

Architetture dati

Progetto MongoDB

Relazione di: Riccardo Andena 859643
Andrea Tirico 851958
Andrea Messa 856435

Anno Accademico 2022-2023

Indice

| | | |
|----------|---|-----------|
| 1 | Introduzione | 1 |
| 2 | Tecnologie utilizzate | 2 |
| 3 | Architettura | 4 |
| 3.1 | Gestione dell'architettura nel progetto | 5 |
| 4 | Distribuzione dei dati | 11 |
| 4.1 | Database | 11 |
| 4.2 | Creazione Network | 13 |
| 4.3 | Distribuzione Chunk | 15 |
| 5 | Test | 18 |
| 5.1 | Caricamento dei dati | 18 |
| 5.2 | Transazioni | 19 |
| 5.2.1 | Scenari | 23 |
| 5.3 | Spiegimento dei nodi | 42 |
| 5.3.1 | Spiegimento di un host | 42 |
| 5.3.2 | Spiegimento di un nodo | 42 |
| 5.3.3 | Spiegimento dei nodi in locale | 43 |
| 5.4 | Script | 44 |
| 5.4.1 | Script con eccezioni | 44 |
| 5.4.2 | Script per altri metodi | 46 |
| 6 | Problemi riscontrati | 50 |
| 6.1 | Problema rete | 50 |
| 6.2 | Problema firewall | 51 |
| 6.3 | Problema Virtual Machine | 51 |
| 7 | Conclusioni | 52 |

Capitolo 1

Introduzione

Il progetto si focalizza sulla verifica della documentazione di MongoDB, con particolare attenzione alla sezione dedicata alle transazioni e la verifica della gestione dei dati in caso di spegnimento di un nodo.

L'obiettivo principale consiste nel garantire che ogni aspetto descritto nella documentazione sia adeguatamente testato e che non si verifichino eventuali problematiche in tal senso.

Come tecnologia di containerizzazione è stato utilizzato Docker e per ogni container è stata installata l'ultima immagine disponibile di MongoDB in modo da poter creare nodi ed effettuare tutti i test richiesti. Nel contesto del progetto, è stato implementato un database dedicato alla registrazione degli utenti presso varie banche. L'utente che si registra fornisce le seguenti informazioni:

- Informazioni personali come nome, cognome e data di nascita.
- Nome della banca a cui è associato.
- Transazioni bancarie effettuate.

Attraverso questo database, si sono sviluppati dei casi di test per verificare le diverse operazioni descritte nella documentazione nella sezione relativa alle transazioni.

L'obiettivo è garantire che tutte le funzionalità e le iterazioni descritte siano correttamente implementate e rispettino i requisiti previsti.

Capitolo 2

Tecnologie utilizzate

Per la realizzazione di questo progetto, sono state utilizzate diverse tecnologie. In particolare:

- **Docker:** è una piattaforma open-source che consente la creazione, la distribuzione e l'esecuzione di applicazioni all'interno di container.

L'utilizzo di quest'ultimo offre diversi vantaggi. In primo luogo, semplifica la distribuzione delle applicazioni, in quanto i container possono essere eseguiti in modo coerente su diversi ambienti, come sviluppo, test e produzione, evitando così problemi di compatibilità. Inoltre, i container Docker sono altamente scalabili, consentendo di aumentare o ridurre rapidamente il numero di istanze in base alle esigenze di carico. Favorisce anche la modularità e la separazione delle responsabilità.

Infine, Docker offre un ambiente isolato per l'esecuzione delle applicazioni, garantendo che ciascun container abbia le proprie risorse isolate e non interferisca con altre applicazioni o risorse di sistema.

- **MongoDB:** è un database non relazionale ovvero NoSQL orientato ai documenti.

È progettato per gestire grandi quantità di dati eterogenei in modo flessibile e scalabile. A differenza dei database relazionali tradizionali, in cui i dati sono organizzati in tabelle con righe e colonne, MongoDB memorizza i dati come documenti JSON all'interno di collezioni. I documenti sono strutture dati flessibili e autosufficienti, che possono contenere campi di diversi tipi e non richiedono uno schema rigido predefinito. Questo rende MongoDB particolarmente adatto per applicazioni con esigenze di dati complessi o in continua evoluzione.

MongoDB supporta operazioni di inserimento, aggiornamento, query e cancellazione dei documenti in modo efficiente. Inoltre, offre funzionalità avanzate come gli indici per migliorare le prestazioni delle query, l'aggregazione dei dati per eseguire analisi complesse e la replica dei dati per garantire l'alta disponibilità e la ridondanza.

MongoDB è anche noto per la sua scalabilità orizzontale, consentendo di distribuire i dati su più server per gestire grandi volumi di informazioni e carichi di lavoro elevati.

- **MongoCompass:** è uno strumento di interfaccia grafica per MongoDB. È un'applicazione desktop che consente agli sviluppatori e agli amministratori di gestire e interagire con un database MongoDB in modo visuale e intuitivo.

Con questa interfaccia grafica, gli utenti possono esplorare le collezioni, visualizzare i documenti, eseguire query complesse e modificare i dati direttamente dall'interfaccia.

- **Visual Studio Code:** è un editor di codice sorgente leggero, altamente personalizzabile e potente, che supporta una vasta gamma di linguaggi di programmazione.

Grazie al suo ecosistema di estensioni, offre un'esperienza di sviluppo flessibile e adattabile alle esigenze degli sviluppatori. Le estensioni da noi scaricate per il progetto sono state:

1. Docker: per avere un panoramica e poter amministrare i container creati.
2. Remote Explorer: consente agli sviluppatori di connettersi e interagire con risorse remote come server, container, macchine virtuali e altro ancora, direttamente dall'editor.
In questo caso specifico, grazie all'utilizzo di una connessione SSH tramite l'editor, si è instaurata una connessione per lavorare direttamente sulla macchina virtuale fornita nel cloud.

- **Azure Virtual Machine:** è un servizio di infrastruttura cloud che consente di creare, configurare e gestire macchine virtuali basate su cloud per eseguire applicazioni e servizi. Offre flessibilità, scalabilità e affidabilità.

Nel nostro caso, l'utilizzo di Macchine Virtuali ci ha permesso di lavorare sotto la stessa sottorete. Uno dei vantaggi principali di questa scelta è che ci consente di semplificare la comunicazione e di evitare la necessità di configurare regole di firewall o di rete.

- **Python:** è un linguaggio di programmazione ad alto livello, interpretato e general-purpose.
Il suo obiettivo principale è quello di fornire una sintassi semplice e leggibile, favorendo la scrittura di codice chiaro e comprensibile.

Capitolo 3

Architettura

Per il progetto è stata utilizzata un'architettura di sharding ovvero un'architettura di database distribuita che consente di suddividere i dati tra più server o nodi, nei quali ci possono essere uno o più shard. Il termine shard intuitivamente si riferisce ad una partizione o ad una porzione dei dati complessivi.

L'obiettivo principale del sistema di sharding è consentire la scalabilità orizzontale dei database, ovvero la capacità di gestire volumi di dati sempre più grandi e di aumentare le prestazioni del sistema distribuendo il carico di lavoro su più nodi.

Nel contesto di MongoDB, un sistema di sharding viene implementato utilizzando i seguenti componenti:

- **Shard:** è un'istanza di MongoDB presente in un server o un set di replica, che garantisce l'affidabilità e l'alta disponibilità dei dati. I dati vengono distribuiti tra i diversi shard in base a un criterio di partizionamento, solitamente una chiave di shard.
- **Config Server:** sono responsabili della memorizzazione dei metadati del cluster di sharding. Contengono le informazioni sul mapping tra i dati e i relativi shard. Queste informazioni vengono utilizzate dal router per instradare le richieste verso gli shard corretti.
- **Router:** detto anche **Mongos**, è il componente che funge da intermediario tra le applicazioni e il cluster sharded. Riceve le richieste dalle applicazioni e le instrada verso gli shard appropriati in base alle informazioni di mapping presenti nei Config Server. Il router gestisce anche le operazioni di scrittura e lettura per garantire una distribuzione bilanciata del carico di lavoro tra gli shard.

Entrando nello specifico come detto in precedenza uno shard può essere:

- **Singolo Server:** indica che tutti i dati sono collocati su un unico server fisico o virtuale anziché su più server distribuiti. L'approccio ad un singolo server shard potrebbe essere utilizzato in uno scenario in cui le dimensioni dei dati o il carico di lavoro non richiedono la distribuzione su più server, questo può semplificare la configurazione e la gestione del database, specialmente per ambienti con un numero relativamente limitato di dati o utenti.
- **Set di Replica:** un gruppo di server che replicano gli stessi dati e collaborano per garantire la continuità del servizio in caso di guasti. Nel dettaglio una replica set è composta da:
 1. **Primary:** è il nodo attivo all'interno del replica set. Riceve tutte le operazioni di scrittura dal client e replica i dati agli altri membri. In caso di guasto del primary, uno dei membri del replica set viene eletto come nuovo primary.
 2. **Secondary:** sono i nodi di replica che replicano i dati dal primary. I secondary possono essere utilizzati per le operazioni di lettura, riducendo il carico sul primary e migliorando le prestazioni complessive del sistema. In caso di guasto del primary, uno dei secondary viene promosso a primary.

Quando un sistema di sharding è correttamente configurato, i dati vengono distribuiti tra i vari shard in modo equilibrato, consentendo una maggiore capacità di archiviazione e una migliore elaborazione delle richieste. Ciò consente di scalare il database in modo efficiente, gestendo volumi di dati molto grandi e supportando carichi di lavoro intensi.

3.1 Gestione dell'architettura nel progetto

L'architettura scelta per soddisfare tutti i requisiti del progetto è stata quella precedentemente definita.

In particolare, sono state coinvolte tre macchine virtuali che, attraverso un sistema di sharding, hanno distribuito in modo equo tutti i dati del nostro database.

Nello specifico ogni shard sarà una replica set di tre elementi dove avremo un nodo primario e due nodi secondari che fungeranno da replica.

I componenti che sono stati utilizzati nella nostra architettura sono:

- **Singolo Router:** situato in una singola macchina, che prenderà il nome di *VM Master*.
- **Config Server:** che sarà in replica set ed è situato su una singola macchina la *VM Master*.
- **Sette Shard:** che sono tutti replica set di tre nodi ciascuno che saranno distribuiti rispettivamente sulle tre macchine virtuali: *VM Master*, *VM WorkerA*, *VM WorkerT*.

Per illustrare meglio l'architettura del progetto segue in figura una rappresentazione visuale di quest'ultima.

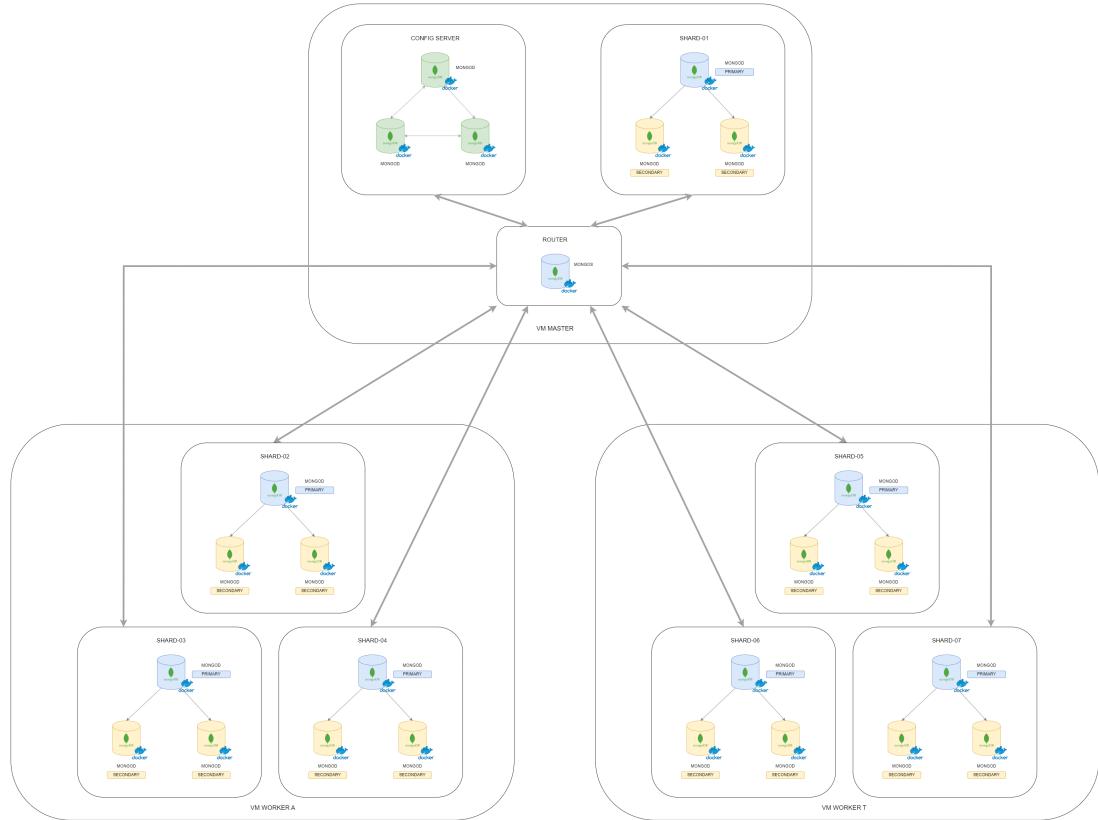


Figura 3.1: Architettura Progetto

Dopo aver definito l'architettura, ci si è dedicati alla scrittura del file YAML in cui risiedono tutte le configurazioni dei componenti che sono stati utilizzati.

Lo YAML è un linguaggio di serializzazione dei dati che può essere utilizzato per rappresentare informazioni strutturate come configurazioni, dati di configurazione, elenchi e dizionari. Il file di configurazione in formato YAML per l'orchestrazione della nostra infrastruttura MongoDB avrà le seguenti componenti:

- **Version:** "3.2", specifica la versione del formato di configurazione di Docker-Compose utilizzata.
- **Services:** questa sezione contiene la definizione dei vari servizi che costituiscono l'infrastruttura MongoDB che sono:
 1. **Router:** definisce il servizio per il router MongoDB (Mongos). Viene utilizzata come immagine del container *"mongo:latest"* la quale crea il container con l'ultima versione di MongoDB. Viene specificato anche il nome a cui si vuole associare il container *router-01*.

```

## Router
router01:
  image: mongo:latest
  container_name: router-01
  command: mongos --port 27017 --configdb rs-config-server/configsvr01:27017 --bind_ip_all
  ports:
    - 27117:27017 # sx host, dx container
  volumes:
    - ./scripts:/scripts
  deploy:
    placement:
      constraints:
        - node.hostname == master
  networks:
    - net_overlay

```

Figura 3.2: Router YAML

Entrando più nello specifico si può notare dalla figura precedente che vi sono alcuni comandi specifici che sono degni di nota.

Si può notare come la porta di destra sia quella del container e quella di sinistra quella interna dell'host *27117:27017*.

Viene definito anche il volume da montare, il quale si riferisce all'uso di volumi di Docker per gestire e condividere dati tra i container e l'host.

Nel nostro caso abbiamo: *./scripts:/scripts*, il quale indica che viene creato un volume Docker che collega la directory locale che nel nostro specifico caso è *./scripts* (parte di sinistra) sul sistema host alla directory */scripts* all'interno del container.

Questo significa che qualsiasi file o dato che è presente nella directory *./scripts* dell'host sarà accessibile e condivisibile dal container MongoDB.

Viene anche specificato il deploy del servizio e la rete di appartenenza che tratteremo nel capitolo successivo.

Un'altra sezione importante è quella indicata come *command*.

Grazie a questa sezione viene eseguita la stringa che attraverso il comando *mongos* avvia il router specificando la porta su cui il servizio router sarà in ascolto per le connessioni dei client ovvero *-port 27017*. Proseguendo nella stringa abbiamo il comando *-configdb rs-config-server/configsvr01:27017* che indica al servizio router dove trovare il set di configurazione dei server. Nel codice fornito, il set di configurazione viene specificato come *rs-config-server* e il server di configurazione primario viene indicato come *configsvr01* sulla porta 27017, infine si ha come parametro *-bind_ip_all* che indica al servizio router di legarsi a tutte le interfacce IP disponibili.

L'immagine successiva mostra il file JS che viene inizializzato ed eseguito, il quale aggiunge i relativi shard in connessione con il router.

```
sh.addShard("rs-shard-01/shard01-a:27017")
sh.addShard("rs-shard-01/shard01-b:27017")
sh.addShard("rs-shard-01/shard01-c:27017")

sh.addShard("rs-shard-02/shard02-a:27017")
sh.addShard("rs-shard-02/shard02-b:27017")
sh.addShard("rs-shard-02/shard02-c:27017")

sh.addShard("rs-shard-03/shard03-a:27017")
sh.addShard("rs-shard-03/shard03-b:27017")
sh.addShard("rs-shard-03/shard03-c:27017")

sh.addShard("rs-shard-04/shard04-a:27017")
sh.addShard("rs-shard-04/shard04-b:27017")
sh.addShard("rs-shard-04/shard04-c:27017")

sh.addShard("rs-shard-05/shard05-a:27017")
sh.addShard("rs-shard-05/shard05-b:27017")
sh.addShard("rs-shard-05/shard05-c:27017")

sh.addShard("rs-shard-06/shard06-a:27017")
sh.addShard("rs-shard-06/shard06-b:27017")
sh.addShard("rs-shard-06/shard06-c:27017")

sh.addShard("rs-shard-07/shard07-a:27017")
sh.addShard("rs-shard-07/shard07-b:27017")
sh.addShard("rs-shard-07/shard07-c:27017")
```

Figura 3.3: init-router

2. **configsvr01** e le sue repliche:

```
## Config Servers
configsvr01:
  image: mongo:latest
  container_name: mongo-config-01
  command: mongod --port 27017 --configsvr --replicaSet rs-config-server
  volumes:
    - ./scripts:/scripts
  ports:
    - 27118:27017
  deploy:
    placement:
      constraints:
        - node.hostname == master
  networks:
    - net_overlay

configsvr02:
  image: mongo:latest
  container_name: mongo-config-02
  command: mongod --port 27017 --configsvr --replicaSet rs-config-server
  volumes:
    - ./scripts:/scripts
  ports:
    - 27119:27017
  deploy:
    placement:
      constraints:
        - node.hostname == master
  networks:
    - net_overlay

configsvr03:
  image: mongo:latest
  container_name: mongo-config-03
  command: mongod --port 27017 --configsvr --replicaSet rs-config-server
  volumes:
    - ./scripts:/scripts
  ports:
    - 27120:27017
  deploy:
    placement:
      constraints:
        - node.hostname == master
  networks:
    - net_overlay
```

Figura 3.4: Config Server YAML

Anche in questo caso sono stato creato un file JS ah hoc per inizializzare ed eseguire config server e le sue repliche.

3. shard01 e le sue repliche:

```
## Shards 01
shard01-a:
  image: mongo:latest
  container_name: shard-01-node-a
  command: mongod --port 27017 --shardsvr --replicaSet rs-shard-01
  volumes:
    - ./scripts:/scripts
  ports:
    - 27122:27017
  deploy:
    placement:
      constraints:
        - node.hostname == master
  networks:
    - net_overlay
shard01-b:
  image: mongo:latest
  container_name: shard-01-node-b
  command: mongod --port 27017 --shardsvr --replicaSet rs-shard-01
  volumes:
    - ./scripts:/scripts
  ports:
    - 27123:27017
  deploy:
    placement:
      constraints:
        - node.hostname == master
  networks:
    - net_overlay
shard01-c:
  image: mongo:latest
  container_name: shard-01-node-c
  command: mongod --port 27017 --shardsvr --replicaSet rs-shard-01
  volumes:
    - ./scripts:/scripts
  ports:
    - 27124:27017
  deploy:
    placement:
      constraints:
        - node.hostname == master
  networks:
    - net_overlay
```

Figura 3.5: Shard YAML

Come si nota in figura, questa è la configurazione del servizio chiamato *Shard-01*.

La stessa configurazione verrà applicata anche per tutti gli altri shard che compongono l'architettura del nostro progetto.

Anche in questa situazione viene messo a disposizione un file JS per inizializzare ed eseguire tutti gli shard.

Capitolo 4

Distribuzione dei dati

Dopo la creazione dell’architettura, la fase successiva è la distribuzione dei dati, ovvero nella creazione di un database ad hoc e la distribuzione dei documenti nei vari shard. La creazione di una network è stato un altro passaggio importante che reso possibile la comunicazione tra shard, dato che prima quest’ultimi sotto reti diverse.

4.1 Database

Per quanto riguarda la creazione del database, grazie ad uno script in Python, si è costruita una collezione di un milione di utenti registrati in una ventina di banche.

Nell’immagine successiva viene mostrato l’albero degli attributi che struttura la classe user.

```
{'id': 1,  
 'name': 'Pippo',  
 'surname': 'Pluto',  
 'date_of_birth': '18/03/2000',  
 'bank': 'UniCredit',  
 'balance': '10000'}
```

Figura 4.1: Albero JSON

Come si può evincere dalla figura l’utente avrà degli attributi relativi a informazioni personali, il nome della banca a cui è associato e la giacenza sul conto corrente, la quale in fase di test verrà modificata a seconda se l’utente esegue una transazione in entrata oppure in uscita.

Per creare i dati del nostro database è stato utilizzato il package `random`, che ha permesso la scelta randomica di un elemento dalle liste statiche contenenti i nomi, cognomi, e banche. Dopo la scelta, questi elementi pescati, verranno caricati in una lista. Questa

operazione si è ripetuta per ogni iterazione del ciclo for, che prosegue fino a raggiungere un valore da noi predefinito, e può essere grande o piccolo. Infine, questa lista ottenuta verrà convertita in un file json, che sarà successivamente importato grazie al comando *mongoimport*. L'immagine successiva mostra tutto lo script per la generazione dei dati:

```
import json
import random

example = {
    "id": 1,
    "name": "Pippo",
    "surname": "Pluto",
    "date_of_birth": "18/03/2000",
    "bank": "UniCredit",
    "balance": "10000"
}

list_name = ['Sofia', 'Matteo', 'Giuseppe', 'Isabella', 'Federico', 'Giulia', 'Carlo', 'Veronica', '']
list_surname = ['Rossi', 'Bianchi', 'Russo', 'Esposito', 'Ricci', 'Ferrari', 'Rizzo', 'Romano', 'Con'
italian_banks = [
    "Intesa Sanpaolo",
    "UniCredit",
    "Banco BPM",
    "Monte dei Paschi di Siena",
    "UBI Banca",
    "Banca Nazionale del Lavoro",
    "Banca Generali",
    "Mediobanca",
    "Cassa Depositi e Prestiti",
    "Banca Popolare di Sondrio"
]
]

banks = []
type(banks)

list

for i in range(1, 1001):
    item = example.copy()

    rand_name = random.randint(0, 99)
    rand_year = random.randint(1960, 2004)
    rand_month = random.randint(1, 12)
    rand_day = random.randint(1, 31)
    rand_banca = random.randint(0,9)
    rand_balance = random.randint(1000, 1000000)

    item["id"] = i
    item["name"] = list_name[rand_name]
    item["surname"] = list_surname[rand_name]
    item["date_of_birth"] = f"{rand_day}/{rand_month}/{rand_year}"
    item["bank"] = italian_banks[rand_banca]
    item["balance"] = rand_balance

    banks.append(item)

with open("banks.json", "w") as outfile:
    json.dump(banks, outfile)

example

{'id': 1,
 'name': 'Pippo',
 'surname': 'Pluto',
 'date_of_birth': '18/03/2000',
 'bank': 'UniCredit',
 'balance': '10000'}
```

Figura 4.2: Script Python

4.2 Creazione Network

Per la creazione network ci siamo affidati a una rete **overlay**. Quest'ultima è una rete virtuale che consente alle diverse istanze distribuite di Docker di comunicare tra loro, come se fossero collegate direttamente sulla stessa rete locale.

Nel nostro caso viene utilizzata questa tipologia di rete per connettere appunto tutti gli shard e il router tra di loro anche se sono istanziati su macchine diverse, che ricordiamo sono *VM Master*, *VM WorkerA* e *VM WorkerT*.

Vi sono diversi passaggi da eseguire per creare una network overlay in Docker, questi sono:

1. Configurare un'infrastruttura di rete sottostante che supporti l'overlay, per fare ciò verrà utilizzato Docker Swarm, il quale permette di avere un'architettura scalabile per gestire i container distribuiti su più host.

Sotto vengono riportati i comandi per la creazione di un cluster con Docker Swarm:

- (a) Il comando che viene eseguito per inizializzare un cluster con Docker Swarm è il seguente: `docker swarm init`.

Questo comando viene eseguito su un nodo dell'architettura, il quale diventa *nodo manager*.

L'output del precedente comando è il seguente:

```
studente@master:~/Desktop/project_mongo/doc_mongo/project_mongoDB$ docker swarm init
Swarm initialized: current node (db8mif3gbcaa2z70341hv891ci) is now a manager.

To add a worker to this swarm, run the following command:
  docker swarm join --token SWMTKN-1-3pfq3wv0y28bwtu90zh1adokejemztnu82k7wmzylmhs80igh-600iog7maceud75hjqz6jomqf 10.0.0.26:2377
To add a manager to this swarm, run 'docker swarm join --token manager' and follow the instructions.
```

Figura 4.3: Docker Swarm Init

Si può notare notare come l'output restituisca un **token** il quale verrà utilizzato per connettere gli altri nodi alla rete overaly e un **IP manager** con la relativa porta di ingresso del nodo manager.

- (b) Il comando successivo che viene effettuato è:

`docker swarm join -token <token> <IP_manager>:<porta>`.

Questo comando viene eseguito su un nodo *worker* per unirsi al cluster Swarm utilizzando il token fornito durante l'inizializzazione del cluster e IP manager con relativa porta.

- (c) il comando: `docker node ls` permette di verificare quali macchine si sono unite attraverso la join alla rete overlay.

L'immagine successiva mostra come il nodo denominato *master* è il nodo leader della rete, mentre invece i nodi denominati *workerA* e *workerT* sono gli host che si sono collegati alla rete overlay.

| ID | HOSTNAME | STATUS | AVAILABILITY | MANAGER STATUS | ENGINE VERSION |
|-----------------------------|----------|--------|--------------|----------------|----------------|
| d8mif3gbcaauz70341hv891ci * | master | Ready | Active | Leader | 19.03.13 |
| n7pixew0jr7qzkz0frz4ywr9q | workerA | Ready | Active | | 19.03.13 |
| nfwdk1e652akr5t4z9yq1ftlj | workerT | Ready | Active | | 19.03.13 |

Figura 4.4: Docker Node ls

- La seconda operazione da effettuare è proprio la creazione di un network overlay.

Il comando utilizzato è il seguente:

`docker network create -driver overlay my-overlay-network`, dove *my-overlay-network* è il nome che si vuole assegnare alla rete overlay.

- L'operazione successiva consente di distribuire uno stack di servizi Docker utilizzando un file di composizione YAML, nel nostro caso quello citato nel precedente capitolo.

Il comando `docker stack deploy` legge il file di composizione YAML per determinare i servizi, le configurazioni e le dipendenze specificate.

Quindi, crea e avvia i servizi definiti all'interno dello stack sul cluster Swarm.

```
● studente@master:~/Desktop/project_mongo/doc_mongo/project_mongoDB$ docker stack deploy --compose-file docker-compose.yml app
Ignoring deprecated options:
container_name: Setting the container name is not supported.

Creating service app_shard02-b
Creating service app_configsvr02
Creating service app_shard05-a
Creating service app_shard01-c
Creating service app_shard06-b
Creating service app_shard03-b
Creating service app_shard07-c
Creating service app_shard04-b
Creating service app_shard01-a
Creating service app_shard06-c
Creating service app_shard07-a
Creating service app_shard03-c
Creating service app_shard05-c
Creating service app_shard04-c
Creating service app_shard07-b
Creating service app_shard04-a
Creating service app_configsvr01
Creating service app_router01
Creating service app_shard02-c
Creating service app_shard02-a
Creating service app_shard05-b
Creating service app_configsvr03
Creating service app_shard01-b
Creating service app_shard03-a
Creating service app_shard06-a
```

Figura 4.5: Docker stack deploy

Notiamo dall'immagine come attraverso l'utilizzo del deploy vengono creati i servizi definiti nel file YAML come router, shard e config server.

4. Attraverso l'uso del comando: `docker stack ps <app>` vengono distribuiti i nodi sugli host che appartengono alla rete overlay.

| ID | NAME | IMAGE | NODE | DESIRED STATE | CURRENT STATE | ERROR | PORTS |
|---------------|-------------------|--------------|---------|---------------|----------------------------|-------|-------|
| 01nxup32kjgp | app_configsrv01.1 | mongo:latest | master | Running | Running about a minute ago | | |
| Zodka8znvqb5 | app_configsrv02.1 | mongo:latest | master | Running | Running 2 minutes ago | | |
| r23pzkwk6puc9 | app_configsrv03.1 | mongo:latest | master | Running | Running about a minute ago | | |
| ur4d98dtpt1ax | app_router01.1 | mongo:latest | master | Running | Running about a minute ago | | |
| zkrykogyqv6 | app_shard01-a.1 | mongo:latest | master | Running | Running about a minute ago | | |
| j310nphrerpo | app_shard01-b.1 | mongo:latest | master | Running | Running about a minute ago | | |
| wi0gwdx6l7gb | app_shard01-c.1 | mongo:latest | master | Running | Running about a minute ago | | |
| gw2pt25zikb0 | app_shard02-a.1 | mongo:latest | workerA | Running | Running about a minute ago | | |
| rg19xnv1bzuk | app_shard02-b.1 | mongo:latest | workerA | Running | Running 2 minutes ago | | |
| qf7ferctkv1w | app_shard02-c.1 | mongo:latest | workerA | Running | Running about a minute ago | | |
| ptmuwwwu94hu | app_shard03-a.1 | mongo:latest | workerA | Running | Running about a minute ago | | |
| rf49okp79j0w | app_shard03-b.1 | mongo:latest | workerA | Running | Running about a minute ago | | |
| iv30w09m0jgs | app_shard03-c.1 | mongo:latest | workerA | Running | Running about a minute ago | | |
| l05b5j70v87h | app_shard04-a.1 | mongo:latest | workerA | Running | Running about a minute ago | | |
| ncl6pvup1dx7 | app_shard04-b.1 | mongo:latest | workerA | Running | Running about a minute ago | | |
| slgyr4sfagg3 | app_shard04-c.1 | mongo:latest | workerA | Running | Running about a minute ago | | |
| hp0vijqs96t | app_shard05-a.1 | mongo:latest | workerT | Running | Running about a minute ago | | |
| mpx3jidyqpw7 | app_shard05-b.1 | mongo:latest | workerT | Running | Running about a minute ago | | |
| kd20daise4ca | app_shard05-c.1 | mongo:latest | workerT | Running | Running about a minute ago | | |
| si2alds2tvsf | app_shard06-a.1 | mongo:latest | workerT | Running | Running about a minute ago | | |
| ui8dbslyc903 | app_shard06-b.1 | mongo:latest | workerT | Running | Running about a minute ago | | |
| mthwg4izewf | app_shard06-c.1 | mongo:latest | workerT | Running | Running about a minute ago | | |
| 2y36gin1ya27 | app_shard07-a.1 | mongo:latest | workerT | Running | Running about a minute ago | | |
| ubwb00c5mr1s | app_shard07-b.1 | mongo:latest | workerT | Running | Running about a minute ago | | |
| q2kjfp14kaaz | app_shard07-c.1 | mongo:latest | workerT | Running | Running about a minute ago | | |

Figura 4.6: Docker service ls

Il termine *app* indica il nome del servizio o dell'applicazione che si desidera distribuire utilizzando Docker Swarm.

4.3 Distribuzione Chunk

L'obiettivo di questa sezione è l'utilizzo della funzionalità di sharding di MongoDB, la quale suddivide i dati del database in chunk e li distribuisce su più nodi.

Con la parola chunk si intende una porzione di dati suddivisa in base a una chiave di sharding specifica, che in questo caso è *ID*.

E' stata scelta come chiave quest'ultima perché distribuisce in modo equo i dati del nostro database tra tutti gli shard.

L'obiettivo principale della suddivisione dei dati in chunk è consentire una distribuzione dei dati scalabile e bilanciata su più nodi shard.

In questo modo, MongoDB può gestire un elevato volume di dati e richieste, garantendo allo stesso tempo un'alta disponibilità e prestazioni ottimali.

La procedura utilizzata per distribuire in modo omogeneo il database su ogni shard che compone l'architettura è la seguente:

1. Il primo passo che si effettua è abilitare lo sharding del database che è *db_banks*. In questo modo con i passaggi successivi viene resa disponibile la possibilità di suddividere i dati in modo omogeneo nella nostra architettura. Il comando quindi che viene utilizzato è il seguente: `sh.enableSharding("db_banks")`
2. Come secondo step verrà abilitata la collezione che fa parte del database, in questo caso *banks*. In questo caso l'operazione viene effettuata per "shardare" i dati contenuti nella collezione.

Si definisce quale sarà la chiave che verrà prese in analisi che porterà a distribuire il database in modo equivalente. Avendo un database che è composto da utenti che sono registrati in banca, verrà usata come chiave *ID*.

Il comando utilizzato per effettuare questa operazione sarà:

```
sh.shardCollection("db_banks.banks", id : "hashed").
```

L'immagine successiva mostra la procedura adottata da terminale per effettuare le prime due operazioni.

```
[direct: mongos] test> sh.enableSharding("db_users")
{
  ok: 1,
  '$clusterTime': {
    clusterTime: Timestamp({ t: 1685530354, i: 466 }),
    signature: {
      hash: Binary(Buffer.from("00000000000000000000000000000000", "hex"), 0),
      keyId: Long("0")
    }
  },
  operationTime: Timestamp({ t: 1685530354, i: 464 })
}
[direct: mongos] test> sh.shardCollection("db_users.users", {"id":"hashed"})
{
  collectionsharded: 'db_users.users',
  ok: 1,
  '$clusterTime': {
    clusterTime: Timestamp({ t: 1685530434, i: 6 }),
    signature: {
      hash: Binary(Buffer.from("00000000000000000000000000000000", "hex"), 0),
      keyId: Long("0")
    }
  },
  operationTime: Timestamp({ t: 1685530434, i: 2 })
}
[direct: mongos] test> show dbs
admin      80.00 KiB
config     3.44 MiB
db_users   84.00 KiB
```

Figura 4.7: Enable Sharding

3. Il passo successivo è quello di caricare il database dentro l'infrastruttura.
Con il seguente comando soddisfiamo la richiesta: `mongoimport -jsonArray -db db_banks -collection banks -file banks.json`.
Possiamo notare dal comando che viene eseguito un *mongoimport*, ovvero un caricamento di documenti che avranno come formato *array*. Questi ultimi saranno caricati nel database *db_banks* e nella relativa collezione *banks*.

4. In conclusione per visualizzare se effettivamente il database è stato suddiviso in modo equivalente tra gli shard, si esegue il seguente comando:

`db_bank.getShardDistribution()`. L'immagine successiva mostra come l'architettura viene suddivisa nei vari chunk (partizioni) e mostra la percentuali di dati che sono contenuti nei relativi nodi.

```
---  
Totals  
{  
    data: '131.73MiB',  
    docs: 1000000,  
    chunks: 14,  
    'Shard rs-shard-02': [  
        '14.25 % data',  
        '14.25 % docs in cluster',  
        '138B avg obj size on shard'  
    ],  
    'Shard rs-shard-03': [  
        '14.33 % data',  
        '14.33 % docs in cluster',  
        '138B avg obj size on shard'  
    ],  
    'Shard rs-shard-05': [  
        '14.28 % data',  
        '14.28 % docs in cluster',  
        '138B avg obj size on shard'  
    ],  
    'Shard rs-shard-06': [  
        '14.32 % data',  
        '14.32 % docs in cluster',  
        '138B avg obj size on shard'  
    ],  
    'Shard rs-shard-04': [  
        '14.19 % data',  
        '14.19 % docs in cluster',  
        '138B avg obj size on shard'  
    ],  
    'Shard rs-shard-01': [  
        '14.34 % data',  
        '14.34 % docs in cluster',  
        '138B avg obj size on shard'  
    ],  
    'Shard rs-shard-07': [  
        '14.26 % data',  
        '14.26 % docs in cluster',  
        '138B avg obj size on shard'  
    ]  
}
```

Figura 4.8: Distribuzione Chunk

Dall'immagine si può notare come un milione di documenti vengono suddivisi in 14 chunk, due per ogni shard. Oltre a questo, si evince anche che in tutti gli shard i documenti sono suddivisi in modo equo, ognuno con una percentuale di 14.25% rispetto al totale.

Capitolo 5

Test

5.1 Caricamento dei dati

In questa prima parte si vuole testare il caricamento dei dati e valutare le tempistiche di caricamento. L'idea è aumentare progressivamente le dimensioni del file originale, al fine di ottenere indicazioni sul tempo necessario per il caricamento di possibili futuri file.

Nel corso di questo test, si procede inserendo un file sempre più ampio e si monitora il tempo richiesto per il caricamento. Per eseguire questa operazione, dalla shell del router, si utilizza il comando `mongoimport` per importare il dataset, che verrà distribuito tra i vari shard, garantendo una gestione scalabile dei dati.

Di seguito è presentato un grafico che illustra la progressione del tempo di caricamento dei dati nel corso del tempo..

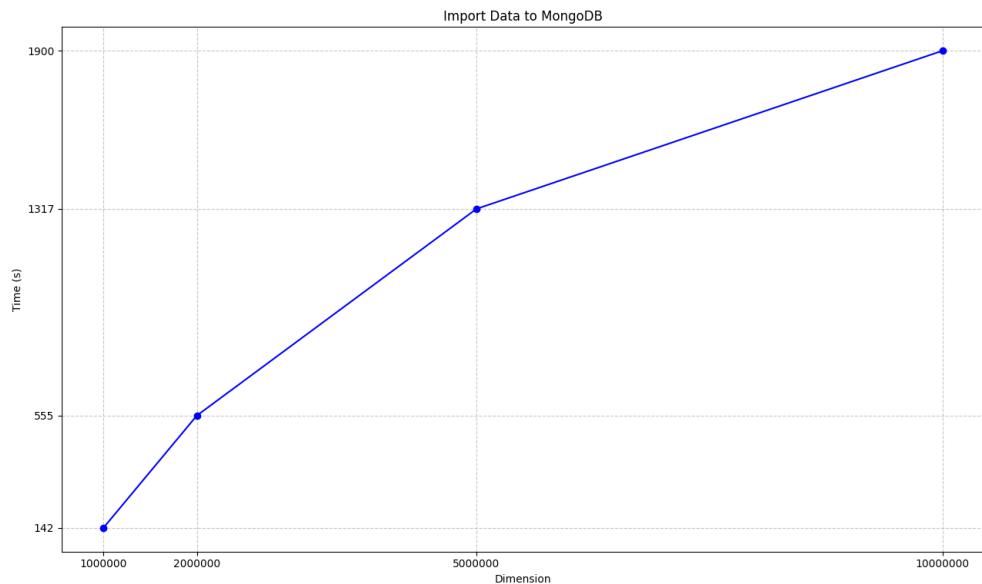


Figura 5.1: Mongo Import Plot

5.2 Transazioni

In MongoDB, le transazioni sono un meccanismo che permette di garantire la coerenza e l'integrità dei dati durante l'esecuzione di operazioni multiple all'interno di un singolo contesto transazionale. Le transazioni consentono di effettuare operazioni atomiche su più documenti, garantendo che tutte le modifiche vengano applicate in modo coerente e duraturo.

Durante questa fase di test, si esamineranno le transazioni al fine di valutare come varia la situazione in base al livello di isolamento utilizzato. Per condurre questo test, verranno impiegati due terminali, ognuno dei quali aprirà una transazione e potrà eseguire le seguenti operazioni atomiche:

- `find()`
- `update()`
- `insert()`
- `delete()`

Le transazioni in MongoDB seguono il modello ACID (Atomicità, Coerenza, Isolamento, Durabilità). Ciò significa che le transazioni sono **atomiche** (tutte le operazioni vengono eseguite o nessuna), **coerenti** (rispettano vincoli e regole di business), **isolate** (le transazioni non si influenzano a vicenda) e **durate** (le modifiche sono persistenti anche in caso di errore o riavvio del sistema).

Per quanto riguarda i vari livelli di isolamento possiamo distinguerne quattro:

- **Read Uncommitted:** in questo livello, una transazione può leggere dati non ancora confermati da altre transazioni. Ciò potrebbe comportare la lettura di dati sporchi o inconsistenti.
- **Read Committed:** in questo livello, una transazione può leggere solo dati che sono stati confermati da altre transazioni. Ciò garantisce che i dati letti siano coerenti, ma potrebbe comportare la lettura di versioni diverse dei dati durante una transazione.
- **Repeatable Read:** in questo livello, una transazione garantisce che i dati letti durante la transazione rimangano coerenti durante la transazione stessa. Ciò significa che se una transazione esegue più letture degli stessi dati, i dati non saranno modificati da altre transazioni fino alla fine della transazione corrente.
- **Serializable:** In questo livello, una transazione offre il massimo livello di isolamento. Garantisce che le transazioni si comportino come se fossero state eseguite in sequenza, una dopo l'altra, senza interferenze concorrenti. Ciò implica un blocco rigoroso e potrebbe influire sulle prestazioni in caso di carichi di lavoro pesanti.

Si può riassumere quanto detto nella Figura 5.2 in cui è mostrato per ogni livello il rapporto che ha tra concorrenza e isolamento.

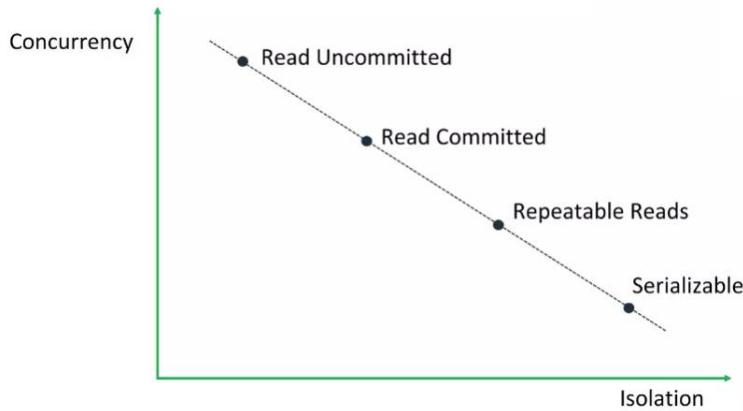


Figura 5.2: Livelli di isolamento.

Quando si lavora con transazioni in MongoDB, è importante considerare le potenziali anomalie che potrebbero verificarsi durante l'esecuzione di operazioni concorrenti. Queste anomalie possono compromettere l'integrità e la coerenza dei dati. I livelli di isolamento esposti prima possono essere utilizzati per affrontare tali problematiche.

Le anomalie che possono verificarsi sono:

- **Dirty Reads:** una transazione legge dati modificati da un'altra transazione che non ha ancora confermato le modifiche, come nell'esempio in figura 5.3.
- **Non-Repeatable Reads:** una transazione esegue più letture dello stesso dato durante l'esecuzione e i valori letti cambiano tra le letture a causa delle modifiche apportate da altre transazioni, come nell'esempio in figura 5.4.
- **Phantom Reads:** una transazione esegue una query che restituisce un set di risultati, ma durante l'esecuzione della transazione, un'altra transazione inserisce o elimina dati che soddisfano la condizione della query, risultando in un set di risultati diverso se la stessa query viene eseguita nuovamente all'interno della transazione, come nell'esempio in figura 5.5.

| T1 | T2 |
|--------------|--------------|
| START | |
| | START |
| Read → 1000 | |
| Write → -100 | |
| | Read → 900 |
| COMMIT | |
| | Write → -100 |
| | COMMIT |

Figura 5.3: Dirty Read

| T1 | T2 |
|-------------|--------------|
| START | |
| | START |
| Read → 1000 | |
| | Read → 1000 |
| | Write → -100 |
| | COMMIT |
| READ → 900 | |
| COMMIT | |

Figura 5.4: Non-repeatable Read

| T1 | T2 |
|--|-------------------------------------|
| START | |
| | START |
| Read → 9000 righe dove balance > 900.000 | |
| | Read → righe dove balance > 900.000 |
| | Write → balance di uno = 300 |
| | COMMIT |
| Read → 8999 righe dove balance > 900.000 | |
| COMMIT | |

Figura 5.5: Phantom Read

In 5.6 è mostrata una tabella in cui viene indicato per ogni livello di isolamento che tipo di anomalie risolve.

| | Dirty Reads | Non-Repeatable Reads | Phantom Reads |
|-------------------------|--------------------|-----------------------------|----------------------|
| Read Uncommitted | yes | yes | yes |
| Read Committed | no | yes | yes |
| Repeatable Reads | no | no | yes |
| Serializable | no | no | no |

Figura 5.6: Livelli di isolamento che risolvono le anomalie.

Read Concern

Per cambiare le impostazioni di default dell’isolamento vengono modificate la Read Concern e la Write Concern. La ReadConcern specifica il livello di coerenza dei dati richiesto durante un’operazione di lettura. Determina quali dati sono visibili durante la lettura, tenendo conto delle operazioni di scrittura in corso su quei dati. MongoDB supporta diversi livelli di ReadConcern:

- **local**: questo è il livello di Read Concern predefinito. Non garantisce la lettura di dati che siano stati scritti nella maggior parte dei membri del set di repliche.
- **available**: i dati possono essere letti da qualsiasi replica disponibile, inclusi i dati non ancora confermati dalla replica primaria. Per gli sharded cluster, Questa modalità fornisce le letture con la latenza più bassa possibile tra tutte le Read Concern. Tuttavia, ciò va a scapito della coerenza poiché può restituire documenti orfani durante la lettura da una collection frammentata.
- **majority**: in questa modalità i dati possono essere letti solo se sono stati replicati sulla maggioranza delle repliche nel set di replica. Ciò garantisce la coerenza dei dati nella maggior parte dei casi.
- **snapshot**: questa modalità assicura che la lettura includa solo le modifiche confermate al momento in cui è iniziata la lettura, escludendo le modifiche non ancora confermate da altre transazioni.

Write Concern

La Write Concern, specifica il numero di repliche che devono confermare un’operazione di scrittura prima che venga considerata completata. Determina la durabilità e la conferma dell’operazione di scrittura. Alcuni esempi di Write Concern includono:

- **w: 1**: questo è il Write Concern predefinito. Richiede la conferma dell’operazione di scrittura solo dalla replica primaria.
- **w: majority**: richiede la conferma dell’operazione di scrittura dalla maggioranza delle repliche nel set di replica, garantendo così una maggiore durabilità dei dati.
- **w: n**: richiede la conferma dell’operazione di scrittura da almeno n repliche nel set di replica (dove n è un numero che può andare da 1 al numero massimo di repliche).

5.2.1 Scenari

In questa fase di test abbiamo creato diversi scenari che evidenziano i pro e i contro di ogni livello di isolamento, impostato in base alla Read e Write Concern. Inoltre sono stati effettuati i seguenti passaggi:

- Su tutti e due i terminali abbiamo avviato una sessione col comando:
`var session = db.getMongo().startSession();`
- Dopo di che la collezione sotto quella sessioni viene caricata in una variabile:
`var collection = session.getDatabase("db_bank").getCollection("bank");`
- Si effettua la ricerca di un elemento con un certo id:
`collection.find(_id: "123")`
- Si esegue la modifica del valore della variabile balance:
`collection.updateOne(_id: "123" , $inc: balance: -100)`
- Infine viene effettuata la commit o l'abort con uno dei seguenti comandi:
`session.commitTransaction()`
`session.abortTransaction()`

Dopo aver completato tutti casi di test con le varie combinazioni di Read Concern e Write Concern per gestire i livelli di isolamento, ci si è resi conto che MongoDB non permette le varie anomalie Dirty read, Non-repeatable read e Phantom read. Infatti va a gestire automaticamente le varie anomalie andando a produrre errori nel caso in cui vengano sollevate.

Di seguito sono mostrati alcuni test in cui sono stati utilizzati due terminali e in ognuno di essi è stata aperta una sessione in cui sono state effettuate diverse operazioni per sollevare le anomalie. In tutti i casi effettuati con i diversi livelli di isolamento o è stato prodotto un errore che non permetteva di generare l'anomalia o non si verificava il problema analizzato.

RC: majority, WC: majority - Dirty Read

| Transazioni | Comandi |
|-------------|---|
| T1 | <pre>mm = db.getMongo() ss = mm.startSession({readConcern: 'majority', writeConcern: 'majority'}) ss.startTransaction({readConcern: 'majority', writeConcern: 'majority'}) collection = ss.getDatabase("db_bank").getCollection("bank") db_bank.bank</pre> |
| T2 | <pre>mm = db.getMongo() ss = mm.startSession({readConcern: 'majority', writeConcern: 'majority'}) ss.startTransaction({readConcern: 'majority', writeConcern: 'majority'}) collection = ss.getDatabase("db_bank").getCollection("bank") db_bank.bank</pre> |
| T1 | <pre>> collection.findOne({_id: ObjectId('64a958a37db95684b99cf284')}) < { _id: ObjectId("64a958a37db95684b99cf284"), id: 48, name: 'Mauro', surname: 'Silvestri', date_of_birth: '14/12/1974', bank: 'Mediobanca', balance: 574169 } collection.updateOne({_id: ObjectId('64a958a37db95684b99cf284')}, {'\$inc': {'balance': -100}) { acknowledged: true, insertedId: null, matchedCount: 1, modifiedCount: 1, upsertedCount: 0 }</pre> |
| T2 | <pre>collection.findOne({_id: ObjectId('64a958a37db95684b99cf284')}) { _id: ObjectId("64a958a37db95684b99cf284"), id: 48, name: 'Mauro', surname: 'Silvestri', date_of_birth: '14/12/1974', bank: 'Mediobanca', balance: 574169 }</pre> |
| T1 | <pre>ss.commitTransaction() { ok: 1, '\$clusterTime': { clusterTime: Timestamp({ t: 1688824237, i: 5 }), signature: { hash: Binary(Buffer.from("00000000000000000000000000000000", "hex"), 0), keyId: 0 } }, operationTime: Timestamp({ t: 1688824237, i: 5 }), recoveryToken: { recoveryShardId: 'rs-shard-02' } }</pre> |
| T2 | <pre>collection.updateOne({_id: ObjectId('64a958a37db95684b99cf284')}, {'\$inc': {'balance': -100}) * error from shard01:oid275517 during a transaction:: caused by :: WriteConflict error: this operation conflicted with another operation. Please retry your operation or multi-document transaction.</pre> |

RC: majority, WC: majority - Non repeatable read

| Transazioni | Comandi |
|-------------|--|
| T1 | <pre>mm = db.getMongo() ss = mm.startSession({readConcern: 'majority', writeConcern: 'majority'}) ss.startTransaction({readConcern: 'majority', writeConcern: 'majority'}) collection = ss.getDatabase("db_bank").getCollection("bank") db_bank.bank</pre> |
| T2 | <pre>mm = db.getMongo() ss = mm.startSession({readConcern: 'majority', writeConcern: 'majority'}) ss.startTransaction({readConcern: 'majority', writeConcern: 'majority'}) collection = ss.getDatabase("db_bank").getCollection("bank") db_bank.bank</pre> |
| T1 | <pre>collection.findOne({_id: ObjectId('64a958a37db95684b99cf284')}) { _id: ObjectId("64a958a37db95684b99cf284"), id: 48, name: 'Mauro', surname: 'Silvestri', date_of_birth: '14/12/1974', bank: 'Mediobanca', balance: 574069 }</pre> |
| T2 | <pre>collection.findOne({_id: ObjectId('64a958a37db95684b99cf284')}) { _id: ObjectId("64a958a37db95684b99cf284"), id: 48, name: 'Mauro', surname: 'Silvestri', date_of_birth: '14/12/1974', bank: 'Mediobanca', balance: 574069 } collection.updateOne({_id: ObjectId('64a958a37db95684b99cf26a')}, {'\$inc': {'balance': -100}}) { acknowledged: true, insertedId: null, matchedCount: 1, modifiedCount: 1, upsertedCount: 0 } ss.commitTransaction() { ok: 1, '\$clusterTime': { clusterTime: Timestamp({ t: 1688905228, i: 7 }), signature: { hash: Binary(Buffer.from("00000000000000000000000000000000", "hex"), 0), keyId: 0 } }, operationTime: Timestamp({ t: 1688905228, i: 7 }), recoveryToken: { recoveryShardId: 'rs-shard-01' } }</pre> |
| T1 | <pre>collection.findOne({_id: ObjectId('64a958a37db95684b99cf284')}) { _id: ObjectId("64a958a37db95684b99cf284"), id: 48, name: 'Mauro', surname: 'Silvestri', date_of_birth: '14/12/1974', bank: 'Mediobanca', balance: 574069 } ss.commitTransaction() { ok: 1, '\$clusterTime': { clusterTime: Timestamp({ t: 1688905243, i: 1 }), signature: { hash: Binary(Buffer.from("00000000000000000000000000000000", "hex"), 0), keyId: 0 } }, operationTime: Timestamp({ t: 1688905243, i: 1 }), recoveryToken: {} }</pre> |

RC: majority, WC: majority - Phantom read

| Transazioni | Comandi |
|-------------|---|
| T1 | <pre>mm = db.getMongo() ss = mm.startSession({readConcern: 'majority', writeConcern: 'majority'}) ss.startTransaction({readConcern: 'majority', writeConcern: 'majority'}) collection = ss.getDatabase("db_bank").getCollection("bank") db_bank.bank</pre> |
| T2 | <pre>mm = db.getMongo() ss = mm.startSession({readConcern: 'majority', writeConcern: 'majority'}) ss.startTransaction({readConcern: 'majority', writeConcern: 'majority'}) collection = ss.getDatabase("db_bank").getCollection("bank") db_bank.bank</pre> |
| T1 | <pre>collection.aggregate([{ \$match: {balance: {\$gt: 900000}}}, { \$group: {_id: null, count: {\$sum: 1}}}] { _id: null, count: 99960 }</pre> |
| T2 | <pre>collection.aggregate([{ \$match: {balance: {\$gt: 900000}}}, { \$group: {_id: null, count: {\$sum: 1}}}] { _id: null, count: 99960 } collection.updateOne({_id: ObjectId('64a958a37db95684b99cf2b1')}, {'\$set': {'balance': 100}} { acknowledged: true, insertedId: null, matchedCount: 1, modifiedCount: 0, upsertedCount: 0 } ss.commitTransaction() { ok: 1, '\$clusterTime': { clusterTime: Timestamp({ t: 1688823116, i: 9 }), signature: { hash: Binary(Buffer.from("00000000000000000000000000000000", "hex"), 0), keyId: 0 } }, operationTime: Timestamp({ t: 1688823116, i: 9 }), recoveryToken: { recoveryShardId: 'rs-shard-01' } }</pre> |
| T1 | <pre>collection.aggregate([{ \$match: {balance: {\$gt: 900000}}}, { \$group: {_id: null, count: {\$sum: 1}}}] { _id: null, count: 99960 } ss.commitTransaction() { ok: 1, '\$clusterTime': { clusterTime: Timestamp({ t: 1688823131, i: 1 }), signature: { hash: Binary(Buffer.from("00000000000000000000000000000000", "hex"), 0), keyId: 0 } }, operationTime: Timestamp({ t: 1688823131, i: 1 }), recoveryToken: {} }</pre> |

RC: majority, WC: 1 - Dirty Read

| Transazioni | Comandi |
|-------------|---|
| T1 | <pre>mm = db.getMongo() ss = mm.startSession({readConcern: 'majority', writeConcern: 1}) ss.startTransaction({readConcern: 'majority', writeConcern: 1}) collection = ss.getDatabase("db_bank").getCollection("bank") db_bank.bank</pre> |
| T2 | <pre>mm = db.getMongo() ss = mm.startSession({readConcern: 'majority', writeConcern: 1}) ss.startTransaction({readConcern: 'majority', writeConcern: 1}) collection = ss.getDatabase("db_bank").getCollection("bank") db_bank.bank</pre> |
| T1 | <pre>> collection.findOne({_id: ObjectId('64a958a37db95684b99cf284')}) < _id: ObjectId("64a958a37db95684b99cf284"), id: 48, name: 'Mauro', surname: 'Silvestri', date_of_birth: '14/12/1974', bank: 'Mediobanca', balance: 574169 } collection.updateOne({_id: ObjectId('64a958a37db95684b99cf284')}, {'\$inc': {'balance': -100}} { acknowledged: true, insertedId: null, matchedCount: 1, modifiedCount: 1, upsertedCount: 0 }</pre> |
| T2 | <pre>collection.findOne({_id: ObjectId('64a958a37db95684b99cf284')}) { _id: ObjectId("64a958a37db95684b99cf284"), id: 48, name: 'Mauro', surname: 'Silvestri', date_of_birth: '14/12/1974', bank: 'Mediobanca', balance: 574169 }</pre> |
| T1 | <pre>ss.commitTransaction() { ok: 1, '\$clusterTime': { clusterTime: Timestamp({ t: 1688824237, i: 5 }), signature: { hash: Binary(Buffer.from("00000000000000000000000000000000", "hex"), 0), keyId: 0 } }, operationTime: Timestamp({ t: 1688824237, i: 5 }), recoveryToken: { recoveryShardId: 'rs-shard-02' } }</pre> |
| T2 | <pre>collection.updateOne({_id: ObjectId('64a958a37db95684b99cf284')}, {'\$inc': {'balance': -100}}) > E1 from shard01-shard01[0] during a transaction :: caused by :: WriteConflict error: this operation conflicted with another operation. Please retry your operation or multi-document transaction</pre> |

RC: majority, WC: 1 - Non repeatable read

| Transazioni | Comandi |
|-------------|--|
| T1 | <pre>mm = db.getMongo() ss = mm.startSession({readConcern: 'majority', writeConcern: 1}) ss.startTransaction({readConcern: 'majority', writeConcern: 1}) collection = ss.getDatabase("db_bank").getCollection("bank") db_bank.bank</pre> |
| T2 | <pre>mm = db.getMongo() ss = mm.startSession({readConcern: 'majority', writeConcern: 1}) ss.startTransaction({readConcern: 'majority', writeConcern: 1}) collection = ss.getDatabase("db_bank").getCollection("bank") db_bank.bank</pre> |
| T1 | <pre>collection.findOne({_id: ObjectId('64a958a37db95684b99cf284')}) { _id: ObjectId("64a958a37db95684b99cf284"), id: 48, name: 'Mauro', surname: 'Silvestri', date_of_birth: '14/12/1974', bank: 'Mediobanca', balance: 574069 }</pre> |
| T2 | <pre>collection.findOne({_id: ObjectId('64a958a37db95684b99cf284')}) { _id: ObjectId("64a958a37db95684b99cf284"), id: 48, name: 'Mauro', surname: 'Silvestri', date_of_birth: '14/12/1974', bank: 'Mediobanca', balance: 574069 } collection.updateOne({_id: ObjectId('64a958a37db95684b99cf26a')}, {'\$inc': {'balance': -100}}) { acknowledged: true, insertedId: null, matchedCount: 1, modifiedCount: 1, upsertedCount: 0 } ss.commitTransaction() { ok: 1, '\$clusterTime': { clusterTime: Timestamp({ t: 1688905228, i: 7 }), signature: { hash: Binary(Buffer.from("00000000000000000000000000000000", "hex"), 0), keyId: 0 } }, operationTime: Timestamp({ t: 1688905228, i: 7 }), recoveryToken: { recoveryShardId: 'rs-shard-01' } }</pre> |
| T1 | <pre>collection.findOne({_id: ObjectId('64a958a37db95684b99cf284')}) { _id: ObjectId("64a958a37db95684b99cf284"), id: 48, name: 'Mauro', surname: 'Silvestri', date_of_birth: '14/12/1974', bank: 'Mediobanca', balance: 574069 } ss.commitTransaction() { ok: 1, '\$clusterTime': { clusterTime: Timestamp({ t: 1688905243, i: 1 }), signature: { hash: Binary(Buffer.from("00000000000000000000000000000000", "hex"), 0), keyId: 0 } }, operationTime: Timestamp({ t: 1688905243, i: 1 }), recoveryToken: {} }</pre> |

RC: majority, WC: 1 - Phantom read

| Transazioni | Comandi |
|-------------|--|
| T1 | <pre>mm = db.getMongo() ss = mm.startSession({readConcern: 'majority', writeConcern: 1}) ss.startTransaction({readConcern: 'majority', writeConcern: 1}) collection = ss.getDatabase("db_bank").getCollection("bank") db_bank.bank</pre> |
| T2 | <pre>mm = db.getMongo() ss = mm.startSession({readConcern: 'majority', writeConcern: 1}) ss.startTransaction({readConcern: 'majority', writeConcern: 1}) collection = ss.getDatabase("db_bank").getCollection("bank") db_bank.bank</pre> |
| T1 | <pre>collection.aggregate([{\$match: {balance: {\$gt: 900000}}}, {\$group: {_id: null, count: {\$sum: 1}})]) { _id: null, count: 99960 }</pre> |
| T2 | <pre>collection.aggregate([{\$match: {balance: {\$gt: 900000}}}, {\$group: {_id: null, count: {\$sum: 1}})]) { _id: null, count: 99960 } collection.updateOne({_id: ObjectId('64a958a37db95684b99cf4e6')}, {'\$set': {'balance': 300}}) { acknowledged: true, insertedId: null, matchedCount: 1, modifiedCount: 1, upsertedCount: 0 } ss.commitTransaction() { ok: 1, '\$clusterTime': { clusterTime: Timestamp({ t: 1688906823, i: 6 }), signature: { hash: Binary(Buffer.from("00000000000000000000000000000000", "hex"), 0), keyId: 0 } }, operationTime: Timestamp({ t: 1688906823, i: 6 }), recoveryToken: { recoveryShardId: 'rs-shard-01' } }</pre> |
| T1 | <pre>collection.aggregate([{\$match: {balance: {\$gt: 900000}}}, {\$group: {_id: null, count: {\$sum: 1}})]) { _id: null, count: 99960 } ss.commitTransaction() { ok: 1, '\$clusterTime': { clusterTime: Timestamp({ t: 1688906836, i: 1 }), signature: { hash: Binary(Buffer.from("00000000000000000000000000000000", "hex"), 0), keyId: 0 } }, operationTime: Timestamp({ t: 1688906836, i: 1 }), recoveryToken: {} }</pre> |

RC: local, WC: majority - Dirty Read

| Transazioni | Comandi |
|-------------|--|
| T1 | <pre>mm = db.getMongo() ss = mm.startSession({readConcern: 'local', writeConcern: 'majority'}) ss.startTransaction({readConcern: 'local', writeConcern: 'majority'}) collection = ss.getDatabase("db_bank").getCollection("bank") db_bank.bank</pre> |
| T2 | <pre>mm = db.getMongo() ss = mm.startSession({readConcern: 'local', writeConcern: 'majority'}) ss.startTransaction({readConcern: 'local', writeConcern: 'majority'}) collection = ss.getDatabase("db_bank").getCollection("bank") db_bank.bank</pre> |
| T1 | <pre>collection.findOne({_id: ObjectId('64a958a37db95684b99cf284')}) { _id: ObjectId("64a958a37db95684b99cf284"), id: 48, name: 'Mauro', surname: 'Silvestri', date_of_birth: '14/12/1974', bank: 'Mediobanca', balance: 574369 } collection.updateOne({_id: ObjectId('64a958a37db95684b99cf284')}, {'\$inc': {'balance': -100}) { acknowledged: true, insertedId: null, matchedCount: 1, modifiedCount: 1, upsertedCount: 0 }</pre> |
| T2 | <pre>collection.findOne({_id: ObjectId('64a958a37db95684b99cf284')}) { _id: ObjectId("64a958a37db95684b99cf284"), id: 48, name: 'Mauro', surname: 'Silvestri', date_of_birth: '14/12/1974', bank: 'Mediobanca', balance: 574369 }</pre> |
| T1 | <pre>ss.commitTransaction() { ok: 1, '\$clusterTime': { clusterTime: Timestamp({ t: 1688898863, i: 6 }), signature: { hash: Binary(Buffer.from("00000000000000000000000000000000", "hex"), 0), keyId: 0 } }, operationTime: Timestamp({ t: 1688898863, i: 6 }), recoveryToken: { recoveryShardId: 'rs-shard-01' } }</pre> |
| T2 | <pre>collection.updateOne({_id: ObjectId('64a958a37db95684b99cf284')}, {'\$inc': {'balance': -100}) * Error shard01:rs0:2017 during a transaction or caused by a WriteConflict error: this operation conflicted with another operation. Please retry your operation or multi-document transaction.</pre> |

RC: local, WC: majority - Non repeatable read

| Transazioni | Comandi |
|-------------|---|
| T1 | <pre>mm = db.getMongo() ss = mm.startSession({readConcern: 'local', writeConcern: 'majority'}) ss.startTransaction({readConcern: 'local', writeConcern: 'majority'}) collection = ss.getDatabase("db_bank").getCollection("bank") db_bank.bank</pre> |
| T2 | <pre>mm = db.getMongo() ss = mm.startSession({readConcern: 'local', writeConcern: 'majority'}) ss.startTransaction({readConcern: 'local', writeConcern: 'majority'}) collection = ss.getDatabase("db_bank").getCollection("bank") db_bank.bank</pre> |
| T1 | <pre>collection.findOne({_id: ObjectId('64a958a37db95684b99cf284')}) { _id: ObjectId("64a958a37db95684b99cf284"), id: 48, name: 'Mauro', surname: 'Silvestri', date_of_birth: '14/12/1974', bank: 'Mediobanca', balance: 574269 }</pre> |
| T2 | <pre>collection.findOne({_id: ObjectId('64a958a37db95684b99cf284')}) { _id: ObjectId("64a958a37db95684b99cf284"), id: 48, name: 'Mauro', surname: 'Silvestri', date_of_birth: '14/12/1974', bank: 'Mediobanca', balance: 574269 } collection.updateOne({_id: ObjectId('64a958a37db95684b99cf284'), {"\$inc": {"balance": -100}}) { acknowledged: true, insertedId: null, matchedCount: 1, modifiedCount: 1, upsertedCount: 0 } ss.commitTransaction() { ok: 1, '\$clusterTime': { clusterTime: Timestamp({ t: 1688899679, i: 6 }), signature: { hash: Binary(Buffer.from("00000000000000000000000000000000", "hex"), 0), keyId: 0 } }, operationTime: Timestamp({ t: 1688899679, i: 6 }), recoveryToken: { recoveryShardId: 'rs-shard-01' } }</pre> |
| T1 | <pre>collection.findOne({_id: ObjectId('64a958a37db95684b99cf284')}) { _id: ObjectId("64a958a37db95684b99cf284"), id: 48, name: 'Mauro', surname: 'Silvestri', date_of_birth: '14/12/1974', bank: 'Mediobanca', balance: 574269 } ss.commitTransaction() { ok: 1, '\$clusterTime': { clusterTime: Timestamp({ t: 1688899690, i: 1 }), signature: { hash: Binary(Buffer.from("00000000000000000000000000000000", "hex"), 0), keyId: 0 } }, operationTime: Timestamp({ t: 1688899690, i: 1 }), recoveryToken: {} }</pre> |

RC: local, WC: majority - Phantom read

| Transazioni | Comandi |
|-------------|--|
| T1 | <pre>mm = db.getMongo() ss = mm.startSession({readConcern: 'local', writeConcern: 'majority'}) ss.startTransaction({readConcern: 'local', writeConcern: 'majority'}) collection = ss.getDatabase("db_bank").getCollection("bank") db_bank.bank</pre> |
| T2 | <pre>mm = db.getMongo() ss = mm.startSession({readConcern: 'local', writeConcern: 'majority'}) ss.startTransaction({readConcern: 'local', writeConcern: 'majority'}) collection = ss.getDatabase("db_bank").getCollection("bank") db_bank.bank</pre> |
| T1 | <pre>collection.aggregate([{\$match: {balance: {\$gt: 900000}}}, {\$group: {_id: null, count: {\$sum: 1}})]) { _id: null, count: 99960 }</pre> |
| T2 | <pre>collection.aggregate([{\$match: {balance: {\$gt: 900000}}}, {\$group: {_id: null, count: {\$sum: 1}})]) { _id: null, count: 99960 } collection.updateOne({_id: ObjectId('64a958a37db95684b99cf2b1')}, {'\$set': {'balance': 300}}) { acknowledged: true, insertedId: null, matchedCount: 1, modifiedCount: 1, upsertedCount: 0 } ss.commitTransaction() { ok: 1, '\$clusterTime': { clusterTime: Timestamp({ t: 1688900218, i: 6 }), signature: { hash: Binary(Buffer.from("00000000000000000000000000000000", "hex"), 0), keyId: 0 } }, operationTime: Timestamp({ t: 1688900218, i: 6 }), recoveryToken: { recoveryShardId: 'rs-shard-01' } }</pre> |
| T1 | <pre>collection.aggregate([{\$match: {balance: {\$gt: 900000}}}, {\$group: {_id: null, count: {\$sum: 1}})]) { _id: null, count: 99960 } ss.commitTransaction() { ok: 1, '\$clusterTime': { clusterTime: Timestamp({ t: 1688900229, i: 2 }), signature: { hash: Binary(Buffer.from("00000000000000000000000000000000", "hex"), 0), keyId: 0 } }, operationTime: Timestamp({ t: 1688900229, i: 2 }), recoveryToken: {} }</pre> |

RC: local, WC: 1 - Dirty Read

| Transazioni | Comandi |
|-------------|---|
| T1 | <pre>mm = db.getMongo() ss = mm.startSession({readConcern: 'local', writeConcern: '1'}) ss.startTransaction({readConcern: 'local', writeConcern: '1'}) collection = ss.getDatabase("db_bank").getCollection("bank") db_bank.bank</pre> |
| T2 | <pre>mm = db.getMongo() ss = mm.startSession({readConcern: 'local', writeConcern: '1'}) ss.startTransaction({readConcern: 'local', writeConcern: '1'}) collection = ss.getDatabase("db_bank").getCollection("bank") db_bank.bank</pre> |
| T1 | <pre>collection.findOne({_id: ObjectId('64a5cccd605967ed9e5813d1f')}) { _id: ObjectId("64a5cccd605967ed9e5813d1f"), id: 15, name: 'Caterina', surname: 'Moretti', date_of_birth: '23/5/1990', bank: 'UBI Banca', balance: 12882 } collection.updateOne({_id: ObjectId('64a5cccd605967ed9e5813d1f')}, {'\$inc': {'balance': -100}) { acknowledged: true, insertedId: null, matchedCount: 1, modifiedCount: 1, upsertedCount: 0 }</pre> |
| T2 | <pre>collection.findOne({_id: ObjectId('64a5cccd605967ed9e5813d1f')}) { _id: ObjectId("64a5cccd605967ed9e5813d1f"), id: 15, name: 'Caterina', surname: 'Moretti', date_of_birth: '23/5/1990', bank: 'UBI Banca', balance: 12882 }</pre> |
| T1 | <pre>ss.commitTransaction() { ok: 1, '\$clusterTime': { clusterTime: Timestamp({ t: 1688814500, i: 7 }), signature: { hash: Binary(Buffer.from("00000000000000000000000000000000", "hex"), 0), keyId: 0 } }, operationTime: Timestamp({ t: 1688814500, i: 7 }), recoveryToken: { recoveryShardId: 'rs-shard-02' } }</pre> |
| T2 | <pre>collection.updateOne({_id: ObjectId('64a5cccd605967ed9e5813d1f')}, {'\$inc': {'balance': -100}) + MongoDBError[error] Encountered error from shard02-a:27017 during a transaction :: caused by :: WriteConflict error: this operation conflicted with another operation. Please retry your operati</pre> |

RC: local, WC: 1 - Non repeatable read

| Transazioni | Comandi |
|-------------|---|
| T1 | <pre>mm = db.getMongo() ss = mm.startSession({readConcern: 'local', writeConcern: '1'}) ss.startTransaction({readConcern: 'local', writeConcern: '1'}) collection = ss.getDatabase("db_bank").getCollection("bank") db_bank.bank</pre> |
| T2 | <pre>mm = db.getMongo() ss = mm.startSession({readConcern: 'local', writeConcern: '1'}) ss.startTransaction({readConcern: 'local', writeConcern: '1'}) collection = ss.getDatabase("db_bank").getCollection("bank") db_bank.bank</pre> |
| T1 | <pre>collection.findOne({_id: ObjectId('64a958a37db95684b99cf26a')}) { _id: ObjectId("64a958a37db95684b99cf26a"), id: 22, name: 'Francesca', surname: 'Martini', date_of_birth: '14/2/2003', bank: 'Cassa Depositi e Prestiti', balance: 537374 }</pre> |
| T2 | <pre>collection.findOne({_id: ObjectId('64a958a37db95684b99cf26a')}) { _id: ObjectId("64a958a37db95684b99cf26a"), id: 22, name: 'Francesca', surname: 'Martini', date_of_birth: '14/2/2003', bank: 'Cassa Depositi e Prestiti', balance: 537374 } collection.updateOne({_id: ObjectId('64a958a37db95684b99cf26a'), {'\$inc': {'balance': -100}}) { acknowledged: true, insertedId: null, matchedCount: 1, modifiedCount: 1, upsertedCount: 0 } ss.commitTransaction() { ok: 1, '\$clusterTime': { clusterTime: Timestamp({ t: 1688820350, i: 5 }), signature: { hash: Binary(Buffer.from("00000000000000000000000000000000", "hex"), 0), keyId: 0 } }, operationTime: Timestamp({ t: 1688820350, i: 5 }), recoveryToken: { recoveryShardId: 'rs-shard-02' } }</pre> |
| T1 | <pre>collection.findOne({_id: ObjectId('64a958a37db95684b99cf26a')}) { _id: ObjectId("64a958a37db95684b99cf26a"), id: 22, name: 'Francesca', surname: 'Martini', date_of_birth: '14/2/2003', bank: 'Cassa Depositi e Prestiti', balance: 537374 } ss.commitTransaction() { ok: 1, '\$clusterTime': { clusterTime: Timestamp({ t: 1688820370, i: 1 }), signature: { hash: Binary(Buffer.from("00000000000000000000000000000000", "hex"), 0), keyId: 0 } }, operationTime: Timestamp({ t: 1688820370, i: 1 }), recoveryToken: {} }</pre> |

RC: local, WC: 1 - Phantom read

| Transazioni | Comandi |
|-------------|--|
| T1 | <pre>mm = db.getMongo() ss = mm.startSession({readConcern: 'local', writeConcern: '1'}) ss.startTransaction({readConcern: 'local', writeConcern: '1'}) collection = ss.getDatabase("db_bank").getCollection("bank") db_bank.bank</pre> |
| T2 | <pre>mm = db.getMongo() ss = mm.startSession({readConcern: 'local', writeConcern: '1'}) ss.startTransaction({readConcern: 'local', writeConcern: '1'}) collection = ss.getDatabase("db_bank").getCollection("bank") db_bank.bank</pre> |
| T1 | <pre>collection.aggregate([{\$match: {balance: {\$gt: 900000}}}, {\$group: {_id: null, count: {\$sum: 1}}}}) { _id: null, count: 99961 }</pre> |
| T2 | <pre>collection.aggregate([{\$match: {balance: {\$gt: 900000}}}, {\$group: {_id: null, count: {\$sum: 1}}}}) { _id: null, count: 99961 } collection.updateOne({_id: ObjectId('64a958a37db95684b99cf2b1')}, {\$set: {"balance": 100}}) { acknowledged: true, insertedId: null, matchedCount: 1, modifiedCount: 1, upsertedCount: 0 } ss.commitTransaction() { ok: 1, '\$clusterTime': { clusterTime: Timestamp({t: 1688823116, i: 9}), signature: { hash: Binary(Buffer.from("00000000000000000000000000000000", "hex"), 0), keyId: 0 } }, operationTime: Timestamp({t: 1688823116, i: 9}), recoveryToken: {recoveryShardId: 'rs-shard-01'} }</pre> |
| T1 | <pre>collection.aggregate([{\$match: {balance: {\$gt: 900000}}}, {\$group: {_id: null, count: {\$sum: 1}}}}) { _id: null, count: 99961 } ss.commitTransaction() { ok: 1, '\$clusterTime': { clusterTime: Timestamp({t: 1688823131, i: 1}), signature: { hash: Binary(Buffer.from("00000000000000000000000000000000", "hex"), 0), keyId: 0 } }, operationTime: Timestamp({t: 1688823131, i: 1}), recoveryToken: {} }</pre> |

RC: snapshot, WC: 1 - Dirty Read

| Transazioni | Comandi |
|-------------|--|
| T1 | <pre>mm = db.getMongo() ss = mm.startSession({readConcern: 'snapshot', writeConcern: 1}) ss.startTransaction({readConcern: 'snapshot', writeConcern: 1}) collection = ss.getDatabase("db_bank").getCollection("bank") db_bank.bank</pre> |
| T2 | <pre>mm = db.getMongo() ss = mm.startSession({readConcern: 'snapshot', writeConcern: 1}) ss.startTransaction({readConcern: 'snapshot', writeConcern: 1}) collection = ss.getDatabase("db_bank").getCollection("bank") db_bank.bank</pre> |
| T1 | <pre>collection.findOne({_id: ObjectId('64a958a37db95684b99cf284')}) { _id: ObjectId("64a958a37db95684b99cf284"), id: 48, name: 'Mauro', surname: 'Silvestri', date_of_birth: '14/12/1974', bank: 'Mediobanca', balance: 573969 } collection.updateOne({_id: ObjectId('64a958a37db95684b99cf284')}, {'\$inc': {'balance': -100}) { acknowledged: true, insertedId: null, matchedCount: 1, modifiedCount: 1, upsertedCount: 0 }</pre> |
| T2 | <pre>collection.findOne({_id: ObjectId('64a958a37db95684b99cf284')}) { _id: ObjectId("64a958a37db95684b99cf284"), id: 48, name: 'Mauro', surname: 'Silvestri', date_of_birth: '14/12/1974', bank: 'Mediobanca', balance: 573969 }</pre> |
| T1 | <pre>ss.commitTransaction() { ok: 1, '\$clusterTime': { clusterTime: Timestamp({ t: 1688918986, i: 5 }), signature: { hash: Binary(Buffer.from("00000000000000000000000000000000", "hex"), 0), keyId: 0 }, operationTime: Timestamp({ t: 1688918986, i: 5 }), recoveryToken: { recoveryShardId: 'rs-shard-01' } }</pre> |
| T2 | <pre>collection.updateOne({_id: ObjectId('64a958a37db95684b99cf284')}, {'\$inc': {'balance': -100}) MonodatabaseError: Encountered error from shard01-c12701 during a transaction :: caused by :: WriteConflict error: this operation conflicted with another operation. Please retry your update!</pre> |

RC: snapshot, WC: 1 - Non repeatable read

| Transazioni | Comandi |
|-------------|---|
| T1 | <pre>mm = db.getMongo() ss = mm.startSession({readConcern: 'snapshot', writeConcern: 1}) ss.startTransaction({readConcern: 'snapshot', writeConcern: 1}) collection = ss.getDatabase("db_bank").getCollection("bank") db_bank.bank</pre> |
| T2 | <pre>mm = db.getMongo() ss = mm.startSession({readConcern: 'snapshot', writeConcern: 1}) ss.startTransaction({readConcern: 'snapshot', writeConcern: 1}) collection = ss.getDatabase("db_bank").getCollection("bank") db_bank.bank</pre> |
| T1 | <pre>collection.findOne({_id: ObjectId('64a958a37db95684b99cf284')}) { _id: ObjectId("64a958a37db95684b99cf284"), id: 48, name: 'Mauro', surname: 'Silvestri', date_of_birth: '14/12/1974', bank: 'Mediobanca', balance: 573869 }</pre> |
| T2 | <pre>collection.findOne({_id: ObjectId('64a958a37db95684b99cf284')}) { _id: ObjectId("64a958a37db95684b99cf284"), id: 48, name: 'Mauro', surname: 'Silvestri', date_of_birth: '14/12/1974', bank: 'Mediobanca', balance: 573869 } collection.updateOne({_id: ObjectId('64a958a37db95684b99cf284')}, {'\$inc': {'balance': -100}) { acknowledged: true, insertedId: null, matchedCount: 1, modifiedCount: 1, upsertedCount: 0 } ss.commitTransaction() { ok: 1, '\$clusterTime': { clusterTime: Timestamp({ t: 1688919579, i: 6 }), signature: { hash: Binary(Buffer.from("00000000000000000000000000000000", "hex"), 0), keyId: 0 } }, operationTime: Timestamp({ t: 1688919579, i: 6 }), recoveryToken: { recoveryShardId: 'rs-shard-01' } }</pre> |
| T1 | <pre>collection.findOne({_id: ObjectId('64a958a37db95684b99cf284')}) { _id: ObjectId("64a958a37db95684b99cf284"), id: 48, name: 'Mauro', surname: 'Silvestri', date_of_birth: '14/12/1974', bank: 'Mediobanca', balance: 573869 } ss.commitTransaction() { ok: 1, '\$clusterTime': { clusterTime: Timestamp({ t: 1688919591, i: 1 }), signature: { hash: Binary(Buffer.from("00000000000000000000000000000000", "hex"), 0), keyId: 0 } }, operationTime: Timestamp({ t: 1688919591, i: 1 }), recoveryToken: {} }</pre> |

RC: snapshot, WC: 1 - Phantom read

| Transazioni | Comandi |
|-------------|--|
| T1 | <pre> mm = db.getMongo() ss = mm.startSession({readConcern: 'snapshot', writeConcern: 1}) ss.startTransaction({readConcern: 'snapshot', writeConcern: 1}) collection = ss.getDatabase("db_bank").getCollection("bank") db_bank.bank </pre> |
| T2 | <pre> mm = db.getMongo() ss = mm.startSession({readConcern: 'snapshot', writeConcern: 1}) ss.startTransaction({readConcern: 'snapshot', writeConcern: 1}) collection = ss.getDatabase("db_bank").getCollection("bank") db_bank.bank </pre> |
| T1 | <pre> collection.aggregate([{\$match: {balance: {\$gt: 900000}}}, {\$group: {_id: null, count: {\$sum: 1}})]) { _id: null, count: 99959 } </pre> |
| T2 | <pre> collection.aggregate([{\$match: {balance: {\$gt: 900000}}}, {\$group: {_id: null, count: {\$sum: 1}})]) { _id: null, count: 99959 } collection.updateOne({_id: ObjectId('64a958a37db95684b99cf35e')}, {\$set: {'balance': 300}}) { acknowledged: true, insertedId: null, matchedCount: 1, modifiedCount: 1, upsertedCount: 0 } ss.commitTransaction() { ok: 1, '\$clusterTime': { clusterTime: Timestamp({t: 1688920035, i: 7}), signature: { hash: Binary(Buffer.from("00", "hex"), 0), keyId: 0 } }, operationTime: Timestamp({t: 1688920035, i: 7}), recoveryToken: {recoveryShardId: 'rs-shard-01'} } </pre> |
| T1 | <pre> collection.aggregate([{\$match: {balance: {\$gt: 900000}}}, {\$group: {_id: null, count: {\$sum: 1}})]) { _id: null, count: 99959 } ss.commitTransaction() { ok: 1, '\$clusterTime': { clusterTime: Timestamp({t: 1688920051, i: 1}), signature: { hash: Binary(Buffer.from("00", "hex"), 0), keyId: 0 } }, operationTime: Timestamp({t: 1688920051, i: 1}), recoveryToken: {} } </pre> |

RC: snapshot, WC: majority - Dirty Read

| Transazioni | Comandi |
|-------------|--|
| T1 | <pre>mm = db.getMongo() ss = mm.startSession({readConcern: 'snapshot', writeConcern: 'majority'}) ss.startTransaction({readConcern: 'snapshot', writeConcern: 'majority'}) collection = ss.getDatabase("db_bank").getCollection("bank") db_bank.bank</pre> |
| T2 | <pre>mm = db.getMongo() ss = mm.startSession({readConcern: 'snapshot', writeConcern: 'majority'}) ss.startTransaction({readConcern: 'snapshot', writeConcern: 'majority'}) collection = ss.getDatabase("db_bank").getCollection("bank") db_bank.bank</pre> |
| T1 | <pre>collection.findOne({_id: ObjectId('64a958a37db95684b99cf284')}) { _id: ObjectId("64a958a37db95684b99cf284"), id: 48, name: 'Mauro', surname: 'Silvestri', date_of_birth: '14/12/1974', bank: 'Mediobanca', balance: 573769 } collection.updateOne({_id: ObjectId('64a958a37db95684b99cf284')}, {'\$inc': {'balance': -100}} { acknowledged: true, insertedId: null, matchedCount: 1, modifiedCount: 1, upsertedCount: 0 }</pre> |
| T2 | <pre>collection.findOne({_id: ObjectId('64a958a37db95684b99cf284')}) { _id: ObjectId("64a958a37db95684b99cf284"), id: 48, name: 'Mauro', surname: 'Silvestri', date_of_birth: '14/12/1974', bank: 'Mediobanca', balance: 573769 }</pre> |
| T1 | <pre>ss.commitTransaction() { ok: 1, '\$clusterTime': { clusterTime: Timestamp({ t: 1688921441, i: 6 }), signature: { hash: Binary(Buffer.from("00000000000000000000000000000000", "hex"), 0), keyId: 0 } }, operationTime: Timestamp({ t: 1688921441, i: 6 }), recoveryToken: { recoveryShardId: 'rs-shard-01' }</pre> |
| T2 | <pre>collection.updateOne({_id: ObjectId('64a958a37db95684b99cf284')}, {'\$inc': {'balance': -100}}) *WriteConflictError: encountered error from shard01-rs2701 during a transaction ... caused by ... WriteConflict error: this operation conflicted with another operation. Please retry your operation.</pre> |

RC: snapshot, WC: majority - Non repeatable read

| Transazioni | Comandi |
|-------------|--|
| T1 | <pre>mm = db.getMongo() ss = mm.startSession({readConcern: 'snapshot', writeConcern: 'majority'}) ss.startTransaction({readConcern: 'snapshot', writeConcern: 'majority'}) collection = ss.getDatabase("db_bank").getCollection("bank") db_bank.bank</pre> |
| T2 | <pre>mm = db.getMongo() ss = mm.startSession({readConcern: 'snapshot', writeConcern: 'majority'}) ss.startTransaction({readConcern: 'snapshot', writeConcern: 'majority'}) collection = ss.getDatabase("db_bank").getCollection("bank") db_bank.bank</pre> |
| T1 | <pre>collection.findOne({_id: ObjectId('64a958a37db95684b99cf284')}) { _id: ObjectId("64a958a37db95684b99cf284"), id: 48, name: 'Mauro', surname: 'Silvestri', date_of_birth: '14/12/1974', bank: 'Mediobanca', balance: 573669 }</pre> |
| T2 | <pre>collection.findOne({_id: ObjectId('64a958a37db95684b99cf284')}) { _id: ObjectId("64a958a37db95684b99cf284"), id: 48, name: 'Mauro', surname: 'Silvestri', date_of_birth: '14/12/1974', bank: 'Mediobanca', balance: 573669 } collection.updateOne({_id: ObjectId('64a958a37db95684b99cf284')}, {"\$inc": {"balance": -100}}) { acknowledged: true, insertedId: null, matchedCount: 1, modifiedCount: 1, upsertedCount: 0 } ss.commitTransaction() { ok: 1, '\$clusterTime': { clusterTime: Timestamp({ t: 1688921916, i: 6 }), signature: { hash: Binary(Buffer.from("00000000000000000000000000000000", "hex"), 0), keyId: 0 } }, operationTime: Timestamp({ t: 1688921916, i: 6 }), recoveryToken: { recoveryShardId: 'rs-shard-01' } }</pre> |
| T1 | <pre>collection.findOne({_id: ObjectId('64a958a37db95684b99cf284')}) { _id: ObjectId("64a958a37db95684b99cf284"), id: 48, name: 'Mauro', surname: 'Silvestri', date_of_birth: '14/12/1974', bank: 'Mediobanca', balance: 573669 } ss.commitTransaction() { ok: 1, '\$clusterTime': { clusterTime: Timestamp({ t: 1688921929, i: 1 }), signature: { hash: Binary(Buffer.from("00000000000000000000000000000000", "hex"), 0), keyId: 0 } }, operationTime: Timestamp({ t: 1688921929, i: 1 }), recoveryToken: {} }</pre> |

RC: snapshot, WC: majority - Phantom read

| Transazioni | Comandi |
|-------------|--|
| T1 | <pre>mm = db.getMongo() ss = mm.startSession({readConcern: 'snapshot', writeConcern: 'majority'}) ss.startTransaction({readConcern: 'snapshot', writeConcern: 'majority'}) collection = ss.getDatabase("db_bank").getCollection("bank") db_bank.bank</pre> |
| T2 | <pre>mm = db.getMongo() ss = mm.startSession({readConcern: 'snapshot', writeConcern: 'majority'}) ss.startTransaction({readConcern: 'snapshot', writeConcern: 'majority'}) collection = ss.getDatabase("db_bank").getCollection("bank") db_bank.bank</pre> |
| T1 | <pre>collection.aggregate([{ \$match: {balance: {\$gt: 900000}}}, { \$group: {_id: null, count: {\$sum: 1}}}) { _id: null, count: 99958 }</pre> |
| T2 | <pre>collection.aggregate([{ \$match: {balance: {\$gt: 900000}}}, { \$group: {_id: null, count: {\$sum: 1}}}) { _id: null, count: 99958 } collection.updateOne({_id: ObjectId('64a958a37db95684b99cf37f')}, {'\$set': {'balance': 300}}) { acknowledged: true, insertedId: null, matchedCount: 1, modifiedCount: 1, upsertedCount: 0 } ss.commitTransaction() { ok: 1, '\$clusterTime': { clusterTime: Timestamp({ t: 1688922339, i: 7 }), signature: { hash: Binary(Buffer.from("00000000000000000000000000000000", "hex"), 0), keyId: 0 } }, operationTime: Timestamp({ t: 1688922339, i: 7 }), recoveryToken: { recoveryShardId: 'rs-shard-02' } }</pre> |
| T1 | <pre>collection.aggregate([{ \$match: {balance: {\$gt: 900000}}}, { \$group: {_id: null, count: {\$sum: 1}}}) { _id: null, count: 99958 } ss.commitTransaction() { ok: 1, '\$clusterTime': { clusterTime: Timestamp({ t: 1688922354, i: 1 }), signature: { hash: Binary(Buffer.from("00000000000000000000000000000000", "hex"), 0), keyId: 0 } }, operationTime: Timestamp({ t: 1688922354, i: 1 }), recoveryToken: {} }</pre> |

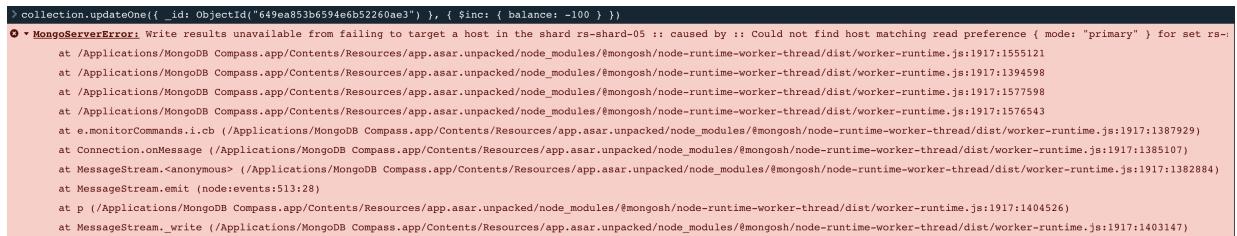
5.3 Spegnimento dei nodi

Durante questa fase di test, l'obiettivo è quello di verificare come i dati vengano distribuiti e gestiti in base ai diversi livelli di isolamento quando si spegne uno o più nodi del sistema. In particolare, l'attenzione verrà posta sull'effetto dello spegnimento dei nodi sulla distribuzione dei dati e come i livelli di isolamento influenzino questo processo.

5.3.1 Spegnimento di un host

In questa prima fase di test si è voluto testare l'esecuzione di una query andando a spegnere tutti i nodi di un host e valutato come si comporta MongoDB. In docker swarm è possibile gestire questa situazione con diversi comandi:

- Se si usa il comando `docker node update -availability pause id_host`, si va a mettere in pausa un host ed in questo caso si possono fare le read ma non l'update. Infatti provando a fare l'update viene mostrato l'errore in Figura 5.7.
- Se si usa il comando `docker node update -availability drain id_host`, vengono spenti i container senza dare la possibilità di effettuare la find e l'update.
- Nel momento in cui si vuole riattivare i nodi dell'host si esegue il comando `docker node update -availability active id_host`, ma in questo caso vengono duplicati i container tenendo spenti quelli che erano già spenti prima e accendendo quelli duplicati che però avranno un altro id quindi il router e il configserver non potranno riconoscerli e non sarà possibile comunque fare la find e l'update. In Figura 5.8 è mostrata la situazione dei container dopo aver effettuato il comando appena esposto.



```
> collection.updateOne({ _id: ObjectId("649ea853b6594e6b52260ae3") }, { $inc: { balance: -100 } })
MongoServerError: Write results unavailable from failing to target a host in the shard rs-shard-05 :: caused by :: Could not find host matching read preference { mode: "primary" } for set rs-05
    at /Applications/MongoDB Compass.app/Contents/Resources/app.asar.unpacked/node_modules/@mongosh/node-runtime-worker-thread/dist/worker-runtime.js:1917:1555121
    at /Applications/MongoDB Compass.app/Contents/Resources/app.asar.unpacked/node_modules/@mongosh/node-runtime-worker-thread/dist/worker-runtime.js:1917:1394598
    at /Applications/MongoDB Compass.app/Contents/Resources/app.asar.unpacked/node_modules/@mongosh/node-runtime-worker-thread/dist/worker-runtime.js:1917:1577598
    at /Applications/MongoDB Compass.app/Contents/Resources/app.asar.unpacked/node_modules/@mongosh/node-runtime-worker-thread/dist/worker-runtime.js:1917:1576543
    at e.monitorCommands.i.cb (/Applications/MongoDB Compass.app/Contents/Resources/app.asar.unpacked/node_modules/@mongosh/node-runtime-worker-thread/dist/worker-runtime.js:1917:1387929)
    at Connection.onMessage (/Applications/MongoDB Compass.app/Contents/Resources/app.asar.unpacked/node_modules/@mongosh/node-runtime-worker-thread/dist/worker-runtime.js:1917:1385107)
    at MessageStream.<anonymous> (/Applications/MongoDB Compass.app/Contents/Resources/app.asar.unpacked/node_modules/@mongosh/node-runtime-worker-thread/dist/worker-runtime.js:1917:1382884)
    at MessageStream.emit (node:events:513:28)
    at p (/Applications/MongoDB Compass.app/Contents/Resources/app.asar.unpacked/node_modules/@mongosh/node-runtime-worker-thread/dist/worker-runtime.js:1917:1404526)
    at MessageStream._write (/Applications/MongoDB Compass.app/Contents/Resources/app.asar.unpacked/node_modules/@mongosh/node-runtime-worker-thread/dist/worker-runtime.js:1917:1403147)
```

Figura 5.7: Errore con nodi host spenti facendo l'update.

5.3.2 Spegnimento di un nodo

In questa fase di test si è voluto testare lo spegnimento di un nodo di uno shard andando a simulare un errore di sistema. Il problema in questo caso è stato simile a quello esposto in Figura 5.8, in cui andando a spegnere uno nodo dello shard in docker swarm, viene replicato il container in quanto il comportamento predefinito è che il cluster di Swarm si adatta automaticamente per mantenere un numero desiderato di nodi in esecuzione. In Figura 5.9 viene mostrata l'esecuzione di una query prima di effettuare lo spegnimento del nodo che darà un risultato positivo e successivamente la stessa query ma dopo aver effettuato lo spegnimento del nodo che darà un errore dovuto al fatto che non riesce a recuperare il dato. Viene quindi abortita la transazione.

```

> ▶ mongo:latest@sha256:e3fa459b4f4b72f3257c67a23c145e250b8b5700f033860392c68539b998bbe3 app_shard03-c.1.uq7wbmq72nc51mdj5hwcqkn23 - Up Less than a minute
> ▶ mongo:latest@sha256:e3fa459b4f4b72f3257c67a23c145e250b8b5700f033860392c68539b998bbe3 app_shard04-b.1.1yfelk3eyiylmqobcnxiwy4 - Up Less than a minute
> ▶ mongo:latest@sha256:e3fa459b4f4b72f3257c67a23c145e250b8b5700f033860392c68539b998bbe3 app_shard04-c.1.9cpio4y37skcb5hvq3gow3ojq - Up Less than a minute
> ▶ mongo:latest@sha256:e3fa459b4f4b72f3257c67a23c145e250b8b5700f033860392c68539b998bbe3 app_shard04-d.a.1.u8mgg63duqlsz701zv887m725 - Up Less than a minute
> ▶ mongo:latest@sha256:e3fa459b4f4b72f3257c67a23c145e250b8b5700f033860392c68539b998bbe3 app_shard03-b.1.4q77bh5lnnp494i0vx63lb9h - Up Less than a minute
> ▶ mongo:latest@sha256:e3fa459b4f4b72f3257c67a23c145e250b8b5700f033860392c68539b998bbe3 app_shard02-b.1.cj398gegb016dc4pk0by5gn - Up Less than a minute
> ▶ mongo:latest@sha256:e3fa459b4f4b72f3257c67a23c145e250b8b5700f033860392c68539b998bbe3 app_shard02-c.1.gjmbauiuvx5rjq8occmcgkpg - Up Less than a minute
> ▶ mongo:latest@sha256:e3fa459b4f4b72f3257c67a23c145e250b8b5700f033860392c68539b998bbe3 app_shard02-a.1.xec74wd2cv7ack6s6w77vx8 - Up Less than a minute
> ▶ mongo:latest@sha256:e3fa459b4f4b72f3257c67a23c145e250b8b5700f033860392c68539b998bbe3 app_shard04-a.1.d17f65om0loufas4i3up4rpf - Up Less than a minute
> □ mongo:latest@sha256:e3fa459b4f4b72f3257c67a23c145e250b8b5700f033860392c68539b998bbe3 app_shard03-c.1.aldb4lbzj37zz53dd5oe1kmilm - Exited (137) About a minute ago
> □ mongo:latest@sha256:e3fa459b4f4b72f3257c67a23c145e250b8b5700f033860392c68539b998bbe3 app_shard02-b.1.tup5s5n305px2vz92f6o2qqot - Exited (137) About a minute ago
> □ mongo:latest@sha256:e3fa459b4f4b72f3257c67a23c145e250b8b5700f033860392c68539b998bbe3 app_shard04-a.1.v9a60jopsdp9pnec6ijtj06sw - Exited (137) About a minute ago
> □ mongo:latest@sha256:e3fa459b4f4b72f3257c67a23c145e250b8b5700f033860392c68539b998bbe3 app_shard03-b.1.im89r2isutvcfgmkhu3wrggsa - Exited (137) About a minute ago
> □ mongo:latest@sha256:e3fa459b4f4b72f3257c67a23c145e250b8b5700f033860392c68539b998bbe3 app_shard02-a.1.aq0etnx46gwqv2tmlqd1tp8 - Exited (137) About a minute ago
> □ mongo:latest@sha256:e3fa459b4f4b72f3257c67a23c145e250b8b5700f033860392c68539b998bbe3 app_shard04-b.1.l6g9m39apsvzw3jykaygut6n - Exited (137) About a minute ago
> □ mongo:latest@sha256:e3fa459b4f4b72f3257c67a23c145e250b8b5700f033860392c68539b998bbe3 app_shard03-a.1.l47na5c1xe75b0ohzywquuj - Exited (137) About a minute ago
> □ mongo:latest@sha256:e3fa459b4f4b72f3257c67a23c145e250b8b5700f033860392c68539b998bbe3 app_shard02-c.1.8521nxdmjqkv2lukap1nkpmof - Exited (137) About a minute ago
> □ mongo:latest@sha256:e3fa459b4f4b72f3257c67a23c145e250b8b5700f033860392c68539b998bbe3 app_shard04-c.1.4u22gb9q0lmfgv3n835bc8ss5 - Exited (137) About a minute ago

```

Figura 5.8: Container docker dopo aver riattivato i nodi dell'host.

```

> collection.findOne({_id: ObjectId('649ea853b6594e6b52260ae6')});
<   {
      _id: ObjectId('649ea853b6594e6b52260ae6'),
      id: 7,
      name: 'Ginevra',
      surname: 'Moretti',
      date_of_birth: '6/8/1967',
      bank: 'Cassa Depositi e Prestiti',
      balance: 301686
    }
> collection.findOne({_id: ObjectId('649ea853b6594e6b52260ae6')});
● • MongoServerError: Encountered non-retryable error during query :: caused by :: Transaction with { txnNumber: 2 } has been aborted.
    at Connection.onMessage (/Applications/MongoDB Compass.app/Contents/Resources/app.asar.unpacked/node_modules/@mongosh/node-runtime-worker-thread/dist/worker-runtime.js:1917:1385004)
    at MessageStream.<anonymous> (/Applications/MongoDB Compass.app/Contents/Resources/app.asar.unpacked/node_modules/@mongosh/node-runtime-worker-thread/dist/worker-runtime.js:1917:1382884)
    at MessageStream.emit (node:events:513:28)
    at p (/Applications/MongoDB Compass.app/Contents/Resources/app.asar.unpacked/node_modules/@mongosh/node-runtime-worker-thread/dist/worker-runtime.js:1917:1404526)
    at MessageStream._write (/Applications/MongoDB Compass.app/Contents/Resources/app.asar.unpacked/node_modules/@mongosh/node-runtime-worker-thread/dist/worker-runtime.js:1917:1403147)
    at writeOrBuffer (node:internal/streams/writable:391:12)
    at _write (node:internal/streams/writable:332:10)
    at Writable.write (node:internal/streams/writable:336:10)
    at Socket.ondata (node:internal/streams/readable:754:22)
    at Socket.emit (node:events:513:28)

```

Figura 5.9: Errore read dopo aver spento un nodo.

5.3.3 Spegnimento dei nodi in locale

Come ultima fase di test si è voluto testare lo spegnimento dei nodi con vari livelli di isolamento utilizzando docker compose su un architettura in locale con un router, un config server e due shard con un nodo primario e due nodi secondari ciascuno. Questo è stato fatto per vedere le differenze rispetto a docker swarm che andava a duplicare i container per mantenere sempre lo stesso numero di nodi.

RC: majority, WC: majority

- Avvio una transazione con impostazioni:

```
session.startTransaction(readConcern: 'majority', writeConcern: w: 1);
```

- Vengono spenti 2 nodi su 3

In questo caso andando ad effettuare un update rimane in attesa per un po' di tempo e poi restituisce un errore.

```
[direct: mongos] test> collection.updateOne({ _id: ObjectId("646bd4c8cc2d06e28d68c6d3") }, { $inc: { balance: -100 } })
MongoServerError: Write results unavailable from failing to target a host in the shard rs-shard-02 :: caused by :: Could not fi
nd host matching read preference { mode: "primary" } for set rs-shard-02
```

Figura 5.10: Spegnimento di due nodi con RC:majority, WC: majority.

RC: majority, WC: 1

- Avvio una transazione con impostazioni:
`session.startTransaction(readConcern: level: 'majority', writeConcern: w: 1);`
- Vengono spenti 2 nodi su 3

```
[direct: mongos] test> collection.updateOne({ _id: ObjectId("646bd4c8cc2d06e28d68c6d3") }, { $inc: { balance: -100 } })
{
  acknowledged: true,
  insertedId: null,
  matchedCount: 1,
  modifiedCount: 1,
  upsertedCount: 0
}
```

Figura 5.11: Spegnimento di due nodi con RC:majority, WC: majority.

La transazione avviene correttamente in quanto con wc: 1 è sufficiente che la modifica venga accettata da almeno il nodo primario.

5.4 Script

In questa sezione verranno mostrati alcuni script che sono stati prodotti. La decisioni nel produrre questi script è nata perché:

- Si vuole gestire la questione delle transazioni 'concorrenti' e vedere come si potessero comportare a livello di codice. In questo caso, grazie a Python, è stata introdotta nello script anche la parte per gestire le eccezioni con il blocco 'try-except-finally'.
- Si vogliono verificare le singole transazioni con alcune operazioni. Le operazioni scelte in questo caso possono essere 'reali', cioè che è sensato verificare, come: spostamento di soldi, ricerca clienti di una banca o inserimento dei clienti.

5.4.1 Script con eccezioni

E' uno script che permette di far scattare un'eccezione durante la fase di aggiornamento dello stesso dato precedente letto dalle singole transazioni. Grazie a Python, nello script è stato aggiunto il blocco per gestire l'eccezione. Di seguito è mostrato il codice e il relativo output.

```

# Connessione al server MongoDB
client1 = MongoClient('mongodb://localhost:27117')
client2 = MongoClient('mongodb://localhost:27117')

# Imposta la write concern per entrambe le sessioni
read_concern = pymongo.read_concern.ReadConcern("local")
write_concern = WriteConcern(w=1)

trans = pymongo.client_session.TransactionOptions(read_concern = read_concern, write_concern = write_concern)

# Avvia due sessioni
session1 = client1.start_session(default_transaction_options=trans)
session2 = client2.start_session(default_transaction_options=trans)

```

Figura 5.12: Scelta Read/WriteConcerne e faccio partire le sessioni

```

try:
    # Avvia la transazione per la sessione 1
    with session2.start_transaction(read_concern=read_concern, write_concern=write_concern) as s2,
        session1.start_transaction(read_concern=read_concern, write_concern=write_concern) as s1:
        # Ottieni la collezione nella sessione 1
        collection1 = session1.client['db_bank']['bank']
        collection2 = session2.client['db_bank']['bank']

        result1 = collection1.find_one({'_id': 299})
        print("Lettura in T1: ", result1)

        # Esegui l'operazione di modifica nella sessione 1
        collection1.update_one(
            {'_id': result1['_id']},
            {'$inc': {'balance': 100}}, session = session1
        )

        # Esegui l'operazione di lettura nella sessione 2
        result2 = collection2.find_one({'_id': 299}, session = session2)
        print("Lettura in T2: ", result2)

        session1.commit_transaction()
        print("Transazione 1 committata")

        # Esegui l'operazione di modifica nella sessione 2
        collection2.update_one(
            {'_id': result2['_id']},
            {'$inc': {'balance': 100}}, session = session2
        )

        # Commit della transazione per la sessione 2
        session2.commit_transaction()
        print("Transazione 2 committata")

        result2 = collection2.find_one({'_id': 299}, session = session2)
        print(result2)

except Exception as e:
    # Esegui il rollback delle transazioni in caso di errore
    # session1.abort_transaction()
    # session2.abort_transaction()
    collection2.update_one(
        {'_id': result2['_id']},
        {'$inc': {'balance': 100}}, session = session2
    )
    print("Transazione 2 committata")
    result2 = collection2.find_one({'_id': 299}, session = session2)
    print("Lettura finale ", result2)

finally:
    # Chiudi le sessioni
    session1.end_session()
    session2.end_session()

```

Figura 5.13: Eseguo le transazioni e le operazioni associate ad esso, con relativa gestione delle exception

```

Lettura in T1: {'_id': ObjectId('64a958a37db95684b99cf37f'), 'id': 299, 'name': 'Gaia', 'surname': 'Pellegrini', 'date_of_birth': '1/11/1985', 'bank': 'UBI Banca', 'balance': 700}
Lettura in T2: {'_id': ObjectId('64a958a37db95684b99cf37f'), 'id': 299, 'name': 'Gaia', 'surname': 'Pellegrini', 'date_of_birth': '1/11/1985', 'bank': 'UBI Banca', 'balance': 700}
Transazione 1 committata
Transazione 2 committata
Lettura finale {'_id': ObjectId('64a958a37db95684b99cf37f'), 'id': 299, 'name': 'Gaia', 'surname': 'Pellegrini', 'date_of_birth': '1/11/1985', 'bank': 'UBI Banca', 'balance': 900}

```

Figura 5.14: Risultato dell'esecuzione

5.4.2 Script per altri metodi

E' un notebook in cui sono presenti diversi metodi che permettono di fare alcune operazioni in singole transazioni. Le operazioni di scambio di denaro permette di rimuovere e aggiungere lo stessa cifra da entrambi gli utenti, ma in più verra creato/aggiornato un array di transazioni, in cui si potranno vedere tutti i movimenti relativi a quella persona. C'è anche un'operazione riguardante la ricerca di movimenti di denaro in determinati giorni. Infine, le altre operazioni sono l'inserimento/rimozione/ricerca di tutti i clienti di una banca. Di seguito sono riportate le immagini del notebook.

INSERT

```
def generate_array_to_insert(collection):
    example = {
        "id": 1,
        "name": "Pippo",
        "surname": "Pluto",
        "date_of_birth": "18/03/2000",
        "bank": "UniCredit",
        "balance": "10000",
        "test" : 1,
    }
    list_name = ['Sofia', 'Matteo', 'Giuseppe', 'Isabella', 'Federico', 'Giulia', 'Carlo', 'Veronica', 'Luca', 'Aurora', 'Marco',
    list_surname = ["Rossi", "Bianchi", "Russo", "Esposito", "Ricci", "Ferrari", "Rizzo", "Romano", "Conti", "Santoro", "Marino"]
    italian_banks = [
        "Intesa Sanpaolo",
        "UniCredit",
        "Banco BPM",
        "Monte dei Paschi di Siena",
        "UBI Banca",
        "Banca Nazionale del Lavoro",
        "Banca Generali",
        "Mediobanca",
        "Cassa Depositi e Prestiti",
        "Banca Popolare di Sondrio"
    ]
    banks = []

    start_point = collection.find_one(sort=[("id", pymongo.DESCENDING)])
    start_point = start_point["id"]

    for i in range(start_point, start_point+1000):
        item = example.copy()

        rand_name = random.randint(0, 99)
        rand_year = random.randint(1960, 2005)
        rand_month = random.randint(1, 12)
        rand_day = random.randint(1, 31)
        rand_banca = random.randint(0,9)
        rand_balance = random.randint(1000, 1000000)

        item["id"] = i
        item["name"] = list_name[rand_name]
        item["surname"] = list_surname[rand_name]
        item["date_of_birth"] = f'{rand_day}/{rand_month}/{rand_year}'
        item["bank"] = italian_banks[rand_banca]
        item["balance"] = rand_balance
        item["date"] = datetime.datetime.now()
        item["test"] = 1

        banks.append(item)

    return banks
```

Figura 5.15: Funzione per generare i dati da inserire

```
#insert single document multiple times
def insert_many_user_insert_one(session):
    coll = session.client.db_bank.bank
    value_to_insert = generate_array_to_insert(coll)
    app = []
    print("---")
    print(value_to_insert)
    print("---")
    for i in range(1, len(value_to_insert)):
        id_in = coll.insert_one(value_to_insert[i], session=session)
        app.append(id_in.inserted_id)
    return app
```

Figura 5.16: Metodo per inserire i valori nel database

```

single update
def wrapper_trans(id_snd, id_rcv, val):
    def callback(session):
        coll = session.client.db_bank.bank
        json_snd = coll.find_one({"id": id_snd}, session = session)
        json_rcv = coll.find_one({"id": id_rcv}, session = session)
        bal = json_snd["balance"]
        if bal >= val:
            new_bal_snd = json_snd["balance"] + val
            new_bal_rcv = json_snd["balance"] - val

            coll.update_one(
                {"_id": json_snd["_id"]},
                {"$set": {"balance": new_bal_snd}}, session = session)
            coll.update_one(
                {"_id": json_rcv["_id"]},
                {"$set": {"balance": new_bal_rcv}}, session = session)

            coll.update_one(
                {"_id": json_snd["_id"]},
                {"$push": {"trans": {
                    "id_user": json_rcv["_id"],
                    "type" : -1 ,
                    "value": val,
                    "date": datetime.datetime.now().strftime("%x")}}}, session = session)
            coll.update_one(
                {"_id": json_rcv["_id"]},
                {"$push": {"trans": {
                    "id_user": json_snd["_id"],
                    "type" : 1 ,
                    "value": val,
                    "date": datetime.datetime.now().strftime("%x")}}}, session = session)
        else:
            raise Exception("transaction rejecteded for insufficient money")
    final_snd = coll.find_one({"id": id_snd}, session = session)
    final_rcv = coll.find_one({"id": id_rcv}, session = session)
    app = [final_snd, final_rcv]
    return app

    return callback

```

```

update_users_transaction = wrapper_trans(1000007, 1000011, 1000)

```

Figura 5.17: Metodo wrapper per eseguire uno scambio di denaro tra due utenti

| |
|--|
| FIND |
| <pre> # wrapper to find def wrapper_bank_user(val): def callback(session): coll = session.client.db_bank.bank az = coll.find({"bank": val}, session = session) app = [] for a in az: app.append(a) return app return callback </pre> |
| <pre> find_bank_user = wrapper_bank_user("UniCredit") </pre> |

Figura 5.18: Metodo wrapper per cercare i clienti di una banca

```

def wrapper_date_transaction(date):
    def callback(session):
        coll = session.client.db_bank.bank
        az = coll.find({"trans.date" : date}, session = session)
        app = []
        for a in az:
            app.append(a)
        return app
    return callback

find_date_transaction = wrapper_date_transaction(datetime.datetime.now().strftime("%x"))

```

Figura 5.19: Metodo wrapper per vedere gli scambi di denaro in una certa data

```

def wrapper_remove(bank):
    def callback(session):
        coll = session.client.db_bank.bank
        found = coll.find({"bank": bank}, session = session)
        coll.delete_many({"bank": bank}, session = session)

        app = []
        for a in found:
            app.append(a)
        return app
    return callback

remove_bank_many = wrapper_remove("UBI Banca")

```

Figura 5.20: Metodo wrapper per cancellare i clienti da una banca

```

def execute_func(client, callback, write_c, read_c, read_p): #Step 2: Start a client session.
    try:
        az = 0
        with client.start_session() as session:
            # Step 3: Use with_transaction to start a transaction, execute the callback, and commit (o
            az = session.with_transaction(
                callback,
                #read_concern=ReadConcern("Local"),
                read_concern = read_c,
                write_concern=write_c,
                #read_preference=ReadPreference.PRIMARY,
                read_preference = read_p
            )
    except Exception as ex:
        print("OS Exception:", ex)
    finally:
        print("it's ok")
        return az

```

Figura 5.21: Metodo per eseguire le callback relative ad una certa transazione

```

#sti tre sono da modificare in base a che test si vuol fare
read_concern=pymongo.read_concern.ReadConcern("local")
write_concern=pymongo.write_concern.WriteConcern("majority")
read_preference=pymongo.ReadPreference.PRIMARY

execute_func(client, insert_many_user_insert_one, read_c = read_concern, write_c = write_concern, read_p = read_preference)
--+
[{"id": 100000, "name": "Fabio", "surname": "Marchetti", "date_of_birth": "30/5/2003", "bank": "Banca Nazionale del Lavoro", "balance": 632420, "test": 1, "date": datetime.datetime(2023, 7, 12, 11, 25, 24, 158139)}, {"id": 1000001, "name": "Vale ntina", "surname": "Montanari", "date_of_birth": "6/10/2005", "bank": "Monte dei Paschi di Siena", "balance": 365216, "test": 1, "date": datetime.datetime(2023, 7, 12, 11, 25, 24, 158139)}, {"id": 1000002, "name": "Tomaso", "surname": "Costa", "date_of_birth": "7/10/1985", "bank": "Monte dei Paschi di Siena", "balance": 507120, "test": 1, "date": datetime.datetime(2023, 7, 12, 11, 25, 24, 158139)}, {"id": 1000003, "name": "Matteo", "surname": "Fontana", "date_of_birth": "25/3/2000", "bank": "Banco BPM", "balance": 630116, "test": 1, "date": datetime.datetime(2023, 7, 12, 11, 25, 24, 158139)}, {"id": 1000004, "name": "Martina", "surname": "Piras", "date_of_birth": "23/3/1989", "bank": "Banca Popolare di Sondrio", "balance": 602581, "test": 1, "date": datetime.datetime(2023, 7, 12, 11, 25, 24, 158139)}, {"id": 1000005, "name": "Aurora", "surname": "Santoro", "date_of_birth": "1/2/1982", "bank": "Unicredit", "balance": 953072, "test": 1, "date": datetime.datetime(2023, 7, 12, 11, 25, 24, 158139)}, {"id": 1000006, "name": "Elisa", "surname": "Caruso", "date_of_birth": "22/2/2003", "bank": "Cassa Depositi e Prestiti", "balance": 211451, "test": 1, "date": datetime.datetime(2023, 7, 12, 11, 25, 24, 158139)}, {"id": 1000007, "name": "Enrico", "surname": "Marini", "date_of_birth": "28/10/1962", "bank": "Unicredit", "balance": 925677, "test": 1, "date": datetime.datetime(2023, 7, 12, 11, 25, 24, 158139)}, {"id": 1000008, "name": "Davide", "surname": "Lombardo", "date_of_birth": "22/11/1994", "bank": "Banca Popolare di Sondrio", "balance": 188557, "test": 1, "date": datetime.datetime(2023, 7, 12, 11, 25, 24, 158139)}, {"id": 1000009, "name": "Ginevra", "surname": "Moretti", "date_of_birth": "21/9/1972", "bank": "Monte dei Paschi di Siena", "balance": 664164, "test": 1, "date": datetime.datetime(2023, 7, 12, 11, 25, 24, 158139)}, {"id": 1000010, "name": "Andrea", "surname": "Rossetti", "date_of_birth": "22/6/1966", "bank": "Banco BPM", "balance": 620265, "tes

```

Figura 5.22: Risultato inserimento dati

```
#example_update
execute_func(client, update_users_transaction, read_c = read_concern, write_c = write_concern, read_p = read_preference)

it's ok

[{"_id": ObjectId('64ae718457a8c8cbd2746489'),
 'id': 1000007,
 'name': 'Enrico',
 'surname': 'Marini',
 'date_of_birth': '28/10/1962',
 'bank': 'UniCredit',
 'balance': 926677,
 'test': 1,
 'date': datetime.datetime(2023, 7, 12, 11, 25, 24, 158000),
 'trans': [{"id_user": ObjectId('64ae718457a8c8cbd274648d'),
   'type': -1,
   'value': 1000,
   'date': '07/12/23'}]},
 {"_id": ObjectId('64ae718457a8c8cbd274648d'),
 'id': 1000011,
 'name': 'Enrico',
 'surname': 'Donati',
 'date_of_birth': '8/2/1988',
 'bank': 'Banca Nazionale del Lavoro',
 'balance': 924677,
 'test': 1,
 'date': datetime.datetime(2023, 7, 12, 11, 25, 24, 158000),
 'trans': [{"id_user": ObjectId('64ae718457a8c8cbd2746489'),
   'type': 1,
   'value': 1000,
   'date': '07/12/23'}]}]
```

Figura 5.23: Risultato scambio di denaro

```
found = execute_func(client, find_bank_user, read_c = read_concern, write_c = write_concern, read_p = read_preference)
print(len(found))

it's ok
100266

found = execute_func(client, find_date_transaction, read_c = read_concern, write_c = write_concern, read_p = read_preference)
print(found)

it's ok
[{"_id": ObjectId('64ae718457a8c8cbd2746489'), 'id': 1000007, 'name': 'Enrico', 'surname': 'Marini', 'date_of_birth': '28/10/1962', 'bank': 'UniCredit', 'balance': 926677, 'test': 1, 'date': datetime.datetime(2023, 7, 12, 11, 25, 24, 158000), 'trans': [{"id_user": ObjectId('64ae718457a8c8cbd274648d'), 'type': -1, 'value': 1000, 'date': '07/12/23'}]}, {"_id": ObjectId('64ae718457a8c8cbd274648d'), 'id': 1000011, 'name': 'Enrico', 'surname': 'Donati', 'date_of_birth': '8/2/1988', 'bank': 'Banca Nazionale del Lavoro', 'balance': 924677, 'test': 1, 'date': datetime.datetime(2023, 7, 12, 11, 25, 24, 158000), 'trans': [{"id_user": ObjectId('64ae718457a8c8cbd2746489'), 'type': 1, 'value': 1000, 'date': '07/12/23'}]}]

execute_func(client, remove_bank_many, read_c = read_concern, write_c = write_concern, read_p = read_preference)

it's ok
[]
```

Figura 5.24: Risultato della ricerca per banca, delle transazioni in una certa data e dell'eliminazione dei clienti di una banca

Capitolo 6

Problemi riscontrati

In questa sezione della nostra relazione sono stati riportati tutte le problematiche che sono state riscontrate durante l'implementazione del progetto.

6.1 Problema rete

Il problema riscontrato con la rete con dominio unimib si manifesta nel seguente modo.

Durante il tentativo di stabilire una connessione locale, senza l'impiego di macchine virtuali, è stato impossibile mettere in comunicazione le nostre tre macchine. Si ritiene che tale problematica derivi da regole restrittive imposte dall'Università Bicocca.

Nel tentativo di comprendere la situazione, grazie al `ping` si è cercato di far comunicare le nostre macchine, fornendo loro l'indirizzo IP della macchina di destinazione. Tuttavia, questo tentativo si è rivelato infruttuoso.

Successivamente si è scelto un approccio simile, utilizzando lo strumento `telnet`. Oltre a specificare l'indirizzo IP della macchina desiderata, bisognava scegliere la porta attraverso cui desideravamo comunicare, essendo questa associata a uno dei container di Docker. Anche in questo caso, il risultato è stato negativo.

Un'altra strategia per affrontare il problema è stata quella di utilizzare l'hotspot del cellulare. Purtroppo, si è mostrata la stessa problematica e in questo caso il problema poteva derivare dalle rigide policy dei gestori telefonici.

Al fine di risolvere questa problematica, è stata fornita una soluzione che ha offerto l'università: un ambiente virtuale.

Abbiamo optato per l'utilizzo di tre macchine virtuali ospitate su Azure, una piattaforma di cloud computing. Grazie a questa scelta, le macchine virtuali sono stati inserite sotto una sottorete locale all'interno dell'ambiente virtuale. Questa sottorete presentava politiche di comunicazione tra macchine meno restrittive rispetto a quelle riscontrate precedentemente.

6.2 Problema firewall

Durante il tentativo di stabilire una connessione tra i differenti laptop personali, è sorto un problema particolarmente ostico.

Alcune delle macchine coinvolte, quelle con sistema operativo Windows, presentavano politiche di firewall estremamente restrittive. In particolare, il servizio Windows Defender bloccava la comunicazione sia in entrata che in uscita tra le macchine.

Per risolvere questa problematica, la soluzione è stata disabilitare tutte le policy del firewall, rendendo così le nostre macchine più vulnerabili da un punto di vista della sicurezza.

Nonostante questa soluzione, si è effettuata la comunicazione desiderata tra le macchine coinvolte attraverso il tool `ping`.

6.3 Problema Virtual Machine

Una volta risolto il problema di rete grazie all'implementazione delle macchine virtuali, è emerso un nuovo inconveniente legato alle stesse.

Sebbene fosse possibile effettuare la connessione `telnet` tra i vari container, per il progetto era necessario avere più nodi interconnessi sulle tre macchine differenti. Per raggiungere questo obiettivo, è stata utilizzata una rete che consentisse tale connettività.

Dopo diversi test con la bridge network, si è scelto di utilizzare l'*overlay network*. Come descritto nella documentazione, questa soluzione fornisce la possibilità di creare una rete comune tra più host attraverso Docker Swarm. Tuttavia, c'è stato un problema durante l'esecuzione del comando `docker swarm init`, cioè quello che restituisce un token per connettersi ad altri host. In questo caso, durante l'esecuzione della `docker swarm join`, si è verificato un errore che indicava l'impossibilità di trovare l'host che aveva generato il token.

Dopo aver segnalato il problema al Professor Ciavotta e confermato l'incapacità di stabilire la connessione tra le macchine virtuali fornite, queste sono state resettate e sono state fornite altre tre macchine virtuali. Con queste nuove macchine virtuali, la comunicazione tra di esse poteva avvenire.

Capitolo 7

Conclusioni

Dopo aver tirato su l'architettura a shard nella macchina virtuale fornita, sono stati effettuati diversi test. Il test del caricamento è risultato positivo, perché col crescere delle dimensioni del file da importare, sembra che il tempo diminuisca. Il test per l'isolamento è stato utile per comprendere che le transazioni, anche modificando i valori di `readConcern` e `writeConcern`, hanno un isolamento alto, tale che non permette: Dirty Read, Non-repeatable Read e Phantom Read. Infine il test con lo spegnimento dei nodi ha dimostrato che in base a che `writeConcern` si utilizza, dopo la caduta di uno nodo, MongoDB riesce a gestire o meno le write sul database.