Santin Simone 886116

Corso di Architettura Dati:

Progettazione e analisi di un cluster MongoDB

Sommario

1. Introduzione

L'obiettivo del progetto è simulare un sistema di cluster distribuito utilizzando MongoDB, configurato con 4 shard. Ogni shard è strutturato secondo l'architettura replica set, composta da un nodo primario, uno secondario e un arbitro, al fine di garantire tolleranza ai guasti e disponibilità elevata del sistema anche in caso di malfunzionamenti o interruzioni di singoli nodi.

Il progetto si propone di analizzare il comportamento di MongoDB in tre aree fondamentali:

1. **Performance e scalabilità**: vengono misurati i tempi di caricamento e di interrogazione di un dataset di medie-grandi dimensioni, con particolare attenzione alle query multi-documento distribuite tra i diversi shard.
2. **Politiche di isolamento nelle transazioni:** si studia come MongoDB gestisce i principali fenomeni legati alla concorrenza, come *lost updates*, *dirty reads*, *non-repeatable reads* e *phantom reads*, verificando quali problematiche vengano effettivamente evitate grazie al livello di isolamento supportato.
3. **Gestione dei guasti durante le transazioni**: si simulano scenari in cui un nodo fallisce durante l'esecuzione di una transazione, per osservare come il sistema replica set mantiene la coerenza e l'affidabilità dei dati.

In conclusione, lo scopo del progetto è valutare l'efficacia e l'affidabilità di MongoDB nel gestire dati distribuiti in un ambiente realistico, caratterizzato da carichi rilevanti, concorrenza tra operazioni e possibili fault di sistema.

2. Creazione del cluster

Per sviluppare il cluster e garantire una simulazione quanto più realistica possibile, è stata utilizzata una macchina virtuale all'interno della quale è stata creata l'intera struttura del sistema. Questa scelta, pur non replicando fedelmente un ambiente distribuito reale (in cui ogni shard risiederebbe idealmente su una macchina separata), consente comunque di introdurre un certo livello di latenza e complessità nella comunicazione tra nodi.

Anche se non è stata creata una macchina virtuale distinta per ogni replica set, la configurazione adottata fornisce dati comunque significativi e affidabili, in particolare per quanto riguarda i tempi di caricamento e le prestazioni nella gestione di un dataset di dimensioni rilevanti.

2.1 Creazione del cluster

Per la creazione del cluster è stato utilizzato Docker Compose. Il cluster è strutturato su 4 shard distinti, ognuno configurato con un nodo primario, uno secondario e un arbitro, al fine di garantire una maggiore resistenza ai guasti e una maggiore disponibilità del sistema.

In fase di avvio, sono stati innanzitutto creati i container configsvr e mongos. Il configsvr ha il compito di memorizzare la mappa dell'intero cluster sharded, mantenendo traccia della distribuzione dei dati e ascoltando su una porta dedicata. Il mongos, invece, funge da router: riceve le richieste da parte del client e le inoltra agli shard corretti, utilizzando le informazioni fornite dal configsvr.

```yaml
services:
  configsvr:
    image: mongo:4.4.0-bionic
    container_name: configsvr
    command: ["mongod", "--configsvr", "--replSet", "rsConfig", "--port", "27019"]
    ports:
      - 27019:27019
    volumes:
      - configsvr:/data/db
    networks:
      - mongonet

  mongos:
    image: mongo:4.4.0-bionic
    container_name: mongos
    command: ["mongos", "--configdb", "rsConfig/configsvr:27019", "--port", "27017", "--bind_ip_all"]
    ports:
      - 27017:27017
    depends_on:
      - configsvr
      - shard1
      - shard2
      - shard3
      - shard4
    networks:
      - mongonet
```

Figura 1: codice configsvr e mongos

Per quanto riguarda gli shard, ciascuno è stato configurato come replica set con nodo primario, secondario e arbitro, per garantire resilienza ai failover e assicurare sempre una maggioranza valida durante l'elezione del primario, grazie al nodo arbitro. A ogni nodo primario è stato associato un volume bind mount, montato su un percorso specifico del file system della macchina virtuale. Questa configurazione permette di simulare un ambiente distribuito in cui ogni shard legge e scrive su dischi separati, riducendo così il rischio di colli di bottiglia sull'I/O durante operazioni intensive come le transazioni.

```yaml
shard1:
  image: mongo:4.4.0-bionic
  container_name: shard1
  command: ["mongod", "--shardsvr", "--replSet", "rsShard1", "--port", "27018"]
  volumes:
    - /mnt/shard1_data:/data/db
  networks:
    - mongonet

shard1_secondary:
  image: mongo:4.4.0-bionic
  container_name: shard1_secondary
  command: ["mongod", "--shardsvr", "--replSet", "rsShard1", "--port", "27018"]
  volumes:
    - shard1_secondary:/data/db
  networks:
    - mongonet

shard1_arbiter:
  image: mongo:4.4.0-bionic
  container_name: shard1_arbiter
  command: ["mongod", "--replSet", "rsShard1", "--port", "27018"]
  volumes:
    - shard1_arbiter:/data/db
  networks:
    - mongonet
```

Figura 2: codice struttura shard

Per tutti gli altri container (nodi secondari, arbitri, mongos e configsvr), sono stati invece utilizzati volumi gestiti da Docker (named volumes), in quanto sufficienti per le loro esigenze e più semplici da configurare e gestire.

Infine, tutti i container sono stati connessi alla stessa rete virtuale, chiamata mongonet, per permettere una comunicazione diretta e stabile tra i vari componenti del cluster.

Questa configurazione consente di ricreare fedelmente le dinamiche di un cluster MongoDB sharded, permettendo di testarne le performance, la tolleranza ai guasti e la gestione delle transazioni in un ambiente controllato.

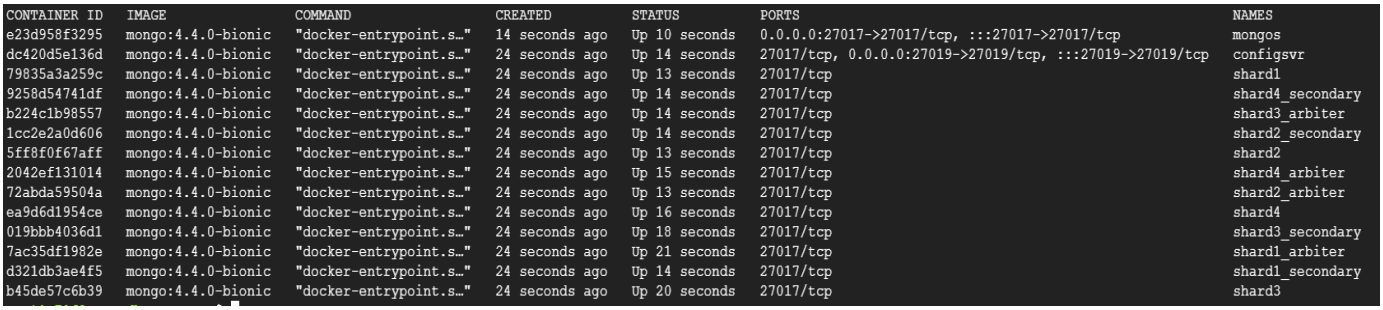
2.2 Setup del cluster

Per il setup del cluster sono stati eseguiti una serie di comandi attraverso la ssh della macchina virtuale, piu precisamente:

- **docker-compose up -d**
  Avvia tutti i container definiti nel file docker-compose in modalità detached (in background).
- **docker exec -it configsvr bash**
  Entra nella shell bash del container configsvr per eseguire comandi all'interno.
- **mongo --port 27019 --eval 'rs.initiate({...})' (nel container configsvr)**
  Inizializza il replica set del config server, che gestisce la configurazione del cluster sharded.
- **docker exec -it shard1 bash**
  Entra nella shell bash del container shard1 (primo shard).
- **mongo --port 27018 --eval 'rs.initiate({...})' (nel container shard1)**
  Inizializza il replica set per il primo shard, definendo membri primario, secondario e arbitro.
- **Comandi analoghi per shard2, shard3, shard4**
  Entrano nei rispettivi container e inizializzano i replica set di ogni shard con la stessa struttura (primario, secondario, arbitro).
- **docker exec -it mongos bash**
  Entra nella shell bash del container mongos, il router del cluster sharded.
- **mongo --port 27017 --eval 'sh.addShard("rsShardX/shardX:27018")'**
  Aggiunge ciascun replica set shard al cluster tramite mongos.
- **mongo --port 27017 --eval 'sh.enableSharding("mongoDB")'**
  Abilita lo sharding sul database mongoDB.
- **mongo --port 27017 --eval 'sh.shardCollection("mongoDB.collection", { "artist": 1 })'**

Abilita lo sharding sulla collezione collection del database mongoDB usando il campo artist come chiave di shard.

Questo processo consente quindi di configurare un cluster MongoDB sharded completo, dove i dati vengono distribuiti tra diversi shard, ciascuno con i propri nodi primari, secondari e arbitri. In questo modo si ottiene una maggiore scalabilità orizzontale, una migliore gestione del carico di lavoro e una maggiore disponibilità del sistema.



CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
e23d958f3295	mongo:4.4.0-bionic	"docker-entrypoint.s…"	14 seconds ago	Up 10 seconds	0.0.0.0:27017->27017/tcp, :::27017->27017/tcp	mongos
dc420d5e136d	mongo:4.4.0-bionic	"docker-entrypoint.s…"	24 seconds ago	Up 14 seconds	27017/tcp, 0.0.0.0:27019->27019/tcp, :::27019->27019/tcp	configsvr
79835a3a259c	mongo:4.4.0-bionic	"docker-entrypoint.s…"	24 seconds ago	Up 13 seconds	27017/tcp	shard1
9258d54741df	mongo:4.4.0-bionic	"docker-entrypoint.s…"	24 seconds ago	Up 14 seconds	27017/tcp	shard4_secondary
b224c1b98557	mongo:4.4.0-bionic	"docker-entrypoint.s…"	24 seconds ago	Up 14 seconds	27017/tcp	shard3_arbiter
1cc2e2a0d606	mongo:4.4.0-bionic	"docker-entrypoint.s…"	24 seconds ago	Up 14 seconds	27017/tcp	shard2_secondary
5ff8f0f67aff	mongo:4.4.0-bionic	"docker-entrypoint.s…"	24 seconds ago	Up 13 seconds	27017/tcp	shard2
2042ef131014	mongo:4.4.0-bionic	"docker-entrypoint.s…"	24 seconds ago	Up 15 seconds	27017/tcp	shard4_arbiter
72abda59504a	mongo:4.4.0-bionic	"docker-entrypoint.s…"	24 seconds ago	Up 13 seconds	27017/tcp	shard2_arbiter
ea9d6d1954ce	mongo:4.4.0-bionic	"docker-entrypoint.s…"	24 seconds ago	Up 16 seconds	27017/tcp	shard4
019bbb4036d1	mongo:4.4.0-bionic	"docker-entrypoint.s…"	24 seconds ago	Up 18 seconds	27017/tcp	shard3_secondary
7ac35df1982e	mongo:4.4.0-bionic	"docker-entrypoint.s…"	24 seconds ago	Up 21 seconds	27017/tcp	shard1_arbiter
d321db3ae4f5	mongo:4.4.0-bionic	"docker-entrypoint.s…"	24 seconds ago	Up 14 seconds	27017/tcp	shard1_secondary
b45de57c6b39	mongo:4.4.0-bionic	"docker-entrypoint.s…"	24 seconds ago	Up 20 seconds	27017/tcp	shard3

Figura 3: struttura cluster dopo l'avvio

3. Performance e Scalabilità

Per testare le performance e la scalabilità del cluster è stato utilizzato un dataset di Spotify contenente circa 237.000 voci di canzoni. Inizialmente è stato misurato il tempo di caricamento progressivo mantenendo costante il numero di shard e aumentando gradualmente la dimensione del dataset. Successivamente, mantenendo la quantità di dati fissa, è stato valutato l'impatto della variazione del numero di nodi sul tempo di caricamento.

3.1 Caricamento e query dataset

Per quanto riguarda il funzionamento del caricamento del dataset sono state create le seguenti funzioni:

```
public static List<Document> getData() {
    int MAX_ROWS = 236988;
    int currentLine = 0;
    List<Document> documents = new ArrayList<>();
    try (BufferedReader reader = new BufferedReader(new FileReader(fileName:"spotify_dataset.json"))) {
        String line;
        while ((line = reader.readLine()) != null && currentLine < MAX_ROWS) {
            currentLine++;
            JSONObject jsonObject = new JSONObject(line);
            Document doc = Document.parse(jsonObject.toString());
            documents.add(doc);
        }
    } catch (IOException | JSONException e) {
        e.printStackTrace();
    }
    return documents;
}
```

Figura 4: funzione per caricamento dati

La funzione getData() legge fino a un massimo di 236.988 righe da un file chiamato spotify_dataset.json. Per ogni riga, interpreta il contenuto come un oggetto JSON, lo converte in un documento MongoDB (Document) e lo aggiunge a una lista. Alla fine, restituisce questa lista di documenti.

La variabile MAX_ROWS viene sfruttata per diminuire la quantità di dati da caricare, in modo da riuscire a testare i tempi di caricamento successivamente.

```
MongoDatabase database = mongoClient.getDatabase(databaseName:"mongoDB");
MongoCollection<Document> collection = database.getCollection(collectionName:"collection");
System.out.println(x:"Connesso");
collection.drop();
List<Document> songs = getData();
startTime = System.currentTimeMillis();
collection.insertMany(songs);
docs = (int) collection.countDocuments(Filters.eq(fieldName:"artist", value:"Metallica"));
System.out.println("trovati " + docs + " canzoni dei Metallica");
long endTime = System.currentTimeMillis();
long duration = endTime - startTime;
System.out.println("\n Trovati " +docs+ " documenti, in " + duration + " millisecondi");
```

Figura 5: testing caricamento e query sul dataset

Infine, una volta stabilita la connessione al database, la collezione esistente viene eliminata per permettere un nuovo caricamento pulito del dataset. Successivamente, i documenti ottenuti vengono inseriti nella collezione corrispondente del database, sulla quale viene infine eseguita una query multi-documento per analizzare o verificare i dati caricati.

3.2 Statistiche Incremento dataset

Nello studio dell'incremento del dataset, mantenendo costante il numero di nodi, sono stati effettuati quattro test con dataset di dimensioni crescenti: un decimo, un quinto, metà e infine l'intero dataset. I risultati ottenuti sono i seguenti:

- Un decimo del dataset:

```
Trovati 87 documenti, in 4198 millisecondi
```

- Un quinto del dataset:

```
Trovati 155 documenti, in 8575 millisecondi
```

- Un mezzo del dataset:

```
Trovati 155 documenti, in 20740 millisecondi
```

- Intero dataset:

```
Trovati 284 documenti, in 40178 millisecondi
```

3.3 Statistiche Incremento nodi

Per quanto riguarda invece lo studio dell'incremento dei nodi mantenendo costante il numero di dati sono stati ottenuti i seguenti risultati:

- Un nodo:

```
Trovati 284 documenti, in 44968 millisecondi
```

- Due nodi:

```
Trovati 284 documenti, in 44224 millisecondi
```

- Tre nodi:

```
Trovati 284 documenti, in 41561 millisecondi
```

- Quattro nodi:

```
Trovati 284 documenti, in 40178 millisecondi
```

3.4 Analisi delle performance

In conclusione, è possibile trarre alcune osservazioni rilevanti. Nel caso del caricamento graduale dei dati mantenendo costante il numero di nodi, si osserva un incremento significativo del tempo richiesto per le operazioni: all'aumentare del volume del dataset, cresce rapidamente anche il tempo necessario per completare le query, arrivando fino a 40 secondi per una ricerca sull'intero insieme di dati.

Al contrario, aumentando progressivamente il numero di nodi a parità di dati, si nota una riduzione nei tempi di risposta. Questo miglioramento è inizialmente modesto , soprattutto nel passaggio da uno a due nodi, ma diventa più evidente con l'aggiunta di ulteriori nodi, fino a quattro.

È importante sottolineare che, sebbene il cluster sia stato implementato all'interno di una singola macchina virtuale, e quindi non rifletta fedelmente le condizioni di un'infrastruttura distribuita reale (dove la latenza tra nodi sarebbe maggiore), questo setup consente comunque di osservare in modo significativo l'impatto che l'architettura distribuita ha sul caricamento e sull'interrogazione dei dati in MongoDB, simulando uno scenario cloud in maniera efficace.

4. Politiche di isolamento

Per quanto riguarda il testing delle politiche di isolamento, sono state analizzate quattro anomalie:

- **Lost updates**: due scritture concorrenti sovrascrivono i dati senza tener conto l'una dell'altra.
- **Dirty reads**: si leggono dati scritti da una transazione non ancora confermata.
- **Non-repeatable reads**: lo stesso dato letto due volte restituisce valori diversi.
- **Phantom reads**: la stessa query eseguita due volte restituisce risultati diversi a causa di nuovi dati inseriti o rimossi.

Ognuna è stata testata all'interno di una transazione con due thread per simulare concorrenza tra processi. I test sono stati effettuati sia con l'uso della variabile *session*, che garantisce l'isolamento in MongoDB, sia senza, per evidenziare i casi in cui tale isolamento viene meno.

4.1 Lost updates

Per quanto riguarda i lost updates sono stati eseguiti due Thread in concorrenza, entrambi leggono lo stesso documento allo stesso momento e aggiornano la stessa variabile a due valori differenti.

```
Runnable userA = () -> {
    try (ClientSession session = mongoClient.startSession()) {
        session.startTransaction();
        Document doc = collection.find(session, Filters.eq(fieldName:"song", value:"Master of Puppets")).first();
        System.out.println("A legge: " + doc.getInteger(key:"Tempo"));
        int tempo = doc.getInteger(key:"Tempo") - 5;
        collection.updateOne(session, Filters.eq(fieldName:"song", value:"Master of Puppets"), set(fieldName:"Tempo", tempo));
        session.commitTransaction();
        System.out.println("A ha scritto: " + tempo);
    } catch (Exception e) {
        e.printStackTrace();
    }
};

Runnable userB = () -> {
    try (ClientSession session = mongoClient.startSession()) {
        session.startTransaction();
        Document doc = collection.find(session, Filters.eq(fieldName:"song", value:"Master of Puppets")).first();
        System.out.println("B legge: " + doc.getInteger(key:"Tempo"));
        int tempo = doc.getInteger(key:"Tempo") - 10;
        collection.updateOne(session, Filters.eq(fieldName:"song", value:"Master of Puppets"), set(fieldName:"Tempo", tempo));
        session.commitTransaction();
        System.out.println("B ha scritto: " + tempo);
    } catch (Exception e) {
        e.printStackTrace();
    }
};


Thread tA = new Thread(userA);
Thread tB = new Thread(userB);

tA.start();
tB.start();

tA.join();
tB.join();

Document finalDoc = collection.find(Filters.eq(fieldName:"song", value:"Master of Puppets")).first();
System.out.println("Valore finale nel database: " + finalDoc);
```

Figura 6: testing lost updates

Nel caso di esecuzione concorrente in due transazioni, MongoDB protegge dai *lost update*: se due operazioni tentano di modificare lo stesso documento, viene segnalato un conflitto *WriteConflict*.

Tuttavia, omettendo la variabile session, il comportamento cambia, poiché è proprio questa a garantire consistenza e isolamento tra le operazioni. Senza di essa, si perdono le protezioni offerte dalle transazioni, con i seguenti risultati:

```
A legge: 80
B legge: 80
```

```
B ha scritto: 70
A ha scritto: 75
```

```
Valore finale nel database: Document{{_id=68459441bea7dc4c20c5a425, song=Master of Puppets, Loudness=-9.11, Positiveness=56, Acousticness=0, Speechiness=4, artist=Metallica, Liveness=15, Popularity=77, Explicit=No, Instrumentalness=43, Danceability=54, emotion=sadness, Tempo=70, Energy=84, variance=0.8335142456515902, Genre=metal, Key=E min, Release Date=1986}}
```

Quindi si può notare la presenza di un lost update nel caso si escluda tale variabile; infatti, il valore finale dovrebbe essere 80-10-5 = 65, invece B sovrascrive semplicemente il valore di A.

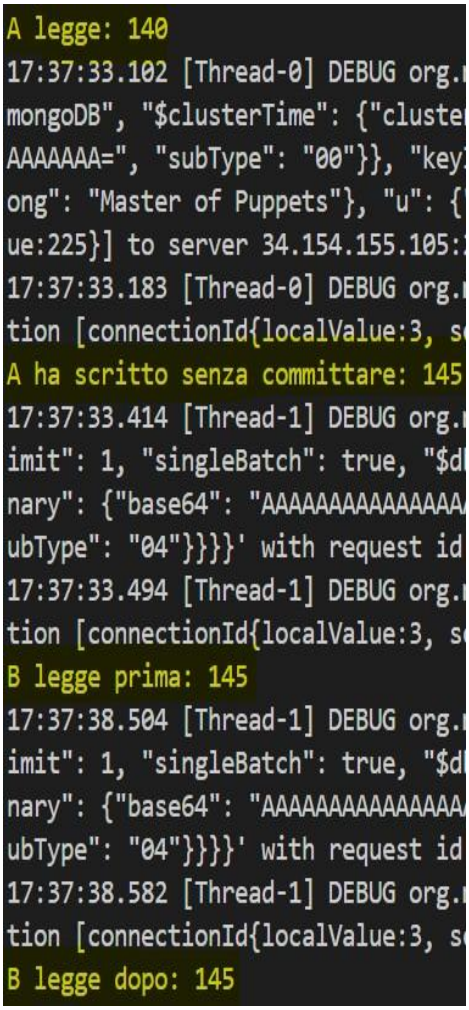
4.2 Dirty reads

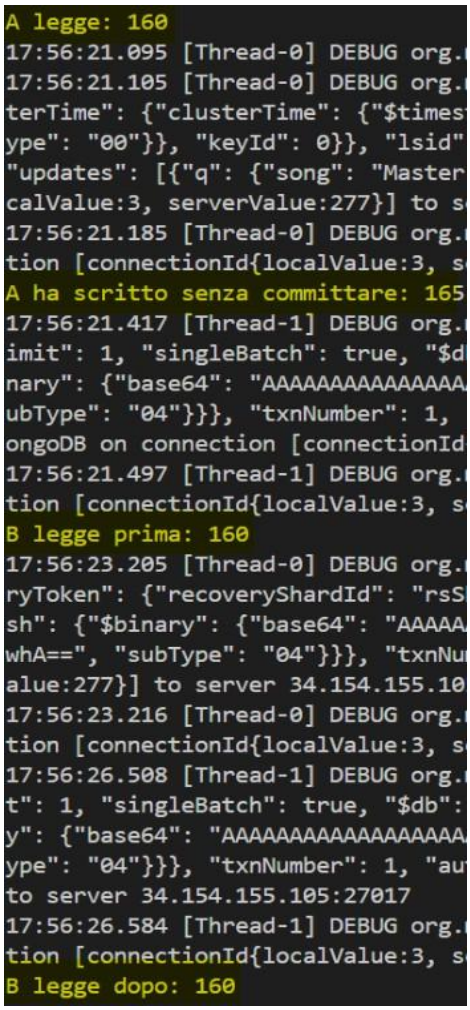
Per testare i *dirty reads* sono stati creati due thread separati. Dopo l'avvio del primo thread (writer), viene introdotto un ritardo di mezzo secondo prima di avviare il secondo (reader). In questo modo, writer può iniziare una transazione e modificare un documento senza committarla, rimanendo in pausa per 5 secondi prima di eseguire un rollback. Durante questo intervallo, il thread reader tenta di leggere lo stesso documento, così da verificare se riesce a vedere la modifica non ancora confermata.

```
Runnable writer = () -> {
    try (ClientSession session = mongoClient.startSession()) {
        session.startTransaction();
        Document doc = collection.find(session, Filters.eq(fieldName:"song", value:"Master of Puppets")).first();
        System.out.println("A legge: " + doc.getInteger(key:"Tempo"));
        int tempo = doc.getInteger(key:"Tempo") + 5;
        collection.updateOne(session, Filters.eq(fieldName:"song", value:"Master of Puppets"), set(fieldName:"Tempo", tempo));
        System.out.println("A ha scritto senza committare: " + tempo);
        Thread.sleep(millis:2000);  // Simula un ritardo per il lettore
        session.abortTransaction();
    } catch (Exception e) {
        e.printStackTrace();
    }
};

Runnable reader = () -> {

    try (ClientSession session = mongoClient.startSession()) {
        session.startTransaction();
        Document doc = collection.find(session, Filters.eq(fieldName:"song", value:"Master of Puppets")).first();
        System.out.println("B legge prima: " + doc.getInteger(key:"Tempo"));
        Thread.sleep(millis:5000);
        Document doc2 = collection.find(session, Filters.eq(fieldName:"song", value:"Master of Puppets")).first();
        System.out.println("B legge dopo: " + doc2.getInteger(key:"Tempo"));
        session.commitTransaction();
    } catch (Exception e) {
        e.printStackTrace();
    }
};


Thread tA = new Thread(writer);
Thread tB = new Thread(reader);

tA.start();
Thread.sleep(millis:500);
tB.start();

tA.join();
tB.join();
```

Figura 7: testing dirty reads

Come già osservato in precedenza, sono stati considerati due casi: con e senza la variabile *session*. Questa variabile, infatti, garantisce l'isolamento all'interno delle operazioni di transazione, permettendo alle query di vedere lo snapshot del database all'inizio della transazione.





Quindi, come si può osservare, escludendo la variabile *session* e rinunciando al relativo isolamento, si verifica un dirty read: il secondo thread riesce a leggere dati non ancora committati, cosa che non accade quando la variabile *session* è presente.

4.3 Non-repeatable reads

Per i non-repeatable reads, sono stati testati, come gia fatto in precedenza, due casi, con e senza la variabile *session*, per verificare se tra la prima e la seconda

lettura, separate da una pausa di 5 secondi, il documento venga aggiornato dal thread di scrittura.

```
Runnable writer = () -> {
   try (ClientSession session = mongoClient.startSession()) {
        session.startTransaction();
        Document doc = collection.find(session, Filters.eq(fieldName:"artist", value:"ABBA")).first();
        int tempo = doc.getInteger(key:"Tempo") - 15;
        collection.updateOne(session, Filters.eq(fieldName:"artist", value:"ABBA"), set(fieldName:"Tempo", tempo));
        Document doc2 = collection.find(session, Filters.eq(fieldName:"artist", value:"ABBA")).first();
        System.out.println("Writer ha scritto: " + doc2.getInteger(key:"Tempo"));
        session.commitTransaction();
    } catch (Exception e) {
        e.printStackTrace();
    }
};

Runnable reader = () -> {
    try (ClientSession session = mongoClient.startSession()) {
        session.startTransaction();
        Document docFirstRead = collection.find(session, Filters.eq(fieldName:"artist", value:"ABBA")).first();
        System.out.println("Prima lettura: " + docFirstRead.getInteger(key:"Tempo"));
        Thread.sleep(millis:5000);
        Document docSecondRead = collection.find(session, Filters.eq(fieldName:"artist", value:"ABBA")).first();
        System.out.println("Seconda lettura: " + docSecondRead.getInteger(key:"Tempo"));
        session.commitTransaction();
    } catch (Exception e) {
        e.printStackTrace();
    }
};

Thread tA = new Thread(writer);
Thread tB = new Thread(reader);

tA.start();
tB.start();

tA.join();
tB.join();
```

Figura 8: Testing non-repetable reads

Come già osservato in precedenza, la presenza o meno della variabile *session* influenza i risultati. Quando la *session* non viene utilizzata, tra la prima e la seconda lettura il valore letto cambia, mostrando quindi un non-repeatable read. Al contrario, con la *session* attiva, grazie all'isolamento basato su snapshot garantito dalla transazione, la lettura rimane coerente e identica sia prima che dopo la pausa del thread:

4.4 Phantom reads

Infine, per quanto riguarda i phantom reads, la logica è simile a quella dei non-repeatable reads, ma invece di considerare un singolo documento, si valuta un insieme di documenti. Il primo thread esegue quindi un conteggio degli elementi che soddisfano una certa query, poi si mette in pausa per 3 secondi. Durante questa pausa, il secondo thread aggiunge un nuovo documento che rispetta i criteri della query. Al termine, il primo thread riesegue la stessa query per verificare se il numero di documenti è aumentato, evidenziando così la presenza di phantom reads:

```
Runnable userA = () -> {
    try (ClientSession session = mongoClient.startSession()) {
        session.startTransaction();
        List<Document> firstRead = collection.find(session, Filters.gt(fieldName:"artist", value:"ABBA")).into(new ArrayList<>());
        System.out.println("UserA: Primo risultato count = " + firstRead.size());
        Thread.sleep(millis:3000);
        List<Document> secondRead = collection.find(session, Filters.gt(fieldName:"artist", value:"ABBA")).into(new ArrayList<>());
        System.out.println("UserA: Secondo risultato count = " + secondRead.size());
        session.commitTransaction();

    } catch (Exception e) {
        e.printStackTrace();
    }
};

Runnable userB = () -> {
    try (ClientSession session = mongoClient.startSession()) {
        session.startTransaction();

        Document newDoc = new Document(key:"artist", value:"ABBA")
            .append(key:"song", value:"example song")
            .append(key:"emotion", value:"joy")
            .append(key:"variance", value:0.4714285714285714)
            .append(key:"Genre", value:"pop")
            .append(key:"Release Date", value:1975)
            .append(key:"Key", value:"A# Maj")
            .append(key:"Tempo", value:82)
            .append(key:"Loudness", -10.57)
            .append(key:"Explicit", value:"No")
            .append(key:"Popularity", value:38)
            .append(key:"Energy", value:47)
            .append(key:"Danceability", value:66)
            .append(key:"Positiveness", value:84)
            .append(key:"Speechiness", value:6)
            .append(key:"Liveness", value:7)
            .append(key:"Acousticness", value:53)
            .append(key:"Instrumentalness", value:0);

        collection.insertOne(session, newDoc);
        session.commitTransaction();
        System.out.println(x:"UserB: Inserimento completato");
    } catch (Exception e) {
        e.printStackTrace();
    }
};
```

Figura 9: Testing phantom reads

Analogamente a quanto avviene con i non-repeatable reads, escludendo la variabile session si ottengono risultati diversi. Infatti, all'interno di una transazione, la sessione garantisce l'isolamento, impedendo così la comparsa di phantom reads. Nei risultati ottenuti, i phantom reads non sono stati evidenziati perché il tempo necessario al thread per eseguire la query su tutto il dataset è molto più lungo dello stop impostato, rendendo il test complesso. Tuttavia, la somiglianza nella logica con i non-repeatable reads suggerisce che il comportamento sia analogo.

```
UserA: Primo risultato count = 232973
UserA: Secondo risultato count = 232973
```

5. Gestione dei guasti

Infine, è stata testata la gestione dei guasti in una transazione multi-documento. La struttura di MongoDB, composta da nodo primario, secondario e arbitro, è progettata per garantire la resilienza ai guasti, permettendo l'elezione automatica di un nuovo nodo primario in caso di failover. L'esperimento verifica se, durante una transazione, l'interruzione improvvisa di uno shard causa il fallimento della transazione oppure se questa riesce a completarsi con successo.

5.1 Operazioni svolte

Per testare il funzionamento dei guasti è stato testato il seguente codice:

```
try (ClientSession session = mongoClient.startSession()) {
    Document song1 = collection.find(Filters.eq(fieldName:"song", value:"Master of Puppets")).first();
    Document song2 = collection.find(Filters.eq(fieldName:"song", value:"King Nothing")).first();
    System.out.println("Canzoni prima di essere aggiornati: \n" + song1.getInteger(key:"Tempo") +" "+ song2.getInteger(key:"Tempo"));
    session.startTransaction();

    collection.updateOne(session, Filters.eq(fieldName:"song", value:"Master of Puppets"),
            new Document(key:"$set", new Document(key:"Tempo", value:115)));
    Thread.sleep(millis:10000);
    collection.updateOne(session, Filters.eq(fieldName:"song", value:"King Nothing"),
            new Document(key:"$set", new Document(key:"Tempo", value:120)));

    session.commitTransaction();
    System.out.println(x:"Transazione completata con successo.");
    Document updatedSong1 = collection.find(Filters.eq(fieldName:"song", value:"Master of Puppets")).first();
    Document updatedSong2 = collection.find(Filters.eq(fieldName:"song", value:"King Nothing")).first();
    System.out.println("Canzoni dopo essere aggiornati: \n" + updatedSong1.getInteger(key:"Tempo") +" "+ updatedSong2.getInteger(key:"Tempo"));
} catch (Exception e) {
    e .printStackTrace();
}
```

Figura 10: Testing failover di un nodo

Vengono eseguite due query multi-documento per recuperare due documenti diversi. Prima viene aggiornato il valore del primo documento, quindi, dopo una pausa di dieci secondi durante la quale il nodo viene spento, si aggiorna il secondo documento e si committa la transazione. Infine, i documenti vengono ricercati nel dataset per verificare che le operazioni siano state completate con successo.

La sequenza di azioni eseguita è quindi la seguente:

1. Avvio della transazione
2. Spegnimento del nodo primario *(docker stop shard1)*

3. Visualizzazione dei risultati
4. Conferma del failover del nodo e dell'elezione a nodo primario il nodo secondario (mongo –port 27018 eval 'rs.status()' all'interno della bash dello shard1_secondary)

5.2 Risultati

Dopo aver eseguito l'interruzione del nodo durante lo stop della transazione, si sono potuti notare i seguenti risultati:

```
Canzoni prima di essere aggiornati:
105 100
```

```
Canzoni dopo essere aggiornati:
115 120
```

Da come si può notare, l'operazione è stata completata con successo, il che indica che, durante l'interruzione del nodo primario del primo shard, il nodo secondario è stato correttamente promosso a primario. Analizzando più nel dettaglio:

```
"members" : [
        {
                "_id" : 0,
                "name" : "shard1:27018",
                "health" : 0,
                "state" : 8,
                "stateStr" : "(not reachable/healthy)",
                "uptime" : 0,
                "optime" : {
                        "ts" : Timestamp(0, 0),
                        "t" : NumberLong(-1)
                },
                "optimeDurable" : {
                        "ts" : Timestamp(0, 0),
                        "t" : NumberLong(-1)
                },
                "optimeDate" : ISODate("1970-01-01T00:00:00Z"),
                "optimeDurableDate" : ISODate("1970-01-01T00:00:00Z"),
                "lastHeartbeat" : ISODate("2025-06-09T16:30:36.963Z"),
                "lastHeartbeatRecv" : ISODate("2025-06-09T16:30:23.775Z"),
                "pingMs" : NumberLong(0),
                "lastHeartbeatMessage" : "Couldn't get a connection within the time limit of 627ms",
                "syncSourceHost" : "",
                "syncSourceId" : -1,
                "infoMessage" : "",
                "configVersion" : 2,
                "configTerm" : 9
        },
        {
                "_id" : 1,
                "name" : "shard1_secondary:27018",
                "health" : 1,
                "state" : 1,
                "stateStr" : "PRIMARY",
                "uptime" : 423,
                "optime" : {
                        "ts" : Timestamp(1749486631, 1),
                        "t" : NumberLong(9)
                },
                "optimeDate" : ISODate("2025-06-09T16:30:31Z"),
                "syncSourceHost" : "",
                "syncSourceId" : -1,
                "infoMessage" : "Could not find member to sync from",
                "electionTime" : Timestamp(1749486591, 1),
                "electionDate" : ISODate("2025-06-09T16:29:51Z"),
                "configVersion" : 2,
                "configTerm" : 9,
                "self" : true,
                "lastHeartbeatMessage" : ""
        },
        {
                "_id" : 2,
                "name" : "shard1_arbiter:27018",
                "health" : 1,
                "state" : 7,
                "stateStr" : "ARBITER",
                "uptime" : 212,
                "lastHeartbeat" : ISODate("2025-06-09T16:30:37.345Z"),
                "lastHeartbeatRecv" : ISODate("2025-06-09T16:30:37.381Z"),
                "pingMs" : NumberLong(0),
                "lastHeartbeatMessage" : "",
                "syncSourceHost" : "",
                "syncSourceId" : -1,
                "infoMessage" : "",
                "configVersion" : 2,
                "configTerm" : 9
        }
],
```


Figura 11: stato dello shard1_secondary

Come si può osservare, la struttura con nodo primario, secondario e arbitro garantisce la resistenza ai failover assicurando la maggioranza. Infatti, il nodo shard1_secondary, che prima era secondario, è stato promosso a primario, mentre il precedente nodo primario shard1 è diventato irraggiungibile.

6. Conclusioni

L'obiettivo del progetto era analizzare le caratteristiche di scalabilità, prestazioni e affidabilità di un cluster MongoDB configurato in ambiente distribuito simulato.

Le analisi condotte hanno mostrato con chiarezza come l'aumento del numero di documenti, a parità di configurazione del cluster, comporti un incremento significativo dei tempi di caricamento e delle query, soprattutto quando si raggiunge la scala del dataset completo. Al contrario, aumentando il numero di nodi mantenendo fisso il dataset, si osserva una riduzione dei tempi di risposta, seppur non lineare, questo conferma come la scalabilità orizzontale offerta da MongoDB sia effettivamente vantaggiosa.

È stata poi testata l'analisi delle politiche di isolamento, con test condotti su dirty reads, lost updates, non-repeatable reads e phantom reads. In tutti i casi, è emersa l'importanza dell'utilizzo esplicito della variabile session nelle transazioni per garantire l'isolamento e la consistenza dei dati. In sua assenza, infatti, anche in presenza di transazioni formalmente corrette, è possibile incorrere in letture incoerenti o comportamenti inattesi.

Infine, è stata testata la gestione dei guasti simulando l'interruzione del nodo primario durante una transazione. I risultati hanno confermato la resilienza del sistema: grazie alla struttura a replica set, il nodo secondario ha assunto correttamente il ruolo di primario, permettendo alla transazione di concludersi con successo, evidenziando cosi l'affidabilità del meccanismo di failover automatico di MongoDB, anche in scenari critici.

Nel complesso, il lavoro ha permesso di approfondire sia le caratteristiche architetturali che i limiti operativi di MongoDB in un contesto realistico. Le simulazioni condotte, pur essendo state eseguite in un ambiente virtualizzato non completamente aderente alla realtà, hanno comunque evidenziato le

principali sfide e i benefici legati all’utilizzo di database distribuiti in scenari ad alta disponibilità e forte concorrenza.