UNIVERSITÀ DEGLI STUDI DI MESSINA

DIPARTIMENTO DI SCIENZE MATEMATICHE E INFORMATICHE,
SCIENZE FISICHE E SCIENZE DELLA TERRA

Corso di Laurea Triennale in Informatica

Rilevamento delle frodi e-commerce. MongoDB vs Neo4j

Studente: Emanuele Russo,
MAT [526889]

ANNO ACCADEMICO 2022/2023

1 Sommario

L'obiettivo del progetto consiste nell'effettuare un confronto delle prestazioni tra due sistemi di gestione di database NoSQL: MongoDB, che è orientato ai documenti, e Neo4j, specializzato in database a grafo. Per raggiungere questo obiettivo, è stato selezionato uno specifico scenario di utilizzo, per il quale è stato creato un modello di dati per popolare entrambi i DBMS, tenendo conto delle loro peculiarità. Successivamente, sono state eseguite quattro query con crescente complessità computazionale al fine di valutare le prestazioni dei due sistemi, misurando i tempi di esecuzione delle query.

Tutte le operazioni, incluse la generazione e l'inserimento dei dati, sono state automatizzate mediante l'uso di script in Python. Anche le interrogazioni ai database sono state automatizzate. I tempi di esecuzione delle query sono stati registrati in un file CSV e successivamente analizzati utilizzando grafici appropriati.

Le conclusioni del progetto hanno evidenziato che ciascun DBMS ha dimostrato ottimali prestazioni in situazioni specifiche. Neo4j si è rivelato particolarmente efficiente nelle query che coinvolgono più entità e relazioni, mentre MongoDB ha mostrato una maggiore reattività quando si trattava di interrogazioni relative a singole entità.

2 Introduzione

I database NoSQL (Not Only SQL) sono sistemi di gestione dati progettati per affrontare il compito di memorizzare e manipolare grandi volumi di informazioni, offrendo soluzioni altamente scalabili e adattabili. Questi database consentono di lavorare con dati che non seguono uno schema rigido, a differenza dei tradizionali database relazionali (DBMS relazionali). Nel tempo, questa evoluzione ha portato alla creazione di diverse categorie di database per gestire i Big Data, i quali variano in termini di volume e complessità dei dati generati. Queste categorie includono database di tipo chiave-valore, orientati alle colonne, orientati ai documenti e database a grafo, ciascuno con le proprie caratteristiche e applicazioni specifiche.

È importante notare che i DBMS NoSQL rispettano il Teorema CAP, noto anche come Teorema di Brewer, il quale afferma che in un database distribuito non è possibile garantire simultaneamente le seguenti tre proprietà:

1. Consistenza (Consistency)
2. Disponibilità (Availability)
3. Tolleranza al partizionamento (Partitioning)

In base a questa considerazione, i due database selezionati per la comparazione sono MongoDB, con priorità alla consistenza e la tolleranza al partizionamento (CP), e Neo4j, che si concentra sulla coerenza e la disponibilità (CA).

3 MongoDB

MongoDB è un tipo di database NoSQL orientato ai documenti, il che significa che archivia le informazioni in documenti che sono raggruppati all'interno di collezioni. Una delle sue caratteristiche principali è la flessibilità dello schema, ciò significa che non è necessario seguire uno schema dati rigido o semirigido. MongoDB offre query semplici ed efficienti indicizzazioni, sharding e replica per bilanciare la velocità e l'affidabilità dei dati.

I documenti in MongoDB sono essenzialmente dati memorizzati come coppie chiave-valore e consentono un accesso diretto ai dati. Questi documenti sono rappresentati nel formato BSON (Binary Serialized dOcument Notation), una rappresentazione binaria derivata dal formato JSON (JavaScript Object Notation) in formato testuale. MongoDB consente anche la distribuzione fisica dei dati su nodi separati, noto come sharding, che suddivide i dati su più nodi, consentendo l'elaborazione parallela e prestazioni elevate.

Nel contesto specifico di questo studio, MongoDB è stato utilizzato tramite l’applicazione MongoDBCompass, il software standalone fornito da MongoDB per gestire e interrogare i database, e sono state utilizzate API per accedere al database tramite il linguaggio di programmazione Python.

4 Neo4j

Neo4j è un sistema di gestione di database NoSQL progettato specificamente per gestire dati complessi in modo efficiente attraverso la struttura a grafo. Questo database si basa su due elementi fondamentali: i nodi, che rappresentano porzioni di dati, e le relazioni, che sono archi orientati con proprietà. Questa struttura consente di sfruttare al meglio le caratteristiche dei grafi, come attraversare e interrogare grafi completi o sottografi, con accesso diretto alle proprietà dei nodi e delle relazioni attraverso coppie chiave-valore.

Un aspetto importante è che i database a grafo, compreso Neo4j, sono privi di uno schema rigido. Questo significa che i nodi possono avere proprietà diverse anche all'interno della stessa categoria, rendendoli estremamente versatili e adatti per gestire relazioni complesse tra diverse entità.

Per quanto riguarda l'interfacciamento con Neo4j nel contesto di questo studio, è stato utilizzato l'applicativo Neo4j Desktop. Questo strumento fornisce un'interfaccia grafica per la gestione dei database basati su Neo4j, consentendo di creare, avviare, arrestare ed eliminare database, visualizzare i dati in modo interattivo, scrivere ed eseguire query utilizzando il linguaggio di query specifico per i database a grafo chiamato Cypher. Come nel caso di MongoDB, sono state utilizzate anche le API di Neo4j per consentire l'accesso al database tramite il linguaggio di programmazione Python.

5 Caso di studio

La tematica affrontata in questo caso di studio è il rilevamento delle frodi nell'e-commerce. Le frodi nell'e-commerce rappresentano una sfida significativa per le aziende online, comportando perdite di miliardi di dollari ogni anno, secondo le stime. Per affrontare questa sfida, è cruciale che le piattaforme di e-commerce dispongano di strumenti efficaci per individuare e prevenire le attività fraudolente perpetrate da organizzazioni criminali o individui malintenzionati. Tuttavia, l'individuazione delle frodi nell'e-commerce rimane una sfida complessa.

Spesso, le frodi nell'e-commerce coinvolgono schemi sofisticati, come l'uso di identità false, l'acquisto di prodotti con carte di credito rubate o l'uso di molteplici account per compiere attività fraudolente. Queste attività di frode di solito non sono opera di singoli individui, ma sono orchestrate da gruppi criminali organizzati o reti complesse, rendendo difficile la loro individuazione per le piattaforme di e-commerce. Uno dei trucchi più comuni utilizzati dai truffatori è la creazione di identità false e l'uso di molteplici account per acquistare prodotti in modo fraudolento.

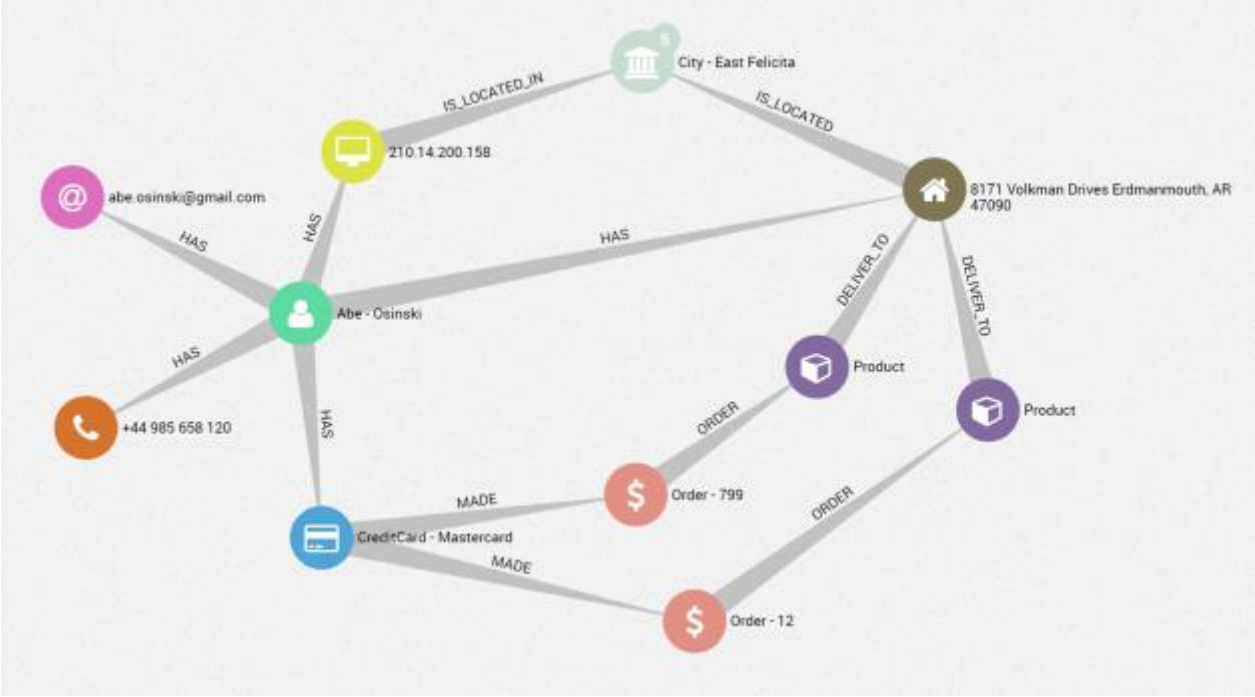
Le piattaforme di e-commerce spesso gestiscono enormi quantità di dati relativi ai propri clienti, transazioni e acquisti di prodotti. Per contrastare queste frodi, è essenziale disporre di strumenti di analisi avanzati che possano rivelare schemi sospetti all'interno di questi dati. Tuttavia, i tradizionali sistemi di gestione dei dati, come i database relazionali, possono essere limitati nell'analisi di grandi volumi di dati connessi e possono non essere sufficientemente reattivi per rilevare frodi in tempo reale.

Per affrontare questo studio, è possibile adottare un approccio basato sui database NoSQL, come Neo4j e MongoDB. Questi database consentono di rappresentare i dati come grafi, in cui ogni informazione (come l'identità del cliente, l'indirizzo, gli acquisti di prodotti) è un nodo connesso ad altri nodi tramite archi che conservano informazioni sulle relazioni.

Questo approccio consente di individuare connessioni sospette all'interno dei dati, ad esempio, se due clienti condividono lo stesso indirizzo e-mail o se ci sono molteplici acquisti di prodotti associati a molteplici account. L'obiettivo principale di utilizzare questi database NoSQL è tracciare tutte le connessioni a partire da un sospetto, aiutando così a individuare e prevenire frodi nell'e-commerce.

I dati utilizzati per questo studio sono stati generati casualmente e salvati in file CSV per consentirne un facile caricamento nei database Neo4j e MongoDB.

6 Datamodel



È possibile comprendere la presenza di tre entità che vengono utilizzate per realizzare i *datamodel* per i due DBMS utilizzati:

1. Utenti, caratterizzati da un nome, un indirizzo, un numero di telefono e una e-mail.
2. Prodotti, caratterizzati da un nome che ne identifica la tipologia ed un prezzo.
3. Transazioni, alle quali sono collegati gli utenti e i prodotti tenendo così traccia di colui che effettua la transazione e del prodotto acquistato. L'entità transazioni, inoltre, riporta la data della transazione e il metodo di pagamento utilizzato.

Bisogna comunque specificare che il DBMS MongoDB non può definire esplicitamente le relazioni, a differenza di Neo4j in cui le relazioni possono essere rappresentate come archi del grafo.

7 Generazione

Per ottenere i dati destinati ai database, è stato creato uno script in linguaggio Python. Questo script è stato sviluppato con l'obiettivo di generare dati da inserire nei file CSV e, successivamente, alimentare sia il database MongoDB che quello Neo4j. Per generare questi dati, lo script fa uso del modulo Python chiamato "Faker". Questo modulo permette di generare dati casuali, ma realistici, per varie entità coinvolte nel progetto, tra cui: nomi, e-mail, indirizzi, numeri di telefono, prezzi dei prodotti, metodi di pagamento e date delle transazioni.

Lo script fa uso della variabile denominata "total_users" per memorizzare il numero di utenti da generare. Da tale variabile nascono poi il numero di elementi delle altre entità, infatti, per quanto riguarda i prodotti, ne vengono generati tre per ogni utente, in tal modo si hanno molteplici prodotti diversi che simulano l'ingente numero di prodotti presenti in un e-commerce. Per quanto riguarda, invece, il numero di elementi relativo all'entità transazioni, ne vengono generati fino a quattro per ogni utente, in tal modo si simula che uno stesso utente possa effettuare fino a quattro transazioni in quell'e-commerce.

La quantità di dati generati è relativa a una percentuale specifica del numero massimo di elementi di ciascuna entità. Nello specifico, la quantità di dati generati può corrispondere al 25%, al 50%, al 75% o al 100% del numero massimo stabilito. La percentuale esatta è determinata dai valori nella lista "percentages" presente nello script e che verrà utilizzato nel ciclo for con lo scopo di generare i dataset di diverso contenuto informativo.

I dati generati vengono successivamente salvati in file CSV separati per ciascuna delle entità coinvolte, contenendo così i dati generati in accordo alle specifiche richieste del progetto. Di seguito la tabella dei dataset:

Entità	**25%**	**50%**	**75%**	**100%**
Utenti	2500	5000	7500	10000
Prodotti	7500	1500	22500	30000
Transazioni	9684	19376	29091	38787

8 Inserimento

8.1 MongoDB

Tenendo conto che in MongoDB, le entità vengono organizzate all'interno di collezioni di documenti i quali, a loro volta si trovano all'interno di un database. È stato, quindi, impostato un db denominato "frodi_ecommerce" che contiene al suo interno dodici collezioni poiché per ogni entità si ha una collezione relativa alla percentuale del contenuto informativo.

Nello script python di inserimento dei file CSV precedentemente creati all'interno del database MongoDB vengono esplicitate diverse condizioni in modo tale che ogni prodotto presenti un proprio codice identificativo che viene riportato nell'entità transazioni. La stessa cosa avviene con ogni utente. In questo modo si cerca di ovviare al problema della mancanza di relazioni esplicite in MongoDB.
Lo script è così definito:

```python
import os
import pandas as pd
from pymongo import MongoClient

# Cartella in cui si trova lo script Python
script_directory = os.path.dirname(os.path.abspath(__file__))

# Connessione al database MongoDB
client = MongoClient('mongodb://localhost:27017/')
db = client['frodi_ecommerce']

# Percentuali dei dataset rispetto al 100%
percentages = [100, 75, 50, 25]

# Tipi di dati
data_types = ['transazioni', 'utenti', 'prodotti']

for data_type in data_types:
    for percentage in percentages:
        collection_name = f'{data_type}{percentage}%'
        csv_filename = f'{data_type}_{percentage}%.csv'

        # Percorso completo al file CSV
        csv_path = os.path.join(script_directory, f'{percentage}%', csv_filename)

        # Leggi i dati dal file CSV utilizzando pandas
        data = pd.read_csv(csv_path, encoding='ISO-8859-1')

        # Converti i dati in formato JSON
        data_json = data.to_dict(orient='records')

        # Inserisci i dati nella collezione del database
        collection = db[collection_name]
        collection.insert_many(data_json)
```

8.2 Neo4j

In Neo4j, le entità sono rappresentate come "labels" (etichette). Le relazioni, a differenza di MongoDB, sono chiaramente definite come archi all'interno del grafo. Per gestire i database Neo4j e inserire i file CSV generati, si è scelto di utilizzare Neo4j Desktop. Questa applicazione fornisce agli utenti un'interfaccia user-friendly per la creazione dei database di interesse.

All'interno dell'applicazione desktop, sono stati creati quattro database Neo4j, ciascuno corrispondente a una diversa dimensione del dataset considerato (25%, 50%, 75%, 100%). Per ciascun database, è necessario caricare i file CSV corrispondenti nella cartella di importazione del database.

Inoltre, è stato sviluppato uno script Python per l'inserimento dei dati dai file CSV all'interno del database Neo4j. Prima di eseguire questo script, è fondamentale avviare il database Neo4j e, successivamente, eseguire lo script stesso.

Le relazioni in questo caso sono esplicitate tramite la funzione dell'API di python. Tale funzione è denominata "Relationship" e viene utilizzata per creare le relazioni che riguardano l'entità transazioni con le altre due entità utenti e prodotti. Lo script è così definito:

```
# Ottieni il grafo corrispondente alla percentuale
graph = graphs_by_percentage[percentage]

# Inserisci i dati nel grafo
for index, row in data.iterrows():
    node = Node(data_type, **row.to_dict())
    #print(node)
    graph.create(node)

    if data_type == 'transazioni':
        # Crea relazione con utente
        user_id = row['user_id']
        user_node = graph.nodes.match('utenti', user_id=user_id).first()
        if user_node:
            transaction_to_user = Relationship(user_node, 'EFFETTUATO_DA', node)
            graph.create(transaction_to_user)
            #print("relazione in corso trans-ute")

        # Crea relazione con prodotto
        product_id = row['product_id']
        #print(product_id)
        product_node = graph.nodes.match('prodotti', product_id=product_id).first()
        if product_node:
            #print("sono qui")
            transaction_to_product = Relationship(node, 'CONTIENE', product_node)
            #print(transaction_to_product)
            graph.create(transaction_to_product)
            #print("relazione in corso trans-prod")
```

9 Query

Sono state condotte quattro diverse query su entrambi i database al fine di valutare come si comportano MongoDB e Neo4j in risposta a crescenti livelli di complessità delle query e dimensioni dei dataset. L'obiettivo era identificare i punti di forza e le debolezze di entrambi i sistemi. L'esecuzione delle query su entrambi i database è stata automatizzata attraverso script Python.

Dopo la prima esecuzione, effettuata in un ambiente appena configurato, sono state eseguite ulteriori 50 misurazioni per confrontare i meccanismi di caching dei due DBMS, che possono accelerare le esecuzioni successive delle stesse query. I tempi di esecuzione sono stati registrati e salvati in un file CSV con il seguente formato:

- Query.
- Percentuale contenuto formativo.
- Tempo della prima esecuzione.
- Media dei tempi delle 50 esecuzioni.

Successivamente, i dati ottenuti sono stati utilizzati in un nuovo script python per la creazione dei relativi istogrammi. Il primo istogramma mostra i tempi della prima esecuzione della query, mentre il secondo mostra la media dei tempi delle 50 esecuzioni successive, inclusi anche gli intervalli di confidenza al 95%. Ogni grafico visualizza i dati relativi alla query corrente per i quattro dataset presi in considerazione.

Per eseguire le query sui due database, sono stati creati due script distinti: "query_Neo4j" e "query_MongoDB". Quest'ultimo si occupa dell'esecuzione delle query su MongoDB. Inoltre, è stato utilizzato il metodo "aggregate()", che permette di eseguire complesse operazioni di aggregazione sui dati. Questo metodo consente di specificare la pipeline di aggregazione come una lista di dizionari, dove ogni oggetto rappresenta una fase dell'aggregazione. L'output finale dell'aggregazione è un insieme di documenti che rappresentano i risultati calcolati in base alla pipeline specificata.
Il pezzo di codice per l'esecuzione delle query è il seguente:

```
for _ in range(NUM_EXPERIMENTS - 1):
    if query_func == search_product_in_store:
        execution_time, _ = run_experiment(query_func, products_collection_name, PRODUCT_NAME)
    elif query_func == find_transactions_for_product:
        execution_time, _ = run_experiment(query_func, transactions_collection_name, product_id)
    elif query_func == find_payment_usage_and_product_transactions:
        execution_time, _ = run_experiment(query_func, transactions_collection_name, PAYMENT_METHOD_BITCOIN, products_collection_name, PRODUCT_NAME)
    elif query_func == find_most_frequent_bitcoin_user_and_product_transactions:
        execution_time, _ = run_experiment(query_func, users_collection_name, transactions_collection_name, products_collection_name, PAYMENT_METHOD
    execution_times.append(execution_time)
```

Esso sfrutta la seguente funzione per l'esecuzione delle query:

```
# Funzione per eseguire un singolo esperimento
def run_experiment(query_func, *args):
    start_time = time.time()
    result = query_func(*args)
    end_time = time.time()
    execution_time = (end_time - start_time) * 1000  # Converti in millisecondi
    return execution_time, result
```

In alternativa, lo script "query_Neo4j" gestisce l'esecuzione delle query su Neo4j utilizzando il metodo "run()", che richiede una stringa contenente l'interrogazione da eseguire.
Il pezzo di codice per l'esecuzione delle query è il seguente:

```
for query_idx, query in enumerate(queries):
    print(f"Query {query_idx + 1}")

    execution_times_query = []

    for _ in range(51):
        start_time = time.time()

        if query_idx == 0:
            result = graph.run(query).data()
            product_id = result[0]['product_id']
        else:
            result = graph.run(query, product_id=product_id).data()

        end_time = time.time()
        execution_time = (end_time - start_time) * 1000
        execution_times_query.append(execution_time)

        print(f"Risultati query {query_idx + 1}:\n{result}")
```

Entrambi gli script Python registrano i tempi di esecuzione delle query su un file CSV dedicato.

10 Query n°1

La prima query cerca tra tutti i prodotti uno specifico, una “macchina fotografica”.

Per MongoDB:

```
result = db[products_collection_name].find_one({'name': product_name})
```

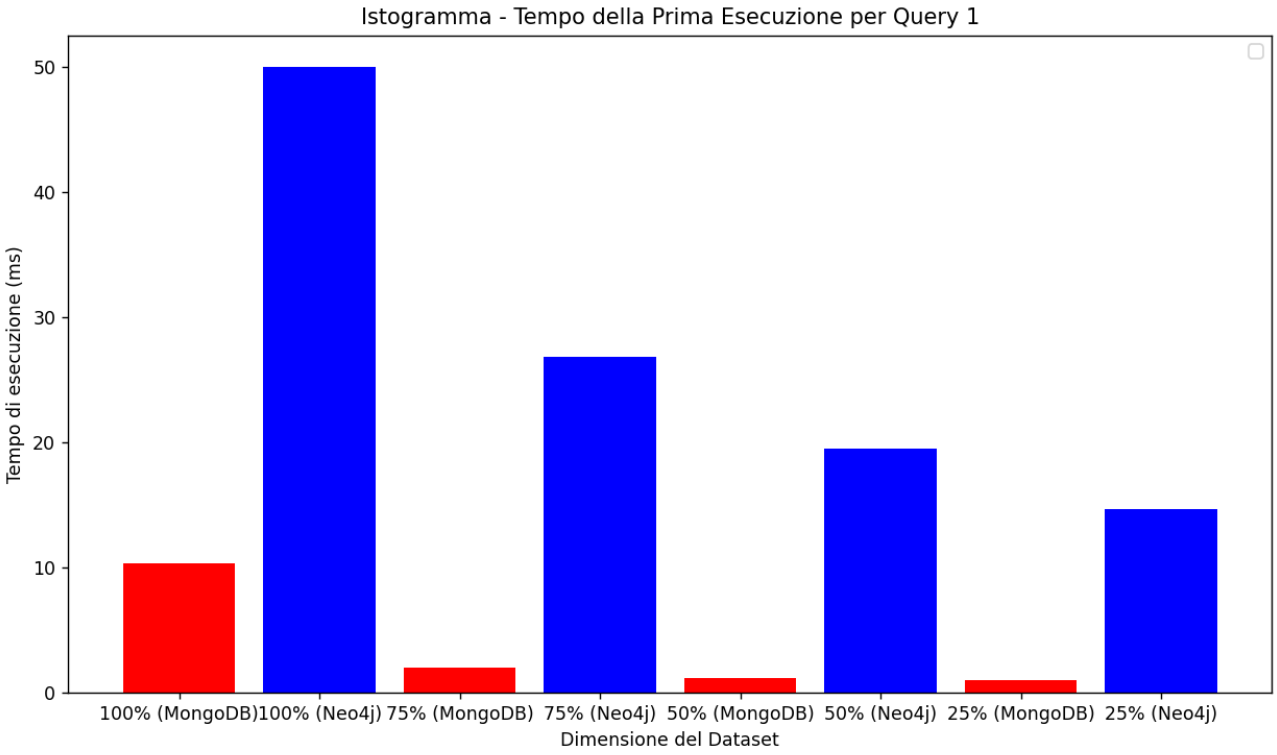
Per Neo4j:

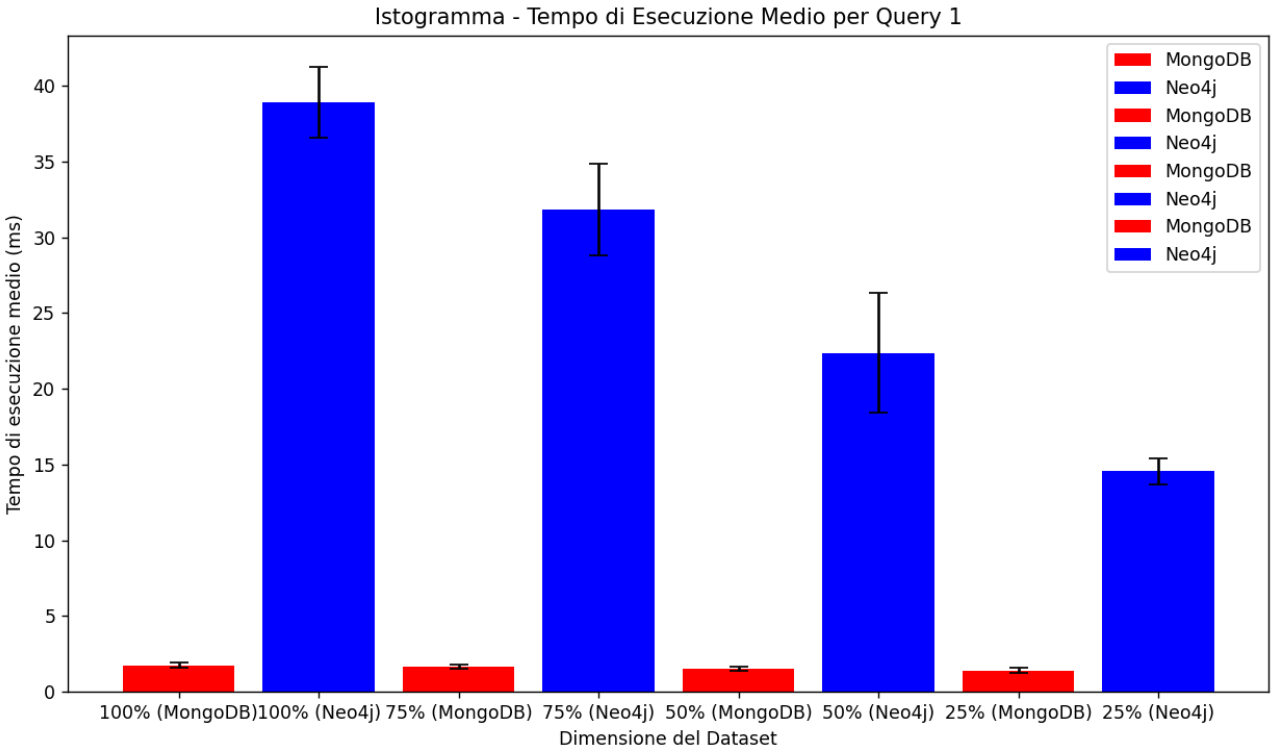
```
MATCH (p:prodotti {name: 'macchina fotografica'}) RETURN p, p.product_id as product_id
```

I tempi registrati sono i seguenti:

Primi Tempi	MongoDB	Neo4j
100%	10.390520095825195	50.05669593811035
75%	2.0151138305664062	26.847362518310547
50%	1.1408329010009766	19.501209259033203
25%	1.046895980834961	14.657735824584961

Tempi Medi	MongoDB	Neo4j
100%	1.761178970336914	38.894572257995605
75%	1.6464614868164062	31.798229217529297
50%	1.50665283203125	22.343149185180664
25%	1.3965034484863281	14.549393653869629


Istogramma - Tempo della Prima Esecuzione per Query 1
Tempo di esecuzione (ms)
0
10
20
30
40
50
100% (MongoDB)
100% (Neo4j)
75% (MongoDB)
75% (Neo4j)
50% (MongoDB)
50% (Neo4j)
25% (MongoDB)
25% (Neo4j)
Dimensione del Dataset


Istogramma - Tempo di Esecuzione Medio per Query 1
Tempo di esecuzione medio (ms)
0
5
10
15
20
25
30
35
40
MongoDB
Neo4j
MongoDB
Neo4j
MongoDB
Neo4j
MongoDB
Neo4j
100% (MongoDB)
100% (Neo4j)
75% (MongoDB)
75% (Neo4j)
50% (MongoDB)
50% (Neo4j)
25% (MongoDB)
25% (Neo4j)
Dimensione del Dataset

11 Query n°2

La seconda query, oltre a cercare il prodotto specifico "macchina fotografica", cerca il numero di transazioni relative a quel prodotto.

Per MongoDB:

```
transaction_count = db[transactions_collection_name].count_documents({'product_id': product_id})
```

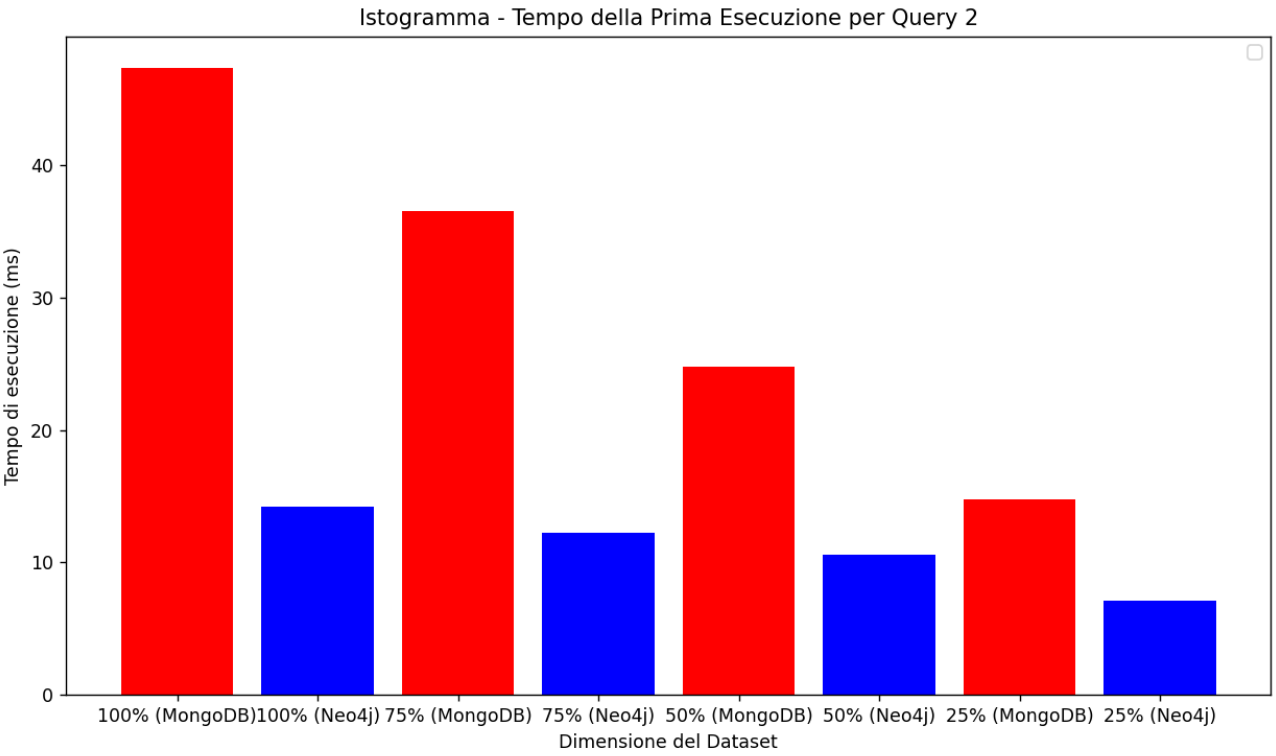
Per Neo4j:

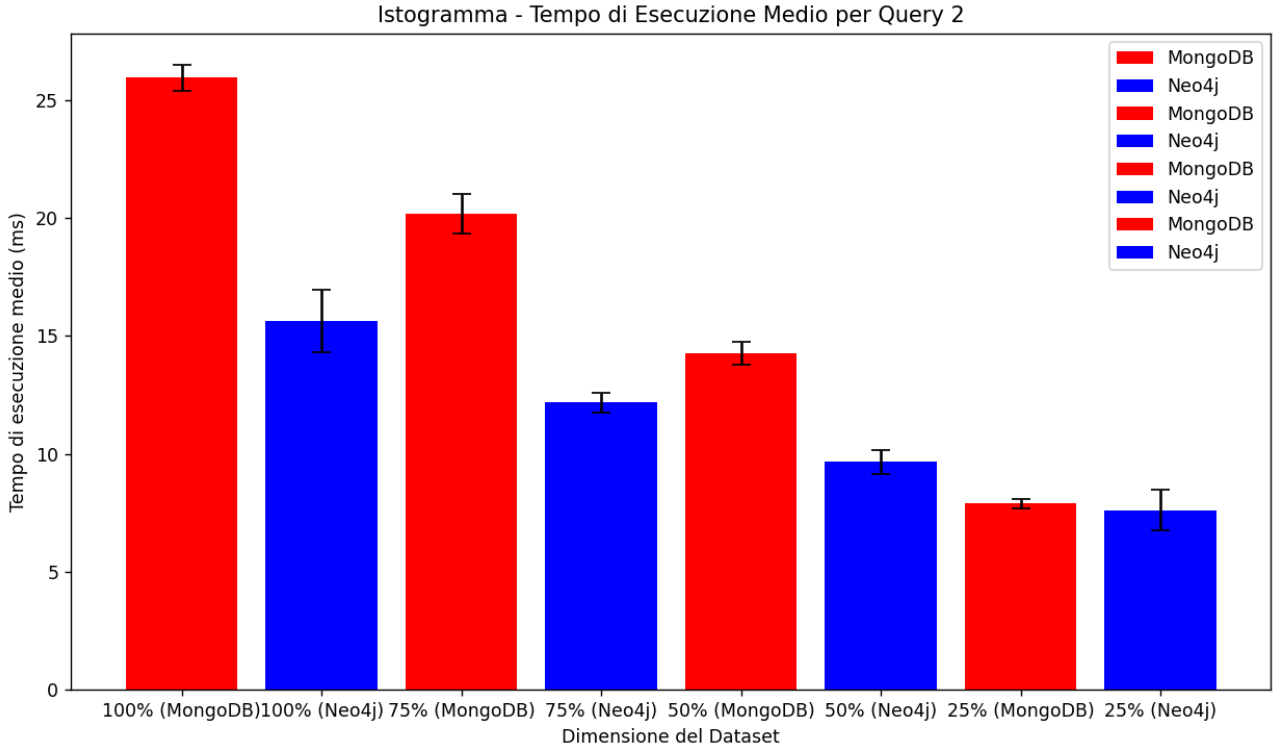
```
MATCH (p:prodotti) WHERE p.name = 'macchina fotografica' RETURN count(p) as camera_product_count
```

I tempi registrati sono i seguenti:

Primi Tempi	MongoDB	Neo4j
100%	47.38783836364746	14.224529266357422
75%	36.589860916137695	12.228250503540039
50%	24.824142456054688	10.569095611572266
25%	14.769792556762695	7.102727890014648

Tempi Medi	MongoDB	Neo4j
100%	25.953068733215332	15.649375915527344
75%	20.173163414001465	12.17801570892334
50%	14.258646965026855	9.665794372558594
25%	7.906398773193359	7.61838436126709


Istogramma - Tempo della Prima Esecuzione per Query 2
Tempo di esecuzione (ms)
0
10
20
30
40
100% (MongoDB)
100% (Neo4j)
75% (MongoDB)
75% (Neo4j)
50% (MongoDB)
50% (Neo4j)
25% (MongoDB)
25% (Neo4j)
Dimensione del Dataset


Istogramma - Tempo di Esecuzione Medio per Query 2
Tempo di esecuzione medio (ms)
0
5
10
15
20
25
MongoDB
Neo4j
MongoDB
Neo4j
MongoDB
Neo4j
MongoDB
Neo4j
100% (MongoDB)
100% (Neo4j)
75% (MongoDB)
75% (Neo4j)
50% (MongoDB)
50% (Neo4j)
25% (MongoDB)
25% (Neo4j)
Dimensione del Dataset

12 Query n°3

La terza query effettuata calcola il numero di volte in cui è stato utilizzato il metodo di pagamento "bitcoin" e l'importo speso usando quel metodo. Successivamente cerca un prodotto specifico "macchina fotografica" e il numero di transazioni relative a quel prodotto.

Per MongoDB:

```
pipeline = [
    {
        '$match': {
            'payment_method': payment_method
        }
    },
    {
        '$group': {
            '_id': None,
            'bitcoin_transactions': {'$sum': 1},
            'total_spent_with_bitcoin': {'$sum': '$amount'}
        }
    }
]
result = db[transactions_collection_name].aggregate(pipeline)
```

Unita alla chiamata alla query precedente:

```
find_transactions_for_product(transactions_collection_name, product_id)
```

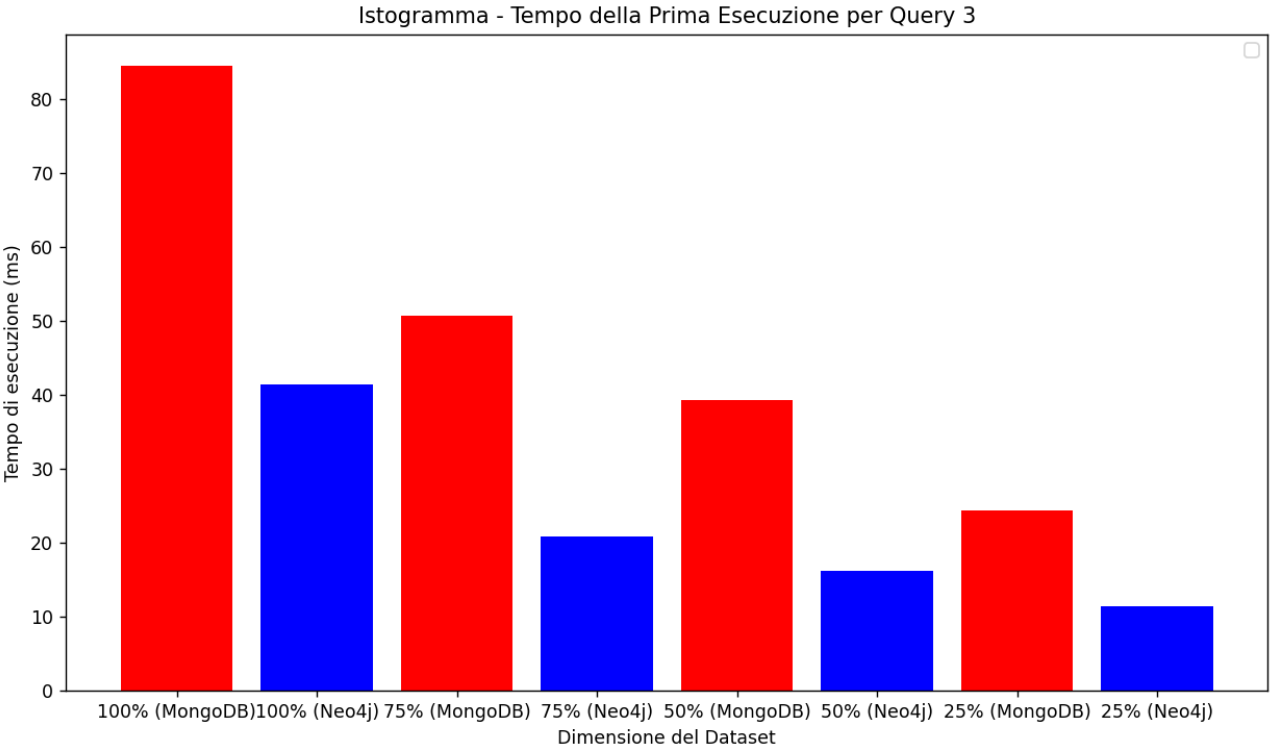
Per Neo4j:

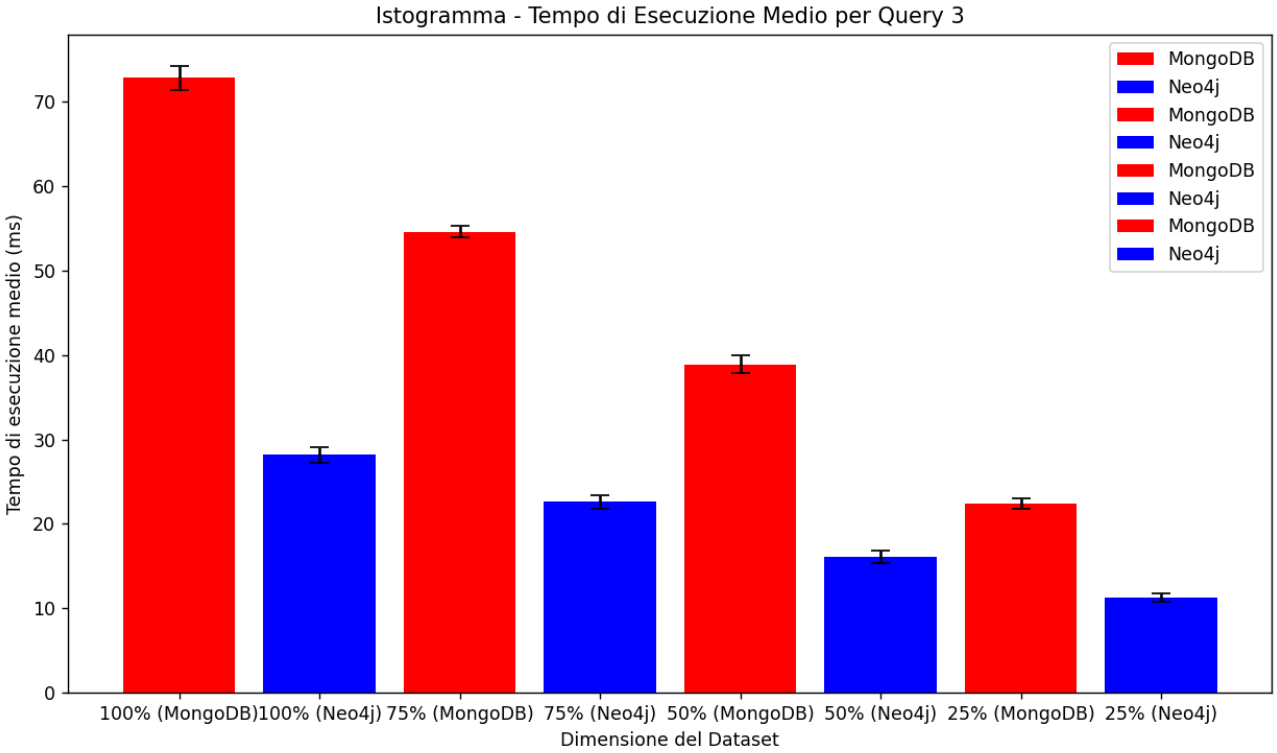
```
MATCH (t:transazioni)
WHERE t.payment_method = 'bitcoin'
WITH count(t) as bitcoin_transactions, sum(t.amount) as total_spent_with_bitcoin
MATCH (p:prodotti) WHERE p.name = 'macchina fotografica'
RETURN bitcoin_transactions, total_spent_with_bitcoin, count(p) as camera_product_count
```

I tempi registrati sono:

Primi Tempi	MongoDB	Neo4j
100%	84.5487117767334	41.38755798339844
75%	50.707101821899414	20.828723907470703
50%	39.371490478515625	16.17145538330078
25%	24.324655532836914	11.396646499633789

Tempi Medi	MongoDB	Neo4j
100%	72.82143115997314	28.206825256347656
75%	54.60650444030762	22.59212017059326
50%	38.87239456176758	16.079745292663574
25%	22.399773597717285	11.249899864196777


Istogramma - Tempo della Prima Esecuzione per Query 3
Tempo di esecuzione (ms)
0
10
20
30
40
50
60
70
80
100% (MongoDB)
100% (Neo4j)
75% (MongoDB)
75% (Neo4j)
50% (MongoDB)
50% (Neo4j)
25% (MongoDB)
25% (Neo4j)
Dimensione del Dataset


Istogramma - Tempo di Esecuzione Medio per Query 3
Tempo di esecuzione medio (ms)
0
10
20
30
40
50
60
70
MongoDB
Neo4j
MongoDB
Neo4j
MongoDB
Neo4j
MongoDB
Neo4j
100% (MongoDB)
100% (Neo4j)
75% (MongoDB)
75% (Neo4j)
50% (MongoDB)
50% (Neo4j)
25% (MongoDB)
25% (Neo4j)
Dimensione del Dataset

13 Query n°4

La quarta query effettuata trova il nome dell'utente che ha usato più volte il metodo di pagamento "bitcoin", l'importo speso da questo utente utilizzando questo metodo di pagamento. Successivamente cerca un prodotto specifico "macchina fotografica" e il numero di transazioni relative a quel prodotto.

Per MongoDB:

```
pipeline = [
    {
        '$match': {
            'payment_method': payment_method,
        }
    },
    {
        '$group': {
            '_id': '$user_id',
            'transaction_count': {'$sum': 1},
            'total_spent': {'$sum': '$amount'}
        }
    },
    {
        '$sort': {'transaction_count': -1}
    },
    {
        '$limit': 1
    }
]
result = db[transactions_collection_name].aggregate(pipeline)
```

Unita alla chiamata alla query precedente:

```
find_transactions_for_product(transactions_collection_name, product_id)
```

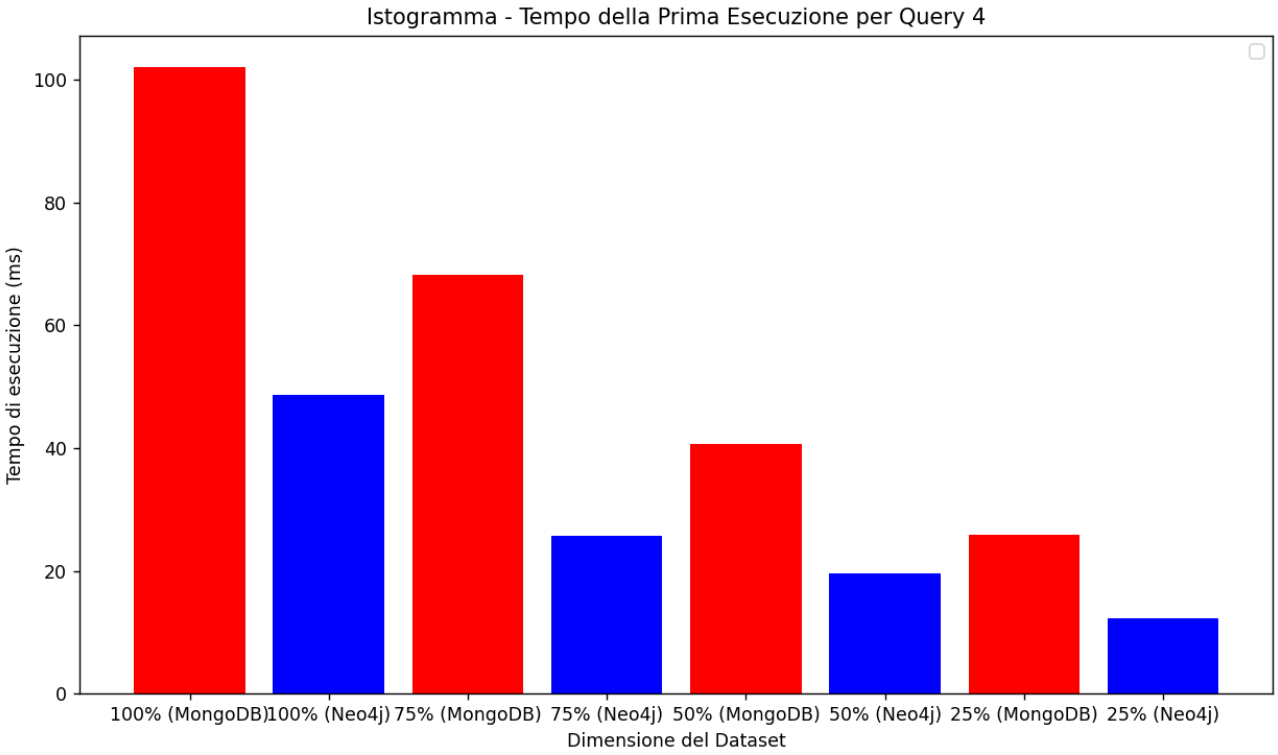
Per Neo4j:

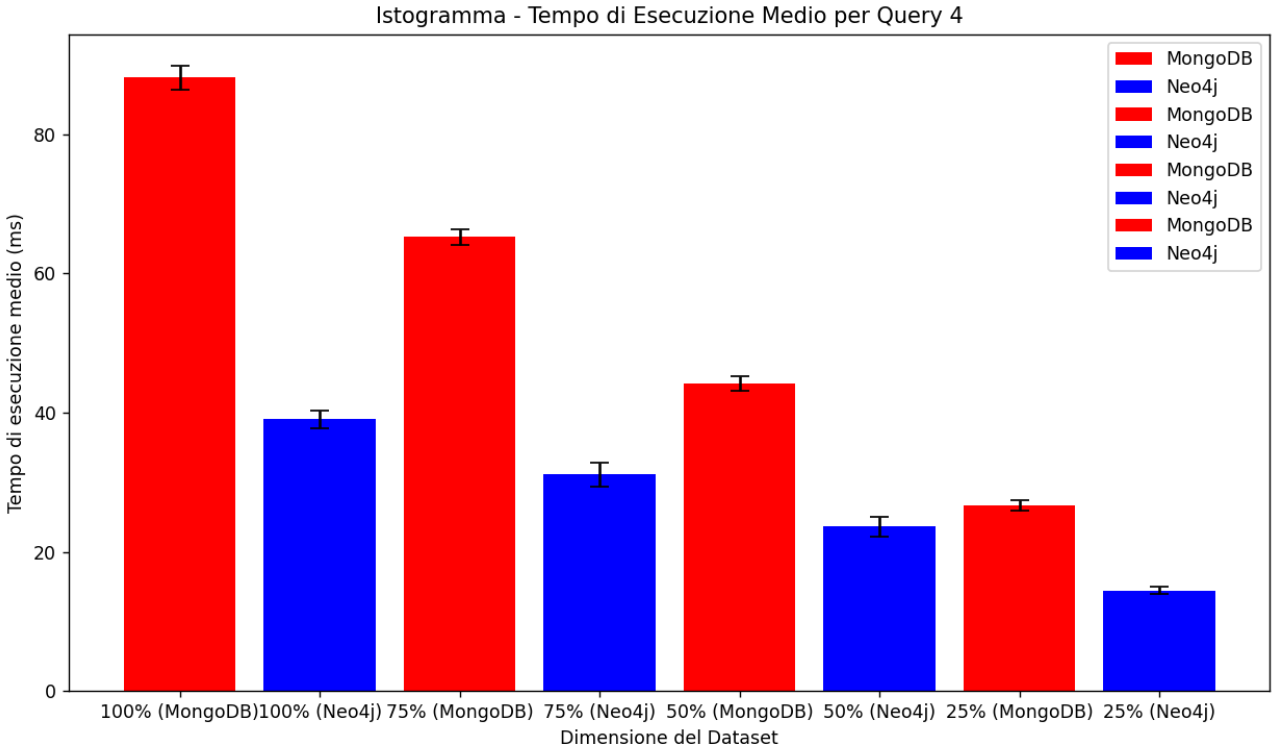
```
MATCH (u:utenti)-[:EFFETTUATO_DA]->(t:transazioni {payment_method: 'bitcoin'})
WITH u, count(t) as transaction_count, sum(t.amount) as total_spent
ORDER BY transaction_count DESC
LIMIT 1
OPTIONAL MATCH (p:prodotti {name: 'macchina fotografica'})<-[r:CONTIENE]-(t2:transazioni)
RETURN u.name as user_name, transaction_count, total_spent, count(t2) as camera_transactions
```

I tempi registrati sono:

Primi Tempi	MongoDB	Neo4j
100%	102.10895538330078	48.73204231262207
75%	68.28427314758301	25.67458152770996
50%	40.714263916015625	19.611835479736328
25%	25.925874710083008	12.227773666381836

Tempi Medi	MongoDB	Neo4j
100%	88.17772388458252	39.04977798461914
75%	65.2395486831665	31.114249229431152
50%	44.180307388305664	23.62048625946045
25%	26.63543701171875	14.449453353881836


Istogramma - Tempo della Prima Esecuzione per Query 4
Tempo di esecuzione (ms)
0
20
40
60
80
100
100% (MongoDB)
100% (Neo4j)
75% (MongoDB)
75% (Neo4j)
50% (MongoDB)
50% (Neo4j)
25% (MongoDB)
25% (Neo4j)
Dimensione del Dataset


Istogramma - Tempo di Esecuzione Medio per Query 4
Tempo di esecuzione medio (ms)
0
20
40
60
80
MongoDB
Neo4j
MongoDB
Neo4j
MongoDB
Neo4j
MongoDB
Neo4j
100% (MongoDB)
100% (Neo4j)
75% (MongoDB)
75% (Neo4j)
50% (MongoDB)
50% (Neo4j)
25% (MongoDB)
25% (Neo4j)
Dimensione del Dataset

14 Osservazioni e conclusioni

È possibile notare la differenza nei tempi di esecuzione delle query quando vengono eseguite inizialmente, senza alcun beneficio dalla memorizzazione nella cache dei dati, e quando vengono eseguite successivamente con dati in cache. Dai risultati delle query emergono le forze distintive dei due DBMS. Risulta evidente che MongoDB è significativamente più efficiente quando si tratta di operazioni su singole entità. Tuttavia, a partire dalla terza query, Neo4j mostra una costante superiorità all'aumentare della complessità delle query. Questo indica che Neo4j si comporta meglio quando si hanno corrispondenze tra diverse entità e la complessità delle operazioni aumenta.
Osservazioni specifiche sui risultati ottenuti:

- Nella prima query, entrambi i database hanno mostrato tempi di esecuzione relativamente bassi, ma MongoDB ha dimostrato tempi inferiori, evidenziando una maggiore efficienza nelle operazioni su singole entità.

- Nella seconda query, Neo4j inizia a mostrare tempi di esecuzione inferiori rispetto a MongoDB.

- Dalla terza alla quarta query, i tempi di esecuzione di MongoDB hanno registrato un notevole aumento in caso di query complesse. Ciò suggerisce che Neo4j eccelle nella gestione di query complesse con molte relazioni.

In sintesi, i risultati indicano che MongoDB è una scelta appropriata quando si affrontano query semplici su grandi volumi di dati, con meno relazioni tra le entità. D'altra parte, Neo4j, essendo un database basato su grafo, si dimostra più efficiente quando si affrontano query complesse e si richiede una rapida ricerca di dati con numerose relazioni interconnesse.