UniMe
1548

Corso di Laurea Triennale in INFORMATICA

Progetto di Basi di Dati Mod.B

Realizzato da:

Gabriele Agosta

Vitaliy Lyaskovskiy

Indice

Problematica Affrontata

“Rilevamento delle frodi nell'e-commerce”

Le frodi nell'e-commerce sono un fenomeno in rapida crescita alimentato dall'espansione costante e inarrestabile dello shopping online. L'anonimato fornito dal web, l'accesso globale e la natura senza limiti del commercio elettronico creano un terreno fertile per le attività fraudolente. Da frodi con carta di credito a operazioni di phishing, gli attacchi fraudolenti possono assumere molteplici forme ed il miglioramento della loro sofisticazione ne rendono difficile la prevenzione.

In questo contesto, questo progetto si propone come uno strumento chiave per l'identificazione e la prevenzione delle frodi nell'e-commerce. L'obiettivo è sviluppare un sistema di database robusto e affidabile progettato per monitorare, analizzare e identificare comportamenti sospetti e attività irregolari nelle transazioni online.

Per raggiungere questo obiettivo il sistema sfrutta una vasta gamma di informazioni, queste includono i dati sugli utenti, le transazioni effettuate, i prodotti acquistati e le segnalazioni di frode già verificate. Ciascuna di queste categorie fornisce una parte dello schema complessivo grazie al quale l'analisi dei modelli di comportamento degli utenti può rivelare anomalie come frequenti transazioni di grandi dimensioni o acquisti ripetuti di un singolo prodotto, contribuendo quindi a rendere il commercio elettronico un ambiente più sicuro e affidabile per consumatori e venditori.

Soluzione DBMS considerata

In questo progetto verranno utilizzati due database per la gestione e l'analisi dei dati, ognuno di essi presenta un insieme di caratteristiche che lo rendono particolarmente adatto a specifici tipi di operazioni e analisi.

Iniziamo con una breve presentazione delle principali caratteristiche di MongoDB, il primo dei due database.

MongoDB è un database di tipo NoSQL orientato ai documenti in cui i dati vengono archiviati utilizzando il formato BSON, una variante binaria del formato JSON.

I documenti in MongoDB sono insiemi di coppie *chiave – valore*, vengono organizzati all'interno di collezioni tramite le quali è possibile raggruppare entità simili senza richiedere una struttura uniforme. Questa flessibilità rende MongoDB particolarmente adatto per gestire grandi volumi di dati non strutturati o semi-strutturati.

MongoDB supporta la scalabilità orizzontale attraverso il meccanismo dello sharding permettendo una distribuzione automatica dei dati su più server, questa caratteristica facilita l'ampliamento della capacità di elaborazione del database andando semplicemente ad aggiungere più server al sistema.

Un'altra importante caratteristica di MongoDB è l'uso dei Replica Set per garantire la disponibilità dei dati infatti, in caso di guasto di un nodo, le operazioni possono essere automaticamente reindirizzate verso un altro nodo del Replica Set assicurando così la continuità del servizio.

Passiamo ora al secondo database utilizzato nel progetto, ovvero Neo4j.

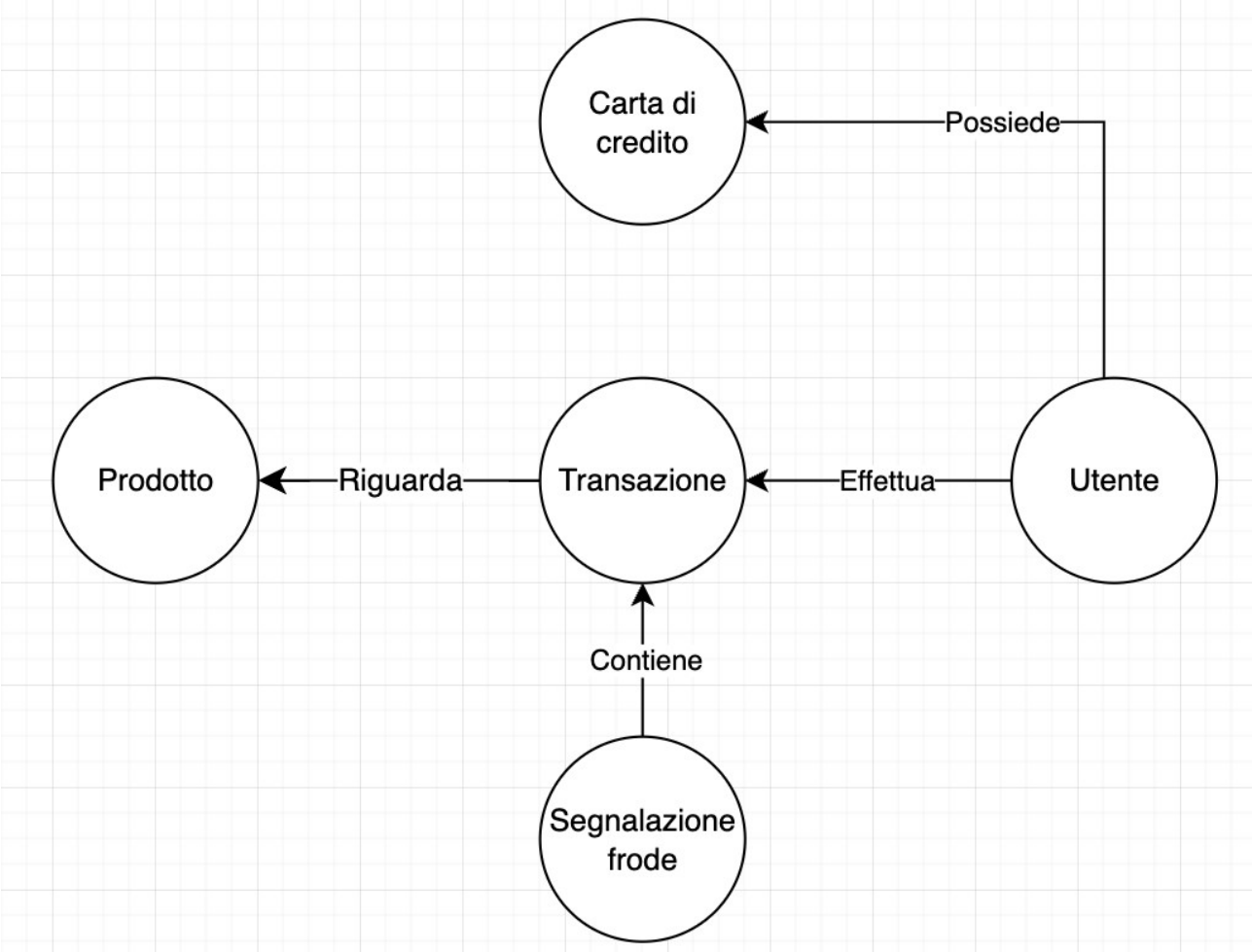
Neo4j è un database di tipo NoSQL orientato ai grafi, questa caratteristica è particolarmente utile in contesti come i social network o il rilevamento delle frodi nell'e-commerce in quanto permette di rappresentare le relazioni tra i dati in modo naturale.

Il modello di dati a grafo di Neo4j si basa su nodi, archi e proprietà. I nodi rappresentano le entità e incorporano al loro interno le proprietà li caratterizzano, gli archi possiedono una direzione e rappresentano le relazioni tra i nodi consentendo una rappresentazione visiva chiara delle relazioni tra i dati.

Uno dei punti di forza di Neo4j è il suo linguaggio di interrogazione Cypher che permette di formulare query complesse per recuperare dati specifici dai grafi.

L'ultimo aspetto di Neo4j di cui vogliamo parlare è la sua natura senza schema, infatti non è necessario definire uno schema rigido in anticipo consentendo quindi una maggiore flessibilità nell'aggiungere, modificare o rimuovere nodi, archi e proprietà.

Design



Lo schema soprastante rappresenta le relazioni tra i dati: in particolare abbiamo che, per quanto riguarda gli utenti, i dati che andiamo a rappresentare sono:

- **ID Utente**: Identificativo univoco per ogni utente, generato con un numero casuale univoco.
- **Nome**: Il nome dell'utente.
- **Cognome**: Il cognome dell'utente.
- **Email**: L'email dell'utente.
- **Stato Account**: Lo stato dell'account dell'utente, selezionato casualmente da una lista di stati account.
- Riferimenti alla transazione e alla carta di credito, per poter effettuare eventuali operazioni di join in MongoDB

Passando alla carta di credito abbiamo:

- **Numero Carta di Carta**: Numero univoco della carta di credito.
- **Scadenza**: La data di scadenza della carta di credito.
- **Banca**: Il nome della banca che ha emesso la carta di credito.

Per quanto riguarda le transazioni abbiamo usato:

- **ID Transazione**: Identificativo univoco per ogni transazione.
- **Data**: La data in cui è stata effettuata la transazione
- **Metodo di Pagamento**: Il metodo di pagamento utilizzato, selezionato casualmente da una lista di metodi di pagamento.
- **Indirizzo IP**: L'indirizzo IP da cui è stata effettuata la transazione.
- **Browser**: Il browser utilizzato per effettuare la transazione.
- **Paese**: Il paese da cui è stata effettuata la transazione.
- Riferimento al prodotto per poter effettuare l'operazione di join in MongoDB

Proseguiamo con le segnalazioni di frode:

- **ID Segnalazione**: Identificativo univoco per ogni segnalazione di frode.
- **Data**: La data in cui è stata segnalata la frode.
- **Dettagli**: Dettagli specifici o descrizione della segnalazione di frode
- Riferimento alla transazione per poter effettuare eventuali operazioni di join in MongoDB

Ed infine per quanto riguarda il prodotto:

- **ID Prodotto**: Identificativo univoco per ogni prodotto.
- **Nome Prodotto**: Il nome del prodotto, selezionato casualmente da una lista di nomi di prodotti.
- **Categoria**: La categoria del prodotto, selezionata casualmente da una lista di categorie di prodotti.
- **Prezzo**: Il prezzo del prodotto.

Per quanto riguarda la loro generazione è stata usata faker, una libreria di Python che consente di generare dati fittizi. Di seguito viene riportato lo script per intero per generare i file csv i quali saranno successivamente importati dai database.

```python
for dimensione in dimensioni_dataset:
    nome_file = "./Dati/dati_" + str(dimensioni_dataset) + ".csv"
    with open(nome_file, 'w', newline='') as file:
        writer = csv.writer(file)

        # Intestazione del CSV
        writer.writerow(["ID Transazione", "Data", "Metodo di Pagamento", "Indirizzo IP", "Browser", "Paese",

        # Generazione dei record (1.000, 10.000, 100.000, 1.000.000)
        for x in range(dimensione):
            writer.writerow([
                fake.unique.random_number(digits=10),  # ID Transazione
                fake.date(),  # Data
                random.choice(metodi_pagamento),  # Metodo di Pagamento
                fake.ipv4(),  # Indirizzo_IP
                fake.user_agent(),  # Browser
                fake.country(),  # Paese
                fake.random_number(digits=10),  # Rif Prodotto
                fake.random_number(digits=10),  # Rif Utente
                fake.unique.random_number(digits=10),  # ID Utente
                fake.first_name(),  # Nome
                fake.last_name(),  # Cognome
                fake.email(),  # Email
                random.choice(stati_account),  # Stato Account
                fake.random_number(digits=10),  # Rif Carta di Credito
                fake.unique.random_number(digits=10),  # ID Prodotto
                random.choice(nome_prodotti), # Nome Prodotto
                fake.random_number(digits=4),  # Prezzo
                fake.unique.random_number(digits=10),  # ID Segnalazione
                fake.date(),  # Data_Segnalazione
                fake.text(max_nb_chars=200),  # Dettagli
                fake.random_number(digits=10),  # Rif Transazione
                fake.unique.random_number(digits=10), # ID Carta di Credito
                fake.credit_card_number(),  # Numero Carta Di Credito
                fake.date_time_between(start_date='now', end_date='+5y').date(),  # Scadenza
                fake.company()  # Banca
            ])
```

Implementazione

In questa fase verranno riportate e descritte le parti principali degli script utilizzati per importare i quattro dataset nei due database considerati, successivamente verrà descritta l'implementazione delle quattro query. L'implementazione è stata eseguita sfruttando il linguaggio Python.

Inserimento dei dati

Per quanto riguarda l'inserimento dei dati, partiamo da MongoDB.

```
client = pymongo.MongoClient("mongodb://root:root@localhost:27017/")
data_sets = ["Dataset25", "Dataset50", "Dataset75", "Dataset100"]
dimensioni = 100000
```

In questo primo blocco il programma si connette al server MongoDB utilizzando la funzione MongoClient di PyMongo. L'URL del server è specificato come parametro. Si definiscono le dimensioni del dataset e l'array data_sets che contiene i nomi dei vari dataset utilizzati.

```
for i in range(0, 4):
    file = "Dati/dati_" + str(dimensioni) + ".csv"
    dimensioni += 100000

    # Selezione del database
    db = client[data_sets[i]]


    # Selezione delle collezioni
    col_transazioni = db["Transazione"]
    col_utenti = db["Utente"]
    col_carte = db["Carta_di_Credito"]
    col_prodotti = db["Prodotto"]
    col_segnalazioni = db["Segnalazione_Frode"]
```

Questo blocco di codice esegue un ciclo attraverso i quattro dataset. Per ogni iterazione seleziona un file CSV basato sulla dimensione corrente del dataset, incrementa la dimensione del dataset per la prossima iterazione e poi seleziona il database MongoDB corrispondente. Successivamente, all'interno di ciascun

database, seleziona le specifiche collezioni con cui lavorare: "Transazione", "Utente", "Carta_di_Credito", "Prodotto" e "Segnalazione_Frode".

```
with open(file, 'r') as file:
    reader = csv.DictReader(file)

    # Inserimento dei dati nelle rispettive collezioni
    for row in reader:
        # Inserimento utente
        utente = {
            "_id": int(row["ID Utente"]),
            "Nome": row["Nome"],
            "Cognome": row["Cognome"],
            "Email": row["Email"],
            "Stato_Account": row["Stato Account"],
            "Rif_Carta_di_Credito": int(row["ID Carta di Credito"]),
            "Rif_Transazione": int(row["ID Transazione"])
        }
        col_utenti.insert_one(utente)
```

Per concludere nel seguente blocco di codice il programma apre in lettura il file CSV selezionato e crea un oggetto DictReader che consente di accedere ai dati del CSV come dizionari. Successivamente itera attraverso ogni riga del file CSV e per ognuna di queste estrae le informazioni necessarie, crea un nuovo dizionario che rappresenta un utente ed infine lo inserisce nella collezione "Utenti" del database. La procedura viene ripetuta per tutte le collezioni precedentemente indicate.

Passando ora a Neo4j l'importazione dei dati avviene in maniera simile tramite l'utilizzo della libreria py2neo. Vengono prima creati i nodi

```
def import_data(data):
    tx = graph.begin()
    count = 0

    for row in data:
        count += 1
        transazione = Node("Transazione", id=row["ID Transazione"], Data=row[
        prodotto = Node("Prodotto", id=row["ID Prodotto"], Nome_Prodotto=row[
        utente = Node("Utente", id=row["ID Utente"], Nome=row["Nome"], Cognom
        carta_di_credito = Node("Carta_di_Credito", id=row["ID Carta di Credi
        segnalazione = Node("Segnalazione_frode", id=row["ID Segnalazione"],

        tx.create(transazione)
        tx.create(prodotto)
        tx.create(utente)
        tx.create(carta_di_credito)
        tx.create(segnalazione)
```

Successivamente vengono create le relazioni seguendo lo schema del modello prima descritto.

```
possiede_utente = Relationship(utente, "POSSIEDE", carta_di_credito)
effettua_utente = Relationship(utente, "EFFETTUA", transazione)
riguarda_transazione = Relationship(transazione, "RIGUARDA", prodotto)
contiene_segnalazione = Relationship(segnalazione, "CONTIENE", transazione)

tx.create(possiede_utente)
tx.create(effettua_utente)
tx.create(riguarda_transazione)
tx.create(contiene_segnalazione)
```

Query

Per quanto riguarda la prima query vengono recuperati i dati di un utente specifico. Di seguito le rappresentazioni, rispettivamente in Neo4j e MongoDB.

```
def query1(user_id, stampa):
    result = tx.run("MATCH (u:Utente {id: $user_id}) RETURN u", user_id=user_id)
    return 1
```

```
def query1(db, user_id, prima_esecuzione, stampa):
    # Ricerca di un utente specifico
    if (prima_esecuzione):
        db.command({"planCacheClear": "Transazione"})
        db.command({"planCacheClear": "Utente"})
        db.command({"planCacheClear": "Segnalazione_Frode"})
        db.command({"planCacheClear": "Carta_di_Credito"})
        db.command({"planCacheClear": "Prodotto"})

    result = db.Utente.find({"_id": user_id})

    if (stampa):
        print(list(result))
    return 1
```

Passando poi alla seconda query vengono recuperati i prodotti il cui prezzo rientra nel range specificato. Anche in questo caso mostriamo prima la query in MongoDB e successivamente quella in Neo4j.

```python
def query2(min_price, max_price, stampa):
    result = tx.run("""
        MATCH (p:Prodotto)
        WHERE p.prezzo >= $min_price AND p.prezzo <= $max_price RETURN p",
        min_price=min_price, max_price=max_price""")
    return 1
```

```python
def query2(db, min_price, max_price, prima_esecuzione, stampa):
    # Ricerca numero prodotti nel range di prezzo specificato
    if (prima_esecuzione):
        db.command({"planCacheClear": "Transazione"})
        db.command({"planCacheClear": "Utente"})
        db.command({"planCacheClear": "Segnalazione_Frode"})
        db.command({"planCacheClear": "Carta_di_Credito"})
        db.command({"planCacheClear": "Prodotto"})

    result = db.Prodotto.find({
        "Prezzo": {
            "$gte": min_price,
            "$lte": max_price
        }
    })

    if (stampa):
        print(list(result))
    return 1
```

La terza query effettua una ricerca delle transazioni che sono state effettuate da un certo paese con un certo metodo di pagamento tra quelli a disposizione partendo da una data specifica. Di seguito le rappresentazioni, rispettivamente in Neo4j e MongoDB.

```python
def query3(metodo_pagamento, year, month, day, paese, stampa):
    result = tx.run("""
        MATCH (t:Transazione)
        WHERE t.Metodo_di_Pagamento = $metodo_pagamento and
        t.Data >= date({year: $year, month: $month, day: $day}) and t.Paese = $paese
        RETURN t
    """, year=year, month=month, day=day, metodo_pagamento=metodo_pagamento, paese=paese)
    return 1
```

```python
def query3(db, metodo_pagamento, year, month, day, paese, prima_esecuzione, stampa):
    # Ricerca transazioni che sono state effettuate con un certo metodo di pagamento, da un certo paese e dopo una certa data
    if (prima_esecuzione):
        db.command({"planCacheClear": "Transazione"})
        db.command({"planCacheClear": "Utente"})
        db.command({"planCacheClear": "Segnalazione_Frode"})
        db.command({"planCacheClear": "Carta_di_Credito"})
        db.command({"planCacheClear": "Prodotto"})

    result = db.Transazione.aggregate([
        {
            "$match": {
                "Metodo_di_Pagamento": metodo_pagamento
            }
        },
        {
            "$match": {
                "Data": {"$gte": datetime(year, month, day)}
            }
        },
        {
            "$match": {
                "Paese": paese
            }
        },
    ])

    if (stampa):
        print(list(result))
    return 1
```

Ed infine la quarta query ricerca le transazioni che riguardano prodotti che sono stati acquistati dopo una certa data, il risultato viene ordinato prima per data e poi per prezzo, viene prima mostrata l'implementazione in Neo4j e successivamente per MongoDB.

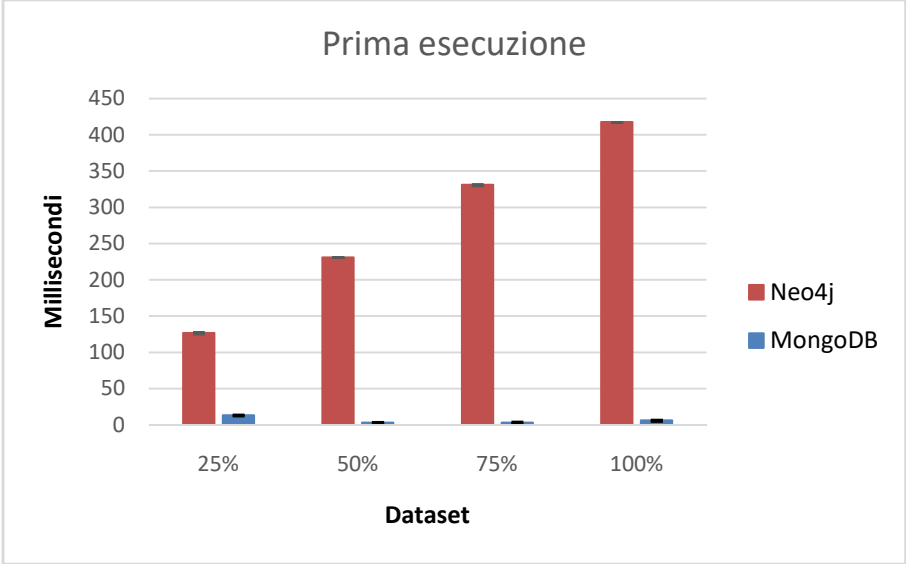
```python
def query4(year, month, day, stampa):
    result = tx.run("""
        MATCH (p:Prodotto)<-[:RIGUARDA]-(t:Transazione)
        WHERE t.Data > date({year: $year, month: $month, day: $day})
        RETURN t
        ORDER BY t.Data, p.Prezzo
    """, year=year, month=month, day=day)
    return 1
```

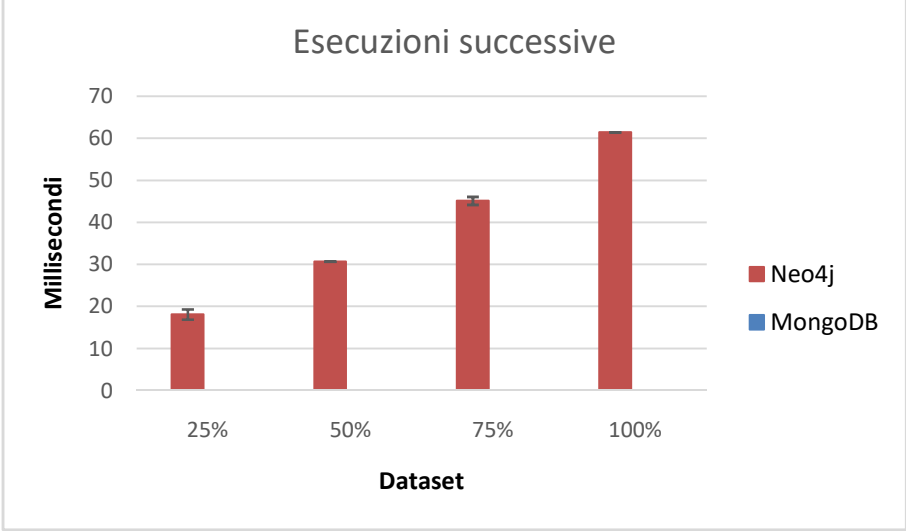
```python
def query4(db, year, month, day, prima_esecuzione, stampa):
    # Ricerca transazioni che sono state effettuate per acquistare prodotto dopo una certa data in ordine ascendente
    if (prima_esecuzione):
        db.command({"planCacheClear": "Transazione"})
        db.command({"planCacheClear": "Utente"})
        db.command({"planCacheClear": "Segnalazione_Frode"})
        db.command({"planCacheClear": "Carta_di_Credito"})
        db.command({"planCacheClear": "Prodotto"})

    result = db.Transazione.aggregate([
        { "$match": {
            "Data": { "$gt": datetime(year, month, day) }
          }
        },
        {
          "$lookup": {
            "from": "Prodotto",
            "localField": "Rif_Prodotto",
            "foreignField": "_id",
            "as": "prodotto"
          }
        },
        {
          "$sort": {
            "Data": 1,
            "Prezzo": 1
          }
        },
        {
          "$project": {
            "Transazione": "$$ROOT"
          }
        }
    ])
```

Esperimenti

1 Query

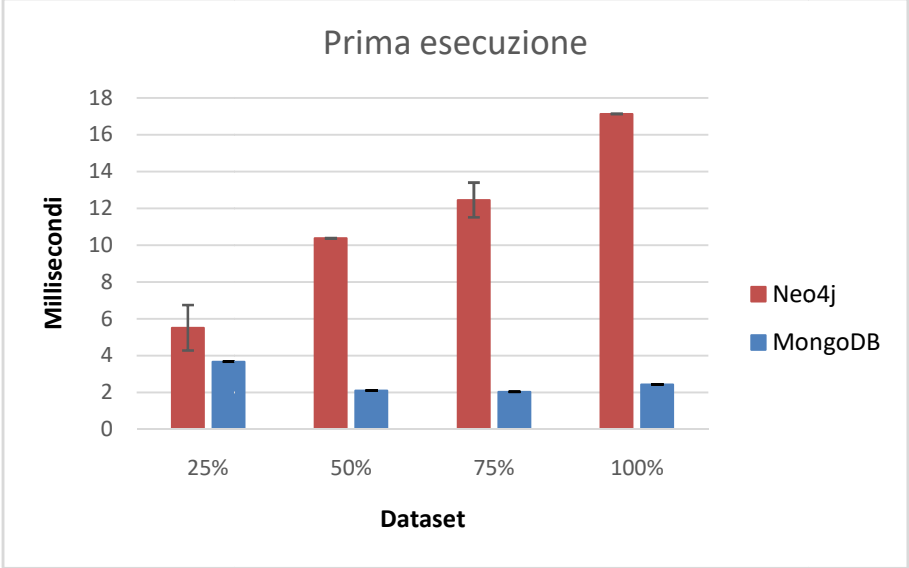
	Neo4j	MongoDB	Neo4j	MongoDB	Neo4j	MongoDB	Neo4j	MongoDB
Primi tempi	126,713834	13,23824999	231,049084	2,997707983	331,024042	3,137541004	417,55175	6,012541999
Media	18,14399443	0,004901434	30,70314303	0,004931991	45,17302913	0,004524961	61,47669037	0,004565235
Deviazione standard	3,444101898	0,002365587	2,637628653	0,001260261	4,805463223	0,000932342	11,98558213	0,000751785
95% confidenza	1,232433389	0,000846499	0,943845948	0,00045097	1,719581331	0,000333628	4,28890667	0,000269018
Percentuale dataset	25%		50%		75%		100%	

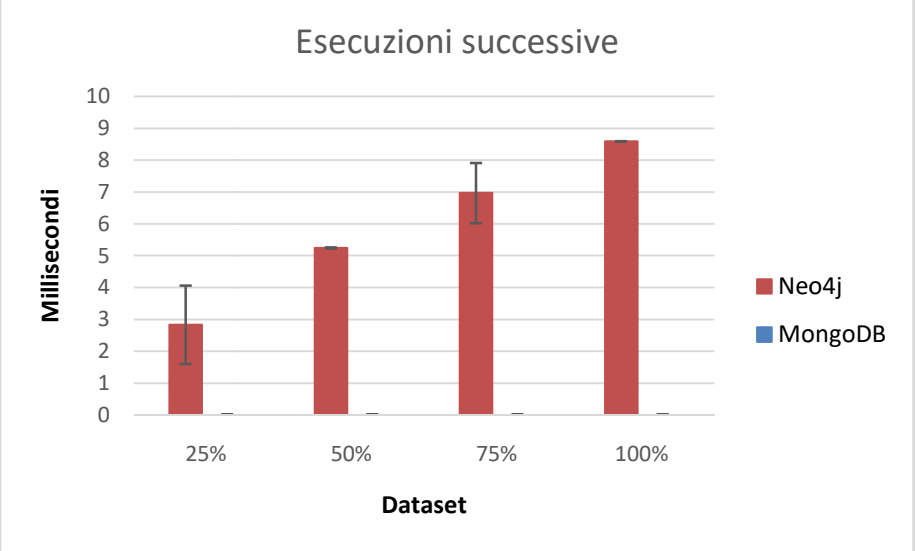




2 Query

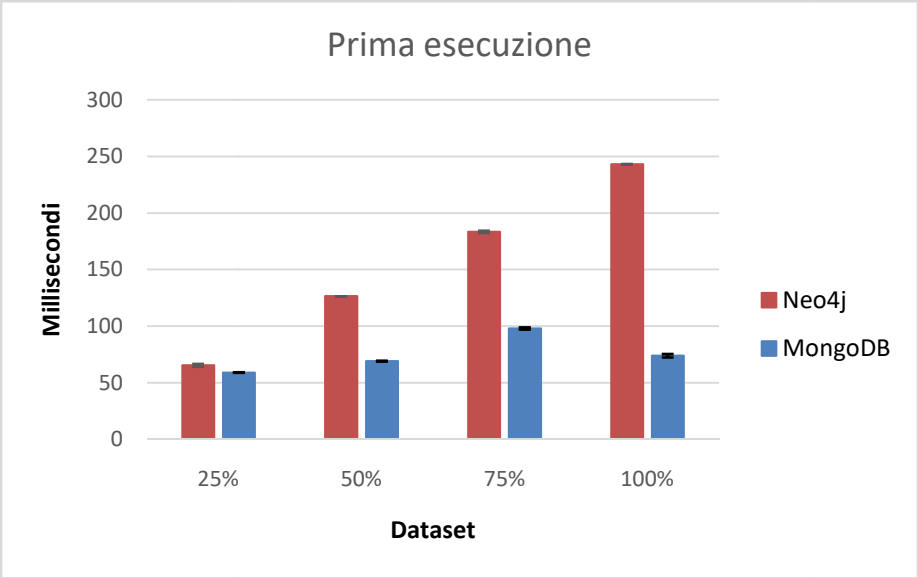
	Neo4j	MongoDB	Neo4j	MongoDB	Neo4j	MongoDB	Neo4j	MongoDB
Primi tempi	5,514500022	3,667375015	10,378459	2,099084028	12,46070804	2,033625031	17,12912502	2,430708962
Media	2,837597202	0,0041695	5,247045764	0,004238872	6,970825029	0,004373502	8,595323675	0,004616764
Deviazione standard	0,468388815	0,001047798	1,748256315	0,000593967	1,140187239	0,000356959	0,191485906	0,000762555
95% confidenza	0,167607705	0,000374943	0,625593992	0,000212545	0,408003266	0,000127734	0,068521092	0,000272872
Percentuale dataset	25%		50%		75%		100%	

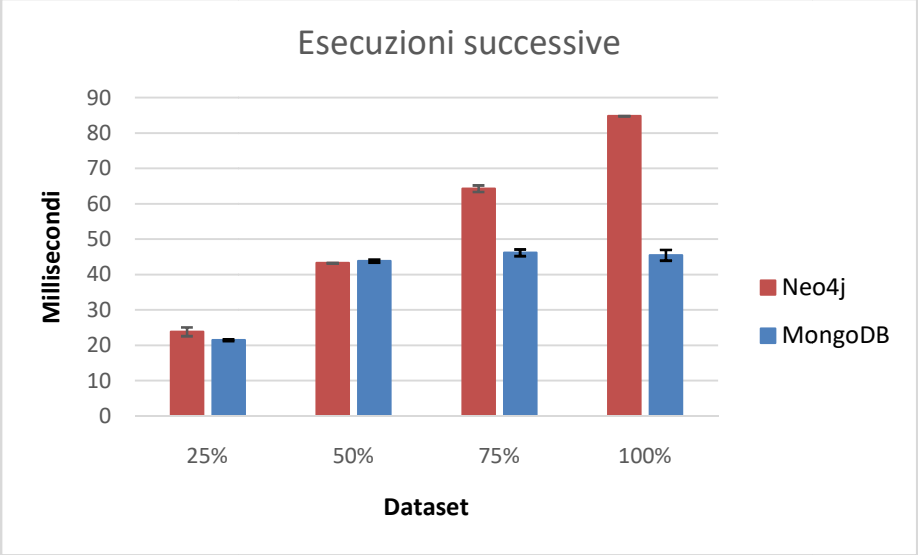




3 Query

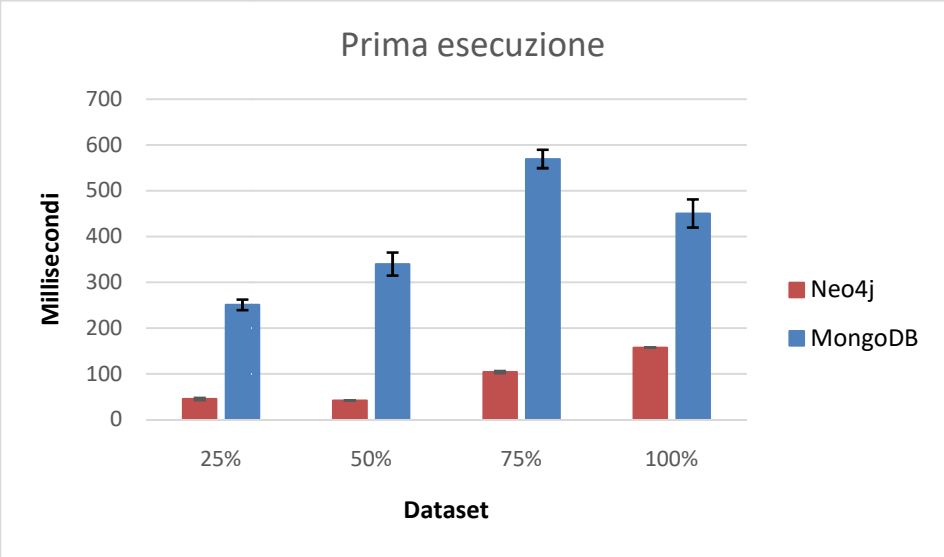
	Neo4j	MongoDB	Neo4j	MongoDB	Neo4j	MongoDB	Neo4j	MongoDB
Primi tempi	65,54808299	59,19712503	126,561375	69,26591595	183,577542	98,07629202	243,116916	73,99883302
Media	23,8949792	21,55172917	43,28553487	43,86474723	64,3175764	46,21887636	84,82299449	45,5157015
Deviazione standard	5,038306121	0,748164955	1,649806673	1,128327102	1,891547565	2,682927801	1,533011607	4,193274673
95% confidenza	1,802901561	0,267722471	0,590364888	0,403759249	0,676869165	0,960055742	0,548571078	1,500516497
Percentuale dataset	25%		50%		75%		100%	

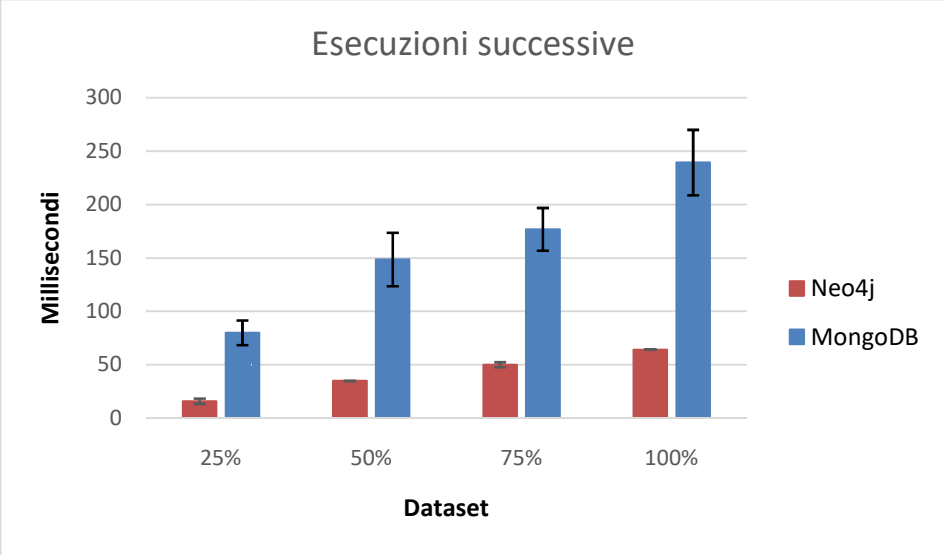




4 Query

	Neo4j	MongoDB	Neo4j	MongoDB	Neo4j	MongoDB	Neo4j	MongoDB
Primi tempi	45,171292	250,719625	41,865833	339,994958	104,00287	569,392834	157,56146	450,286333
Media	15,934279	79,97312087	34,949854	148,7124015	50,105869	176,9289638	64,181453	239,4180889
Deviazione standard	2,068016	32,23997004	9,5105203	69,93410095	7,353331	56,05704713	5,0928602	85,8017202
95% confidenza	0,7400164	11,53671312	3,4032334	25,02513677	2,6313074	20,05938809	1,8224231	30,70318706
Percentuale dataset	25%		50%		75%		100%	





Conclusione con Analisi dei Risultati

Le query sulle quali sono stati eseguiti i benchmark sono equivalenti in termini di funzionalità ma sono strutturate in modi diversi a causa delle differenze nei linguaggi di query e nei modelli di dati dei due database. Ecco una analisi dei risultati:

- **1 Query**: MongoDB ha dimostrato prestazioni significativamente migliori rispetto a Neo4j, dove la media dei tempi di esecuzione era quasi trascurabile rispetto a quella di Neo4j.
- **2 Query**: Ancora una volta MongoDB ha avuto una media dei tempi di esecuzione inferiore rispetto a Neo4j anche se la differenza non è stata così drastica come nella prima query.
- **3 Query:** In questi risultati è interessante notare come la media dei tempi di esecuzione per MongoDB e Neo4j sembrano essere molto più vicini rispetto alle precedenti query. Tuttavia per il dataset completo MongoDB continua ad avere tempi di esecuzione leggermente migliori.
- **4 Query**: Nei risultati di quest'ultima query Neo4j ha dimostrato di avere un tempo medio di esecuzione inferiore rispetto a MongoDB, ciò evidenzia che Neo4j rappresenta una soluzione più efficace per l'elaborazione di interrogazioni che prevedono una gestione complessa di relazioni tra i dati.

Nell'analisi complessiva dei risultati si può osservare come MongoDB ottenga risultati più performanti nelle operazioni di interrogazione 'convenzionali', ovvero quelle principalmente focalizzate sulle proprietà degli oggetti. Dall'altra parte Neo4j mostra una superiorità nelle situazioni in cui la complessità delle relazioni è un fattore predominante. Concludendo possiamo affermare che la scelta tra i due sistemi potrebbe essere guidata dal tipo specifico di operazioni e dalla struttura dei dati con cui si prevede di lavorare.