophodno je da koristite keširanje baze podataka.

U ovakvim sitacijama je Redis najbolje rešenje za keširanje i unapređenje performansi aplikacije.