|  |  |  |
| --- | --- | --- |
|  | Univerzitet u Nišu  Elektronski fakultet |  |

Seminarski rad

**Interna struktura i organizacija podataka MongoDB**

**Mentor:** **Student:**

Doc. Dr Stanimirovic Aleksandar Kostić Jovana 1436

Sadržaj

[1. Uvod 3](#_Toc101907872)

[*2.* *NoSQL* 4](#_Toc101907873)

[2.1 Tipovi NoSQL baza podataka 4](#_Toc101907874)

[*2.2* SQL vs NoSQL 5](#_Toc101907875)

[3. *Document oriented* baze podataka 8](#_Toc101907876)

[4. MongoDB 11](#_Toc101907877)

[4.1 Arhitektura MongoDB 12](#_Toc101907878)

[4.2 Struktura i organizacija podataka 12](#_Toc101907879)

[4.2.1 Embedded data model 13](#_Toc101907880)

[*4.2.2* Normalized data model 14](#_Toc101907881)

[*4.2.3* Atomičnost *Write* operacija 15](#_Toc101907882)

[*4.2.4* Modelovanje 1:1 relacija pomoću embedded dokumenata 15](#_Toc101907883)

[*4.2.5* Modelovanje 1:n relacija pomoću embedded dokumenata 17](#_Toc101907884)

[*4.2.6* Modelovanje 1:n relacija pomoću normalizovanih dokumenata 18](#_Toc101907885)

[*4.2.6* Modelovanje n:m relacija 21](#_Toc101907886)

[*4.2.7* Modeliranje velike količine hijerarhijskih i ugnježdenih podataka 23](#_Toc101907887)

[*4.2.8* Collections 26](#_Toc101907888)

[*4.2.9* Views 27](#_Toc101907889)

[5. Postupak kreiranja modela podataka 29](#_Toc101907890)

[Zaključak 35](#_Toc101907891)

[Literatura 36](#_Toc101907892)

# Uvod

Usložnjavanje strukture i organizacije podataka na Web-u dovelo je do potrebe za novim i drugačije organizovanim arhitekturama za skladištenje podataka. Ovakve podatke karakterišu raznolikost, povezanost, velika količina takvih podataka, polustruktuiranost, kao i aplikativno specifična arhitektura. Struktuirani podaci prvenstveno zavise od kreiranja modela podataka odnosno tipova podataka poslovne logike koji će biti evidentirani i kako će oni biti skladišteni, kako će se obrađivati i način na koji će im se pristupati, što uključuje definisanje tipova samih podataka i ograničenja ulaznih podataka. Ovi podaci su veoma zavisni i ne tako fleksibilni i skalabilni. Nasuprot tome, polustruktuirani podaci su informacije koje se ne skladište u relacionim bazama podataka, ali imaju svoja karakteristična organizaciona svojstva koja olakšavaju analizu, novi pristup i nove tehnike za obradu. U ovom slučaju podatke možemo skladištiti u formatu mnogo približnijem izvornoj strukturi, odnosno poslovnoj logici. Format podataka nije konzistentan što ih čini mnogo fleksibilnijim i skalabilnijim. Primeri polustruktuiranih podataka su XML, JSON dokumenti i NoSQL baze podataka.

Relacione baze podataka postoje već više od 40 godina i one se zadovoljavale sve potrebe u vremenima kada su strukture podataka bile mnogo jednostavnije i statičnije. Međutim, kako su tehnologija i BigData aplikacije napredovale, RDBMS zasnovan na SQL-u nije mogao da isprati obimne raznolike podatke koji se brzo šire i usložnjavaju. Zato su NoSQL baze podataka postale popularnije jer nude fleksibilniju, skalabilniju i isplativiju alternativu tradicionalnim relacionim bazama podataka.

# *NoSQL*

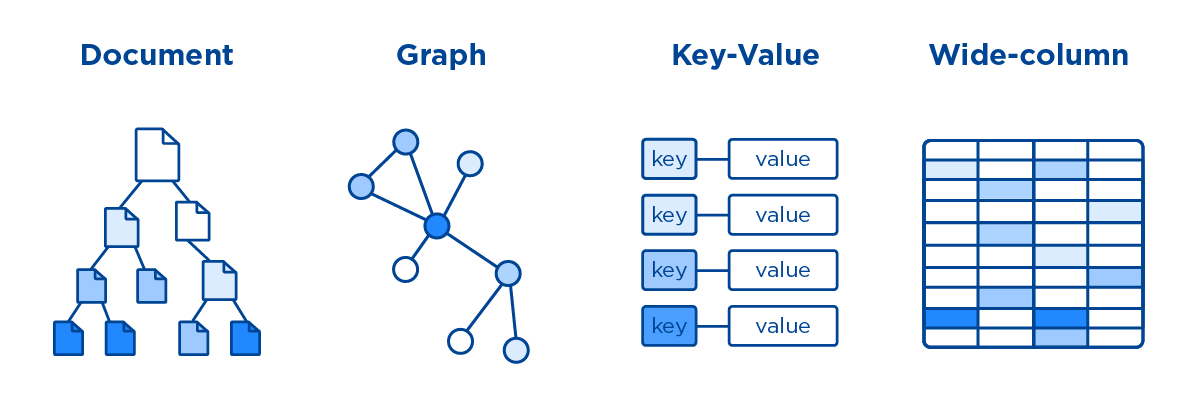
*NoSQL* baze podataka su se pojavile kasnih 2000-ih kada su se troskovi skladištenja dramatično smanjili. Kako su se troškovi skladištenja brzo smanjivali, povečavala se količina podataka koje su aplikacije trebale da skladište i obradjuju. Ovi podaci su dolazili u svim oblicima i veličinama, struktuirani, polustruktuirani i polimorfni, što je dovelo do toga da je definisanje šeme unapred postalo gotvo nemoguće. *NoSQL* baze podataka omogućavaju programerima da skladište ogromne količine nestruktuiranih podataka, dajući im veliku fleksibilnost.

*Cloud* računarstvo takođe postaje sve popularnije, a programeri počinju da koriste javne *cloud* sisteme za hostovanje svojih aplikacija i podataka. Potreba za mogućnošću distribucije podataka na više servera postajala je sve veća, kako bi aplikacije bile otpornije, da bi se smanjivale umesto da se povećavaju i da pamentno geopozicioniraju svoje podatke. Neke od *NoSQL* baza podataka, kao što je *MongoDB*, pružaju sve ove mogućnosti.

## 2.1 Tipovi NoSQL baza podataka

Vremenom su se oformila četiri glavna tipa *NoSQL* baza podataka:

* *Document oriented database*
* *Graph database*
* *Key-value database*
* *Wide-column database*



Slika 1 – Tipovi *NoSQL* baza podataka

*Document oriented* baze podataka čuvaju podatke u dokumentima sličnim *JSON* objektima. Svaki dokument sadrži parove polja i vrednosti. Vrednosti obično mogu biti različitih tipova uključujući stringove, brojeve, logičke vrednosti, nizove ili objekte.

*Graph* baze podataka čuvaju podatke u čvorovima i potezima. Čvorovi obično čuvaju informacije o ljudima, mestima i stvarima, odnosno entitetima, dok potezi čuvaju informacije o vezama između čvorova.

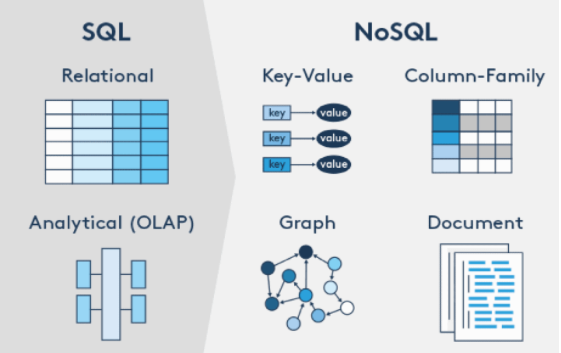
*Key-value* baze podataka su jednostavniji tip baze gde svaka stavka sadrži ključeve i vrednosti.

*Wide-column* skladišta podataka čuvaju podatke u tabelama, redovima i dinamičkim kolonama.

## *2.2* SQL vs NoSQL

Relacione baze podataka su decenijama preovlađujuća tehnologija, dokazane i široko primenjene. Obezbeđuju skladište povezanih tabela podataka koje imaju fiksni šemu, koriste SQL – Structured Query Language za upravljanje podacima i podržavaju ACID koncepte.

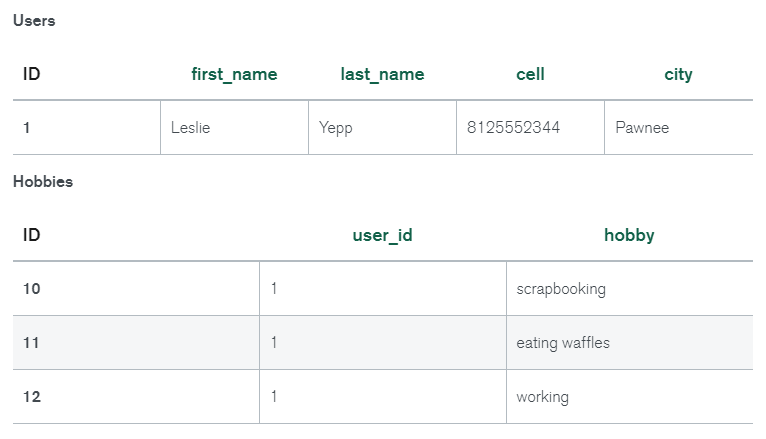
NoSQL baze podataka predstavljaju nerelaciona skladišta podataka visokih performansi. Odlikuju se svojim karakteristikama jednostavnosti upotrebe, skalabilnosti otpornosti i dostupnosti. Umesto spajanja tabela normalizovanih podataka, NoSQL skladišti nestruktuirane i polustruktuirane podatke, često u parovima ključ-vrednost ili JSON dokumentima. Obično ne podržavaju ACID koncepte izvan obima jedne particije baze podataka. Servisi velikog obima koji zahtevaju jako kratko vreme odziva, na primer manje od sekunde, favorizuju NoSQL skladišta podataka.



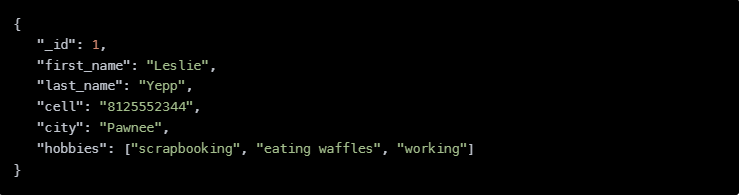
Slika 2 – *SQL vs NoSQL*

Relacione baze podataka su vertikalno skalabilne, ali su obično skupe. Pošto im je potreban jedan server za hostovanje cele baze podataka, da bi se izvršilo skaliranje mora se obezbediti veći i skuplji server. Skaliranje NoSQL baze podataka je mnogo jeftinije u poređenju sa relacionom bazom podataka, jer se kapacitet može dodatu horizontalnim skaliranjem uz pomoć jeftinijih servera.

Iako postoji niz razlika između sistema za upravljanje relacionim bazama podataka *RDBMS* i *NoSQL* baza podataka, jedna od ključnih razlika je način na koji se podaci modeluju u bazi podataka, što jasno možemo videti na jako jednostavnom primeru.



Slika 3 – Relacioni model podataka  
U relacionoj bazi podataka bismo za modelovanje korisnika i njegovih hobija morali da kreiramo dve tabele ako želimo da ih predstavimo kao dva povezana entiteta, kao što možemo videti na slici. Pored toga, da bismo dobili sve informacije o korisniku i njegovim hobijima, moramo pristupiti dvema tabelama i moramo ih *join*-ovati, što stvara dodatne troškove.



Slika 4 – Nerelacioni model podataka

Međutim, kada ovakve podatke skladištimo u *document oriented* bazi podataka, imamo mogućnost da sačuvamo jedan objekat u okviru drugog, odnosno da u okviru korisnika čuvamo niz njegovih hobija, što dovodi do niza pogodnosti. Pribavljanje informacija o korisniku i njegovim hobijima postaje daleko jednostavnije, imamo samo jedan pristup bazi jer pribavljamo jedan dokument iz baze i nisu nam potrebni nikakvi *join*-ovi, što kao rezultat daje brže pretraživanje i pribavljanje.

Glavne potrebe koje NoSQL baze podataka ispunjavaju su:

* Brz razvojni proces
* Skladištenje struktuiranih, polustruktuiranih i nestruktuiranih tipova podataka
* Ogromne količine podataka
* Zahtevi za skalabilnom arhitekturom
* Moderne aplikacione paradigme kao što su mikroservisi i strimovanje u realnom vremenu

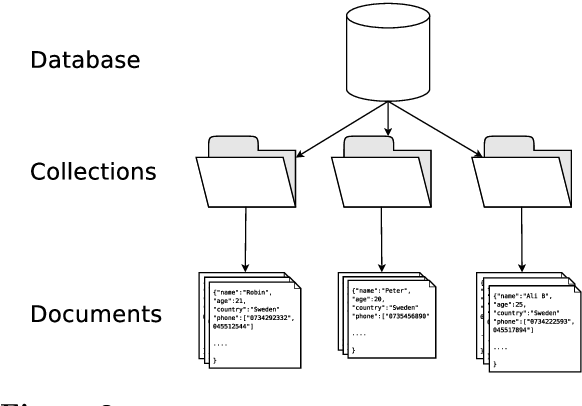
Kada govorimo o prednostima NoSQL baza podataka, moramo pomenuti i neke zablude koje ih prate. Prva zabluda jeste da su podaci o relacijama najbolje obrađeni u konceptu relacionih baza, a zapravo i nerealcione baze podataka čuvaju podatke o relacijama samo na drugačiji način. U poređenju sa relacionim bazama podataka, zapravo je lakše modeliranje relacija u *NoSQL* bazama podataka jer povezani podaci ne moraju da se dele u tabele. Nerelacioni modeli podataka omogućavaju da povezani podaci budu ugnježdeni unutar jedne strukture podataka.

Još jedna zabluda na koju se često nailazi je da nerelacione baze uopšte ne podržavaju *ACID* koncept. Neke *NoSQL* baze podataka kao što je *MongoDB* zapravo podržavaju *ACID* transakcije. Međutim, treba imati na umu da način na koji se modeluju podaci mogu eliminisati potrebu za transakcijama, kao što je slučaj u našem primeru, jer bi ažuriranje u relacionom modelu podrazumevalo transakciju ažuriranja dve tabele, dok u nerelacionom ažuriramo samo jedan document koji ne zahteva *multi-record* transakciju.

# *Document oriented* baze podataka

*Document* baze podataka su baze koje skladište informacije odnosno podatke u dokumentima. Ovaj tip baza podataka nudi veliki broj pogodnosti, uključujući intuitivni model podataka koji omogućava brži i lakši rad, fleksibilnu šemu koja omogučava da se model podataka razvija i menja u skladu sa potrebama aplikacije, kao i mogućnost horizontalnog skaliranja.

*Document oriented* baze podataka umesto skladištenja u fiksnim redovima i kolonama, koriste fleksibilne dokumente i smatraju se najrasprostranjenijom alternativom tabelarnim, relacionim bazama podataka. Dokument predstavlja *record,* odnosno zapis ili instancu u samoj bazi. Dokument obično čuva informacije o jednom objektu i svim njegovim relacijama u formatu atribut-vrednost(*field-value*). Vrednosti mogu biti različite vrste i strukture, uključujući stringove, brojeve, datume, polja ili objekte. Ovi dokumenti mogu se čuvati u formatima kao što su *JSON, BSON* i *XML*.

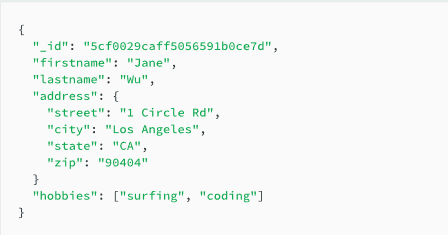


Slika 5 – *Document oriented* baza podataka

Grupisanje dokumenata prema sličnom sadržaju možemo postići korišćenjem kolekcija(*collections*). Međutim, ne moraju svi dokumenti jedne kolekcije da imaju ista polja(*fields*) zato što ovaj tip baze podataka ima veoma fleksibilnu šemu podataka. Ako ipak postoji potreba za striktnom šemom, *document oriented* baze pružaju mogućnost validacije šeme, tako da se opciono mogu primeniti pravila odnosno ograničenja koja preciziraju strukturu dokumenata.

Tri ključna faktora koja čine ovaj tip baze podataka drugačijim od standardnih relacionih baza su: intuitivnost modela podataka, korišćenje *JSON* dokumenata i fleksibilnost šeme podataka.

Kada govorimo o intuitivnosti modela, dokumenti se mapiraju na objekte direktno u samom kodu konkretnog programskog jezika, pa je samim tim mnogo prirodnije raditi s njim. Nema potrebe za dekompozicijom podataka po tabelama, izvršavanjem skupih *join*-ova ili integrisanjem zasebnog sloja za relaciono mapiranje objekata. Podaci kojima se pristupa zajedno se i čuvaju zajedno, što developerima omogućava manju količinu koda, a korisnicima bolje performanse.



Slika 6 – Skladištenje podataka u dokumentu

*JSON* je postao ustaljeni standard za razmenu i skladištenje podataka koji je nezavistan od jezika i čitljiv za ljude. Dokumenti su nadskup svih drugih modela podataka, tako da developeri mogu struktuirati podatke na način shodan njihovim aplikacijama.

Šema dokumenta odnosno modela je dinamična i ne zahteva predefinisanje podataka pre samog kreiranja i dodavanja. Atributi se mogu razlikovati od dokumenta do dokumenta, što omogučava developerima da modifikuju strukturu u bilo kom trenutku, izbegavajući kritične migracije šeme.

*Document oriented* baze podataka su distribuirane što omogućava horizontalno skaliranje koje je daleko jeftinije od vertikalnog skaliranja, a njihova sposobnost replikacije doprinosi otpornosti i pouzdanosti sistema.

Pomoću *API*-ja ili jezika upita podržan je rad sa *CRUD* operacijama, kao i pretraživanje dokumenata na osnovu jedinstvenih identifikatora ili vrednosti atributa, što podrazumeva:

**Create** – kreiranje dokumenata, pri čemu svaki od njih ima jedinstveni identifikator

**Read** - API ili jezik upita omogućavaju pretraživanje podataka koristeći njihove jedinstvene identifikatore ili vrednosti atributa, a mogu biti dodati i indeksi radi poboljšanja performansi čitanja

**Update** – Postojeći dokumenti mogu biti ažurirani kako parcijalno tako i u celini

**Delete** – Dokumenti mogu biti izbrisani iz baze podataka

*Document oriented* baze podataka su baze podataka opšte namene i široko su primenjene, kako u transakcionim tako i u analitičkim aplikacijama. Neki od konkretnih primera primene ovog tipa su:

* *Data hub*
* Korisničko upravljanje podacima
* *Internet Of Things*
* Obrada plaćanja
* Mobilne aplikacije
* *Real-time* analitika

# MongoDB

Predstavnikom *NoSQL* *document oriented* baza podataka smatra se *MongoDB* koja je *open-source cross-platform* baza podataka. Softverska kompanija 10gen počela je da razvija *MongoDB* 2007. godine kao deo planirane servisne platforme, a već 2009. kompanija je prešla na *open-source* razvoj, pri čemu je nudila komercijalnu podršku i druge servise. *MongoDB* se od 2019. udružio sa *Alibaba Cloud*-om i od tada nudi svojim klijentima *MongoDB* kao servisno rešenje.

Ključne prednosti ove baze podataka ogledaju se pre svega u tome što pružaju visoke performanse i perzistentnost podataka. Podrška za ugrađene modele podataka smanjuje *I/O* aktivnosti u samoj bazi podataka, dok indeksi podržavaju brže upite i mogu uključivati ključeve iz ugrađenih dokumenata i nizova.

*MongoDB* nudi bogat jezik upita koji pored *CRUD* operacija nudi i upite koji koriste regularne izraze za pretraživanje, pretraživanje u okviru određenih rangova odnosno opsega, agregaciju podataka i kreiranje geoprostornih upita.

Takođe, *MongoDB*-ova mogućnost replikacije nazvana je skup replika, sastoji se od dve ili više kopija podataka. Svaki član skupa replika, odnosno svaka kopija, može se naći kako u ulozi primarne tako i u ulozi sekundarne replike u bilo kom trenutku. Sva upisivanja i čitanja se podrazumevano obavljaju na primarnoj replici. Kada dodje do greške u primarnoj replici, nad skupom replika se sprovodi izborni proces da bi se odredilo koja če sekundarna replika postati primarna. Zahvaljujući ovakvom mehanizmu, skup replika se može definisati kao grupa *MongoDB* servera koji održavaju isti skup podataka, obezbeđujući redundantnost i povećavajući dostupnost podataka kao i njihovu pouzdanost.

Još jedna od veoma značajnih prednosti je horizontalno skaliranje omogučeno *sharding* metodom za distribuciju podataka kroz *cluster* mašina. Korisnik bira *shard* ključ koji određuje koji će podaci biti distribuirani. Podaci se na osnovu ključa dele na opsege i distribuiraju na više delova. *MongoDB* može da radi na više servera, balansirajuci opterećenje i duplikaciju podataka kako bi sistem održao u funkciji i slučaju greške.

## 4.1 Arhitektura MongoDB

*MongoDB* ispunjava zahteve modernih aplikacija sa platformom podataka aplikacije izgrađenom na nekoliko arhitektonskih osnova:

1. Intuintivan dokumentni model podataka i moćan interfejs za upite koji developerima daju najbrži način za inovacije u kreiranju transakcionih, operativnih i analitičkih aplikacija.
2. *Multi-cloud* globalna baza podataka koja daje slobodu pokretanja aplikacije bilo gde i fleksibilnost da se premeštaju na privatne ili javne *cloud*-e u zavisnosti od zahteva, bez potrebe za promenom i jedne linije koda aplikacije.
3. Jedinstveno iskustvo za moderne aplikaije podržano sveobuhvatnim i integrisanim servisima za developere

## 4.2 Struktura i organizacija podataka

Iako *MongoDB* nema striktnu šemu, postoji podrazumevana struktura, a samim tim i model podataka. Pri kreiranju modela treba voditi računa o tome koji će se podaci skladištiti, kojim podacima će se verovatno pristupati zajedno, koliko često će se pristupati nekom delu podataka i koliko će brzo podaci rasti i da li će biti neograničeni. Odgovori na sva ova pitanja dovode do modela koji odgovara konkretnoj aplikaciji. Svakako, ne postoji prethodna priprema podataka u smislu procesa formatiranja, algoritama ili pravila.

Pošto nemamo šeme, lako možemo promeniti strukturu u kojoj se skladište podaci. Dokumenti u kolekciji ne moraju da budu u skladu sa jednom stogo striktnom strukturom podataka. Međutim, cenu nepostojanja šeme možemo platiti softverskim greškama, jer kada se podaci čuvaju na mnogo različitih načina složenost aplikacije se povećava, a samim tim i verovatnoća nastanka greške. Ovaj nedostatak možemo nadomestiti sofisticiranim validacijama koje će kod aplikacije učiniti jednostavnijim.

Dok je *MongoDB* bez šeme, u stvarnosti podaci su retko potpuno nestruktuirani. Većina podataka ima neku podrazumevanu strukturu, iako se možda neće vršiti konkretna eksplicitna validacija. Čak i kada se podaci vremenom razvijaju, postoji neka osnovna struktura koja ostaje konstanta.

Validacija dokumenata moguća je počevši od verzije 3.2. Može biti jako korisna u slučajevima kada imamo česte promene strukture dokumenta, kao i kod podataka sa komplikovanom strukturom, gde olakšava razumevanje i obradu podataka.

Fleksibilnost šema olakšava mapiranje dokumenata na entitete ili objekte. Svaki dokument odgovara konkretnim atributima entiteta, čak i ako dokument ima varijacije i odstupanja u odnosu na ostale dokumente u kolekciji.

Dokumenti koji koristi *MondoDB* slični su *JSON* objektima. Vrednosti atributa mogu uključivati druge dokumente, polja kao i nizove dokumenata. *Record*, odnosno jedna instanca je zapravo jedan dokument, čija je struktura podatak organizovana u parove atribut-vrednost.



Slika 7 – *Field – value*  parovi

Glavne prednosti korišćenja dokumenata su:

* Dokumenti odgovaraju izvornim tipovima podataka u mnogim programskim jezicima
* Ugnježdeni dokumenti i polja smanjuju potrebu za skupim *join-*ovima
* Fleksibilna i dinamička šema, odnosno model podržavaju polimorfizam

Ključna odluka u kreiranju modela podataka zavisi od strukture dokumenta i načina na koji aplikacija predstavlja relacije medju podacima, odnosno da li ima smisla ugraditi informacije u dokument ili ih čuvati u zasebnoj kolekciji. *MongoDB* omogućava da se srodni podaci ,,ugrade“ u jedan dokument, pa kao rezultat toga imamo dva načina organizacije podataka:

* *Embedded data model*
* *Normalized data model*

## 4.2.1 Embedded data model

*MongoDB* pruža mogućnost da se povezani podaci ugrade u jednu strukturu, odnosno dokument. Ova šema je poznata kao ,,denormalizovani“ model. *Embedded data model* dozvoljava aplikaciji da skladišti povezane delove informacija u okviru istog skupa podatka odnosno *record*-a, u vidu atributa ili niza.

  
Slika 8 – *Embedded* model podataka

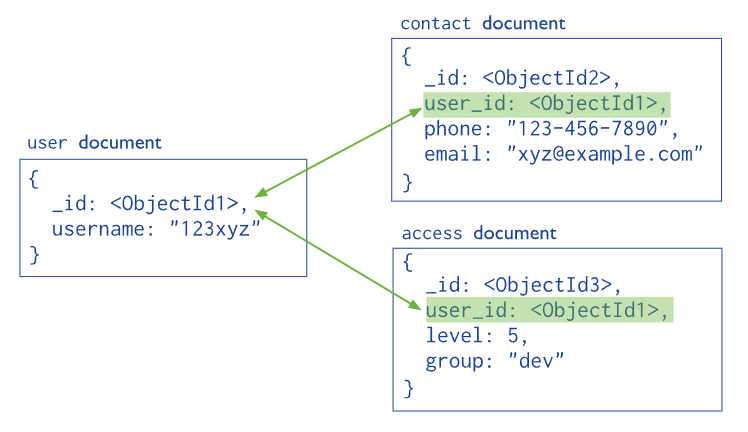
Glavna primena ovog tipa strukture podataka je:

* Modelovanje 1:1 relacije
* Modelovanje 1:n relacije

Ovaj model podataka je primenjiv kada postoji relacija ,,sadrži“. Čitanja su efikasna kada postoji potreba da se se pristupi ugnježdenim dokumentima zajedno sa glavnim dokumentom. Dokument se takođe može ažurirati u jednoj atomičnoj operaciji. Za modeliranje relacija 1:n, niz dokumenata može biti ugnježden u jednu strukturu. Međutim *MongoDB* ograničava veličinu dokumenta na 16MB i najviše 100 nivoa ugneždavanja.

## *4.2.2* Normalized data model

Normalizovani model podataka koristi reference objekata za modeliranje odnosa izmedju dokumenata. *MongoDB* takođe omogućava da referenciranjem *join*-ujemo podatke iz različitih kolekcija.



Slika 9 – Normalizovani model podataka

Najčešća upotreba ovog modela podataka:

* Sprečavanje dupliranja podataka
* Modelovanje n:m relacija
* Modelovanje ogromnih hijerarhijskih modela podataka

Takav model sprečava dupliranje podataka jer se na dokument može referencirati mnogi drugi elementi bez dupliranja njegovog sadržaja, što embedded modeli mogu izazvati. Složene relacije n:m se veoma jednostavno modeliraju na ovaj način. Ogromni hijerarhijski skupovi podataka se mogu jednostavno modelovati i mogu se referencirati na kolekcije.

## *4.2.3* Atomičnost *Write* operacija

*Single Document Atomicity –* u *MongoDB*-u operacija upisa je atomična na nivou jednog dokumenta, čak i ako operacija modifikuje više ugrađenih dokumenata unutar jednog dokumenta. Denormalizovani model podataka sa ugrađenim podacima kombinuje sve povezane podatke u jednom dokumentu umesto da se normalizuje u više dokumenata i kolekcija. Ovaj model podataka doprinosi atomičnosti operacija.

*Multi-Document Transactions* – kada jedna operacija upisa modifikuje više dokumenata, modifikacija svakog dokumenta je atomična operacija, ali operacija u celini nije atomična. Kada se obavljaju *multi-document* operacije upisa, bilo da se radi o *single write* ili *multiple write* operacijama,druge operacije se mogu preplitati. Za situacije koje zahtevaju atomičnost čitanja ili pisanja u više dokumenata, u jednoj ili više kolekcija, *MongoDB* podžava transakcije sa više dokumenata. Na primer, u verziji 4.0 podržava *multi-document* transakcije nad skupovima replika. Takođe, u verziji 4.2, *MongoDB* uvodi distribuirane transakcije, što dodaje podršku za *multi-document* transakcije nad *sharded cluster*-ima i nad njima uključuje primenu postojeće podrške za ove transkacije nad skupovima replika.

## *4.2.4* Modelovanje 1:1 relacija pomoću embedded dokumenata

Sledećim primerom ilustrovano je mapiranje 1:1 relacije uz pomoć *embedded* modela podataka na primeru korisnika i detalja vezanih za njegovu adresu. Pre svega imamo dve dokumenta koja predstavljaju dva zasebna entiteta, odnosno objekta.

  
Slika 10 – Dva entiteta predstavljena dokumentima

Ovaj primer ilustruje prednost ugrađivanja podataka u odnosu na referenciranje ako posmatramo jedan entitet podataka u kontekstu drugog. U ovoj relaciji 1:1 između podataka korisnika i adrese, adresa pripada korisniku i treba biti ,,ugradjena“ u model korisnika.

  
Slika 11 - Modelovanje 1:1 relacija pomoću *embedded* dokumenata

Sa ovakvim *embedded* modelom podatka, sve informacije o jednom korisniku mogu se pribaviti jednim *query*-ijem.

U kontekstu ovog primera, u normalizovanom modelu podataka, adresni dokument bi sadržao referencu na korisnički dokument.

Potencijalni problem sa ovim *embedded document pattern*-om je to što može dovesti do velikih dokumenata koji mogu sadržati polja koja aplikaciji nisu potrebna. Ovi nepotrebni podaci mogu izazvati dodatno opterećenje i usporiti aplikaciju. Umoesto toga, može se iskoristiti *subset pattern* da bi se pribavio podskup podataka kome se najčešće pristupa. Ovo je suprotni pristup od *embedded document pattern*-a, ali je pogodan za dobru organizaciju podataka ako imamo deo podataka kojima se ne pristupa često. Ovaj *pattern* zapravo ima za cilj da se jedan dokument podeli na dva, tako da imamo jedan dokument kojem se često pristupa, a drugi kojem redje pristupamo čime poboljšavamo performanse sistema jer ne pristupamo uvek svim podacima i ako se oni zapravo kontekstno odnose na isti objekat.

Korišćenje manjih dokumenata koji sadrže podatke kojima se češće pristupa smanjuje ukupnu veličinu radnog skupa. Ovi manji dokumenti rezultati rezultiraju poboljšanim performansama čitanja i čine višre memorije dostupnom za aplikaciju. Međutim, i sa ovim pristupom treba biti pažljiv, razumeti potrebe aplikacije i što bolje osmisliti koncept. Ako se pogrešno podele podaci u više kolekcija, aplikacija će često morati da pristupa bazi i da se oslanja na *join* operacije da bi preuzela sve potrebne podatke.

## *4.2.5* Modelovanje 1:n relacija pomoću embedded dokumenata

Pored relacija 1:1, uz pomoć *embedded data modela* mogu se modelirati i relacije 1:n, na jako jednostavan način sličan prethodnom, održavajući i dalje atomičnost pristupa i ažuriranja. I dalje će biti moguće jednim pristupom obaviti pribavljanje svih informacija o jednom entitetom čak i ako on u sebi sadrži više drugih, odnosno niz drugih objekata.

  
Slika 12 – Tri entiteta predstavljena dokumentima

Ovo modelovanje možemo ilustrovati gotovo istim primerom samo će sada korisnik imati više od jedne adrese. Primer takodje ilustruje i prednost ugrađivanja u odnosu na referenciranje, u slučaju da treba da imamo više entiteta u kontekstu jednog. U ovoj relaciji ,,jedan na više“ između podataka korisnika i adrese, korisnik ima više entiteta adrese.

Ako aplikacija često preuzima osnovne podatke o korisniku, kao što je na primer ime korisnika, zajedno sa podacima o adresama, na primer naziv ulice, najpogodnije rešenje su *embedded document pattern*-i, jer će kao što je već pomenuto svi neophodni podaci o jednom korisniku biće pribavljeni jednim pristupom bazi.

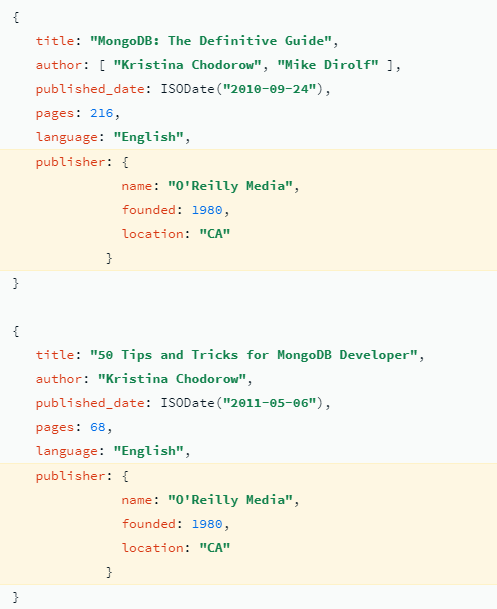
  
Slika 13 - Modelovanje 1:n relacija pomoću *embedded* dokumenata

U ekvivalentnom normalizovanom modelu, svi adresni dokumenti imali bi referencu na korisnički dokument.

## *4.2.6* Modelovanje 1:n relacija pomoću normalizovanih dokumenata

Za razliku od modeliranja relacija 1:1 i 1:n korišćenjem *embedded documet pattern*-a, sledi primer gde je pogodnije koristiti normalizovani model podataka, koji koristi reference da bi povezao entitete između kojih postoje zavisnosti. Sledeći primer ilustruje prednost referenciranja u odnosu na ugrađivanje kako bi se izbeglo ponavljanje informacija.

U sledećem primeru, metoda ugrađivanja dokumenta koji sadrži informacija o izdavaču u dokument koji sadrži informacije o knjizi, dovelo bi do dupliranja podataka izdavača i sve prednosti korišćenja ovog načina modelovanja podstale bi zapravo mane, jer bi se skup podataka povećavao a zapravo se radi o dupliranim a ne o korisnim podacima.

  
Slika 14 – Ilustracija negativnih strana *embedded* pristupa

Da bi se izbeglo ponavljanje podataka, kao pogodno rešenje nameću se normalizovoni modeli podataka gde bismo u okviru dokumenta sa informacijama o knjizi čuvati samo reference na izdavača, a sam izdavač nalaziće se u drugom dokumentu.

Kada koristimo referenciranje, rast broja relacija određuje gde će se referenca čuvati. Ako je broj knjiga po izdavaču mali ili sa ograničenim rastom, čuvanje reference knjige u dokumentu izdavača može biti korisno. U suprotnom, ako je broj knjiga po izdavaču neograničen, ovaj model podataka bi doveo do često promenljivih i rastućih nizova kao u sledećem primeru:

  
Slika 15 – Prednost referenciranja

Da bi se izbegao ovakav tip nizova, promenljiv, rastući i neograničen, referencu izdavača je bolje sačuvati u okviru dokumenta knjige, što je ilustrovano primerom na slici . Međutim ovaj tip organizacije podataka u mnogome zavisi od potreba i zahteva aplikacije, kao i od toga kako će podaci rasti i usložnjavati se.

  
Slika 16 – Obrnuti pristup kod referenciranja

## *4.2.6* Modelovanje n:m relacija

U slučaju veza više na više način modeliranja najviše zavisi od potreba aplikacije. Za razliku od relacionih modela, *MongoDB* omogućava modelovanje ovakvih relacija bez kreiranja dodatnih entiteta i dokumenata. Takođe se mogu koristiti i *embedded i normalized* modeli podataka ali možemo primeniti i dva načina pristupa kao što su *two way embedding*  i *one way embedding.*

Za potrebe ilustrovanja modeliranja n:m relacija imamo jednostavan primer gde je jedna knjiga napisana od strane više autora, kao i obrnuto, jedan autor može napisati više različitih knjiga. Ako koristimo *two way embedding* pristup, međusobne reference biće uključene u okviru oba entiteta, što će omogućiti pretraživanje u oba smera. U smislu da možemo pretraživati prema tome koji su autori napisali jednu knjigu gde bismo pristupali entitetu knjige ili obrnuto, koje je knjige napisao jedan autor, gde bismo pristupali entitetu autora.

|  |  |
| --- | --- |
|  |  |
| Slika 17 - *two way embedding* pristup kod referenciranja | |

Prvo imamo dva dokumenta autora, sa referencama na knjige koje su napisali, a zatim imamo dve knjige sa referencama na autore koji su ih napisali.

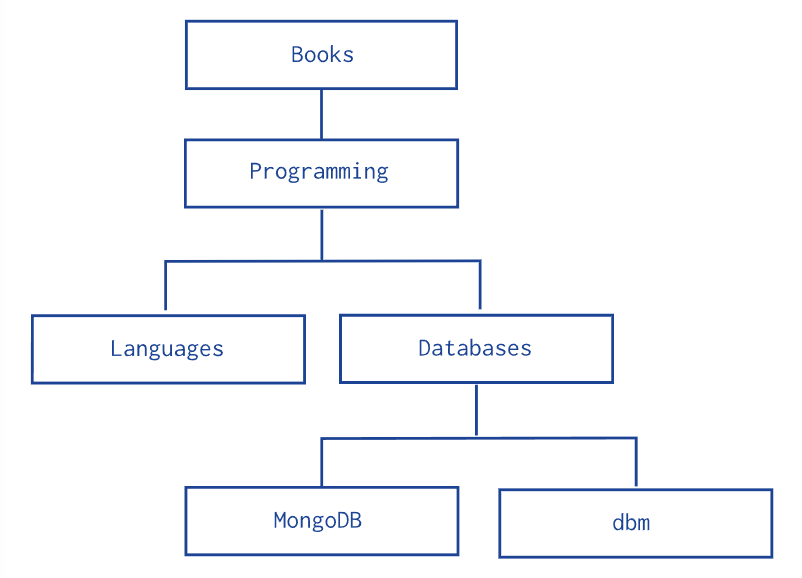
Međutim, postoji i *one way embedding* pristup koji doprinosi optimizaciji performansi čitanja kod n:m relaciija ,,ugrađivanjem“ referenci samo na jednoj strani relacije, odnosno samo u jednom entitetu. Primer može biti jako jednostavan, kada imamo slučaj da nekoliko knjiga spada u više kategorija, ali i kategorije imaju mnogo knjiga.

|  |  |
| --- | --- |
|  |  |
| Slika 18 - *one way embedding* pristup kod referenciranja | |

U ovom slučaju nije neophodno da kategorije imaju reference na sve knjige koje pripadaju toj kategoriji, jer jednostavno aplikaciji to ne mora biti od interesa, a sama fleksibilnost šema nam omogućava ovakav pristup. Knjiga ima referencu na kategoriju, može imati reference na više kategorija ali broj kategorija može biti daleko manji od broja knjiga. Na primer, ako bismo pamtili za svaku kategoriju reference na sve knjige, broj referenci bio bi ogroman, jer mnoštvo knjiga može pripadati jednoj kategoriji, a ta informacija kategorijama a i samoj aplikaciji možda uopšte nije od značaja, zato i modeliranje ovakvih relacija najviše zavisi od konteksta i potreba aplikacije i korisnika.

## *4.2.7* Modeliranje velike količine hijerarhijskih i ugnježdenih podataka

*MongoDB* zaista pruža mnogo načina za skladištenje veoma raznovrsnih podataka. Za velike količine hijerarhijskih podataka ili za ugnježdene podatke *MongoDB* koristi strukturu stabla. Ovaj pristup nam omogućava da svaki čvor stabla pamtimo kao dokument. Za ilustrovanje ovakvog modelovanja uzećemo primer sledeće hijerarhije stabla:

  
Slika 19 – Hijerarhijski model podataka - stablo

U nastavku, biće predstavljeni neki od najčešćih načina korišćenja strukture stabla, na primeru ovog stabla, pri čemu vrednosti ovih čvorova predstavljaju identifikator kategorije kojoj pripadaju.

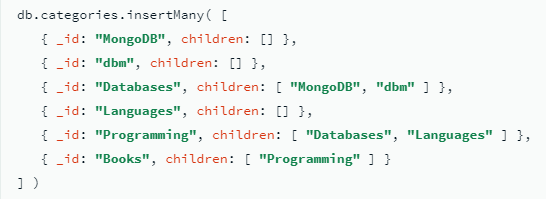
Model Tree Structures with Parent References

Parent references pattern pored toga što skladišti svaki čvor stabla kao dokument, za svaki čvor čuva i identifikator roditeljskog čvora, ako roditelj postoji. Modelovanje u ovom slučaju izgledalo bi ovako:

  
Slika 20 - Model Tree Structures with Parent References

Model Tree Structures with Child References

Child references pattern skladišti svaki čvor stabla kao dokument i za svaki čvor čuva i identifikatore čvorova dece u vidu polja, odnosno niza, ako deca čvorovi postoje. Modelovanje na osnovu istog početnog grafa izgledalo bi sada ovako:

  
Slika 21 - Model Tree Structures with Child References

Model Tree Structures with an Array of Ancestors

The Array of Ancestors pattern takođe čvorove skladišti u dokumente, a uz to i niz identifikatora predaka trenutnog čvora, odnosno putanje od početnog root čvora pa sve do tekućeg. Ovaj pristup pruža brzo i efikasno rešenje za pronalaženje potomaka i predaka čvora kreiranjem indeksa na elementima polja predaka, što ga čini dobrim izborom za rad sa podstablima.

U poređenju sa modelom koji će sledeći biti analiziran – Materialized path pattern, ovaj pristup je nešto sporiji ali jednostavniji za upotrebu.

Modeliranje uz pomoć ovog pristupa a takođe na osnovu početnog grafa vrednosti izgleda ovako:

  
Slika 22 - Model Tree Structures with an Array of Ancestors

Model Tree Structures with Materialized Paths

The Materialized Path pattern kao dodatne informacije o čvorovima skladišti identifikatore predaka ili putanje kao i prethodno opisani pristum, ali sada ne u vidu niza ili polja već u obliku stringa. Iako ovaj pristup zahteva dodatne korake rada sa stringovima i regularnim izrazima, pruža veću fleksibilnost u radu sa putanjom, kao što je pronalaženje čvorova na osnovu parcijalnih putanja.

Sledeći primer modelira do sada korišćeno stablo koristeći materijalizovane putanje, pri čemu string koji se pamti kao atribut path koristi zapetu u okviru stringa, izmedju navedenih vrednosti identifikatora.

  
Slika 23 - Model Tree Structures with Materialized Paths

Model Tree Structures with Nested Sets

Nested Sets pattern identifikuje svaki čvor u stablu kao zaustavljanje u kružom obilasku stabla. Svaki čvor se posećuje dva puta, prvi put tokom inicijalnog obilaska, a drugi pri povratku. Ovaj pristup pored osnovnih informacija o čvoru skladišti i identifikatore roditeljskog čvora, čvora inicijalnog zaustavljanja kao atribut left i čvora zaustavljanja pri povratku kao right.

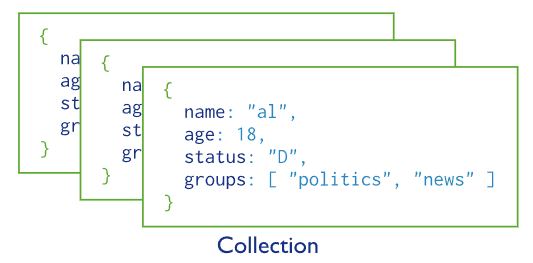
Modelovanje na osnovu polaznog stabla:

  
Slika 24 - Model Tree Structures with Nested Sets

Ovaj model pruža efikasno i brzo rešenje za pronalaženje podstabala, ali nije efikasan za modifikaciju strukture stabla. Kao takav, ovaj obrazac je najbolji za statična stabla koja se ne menjaju.

## *4.2.8* Collections

*MongoDB* skladišti dokumente u kolekcije, gde je uglavnom ključni kriterijum za jednu kolekciju baziran na sličnom sadržaju dokumenata. Kolekcije predstavljaju analogiju za tabele u relacionim bazama podataka, dok sama baza podataka skladišti jednu ili više kolekcija dokumenata. Njihova glavna uloga je grupisanje sličnih dokumenata.

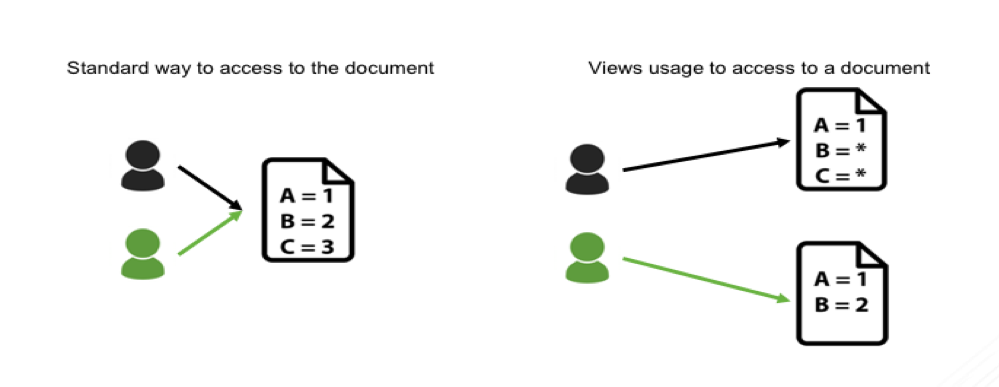


Slika 25 – Kolekcija dokumenata

Podrazumevano, kolekcije ne zahtevaju da njihovi dokumenti imaju istu šemu. Dokumenti u okviru iste kolekcije ne moraju da imaju isti skup atributa i sami tipovi atributa mogu da se razliku medju dokumentima u kolekciji. Da bismo promenili strukturu dokumenata u kolekciji, na primer, dodali nove atribute, uklonili postojece ili promenili vrednost atributa na novi tip, neophodno je samo ažurirati dokumente.

## *4.2.9* Views

*MongoDB View* je *queryable* tip objekta čiji je sadržaj definisan agregacijom *pipeline*-a nad kolekcijama ili drugim *view*-ovima. *MongoDB*-ov agregacioni *pipeline* obrađuje dokumente jedne kolekcije kroz jednu ili više faza. *View*-ovi su *read-only* rezultati tih agregacija koji pružaju mogučnost korisniku da jednostavno pročita te rezultate kada su mu neophodni a ne da ponovo izvršava agregaciju. Kada klijent pretražuje *view*, *MongoDB* nadovezuje njegov *query* na osnovni *pipeline* i vraća rezultat kombinovanog *pipeline*-a. Postoji i mogućnost optimizacije agregacije *pipeline*-a. *MongoDB* može zahtevati *permission-*e od klijenta za pretraživanje samog view-a.



Slika 26 - *Views*

Sadržaj samog *view-a* nije perzistentan na disku i kreira se na zahtev klijenta, što znači da su po prirodi nematerijalizovani, tj. kreiraju se i ažuriraju u hodu. Međutim, od verzije *MondoDB-*a 4.2 materijalizovani *view*-ovi na zahtev su takođe mogući uz pomoć *merge pipeline*-a, pa se tako u zavisnosti od kolekcije mogu kreirati:

* *Read-only Views*
* *On-Demand Materialized Views*

*Read-only Views* zauzimaju veoma malo prostora za skladištenje jer se podaci samog view-a ne skladište, već samo njihova definicija. On*-Demand Materialized Views* čuvaju se u izlaznoj kolekciji koja se ažurira svaki put kada se *pipeline* izvrši.

*Views* su korisni za najrazličitije potrebe. Donose prednosti u pogledu privatnosti podataka, unapređivanja performansi, ponovne upotrebe koda i analitike. Takođe, dodaju sloj apstrakcije izmedju aplikacije i kompleksnih agregacija *pipeline*-a.

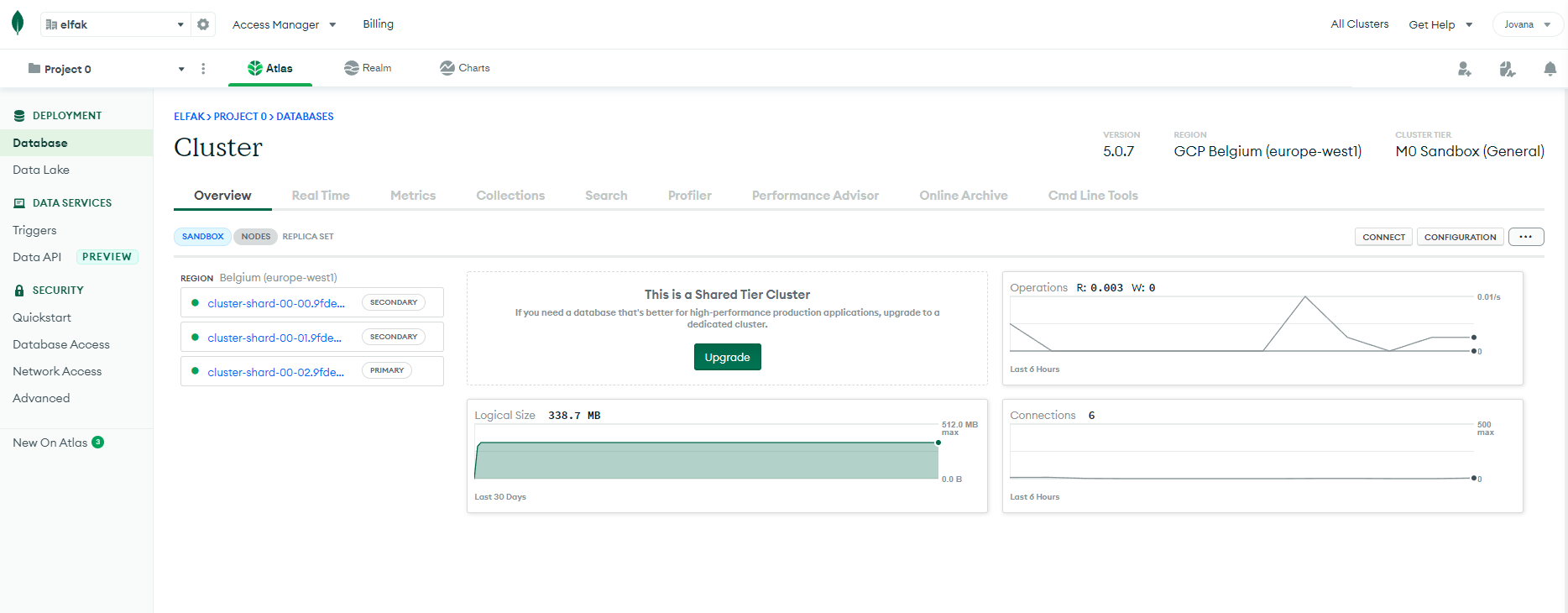
# 5. Postupak kreiranja modela podataka

*MongoDB* nudi i lokalni i *cloud* pristup razvoja. Za lokalni pristup imamo *Community* i *Enterprise* verziju baze podataka, a za *cloud* pristup *MongoDB Atlas* koji je zapravo hostovana *Enterprise* servisna opcija na *cloud-*u koja ne zahteva nikakvu instalaciju. *MongoDB Atlas*  je *multi-cloud* je aplikaciona platforma podataka.

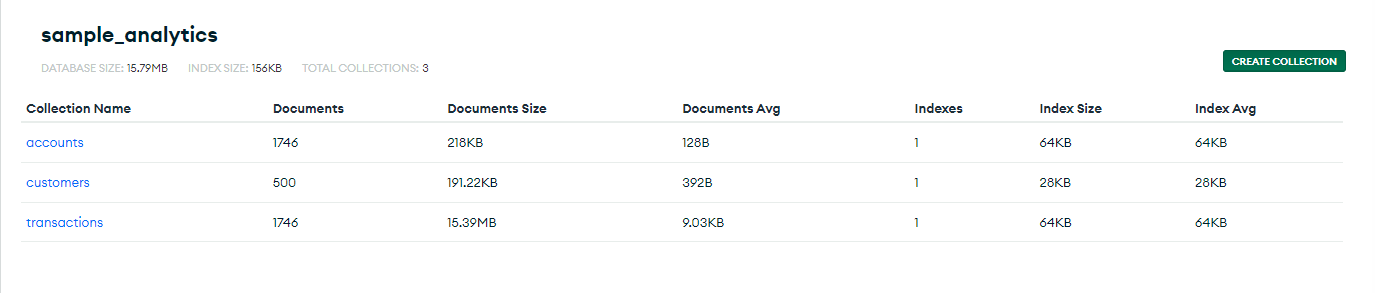
Za ilustrovanje do sada predstavljenih svojstava *MongoDB*-a i načina organizacije podataka korišćen je *colud* pristup. U kontekstu *MongoDB-a* *cluster* je pojam koji se obično koristi za skup replika ili za *sharded cluster.* Skup replika je replikacija grupe *MongoDB* servera koji čuvaju kopije istih podataka, što je fundamentalno svojstvo jer obezbeđuje visoku dostupnost i redundantnost podataka. *Sharded cluster* je takođe često korišćen pristup za horizonlatno skaliranje, gde se podaci distribuiraju kroz veliki broj servera.

Kreiranje baze a i samog *cluster*-a, neophodno je pored kreiranja naloga, odabrati pre svega jedan od tri tipa *cloud*-a: *serverless*, *dedicated* ili *shared. Serverless*, kao što i samo ime kaže za *serverless* aplikacije sa niskom ili promenljivom freknvencijom saobraćaja. *Dedicated* za *production* aplikacije sa softiciranim zahtevima radnog okruženja, a *Shared* je *free* pristup pogodan za učenje i istraživanje i pruža od 512MB do 5GB prostora za skladištenje, kao i deljivi *RAM*.

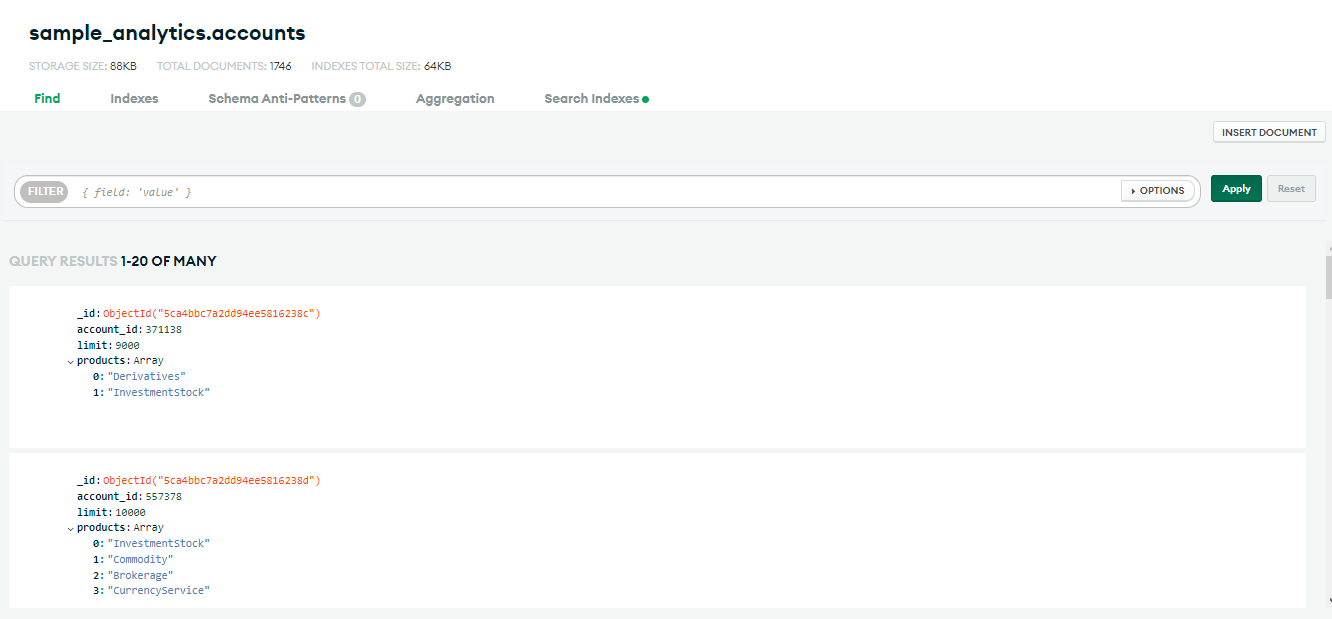
Nakon kreiranja naloga i *cluster*-a inicijalno postoje već kreirani test podaci u vidu baza podataka koje se sastoje od skupa kolekcija dokumenata i grafičko okruženje koje *MongoDB* pruža izgleda ovako:

Slika 27 – Inicijalni izgled *cluster*-a

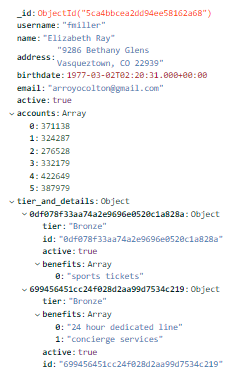
Sledeća test baza podaka *sample\_analytics* sastoji se od 3 kolekcije *accounts, customers* i *transactions*.

  
Slika 28 – *sample\_analytics* baza podataka

Svaka od ovih kolekcija sastoji se od velikog broja dokumenata. Na primer, kolekcija *accounts* prikazana je na sledecoj slici zajedno sa jednostavnom strukturom liste dokumenata od kojih se sastoji. Svaki dokument predstavlja jedan *account* koji od atributa ima pre svega identifikator koi generiše sam *MongoDB*, identifikator samog *account*-a, *limit* i listu proizvoda kao ,,ugrađeni” *embedded* tip podataka.

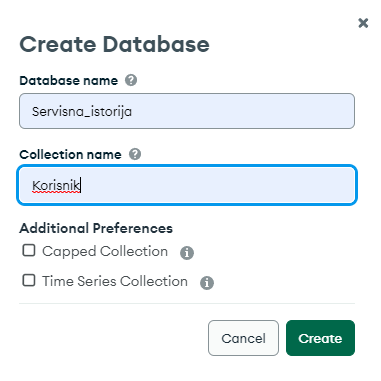
  
Slika 29 – Kolekcija *accounts*

Malo složeniji primer dokumenta imamo u okviru kolekcije *customers*, gde u okviru dokumenta imamo više atributa koji su *embedded* i predstavljaju liste drugih objekata odnosno dokumenata, a i reference na druge objekte kao primer normalizovanih podataka. Ovaj primer prikazan je na sledećim slikama.

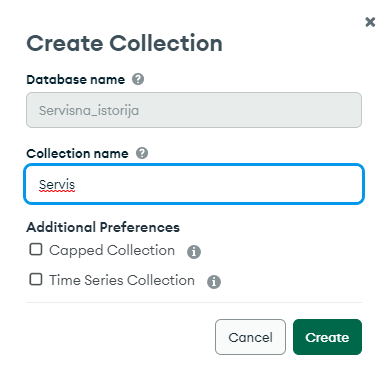
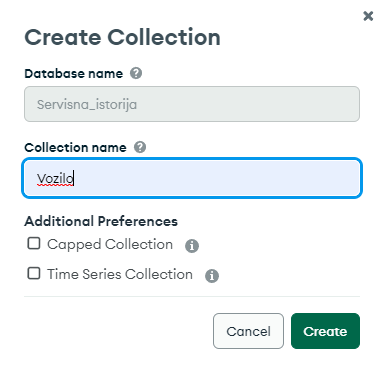
  
Slika 30 – Primer kompleksnih dokumenata

Međutim, kada sami kreiramo model podataka, pre svega moramo uzeti u obzir celokupan koncept aplikacije i slučajeve korišćenja samih podataka, da ne bi došlo do ponavljanja podataka na primer u slučaju *embedded* modela ili neograničenih i promenljivih nizova u slučaju normalizovanih podataka gde se koriste reference.

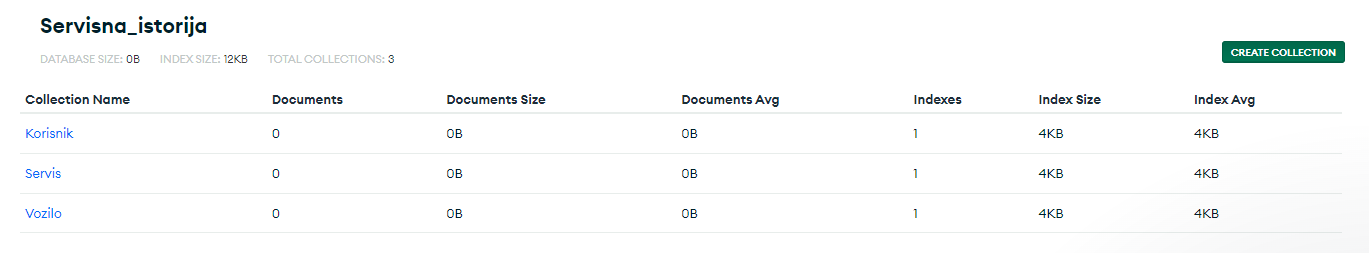
U nastavku, data je ilustracija kreiranja modela podataka, korak po korak. Za početak, da objasnimo koncept baze. Baza podataka predstavlja servisnu istoriju, koja će služiti auto servisu da vodi evidenciju kako o vozilima koja su servisirana tako i o korisnicima koji su vlasnici vozila. U tom slučaju modelovaćemo vezu 1:n kao korisnika koji može imati više vozila, pri čemu ćemo podrazumevati da broj vozila neće biti neograničen, već u realnom slučaju jedno, dva, tri, za svakog korisnika pribavljaće se istovremeno i podaci o samom korisniku i o njegovom vozilu ili više njih, pa ćemo iskoristiti embedded model podataka i u okviru jednog korisnika pamtiti sve podatke o njegovom jednom ili više vozila. Pored entiteta Korisnika i Vozila imaćemo i entitet Servis, uz pomoć kojeg možemo modelovati vezu n:m tako što će jedno vozilo moći da se servisira u više različitih servisa, dok će svaki servis moći da primi više različitih vozila. U te svrhe prvo ćemo kreirati bazu podataka Servisna\_istorija i dodati prvu kolekciju Korisnik, to možemo učiniti na sledeći način, korišćenjem opcije *Create database*:

  
Slika 31 – Kreiranje baze podataka

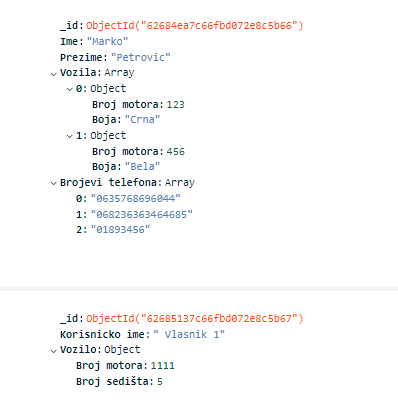
Nakon toga, dodajemo kolekcije Vozilo i Servis na sličan način, korišćenjem *Create collection*:

  
Slika 32 – Kreiranje kolekcija i dodavanje u postojećoj bazi

Sada naša baza podataka sa 3 kolekcije, pre dodavanja dokumenata izgleda ovako:

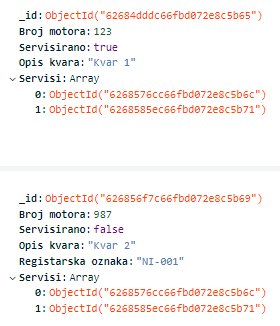
  
Slika 33 – Kreirane kolekcije

Uz pomoć *insert document* možemo kreirati dokument korisnika koji ima jedno vozilo čime ćemo modelirati relaciju 1:1 i korisnika sa više vozila gde ćemo modelirati relaciju 1:n, u oba slučaja koristeći embedded pristup, gde ćemo u okviru samog korisnika čuvati sve informacije o njegovom vozilu, odnosno njegovim vozilima, pri čemu nakon kreiranja ovih dokumenata, kolekcija Korisnik izgleda ovako:

  
Slika 34 – Dokumenti Korisnika sa *embedded* pristupom

U ovom primeru pre svega možemo videti da dva dokumenta u istoj kolekciji, odnosno dva entiteta ne moraju imati nužno sve iste atribute, to nam omogućava fleksibilna šema *MongoDB*-a. Prvi korisnik ima atribute Ime i Prezime kao stringove, viševrednosni atribut Brojevi telefona koja predstavlja listu stringova i listu objekataVozila koja zapravo predstavlja embedded pristup i modelira vezu 1:n. Kada je u pitanju drugi korisnik, imamo atribut Korisničko ime koji je tipa string i jedan objekat vozila koji takodje predstavlja embedded model podataka i modeluje vezu 1:1. Ovaj primer ilustruje to da i ugrađeni dokumenti ne moraju imati istu strukturu. Prvi korisnik ima niz objekata Vozila koji imaju atribute Broj motora tipa integer i atribut Boja tipa string, dok drugi ima objekat jednog vozila i to sa atributima Broj motora i Broj sedišta tipa integer.

U slučaju modelovanja relacija n:m entitet Vozila predstavićemo u vidu zasebne kolekcije pri čemu ćemo u okviru svakog vozila pamtiti reference na Servise u kojima je vozilo servisirano jer može biti servisirano u različitim servisima i isto tako u okviru svakog Servisa pamtićemo reference na Vozila koja su u njemu servisirana. Te reference zapravo predstavljaju listu jedinstvenih identifikatora koji *MongoDB* generiše za svaki dokument. Ovim smo ilustrovali normalizovani pristup modeliranja, pri čemu pored referenci na dokumente, samo dokumenti međusubno i dalje ne moraju biti identični.

  
Slika 35 – Dokumenti Vozila i Servisa sa normalizovanim pristupom

Međutim, ovde se može primeniti i *one way* *embedding* pristup, gde se reference kod m:n relacija čuvaju samo na jednoj strani odnosno samo u jednom dokumentu. To bi bilo pogodno u slučaju da kapacitet servisa nije ograničen i u tom slučaju bismo imali neograničen i promenljiv niz referenci na vozila, gde bi bilo pogodno da se čuvaju samo reference na servise u okviru vozila, a ne i obrnuto jer bi taj broj mogao da bude veoma veliki.

# Zaključak

Sa pojavom *document oriented* baza podataka koje omogućavaju efikasniji rad developerima, većina relacionih baza podataka uvodi dodatnu podršku za rad sa *JSON*-om. Međutim, jednostavno dodavanje *JSON* tipa podataka ne donosi revolucionarne prednosti izvornoj relacionoj bazi podataka. Zašto? Zato što relacioni pristup pre svega umanjuje produktivnost programera, a ne poboljšava je. Pre svega, rad sa dokumentima podrazumeva korišćenje prilagođenih specifičnih *SQL* funkcija koje neće biti poznate većini programera i koje ne rade sa često korišćenim i široko primenjenim *SQL* alatima. Takođe, predstavljanje *JSON* podataka u vidu jednostavnih stringova i brojeva umesto kompleksnih i ugnježdenih tipova podataka usložnjava obradu, upoređivanje i sortiranje podataka i čini ih podložnijim greškama. Pored toga, tradicionalne relacione baze ne nude način za particionisanje baze na više instanci radi skaliranja.

# Literatura

<https://www.mongodb.com>

<https://devopedia.org>

<https://k21academy.com>

<https://www.educba.com>