



# Simplifying UBO identification and verification with graph visualization and analytics

Basi di Dati NoSQL (a.A. 2024/2025)

Demetrio Priolo (543193) | Manuel R. Sciuto (547488) | 30/01/2025

## Introduzione

Negli ultimi anni, la necessità di analizzare e comprendere dati complessi è diventata sempre più critica in vari settori, tra cui quello della conformità finanziaria e dell'antiriciclaggio. In questo contesto, i database **NoSQL** si sono affermati come una soluzione efficace per la gestione di grandi volumi di dati non strutturati o semi-strutturati, offrendo scalabilità e flessibilità superiori rispetto ai tradizionali database relazionali.

Uno dei casi d'uso più rilevanti per i database **NoSQL**, in particolare i database a grafo come **Neo4j**, è l'identificazione e la verifica dei Beneficiari Effettivi Ultimi (UBO, Ultimate Beneficial Owners). Questo processo è essenziale per contrastare il riciclaggio di denaro e le attività illecite nascoste dietro strutture aziendali complesse. Grazie alla capacità di modellare le relazioni tra entità in modo intuitivo e performante, i database a grafo permettono di tracciare connessioni tra persone, aziende e transazioni finanziarie con maggiore efficacia rispetto ai tradizionali modelli tabellari.

Nel contesto del progetto trattato, l'uso di un database **NoSQL** a grafo consente di eseguire query avanzate per scoprire legami nascosti tra entità e identificare rapidamente gli UBO, migliorando così la trasparenza e l'efficacia delle verifiche di conformità normativa.

## **Soluzione DBMS considerata**

I DBMS utilizzati per affrontare il problema sono **MongoDB** e **Neo4j**, due sistemi di gestione di database estremamente differenti ma complementari, che adottano paradigmi e linguaggi di querying differenti.

- **MongoDB** si basa sul paradigma **NoSQL** e utilizza una struttura a documenti JSON/BSON per memorizzare dati non relazionali. Questo lo rende ideale per la raccolta di dati preliminari sulle aziende, come informazioni di base, elenchi di azionisti e documenti pubblici. Il suo linguaggio di querying, basato su **JSON**, è flessibile e consente interrogazioni rapide su dati non strutturati. MongoDB è particolarmente adatto per la gestione di grandi volumi di dati distribuiti e per scenari in cui la struttura dei dati può variare.
- **Neo4j**, invece, si basa sul paradigma **a grafo**, progettato specificamente per rappresentare e analizzare relazioni complesse tra entità. Utilizza il linguaggio **Cypher**, ottimizzato per query relazionali avanzate come il tracciamento di connessioni tra nodi. Questo lo rende ideale per analizzare le strutture multilivello delle proprietà aziendali e per individuare l'**UBO** attraverso catene di proprietà complesse.

Utilizzando entrambi questi DBMS possiamo metterli a paragone in modo da sfruttare le loro diverse potenzialità e trovare il modo migliore e più veloce per poter individuare gli UBO.

## Progettazione

### Caso di studio

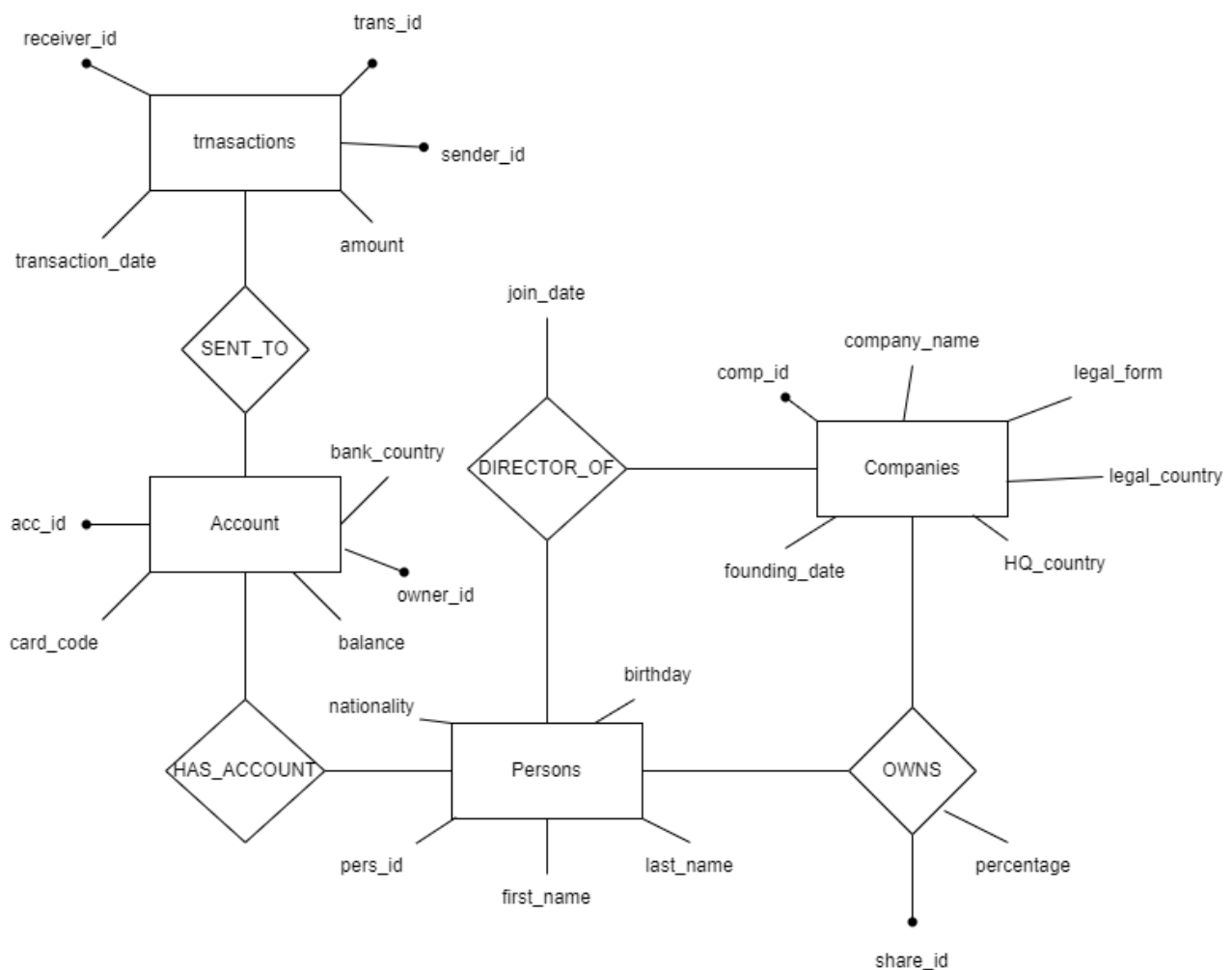
Le strutture delle proprietà aziendali possono risultare complesse e avere molti livelli tra la persona avente il controllo e le aziende sottoposte, questo permette al nostro “Ultimate Beneficial Owners” o UBO di sfruttare queste aziende per attività criminali in maniera più sicura.

### **Le sfide principali:**

- **Dati distribuiti e non standardizzati:** Le informazioni sulle strutture aziendali possono essere sparse tra molteplici fonti e formati.
- **Strutture multilivello:** Le aziende possono essere possedute da altre entità in catene o reti complesse, rendendo difficile risalire all’UBO finale.
- **Giurisdizioni multiple:** Quando i dati aziendali attraversano confini nazionali, la mancanza di standard uniformi complica ulteriormente il processo.
- **Rischio di falsi positivi:** Senza strumenti adeguati, identificare erroneamente un UBO o tralasciare dettagli critici può comportare rischi legali e reputazionali.

Inoltre, la difficoltà di accesso ai dati aggiornati e la presenza di società offshore complicano ulteriormente l’identificazione dell’UBO. Le entità fittizie e le holding stratificate possono celare il vero proprietario, rendendo necessaria un’analisi approfondita delle connessioni tra aziende. Senza strumenti adeguati, il processo diventa lungo e inefficace, aumentando il rischio di lasciare spazi a operazioni illecite. Per affrontare il problema, è essenziale adottare un approccio basato su tecnologie avanzate in grado di elaborare e correlare grandi quantità di dati in modo rapido ed efficace.

## **Datamodel**



Dal seguente data model si evincono le seguenti entità:

- Transactions: trans\_id, sender\_id, receiver\_id, amount, transaction\_date;
- Account: acc\_id, owner\_id, card\_code, balance, bank\_country;
- Persons: pers\_id, first\_name, last\_name, birthday, nationality;
- Companies: comp\_id, company\_name, legal\_form, legal\_country, HQ\_country, founding\_date;

e le seguenti relazioni:

- HAS\_ACCOUNT: che implementa la relazione di possesso di un account tra le entità persons e account;

- OWNS: che implementa la relazione di azionismo tra persons e shareholders;
- DIRECTOR\_OF: che implementa la relazione di dirigenza tra una persona e un'azienda;
- SENT\_TO : che implementa la relazione di aver effettuato una transazione tra account.

## **Implementazione e Inserimento**

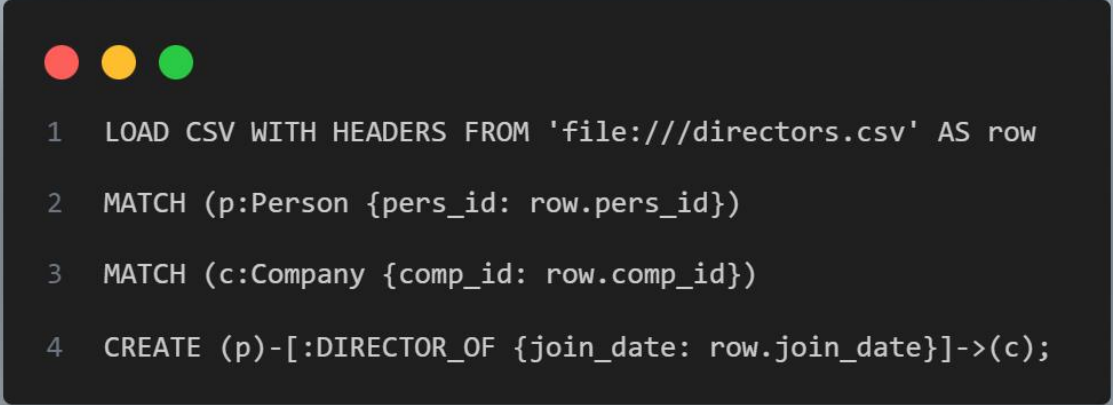
I dati utilizzati per effettuare i test sono stati generati attraverso script Python, utilizzando la libreria Faker e inseriti all'interno di file CSV che in seguito sono stati esportati nei rispettivi database. Questa è la struttura scelta per i dati:

	<b>25%</b>	<b>50%</b>	<b>75%</b>	<b>100%</b>
<b>People</b>	<b>15000</b>	<b>30000</b>	<b>45000</b>	<b>60000</b>
<b>Companies</b>	<b>6250</b>	<b>12500</b>	<b>18750</b>	<b>25000</b>
<b>Accounts</b>	<b>18750</b>	<b>37500</b>	<b>56250</b>	<b>75000</b>
<b>Directors</b>	<b>12500</b>	<b>25000</b>	<b>37500</b>	<b>50000</b>
<b>Transactions</b>	<b>37500</b>	<b>75000</b>	<b>112500</b>	<b>150000</b>
<b>Shareholders</b>	<b>25000</b>	<b>50000</b>	<b>75000</b>	<b>100000</b>

## **Neo4j**

In Neo4j, le entità vengono rappresentate come etichette (labels). Per gestire i database Neo4j e caricare i file CSV generati, è stato utilizzato Neo4j Desktop. Esso offre un'interfaccia intuitiva per la

creazione dei database. Nell'applicazione desktop sono stati creati quattro database Neo4j, ciascuno corrispondente a una diversa dimensione del dataset (25%, 50%, 75%, 100%). Per ogni database, è necessario inserire i file CSV nella corrispettiva cartella import. Di seguito viene riportato l'esempio di query che crea la relazione tra le persone e i direttori:



```
1  LOAD CSV WITH HEADERS FROM 'file:///directors.csv' AS row
2  MATCH (p:Person {pers_id: row.pers_id})
3  MATCH (c:Company {comp_id: row.comp_id})
4  CREATE (p)-[:DIRECTOR_OF {join_date: row.join_date}]->(c);
```

## MongoDB

MongoDB Compass consente di importare dati in modo intuitivo attraverso un'interfaccia grafica. Per eseguire l'importazione, è sufficiente accedere al database, selezionare o creare una collezione e utilizzare l'opzione "Add Data" → "Import File". Dopo aver scelto un file JSON o CSV, è possibile configurare le opzioni di importazione, come l'uso della prima riga come



intestazione per i campi. Infine, confermando l'operazione, i dati vengono inseriti nella collezione senza la necessità di scrivere codice.

## **Esperimenti**

Per testare i due DBMS, abbiamo eseguito quattro query di difficoltà crescente su ciascuno, garantendo un confronto equo per valutarne vantaggi e svantaggi. Ogni query è stata eseguita 31 volte: la prima esecuzione fornisce il tempo iniziale, mentre le 30 successive, grazie agli algoritmi di caching, risultano notevolmente più rapide.

## Neo4j

```
1 def execute_query(query, query_num):
2     filename = f"query_{query_num}_timer.csv"
3     times = []
4
5     with driver.session() as session:
6         for i in range(31):
7             print(f"Esecuzione {i + 1} per la query {query_num}")
8             start_time = time.perf_counter()
9             try:
10                 session.run(query)
11             except Exception as e:
12                 print(f"Errore durante l'esecuzione della query {query_num}: {e}")
13                 times.append(float('nan')) # Registra il fallimento con NaN
14                 continue
15             end_time = time.perf_counter()
16             execution_time = (end_time - start_time) * 1000
17             times.append(execution_time)
18
19     with open(filename, "w", newline="") as csvfile:
20         writer = csv.writer(csvfile)
21         writer.writerow(["Run", "Execution Time"])
22         for i, exec_time in enumerate(times):
23             writer.writerow([i + 1, exec_time])
24
25     # Calcola e salva la media delle esecuzioni (escludendo NaN e la prima esecuzione)
26     valid_times = [t for t in times[1:] if not (t != t)] # esclude NaN
27     if valid_times:
28         average_time = sum(valid_times) / len(valid_times)
29         writer.writerow(["Media", average_time])
30     else:
31         writer.writerow(["Media", "N/A"])
32
33 # Esegui le query una alla volta e salva i tempi in file CSV separati
34 for i, query in enumerate(query_list):
35     execute_query(query, i + 1)
36
37 print("Script completato. I tempi di esecuzione sono stati salvati nei file CSV.")
38
39 # Chiudi il driver al termine
40 driver.close()
```

## MongoDB

```

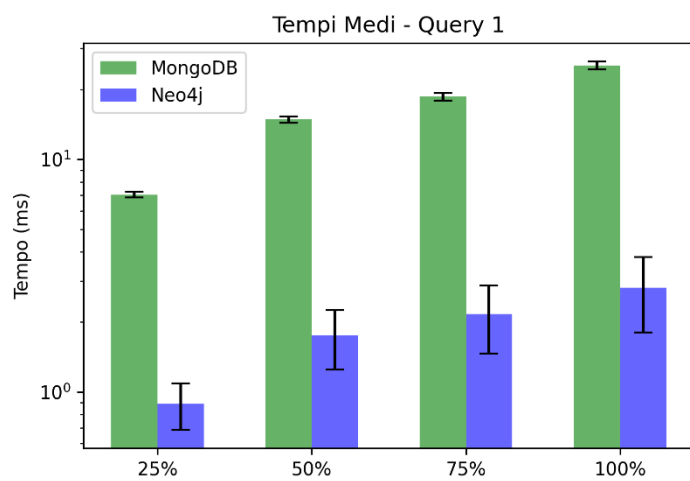
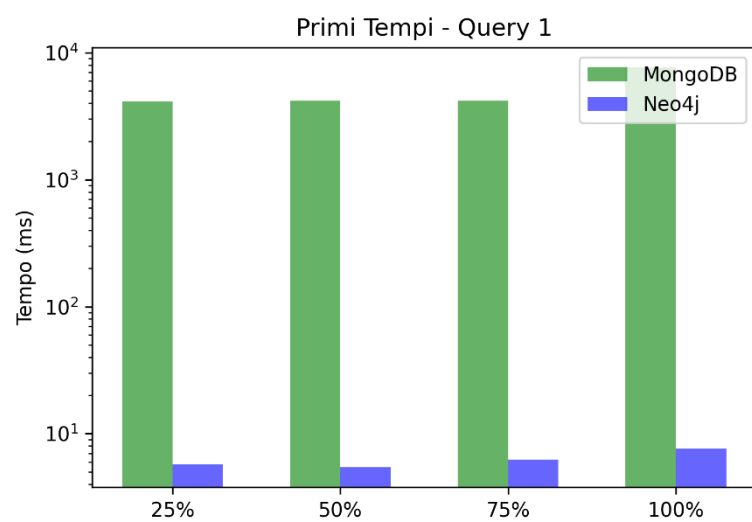
1  def query1():
2      return db.shareholders.aggregate([
3  ])
4
5  def query2():
6      return db.transactions.aggregate([
7
8  ])
9
10
11 def query3():
12     return db.shareholders.aggregate([
13
14 ])
15
16
17 def query4():
18     return db.directors.aggregate([
19 ])
20
21
22
23 queries = [query1, query2, query3, query4]
24
25
26 def execute_query(query_func, query_num):
27     filename = f"times_query_{query_num}.csv"
28     times = []
29
30     for i in range(31):
31         print(f"Esecuzione {i + 1} per la query {query_num}")
32         start_time = time.perf_counter()
33         try:
34             list(query_func())
35         except Exception as e:
36             print(f"Errore durante l'esecuzione della query {query_num}: {e}")
37             times.append(float('nan'))
38             continue
39         end_time = time.perf_counter()
40         execution_time = (end_time - start_time) * 1000
41         times.append(execution_time)
42
43     with open(filename, "w", newline="") as csvfile:
44         writer = csv.writer(csvfile)
45         writer.writerow(["Run", "Execution Time"])
46         for i, exec_time in enumerate(times):
47             writer.writerow([i + 1, exec_time])

```

## Query 1

La prima query estrae l'elenco degli azionisti e delle aziende in cui detengono partecipazioni, ordinando per nome dell'azienda, percentuale di proprietà e nominativi degli azionisti. Calcola anche il numero di quote e la percentuale totale di proprietà per ciascun azionista.

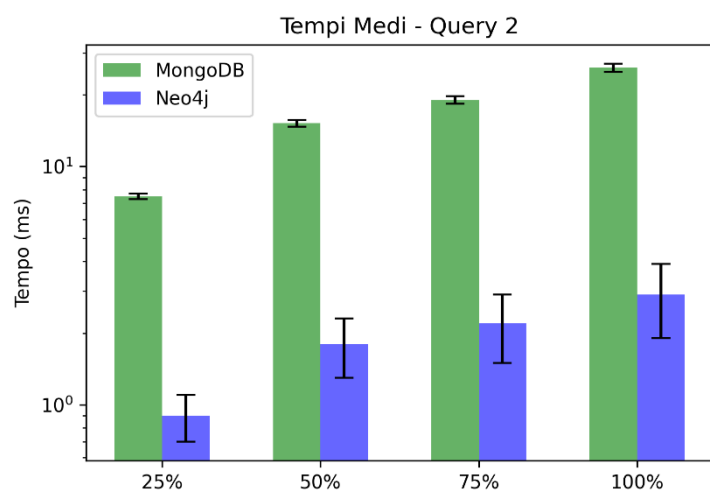
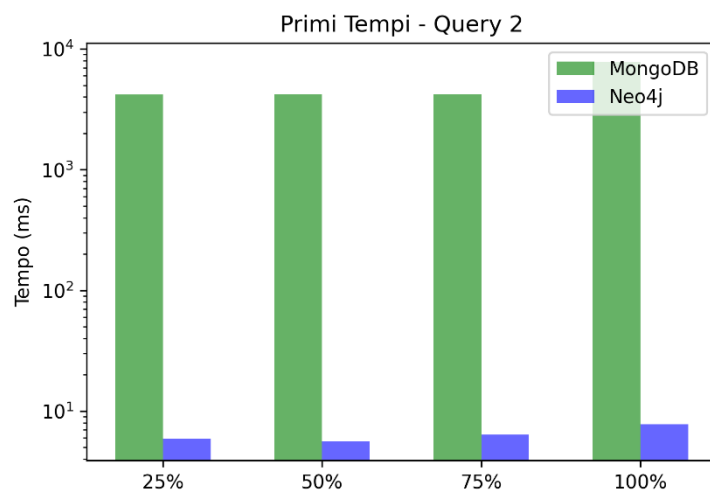
```
1 MATCH (person:Person)-[owns:OWNS]->(company:Company)
2 WITH person, company, owns.percentage AS ownershipPercentage, owns
3 ORDER BY company.name ASC, ownershipPercentage DESC, person.last_name ASC, person.first_name ASC
4 WITH
5     person.first_name AS Shareholder,
6     person.last_name AS LastName,
7     company.company_name AS Company,
8     company.comp_id AS CompanyID,
9     ownershipPercentage,
10    COUNT(owns) AS NumberOfShares,
11    SUM(ownershipPercentage) AS TotalOwnership
12 RETURN
13     Shareholder,
14     LastName,
15     Company,
16     CompanyID,
17     ownershipPercentage AS OwnershipPercentage,
18     TotalOwnership,
19     NumberOfShares
20
```



## Query 2

Identifica le persone che hanno inviato transazioni superiori a 10.000 unità monetarie da un conto bancario situato in un paese diverso dalla loro nazionalità, evidenziando potenziali operazioni sospette.

```
1 MATCH (person:Person)-[:HAS_ACCOUNT]->(account:Account)-[:SENT_TO]->(receiver:Account)
2 WHERE person.nationality <> account.bank_country AND account.balance > 10000
3 RETURN person.first_name, person.last_name, SUM(account.balance) AS total_amount
4 ORDER BY total_amount DESC;
```

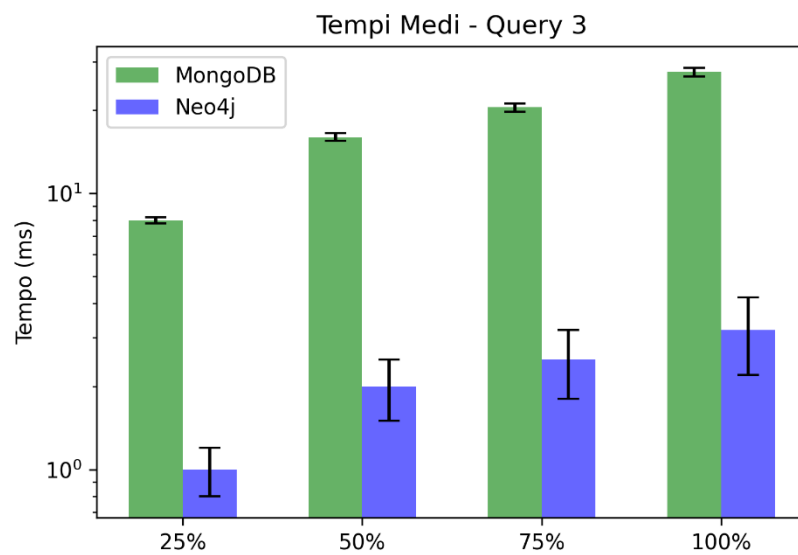
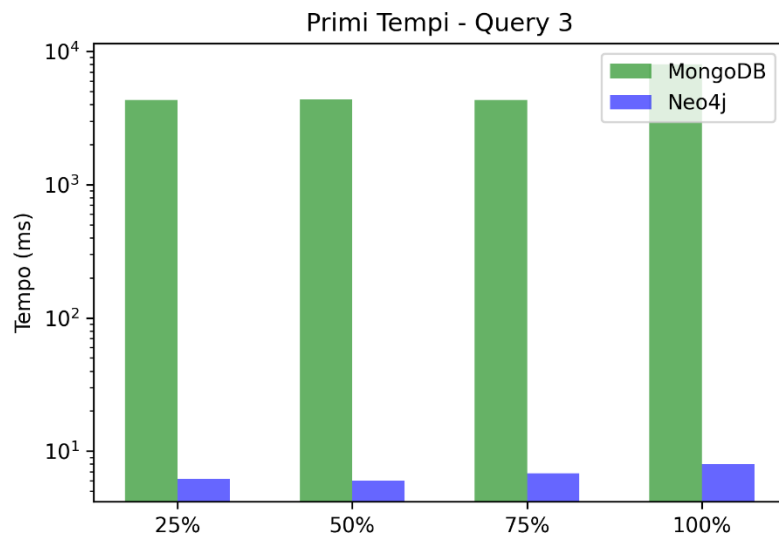


## Query 3

Determina il numero totale di azionisti per ogni azienda, la percentuale totale di proprietà e il numero complessivo di quote, ordinando le aziende in base alla percentuale di proprietà aggregata.



```
1 MATCH (person:Person)-[owns:OWNS]->(company:Company)
2 WITH
3     person.first_name AS Shareholder,
4     person.last_name AS LastName,
5     company.company_name AS Company,
6     company.comp_id AS CompanyID,
7     owns.percentage AS OwnershipPercentage,
8     COUNT(owns) AS NumberOfShares
9 WITH
10    Company,
11    CompanyID,
12    COUNT(DISTINCT Shareholder) AS TotalShareholders,
13    SUM(OwnershipPercentage) AS TotalOwnership,
14    SUM(NumberOfShares) AS TotalShares
15 ORDER BY TotalOwnership DESC, Company ASC
16 RETURN
17     Company,
18     CompanyID,
19     TotalShareholders,
20     TotalOwnership,
21     TotalShares
```

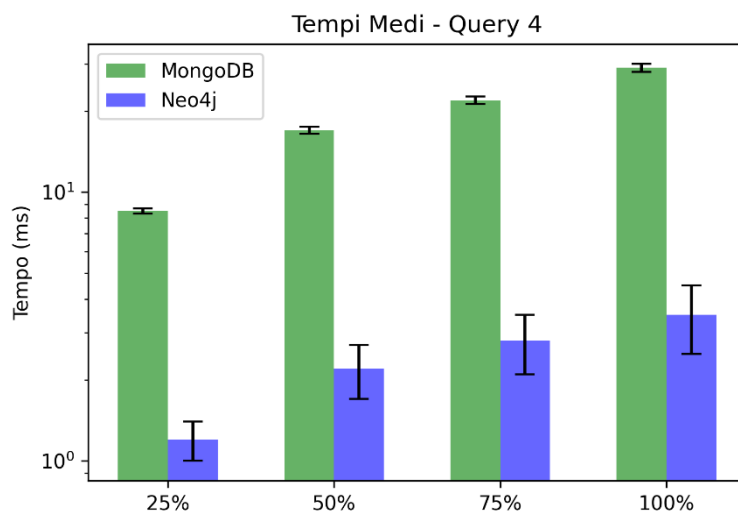
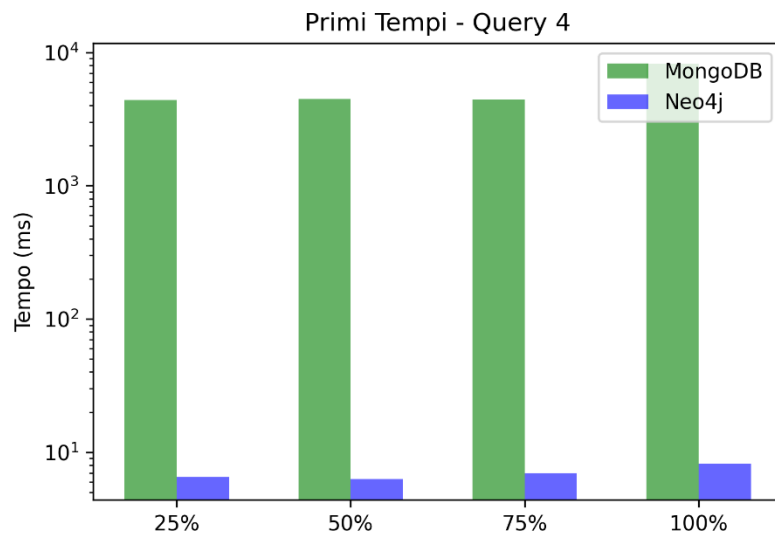


## Query 4



Analizza la relazione tra direttori di aziende, azionisti e conti bancari, calcolando il saldo totale detenuto dagli azionisti per ogni azienda e la percentuale di partecipazione di ciascun investitore.

```
1 MATCH (director:Person)-[:DIRECTS]->(company:Company)
2 MATCH (company)<-[:OWNS]-(shareholder:Person)
3 MATCH (shareholder)-[:HAS_ACCOUNT]->(account:Account)
4 WHERE account.balance > 0 AND account.acc_id IS NOT NULL
5 WITH
6     director.first_name AS DirectorFirstName,
7     company.company_name AS CompanyName,
8     SUM(account.balance) AS TotalAmount,
9     o.percentage AS HoldingPercentage
10 ORDER BY TotalAmount DESC
11 RETURN
12     DirectorFirstName,
13     CompanyName,
14     TotalAmount,
15     HoldingPercentage
16
```



## Conclusioni

Considerati i dataset riportati prima nella sezione “Implementazione e inserimento” e considerati tutti i test effettuati riportati nella sezione “esperimenti”, in un ambiente privo di elementi in memoria cache, si evidenzia una marcata differenza nei tempi di esecuzione, in particolare si evidenzia:

- Come neo4j risulti sempre in netto vantaggio in termini di esecuzione rispetto a mongoDB;
- Come Neo4j risulti più veloce anche in termini di esecuzione media delle query;

Nello specifico risulta che:

- Nel caso della prima query si evidenzia come neo4j sembra essere addirittura 10 volte più veloce rispetto a mongoDB;
- A livello di prime esecuzioni, si evidenzia come nel primo DBMS vi sia una netta differenza tra prima esecuzione e seconda, a differenza del secondo in cui la prima esecuzione sembra essere quasi alla pari con la seconda. In generale questo pattern di tempistiche si ripete per tutte le query in entrambi i DBMS.

I risultati ottenuti sono in linea con le caratteristiche dei due sistemi di gestione del database.

Neo4j è progettato per gestire dati di tipo grafico, in cui le relazioni tra i nodi sono al centro delle operazioni. Ciò lo rende altamente performante nell'analisi di connessioni complesse e nell'esecuzione di query che coinvolgono molteplici entità collegate.

MongoDB, invece, è un database NoSQL orientato ai documenti, che si adatta meglio alla gestione di dati semi-strutturati, ma risulta meno efficiente nelle operazioni che richiedono una gestione complessa delle relazioni tra i dati.

Di conseguenza, nel contesto del caso di studio, Neo4j si dimostra particolarmente più efficiente rispetto a MongoDB quando si tratta di elaborare e analizzare connessioni tra entità, soprattutto su dataset di grandi dimensioni.