



UNIVERZITET U NIŠU  
ELEKTRONSKI FAKULTET



Obrada transakcija, planovi izvršavanja transkacija, izolacija i  
zaključavanje kod *Mongo DB* baze podataka

Seminarski rad

Studijski program: Računarstvo i informatika

Modul: Softversko inženjerstvo

Mentor:

Aleksandar Stanimirović

Student:

Petar Mančić

1.	Uvod u transakcije u <i>MongoDB</i> -u .....	4
1.1	Šta su transakcije? .....	4
1.2	Zašto su transakcije bitne? .....	4
1.3	Kratak istorijat podrške u <i>MongoDB</i> .....	4
1.4	Osnovne karakteristike <i>MongoDB</i> transakcija .....	5
1.5	Kratak primer ( <i>Node.js</i> ).....	5
1.6	Tipične greške i napomene .....	6
2.	Obrada transakcija i planovi izvršavanja transakcija .....	6
2.1	Istorijski kontekst i arhitektonski izazovi.....	6
2.2	Životni ciklus transakcije – detaljan prikaz.....	7
2.3	Tipovi transakcija kod <i>MongoDB</i> .....	7
2.3.1	Jednodokumentne transakcije ( <i>Single-document atomic operations</i> ).....	7
2.3.2	Višedokumentne transakcije ( <i>Multi-document ACID transactions</i> ) .....	8
2.3.3	<i>Read/Write concern</i> konfiguracije transakcija (tipovi po garancijama) .....	8
2.3.4	<i>Implicit</i> i <i>Explicit</i> transakcije (po načinu pokretanja) .....	9
2.4	Planovi izvršavanja unutar transakcija – dubinska analiza .....	10
2.5.1	Fiksiranje plana na početku transakcije .....	10
3.	Izolacija u <i>MongoDB</i> .....	10
3.1	Izolacija van transakcija ( <i>default case</i> ) .....	11
3.2	Izolacija unutar transakcija.....	11
3.3	Zaključavanje i izolacija ( <i>Locking model</i> ) .....	12
3.4	<i>Read Concern</i> i njihov uticaj na izolaciju .....	13
4.	Zaključavanje u <i>MongoDB</i> .....	14
4.1	Evolucija zaključavanja u <i>MongoDB</i> .....	14
4.2	Razvoj granualnosti zaključavanja u <i>MongoDB</i> -u.....	14
4.3	Kako funkcioniše zaključavanje kod transakcija od verzije 4.0 nadalje .....	15
4.4	Koje vrste kratkotrajnih zaključavanja ipak postoje u <i>WiredTiger</i> -u .....	15
4.5	Prednosti i mane optimističkog zaključavanja .....	16

4.6 Kada <i>MongoDB</i> ipak koristi pravo (pesimističko) zaključavanje.....	16
5.Praktični primeri.....	17
5.1. Korišćenje <i>MongoDB Atlas</i> -a i Transakcija u <i>MongoDB</i> -u .....	17
5.2.Obrada transakcija.....	19
5.3.PLANOVI IZVRŠENJA TRANSAKCIJA U <i>MongoDB</i> -u.....	26
5.4. Izolacija u <i>MongoDB</i> .....	31

# 1. Uvod u transakcije u *MongoDB*-u

## 1.1 Šta su transakcije?

Transakcija je sekvenca jedne ili više operacija nad bazom podataka koja se izvršava kao jedna neraskidiva celina. Osnovna ideja je da se ili sve operacije u transakciji uspešno izvrše i trajno zabeleže, ili se, u slučaju greške, nijedna od njih ne odrazi na stanje baze — to je princip **atomičnosti**.

Ukoliko transakcija zadovoljava svojstva **ACID**, to znači:

- **Atomicity (Atomičnost)** — govori o tome da se sve naredbe transakcije izvršavaju kao jedinstvena celina ili se uopšte ne izvršavaju.
- **Consistency (Konzistentnost)** — garantuje da se baza podataka nakon transakcije prevodi iz jednog konzistentnog stanja u drugo konzistentno stanje.
- **Isolation (Izolacija)** — govori o tome da efekti transakcije nisu vidljivi u drugim transakcija sve dok se transakcija ne komituje
- **Durability (Trajnost)** — govori o tome da su sve izmene nad podacima nacinjene tokom transakcije trajne.

## 1.2 Zašto su transakcije bitne?

Transakcije su ključne za očuvanje integriteta podataka u scenarijima gde jedna logička operacija menja više zapisa ili kolekcija (npr. transfer novca između naloga, sinhronizacija stanja više dokumenata, ažuriranje indeksa i metapodataka i sl.). Bez transakcija, u slučaju greške u sredini operacije, baza može ostati u polu-izmenjenom i nekonzistentnom stanju.

## 1.3 Kratak istorijat podrške u *MongoDB*

- *Pre MongoDB 4.0*: atomicnost je bila zagaranovana samo po jednom dokumentu (*single-document atomic operations*). Kompleksne *multi-document* operacije morale su da se rešavaju aplikacionom logikom ili pomoću dvostepene protokole (složenije i skloni greškama).
- *MongoDB 4.0* (izdanje): uvedene su ***multi-document transakcije*** za replika setove.

- *MongoDB* 4.2 i novije verzije: podrška za transakcije proširena i na **sharded** klastere. Ovo je značajan prelaz koji omogućava *MongoDB*-u da konkuriše RDBMS-ima u scenarijima koji zahtevaju ACID svojstva preko više dokumenata i kolekcija.

## 1.4 Osnovne karakteristike *MongoDB* transakcija

- **Session-based:** transakcije se pokreću i održavaju u okviru sesije (*logical session id, LSID*). Klijentske biblioteke (*driveri*) obezbeđuju API za *start/commit/abort*.
- **Snapshot reads:** transakcije koriste snapshot view podataka (u okviru iste transakcije vidite konzistentan snapshot stanja u trenutku početka transakcije — ovo je bitno za izolaciju).
- **Kontrola čitanja i pisanja:** kombinacijom *readConcern* i *writeConcern* podešavanja se kontroliše vidljivost i trajnost podataka.
- **Ograničenja performansi:** transakcije su skuplje od single-document operacija — duge transakcije i veliki broj promena unutar transakcije može da utiče na latenciju i resurse (*memory, oplog size, itd.*).

## 1.5 Kratak primer (*Node.js*)

Evo minimalnog primer koda koji ilustruje kako se u *Node.js* (*MongoDB driver*) započne sesija i transakcija, i kako se radi *commit/abort*:

```
async function runTransactionExample() {
  const client = new MongoClient("mongodb://localhost:27017");
  await client.connect();
  const session = client.startSession();

  try {
    session.startTransaction({
      readConcern: { level: "snapshot" },
      writeConcern: { w: "majority" }
    });

    const users = client.db("bank").collection("users");
    const logs = client.db("bank").collection("logs");

    // primer: transfer 100 sa naloga A na nalog B
    await users.updateOne({ _id: "A" }, { $inc: { balance: -100 } }, { session });
    await users.updateOne({ _id: "B" }, { $inc: { balance: 100 } }, { session });
    await logs.insertOne({ action: "transfer", from: "A", to: "B", amount: 100 }, { session });

    await session.commitTransaction();
    console.log("Transaction committed.");
  } catch (err) {
    console.error("Transaction aborted due to error:", err);
    await session.abortTransaction();
  } finally {
    await session.endSession();
    await client.close();
  }
}

runTransactionExample();
```

Slika 1 Započinjanje translakcija

Kratko objasnjenje ovog koda:

- *startTransaction* prima opcije za *readConcern* i *writeConcern*.
- Sve operacije koje pripadaju transakciji prosleđuju se sa { *session* }.
- U slučaju greške poziva se *abortTransaction*.

## 1.6 Tipične greške i napomene

- Ne koristiti transakcije za bulk operacije koje mogu biti podeljene ili obrađene idempotentno — transakcije su pravljenje za logičke jedinice rada.
- Duge transakcije mogu uzrokovati veći rast oplog-a i povećati *contention*.
- Obavezno pozivanje *endSession()* u *finally* bloku da bi se oslobođili resursi.

# 2. Obrada transakcija i planovi izvršavanja transakcija

## 2.1 Istoriski kontekst i arhitektonski izazovi

Do 2018. godine *MongoDB* je bio tipičan predstavnik *BASE* filozofije. Podrška za atomičnost postojala je samo na nivou jednog dokumenta (*single-document ACID*). Sve operacije nad više dokumenata bile su „*best-effort*“ i zahtevale su kompenzacione akcije na aplikativnom nivou.

Uvođenje višedokumentnih transakcija zahtevalo je sledeće fundamentalne izmene:

- Prebacivanje podrazumevanog *storage engine-a* na *WiredTiger* (od 3.2 obavezno za transakcije)
- Implementaciju distribuiranog *clock-a* za *snapshot* koordinaciju između čvorova
- Proširenje oplog formata (*v2 oplog*) za beleženje transakcionih metapodataka
- Dodavanje *cluster-wide transaction coordinator-a* u *mongod* i *mongos* procesima

## 2.2 Životni ciklus transakcije – detaljan prikaz

1. **Client → Driver → `startSession()`** → vraća `sessionId + serverSession`
2. **`startTransaction(readConcern, writeConcern)`** → kreira `TransactionRecord` u `config.transactions` kolekciji (na `config` serveru kod `sharded` klastera)
3. **Prva operacija u transakciji** → dodeljuje se `txnNumber + snapshot timestamp (clusterTime)`
4. **Svaka naredna operacija** → proverava da li je ista sesija i isti `txnNumber`
5. **`commitTransaction()`** → dvofazni `commit`:  
Faza 1: *prepare* (samo kod `sharded` klastera od 4.2+)  
Faza 2: stvarni *commit* (označava oplog unose kao *committed*)
6. **`abortTransaction()`** → označava sve promene kao *aborted*, *WiredTiger* ih odbacuje

## 2.3. Tipovi transakcija kod *MongoDB*

### 2.3.1. Jednodokumentne transakcije (*Single-document atomic operations*)

*MongoDB* je od početka (od prve verzije) garantovao **atomičnost na nivou jednog dokumenta**.

- Ne tretira se formalno kao “transakcija”, ali ***MongoDB interni engine (WiredTiger)*** ih smatra transakcijama na nivou jednog dokumenta.
- Svaka **jedna operacija** (*insert/update/delete*) nad jednim dokumentom je **atomska**.
- Koristi se implicitno, bez session i bez `startTransaction()`.

#### Kada se koriste?

U 90% *CRUD* operacija – zato se transakcije retko koriste u realnim sistemima.

### 2.3.2. Višedokumentne transakcije (*Multi-document ACID transactions*)

Uvedene u *MongoDB 4.0 (Replica Set)*, a u 4.2 i na *Sharded Cluster*-ima.

- Pokreću se eksplisitno:
- `session.startTransaction()`
- Garantuje se:
  - **Atomičnost** nad više kolekcija i dokumenata
  - **Snapshot Isolation (pravim MVCC)**
  - **Reads-your-own-writes**
  - Serijalizacija konflikata preko *write lock*-ova na dokumentima

**Podtipovi koje razlikujemo:**

#### 1. Transakcije nad *Replica Set*-om

- Najbrže
- Najstabilnije
- Bez mrežnih prelaza između *shard*-ova

#### 2. Transakcije nad *Sharded Cluster*-om

- Najkompleksnije
- Koordinacija preko mongos
- Koristi internu dvostepenu *commit* strategiju

### 2.3.3. *Read/Write concern* konfiguracije transakcija (tipovi po garancijama)

Tip transakcije se takođe može razlikovati i po garanciji konzistentnosti:

#### 1. READ CONCERN:

*MongoDB* transakcije mogu biti:

- **local** – najbrže, nema garancija replikacije
- **majority** – najčešći nivo, čita potvrđene podatke
- **snapshot** – koristi *MVCC*, standard za transakcije

- ***linearizable*** (nedozvoljen u transakcijama)

U transakcijama je najvažniji ***snapshot*** jer daje izolaciju.

## 2. WRITE CONCERN:

- *w:1* – prihvata samo od primarnog
- *w:majority* – najbezbedniji, čeka da većina replika potvrdi
- *w:all* – vrlo retko, skoro nikad se ne koristi

### 2.3.4. *Implicit* i *Explicit* transakcije (po načinu pokretanja)

Na osnovu toga da li se transakcije izvrsavaju implicitno ili ih programmer pokreće eksplisitno postoje 2 tipa transakcija:

#### 1. Implicitne transakcije

- Jednostavne operacije nad jednim dokumentom
- Atomskost je garantovana automatski
- Klijent ne koristi *session*

#### 2. Eksplisitne transakcije

- Programer ih ručno pokreće preko:

```
const session = client.startSession();  
session.startTransaction();
```

- Koriste *read/write concern*
- Imaju *retry* logiku (*transient errors*)

## 2.4. Planovi izvršavanja unutar transakcija – dubinska analiza

### 2.5.1 Fiksiranje plana na početku transakcije

Kada se izvrši prva operacija u transakciji, *query planner* bira plan i **kešira ga za celu transakciju**. To znači:

- Nema plan *re-caching* čak i ako se kardinalitet drastično promeni tokom transakcije
- Može dovesti do lošeg plana ako se podaci značajno promene

Primer:

```
session.startTransaction();  
db.orders.insertOne({ ... });  
for(let i=0; i<1000000; i++) db.orders.insertOne({ ... });
```

## 3. Izolacija u *MongoDB*

Izolacija u *MongoDB*-u definiše kako paralelne operacije čitanja i pisanja međusobno utiču jedna na drugu i koliko su „odvojene“ (izolovane) u toku izvršavanja. Za razliku od tradicionalnih relationalnih baza koje primenjuju različite izolacione nivoje (*READ UNCOMMITTED*, *READ COMMITTED*, *REPEATABLE READ*, *SERIALIZABLE*), *MongoDB* koristi **sopstveni model izolacije**, zasnovan na:

- **atomskim operacijama nad jednim dokumentom**
- **MVCC mehanizmu (*Multi-Version Concurrency Control*)**
- ***snapshot read* modelu u transakcijama**
- **dokument-nivou zaključavanja (*document-level locking*)**
- ***Write Conflict Detection* sistemu**

*MongoDB* ne implementira direktno *SQL* izolacione nivoje, već obezbeđuje ponašanje koje je najbliže kombinaciji ***Snapshot Isolation*** i ***Read Committed***, u zavisnosti od toga da li koristiš transakcije ili ne.

### 3.1. Izolacija van transakcija (*default case*)

Van eksplisitnih transakcija (u regularnim *CRUD* operacijama), *MongoDB* koristi:

#### 1. Atomicnost i izolacija na nivou jednog dokumenta

- Svaka pojedinačna operacija nad jednim dokumentom je **atomska**.
- Čitaoci nikada ne vide delimične promene dokumenta — ili vide stari dokument ili novi.

#### 2. *Read Committed* semantika

*MongoDB* van transakcija omogućava:

- čitanje samo **potvrđenih** (*committed*) podataka
- čitanje podataka koji su vidljivi na primarnom čvoru u trenutku čitanja
- ne garantuje da će sledeće čitanje vratiti isti *snapshot* — mogu se videti nove promene

Dakle, izolacija je na nivou ***read committed***, ali bez klasičnih *SQL* "dirty read" ili "non-repeatable read" problema unutar jedne operacije.

### 3.2. Izolacija unutar transakcija

U eksplisitnim višedokumentnim transakcijama *MongoDB* pruža ***Snapshot Isolation*** korišćenjem *MVCC*-a.

#### 1. *Snapshot Isolation*

Kada transakcija počne:

- dobija **konzistentan snapshot baze**

- sve *READ* operacije tokom transakcije vide **isti snapshot**, bez obzira na kasnija pisanja drugih sesija
- istovremeno, transakcija vidi svoja sopstvena pisanja (*read-your-own-writes*)

*MongoDB* interno koristi **WiredTiger MVCC**, koji čuva više verzija dokumenata.

### Šta *snapshot isolation* sprečava?

- **Dirty reads** – nemoguće
- **Non-repeatable reads** – nemoguće
- **Phantom reads** – sprečeni u većini slučajeva, jer snapshot ne menja vidljiv skup dokumenata (iako *MongoDB* formalno ne garantuje *SERIALIZABLE*)

### Šta *snapshot isolation* ne pruža?

- **Serializable izolaciju** — *MongoDB* ne garantuje striktno serijalizabilno ponašanje
- moguće su **write-write kolizije**, ali se automatski detektuju

## 3.3. Zaključavanje i izolacija (*Locking model*)

*MongoDB* koristi **fine-grained locking**, najčešće na nivou dokumenta:

### 1. *Document-level locking*

- pisanje nad jednim dokumentom blokira samo taj dokument
  - drugi dokumenti u kolekciji ostaju dostupni
- Ovo je glavna razlika u odnosu na *SQL* baze koje često koriste *page-level* ili *table-level lock-ove*.

### 2. *Write Conflict Detection*

Ako dve transakcije pokušaju da izmene isti dokument:

- *MongoDB* detektuje konflikt
- druga transakcija se prekida sa greškom *WriteConflict*
- klijent treba da ponovi transakciju (*retry logic*)
- Ovo je karakteristika *MVCC* sistema.

### 3.4. *Read Concern* i njihov uticaj na izolaciju

U transakcijama je izolacija direktno određena ***read concern*** vrednostima.

#### 1. ***readConcern: "snapshot"***

- podrazumevani i najčešće korišćen u transakcijama
- garantuje *snapshot isolation*
- izolacija čitanja je stabilna tokom celoj transakcije

#### 2. ***readConcern: "local"* (van transakcija)**

- najbrži
- nema garancija replikacije
- izolacija = *read committed*

#### 3. ***readConcern: "majority"***

- čita samo podatke potvrđene od većine replika
- jače garancije konzistentnosti, ali nije isto što i *snapshot*

Zaključak u vezi izolacije u *MongoDB*:

Izolacija u *MongoDB* zasniva se na atomskim operacijama nad jednim dokumentom, MVCC mehanizmu i snapshot čitanju u transakcijama. Van transakcija, *MongoDB* pruža izolaciju sličnu nivou ***Read Committed***, dok u višedokumentnim transakcijama postiže ***Snapshot Isolation***, sprečavajući pojave kao što su *dirty* i *non-repeatable reads*. Korišćenjem zaključavanja na nivou dokumenta i detekcije konflikata, *MongoDB* omogućava efikasnu paralelnu obradu bez teških globalnih lockova. Iako ne obezbeđuje Serializable nivo izolacije, *MongoDB* postiže dobru ravnotežu između konzistentnosti, performansi i skalabilnosti, što ga čini pogodnim za moderne distribuirane sisteme.

## 4.Zaključavanje u *MongoDB*

Zaključavanje (*locking*) u *MongoDB* predstavlja mehanizam koji kontroliše pristup deljenim resursima kako bi se obezbedila konzistentnost podataka tokom konkurentnih operacija. Za razliku od tradicionalnih relacionih baza koje se oslanjaju na tabelarna ili stranica-nivo (*page-level*) zaključavanja, *MongoDB* koristi **fino-granulisan model zaključavanja**, čime postiže visok stepen paralelizma i minimalno blokiranje procesa.

### 4.1. Evolucija zaključavanja u *MongoDB*

Šta zapravo znači zaključavanje u *MongoDB*?

U klasičnim relacionim bazama podataka (npr. *PostgreSQL*, *Oracle*) zaključavanje je pesimističko – transakcija prvo „uzme katanac“ na redove ili tabele i drži ga dok ne završi. Ako druga transakcija hoće iste podatke, mora da čeka. To sprečava konflikte, ali može dovesti do čekanja i mrtvih petlji (*deadlock*).

*MongoDB* od verzije 3.2 (2016) koristi potpuno drugačiji pristup – **nema klasičnih zaključavanja koje korisnik vidi**. Umesto toga koristi **optimističku kontrolu konkurentnosti** preko *storage engine-a WiredTiger*.

### 4.2 Razvoj granualnosti zaključavanja u *MongoDB*-u

Verzija	Nivo zaključavanja	Objašnjenje
do 2.6	GLOBALNI ZAKLJUCAVANE NA CEO SERVER	JEDAN WRITER U JEDNOM TRENUTKU
3.0 -3.2	ZAKLJUCAVANJE PO KOLEKCIJI	VIŠE WRITER-A AKO RADE NA RAZLIČITIM KOLEKCIJAMA
OD 3.2	ZAKLJUCAVANJE PO DOKUMENTU	TRENUTNI STANDARD

Danas je granularnost na nivou pojedinacnog dokumenta, što je najsitnija moguća jedinica.

## 4.3 Kako funkcioniše zaključavanje kod transakcija od verzije 4.0 nadalje

Kada dve ili više transakcija pokušaju istovremeno da menjaju isti dokument, *MongoDB* **ne blokira** nijednu od njih. One rade paralelno, a sukob se otkriva tek u trenutku *commit-a*.

Mehanizam rada:

1. Svaka transakcija radi na svom snimku (*snapshot*) podataka.
2. Kada transakcija hoće da upiše izmene (*commit*), *WiredTiger* proverava da li je dokument u međuvremenu promenjen od strane druge transakcije.
3. Ako jeste – dolazi do greške ***WriteConflict*** (kod 112).
4. Klijentski driver automatski pokreće celu transakciju iz početka (*retry*).
5. Prva transakcija koja uspešno izvrši *commit* – pobeđuje (*first committer wins*).

Zbog toga ne postoje mrtve petlje (*deadlock*) – uvek se samo jedna transakcija „žrtvuje“ i ponavlja.

## 4.4. Koje vrste kratkotrajnih zaključavanja ipak postoje u *WiredTiger-u*

Iako korisnik ne vidi klasično zaključavanje, na najnižem nivou (unutar *storage engine-a*) postoje veoma brzi mehanizmi zaštite:

- ***Latches*** – kratkotrajni spinlock-ovi koji štite strukturu B-stabla (traju mikrosekunde)
- ***Ticket sistem*** – ograničava broj istovremenih upisivača da ne bi preopteretili keš
- ***History store*** – čuva stare verzije dokumenata dok su potrebne aktivnim snapshot-ovima

Ovi mehanizmi su nevidljivi za korisnika i ne izazivaju dugo čekanje.

## 4.5 Prednosti i mane optimističkog zaključavanja

### Prednosti

- Odlične performanse kada je mala verovatnoća sukoba (većina realnih aplikacija)
- Čitanje nikad ne čeka pisanje i obrnuto
- Nema *deadlock*-ova
- Veliki broj paralelnih operacija

### Mane

- Kod velikog broja sukoba (npr. svi klijenti menjaju isti dokument) dolazi do mnogo ponavljanja → pad performansi
- Aplikacija mora biti spremna na automatsko ponavljanje transakcija (*driver to radi sam*, ali ipak troši procesor)

## 4.6 Kada *MongoDB* ipak koristi pravo (pesimističko) zaključavanje

Postoje retki administrativni poslovi gde se i dalje koriste klasični katanci:

- Kreiranje indeksa u pozadini (do verzije 4.2)
- Brisanje cele baze ili kolekcije
- *Initial sync replica* seta

Ovi poslovi su blokirajući, ali se ne odnose na obične korisničke transakcije.

### Zaključak:

*MongoDB* je svesno odustao od tradicionalnog pesimističkog zaključavanja u korist optimističkog modela koji je zasnovan na snimcima i automatskom ponavljanju transakcija u slučaju sukoba. Ovakav pristup omogućava odlično skaliranje u većini realnih situacija, ali zahteva da programer bude svestan mogućnosti *WriteConflict* grešaka kod aplikacija sa velikim brojem konkurentnih upisa na iste dokumente.

## 5. Praktični primeri

### 5.1. Korišćenje MongoDB Atlas-a i Transakcija u MongoDB-u

*MongoDB Atlas* je *cloud* platforma koja obezbeđuje jednostavno kreiranje i upravljanje *MongoDB* bazama podataka. Prednost *Atlasa* je što korisniku omogućava da veoma brzo dođe do funkcionalne baze, bez potrebe za lokalnom instalacijom, konfiguracijom servera ili održavanjem infrastrukture.

#### Kreiranje baze i povezivanje na *Atlas*

Nakon kreiranja naloga na *MongoDB Atlas* platformi, potrebno je:

1. Kreirati *cluster*

The screenshot shows the MongoDB Atlas interface. On the left is a sidebar with sections: Project Overview, DATABASE (Clusters, Search & Vector Search, Backup), STREAMING DATA (Stream Processing, Triggers), SERVICES (Migration, Data Federation, Visualization), and SECURITY. The main area is titled 'Clusters' and shows a single cluster named 'Cluster0'. It includes tabs for 'Connect', 'View Monitoring', 'Browse Collections', and '...'. Below this is a section titled 'Visualize Your Data' with a note: 'Build dashboards and charts, and embed them in your apps with MongoDB Charts.' It displays metrics: Replicas (R: 0, W: 0, Last 6 hours: 0.1/s), Connections (Last 6 hours: 24.0), Network (In: 0.00 B/s, Out: 0.00 B/s, Last 6 hours: 1.22 KB/s), and Data Size (116.59 MB / 512.00 MB (23%), Last 6 days: 512.00 MB). A red arrow points to the green '+ Create' button in the top right corner of the main cluster area.

2. Definisati korisnika baze i postaviti lozinku.

### 3. U sekciji Network Access dodati IP



A screenshot of the MongoDB Atlas Project Overview interface. The left sidebar shows navigation categories: Project Overview, DATABASE (Clusters selected), STREAMING DATA, SERVICES, SECURITY, and Database & Network Access (highlighted with a red arrow). The main content area is titled "IP Access List". It displays a table of IP addresses with columns: IP Address, Comment, Status, and Actions (Edit and Delete buttons). A note at the top states: "You will only be able to connect to your cluster from the following list of IP Addresses:". The table contains three entries: 109.92.110.34/32, 80.93.252.50/32, and 178.220.63.167/32 (includes your current IP address). A red box highlights the "+ADD IP ADDRESS" button in the top right corner.

IP Address	Comment	Status	Actions
109.92.110.34/32	Created as part of the Auto Setup process	Active	<button>Edit</button> <button>Delete</button>
80.93.252.50/32		Active	<button>Edit</button> <button>Delete</button>
178.220.63.167/32 (includes your current IP address)		Active	<button>Edit</button> <button>Delete</button>

### 4. Kopirati connection string i pomoću njega povezati se u Mongo Compass:

```
mongodb+srv://petarmancic_db_user:KfGrWcWPgdbOy36rx@cluster0.6yuwubq.mongodb.net/?appName=Cluster0
```

Sada imamo sve što nam je potrebno da bismo mogli da radimo sa transakcijama.

## 5.2.Obrada transakcija

Najpre ćemo kreirati par korisnika koji će nam omogućiti rad sa transakcijama:

```
db.accounts.insertMany([
  { name: "Alice", balance: 1000 },
  { name: "Bob", balance: 1000 },
  { name: "Charlie", balance: 1000 },
  { name: "Diana", balance: 1000 },
  { name: "Ethan", balance: 1000 }
]);
```

Slika 2 Insertovanje 5 korisnika u bazi

Ovim unosom formiramo pet korisnika sa početnim stanjem od 1000 jedinica (neka to budu dinari).

Što se tiče transakcija u *MongoDB*-u, one nam omogućavaju da se više operacija izvrši kao jedna celina (*ACID* principi).

To bi u prevodu značilo:

- Sve operacije unutar transakcije će biti izvršene
- Nijedna operacija neće biti izvršena

To je veoma važno u finansijskim aplikacijama, posebno kod transfera novca.

### Modeliranje transfera novca u *MongoDB*-u pomoću transakcija

Transfer novca sa jednog računa na drugi predstavlja tipični primer operacije koja mora biti **atomska**, tj. ne sme biti delimično izvršena. Transfer se sastoji iz dve osnovne operacije:

**1. Skidanje određenog iznosa sa računa pošiljaoca**

**2. Uplata tog iznosa na račun primaoca**

Ako se obe operacije ne izvrše uspešno, stanje u sistemu bi bilo nedosledno. Na primer, novac bi mogao biti skinut sa jednog računa, a da nikada ne bude uplaćen na drugi. Zbog toga se ove dve operacije obavezno izvršavaju **u okviru jedne *MongoDB* transakcije**.

```

function transferMoney(from, to, amount) {
    const session = db.getMongo().startSession();
    const accounts = session.getDatabase("bank").accounts;

    session.startTransaction();

    try {
        print(`Transferring ${amount} from ${from} to ${to}`);

        // 1. Skidanje sa 'from'
        const res1 = accounts.updateOne(
            { name: from, balance: { $gte: amount } },
            { $inc: { balance: -amount } }
        );

        if (res1.matchedCount === 0) {
            throw new Error(`Sender ${from} does not have enough funds.`);
        }

        // 2. Uplata na 'to'
        const res2 = accounts.updateOne(
            { name: to },
            { $inc: { balance: amount } }
        );

        if (res2.matchedCount === 0) {
            throw new Error(`Receiver ${to} not found.`);
        }

        session.commitTransaction();
        print("Transfer completed successfully.");
    } catch (e) {
        print("Transfer FAILED: " + e);
        session.abortTransaction();
    } finally {
        session.endSession();
    }
}

```

*Slika 3 Prenos novca sa jednog računa na drugi*

Ovo je kod koji simulira prenos novca sa jednog računa na drugi račun.

**Transakcija će izgledati ovako:**

*Alica-> Bob (300)*

*Bob->Charlie (500)*

*Charlie-> Ethan (200)*

Na početku svako na računu ima istu količinu novca (1000)

Očekivano ponašanje:

**Nakon izvršenja prve transkacije:**

*Alice: 700*

*Bob: Priliv 300-> 1300, odliv 500 -> 800*

*Charlie: Priliv 500-> 1500, odliv 200-> 1300*

**Uspešna transakcija**

```
transferMoney("Alice", "Bob", 300);
transferMoney("Bob", "Charlie", 500);
transferMoney("Charlie", "Ethan", 200);
< Transferring 300 from Alice to Bob
< Transfer completed successfully.
< Transferring 500 from Bob to Charlie
< Transfer completed successfully.
< Transferring 200 from Charlie to Ethan
< Transfer completed successfully.
```

Slika 4 Prva transkacija uspešno izvršena

```
_id: ObjectId('693099c94a0e8bb931c4045a')
name : "Alice"
balance : 700
```

```
_id: ObjectId('693099c94a0e8bb931c4045b')
name : "Bob"
balance : 800
```

```
_id: ObjectId('693099c94a0e8bb931c4045c')
name : "Charlie"
balance : 1300
```

```
_id: ObjectId('693099c94a0e8bb931c4045e')
name : "Ethan"
balance : 1200
```

Slika 5 Stanje novca korisnika nakon uspešno izvršene transakcije

#### Druga transakcija:

Alice: 400

Bob: Priliv 300-> 1100, odliv 500 -> 600

Charlie: Priliv 500-> 1800, odliv 200-> 1600

#### Uspesna transakcija

```
< Transferring 300 from Alice to Bob
< Transfer completed successfully.
< Transferring 500 from Bob to Charlie
< Transfer completed successfully.
< Transferring 200 from Charlie to Ethan
< Transfer completed successfully.
```

Slika 6 Druga transkacija uspešno izvršena

```
_id: ObjectId('693099c94a0e8bb931c4045a')
name : "Alice"
balance : 400
```

```
_id: ObjectId('693099c94a0e8bb931c4045b')
name : "Bob"
balance : 600
```

```
_id: ObjectId('693099c94a0e8bb931c4045c')
name : "Charlie"
balance : 1600
```

```
_id: ObjectId('693099c94a0e8bb931c4045e')
name : "Ethan"
balance : 1400
```

Slika 7 Stanje novca korisnika

U ovoj transkaciji čemo pokušati da sa *Charlie-vog* računa prebacimo više novca nego što on ima. Tu očekujemo da transakcija bude prekinuta i da se desi *rollback* odnosno da stanje na računima korisnika ostane nepromenjeno.

```
transferMoney("Alice", "Bob", 300);
transferMoney("Bob", "Charlie", 500);

< Transferring 300 from Alice to Bob
< Transfer completed successfully.
< Transferring 500 from Bob to Charlie
< Transfer completed successfully.
```

Slika 8 Uspešna transkacija

Nakon što je *Bob* prebacio novac *Charlie*-u, stanje korisnika je:

```
_id: ObjectId('693099c94a0e8bb931c4045a')
name : "Alice"
balance : 100
```

```
_id: ObjectId('693099c94a0e8bb931c4045b')
name : "Bob"
balance : 400
```

```
_id: ObjectId('693099c94a0e8bb931c4045c')
name : "Charlie"
balance : 2100
```

```
_id: ObjectId('693099c94a0e8bb931c4045e')
name : "Ethan"
balance : 1400
```

Slika 9 Stanje korisnika

Sada ćemo pokušati da sa *Charlie*-vog računa prebacimo 3000 na *Ethan*-ovom računu.

```
> transferMoney("Charlie", "Ethan", 3000);
< Transferring 3000 from Charlie to Ethan
< Transfer FAILED: Error: Sender Charlie does not have enough funds.
```

Slika 10 Neuspešna transkacija

```
_id: ObjectId('693099c94a0e8bb931c4045a')
name : "Alice"
balance : 100
```

```
_id: ObjectId('693099c94a0e8bb931c4045b')
name : "Bob"
balance : 400
```

```
_id: ObjectId('693099c94a0e8bb931c4045c')
name : "Charlie"
balance : 2100
```

```
_id: ObjectId('693099c94a0e8bb931c4045e')
name : "Ethan"
balance : 1400
```

Slika 11 Stanje na računima nakon neuspešne transakcije

### Zaključak

Transakcije u *MongoDB*-u obezbeđuju da se više povezanih operacija, poput skidanja i uplate novca, izvrše kao jedna nedeljiva celina, čime se garantuje doslednost i tačnost podataka. U prikazanom primeru one sprečavaju delimične izmene i omogućavaju da finansijski transfer bude uvek ispravno izvršen ili potpuno poništen.

## 5.3.Planovi izvršenja transakcija u *MongoDB*-u

### 1. Šta je *execution plan*?

*Execution plan* (plan izvršenja) pokazuje **kako MongoDB odlučuje da izvrši neki upit**, uključujući:

- koji indeks koristi
- koliko dokumenata skenira
- da li radi **COLLSCAN** (skenira celu kolekciju)
- da li radi **IXSCAN** (koristi indeks)
- koliko je upit koštao (*execution time, docsExamined, keysExamined...*)

### 2. Kako transakcija utiče na *execution plan*?

**U transakcijama plan važi za svaki upit pojedinačno**

*MongoDB* nema neki globalni “plan transakcije” koji primenjuje nad svakom transakcijom — pravi plan za **svaki upit unutar transakcije**.

**Transakcija dodeljuje upitu:**

- **snapshot** podataka (vidi konzistentan pogled)
- **writeConcern: majority**
- koristi **optimistic concurrency**

Plan izvršenja **nije različit**, ali može biti:

- sporiji (zbog izolacije)
- abortovan ako dođe do *write* konflikta

### 3. Kako u praksi pogledati *execution plan* u transakciji?

Ne možeš direktno pozvati `.explain()` unutar `withTransaction()`, ali možeš uraditi najvažnije: izvršiti isti upit van transakcije i prikazati njegov plan opisati da se isti plan koristi i u transakciji (samo radi pod *snapshot*-om)

**To je zvanično preporučeni način** i koristi se u *MongoDB* dokumentaciji.

Primer:

#### **COLLSCAN u transakciji (bez indeksa)**

1. Najpre kreiramo kolekciju i ubacimo podatke

```
> db.accounts.insertMany([
  { name: "Pera", balance: 100 },
  { name: "Mika", balance: 100 },
  { name: "Zika", balance: 100 }
]);
< {
  acknowledged: true,
  insertedIds: {
    '0': ObjectId('6931e5734a0e8bb931c40465'),
    '1': ObjectId('6931e5734a0e8bb931c40466'),
    '2': ObjectId('6931e5734a0e8bb931c40467')
  }
}
```

Slika 12 Kreiranje kolekcije I ubacivanje podataka

2. Pogledamo plan izvršenja (van transakcije):

```
> db.accounts.explain("executionStats").find({  
    balance: { $gt: 50 }  
})
```

Slika 13 Provera plana izvršenja

Ono što je bitno ovde naglasiti jeste da nije korišćen indeks.

Response izvršenog query-ja je ogroman a ono što je bitno izdvojiti je sledeće:

```
winningPlan: {  
    isCached: false,  
    stage: 'COLLSCAN',  
    filter: {  
        balance: {  
            '$gt': 50  
        }  
    },
```

Slika 14 Response izvršenog query-ja

Ova slika nam pokazuje **stage: collscan** – ovaj podatak nam kaže da nije korišćen nikakav indeks i da su svi dokumenti morali biti pregledani.

```
executionStats: {
  executionSuccess: true,
  nReturned: 10,
  executionTimeMillis: 0,
  totalKeysExamined: 0,
  totalDocsExamined: 10,
  executionStages: {
    isCached: false,
    stage: 'COLLSCAN',
    filter: {
      balance: {
        '$gt': 50
      }
    },
  },
}
```



Slika 15 Broj skeniranja tokom izvršenja query-ja

Na ovoj slici možemo uočiti koliko dokumenata su bila skenirana tokom izvršenja ovog *query*-ja (odnosno morali su biti skenirani svi dokumenti u kolekciji).

#### Kako bi izgledao plan izvršenja ukoliko se koristi index?

```
db.accounts.createIndex({ balance: 1 })
```

Nakon izvršenja *query*-ja sada imamo drugaciji *response*:

```
        winningPlan: {
            isCached: false,
            stage: 'FETCH',
            inputStage: {
                stage: 'IXSCAN',
                keyPattern: {
                    balance: 1
                },
                indexName: 'balance_1',
                isMultiKey: false,
                multiKeyPaths: {
                    balance: []
                },
            }
        }
    }
```

Slika 16 Izvršenje query-ja sa korišćenjem indeksa

Na ovoj slici možemo videti da je koriščen IXSCAN, takodje je koriščen index “**balance\_1**”.

```
executionStats: {
    executionSuccess: true,
    nReturned: 3,
    executionTimeMillis: 0,
    totalKeysExamined: 3,
    totalDocsExamined: 3,
    executionStages: {
        isCached: false
    }
}
```

Slika 17 Nastavak izvršenja query-ja korišćenjem indeksa

Sada kada imamo index vidimo da nismo morali da obidjemo sve dokumente u kolekciji vec samo 3.

Iz prikazanog execution plana jasno se vidi da *MongoDB*, u odsustvu indeksa, koristi **COLLSCAN** — potpuno skeniranje kolekcije. To znači da je za potrebe upita morao da pregleda **sve dokumente**, što potvrđuju metričke vrednosti poput *totalDocsExamined = 10* i *totalKeysExamined*

= 0. Iako se na maloj kolekciji ovakav upit izvršava praktično trenutno, na velikim dataset-ovima ovaj pristup postaje izrazito neefikasan i može značajno usporiti bazu.

Kreiranje indeksnog polja drastično menja plan izvršenja — umesto *COLLSCAN* dobija se *IXSCAN*, čime se drastično smanjuje broj pregledanih dokumenata, poboljšava vreme izvršavanja i optimizuje opterećenje nad sistemom.

#### **Zaključak:**

*Execution plan* direktno pokazuje kvalitet i efikasnost upita. *COLLSCAN* ukazuje na potrebu za optimizacijom, dok *IXSCAN* potvrđuje da indeks omogućava mnogo brže i skalabilnije izvršavanje u *MongoDB*-u.

## 5.4. Izolacija u *MongoDB*

### **Kratko podsećanje: *MongoDB* izolacija**

*MongoDB* u transakcijama koristi:

#### ***Snapshot Isolation (SI)***

- transakcija vidi konzistentan *snapshot* baze u trenutku starta
- ne vidi tuđe promene dok se ne commit-uju
- nema *dirty reads*
- nema *non-repeatable reads*
- ima *phantom reads* samo u nekim slučajevima izvan transakcija

Primer – *Snapshot* izolacija: transakcija ne vidi tuđe promene

Cilj ovog primera jeste da pokažemo da transakcija vidi “zamrznuto” stanje čak i ako druge sesije menjaju podatke.

```
_id: ObjectId('6931e5414a0e8bb931c40462')
name : "Pera"
balance : 100
```

Slika 18 Početno stanje na računu

```
> sessionA = db.getMongo().startSession()
  a = sessionA.getDatabase("bank")

  sessionA.startTransaction()

  a.accounts.find({ name: "Pera" })
< {
    _id: ObjectId('6931e5414a0e8bb931c40462'),
    name: 'Pera',
    balance: 100
}
```

Slika 19 Sesija A- start transakcije

```
> db.accounts.updateOne(
  { name: "Pera" },
  { $set: { balance: 999 } }
)
< {
  acknowledged: true,
  insertedId: null,
  matchedCount: 1,
  modifiedCount: 1,
  upsertedCount: 0
}
```



Slika 20 Sesija B- menja podatke van transakcije

Proveravamo podatke direktno u bazi: Očekujemo da je stanje na računu sada 999 zato što smo ga update-ovali iz Sesije B.

```
_id: ObjectId('6931e5414a0e8bb931c40462')
name : "Pera"
balance : 999
```

Slika 21 Uspešno updatovano stanje

Međutim ukoliko I dalje potražimo podatke iz SesijeA, trebali bismo da dobijemo da je stanje na računu i dalje nepromenjeno.

```
> a.accounts.find({ name: "Pera" })
< [
  {
    _id: ObjectId('6931e5414a0e8bb931c40462'),
    name: 'Pera',
    balance: 100
  }
]
```

Slika 22 Stanje na računu je i dalje nepromenjeno

**Objašnjenje:** *Snapshot isolation* — transakcija vidi stanje baze **u trenutku starta**, ne vidi promene iz drugih sesija.

### Primer- slabost *SI*-a

Jos jedan zanimljiv primer koji nam prikazuje slabost *Snapshot isolation*-a

U bolnici moraju biti **2 doktora** dežurna. Svaki vidi snapshot gde je drugi još uvek dežuran. Ukoliko se desi situacija da se odjave, broj doktora padne na 0.

Insertujemo dva doktora u bazi koji su oba na dužnosti.

```
_id: ObjectId('69320ea863a19d7dc6847b78')
doctor : "A"
onDuty : true
```

```
_id: ObjectId('69320ea863a19d7dc6847b79')
doctor : "B"
onDuty : true
```

Slika 23 Insertovanje doktora u bazi

```
> s1 = db.getMongo().startSession()
  d1 = s1.getDatabase("Hospital")

  s1.startTransaction()

  d1.doctors.find({ onDuty: true })
```

Slika 24 Sesija A- doktor A gleda stanje

Kao rezultat vidimo 2 doktora:

```
< {
  _id: ObjectId('69320ea863a19d7dc6847b78'),
  doctor: 'A',
  onDuty: true
}
{
  _id: ObjectId('69320ea863a19d7dc6847b79'),
  doctor: 'B',
  onDuty: true
}
```

Slika 25 Doktor A kao rezultat dobija 2 doktora

Na isti nacin provera stanje i doktor B i dobija isti rezultat, 2 aktivna doktora.

Svako od njih pojedinačno odlučuje da ode kući.

```
> d1.doctors.updateOne({ doctor: "A" }, { $set: { onDuty: false } })
< {
  acknowledged: true,
  insertedId: null,
  matchedCount: 1,
  modifiedCount: 1,
  upsertedCount: 0
}
```

Slika 26 Doktor A stavlja svoje onDuty stanje na false

Takodje to radi i doktor B.

```
> d2.doctors.updateOne({ doctor: "B" }, { $set: { onDuty: false } })
< {
  acknowledged: true,
  insertedId: null,
  matchedCount: 1,
  modifiedCount: 1,
  upsertedCount: 0
}
```

Slika 27 Doktor B stavlja stoju dužnost na false

Obe transakcije odrade *commit*.

```
s1.commitTransaction()
s2.commitTransaction()
```

Rezultat:

```
_id: ObjectId('69320ea863a19d7dc6847b78')
doctor : "A"
onDuty : false

-----
_id: ObjectId('69320ea863a19d7dc6847b79')
doctor : "B"
onDuty : false
```

Slika 28 Nakon commita transkacije nijedan doktor više nije na dužnosti

U demonstraciji sa dežurnim doktorima vidimo **tipičnu anomaliju snapshot izolacije – write skew**:

- Svaka transakcija vidi **snapshot baze** u trenutku starta.
- Doktor A i doktor B čitaju stanje i oba vide da su **dva doktora dežurna**.

- Oba odjavljuju sebe, jer snapshot sugeriše da je još uvek dovoljan broj dežurnih.
- Nakon commit-a obe transakcije, broj dežurnih doktora **padne ispod minimalnog** (npr. na 0), što stvara nekonzistentno stanje.

*Snapshot isolation* garantuje da transakcija vidi konzistentan pogled baze, ali **ne štiti od write skew-a**.

Ovo je slabost *SI* koju je važno uzeti u obzir pri dizajnu kritičnih sistema.

### Primer -*Phantom reads*

*Phantom read* je pojava kada isti upit vraća različite rezultate u dve uzastopne čitanja, jer su novi dokumenti umetnuti od strane druge transakcije ili sesije između dva čitanja.

```
db.values.deleteMany( {} );
db.values.insertMany([
  { value: 1 },
  { value: 2 }
]);

```

Trenutno kolekcija values izgleda ovako:

```
{ value: 1 }
{ value: 2 }
```

Nakon prvog čitanja dobijamo ove dve vrednosti, međutim kada drugi korisnik dodaje novi element:

```
db.values.insertOne({ value: 999 });
```

Sada ce nam upit

```
db.values.find({ value: { $gt: 0 } })
```

Vratiti 3 vrednosti.

### Zašto se ovo dešava

- *MongoDB van transakcija* ne koristi snapshot isolation.
- Svaki upit vidi **trenutno stanje baze**.

- Kada neko doda novi dokument između dva čitanja, on postaje vidljiv → *phantom read*.

### Kako to izgleda unutar transakcije?

```

session = db.getMongo().startSession();
dbs = session.getDatabase("Hospital");
session.startTransaction();

dbs.values.find({ value: { $gt: 0 } });
{
  _id: ObjectId('693219e0c584839100da9fba'),
  value: 1
}
{
  _id: ObjectId('693219e0c584839100da9fbb'),
  value: 2
}
{
  _id: ObjectId('69321a1bd58d886681aff650'),
  value: 999
}

```

Slika 29 Kolekcija values

Sada kada novi korisnik doda vrednost u bazi van te transakcije, mi tu vrednost ne možemo videti to trenutka dok ne *commitujemo* našu transakciju.

Da sumiramo:

- ***Phantom read***: isti upit vraća različite rezultate zbog novih dokumenata umetnutih od strane drugih sesija.
- **Van transakcija** → MongoDB dozvoljava phantom reads.
- **Unutar transakcije (*snapshot isolation*)** → *phantom reads* se NE dešavaju; upit vidi konzistentan *snapshot*.

