GraphDB

Tommaso Ferrario (869005) (@TommasoFerrario18)

Simone Vendramini (866229)(@simone-vendramini)

July 2024

Indice

Capitolo 1

Introduzione

Il nostro obiettivo è quello di scegliere ed analizzare un DBMS a grafo, differente da Neo4j, cercando di rispondere alle seguenti domande:

- Come opera?
- È in grado di scalare orizzontalmente? Se si, come lo fa?

In seguito ad una accurata ricerca abbiamo deciso di analizzare la soluzione offerta da **ArangoDB** in quanto esiste una sua versione open source e nella lista di ranking su db-engines è considerato uno dei più utilizzati.

Per rispondere a queste domande abbiamo deciso di simulare un semplice sito di incontri, in cui dovremo gestire il salvataggio di 100000 utenti, ognuno con le proprie preferenze e relazioni. Successivamente abbiamo verificato alcune soluzioni per effettuare la distribuzione su più server. Nel seguito di questo capitolo introduttivo ci dedicheremo quindi a definire come abbiamo strutturato i dati.

1.1 Dati

Il processo di creazione dei dati per poter popolare il database è principalmente suddiviso in due parti:

- Generazione delle informazioni per i nodi
- Generazione degli archi

La popolazione dei nodi è stata gestita tramite mockaroo, il quale permette di generare mock-data per testare applicazioni. Abbiamo generato informazioni per un totale di 100000 utenti ognuno dei quali è rappresentato mediante l'utilizzo dei seguenti nodi:

- **User**: First_name, Last_name, Email, Phone, Birth_date, Gender, Latitude, Longitude.
- **Color**: Name.
- **Movie**: Title.
- **City**: Name.
- **Continent**: Name.
- **Country**: Name, Code.
- **MovieCategory**: Name.
- **University**: Name.

Nota 1 *La soluzione adottata potrebbe non essere la migliore a seconda delle query da effettuare. Abbiamo adottato questa configurazione per semplificare la fase di test. Ad esempio, una soluzione alternativa per migliorare la rappresentazione si potrebbero accorpare la maggior parte di questi nodi.*

Gli archi che abbiamo generato invece si possono dividere in due categorie principali:

- Archi per strutturare il profilo dell'utente.

- Archi per gestire le interazioni tra gli utenti.

Gli archi utilizzati per la definizione dei profili servono a collegare gli utenti con i nodi rappresentanti gli interessi ed i nodi rappresentanti la locazione geografica. Questi archi sono **IntMovie**, **IntMovieCategory**, **IntColor**, **LivesIn**, **StudiesAt**.

La seconda categoria degli archi (likes e match) serve a rappresentare le relazioni tra gli utenti. Per fare ciò abbiamo utilizzato un approccio randomico aumentando la probabilità di interazione tra utenti della stessa città. Così facendo, abbiamo generato circa 3.5 milioni di interazioni di tipo `Like` tra gli utenti.

Gli archi di interazione sono quindi:

- **Likes**: rappresentano una relazione orientata tra due utenti.
- **Matches**: costituiscono una connessione non orientata stabilita tra due utenti che manifestano reciproci **Like**.

Oltre a queste due tipologie abbiamo adoperato altri archi utili alla costruzione del database, ossia **LocatedIn** e **CountryLocatedIn**.

Capitolo 2

ArangoDB

2.1 Introduzione

ArangoDB si distingue come un database poliglotta, open-source e distribuito, che abbraccia la versatilità dei modelli di dati NoSQL. Offre supporto a tre modelli distinti:

- **Modello Documentale**: in linea con MongoDB, ArangoDB permette di memorizzare i dati in documenti JSON, garantendo flessibilità nella struttura dei dati e facilità di gestione.
- **Modello a Grafo**: ispirandosi a Neo4j, ArangoDB consente di rappresentare e manipolare i dati attraverso nodi e archi, offrendo una soluzione per la gestione di dati altamente connessi.
- **Modello Chiave-valore**: simile a Redis, ArangoDB supporta la memorizzazione dei dati in coppie chiave-valore, ottimizzando le operazioni di recupero e salvataggio.

Nel contesto di questo progetto, ci concentreremo sull'analisi e utilizzo del modello a grafo offerto da ArangoDB, valutando le sue potenzialità nell'affrontare scenari di dati interconnessi in modo efficace ed efficiente.

2.2 Architettura

All'interno di ArangoDB, i dati sono organizzati gerarchicamente, dove i documenti rappresentano gli elementi base, i quali sono raccolti in collezioni contenute nei database.

2.2.1 Documenti

I **documenti** sono memorizzati in formato JSON, offrendo una notevole flessibilità nella struttura dei dati. Ognuno dei quali è costituito da attributi, definiti come coppie chiave-valore. Un attributo cruciale è `_key`, che rappresenta un identificativo univoco all'interno di una collezione e del database stesso. Questo identificativo è immutabile dopo la creazione del documento, agevolando il suo utilizzo come riferimento veloce in query. Oltre a questo attributo, all'interno di ogni documento sono presenti anche gli attributi `_id` e `_rev`. Il primo viene settato in automatico concatenando con l'utilizzo del carattere / la chiave e il nome della collezione. Mentre il secondo viene creato in modo automatico da ArangoDB. Questo valore viene aggiornato ogni volta che viene fatta una modifica al documento. Questo attributo può essere utilizzato come precondizione per le query per evitare situazioni del tipo **lost updates**.

I documenti sono memorizzati in **VelocyPack**, un formato binario che garantisce compattezza, portabilità e facilità di conversione in JSON. Tale formato è impiegato nei kernel dei database per eseguire query efficienti anche su sotto-documenti.

2.2.2 Collezioni

Le **collezioni** sono aggregati di documenti che rappresentano entità simili, agevolando l'organizzazione efficace dei dati. È possibile creare indici sulle collezioni per ottimizzare le query. Ogni collezione è identificata da un ID univoco non modificabile dall'utente, espresso come intero a 64 bit.

Inoltre, è possibile definire uno schema di validazione per le collezioni, specificando quali attributi devono essere presenti e quali tipi devono avere i valori. Questa funzionalità garantisce la coerenza dei dati all'interno del database.

2.2.3 Database

I **database** consistono in insiemi di collezioni, rappresentando entità correlate. Le query coinvolgono solo collezioni all'interno dello stesso database. Ogni istanza di ArangoDB include un database `_system`, immutabile e generato dal server il quale viene usato per la gestione dei database.

A livello fisico, i database sono salvati in file .sst che possono contenere documenti di diverse collezioni.

2.3 Modello a Grafo

ArangoDB gestisce i grafi attraverso la rappresentazione **Property Graph**, consistente nella creazione di due tipologie di oggetti: i vertici e gli archi. Tale operazione si concretizza mediante l'utilizzo dei documenti JSON, che costituiscono il fondamento di ArangoDB. In tal modo, si offre la possibilità di generare nodi e archi con un numero arbitrario di proprietà, assegnando a ciascun documento un identificativo univoco. Si distinguono, in particolare, due categorie di documenti:

- **Vertex**: questi documenti rappresentano i nodi del grafo.
- **Edge**: questi documenti rappresentano gli archi del grafo.

Gli Edge includono attributi speciali come `_from` e `_to`, destinati a indicare rispettivamente il nodo di partenza e quello di arrivo dell'arco, pertanto conferendo al grafo un orientamento. È tuttavia possibile specificare, durante le interrogazioni, la volontà di esplorare il grafo rispettivamente seguendo la direzione (`INBOUND`), nella direzione opposta (`OUTBOUND`), oppure ignorando completamente la direzione (`ANY`).

Nota 2 *I grafi in ArangoDB per risultare flessibili utilizzano una soluzione che aumenta le operazioni di referencing.*

Gli archi possono essere raggruppati in collezioni di tipo edge, alle quali viene assegnato un nome. Quest'operazione è svolta allo scopo di attribuire significati differenti agli archi.

Oltre alla rappresentazione dei dati, ArangoDB mette a disposizione degli algoritmi per eseguire varie operazioni sui grafi, tra cui:

- Attraversamento di un grafo.
- Ricerca di percorsi e percorsi minimi.

Inoltre, sono definite numerose altre funzioni per la manipolazione dei dati a grafo, come quelle per calcolare la centralità dei nodi o il vicinato, e così via.

Si ha la possibilità di specificare due tipologie di grafo: **Unmanaged graph** e **Managed graph**. Nel primo caso, durante le interrogazioni è necessario specificare le collezioni coinvolte, mentre nei secondi, noti anche come **named graph**, ciò non è richiesto, in quanto è sufficiente specificare solamente il nome del grafo. Inoltre, questa tipologia garantisce l'integrità del grafo sia durante l'inserimento che durante la rimozione di vertici o archi. Tale operazione è gestita da Arango, anche se comporta un costo in termini di performance.

Esistono diverse tipologie di grafi managed:

- **General Graphs**: consigliati per applicazioni con grafi di piccola taglia.
- **Smart Graphs**: ottimizzano la distribuzione dei dati, sfruttando le comunità presenti nel grafo.
- **Enterprise Graphs**: utilizzati per la gestione di grafi di grandi dimensioni a livello aziendale.
- **Satellite Graphs**: replicano il grafo su più macchine al fine di eseguire le interrogazioni in modo più efficiente.

Oltre a queste categorie, è possibile definirne altre proprie.

Un'ulteriore caratteristica di rilievo è che ArangoDB offre un'interfaccia web, attraverso la quale è possibile gestire il database nella sua interezza. Permette inoltre di visualizzare la struttura del grafo presente nel database, come riportato in figura 2.1.

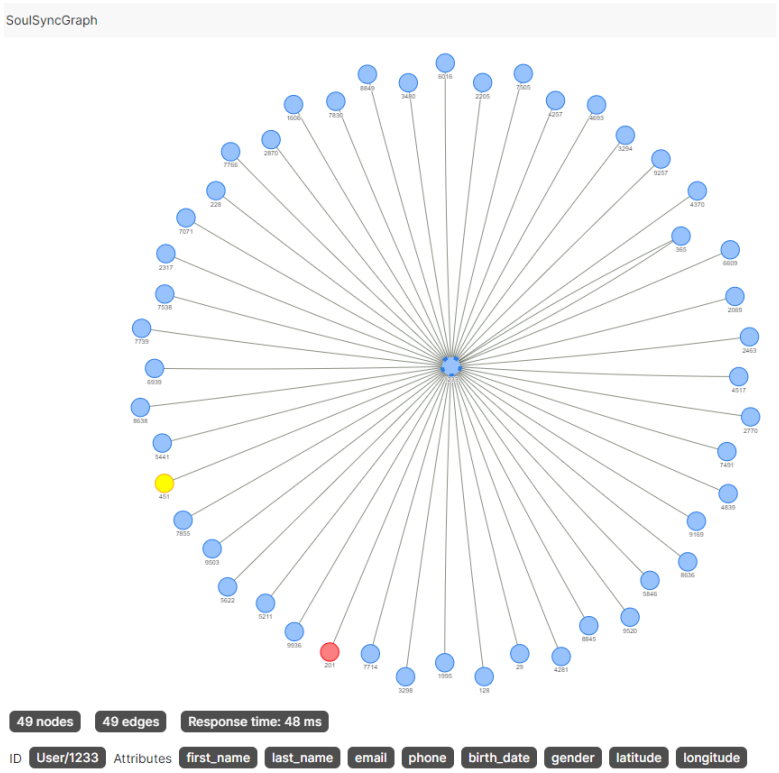


Figura 2.1: Interfaccia grafica dove in blu sono rappresentati gli utenti, in giallo l'università e in rosso il colore preferito.

2.4 Linguaggio di interrogazione

Le informazioni archiviate in un database ArangoDB sono accessibili tramite interrogazioni formulate in **AQL** (Arango Query Language).

AQL è un linguaggio dichiarativo, analogo a SQL, progettato per essere facilmente comprensibile e indipendente dall'ambiente di sviluppo. Una delle sue peculiarità è la capacità di interrogare i diversi modelli dati implementati da ArangoDB.

Tuttavia, AQL non supporta operazioni di definizione dei dati, come la creazione o l'eliminazione di collezioni, indici o database. Si limita esclusivamente alla manipolazione dei dati all'interno del database.

Il processo di esecuzione delle query inizia con l'invio della query AQL dal client al server. Successivamente, il server ArangoDB riceve la query, la analizza e la esegue. Una volta completata l'esecuzione, il server elabora il risultato e lo trasmette al client. Nel caso di un errore nella query, il server notifica il client con un messaggio di errore.

Le query AQL possono restituire un insieme di risultati o eseguire operazioni di manipolazione dei dati che non restituiscono alcun risultato. Queste ultime possono specificare una singola manipolazione alla volta, altrimenti il parser segnala un errore.

AQL consente di esprimere concetti in modo simile a SQL, ma con differenze sintattiche significative. Le parole chiave forniscono al parser informazioni sulle operazioni desiderate, con alcune keyword specifiche per il modello dati utilizzato. Un esempio sono le keyword per specificare la direzione in cui si navigano gli archi del grafo.

Oltre alle keyword, AQL supporta l'uso di variabili, che sono case-sensitive e specifiche per il contesto. Esse includono `CURRENT`, `NEW`, e `OLD`. La keyword `LET` consente di definire variabili per salvare risultati intermedi delle query.

Per ragioni di sicurezza, è consigliabile separare la query dai valori letterali, utilizzando i **bind parameters**. Questi parametri possono essere inseriti ovunque sia possibile utilizzare un letterale nella query e permettono di trasformare costanti in variabili, incluso il riferimento alle collezioni.

Come in SQL, è possibile effettuare subqueries racchiudendole tra parentesi tonde o assegnando il risultato a delle variabili. Il risultato di ogni query è sempre una lista di valori, anche se il risultato è un singolo elemento, mentre le query che modificano i dati di una collezione non restituiscono nulla.

In conclusione, AQL offre un potente strumento per interrogare e manipolare i dati all'interno di un database ArangoDB, con un linguaggio dichiarativo simile a SQL ma con caratteristiche specifiche e distintive.

2.5 Distribuzione

ArangoDB si conforma al **CAP Theorem**, priorizzando la consistenza e la tolleranza al partizionamento a discapito della disponibilità. Ciò implica che in caso di partizionamento della rete, il database privilegia la consistenza rispetto alla disponibilità.

L'architettura di ArangoDB si basa su una struttura "master-master", dove i client possono inviare richieste a un nodo arbitrario e ottenere le stesse informazioni da tutti.

Un **cluster** è composto da un numero finito di istanze di ArangoDB che comunicano tra loro tramite rete. La configurazione del cluster è mantenuta e salvata in un'Agency, che utilizza un meccanismo key-value per tenere traccia delle istanze in esecuzione e utilizza il protocollo di consenso **RAFT Consensus Protocol**.

- **Agents**: uno o più agenti costituiscono l'Agency ed è responsabile di mantenere la configurazione del cluster.
- **DBServers**: I nodi che contengono i dati. Ogni nodo utilizza una replicazione sincrona, dove ogni scrittura viene replicata su tutti i nodi. Ogni nodo può assumere il ruolo di **leader** o **follower**, con il leader che riceve le richieste di scrittura e le replica agli altri nodi. I dati non sono accessibili direttamente, ma solo attraverso i coordinators.
- **Coordinators**: i componenti accessibili esternamente, responsabili dell'esecuzione delle query. Ogni coordinatore conosce la localizzazione dei dati e può eseguire le query dopo un'ottimizzazione preliminare.

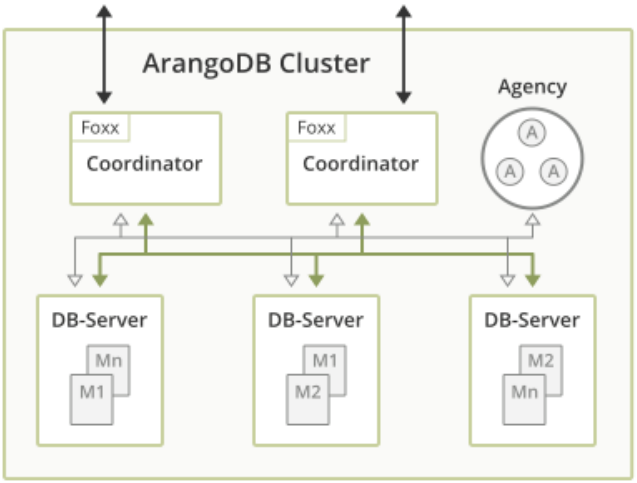


Figura 2.2: Architettura di un cluster ArangoDB

In contesto distribuito, ArangoDB agevola la realizzazione di repliche sincrone, consentendo la duplicazione dei dati su ciascun nodo a ogni operazione di scrittura.

Il modello organizzativo adottato per tali operazioni, noto come **sistema leader-follower**, implica l'assegnazione di un nodo principale, denominato leader, il quale riceve le richieste di scrittura dal Coordinator. Il leader, a sua volta, si incarica di instradare tali richieste ai nodi del cluster che ospitano le repliche.

Nel caso in cui un follower riscontri un malfunzionamento, il leader ne rileva l'evento e procede alla sua temporanea rimozione dopo un intervallo di 3 secondi, proseguendo con le operazioni correnti. Una volta ripristinato, il nodo follower precedentemente rimosso viene automaticamente ri-sincronizzato.

Qualora il leader stesso subisca un guasto, l'Agency attiva automaticamente una procedura di failover entro 15 secondi, mirante a promuovere uno dei follower a nuovo leader. Gli altri nodi si sincronizzano automaticamente con il nuovo leader.

Queste operazioni vengono eseguite in modo trasparente al Coordinator, non influenzando il codice dell'utente.

Va notato che tale tolleranza ai guasti può comportare un rallentamento del servizio, poiché è necessario attendere il completamento delle operazioni di scrittura su tutti i follower.

Per distribuire i dati all'interno di un cluster, è possibile partizionare le collezioni in differenti shard. Questa suddivisione è gestita automaticamente da Arango, con le informazioni sulla distribuzione delle shard memorizzate nelle Agency e condivise con i Coordinator.

Arango determina la collocazione dei dati in ciascun shard mediante funzioni di hash, calcolate utilizzando l'attributo `_key`.

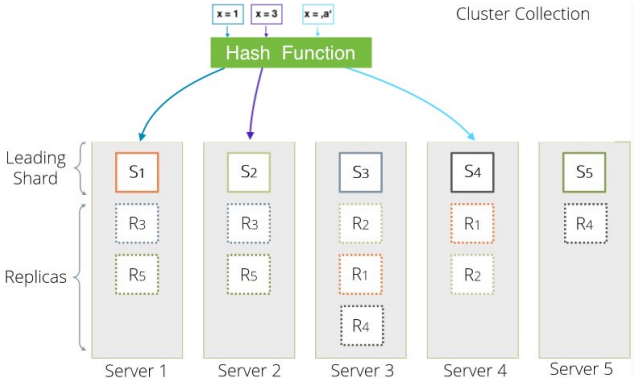


Figura 2.3: Distribuzione dei valori nel database

Per i database a grafo, ArangoDB solamente nella versione Enterprise, utilizza gli **SmartGraphs**. Questo metodo sfrutta un particolare attributo, `smartGraphAttribute`, come chiave per la suddivisione del grafo. Tale approccio consente di migliorare le prestazioni del database, riducendo il numero di operazioni che coinvolgono nodi diversi e garantendo che i vertici connessi siano memorizzati sullo stesso server.

In pratica, si passa da una situazione in cui lo sharding viene eseguito in modo "casuale", ottenendo un risultato come quello riportato in figura 2.4, a una situazione in cui i nodi connessi sono memorizzati nello stesso nodo, come mostrato in figura 2.5.

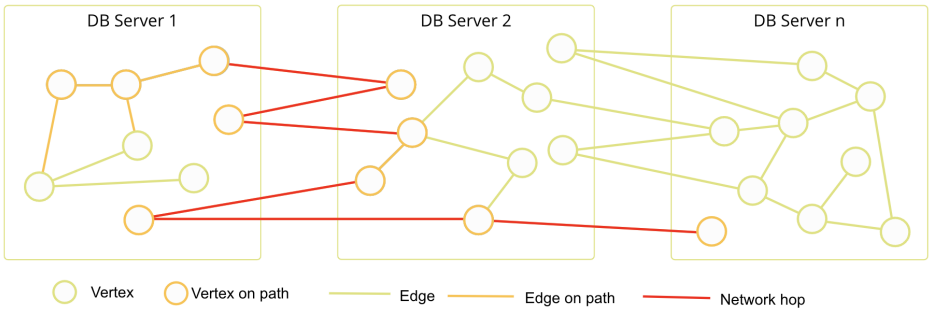


Figura 2.4: Distribuzione dei nodi nel database con sharding casuale

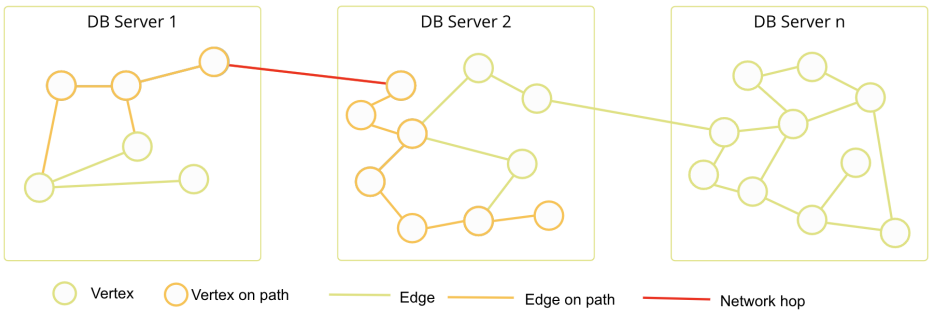


Figura 2.5: Distribuzione dei nodi nel database con SmartGraphs

Nota 3 *Non è stato possibile testare questa funzionalità in quanto è disponibile solamente nella versione a pagamento di ArangoDB.*

2.5.1 RAFT

RAFT è il protocollo utilizzato da ArangoDB per gestire la distribuzione ed è basato sul consenso. Ogni volta che si arriva ad una decisione essa sarà finale. Le decisioni vengono prese a votazione, ottenendo il $50\% + 1$ dei voti. Nel caso di ArangoDB, per prendere la decisione di trasferire un Leader dopo un guasto, è necessario avere almeno due voti favorevoli su tre Agency, anche nel caso in cui ci siano guasti alle Agency stesse. Dal momento in cui più della metà delle Agency falliscono non sarà più in grado di prendere decisioni. Una visualizzazione dell'algoritmo è presente al seguente link (RAFT).

2.6 Transazioni

Nell'impiego di ArangoDB nella sua configurazione centralizzata, è garantito che le query che coinvolgono più documenti o collezioni soddisfino le proprietà ACID (Atomicità, Coerenza, Isolamento, Durabilità). Tuttavia, nell'ambito della versione distribuita, mentre le query che operano su singoli documenti mantengono le predette proprietà, quelle coinvolgenti più documenti o collezioni non le garantiscono.

Nel contesto distribuito, una transazione segue un'architettura distribuita, partendo dal Coordinator il quale converte la transazione in una per ciascun nodo DBServer coinvolto.

Il metodo impiegato per la gestione delle transazioni dipende dal tipo di Storage Engine utilizzato. Nel nostro caso, adottiamo RocksDB, che supporta transazioni distribuite.

Tutte le collezioni utilizzate in modalità di scrittura, lo storage-engine applica un lock condiviso. Ciò permette che le operazioni di scrittura su documenti differenti siano eseguite in parallelo, così come le operazioni di lettura. Tuttavia, se due transizioni simultanee tentano di scrivere sullo stesso documento, una di esse viene interrotta. Per quanto riguarda le operazioni di lettura è possibile acquisire un lock esclusivo su richiesta.

È possibile accedere alle collezioni in modalità **esclusiva**, acquisendo un write lock sulla collezione stessa, che impedisce l'esecuzione di operazioni di scrittura in parallelo, pur consentendo operazioni di lettura simultanee.

2.7 Isolamento

Lo storage engine RocksDB, impiegato da ArangoDB, offre un meccanismo di isolamento a snapshot. In pratica, tutte le operazioni all'interno di una transazione visualizzano la stessa versione dei dati, indipendentemente da eventuali modifiche apportate.

Lo snapshot viene creato all'avvio della transazione e mantenuto per tutta la sua durata. Durante la creazione, non vengono applicati lock per il suo mantenimento, consentendo così l'esecuzione di operazioni senza interruzioni da altre transazioni, a meno che queste ultime non tentino di scrivere sullo stesso documento. A livello distribuito, ogni nodo DBServer mantiene il proprio snapshot.

2.8 Deadlock detection

ArangoDB dispone di un meccanismo per il rilevamento dei deadlock, che termina una delle transazioni coinvolte. Questo meccanismo viene attivato soltanto quando viene acquisito un lock su una collezione in modalità **esclusiva**.

2.9 Prevenzione dei guasti

In caso di guasto, è possibile recuperare lo stato precedente poiché RocksDB utilizza un meccanismo di **write-ahead logging** per garantire la persistenza dei dati. Tale meccanismo comporta la registrazione di ogni operazione di scrittura in un log prima della sua esecuzione.

Questo file di log può anche essere impiegato per la gestione delle repliche, garantendo che i follower mantengano la stessa sequenza di operazioni del leader. Inoltre, il file contiene esclusivamente le transazioni completate, evitando la presenza di transazioni parziali.

Capitolo 3

Esperimenti

In seguito allo studio delle caratteristiche di ArangoDB, si è deciso di effettuare una serie di esperimenti per valutare le prestazioni del database in diverse situazioni. In particolare, si è deciso di confrontare le prestazioni di ArangoDB in un contesto centralizzato e distribuito, in modo da valutare le differenze dei due approcci.

Per effettuare i test è stata utilizzata l'immagine Docker ufficiale di ArangoDB, disponibile su Docker Hub, con il supporto di script python per l'automazione delle operazioni.

Per quanto riguarda il database centralizzato, è stato creato un container Docker con ArangoDB in modalità single server, mentre per il database distribuito sono stati creati sette container Docker, di cui:

- 1 container come Agency.
- 2 container come Coordinators.
- 4 container come DBServers.

Di seguito sono riportati i dettagli degli esperimenti effettuati e i risultati ottenuti. In particolare, sono stati effettuati test per le operazioni di inserimento, lettura, modifica e cancellazione di dati, in modo da valutare le prestazioni di ArangoDB in diversi scenari.

Inoltre, per quanto riguarda la versione distribuita, sono stati effettuati test per valutare le prestazioni in caso di guasto di un nodo. Questo è stato fatto simulando il fallimento di un nodo DBServer e valutando le prestazioni del database in seguito al guasto e al partizionamento della rete.

Nota 4 *Per effettuare i test, è stato utilizzato un computer con le seguenti caratteristiche:*

- *Sistema operativo: Windows 11 Home*
- *Processore: Intel Core i5-1135G7*
- *RAM: 16 GB*
- *Storage: SSD 512 GB*

Inoltre, è stato utilizzato Docker versione 26.0.0 e ArangoDB versione 3.12.0.

Nota 5 *Tutti i comandi per replicare il nostro studio sono presenti all'interno del file readme.*

3.1 Database centralizzato

Il primo passo è stato quello di valutare il database in versione centralizzata in modo da avere a disposizione dei valori di confronto per quello distribuito. Di seguito sono riportati i risultati degli esperimenti effettuati con ArangoDB in modalità centralizzata.

3.1.1 Inserimenti

L'inserimento dei dati è stato realizzato uno script python che ha permesso di effettuare le operazioni in modo automatico.

Gli esperimenti di caricamento sono stati eseguiti utilizzando 5 thread in parallelo, ciascuno dei quali si occupava dell'inserimento di 100 record alla volta. Questo approccio è stato scelto a seguito di diversi esperimenti

preliminari, che hanno mostrato che l'inserimento di pochi record alla volta provoca un rallentamento delle prestazioni a causa dell'overhead introdotto dal database, in quanto ogni operazione di inserimento deve essere confermata. Inoltre, questo approccio ha permesso di verificare le capacità del database di gestire operazioni di scrittura effettuate da più client contemporaneamente.

La fase di inserimento è stata valutata aumentando progressivamente il numero di record inseriti nel database. In particolare, sono stati eseguiti test con 5000, 10000, 25000, 50000, 100000 record per i nodi. Per gli archi, che sono in quantità molto maggiore, si è scelto di effettuare test con 10000, 50000, 100000, 250000, 500000, 1000000 record.

Tutti questi test sono stati ripetuti 10 volte per ogni quantità di record, al fine di ottenere una media dei risultati.

Oltre a questo, per ridurre l'overhead introdotto da database è stato specificato il parametro `sync` che non aspetta le risposte del server alla fine delle operazioni di caricamento. Nei grafici riportati nelle figure 3.2 e 3.1 è riportato il confronto tra i tempi di caricamento quando il parametro viene utilizzato e non. In particolare, si può notare che l'utilizzo di questo parametro permette di ridurre i tempi richiesti per effettuare il caricamento dei dati.

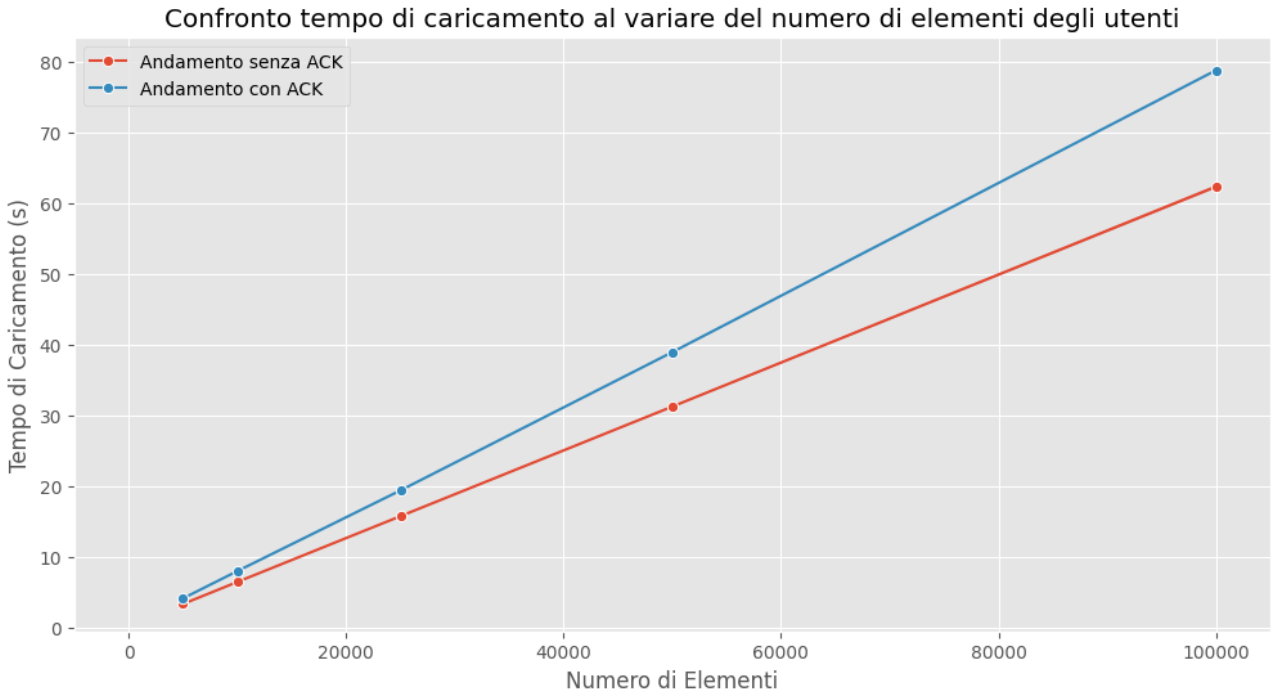


Figura 3.1: Confronto tempi di caricamento con i valori del parametro `sync`. Caricamento di 10^5 utenti.

Inoltre in entrambi i casi possiamo notare un andamento lineare ma con scale differenti a causa del numero degli archi che è superiore rispetto a quello dei nodi. Risulta importante osservare che, nonostante il numero di archi sia nettamente maggiore, essi sono caratterizzati da un numero minore di proprietà rispetto ai nodi il che giustifica la differenza minore rispetto a quella attesa se si va a considerare che gli archi sono 10^6 mentre i nodi sono 10^5.

3.1.2 Letture

Sulla base dei dati selezionati per i test, le operazioni di lettura sono state effettuate con query progettate per rappresentare situazioni reali. Inoltre, si è voluto valutare il comportamento del database nella gestione di query complesse che coinvolgono la struttura a grafo del database.

Nota 6 *Tutte le query sono scritte in AQL ed eseguite tramite la libreria Python-Arango.*

Nonostante sia possibile ottenere informazioni sulle performance di esecuzione delle query direttamente dal database specificando il metodo `getExtra()`, si è scelto di valutare le performance delle query all'interno di uno script Python per simulare un'applicazione reale.

Le query, prima di essere eseguite, passano attraverso un ottimizzatore che crea diversi piani di esecuzione, associando a ogni operazione dei costi e selezionando il piano di esecuzione per costo minimo. Le ottimizzazioni

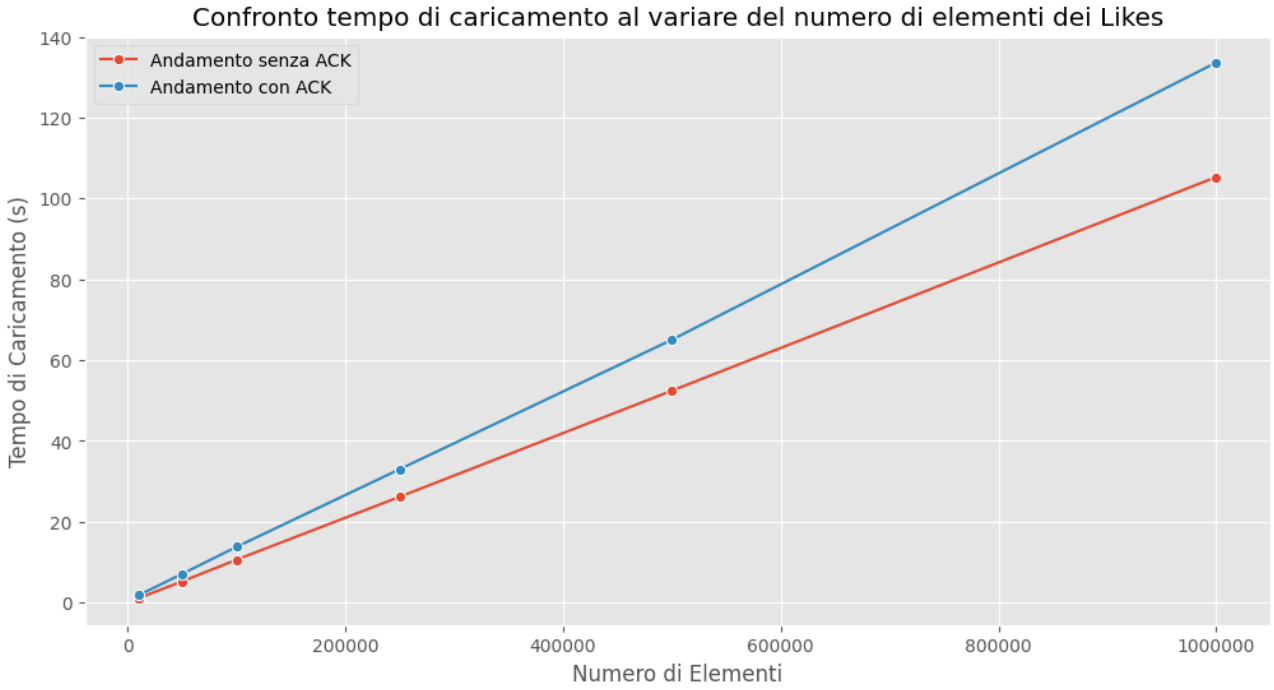


Figura 3.2: Confronto tempi di caricamento con i valori del parametro `sync`. Caricamento di 10^6 archi.

vengono effettuate senza modificare il risultato finale della query, ma possono alterare l'ordine di restituzione dei risultati, a meno che non sia specificato un ordine particolare.

Attraverso il comando `EXPLAIN` è possibile ottenere il piano di esecuzione della query, che include le operazioni eseguite, il costo di ciascuna e il tempo di esecuzione previsto.

Il costo associato alle operazioni si basa su euristiche codificate all'interno del database.

Nota 7 *Alcune regole dell'ottimizzatore possono essere modificate.*

ArangoDB fornisce metodi per fare il profiling delle query, permettendo di ottimizzare le performance attraverso interventi mirati.

Vediamo ora le query, scritte in AQL, che sono state eseguite per valutare il database. In particolare, sono state eseguite per selezionare i campi degli utenti.

```
FOR u IN User
    FILTER u._key == @user
RETURN u
```

Listing 3.1: Ottenere tutte le informazioni sull'utente @user

Nota 8 *La variabile `@user` rappresenta l'identificativo dell'utente.*

Si è deciso di eseguire anche query più complesse che coinvolgessero la struttura a grafo del database. In particolare, sono state effettuate le seguenti query:

```
FOR category, e IN 1..1 OUTBOUND @user IntMovieCategory
    FOR user, edge IN 1..1 INBOUND category IntMovieCategory
        FILTER user != @user
RETURN category
```

Listing 3.2: Trovare tutti gli utenti che hanno la categoria di film preferita in comune con l'utente @user

```
FOR city, e IN 1..1 OUTBOUND @user LivesIn
    FOR user, edge IN 1..1 INBOUND city LivesIn
        FILTER user != @user
RETURN user
```

Listing 3.3: Trovare tutti gli utenti che vivono nella stessa città dell'utente @user

```
FOR category, e IN 1..1 OUTBOUND @user IntMovieCategory
  FOR user1, e1 IN 1..1 INBOUND category IntMovieCategory
    FILTER user1 != @user
  FOR colleague, e2 IN 1..1 OUTBOUND @user StudiesAt
    FOR user2, e3 IN 1..1 INBOUND colleague StudiesAt
      FILTER user2 != @user
  FILTER user1 == user2
RETURN user1
```

Listing 3.4: Trovare tutti gli utenti che hanno la categoria di film preferita in comune con l'utente @user e che hanno frequentato la stessa università.

```
    FOR city, edge IN 1..1 ANY  @country LocatedIn
          FOR user, edge1 IN 1..1 ANY city LivesIn
    RETURN {"User": user._id, "City": city.name}
```

Listing 3.5: Trovare tutti gli utenti che vivono nella nazione @country.

Considerando che i database a grafo sono più efficienti rispetto a quelli relazionali nella gestione di query del tipo "trovare tutti gli amici degli amici di un utente", si è deciso di eseguire query basate su questa caratteristica per valutare le performance di ArangoDB in un contesto di database a grafo.

```
    FOR v, e IN 1..1 ANY @user Matches
          FOR v1, e1 IN 1..1 OUTBOUND v Likes
    RETURN v1
```

Listing 3.6: Trovare tutti i like degli utenti che hanno fatto match con l'utente @user

Le query appena presentate sono state eseguite 30 volte attraverso l'utilizzo della libreria python-arango. Questo procedimento ha permesso di ottenere dei risultati che sono stati utilizzati nella sezione 3.2.2 per valutare le capacità di ArangoDb nel servire le interrogazioni a livello distribuito.

3.1.3 Modifiche

L'operazione di aggiornamento (`update`) consente di modificare parzialmente un documento, modificando gli attributi esistenti o aggiungendone di nuovi. Ogni operazione di aggiornamento può influire al massimo su una singola collezione alla volta, e il nome di tale collezione non può essere dinamicamente associato attraverso una variabile.

È importante notare che durante le operazioni di aggiornamento non è possibile modificare i campi `_key`, `_id`, `_rev`. Per quest'ultimo attributo, il server genera automaticamente un nuovo valore ad ogni modifica. Tuttavia, è possibile modificare i campi `_from` e `_to`.

La sintassi per eseguire l'aggiornamento è la seguente:

`UPDATE <document> IN <collection>` oppure
`UPDATE <keyExpression> WITH <document> IN <collection>`

Entrambe le varianti possono essere seguite da un'opzione `OPTIONS` per specificare le modalità di esecuzione della query. In entrambi i casi, `<document>` contiene gli attributi da modificare, mentre `<collection>` è il nome della collezione da aggiornare.

Esiste un ulteriore metodo per effettuare l'aggiornamento di un documento, chiamato metodo `replace`. Questo metodo sostituisce completamente il documento con uno nuovo, ad eccezione degli attributi `_key`, `_id`, `_rev`. La sintassi di questo metodo è simile a quella dell'operazione `update`, con la sola modifica della parola chiave.

Le opzioni che possono essere specificate alla fine della query sono in formato JSON e includono:

- `ignoreErrors`: un attributo booleano che, se impostato su true, consente di ignorare gli errori che si verificano durante l'esecuzione della query.
- `keepNull`: un attributo booleano che, se impostato su true, mantiene i valori nulli durante l'aggiornamento.
- `mergeObjects`: un attributo booleano che, se impostato su true, unisce gli oggetti in modo ricorsivo.
- `waitForSync`: un attributo booleano che, se impostato su true, fa sì che la query attenda la sincronizzazione dei dati su disco prima di restituire il risultato.
- `ignoreRevs`: un attributo booleano che, se impostato su true, permette di ignorare i controlli di revisione del documento, in particolare l'attributo `_rev`.

- `exclusive`: un attributo booleano che, se impostato su true, consente di eseguire la query in modo esclusivo, impedendo l'accesso concorrente alla collezione. In pratica, acquisisce un lock sulla collezione e lo rilascia solo alla fine dell'operazione.
- `refillIndexCaches`: un attributo booleano che, se impostato su true, permette di ricaricare la cache degli indici prima di eseguire la query.
- `versionAttribute`: un attributo stringa che permette di specificare l'attributo da utilizzare come versione del documento. Se non specificato, viene utilizzato l'attributo `_rev`.

In un ambiente di database centralizzato, le operazioni di aggiornamento sono eseguite secondo una politica **"all or nothing"**, effettuando un rollback generale in caso di fallimento. Le query possono però eseguire dei commit intermedi. Così facendo, nel caso in cui la query raggiunga una specifica soglia, in caso di fallimento, vengono effettuati solamente i rollback delle operazioni di cui non è ancora stato effettuato il commit.

Per testare le operazioni di aggiornamento, sono state eseguite diverse tipologie di query:

- Query di aggiornamento che modificano i campi di un singolo documento.

```
1    UPDATE {_key: @city } WITH { lat:@lat, long: @long} IN City
2
```

Listing 3.7: Query di aggiornamento di un singolo documento

- Query che modificano le relazioni tra i nodi, ovvero gli archi del grafo.

```
1    WITH LivesIn
2    FOR edge in LivesIn
3        FILTER edge._from == @user
4    UPDATE {_key: edge._key} WITH {_to: @city} IN LivesIn
5
```

Listing 3.8: Query di aggiornamento delle relazioni tra i nodi

- Query che sfruttano l'operatore `replace` per sostituire completamente un documento con uno nuovo.

```
1    WITH User
2    REPLACE {_key: @user} WITH @doc IN User
3
```

Listing 3.9: Query di aggiornamento con replace

3.1.4 Cancellazioni

Le operazioni di cancellazione possono essere eseguite mediante l'utilizzo dell'operatore `REMOVE` in una query AQL. Questo operatore consente di eliminare uno o più documenti da una collezione. Analogamente alle operazioni di modifica, la cancellazione può essere effettuata solo su una collezione alla volta.

La sintassi per eseguire la cancellazione è la seguente:

```
REMOVE <keyExpression> IN <collection>
```

dove `<keyExpression>` rappresenta l'espressione che identifica il documento da rimuovere e `<collection>` indica il nome della collezione dalla quale rimuovere il documento.

Anche in questo caso, la query può essere seguita da un'opzione `OPTIONS` per specificare le modalità di esecuzione della query. Le opzioni disponibili includono:

- `ignoreErrors`: un attributo booleano che, se impostato su true, consente di ignorare gli errori che si verificano durante l'esecuzione della query.
- `waitForSync`: un attributo booleano che, se impostato su true, fa sì che la query attenda la sincronizzazione dei dati su disco prima di restituire il risultato.
- `ignoreRevs`: un attributo booleano che, se impostato su true, permette di ignorare i controlli di revisione del documento, in particolare l'attributo `_rev`.
- `exclusive`: un attributo booleano che, se impostato su true, consente di eseguire la query in modo esclusivo, impedendo l'accesso concorrente alla collezione. In pratica, acquisisce un lock sulla collezione e lo rilascia solo alla fine dell'operazione.

- `refillIndexCaches`: un attributo booleano che, se impostato su true, consente di ricaricare la cache degli indici prima di eseguire la query.

Nel contesto di database centralizzato, anche in questo caso le operazioni di cancellazione sono eseguite secondo la politica **"all or nothing"**.

Per le operazioni di cancellazione dei vertici all'interno del database a grafo, il DBMS non gestisce automaticamente l'eliminazione di tutti gli archi associati al vertice. Pertanto, è necessario effettuare manualmente l'eliminazione degli archi associati prima di procedere con la cancellazione del vertice. Alcune librerie che facilitano l'interazione con il database includono metodi specifici per gestire questa particolare casistica.

Per testare le operazioni di cancellazione, sono state eseguite query confrontando il comportamento delle query AQL scritte manualmente con quelle eseguite utilizzando i metodi offerti dalla libreria Python-Arango.

In particolare, sono state eseguite query per cancellare un utente dal grafo, ottenendo comportamenti differenti. Infatti, utilizzando la libreria `python-arango`, la cancellazione di un nodo comporta automaticamente la cancellazione degli archi associati. Questo comportamento non è garantito quando si scrive una query AQL manualmente. In quest'ultimo caso, è necessario cancellare manualmente gli archi associati prima di procedere con la cancellazione del nodo. Ad esempio, la seguente query non rimuove automaticamente tutti gli archi associati al nodo:

```
1    WITH User
2    FOR u IN User
3        FILTER u._id == @user
4    REMOVE u IN User
```

Listing 3.10: Query di cancellazione di un utente

Per ottenere un comportamento simile a quello della libreria Python-Arango, è necessario scrivere una serie di query che rimuovano gli archi dalle collezioni, come ad esempio:

```
1    WITH IntColor
2    FOR edge IN IntColor
3        FILTER edge._from == @user
4    REMOVE {_key: edge._key} IN IntColor
```

Listing 3.11: Query di cancellazione degli archi associati

e infine procedere con la cancellazione del nodo.

3.2 Database distribuito

Per quanto riguarda il database distribuito, sono stati condotti esperimenti simili a quelli effettuati per il database centralizzato. In particolare, sono stati eseguiti test sulle operazioni di inserimento, lettura, modifica e cancellazione di dati, al fine di valutare le prestazioni di ArangoDB in diversi scenari.

Gli esperimenti sono stati condotti con la seguente configurazione:

- 1 nodo Agency.
- 2 nodi Coordinator.
- 4 nodi DBServer.

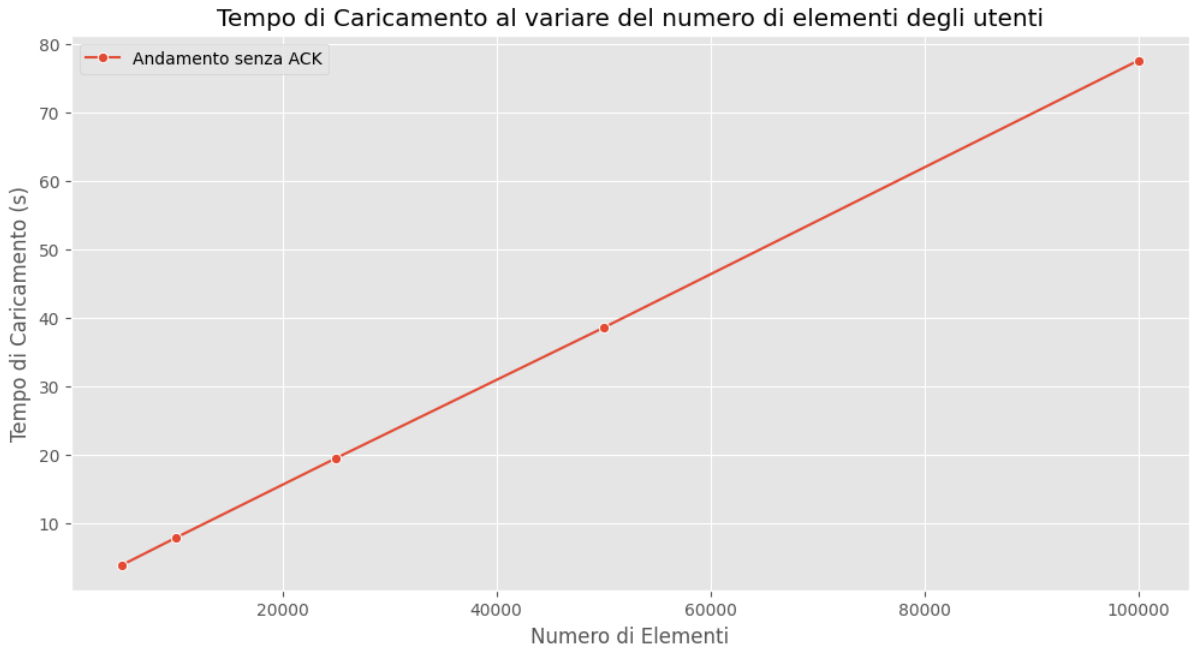
I dati sono stati suddivisi in 4 shard per distribuire il carico tra i nodi DBServer. Inoltre, sono state create delle repliche per garantire la ridondanza dei dati e la tolleranza ai guasti. Il numero di repliche è stato impostato a 3, per assicurare la disponibilità dei dati anche in caso di guasto di un nodo.

3.2.1 Inserimenti

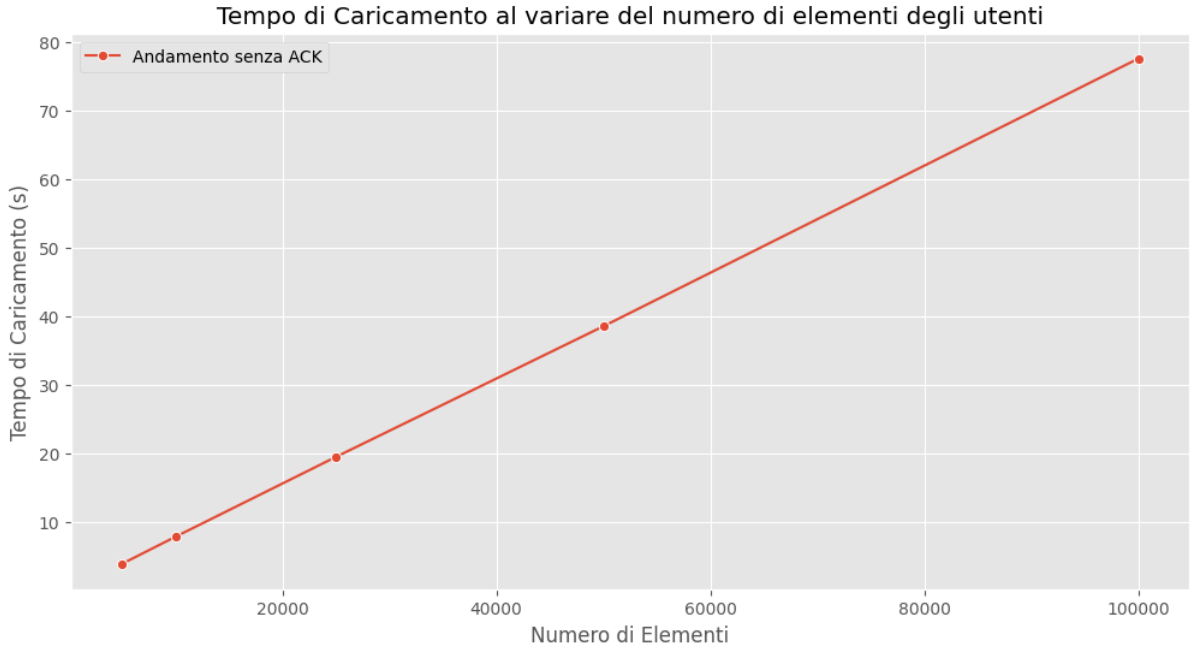
Le operazioni di inserimento dei dati sono state eseguite adottando la stessa logica utilizzata per il database centralizzato, al fine di confrontare le prestazioni di ArangoDB nelle modalità centralizzata e distribuita.

Questo approccio ha permesso di ottenere i risultati riportati in figura 3.3a per quanto riguarda i nodi, e in figura 3.3b per quanto riguarda gli archi.

Come possiamo osservare dai grafici, il tempo di caricamento dei dati segue un andamento lineare al crescere del numero di record inseriti. Questo comportamento è lo stesso osservato per il database centralizzato, come si può osservare in figura 3.4, ma con tempi di esecuzione più lunghi a causa della distribuzione dei dati tra i nodi DBServer e delle operazioni di replica. Nel nostro caso il fattore di replica è stato impostato a 3 e abbiamo specificato la conferma dell'avvenuta scrittura dopo che sono state eseguite almeno 2 scritture in modo



(a) Tempo per il caricamento di 10^5 utenti.



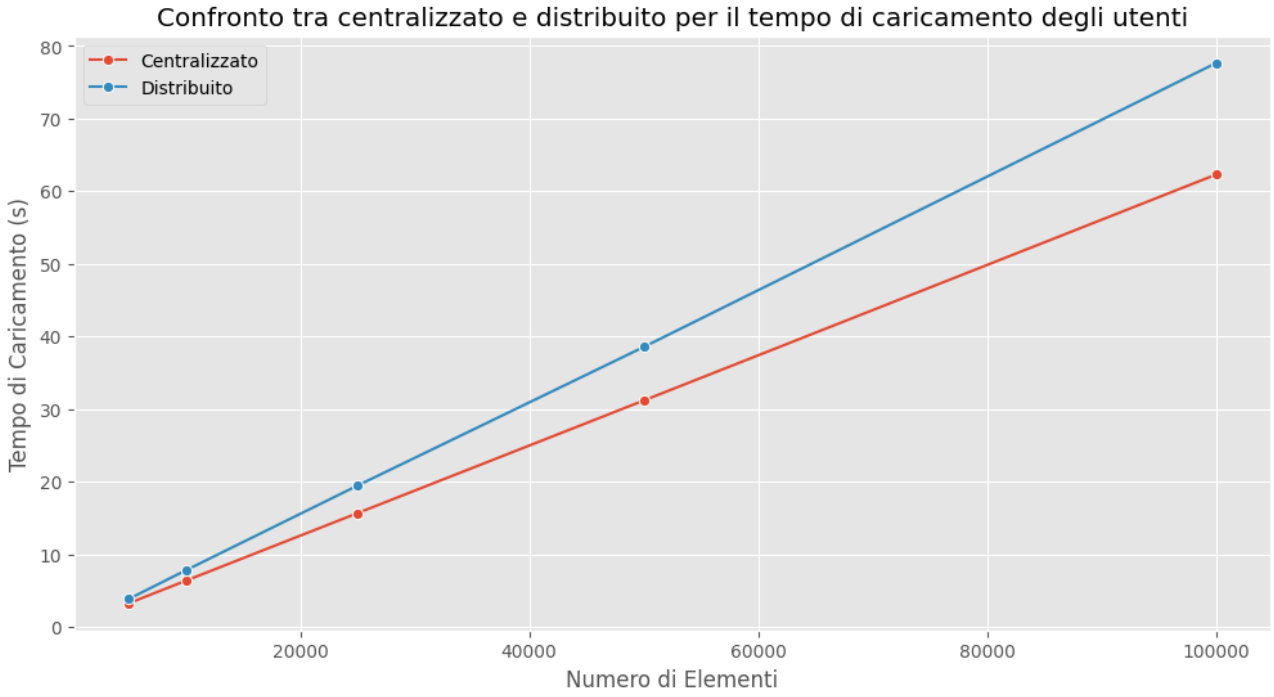
(b) empo per il caricamento di 10^6 Likes.

Figura 3.3: Tempi medi per il caricamento di nodi e archi sul database distribuito.

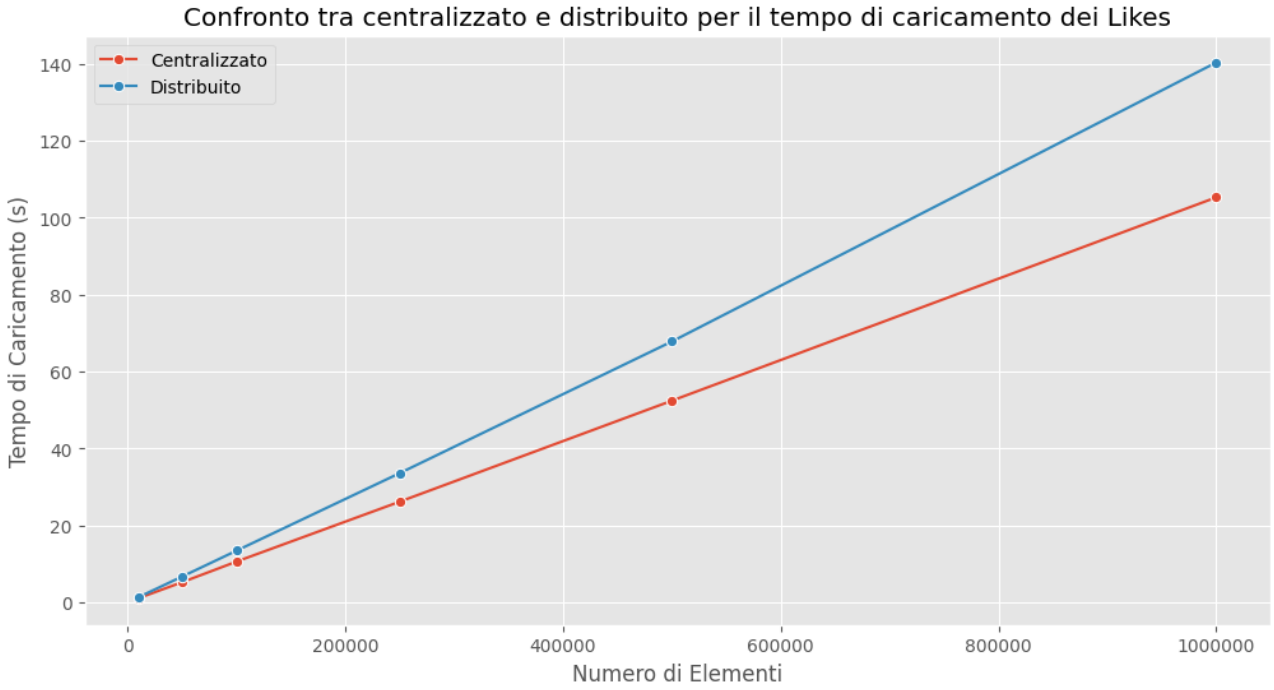
sincrono. Questo comportamento viene specificato alla creazione del grafo distribuito attraverso l'attributo `write_concern`. Per realizzare un confronto più approfondito, abbiamo effettuato i test modificando il valore associato a questo parametro. Nello specifico, abbiamo testato gli inserimenti impostando il valore a 1, 2, 3. In seguito visto il dominio in analisi, abbiamo deciso di migliorare le tempistiche effettuando una sola scrittura e lasciare ai DBServer il compito di sincronizzarsi, impostando questo parametro a 1.

I risultati di quest'ultimo esperimento sono riportati in figura 3.5. Da questi risultati si nota come per i diversi valori del parametro `write_concern` il tempo richiesto per il caricamento non viene influenzato.

Sfruttando l'interfaccia web dei Coordinator, è possibile monitorare la distribuzione dei dati tra i nodi DBServer e verificare il corretto funzionamento della replica dei dati. Tale interfaccia fornisce grafici a torta, riportati in figura 3.6, che mostrano la distribuzione dei dati tra i nodi DBServer e il numero di repliche presenti



(a) Confronto nel caricamento dei nodi.



(b) Confronto nel caricamento degli archi.

Figura 3.4: Centralizzato vs. Distribuito.

per ciascun nodo. Oltre alla verifica della distribuzione degli sharding abbiamo provato a verificare l'attuale contenuto, ma non è stato possibile verificarlo. È possibile verificare l'identificativo dello shard memorizzato su uno specifico DBServer. Inoltre, utilizzando AQL non è possibile specificare il nodo su cui eseguire le query.

Dai grafici possiamo osservare che i dati sono distribuiti in modo abbastanza omogeneo tra i nodi DBServer. Tuttavia esistono casi in in cui la distribuzione tra i vari server non riesce ad essere omogena, ad esempio in caso di guasti, in cui deve essere forzata manualmente la ridistribuzione dei dati tramite l'interfaccia grafica. Inoltre, è possibile distinguere tra i nodi leader e i nodi follower, che contengono una replica dei dati consentendo di verificare la distribuzione degli shard in maniera approfondita.

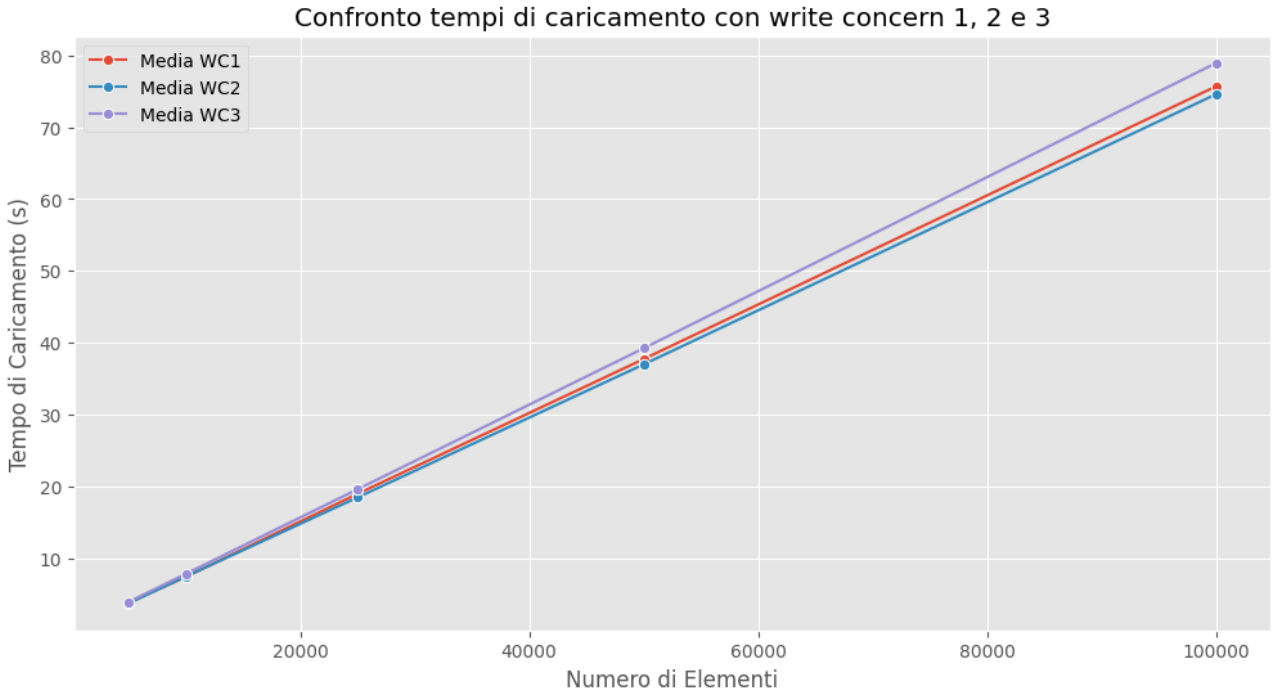


Figura 3.5: Caricamento dei nodi con il parametro `write_concern`

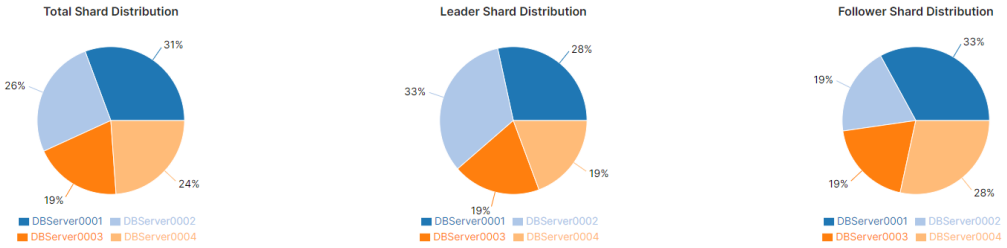


Figura 3.6: Distribuzione dei dati tra i nodi DBServer

3.2.2 Letture

Quanto presentato in precedenza per il database centralizzato riguardo le operazioni di lettura, è valido anche per il database distribuito. Tuttavia, esistono alcune differenze, soprattutto per quanto riguarda l'ottimizzazione delle query.

In un database distribuito, il parsing delle query viene effettuato dal Coordinatore. Una volta ottenuto il piano di esecuzione, il Coordinatore invia le operazioni ai nodi DBServer per l'esecuzione.

L'ottimizzatore cerca di ridurre la quantità di dati trasferiti tra i nodi, eseguendo le operazioni di filtraggio e proiezione il più vicino possibile ai dati.

Dopo questa breve introduzione, passiamo all'esperimento effettuato. Per poter confrontare i risultati rispetto alla modalità centralizzata, sono state eseguite le stesse query in entrambi i contesti. Inoltre per verificare le capacità del database di scalare, abbiamo deciso di variare anche il numero di nodi.

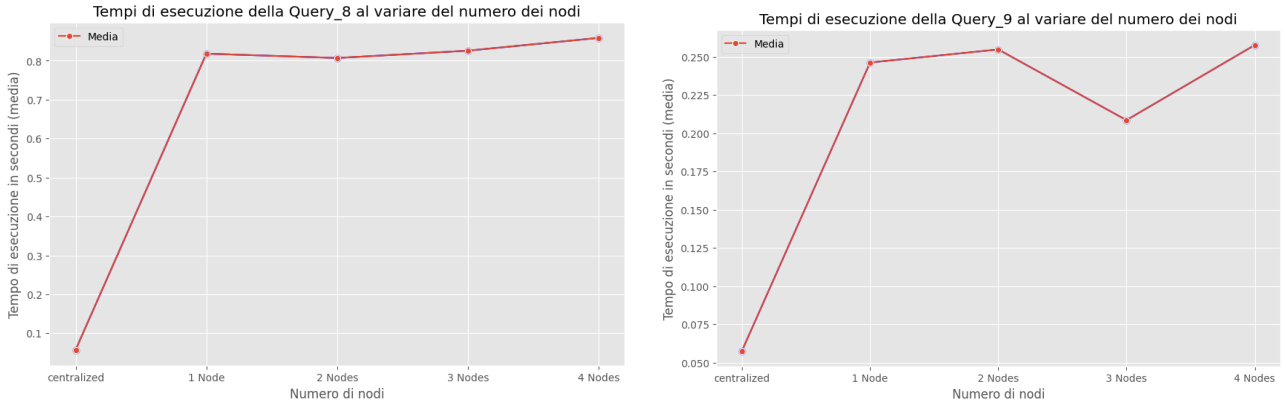
Per quanto riguarda le interrogazioni distribuite, ArangoDB offre un livello di trasparenza molto elevato in quanto all'utente non è richiesto di sapere come sono distribuiti i dati, ma è necessario specificare in anticipo le collezioni da utilizzare nella query, in modo da informare il Coordinatore sui nodi DBServer che dovranno essere coinvolti nell'esecuzione della query. Questo viene fatto tramite il comando `WITH`, ottenendo una query del tipo:

```
1       WITH User
2       FOR u IN User FILTER u._key == @user RETURN u
```

Listing 3.12: Ottenere tutte le informazioni sull'utente @user

Relativamente allo studio sulla scalabilità delle operazioni di lettura, abbiamo eseguito le query per 30 volte al fine di ottenere risultati robusti in termini di tempistiche. Questo esperimento ha evidenziato delle difficoltà

da parte di ArangoDB nella gestione della scalabilità del modello a grafo. In particolare, utilizzando lo sharding di default, si è osservato che i tempi di esecuzione delle query tendono ad aumentare con l'incremento del numero di DBServer. Analizzando il grafico riportato nella figura 3.7, possiamo notare tale fenomeno. I boxplot mostrano chiaramente che la media dei tempi di esecuzione delle query cresce al numero di nodi.



(a) Trovare il numero di utenti per ogni città.

(b) Trovare tutti gli utenti dell'app.

Figura 3.7: Confronto tempi di esecuzione delle letture con diversi nodi.

Come evidenziato dai grafici, i tempi di esecuzione delle query nel contesto distribuito risultano sempre superiori rispetto a quelli osservati nel contesto centralizzato. Questo fenomeno suggerisce che la struttura distribuita di ArangoDB comporta un rallentamento nei tempi di esecuzione delle query. Inoltre, l'impossibilità di verificare la distribuzione dei dati tra i vari nodi complica l'identificazione delle cause di questo rallentamento. Possiamo inoltre osservare che le query che vanno ad interrogare la struttura a grafo non sono in grado di scalare a differenza di quelle che interrogano una singola collezione.

3.2.3 Modifiche

Per questa tipologia di operazioni, il comportamento di ArangoDB in modalità distribuita è simile a quello del database centralizzato. Tuttavia, anche in questo caso, è necessario specificare in anticipo le collezioni coinvolte nella query, in modo da informare il Coordinatore sui nodi DBServer che dovranno essere coinvolti nell'esecuzione della query.

3.2.4 Cancellazioni

Anche per le operazioni di cancellazione, il comportamento di ArangoDB in modalità distribuita è simile a quello del database centralizzato. Tuttavia, è necessario specificare in anticipo le collezioni coinvolte nella query, in modo da informare il Coordinatore sui nodi DBServer che dovranno essere coinvolti nell'esecuzione della query.

Capitolo 4

Tolleranza ai guasti

4.1 Descrizione della tolleranza ai guasti di ArangoDB

Data la struttura a ruoli della versione distribuita di ArangoDB, abbiamo deciso di affrontare il problema della tolleranza ai guasti simulando il fallimento dei diversi nodi del database distribuito uno alla volta.

Prima di procedere con la simulazione, è opportuno fornire una breve panoramica della letteratura esistente riguardo alla tolleranza ai guasti di ArangoDB.

ArangoDB è progettato per rispettare le componenti C (Coerenza) e P (Tolleranza ai Partizionamenti) del teorema CAP. Di conseguenza, in caso di partizionamento della rete, il database privilegia la coerenza dei dati rispetto alla disponibilità.

Si afferma inoltre che ArangoDB non presenta un single point of failure, permettendo al sistema di continuare a funzionare senza interruzioni in caso di guasto di un nodo.

Data la struttura a ruoli del database distribuito, è necessario considerare i comportamenti dei diversi nodi in caso di guasto. In particolare, possiamo considerare i seguenti scenari:

- Guasto di un nodo Agency.
- Guasto di un nodo Coordinator.
- Guasto di un nodo DBServer.

In caso di guasto di tutti i nodi Agency, il database distribuito non è in grado di funzionare correttamente, in quanto l'Agency è responsabile della gestione dei nodi Coordinator e DBServer. Per garantire la tolleranza ai guasti, i nodi Agency devono essere replicati in numero dispari e utilizzano il **Raft Consensus Algorithm** per garantire la coerenza dei dati. Poiché Raft si basa sul concetto di quorum, quando fallisce la maggioranza dei nodi dell'Agency, non sarà possibile apportare modifiche alla configurazione del cluster. Tuttavia, le operazioni basate sulla configurazione corrente potranno ancora essere eseguite.

I nodi Coordinator sono responsabili della gestione e dell'ottimizzazione delle query. In caso di guasto di un nodo Coordinator, il database distribuito continua a funzionare correttamente poiché i Coordinator sono stateless e possono essere attivati e disattivati senza influire sul funzionamento generale. Lo spegnimento di tutti i nodi Coordinator rende impossibile comunicare con il database.

Per quanto riguarda i nodi DBServer, dobbiamo distinguere due casi:

- Guasto di un nodo DBServer leader.
- Guasto di un nodo DBServer follower.

In caso di guasto di un nodo DBServer leader, l'Agency rileva il guasto grazie all'assenza di heartbeat dal nodo. Dopo 15 secondi dall'ultimo heartbeat, l'Agency elegge un nuovo leader tra i nodi DBServer che contengono una replica sincronizzata dei dati. Questo processo include una fase di riconfigurazione dell'Agency e del Coordinator. Il processo di selezione del nuovo nodo leader avviene tramite candidatura e votazione. Durante questa fase, i nodi follower possono candidarsi come nuovi leader, mentre i nodi dell'Agency svolgono il ruolo di determinare quale follower abbia i dati più aggiornati. I nodi dell'Agency votano, concedendo il proprio voto al candidato solo se questo possiede i log più recenti. Il nodo follower sarà promosso a leader solo nel caso in cui riceva la maggioranza dei voti (metà più uno dei nodi Agency).

In aggiunta, se nel cluster sono presenti nodi DBServer che non contengono ancora una replica sincronizzata dei dati, viene creato un nuovo follower per garantire la ridondanza dei dati. Altrimenti, si procede con un numero inferiore di follower.

Quando il nodo DBServer leader torna online, possono verificarsi due scenari:

- Se il fattore di replica è stato mantenuto, il nodo DBServer non viene più considerato.
- Altrimenti, il nodo rileva che contiene una replica dei dati e si sincronizza con il nuovo leader, mantenendo così il fattore di replica.

Nel caso di guasto di un nodo DBServer follower, il leader rileva il guasto dopo 3 secondi e dichiara il nodo out of sync. Il sistema si comporta come nel caso di fallimento del leader, ma senza passare per una fase di elezione poiché il leader è già presente.

Quando il nodo DBServer follower torna online, ripete le operazioni previste in caso di fallimento del leader.

4.2 Simulazione

Per simulare un guasto, abbiamo sfruttato l'esecuzione di ArangoDB all'interno di container Docker. Abbiamo utilizzato il comando `docker stop` per arrestare i container dei nodi DBServer e Coordinator uno alla volta. Inoltre, sfruttando le reti di Docker, abbiamo simulato il partizionamento della rete tra i nodi.

Nella nostra configurazione è presente un solo nodo Agency, rendendo impossibile la gestione della tolleranza ai guasti di questo nodo. Pertanto, ci siamo concentrati sui nodi Coordinator e DBServer.

Per comprendere il comportamento del database distribuito in caso di guasto, abbiamo deciso di osservare il sistema in due scenari distinti:

- Prima dell'esecuzione di una serie di operazioni di inserimento, lettura, modifica e cancellazione.
- Durante l'esecuzione di una serie di operazioni di inserimento, lettura, modifica e cancellazione.

Per quanto riguarda i nodi Coordinator, gli esperimenti fatti hanno confermato quanto viene specificato nella documentazione. Simulando il fallimento di un nodo Coordinator, non abbiamo osservato interruzioni della connessione con il cluster. Chiaramente, spegnendo tutti i nodi Coordinator si ha un interruzione del servizio in quanto non è più possibile accedere al cluster.

Per quanto riguarda il fallimento di un nodo DBServer, abbiamo verificato che il database distribuito continuasse a funzionare correttamente, garantendo la ridondanza e la coerenza dei dati.

L'indisponibilità di un nodo DBServer non incide sulle operazioni, a condizione che rimangano attivi un numero sufficiente di server per gestire le richieste.

Per quanto riguarda le operazioni di modifica dei dati (inserimento, modifica e cancellazione) abbiamo osservato comportamenti che mirano a mantenere la consistenza dei dati. Ad esempio, in caso di fallimento di un nodo non è possibile effettuare delle operazioni di cancellazione dei dati. Lo stesso comportamento è stato osservato nelle operazioni di update. Nello specifico, se si spegne un nodo durante l'esecuzione di una modifica, essa non viene terminata e viene fatto il rollback allo stato precedente. Inoltre, abbiamo osservato che ArangoDB gestisce la consistenza dei dati anche in caso di partizionamento della rete tra i nodi. Nella situazione in cui non si riesce a raggiungere la maggioranza dei nodi, il database distribuito non permette di effettuare operazioni.

Per le operazioni di inserimento abbiamo riscontrato delle difficoltà del sistema nel bilanciare i dati tra i vari nodi in caso di fallimento, rendendo necessaria la ridistribuzione attivata manualmente da interfaccia.

In generale, i comportamenti osservati sono conformi alla scelta di ArangoDB di seguire il modello CP (Consistency and Partition Tolerance) secondo il teorema CAP.

Capitolo 5

Conclusioni

In seguito alla nostra ricerca e esperimenti possiamo affermare:

Features	**ArangoDB**
Costruzione Grafo	Semplicità nella costruzione. Flessibilità nei dati dei nodi (JSON) Solo archi orientati
Linguaggio (AQL)	Linguaggio dichiarativo simile a SQL Non supporta operazioni di definizione dei dati (DDL)
Scalabilità	Offre la possibilità di scalare orizzontalmente ma non sempre risulta efficiente CAP Theorem (CP)
Sharding	Abbastanza omogeneo ad eccezione di alcuni casi Versione efficiente disponibile solo nella versione a pagamento (Non testata)
Architettura distribuita	Peer-to-Peer con 3 ruoli Agency: come sono distribuiti i dati Coordinator: entrypoint della rete DB Server: memorizza i dati
Tolleranza ai guasti	DBServer: Finché riesce a effettuare repliche consente tutte le operazioni. Altrimenti solo lettura Coordinator: ne è richiesta la presenza di almeno 1 Agency: ne è richiesta la presenza di almeno 1
Replica dei dati	Replica con leader-follower Numero di repliche personalizzabile
Locking dei dati	Automatico per modifica dei dati Su richiesta per lettura
Risoluzione deadlock	Rileva il deadlock e lo risolve cancellando una transazione scelta casualmente