UNIVERZITET U NIŠU
ELEKTRONSKI FAKULTET

OPTIMIZACIJA UPITA KOD MONGODB BAZE PODATAKA

Seminarski rad
Studijski program: Računarstvo i informatika
Modul: Softversko inženjerstvo

Student:

Đorđe Petković, br. ind. 1614

Mentor:

prof. Aleksandar Dimitrijević

Niš, April 2024. godine

`

Sadržaj

`

Uvod

Optimizacija upita u radu sa bazama podataka je ključna za efikasno upravljanje, zato što omogućava bolje korišćenje resursa smanjivanjem opterećenja sistema i obezbeđuje optimalne performanse kod različitih operacija. Odabirom najefikasnijih planova izvršavanja, optimizacija upita smanjuje vreme preuzimanja i obrade podataka, što za rezultat daje brži odgovor za korisnika. Osim poboljšavanja produktivnosti, optimizacijom upita takođe postižemo bolje korisničko iskustvo, zbog bržeg rada same aplikacije ili proizvoda koja komunicira sa bazom podataka.

Osim toga, optimizovani upiti olakšavaju skalabilnost i omogućavaju sistemima baza podataka da upravljaju većim obimom podataka i korisničkim opterećenjem bez smanjena performansi. Skalabilnost, zajedno sa efikasnom upotrebom resursa, doprinosi efikasnosti troškova, što je posebno značajno u okruženjima gde su resursi ograničeni ili se naplaćuju na osnovu korišćenja.

Dobro optimizovani upiti podstiču konzistentnost i pouzdanost unutar sistema tako što smanjuju probleme poput mrtvih tačaka i uskih grla. Takođe, dobro optimizovani upiti jačaju sigurnost podataka tako što smanjuju izloženost osetljivih informacija.

Optimizacija upita igra ključnu ulogu u stvaranju efikasnih, skalabilnih i ekonomičnih sistema, istovremeno poboljšavajući performanse, pouzdanost i sigurnost podataka. U ovom radu biće obrađena tema optimizacije upita kod MongoDB baze podataka. Biće objašnjena teorijska osnova, ciljevi i tehnike za optimizaciju upita, kao i primena na realnom skupu podataka. [1]

MongoDB

MongoDB je program za upravljanje bazama podataka otvorenog koda koji se koristi za NoSQL (ne samo SQL) pristup, kao alternativa tradicionalnim relacionim bazama podataka. NoSQL baze podataka su veoma korisne za rad sa velikim skupovima distribuiranih podataka. MongoDB je alat koji može upravljati informacijama koje se skladište kao dokumenti, čuvati ili pribaviti potrebne informacije [1].

Prednost MongoDB je u skladištenju velikih količina podataka uz brze performanse. Takođe, MongoDB se koristi zbog njegovih ad-hoc upita (upita kojima se prosleđuju određene promenjive, na osnovu kojih se izvršava), indeksiranja, balansiranja opterećenja, agregacije, izvršavanja skripti na serverskoj strani, kao i mnogih drugih mogućnosti.

`

Za razliku od strukturiranog upitnog jezika (SQL-a), koji je standardizovani programski jezik koji se koristi za upravljanje relacionim bazama podataka, i koji normalizuje podatke kao šeme i tabele, MongoDB arhitektura se sastoji od kolekcija i dokumenata [1]. Kolekcije, koje su ekvivalent tabelama u SQL bazama podatka, sastoje se od skupova dokumenata. Dokumenti su sačinjeni od parova ključ-vrednost (key-value pair), koji su osnovna jedinica podataka u MongoDB-ju. MongoDB podržava više programskih jezika (C, C++, C#, Go, Java, Python), a najkorišćeniji na globalnom nivou u radu sa MongoDB je JavaScript.

Kako MongoDB funkcioniše?

Dokumenti sadrže podatke koje korisnik želi da skladišti u MongoDB bazi podataka. Dokumenti se sastoje od parova ključ-vrednost. Dokumenti su slični JavaScript Object Notation (JSON) formatu, ali koriste varijantu nazvanu Binary JSON (BSON). Prednost korišćenja BSON-a je mogućnost prilagođavanju na više tipova podataka. Polja u ovim dokumentima možemo zamisliti kao kolone u relacionim bazama podataka. Vrednosti mogu biti različitih tipova podataka (osnovne vrednosti, drugi dokumenti, nizovi osnovnih vrednosti, nizovi dokumenata...). U dokumentima takođe postoji primarni ključ kao jedinstveni identifikator. Osnovni identifikator je tipa ObjectId, koji se automatski generiše pri kreiranju dokumenta, ukoliko drugačije nije navedeno [2]. Struktura dokumenta se menja dodavanjem ili brisanjem novih ili postojećih polja.

Skupovi dokumenata nazivaju se kolekcijama, koje se mogu zamisliti kao ekvivalent SQL tabelama. Kolekcije mogu sadržati bilo koji tip podataka, ali ograničenje je da podaci u kolekciji ne mogu biti raspoređeni u različitim bazama podataka. Korisnici MongoDB-a mogu kreirati više baza podataka sa više kolekcija. Dokumenti u jednoj kolekciji nisu ograničeni nikakvim uslovima, što znači da dva dokumenta u istoj kolekciji mogu imati skroz različite parametre i vrednosti.

Binarna reprezentacija JSON-sličnih dokumenata obezbeđuje format za čuvanje dokumenata i razmenu podataka. Automatsko deoba (sharding) je još jedna ključna mogućnost koja omogućava da podaci u MongoDB kolekciji budu raspoređeni na više sistema radi horizontalne skalabilnosti, kako količina podataka i broj zahteva rasta [2].

NoSQL DBMS koristi jedan master stukturu (single master architecture) arhitekturu za konzistentnost baza podataka između sekundarnih baza podataka koje održavaju kopije primarne baze podataka. Operacije se automatski repliciraju na sekundarne baze podataka za lako prebacivanje na drugu bazu podataka, u slučaju otkazivanja servera (server failover).

`

Prednosti MongoDB-a

Neke od funkcionalnosti i prednosti MongoDB-a su [3]:

Ahritektura bez šeme	Kao i druge NoSQL baze podataka, MongoDB ne zahteva unapred definisane šeme. Skladišti se bilo koji tip podataka, što omogućava fleksibilnost u kreiranju bilo kog dodatnog polja u dokumentu, što čini skaliranje MongoDB baza podataka lakšim u poređenju sa relacionim bazama podataka.
Orijentisano prema dokumentima (document-oriented)	Jedna od prednosti korišćenja dokumenata je to što se ovi objekti mapiraju na osnovne tipove podataka u više programskih jezika. Takođe, korišćenje dokumenata otklanja potrebu za spajanjem baza podataka (JOIN), što može smanjiti troškove i olakšati odrzavanje velikih skupova podataka.
Skalabilnost	Jedna od najvećih prednosti MongoDB-a je horizontalna skalabilnost, što ga čini idealnom bazom podataka za kompanije koje koriste aplikacije sa velikim brojem podataka. Pored toga, deoba (sharding) omogućava bazi da distribuira podatke preko klastera mašina.
Podrška eksternim sistemima (third-party integration)	MongoDB podržava nekoliko sistema za skladištenje (pr. Redis) i pruža API-jeve za skladištenje koji omogućavaju trećim stranama da razvijaju svoje sisteme za skladištenje (storage engines) za MongoDB.
Agregacija	Postoje ugrađene mogućnosti agregacije, što korisnicima omogućava izvršavanje MapReduce koda direktno u bazi podataka umesto izvršavanja MapReduce na Hadoop-u. MapReduce prvo filtrira i sorira podatke, a nakon toga vrši redukciju nad njima. MongoDB takođe uključuje svoj sopstveni fajl sistem nazvan GridFS, sličan Hadoop Distribuiranom Fajl Sistemu. Upotreba fajl sistema je pretežno za skladištenje fajlova većih od ograničenja veličine BSON-a (16 MB po dokumentu). Ove sličnosti omogućavaju MongoDB-u da se koristi umesto Hadoop-a za kreiranje distribuiranih polja podataka, iako se u nekim slučajevima integriše sa Hadoop-om,

`

	Spark-om i drugim framework-ovima za obradu podataka.
Replikacija	Skup replika su dve ili više MongoDB instance koje se koriste za pružanje visoke dostupnosti podataka. Skupovi replika se sastoje od primarnih i sekundarnih servera [4]. Primarni MongoDB server obavlja sve operacije čitanja i pisanja, dok sekundarna replika čuva kopiju podataka, koja je se ažurira nakon odrađenih operacija. U slučaju kvara primarne replike, koristi se sekundarna replika.
Balansiranje opterećenja	MongoDB upravlja balansiranjem opterećenja bez potrebe za zasebnim, posvećenim balanserom opterećenja, pomoću vertikalnog ili horizontalnog skaliranja. Ovo može znatno smanjiti troškove, zbog visoke cene load balancer-a (pr AWS Elastic Load Balancer). Ipak, za dovoljno veliki skup podataka, MongoDB neće pruziti dobre rezultate u balansiranju opterećenja.

Nedostaci MongoDB-a

Pored svih prednosti korišćenja MongoDB, postoje i neki nedostaci. Neki od nedostaka MongoDB-a su:

Kontinualnost i dostupnost	Sa svojom automatskom strategijom preuzimanja, korisnik postavlja samo jedan glavni čvor u klasteru MongoDB-a. Ako glavni čvor otkaže, drugi čvor će automatski postati novi glavni čvor. Ovaj prelazak obećava kontinuitet, ali nije trenutan - može potrajati do minut. U poređenju sa tim, Cassandra NoSQL baza podataka podržava više glavnih čvorova [4]. Ako jedan glavni čvor otkaže, drugi je spreman, što stvara visoko dostupnu infrastrukturu baze podataka.
Ograničenja upisivanja	Jedan glavni čvor MongoDB-a takođe ograničava brzinu kojom se podaci mogu upisivati u bazu podataka. Zapisi podataka moraju biti zabeleženi na glavnom čvoru, a pisanje novih informacija u bazu podataka ograničeno je kapacitetom tog glavnog čvora.

`

Doslednost podataka	MongoDB ne obezbeđuje potpuni referentni integritet kroz upotrebu ograničenja stranih ključeva, što može uticati na doslednost podataka [4].
Bezbednost	Aautentikacija korisnika nije podrazumevano omogućena u MongoDB bazama podataka. Međutim, zbog velikog broj napada na neosigurane MongoDB sisteme, navelo je developere da dodaju podrazumevana podešavanja i ograničenja, koja blokiraju mrežne veze sa bazama podataka ako ih nije konfigurisao administrator baze podataka.

Osnove optimizacije upita

Optimizacija upita, uopšteno, odnosi se na proces poboljšanja performansi i efikasnosti upita baza podataka. Prilikom interakcije sa bazom podataka, izdaju se upiti kako bi se dobili, manipulisali ili skladištili podatci. Upiti mogu varirati u složenosti, što značajno utiče na vreme njihovog izvršenja. Takođe, sa porastom baze podataka po veličini, ali i složenosti, ključno je optimizovati upite koje se nad njom izvršavaju, radi bržeg odziva i boljeg iskustva korisnika koji komunicira sa bazom podataka.

Definicija i ciljevi optimizacije upita

Optimizacija upita podrazumeva pronalaženje najefikasnijeg načina izvršenja datog upita. Ovaj proces obično uključuje analizu upita i šeme baze podataka, razmatranje različitih planova izvršenja i odabir plana koji minimizira korišćenje sistemskih resursa kao što su CPU, memorija i I/O na disku, dok istovremeno proizvodi tačne rezultate.

Krajnji cilj optimizacije je skraćenje vremena izvršenja upita i poboljšanje efikasnosti sistema. Pri optimizovanju upita, potrebno je da ne zanemarimo tačnost rezulata. Neki upiti mogu biti optimizovani na način da se pokuša da se kreira optimalno rešenje upoređujući nekoliko alternativa zasnovanih na zdravom razumu kako bi se pružilo „dovoljno dobar“ rezultat u efikasnom vremenskom intervalu. Potrebno je obezbediti da to rešenje, odnosno odgovor, ne odstupa mnogo od najboljeg mogućeg rezultata.

`

Pored ovog, neki dodatni ciljevi pri optimizaciji upita su omogućavanje skalabilnosti, smanjenje troškova i bezbednost podataka. Ukoliko nam je cilj da skaliramo naše poslovanje, kao i bazu podataka, pisanjem optimizovanih upita možemo obezbediti minimalnu dodatnu latentnost pri izvršavanju upita. Takođe, ukoliko implementiramo neke od tehnika optimizacije, možemo smanjiti troškove, pogotovo ako se koristi neka third-party baza podataka. Ukoliko zahtevamo neke resurse, koji se naplaćuju po izvršenom zahtevu ili količini resursa koje zahtevamo, optimizacijom upita koje šaljemo tim bazama podataka možemo znatno smanjiti troškove poslovanja.

Što se tiče sigurnosti, optimizacijom upita smanjujemo vreme procesiranja upita, čime smanjujemo vremenski interval u kome su naši podaci izloženi potencijalnim pretnjama.

Tehnike za optimizaciju upita

U tradicionalnim relacionim bazama podataka, optimizacija upita često uključuje tehnike kao što su selekcija indeksa, optimizacija spajanja i analiza bazirana na troškovima. Na primer, osoba koja je zadužena za optimizaciju upita može odlučiti da koristi ili kreira indeks kako bi se brzo pronašli redovi u tabeli ili izabere određeni algoritam spajanja kako bi efikasno kombinovao podatke iz više tabela.

Neke od opštih i generalnih tehnika za optimizaciju upita u bazama podataka su:

Indeksi	Indeksi pomažu bazi podataka da brzo pronađe potrebne podatke. Identifikujte često korišćene kolone u upitima, spajanjima i sortiranjima. Na osnovu statistike o upitima, mogu se kreirati indeksi koji će ubrzati pronalaženje željenih podatka.
Optimizacija join-ova	Kod SQL baza, potrebno je efikasno koristiti spajanje tabela. Preporučeno je korišćenje INNER JOIN umesto OUTER JOIN ako je moguće i izbeganje nepotrebnih join-ova.
Izbegavanje pribavljanja svih rezultata	Poželjno je izbegavati pribavljanje svih rezultata (kod SQL izbegavati SELECT **, kod MongoDB izbegavati .find()).
Korišćenje alata za profilisanje baza podataka	Mnoge baze podataka pružaju alate za praćenje performansi i profilisanje baze podataka. Korišćenjem ovih alata mogu se identifikovati uska grla, i na osnovu njih vršiti optimizacija upita.

`

Redovno ažuriranje statistika	Statistike baze podataka pomažu ugrađenom optimizatoru upita da donese bolje odluke. Ukoliko je obezbeđeno redovno ažuriranje statistika, posebno nakon značajnih promena podataka, sam optimizator će davati bolje i brže rezultate, tako što će se adaptirati na osnovu statistike koja mu je pružena.
Redovno održavanje	Ukoliko se redovno obavlja održavanje baze podataka, upiti će proizvoditi bolje rezultate. Tehnike za održavanje baze podataka mogu biti reorganizacija indeksa, reorganizacija baze podataka, vakumiranje...
Korišćenje keširanja	Implementacija keširanja na različitim nivoima (applikativnom, nivou baze podataka, ili nivou upita) mogu značajno optimizovati rad baze podataka. Česti upiti, koji mogu biti keširani, neće se ponavljati i dodatno opterećivati bazu podataka. Takođe, korisniku će se pružiti podaci u mnogo kraćem roku.

Optimizacija upita kod MongoDB baze podataka

Za razliku od SQL baza podataka, NoSQL baze podataka nude drugačiji pristup skladištenju i pribavljanjna podataka, često optimizovan za specifične slučajeve upotrebe kao što su skalabilnost, fleksibilnost i visoka dostupnost. Optimizacija upita u NoSQL bazama podataka fokusira se na optimizaciju šablona pristupa umesto kompleksnih SQL upita.

U ovom delu biće predstavljeni neki od ključnih aspekata optimizacije upita u NoSQL bazama podataka. Neki od aspekata i tehnika su generalno primenjivi na sve baze podataka, dok su neki od njih adaptirani i primenjivi samo nad MongoDB bazom podataka.

Ti aspekti i tehnike su [5]:

Indeksiranje	Pravilno indeksiranje može dramatično ubrzati performanse upita. Potrebno je identifikovati polja koja se često koriste u upitima i napravite indekse na tim poljima. Korišćenje metode **explain()** za analizu performansi upita je neophodno kako bi se osiguralo efikasno korišćenje indeksa.

`

Projekcija upita	Potrebno je dobiti samo polja koja su potrebna umesto celokupnog dokumenta. Ovo će smanjiti količinu podataka prenetih preko mreže i poboljšati performanse upita, posebno za velike dokumenta. Takođe, projekcija upita može poboljšati bezbednost sistema, tako što se neki osetljivi podaci neće pribavljati, ukoliko ne postoji potreba za njima. Ukoliko u bazi podataka imamo šemu korisnika, sa njegovom lozinkom, poželjno je ne pribavljati je (bilo ona čuvana u plain obliku, ili kao neka heš vrednost), ukoliko ona nije striktno potrebna (na primer za prijavljivanje korisnika). Kao i sa lozinkom, pomoću projekcija treba obezbediti da se osetljivi podaci ne pribavljaju, ukoliko striktno nisu potrebni. Time se štedi vreme izvršenja naredbi, ali i poboljšava sigurnost celog sitema.
Filtriranje upita	Potrebno je efikasno koristiti filtere upita kako bi se smanjio rezultat što je više moguće. Korišćenje indeksiranih polja u filterima upita je neophodno. Treba izbegavati korišćenje regularnih izraza ili **$where** upita osim ako nije apsolutno neophodno, jer mogu biti resursno zahtevni.
Optimizacija upita pomoću operatora	Potrebno je obezbediti da su upiti efikasno strukturirani. Treba mudro koristiti **$sort**, **$limit** i **$skip** operatore kako bi se smanjila količina podataka koju baza podataka obrađuje. Ukoliko pribavljamo dokumente iz kolekcije koja ima veliki broj dokumenata, poželjno je osmisliti sistem, koji će te dokumente pribavljati u blokovima, pomoću ovih operatora. Smanjenje broja podataka koji se jednovremeno obrađuje, obezbediće manje vreme latencije.
Agregacija upita	Kada je akcenat na složenim upitima, posebno onima koji uključuju višestruke faze obrade ili transformacija podataka, korišćenje agregacionog okvira koji pruža MongoDB može biti izuzetno korisno za optimizaciju upita. Agregacioni okvir omogućava da se izvrše različite operacije nad podacima kroz seriju faza nazvanih "pipelines". Svaka faza u pipeline-u obavlja određenu operaciju na podacima, poput filtriranja, grupisanja, sortiranja ili primene matematičkih izraza. Ove faze se izvršavaju sekvencijalno, pri čemu izlaz jedne faze služi kao ulaz za sledeću fazu. Ukoliko se koristi agregacioni okvir, dobijaju se: • Višestruke Transformacije: Sa agregacionim okvirom, mogu se izvršiti višestruke transformacije nad podacima unutar jednog upita. Umesto izvršavanja odvojenih upita za svaku transformaciju,

`

mogu se definisati sve potrebne operacije u jednom pipelinu, što može biti efikasnije u smislu vremena izvršenja i iskorišćenja resursa.

- Optimizovana obrada: Agregacioni okvir je dizajniran da optimizuje obradu podataka. Koristi interne optimizacije i mogućnosti paralelne obrade da efikasno upravlja velikim skupovima podataka i složenim upitima.

- Optimizacija Pipelina: MongoDB-jev optimizer upita može analizirati i optimizovati agregacione pipeline-ove radi poboljšanja performansi. Može preurediti faze, koristiti indekse i primeniti različite tehnike optimizacije kako bi pojednostavio izvršenje pipeline-a.

- Agregacioni operatori: MongoDB pruža bogat set agregacionih operatora koji omogućavaju da se izvrši širok spektar operacija, uključujući aritmetičke operacije, manipulacije nizovima, pretragu teksta...

Distribuiranje sistema

Ako skup podataka nije moguće smestiti na jedan server, ili u jednu bazu podataka, potrebno je razmotriti deljenje MongoDB klastera. Deljenje distribuira podatke preko više servera, omogućavajući horizontalno skaliranje i poboljšane performanse upita. Takođe, distribuiranje se može vršiti i na nivou kolekcija i dokumenata.
Ukoliko uzmemo primer da su kolekcije ograničene šemama, koje nam pruža biblioteka Mongoose napravljenja za NodeJS, i pri planiranju sistema i baze podataka primetimo potencijalno veliki broj poziva koje će biti upućeni sa ciljem obrade jednog dela dokumenta neke kolekcije, možemo daj deo izdvojiti u novu kolekciju. Naravno, to sa sobom nosi dosta rizika, jer ukoliko se poveća broj zahteva prema originalnoj kolekciji, biće potrebno često koristiti populaciju te kolekcije novonapravljenom. To može imati suprotan efekat, i usporiti bazu podataka i povećati procesorsko vreme.

Pisanje briga

Ovo je suština kompromisa između brzine i trajnosti podataka. Kada se koriste niže postavke za pisanje, kao što su opcije koje omogućavaju brži zapis podataka na disk, to može poboljšati performanse pisanja, ali može dovesti do gubitka trajnosti podataka u slučaju neočekivanih grešaka ili prekida sistema. To znači da podaci možda neće biti potpuno i sigurno sačuvani na disku u svakom trenutku.

Na primer, ako se koristi opcija "write concern" sa nižim nivoom, MongoDB neće čekati potvrdu da su podaci sigurno zapisani na više replika pre nego što odgovori na

`

	zahtev za pisanje, što može ubrzati odgovor servera ali smanjiti sigurnost podataka.
Profajlisanje	Praćenje performansi upita koristeći funkcionalnost profajliranja MongoDB-a je neophodno. Identifikacija sporo izvršavanih upita i analiza njihovih planova izvršenja su neophodni za identifikaciju oblasti za optimizaciju. Ukoliko znamo prilikom obrade upita koje vrste baza podataka radi sporije, može se fokusirati na otklanjanje i optimizaciju tih upita, koji prave problem.
Konfiguracija servera	Potrebno je obezbediti da je MongoDB server pravilno konfigurisan za određeni radni teret. Prilagođavanje postavki kao što su veličina keša, briga za pisanje i izdvajanje dnevne statistike je neophodno kako biste optimizovali performanse u skladu sa specifičnim zahtevima.
Dizajn šeme	Možda i najbitnija stavka prilikom optimizacije upita, dolazi pre pisanja ijednog upita. Potrebno je dobro dizajnirati šemu baze podataka, napravljenu da radi efikasno sa podacima koji su potrebni, kako bi se optimizovale performanse upita. Potrebno je izvršiti denormalizacija podataka gde je to potrebno kako bi se smanjila potreba za skupim spojevima ili pretragama. Potrebno je da osoba koja dizajnira šemu baze podataka dobro razume obrasce po kojima će aplikacija koja će koristiti tu bazu podataka biti građena, kako bi dizajnirali šemu kojom će efikasno moći da se upravlja iz aplikacije za čiju potrebu je napravljena.
Korišćenje $hint za odabir određenog indeksa	U većini slučajeva, optimizer upita bira optimalni indeks za određenu operaciju. Međutim, MongoDB se može prisiliti da koristi određeni indeks koristeći metod **hint().** Korišćenje **hint()** da se podrži testiranje performansi ili na nekim upitima gde se mora odabrati polje ili polja koje su uključena u nekoliko indeksa.
Korišćenje operatora Inkrementa za izvršavanje operacija na serveru	Može se koristiti MongoDB-ov **$inc** operator za inkrementiranje ili dekrementiranje vrednosti u dokumentima. Operator inkrementira vrednost polja na serverskoj strani, umesto da odabere dokument, obavi jednostavne modifikacije na klijentu, a zatim upisše ceo dokumenta nazad na server. **$inc** operator takođe može pomoći u izbegavanju uslova trke, koji bi nastali kada bi dva upita zatražila dokument, ručno inkrementirala polje i istovremeno sačuvala ceo dokument natrag.

`

Primena optimizacije upita nad podacima

U ovom delu rada, primenićemo tehnike optimizacije upita nad realnim podacima, i dati primere koda sa neoptimizovanim i optimizovanim upita. Za potrebe praktičnog dela, implementiran je jedan sistem nad bazom podataka, za potrebe ski centara. U ovom delu, biće prikazane šeme baze podataka (šeme kolekcija u MongoDB), način pripreme i upisivanja podataka, kao i vreme procesiranja upita. Biće upoređeno vreme izvršenja upita pre i posle implementacije opisanih tehnika optimizacije uslova.

Na praktičnom primeru moguće je primeniti samo neke od opisanih tehnika u predhodnim poglavljima. Tehnike za optimizaciju kao što su dizajn šema, konfiguracija servera, distibuiranje servera uzeti su u obzir prilikom kreiranja same šeme baze podataka.

U svrhu primene praktičnog dela seminarskog rada, izrađena je aplikacija u NestJS-u [6], koja koristi MongoDB kao bazu podataka. Za potrebe izvršenja operacija nad bazom podataka, koristi se NPM biblioteka Mongoose [7]. Takođe, implementirano je keširanje podataka, kao i brisanje istih na osnovu nekog vremenskog intervala korišćenjem storage engine-a Redis [8], kao i NPM biblioteke CRON.

Dizajn baze podataka

Za potrebe praktične primere, dizajnirana je šema baze podataka. Delovi koji će ovde biti prikazani, prikazani su sa ciljem objašnjavanja primene tehnika indeksiranja i distribuiranja (deobe) kolekcija.

```
BookingSchema.index({ serviceId: 1, createdAt: 1 });

BookingSchema.index({ createdAt: 1 });

export type Booking = {
  _id?: string;
  userId: string;
  serviceId: string;
  skiCenterId: string;
  value: number;
  numOfGuests: number;
  dateFrom: Date;
  dateTo: Date;
  serviceType: Service;
  isApproved: boolean;
  isCancelled?: boolean;
  cancelledBy?: string;
};
```

Slika 1.0 Booking šema

`

Na primeru Booking šeme, možemo videti jedan od načina za korišćenje indexa. U ovom slučaju, napravljeni su indeksi nad atributima createdAt, kao i kompozitni index nad atributima createdAt i serviceId. Atribut createdAt je automatski generisani Date podatak, koji prikazuje kada je određeni dokument kreiran. Ukoliko je potrebno implementirati sortiranje, poželjno je napraviti indeks nad tim atributom, radi bržeg sortiranja.

Na primeru prvog indeksa, možemo videti kako se u MongoDB-u mogu napraviti kompozitni indeksi. Kompozitni indeksi funkcionišu tako što se dokumenti grupišu i sortiraju na osnovu prvog atributa, a nakon toga na osnovu svakog sledećeg. Uzimajući u obzir da je ovo aplikacija namenjena radu ski centra, sa zakazivanjem hotela, implementiran je indeks koji prvo dokumente grupiše na osnovu serviceId-a, jer kada je potrebno upravljati Booking-zima, osoba koja upravlja njima može samo da modifikuje Booking-e vezane za servis koji on pruža (bilo to hotel, ili pristup ski stazama). Nakon toga, s' obzirom na to da se zahtevi za zakazivanje trebaju obraditi u što kraćem vremenskom roku, dodat je atribut createdAt u rastućem poretku, što znači da se dokumenti grupišu od najranije kreiranog ka najskorije kreiranim. To omogućava brzo pribavljanje dokumenata u tom obliku, kada ih je potrebno obraditi.

```
HotelSchema.index({ price: 1, numOfStars: 1 });
HotelSchema.index({ numOfStars: 1 });
HotelSchema.index({ name: 1 });

export type Hotel = {
  _id?: string;
  name: string;
  address: string;
  price: number;
  numOfGuests: number;
  numOfStars: number;
  availableDays: string[];
  bookings?: [Booking];
  autoAccept: boolean;
  skiCenterId: string;
};
```

Slika 1.1 Hotel šema

Na primeru šeme za hotel, pored implementiranih indeksa za cenu, broj zvedica, kao i ime hotela, možemo da primetimo i distribuciju kolekcija. Svaki hotel ima Booking-e vezane za njega, ali umesto da u šemi hotela dodamo sve atribute koje Booking ima, napravljena je

`

posebna kolekcija, radi optimizacije upita nad njima. Takođe, s' obzirom da svaki hotel ima veliki broj recenzija, napravljena je posebna kolekcija za recenzije, koja je povezana sa hotelom preko atributa hotelId.

```
export type Review = {
  hotelId: string;
  description: string;
  value: number;
};

ReviewSchema.index({ value: 1 });
```

Slika 1.2 Review šema

Upisivanje podataka (seed-ovanje)

Za potrebe upisivanja velike količine podataka u bazu, za potrebe testiranja optimizacije upita, korišćenja je NPM biblioteka Faker [9]. Za potrebe ovog primera, kreirano je 20000 Booking-a, uz sve propratne šeme (User, Ski centar, Hotel, Ski staza..)

```
const TrackBookings: Booking[] = faker.helpers.multiple(
  this.generateBookings.bind(this, USERS, Tracks, Service.TRACK),
  {
    count: 10000,
  },
);

const HotelBookings: Booking[] = faker.helpers.multiple(
  this.generateBookings.bind(this, USERS, Hotels, Service.HOTEL),
  {
    count: 10000,
  },
);
```

`

```
generateBookings(
  users: User[],
  services: Track[] | Hotel[],
  serviceType: Service,
): Booking {
  const { dateFrom, dateTo } = this.generateDates();

  const user = users[Math.floor(Math.random() * users.length)];
  const service = services[Math.floor(Math.random() * services.length)];
  const numOfDays = this.trackService.getNumbOfDays(dateFrom, dateTo);
  const guests = faker.number.int({ min: 1, max: 12 });
  return {
    userId: user._id,
    serviceId: service._id,
    skiCenterId: service.skiCenterId,
    value: numOfDays * guests * service.price,
    numOfGuests: guests,
    dateFrom: dateFrom,
    dateTo: dateTo,
    serviceType: serviceType,
    isApproved: faker.datatype.boolean(),
  };
}
```

Slike 1.3 i 1.4 Seed-ovanje podataka

`

Izvršavanje upita nad bazom podataka

Za potrebe demonstracije optimizacije upita, biće iskorišćena ruta u kontroleru, koja omogućava adminu pristup svim zahtevima (Booking) u ski centru za koji je on zadužen. Zbog prirode skripte za seed-ovanje podataka, i normalnog rasporeda koji ona obezbeđuje, ovaj admin ima mogućnost upravljanja nad jednom polovinom ukupnih zahteva (2 ski centra, po 10000 Booking-a svaki).

```
@Roles(Role.ADMIN)
@UseGuards(JwtAuthGuard, RolesGuard)
@Get()
async getAllBookings(
  @Query('perPage') perPage: number,
  @Query('page') page: number,
  @Query('userId') userId: string,
  @Query('serviceId') serviceId: string,
  @Query('filter') filter: BookingFilter,
  @UserDec() user: User,
): Promise<Pagination<Booking>> {
  return await this.bookingService.getAllBookings(
    perPage,
    page,
    userId,
    serviceId,
    filter,
    user,
  );
}
```

Slika 1.5 Ruta za pribavljanje Booking-a

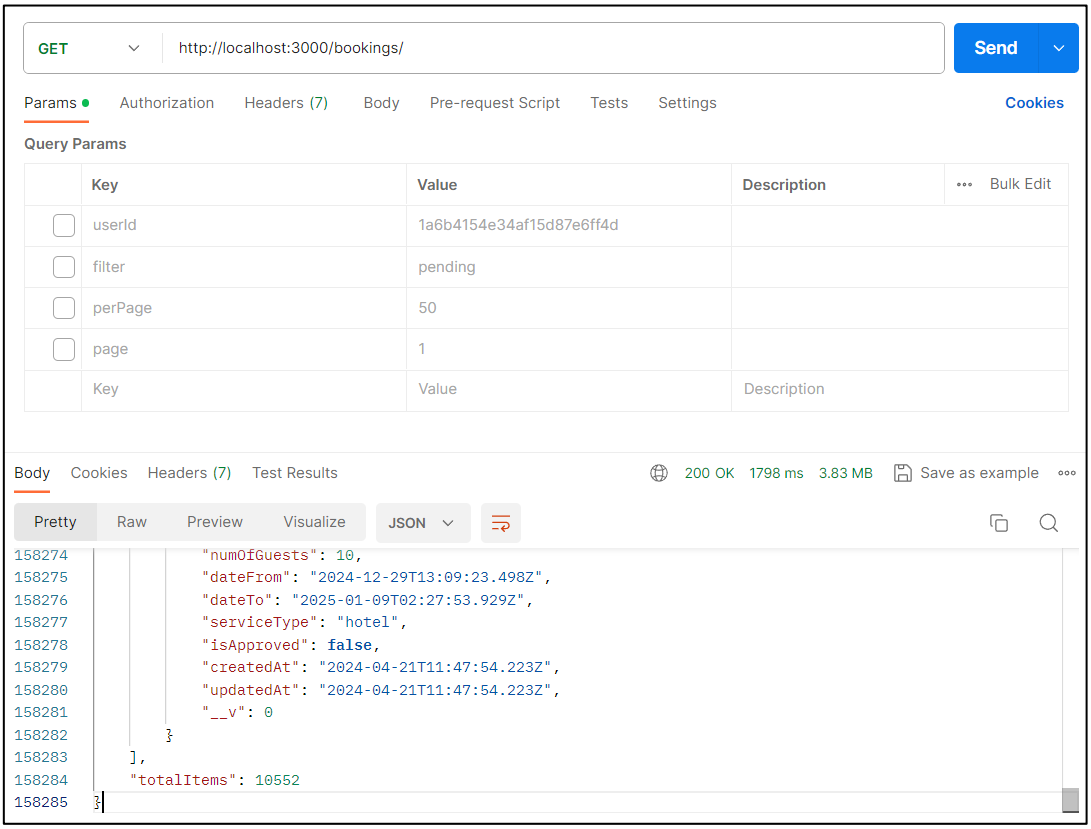
Ova ruta, sve parametre prosleđene kroz query, modifikuje i pretvara u query objekat, koji se prodleđuje **find()** metodi pri komunikaciji sa bazom podataka.

`

```
async findAllBookings(
  query,
  options: { limit: number; skip: number },
  dateSort: 1 | -1,
): Promise<BookingDocument[]> {
  return this.bookingModel.find(query, null, options).sort({
    createdAt: dateSort,
  });
}
```

Slika 1.6 Funkcija za pribavljanje podataka

Na početku, biće demonstriran upit bez primene tehnika optimizacije, odnosno najgori mogući slučaj. Kao što je napomenuto u delu „**Izbegavanje pribavljanja svih rezultata**“, najgori slučaj je ukoliko kroz upit pribavljamo sve dokumente u jednog kolekciji. Upravo taj slučaj biće prikazan na sledećoj slici.



Slika 1.7 Zahtev bez ikakve optimizacije

U ovom slučaju, možemo videti da su vraćeni svi zahtevi za rezervaciju za koje je administrator zadužen. Na dnu slike, vidimo da je vraćeno preko 10000 dokumenata, ukupne veličine od

`

3.83MB. Dobijanje rezultata pomoću ovog zahteva trajao je skoro 1.8 sekundi, što je u realnom slučaju nekog proizvoda, odnosno aplikacije koja se koristi, neprihvatljivo.

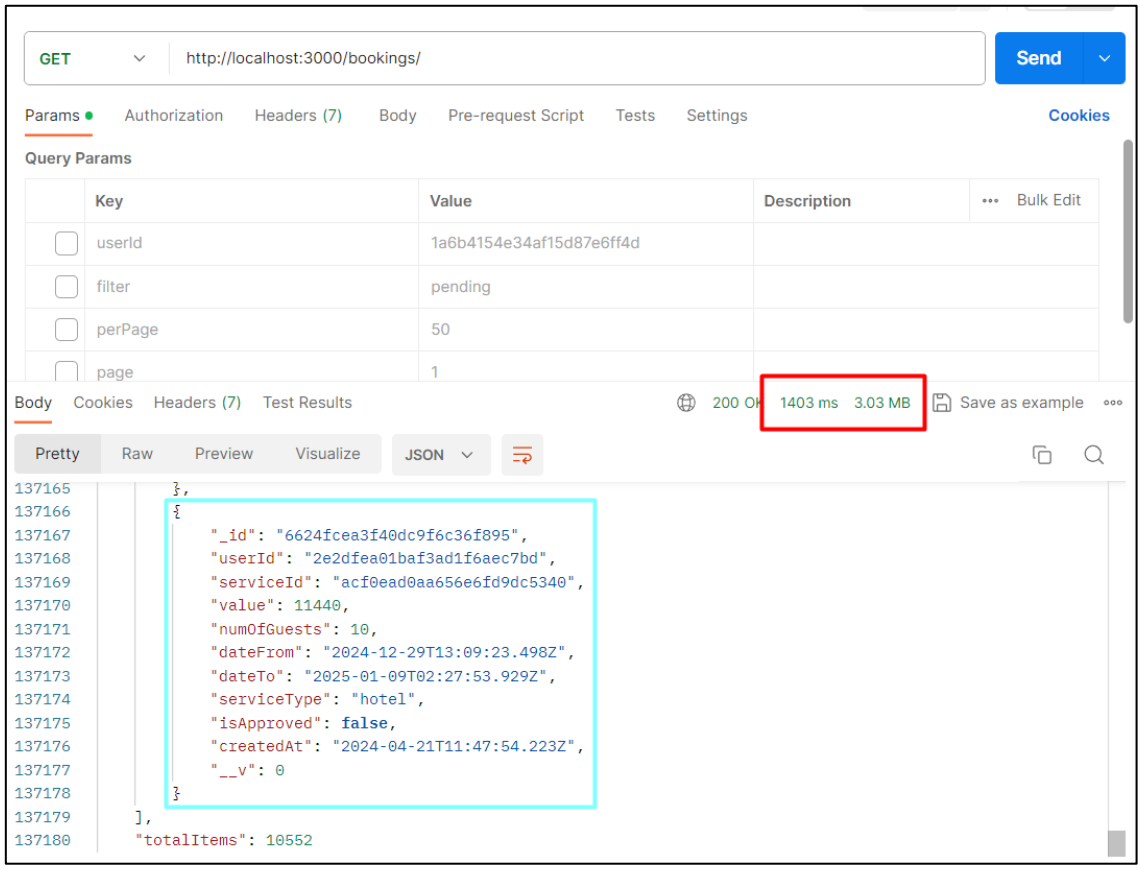
Zbog toga uvodimo tehnike optimizacije upita, koje će limitirati količinu podataka koje dobijamo, i ubrzati procesorsko vreme.

Najpre, iskoristićemo prethodno objašnjenu tehniku optimizacije upita „**Projekcija upita**“. S obzirom na to, da administrator ima pristup samo zahtevima vezanim za njegov skiCenterId, možemo izuzeti taj atribut pri selekciji zahteva. Takođe, u konkretnom slučaju pronalaženja svih zahteva, nije nam potreban atribut updatedAt.

```
async findAllBookings(
  query,
  options: { limit: number; skip: number },
  dateSort: 1 | -1,
): Promise<BookingDocument[]> {
  return this.bookingModel.find(query, {updatedAt: 0, skiCenterId: 0}, options).sort({
    createdAt: dateSort,
  });
}
```

Slika 1.8 Dodavanje projekcija

Ukoliko izuzmemo ta dva atributa pomoću projekcije, dobijamo sledeće rezultate:



Slika 1.9 Zahtev sa projekcijom

`

Na slici se može primetiti, da samo upotrebom projekcije, da izuzmemo dva atributa iz dokumenata, znatno se smanjuje vreme obrade zahteva, kao i količina podataka koja se prenosi kroz mrežu. Primećujemo da je vreme obrade upita brže za 0.4 sekundi, dok je količina podataka sa 3.8 MB smanjena na 3 MB.

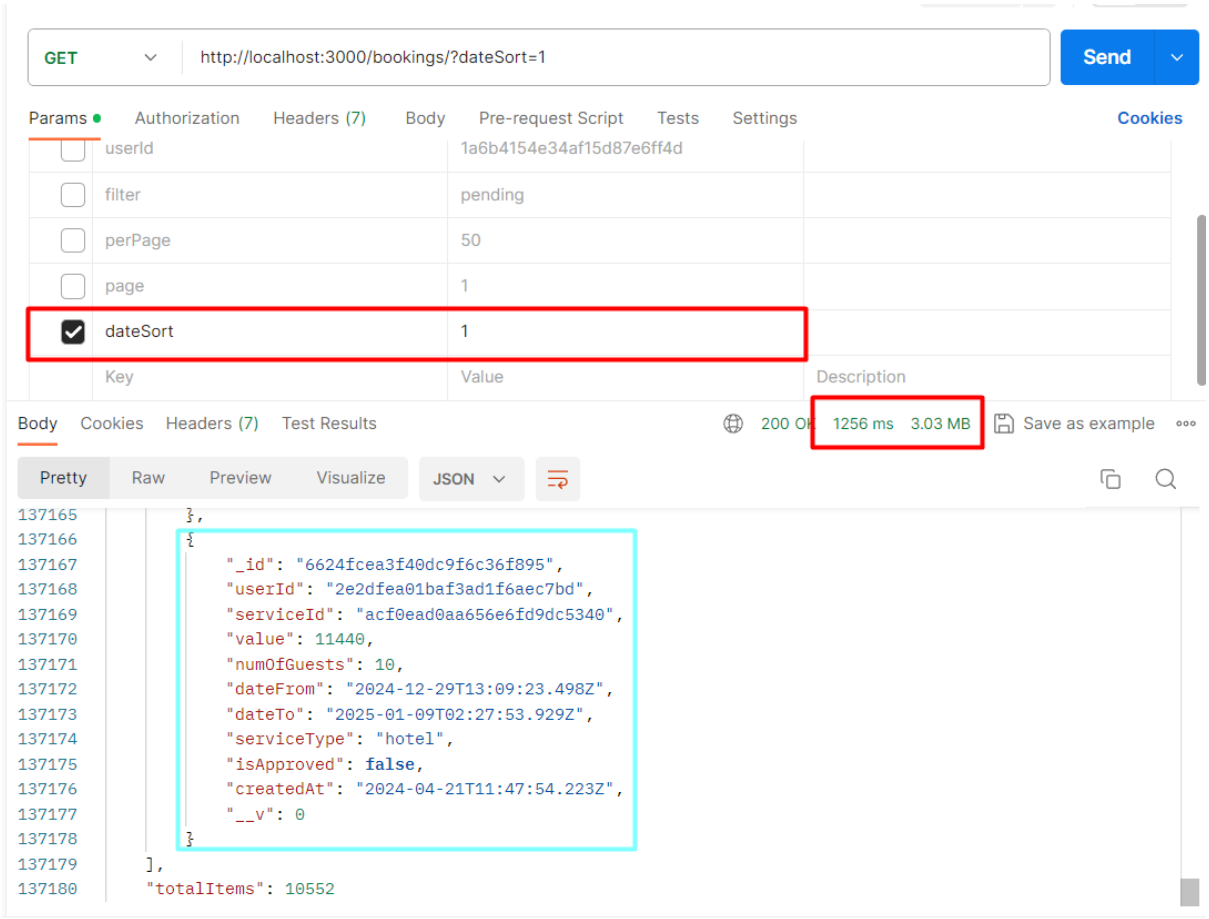
Booking model, u ovom primeru, nema mnogo atributa, koji su redudatni u nekom pozivu, ili koji trebaju ostati tajni (ne deliti se kroz mrežu). Ukoliko bi se ova tehnika primenila pri pronalaženju korisnika, odstranilo bi se više atributa (lozinka, mail, skicenterId...). Na ovom primeru vidimo da projekcija upita može pomoći u optimizaciji upita.

S' obzirom na postojanje indeksa u Booking šemi, koji nam omogućava da dokumente grupisemo na osnovu datuma kreiranja, prosleđivanjem **dateSort** parametra, možemo aktivirati taj indeks, i dobiti bolje rezultate.

```
BookingSchema.index({ createdAt: 1 });
```

Slika 1.10 Indeks za sortiranje

Rezultati sa aktiviranjem ovog indeksa su:



Slika 1.11 Zahtev sa korišćenjem indeksa za sortiranje

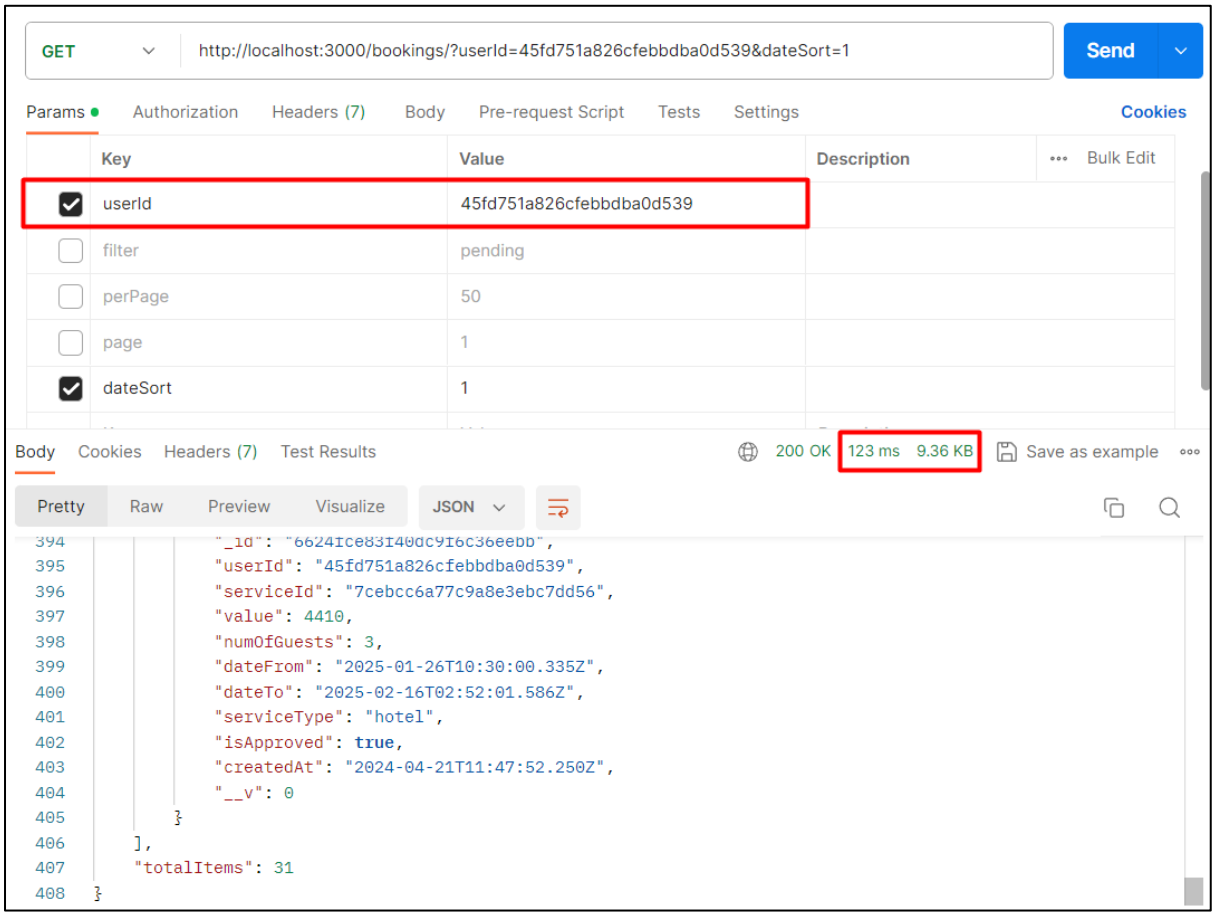
`

Na slici se može videti smanjenje vremena obrade zahteva od 150 milisekundi, dok je veličina podataka ostala ista.

Iako su ove optimizacije smanjile vreme izvšenja upita za čak 550 milisekundi, vreme izvršenja naredbe je i dalje predugo za komercijalnu upotrebu. Ukoliko bi baza sadržavala veći broj zahteva, nakon dužeg korišćenja aplikacije, ovo vreme bi bilo znatno sporije od 1.25 sekundi. U narednom delu, biće prikazane dodatne optimizacije upita, u vidu filtriranja, optimizacije pomoću operatora i agregacije upita.

Za konkretan primer šeme, bilo je logično implementirati određene filtere. Implementirani filteri su vezani za korisnika koji je kreirao taj zahtev, ski centar za koji je taj zahtev vezan, kao i status zahteva (da li je prihvaćen, odbijen, završen, ili je istekao).

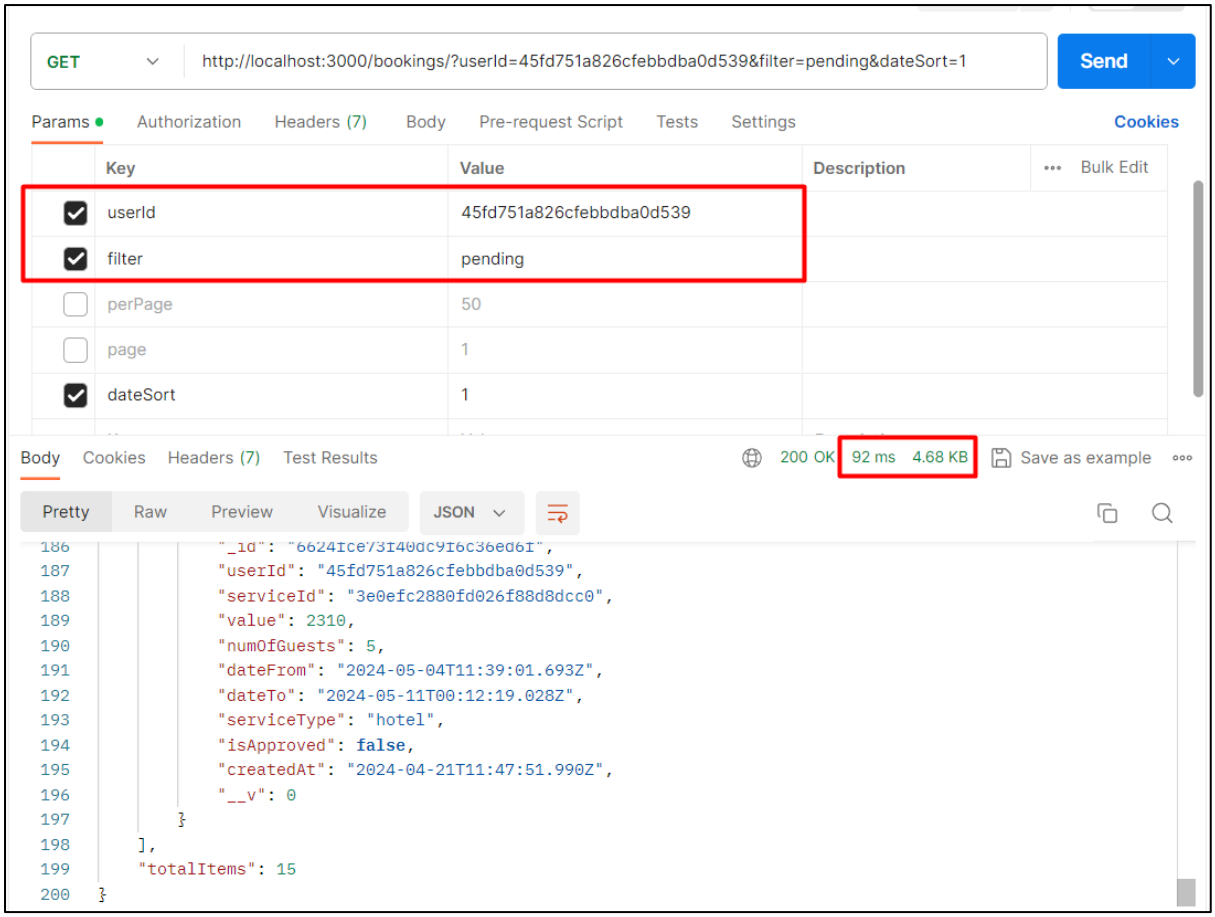
Ukoliko naš upit nadogradimo filterom za specifičnog korisnika, znatno se smanjuje vreme izvršenja zahteva, kao i količina podataka koja se šalje kroz mrežu.



Slika 1.12 Zahtev sa filtriranjem usera

Na slici se može videti, da se pri primeni filtera za određenog korisnika, smanjuje vreme izvršenja sa 1.25 sekundi, na 0.125 sekundi. Takođe, veličina podataka koja se šalje, od 3MB je spala na manje od 10 KB. Ukoliko pak dodamo filter za status zahteva, možemo da dodatno smanjimo vreme izvršenja zahteva.

`

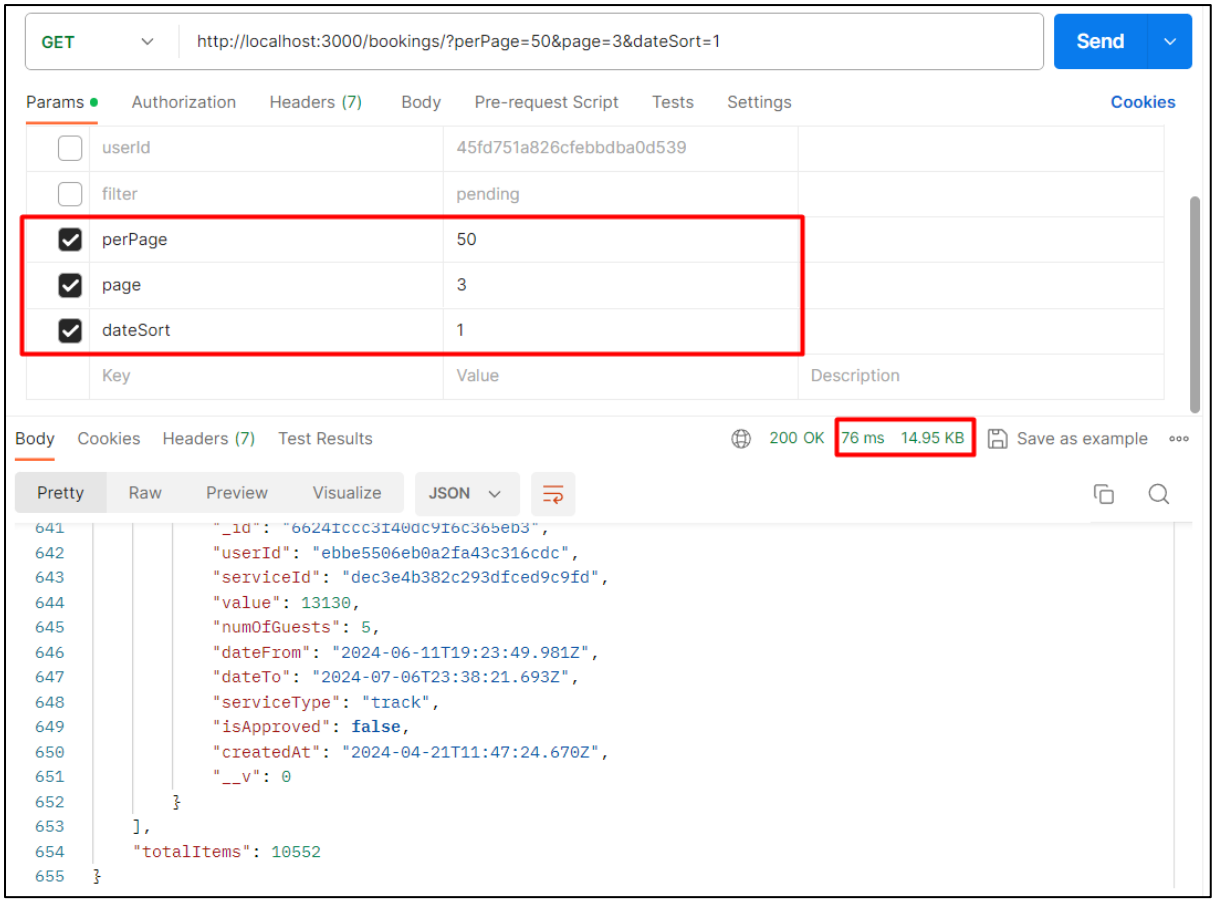


Slika 1.13 Zahtev sa filtriranjem usera i stanja zahteva

Nakon dodatnog filtriranje, za zahteve koji su samo u stanju čekanja, vreme izvšenja funkcije pada ispod 0.1 sekunde. Ova brzina izvršenja bi bila dovoljno dobra za izradu neke komercijalne aplikacije.

Sa druge strane, da bi demonstrirali rezultate optimizacije pomoću operatora **$sort**, **$limit** i **$skip**, prethodno demonstrirani filteri će biti ugašeni, da bi se demonstrirao rad sa velikom količinom podataka. Pre uključivanja filtera, vreme obrade zahteva bilo je 1,25 sekundi, dok se kroz mrežu slalo 3MB podataka. Uključivanjem operatora **$sort**, **$limit** i **$skip**, možemo obezbediti pribavljanje podataka u blokovima, gde operator limit govori koliko se podataka jednovremeno pribavljaju, dok operator skip govori koliko dokumenata treba ignorisati, da bi dobili potreban broj podataka.

`



Slika 1.14 Zahtev sa paginacijom

Primetno je drastično smanjenje vremena izvršenja zahteva, sa 1.25 sekundi na 0.076 sekundi. Prosleđivanjem parametra perPage na 50, postavili smo **$limit** parametar na 50, dok smo prosleđivanjem broja stranice (page = 3) postavili operator **$skip** na 100 (broj stranice – 1 * perPage). Time se obezbeđuje vraćanje manjeg broja podataka u više poziva, ukoliko postoji potreba za dodatnim pozivima. Tako smanjujemo vreme izvršenja individualnog zahteva, dok potencijalno povećavamo broj zahteva koji će biti upućeni serveru.

Primena agregacije upita ne postoji na ruti za vraćanje zahteva, ali postoji na rutama kao što su dodavanje novog zahteva, pretaživanje hotela koji odgovara korisnikovim zahtevima (broj gostiju, broj soba, datum ulaska i datum izslaska iz hotela), kao i ruta za odbijanje i prihvatanje zahteva.

`

```
const [pendingReq, approvedReq, user, service] = await Promise.all([
  this.bookingRepository.findQuery({
    serviceId: booking.serviceId,
    $or: [
      {
        dateFrom: { $lte: booking.dateFrom },
        dateTo: { $gte: booking.dateFrom },
      },
      {
        dateFrom: { $lte: booking.dateTo },
        dateTo: { $gte: booking.dateTo },
      },
    ],
    isApproved: false,
  }),
```

Slika 1.15 Agregacija

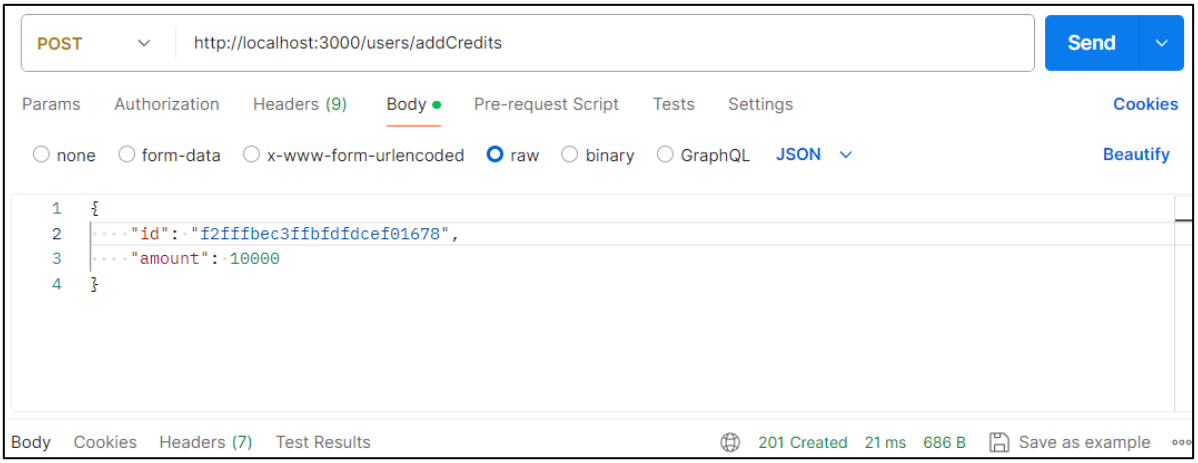
Na ovom primeru rute za prihvatanje zahteva, potrebno je obezbediti automatsko odbijanje zahteva koji su u stanju čekanja, ali zbog novonastale situacije (zauzimanja određenog broja soba od strane gostiju iz prihvaćenog zahteva), njihov zahtev ne može biti prihvaćen. Možemo videti korišcenje agregacije (operatora **$or**) za pronalaženje aktivnih (pending) zahteva, gde pronalazimo sve zahteve čiji se vremenski interval odsedanja u hotelu preklapa sa vremenskim intervalom odsedanja zahteva koji je upravo prihvaćen.

Ukoliko za ovakav zahtev ne bi koristili agregaciju, to bi značilo slanje najmanje dva zahteva serveru, što bi značajno usporilo vreme izvršenja. Uzimajući u obzir da se u istoj ruti pronalaze i aktivni zahtevi u istom vremenskom intervalu, bez korišćenja agregacije, u ovoj ruti bi se slala 5 zahteva serveru, umesto 3 koja se šalju ukoliko se koristi agregacija.

Za operacije izvršenja inkrementa na serveru, možemo videti primenu optimizacije na ruti za dodavanje kredita korisniku. U prvom slučaju, bez optimizacije, prvo pribavljamo korisnika, a nakon toga ga ažuriramo sa dodatnim kreditima.

`

```
async addCredits(id: string, amount: number): Promise<UserDocument> {
  const user = await this.userModel.findById(
    id,
  );
  if (!user) {
    throw new NotFoundException();
  }
  await this.userModel.findByIdAndUpdate(
    id,
    {wallet: user.wallet + amount}
  )
  return user;
}
```

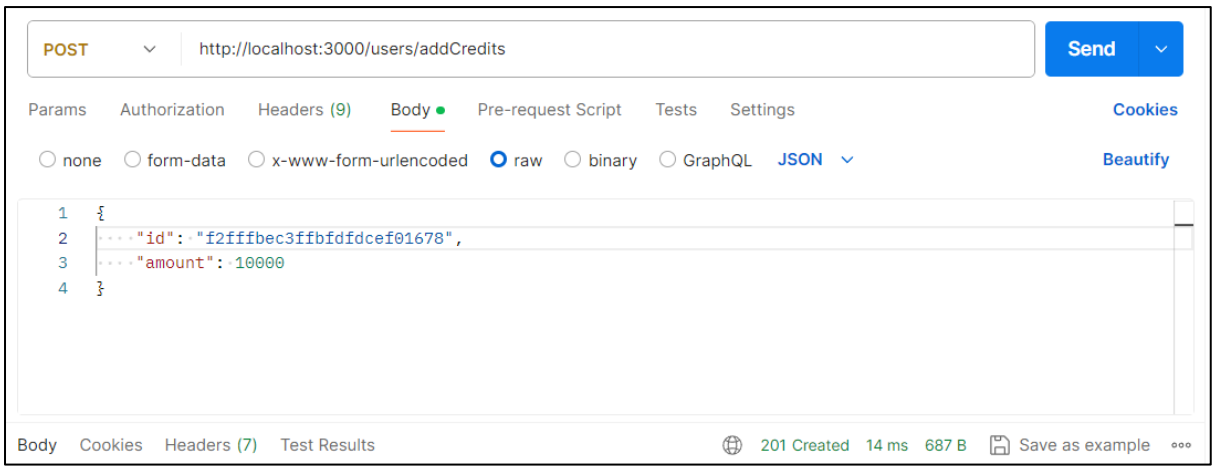


Slike 1.16 i 1.17 inkrementiranje na klijentu

U slučaju gde je primenjena optimizacija inkrementiranja na serveru, u istom zahtevu pribavljamo korisnika, i povećavamo mu broj kredita. Vreme izvršenja je neznatno smanjeno, ali bi, pri primeni na nekom većem skupu podataka ili na većem broju dokumenata istovremeno, pravilo veću razliku.

```
async addCredits(id: string, amount: number): Promise<UserDocument> {
  const user = await this.userModel.findByIdAndUpdate(
    id,
    {
      $inc: { wallet: amount },
    },
    { new: true },
  );
  if (!user) {
    throw new NotFoundException();
  }
  return user;
}
```

`



Slike 1.18 i 1.19 inkrementiranje na serveru

Problemi u optimizaciji upita kod MongoDB baze podataka

Optimizacija upita kod MongoDB baze podataka, nosi sa sobom svoje izazove. Da bi dobro optimizovala upite, osoba zadužena za to mora da ima dobro razumevanje postojećeg MongoDB optimizatora. Takođe, poznavanje rada indeksa u MongoDB-ju je ključno, jer postojanje više indeksa ne znaci nužno brže izvršavanje. Ukoliko je napravljen veći broj indeksa od dovoljnih, to može da ima suprotan efekat od želejnog. odnosno usporenje rada baze podataka. Nije poželjno niti preporučljivo praviti indekse za svaku od promenjiva, ili praviti kompozitne indekse za sve kombinacije promenjivih. Svaki indeks dodatno usporava rad baze, potovo u procesu upisa, kada se svi indeksi trebaju ažurirati, tako da je potrebno dobro razmisliti koje indekse treba kreirati.

Takođe, u nekim slučajevima, developer nema kontrolu nad izgledom šeme baze podataka. Ukoliko je šema lose napravljena, to može prouzrokovati veoma komplikovanu i tešku optimizaciju upita.

Ukoliko se optimizuje određeni upit, potrebno je da se osigura da će ta optimizacija pozitivno uticati na sve delove sistema. Ukoliko se jedan upit koristi na više različitih mesta, treba osigurati da optimizacija ne utiče negativno ni na jednu od tih ruta ili funkcija koje koriste taj upit. Na primer, ukoliko i ruta za pribavljanje korisnika, i ruta za logovanje korisnika koriste isti upit za pribavljanje, tu nije moguće primeniti projekciju, pomoću koje ćemo izostaviti lozinku korisnika pri pribavljanju. U slučaju pregleda korisnika, ona nam nije potrebna, ali u slučaju prijavljivanja, ona nam je neophodna da bi autentikovali korisnika.

`

Zaključak

Optimizacija upita kod bilo kojih baza podataka presudna je u danasnje vreme. U dobu gde ljudi svakodnevno koriste internet i aplikacije na internetu, svako ima kontakta sa različitim bazama podataka. Ljudi koji su zaduženi za kreiranje i održavanje aplikacija koje svakodnevno koristimo, obezbeđuju njihovo brzo i efikasno korišćenje, i štede vreme svim ljudima koji ih koriste.

U ovom radu obrađene su teoretske osnove baze MongoDB, globalne tehnike optimizacije upita, kao i one specifične za MongoDB. Pored toga, na primeru jedne aplikacije, koja se može koristiti u stvarnom svetu, predstavljeno je koliko tehnike optimizacije upita mogu ubrzati rad same aplikacije, skratiti procesorsko vreme, i omogućiti bolje korisničko iskustvo osobama koje bi tu aplikaciju koristile.

Osim boljeg korisničkog iskustva, optimizovani upiti omogućuju lakše održavanje baze podataka, kao i laške skaliranje baze podataka. Takođe, optimizovani upiti mogu dovesti do smanjenja troškova, zbog manjeg korišćenja resursa kao što su procesorsko vreme i memorija.

Na kraju, kreiranje optimizovanih upita ključno je za sve koji žele da naprave svoju aplikaciju ili proizvod, radi lakšeg održavanja, skaliranja i napretka same aplikacije.

`

Reference

[1] https://www.mongodb.com/company/what-is-mongodb

[2] https://www.techtarget.com/searchdatamanagement/definition/MongoDB

[3] https://www.mongodb.com/advantages-of-mongodb

[4] https://www.knowledgenile.com/blogs/pros-and-cons-of-mongodb

[5] https://nodeteam.medium.com/how-optimize-mongodb-queries-8b8a0f0fd049

[6] https://nestjs.com

[7] https://mongoosejs.com

[8] https://redis.io

[9] https://www.npmjs.com/package/@faker-js/faker