Università degli studi di Messina
Dipartimento di Scienze Matematiche, Informatiche,
Scienze Fisiche e Scienze della Terra (MIFT)

Corso di Laurea Triennale in INFORMATICA

Progetto di Basi di Dati II (NoSQL)

A.A. 2021/2022

Professore : Antonio Celesti

Realizzato da: Sidoti Pipirito Calogero

1) PROBLEMATICA AFFRONTATA

Le frodi e-commerce stanno crescendo rapidamente, creando nuove sfide in termini di prevenzione e rilevamento. Negli ultimi anni, il mercato dell'e-commerce si è continuamente espanso, raggiungendo 1,9 trilioni di dollari di valore delle transazioni nel 2016 e le vendite all'interno di e-commerce sono ancora in rapida crescita. Nel frattempo, l'attività di frode dell'e-commerce è diventata un'industria multimiliardaria per i criminali. L'adozione di nuove tecnologie, metodi di pagamento e sistemi di elaborazione dei dati ha avvantaggiato anche i truffatori, aprendo nuove porte per aggirare le misure di sicurezza esistenti e coprire le loro tracce. Per effettuare analisi complesse, i fornitori di e-commerce hanno bisogno di tecnologie in grado di lavorare con dati cross-channel ed eseguire analisi di relazioni su larga scala. Tuttavia, la maggior parte delle soluzioni antifrode odierne si basano ancora su database relazionali, progettati per archiviare i dati in un formato tabulare, per cui il rilevamento di relazioni tra entità diventa computazionalmente intrattabile dopo alcuni hop. Per essere in grado di eseguire analisi complesse e rafforzare il sistema di rilevamento delle frodi, i commercianti di e-commerce si rivolgono alla tecnologia grafica. Questo approccio si basa su un modello di dati grafico in cui tutti i dati vengono archiviati come grafico. Le entità sono memorizzate come nodi, collegati tra loro da archi (che rappresentano le relazioni tra le entità). La tecnologia a grafo consente di raccogliere e connettere dati di clienti, transazioni, comportamenti in un modello di dati unico. Questo è essenziale per scoprire tentativi di frode. L'approccio grafico è ideale per eseguire query sui dati e le relazioni fra essi. Si possono eseguire query che attraversano set di dati di milioni di record per svelare connessioni sospette. Questo è fondamentale per rilevare reti e modelli sospetti in tempo reale.

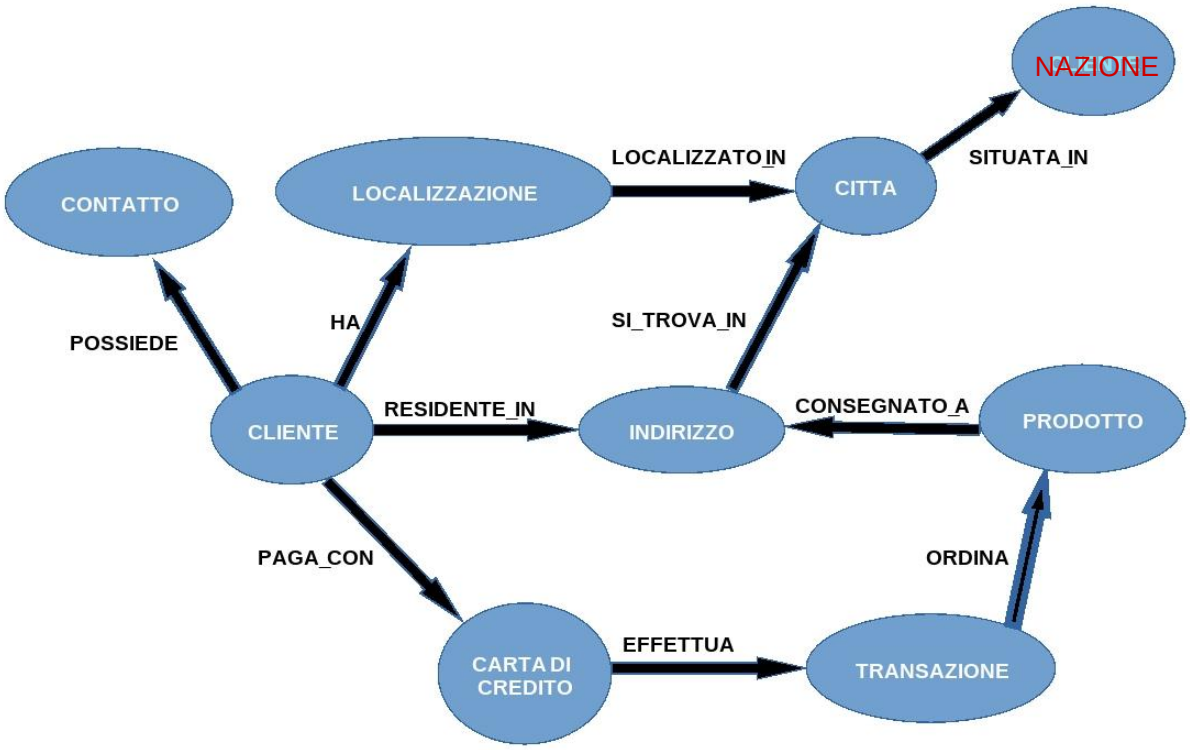
2) DATABASE

Eseguiremo l’analisi prestazionale utilizzando i due database “Neo4j” e “Cassandra”.

Neo4j è un database di tipo NoSql appartenente alla famiglia dei “Graph Database”. Ovviamente, come già detto, questo tipo di database a grafo si presta bene a trattare problematiche come quella in questione, data la sua spiccata capacità di estrarre informazioni, anche abbastanza complesse, da entità legate tra loro da varie relazioni. Infatti l'esplorazione di queste strutture risulta in genere più veloce rispetto a un database organizzato a tabelle perché la ricerca di nodi in relazione con un certo nodo è un'operazione primitiva e non richiede più passaggi. Neo4j supporta un linguaggio per effettuare le query chiamato “Cypher”, un linguaggio dichiarativo ispirato a SQL.

Cassandra è un database di tipo NoSql facente parte della famiglia dei database “Column oriented”. Il database vero e proprio prende il nome di “keyspace” ed è organizzato in Column family. Possiede un datamodel altamente flessibile, il cui funzionamento si basa appunto su colonne con comportamenti dinamici per ogni riga. Inoltre, non è presente il concetto di join, tipico dei database relazionali. Cassandra supporta un linguaggio per effettuare le query chiamato CQL (Cassandra Query Language) molto simile a SQL.

3) PROGETTAZIONE

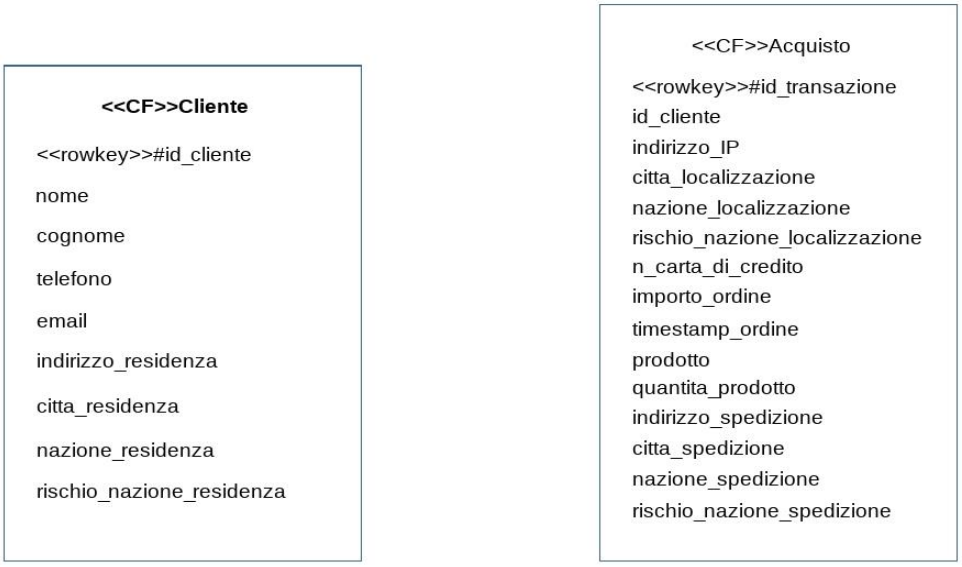


Per quanto riguarda Neo4j:

- l'entità "CLIENTE" avrà un "ID_cliente" per identificare univocamente ogni singolo cliente, un "Nome" e un "Cognome";
- l'entità "CONTATTO" avrà i campi "Telefono" e "eMail";
- l'entità "LOCALIZZAZIONE" avrà "Indirizzo_IP" che indica l'indirizzo IP col quale è stata effettuata la transazione;

- l’entità “CARTA DI CREDITO” avrà “N_carta_di_credito” il cui valore rappresenta il numero della carta di credito con la quale è stato effettuato un pagamento;
- l’entità “TRANSAZIONE” avrà i campi “ID_transazione” per identificare univocamente ogni singola transazione, “Importo_ordine” che indica la somma spesa per l’acquisto, “Timestamp_ordine” che indica quando l’acquisto è stato effettuato;
- l’entità “INDIRIZZO” avrà il campo “Indirizzo” indicante o l’indirizzo di residenza del cliente o l’indirizzo di spedizione per l’ordine effettuato;
- l’entità “PRODOTTO” avrà il campo “Prodotto” per indicare il nome del prodotto acquistato e il campo “Quantita” che indica la quantità acquistata di quel prodotto;
- l’entità “CITTA” avrà il campo “Citta”;
- l’entità “NAZIONE” avrà il campo “Nazione” indicante il nome della nazione e il campo “A_rischio” il cui valore booleano indica se la nazione in questione è nell’elenco delle nazioni considerate a rischio di frode.

Per la progettazione del datamodel di Cassandra abbiamo proceduto nel modo seguente:



Come si può osservare, sono state create due column family, una contenente praticamente solo dati anagrafici del cliente e l'altra contenente tutte le informazioni che interessano ogni singola transazione. Questa scelta è stata presa basandoci sul fatto che conviene raggruppare nella stessa column family le colonne alle quali spesso si accede contemporaneamente, dato che ogni singola column family viene salvata su un file separato.

4) IMPLEMENTAZIONE

Utilizzeremo il linguaggio di programmazione “Python” per interagire con i due database per svolgere le operazioni di inserimento dati e di esecuzione delle query. Gli esperimenti saranno automatizzati.

Produrremo dataset di diverse dimensioni (10.000, 20.000, 30.000, 40.000) appositamente per poter testare le query con dataset di dimensione differente.

N.B. Per semplicità, le descrizioni seguenti faranno riferimento al dataset di dimensioni più piccole (10.000), ma ovviamente gli stessi concetti varranno per i dataset di dimensioni maggiori.

Per prima cosa ci serve un dataset sul quale effettuare le nostre analisi prestazionali. A questo scopo abbiamo creato un file “csv” utilizzando dati fittizzi creati mediante la libreria “Faker” di Python.

A titolo esplicativo, riportiamo, di seguito, il codice sorgente per la creazione del dataset contenente 10.000 record.

```
import csv
from faker import Faker
import random

fake = Faker(['en-US'])

nazione_array = []
a_rischio_array = []

id_cliente = []
nome = []
cognome = []
tel = []
email = []
ind_res = []
citta_res = []
naz_res = []
rischio_naz_res = []


with open('countries001.csv','r') as f:
    data = csv.reader(f)

    for row in data:
        nazione_array.append(row[0])
        a_rischio_array.append(row[1])
```

```
with open('clienti2.csv','r') as f:
    data = csv.reader(f)

    for row in data:
        id_cliente.append(row[0])
        nome.append(row[1])
        cognome.append(row[2])
        tel.append(row[3])
        email.append(row[4])
        ind_res.append(row[5])
        citta_res.append(row[6])
        naz_res.append(row[7])
        rischio_naz_res.append(row[8])


with open('dataset10k.csv', mode='w', newline='') as csv_file:
    fieldnames = ['ID_TRANSAZIONE', 'ID_CLIENTE', 'NOME', 'COGNOME', 'TELEFONO',
'EMAIL',
'INDIRIZZO_RESIDENZA',
                  'CITTA_RESIDENZA', 'NAZIONE_RESIDENZA',
'RISCHIO_NAZIONE_RESIDENZA', 'INDIRIZZO_IP',
'CITTA_LOCALIZZAZIONE', 'NAZIONE_LOCALIZZAZIONE',
'RISCHIO_NAZIONE_LOCALIZZAZIONE', 'N_CARTA_DI_CREDITO',
                 'IMPORTO_ORDINE', 'TIMESTAMP_ORDINE', 'PRODOTTO', 'QUANTITA',
'INDIRIZZO_SPEDIZIONE', 'CITTA_SPEDIZIONE',
                 'NAZIONE_SPEDIZIONE', 'RISCHIO_NAZIONE_SPEDIZIONE']

    writer = csv.DictWriter(csv_file, fieldnames = fieldnames)

    writer.writeheader()
    id = 1

    for x in range(10000):
            rand = random.randint(1,239)
            rand1 = random.randint(1,499)
            rand2 = random.randint(1,239)

            ID_cliente = id_cliente[rand1]
            nome_cliente = nome[rand1]
            cognome_cliente = cognome[rand1]
            telefono_cliente = tel[rand1]
            email_cliente = email[rand1]
            indirizzo_residenza = ind_res[rand1]
            citta_residenza = citta_res[rand1]
            nazione_residenza = naz_res[rand1]
            rischio_nazione_residenza = rischio_naz_res[rand1]
            ip = fake.ipv4()
            citta_localizzazione = fake.city()
            nazione_localizzazione = nazione_array[rand]
            rischio_nazione_localizzazione = a_rischio_array[rand]
            n_carta = random.randint(1111111111, 2111111111)
            importo = random.randint(1, 2000)
            timestamp = fake.date_time_between(start_date = '-3y', end_date =
'now')
            prodotto = fake.catch_phrase()
            quantita = random.randint(1,50)
            indirizzo_spedizione = fake.address()
            citta_spedizione = fake.city()
            nazione_spedizione = nazione_array[rand2]
            rischio_nazione_spedizione = a_rischio_array[rand2]
```

```
            writer.writerow({'ID_TRANSAZIONE' :id, 'ID_CLIENTE' : ID_cliente,
'NOME' : nome_cliente,
'COGNOME' : cognome_cliente,
                        'TELEFONO' : telefono_cliente, 'EMAIL' : email_cliente,
'INDIRIZZO_RESIDENZA' : indirizzo_residenza,
'CITTA_RESIDENZA' : citta_residenza,
                        'NAZIONE_RESIDENZA' : nazione_residenza,
'RISCHIO_NAZIONE_RESIDENZA' : rischio_nazione_residenza,
'INDIRIZZO_IP' : ip, 'CITTA_LOCALIZZAZIONE' : citta_localizzazione,
'NAZIONE_LOCALIZZAZIONE' : nazione_localizzazione,
'RISCHIO_NAZIONE_LOCALIZZAZIONE' : rischio_nazione_localizzazione,
'N_CARTA_DI_CREDITO' : n_carta,
                        'IMPORTO_ORDINE' : importo, 'TIMESTAMP_ORDINE' :
timestamp,
'PRODOTTO' : prodotto, 'QUANTITA' : quantita, 'INDIRIZZO_SPEDIZIONE' :
indirizzo_spedizione,
'CITTA_SPEDIZIONE' : citta_spedizione, 'NAZIONE_SPEDIZIONE' :
nazione_spedizione,
                              'RISCHIO_NAZIONE_SPEDIZIONE' :
rischio_nazione_spedizione})
            id+=1
```

Come possiamo vedere dal codice sorgente, sono stati letti due file .csv precedentemente elaborati per facilitare la creazione del csv finale.

Una volta creato il file csv contenente il dataset possiamo procedere ad importarlo nei due database.

Neo4j

Connessione

Neo4j fornisce driver ufficiali per vari linguaggi di programmazione incluso Python. Tramite il comando “session” utilizzeremo uno stile a blocchi "classico" per l'esecuzione di Cypher. In generale, le sessioni forniscono uno stile di programmazione più semplice con cui lavorare poiché le chiamate API vengono eseguite in modo strettamente sequenziale.

```
from neo4j import GraphDatabase

uri = "neo4j://localhost:7687"
user = "neo4j"
psw = "admin"

driver = GraphDatabase.driver(uri, auth=(user, psw))
session = driver.session()
```

Importazione

Abbiamo provveduto prima ad importare le entità e successivamente a creare le relazioni tra di esse.

Di seguito un esempio di importazione di un’entità in Neo4j:

```
session.run("CREATE CONSTRAINT ON (c:CLIENTE) ASSERT c.ID_cliente IS UNIQUE;")
session.run("""LOAD CSV WITH HEADERS FROM "file:///dataset10k.csv" AS row
            MERGE (c:CLIENTE{ID_cliente : toInteger(row.ID_CLIENTE),
            Nome : row.NOME, Cognome : row.COGNOME});
            """)
```

Qui, oltre all’importazione dell’entità, abbiamo imposto un vincolo interno per il nodo in questione.

Di seguito un esempio di creazioni di relazione tra due entità:

```
session.run("""LOAD CSV WITH HEADERS FROM "file:///dataset10k.csv" AS row
            MATCH (cl:CLIENTE{ID_cliente : toInteger(row.ID_CLIENTE)})
            MATCH (co:CONTATTO{Telefono : toInteger(row.TELEFONO), eMail :
                  row.EMAIL})
            MERGE (cl)-[r:POSSIEDE]->(co);
            """)
```

Query

1. ```
   MATCH (c:CLIENTE)
   WHERE c.ID_cliente > 280
   RETURN c.Nome, c.Cognome
   ```

2. ```
   MATCH (c:CLIENTE{Nome : 'Eric'})
   RETURN c.ID_cliente, c.Cognome
   ```

3. ```
   MATCH (c:CLIENTE)-[r:PAGA_CON]→(n:CARTA_DI_CREDITO)
   WHERE c.ID_cliente > 250 AND c.ID_cliente < 5000
   RETURN c.ID_cliente, n
   ```

4. ```
   MATCH (t:TRANSAZIONE)-[r1:ORDINA]→(p:PRODOTTO)-
    [r2:CONSEGNATO_A]→(i:INDIRIZZO)-[r3:SI_TROVA_IN]→
   (ci:CITTA {Citta : 'Boston'})
   WHERE t.ID_transazione > 5000 AND t.ID_transazione < 35000
   RETURN t.ID_transazione, p.Prodotto
   ```

5. ```
   MATCH (c:CLIENTE)-[r1:PAGA_CON]->(n:CARTA_DI_CREDITO)-
   [r2:EFFETTUA]->(t:TRANSAZIONE)-[r3:ORDINA]->(p:PRODOTTO)-
   [r4:CONSEGNATO_A]->(i:INDIRIZZO)-[r5:SI_TROVA_IN]->(c2:CITTA)-
   [r6:SITUATA_IN]->(nz:NAZIONE{A_rischio : true})
   WHERE t.Importo_ordine < 100 AND p.Quantita < 50
   RETURN c.ID_cliente, t.ID_transazione, t.Timestamp_ordine
   ```

Cassandra

Connessione

Cassandra mette a disposizione i suoi driver anche per i programmatori che utilizzano Python, con i quali creare un Cluster e connettersi a un’istanza di Cassandra.
   ```

```
import csv
from cassandra.cluster import Cluster

KEYSPACE = "cassandra10k"

cluster = Cluster(['127.0.0.1'])
session = cluster.connect()
```

Importazione

Di seguito, la creazione del keyspace, i suoi settaggi, il datamodel, iniziale, perché come sappiamo possono essere create nuove colonne dinamicamente dopo la creazione delle varie column family e infine l'importazione del dataset.

```
session.execute("""
    CREATE KEYSPACE IF NOT EXISTS %s
    WITH replication = {'class': 'SimpleStrategy', 'replication_factor': '1'}
    """ % KEYSPACE)

session.set_keyspace(KEYSPACE)


session.execute("""
    CREATE TABLE IF NOT EXISTS Cliente (
        id_cliente int PRIMARY KEY,
        nome varchar,
        cognome varchar,
        telefono int,
        email varchar,
        indirizzo_residenza varchar,
        citta_residenza varchar,
        nazione_residenza varchar,
        rischio_nazione_residenza boolean);
    """)


session.execute("""
    CREATE TABLE IF NOT EXISTS Acquisto (
        id_transazione int PRIMARY KEY,
        id_cliente int,
        indirizzo_IP varchar,
        citta_localizzazione varchar,
```

```
        nazione_localizzazione varchar,
        rischio_nazione_localizzazione boolean,
        n_carta_di_credito int,
        importo_ordine int,
        timestamp_ordine varchar,
        prodotto varchar,
        quantita_prodotto int,
        indirizzo_spedizione varchar,
        citta_spedizione varchar,
        nazione_spedizione varchar,
        rischio_nazione_spedizione boolean);
    """)



prepared1 = session.prepare("""
        INSERT INTO Cliente (id_cliente, nome, cognome, telefono, email,
indirizzo_residenza, citta_residenza,
            nazione_residenza, rischio_nazione_residenza)
        VALUES (?, ?, ?, ?, ?, ?, ?, ?, ?)
        """)

prepared2 = session.prepare("""
        INSERT INTO Acquisto (id_transazione, id_cliente, indirizzo_IP,
citta_localizzazione,
            nazione_localizzazione, rischio_nazione_localizzazione,
n_carta_di_credito, importo_ordine,
            timestamp_ordine, prodotto, quantita_prodotto, indirizzo_spedizione,
citta_spedizione,
            nazione_spedizione, rischio_nazione_spedizione)
        VALUES (?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?)
        """)


with open('dataset10k.csv','r') as f_csv:
    data = csv.reader(f_csv)
    header = next(data)

    for row in data:
         ID_cliente = int(row[1])
         nome_cliente = row[2]
         cognome_cliente = row[3]
         telefono_cliente = int(row[4])
         email_cliente = row[5]
         indirizzo_residenza = row[6]
         citta_residenza = row[7]
         nazione_residenza = row[8]
         rischio_nazione_residenza = bool(row[9])

         session.execute(prepared1, [ID_cliente, nome_cliente, cognome_cliente,
telefono_cliente,
                                     email_cliente, indirizzo_residenza,
citta_residenza, nazione_residenza,
                                     rischio_nazione_residenza])

f_csv.close()

with open('dataset10k.csv','r') as f_csv:
    data = csv.reader(f_csv)
    header = next(data)

    for row1 in data:
         ID_transazione = int(row1[0])
         ID_cliente = int(row1[1])
```

```
        indirizzo_IP = row1[10]
        citta_localizzazione = row1[11]
        nazione_localizzazione = row1[12]
        rischio_nazione_localizzazione = bool(row1[13])
        n_carta_di_credito = int(row1[14])
        importo_ordine = int(row1[15])
        timestamp_ordine = row1[16]
        prodotto = row1[17]
        quantita_prodotto = int(row1[18])
        indirizzo_spedizione = row1[19]
        citta_spedizione = row1[20]
        nazione_spedizione = row1[21]
        rischio_nazione_spedizione = bool(row1[22])


        session.execute(prepared2, [ID_transazione, ID_cliente, indirizzo_IP,
citta_localizzazione,
                                    nazione_localizzazione,
rischio_nazione_localizzazione, n_carta_di_credito,
                                    importo_ordine, timestamp_ordine, prodotto,
quantita_prodotto, indirizzo_spedizione,
                                    citta_spedizione, nazione_spedizione,
rischio_nazione_spedizione])

f_csv.close()

session.shutdown()
```

Query

1.
```
SELECT nome,cognome
FROM cassandra10k.cliente
WHERE id_cliente > 280
```

2.
```
SELECT id_cliente,cognome
FROM cassandra10k.cliente
WHERE nome = 'Eric'
```

3.
```
SELECT id_cliente, n_carta_di_credito
FROM cassandra10k.acquisto
WHERE id_cliente > 250 AND id_cliente < 5000
```

4.
```
SELECT id_transazione, prodotto
FROM cassandra10k.acquisto
WHERE citta_spedizione = 'Boston'
AND id_transazione > 5000 AND id_transazione < 35000
```

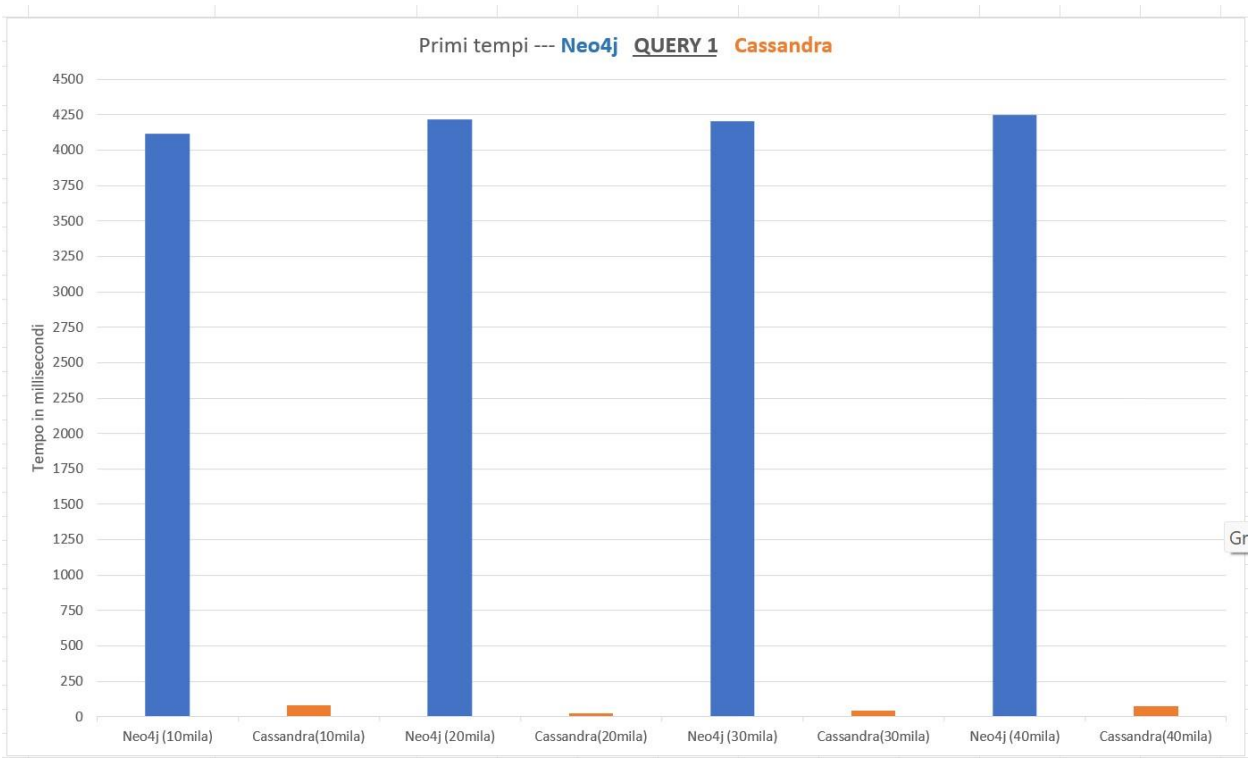
5.
```
SELECT id_cliente, id_transazione, timestamp_ordine
FROM cassandra10k.acquisto
WHERE rischio_nazione_spedizione = true
AND importo_ordine < 100 AND quantita_prodotto < 50
```

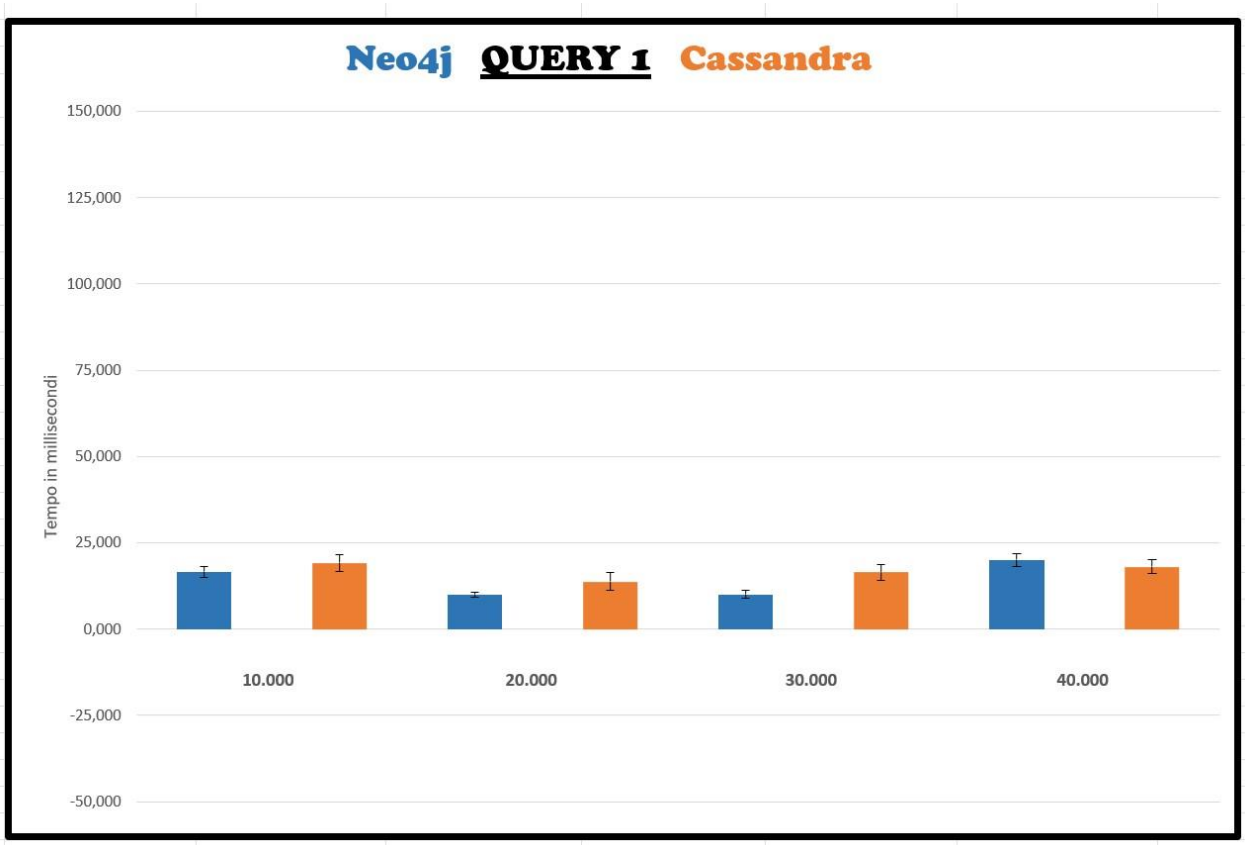
5) ESPERIMENTI

Per effettuare gli esperimenti, come già visto sopra, sono state implementate 5 query con grado di complessità crescente. Di seguito, per ogni query, sono riportate le tabelle dove sono rappresentati i tempi di esecuzione della singola query per 31 volte, escludendo il primo tentativo, la media dei risultati, la deviazione standard e l'intervallo di confidenza calcolato al 95%. E' presente anche un istogramma che mostra la latenza media di esecuzione con la rappresentazione dell'intervallo di confidenza entro il quale i risultati vengono considerati accettabili, e un altro rappresentante i primi tempi di esecuzione.

Query 1

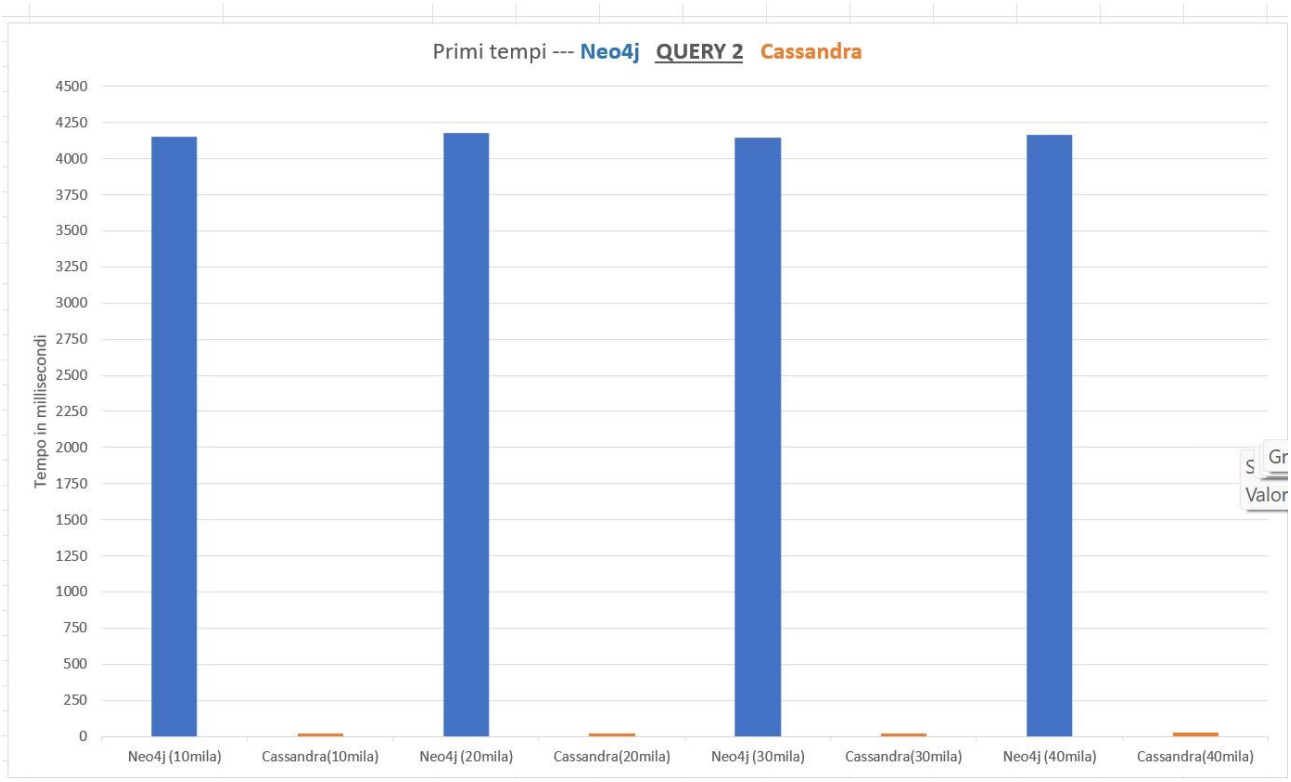
QUERY 1	Neo4j (10mila)	Cassandra(10mila)	Neo4j (20mila)	Cassandra(20mila)	Neo4j (30mila)	Cassandra(30mila)	Neo4j (40mila)	Cassandra(40mila)
Primi tempi	4120	79	4217	25	4206	42	4247	76
MEDIA	16,567	19,167	9,933	13,767	10,000	16,400	19,967	18,067
Dev.Standard	4,516	7,027	1,929	7,190	3,063	6,516	5,411	5,753
95% Conf	1,616	2,515	0,690	2,573	1,096	2,332	1,936	2,059

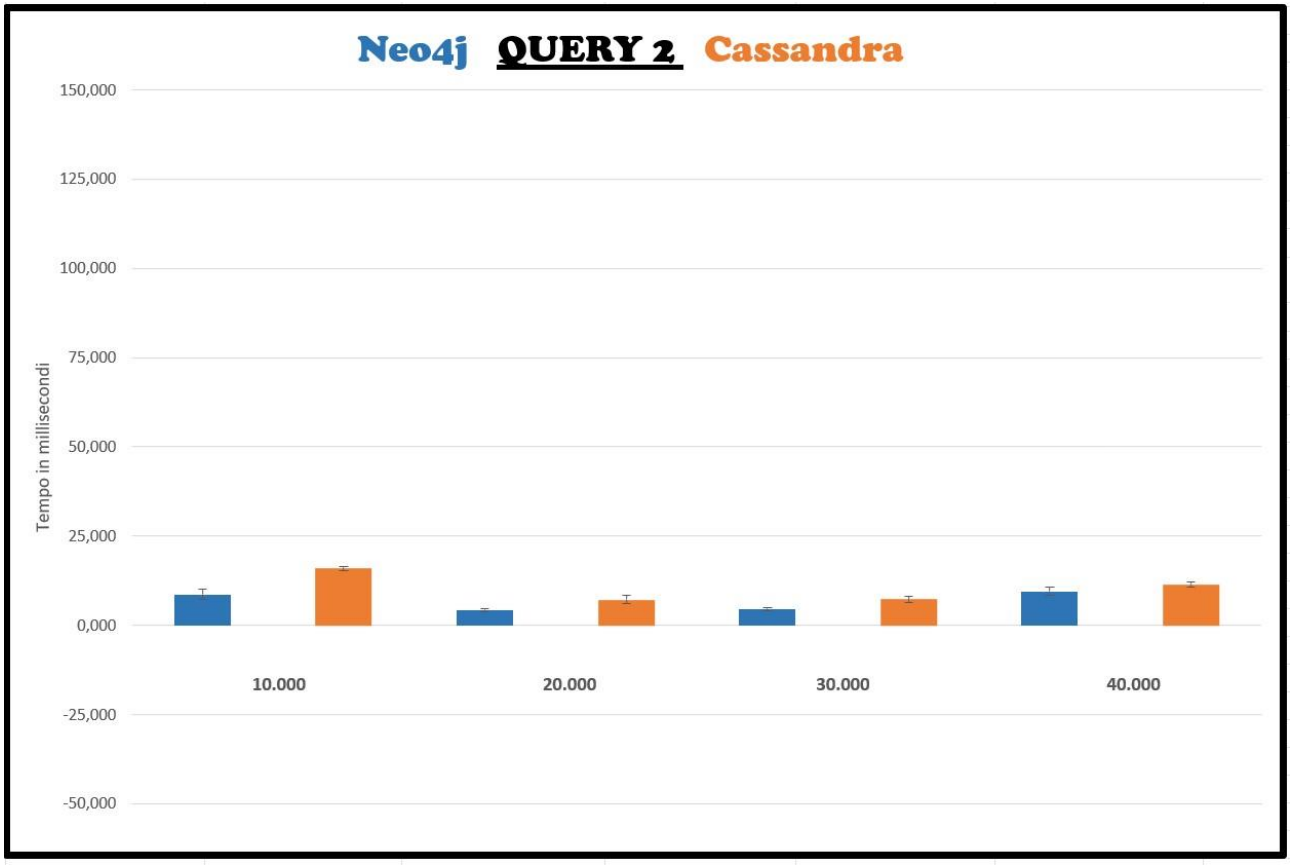




Query 2

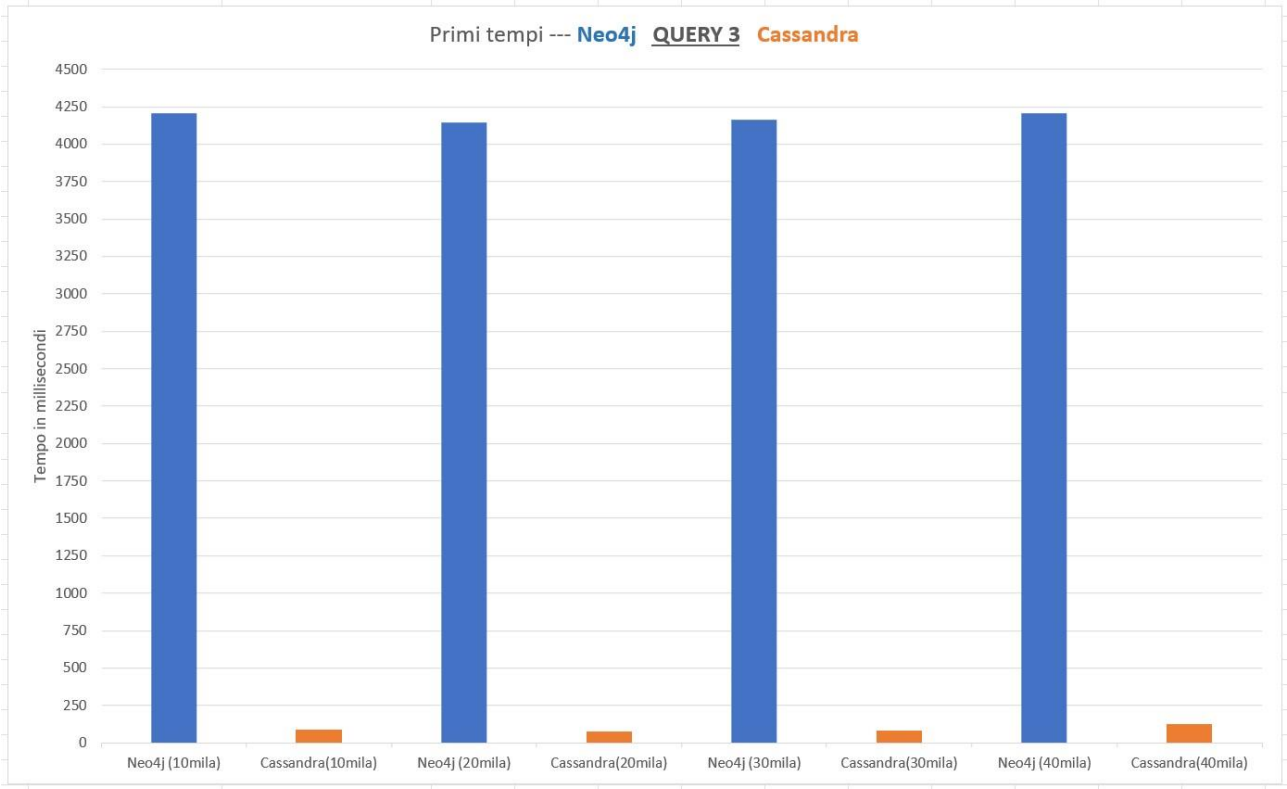
QUERY 2	Neo4j (10mila)	Cassandra(10mila)	Neo4j (20mila)	Cassandra(20mila)	Neo4j (30mila)	Cassandra(30mila)	Neo4j (40mila)	Cassandra(40mila)
Primi tempi	4153	22	4178	19	4146	19	4162	29
MEDIA	8,700	16,033	4,233	7,233	4,533	7,300	9,533	11,500
Dev.Standard	4,027	1,520	1,431	3,036	1,358	2,548	3,224	2,316
95% Conf	1,441	0,544	0,512	1,087	0,486	0,912	1,154	0,829

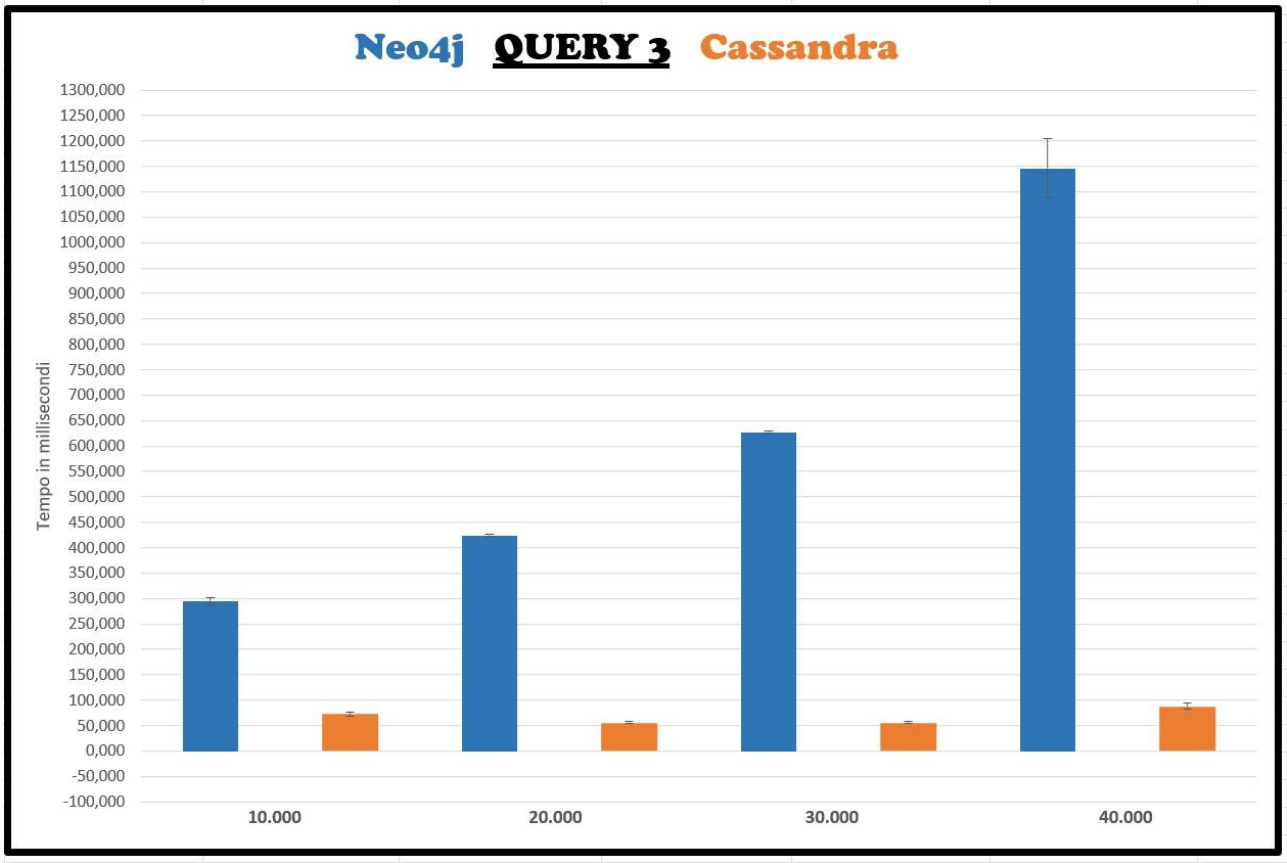

Primi tempi --- Neo4j QUERY 2 Cassandra
Tempo in millisecondi
4500
4250
4000
3750
3500
3250
3000
2750
2500
2250
2000
1750
1500
1250
1000
750
500
250
0
Neo4j (10mila)
Cassandra(10mila)
Neo4j (20mila)
Cassandra(20mila)
Neo4j (30mila)
Cassandra(30mila)
Neo4j (40mila)
Cassandra(40mila)


Neo4j QUERY 2 Cassandra
Tempo in millisecondi
150,000
125,000
100,000
75,000
50,000
25,000
0,000
-25,000
-50,000
10.000
20.000
30.000
40.000

Query 3

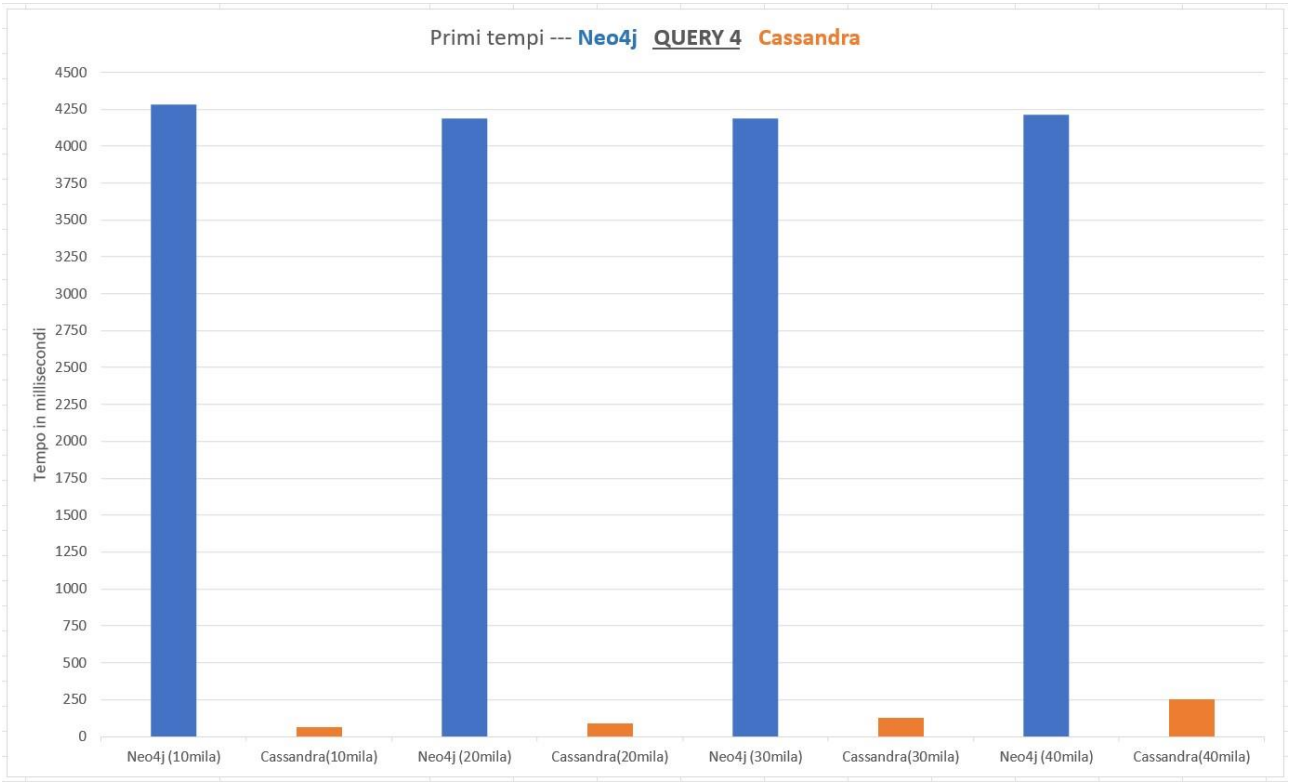
QUERY 3	Neo4j (10mila)	Cassandra(10mila)	Neo4j (20mila)	Cassandra(20mila)	Neo4j (30mila)	Cassandra(30mila)	Neo4j (40mila)	Cassandra(40mila)
Primi tempi	4206	89	4145	77	4165	85	4204	123
MEDIA	294,600	72,700	424,167	55,833	627,533	55,900	1145,767	88,000
Dev.Standard	20,576	10,229	6,320	5,902	6,796	4,894	162,770	16,826
95% Conf	7,363	3,660	2,261	2,112	2,432	1,751	58,245	6,021

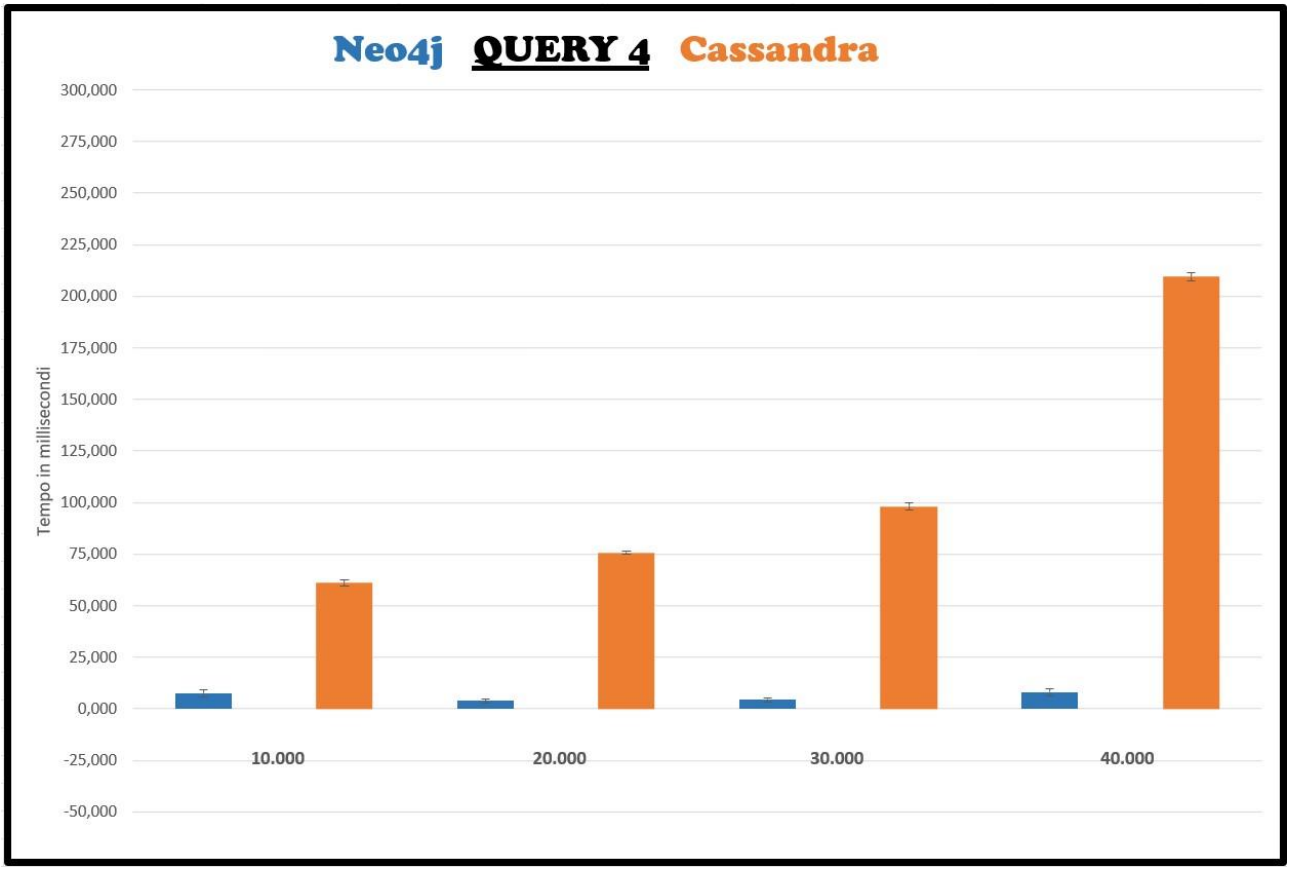




Query 4

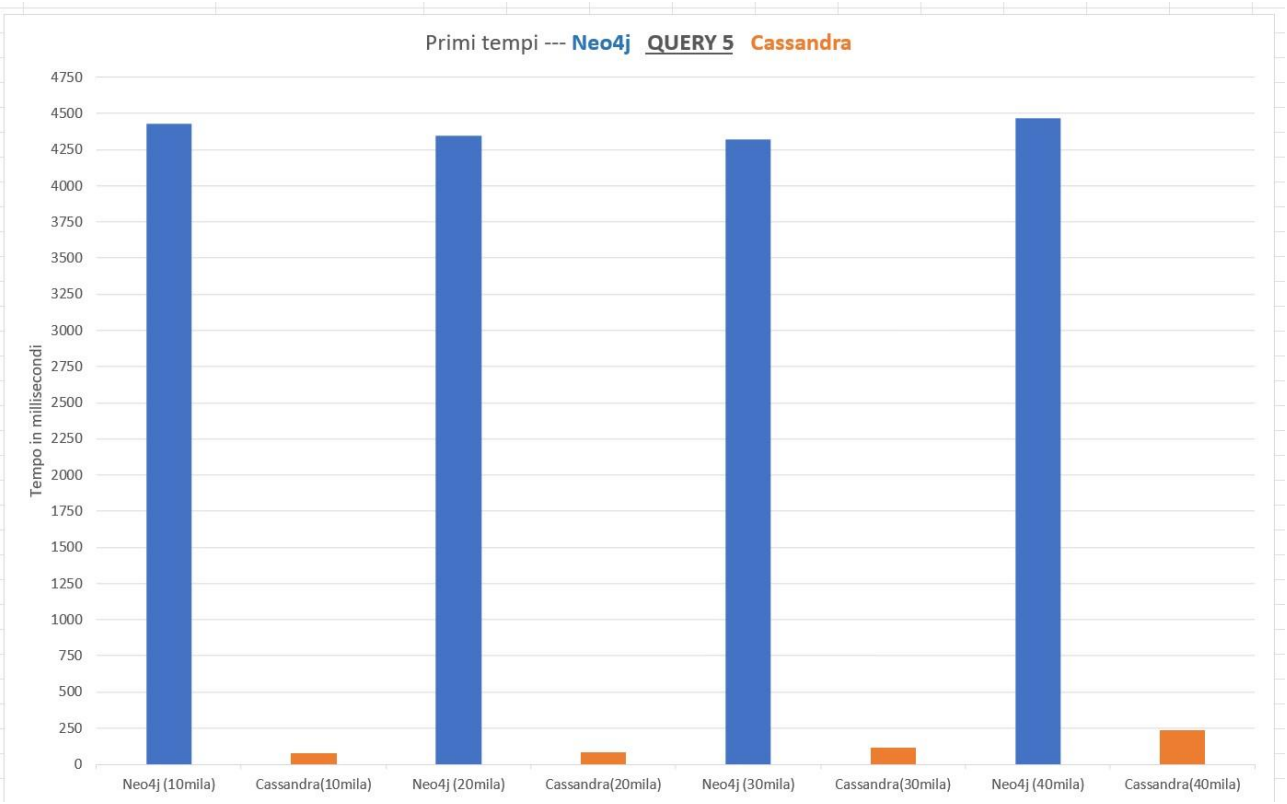
QUERY 4	Neo4j (10mila)	Cassandra(10mila)	Neo4j (20mila)	Cassandra(20mila)	Neo4j (30mila)	Cassandra(30mila)	Neo4j (40mila)	Cassandra(40mila)
Primi tempi	4281	64	4190	91	4190	125	4216	254
MEDIA	7,600	61,100	3,800	75,633	4,433	98,167	8,000	209,367
Dev.Standard	4,966	4,213	2,355	1,847	2,622	4,921	4,814	5,183
95% Conf	1,777	1,508	0,843	0,661	0,938	1,761	1,723	1,855

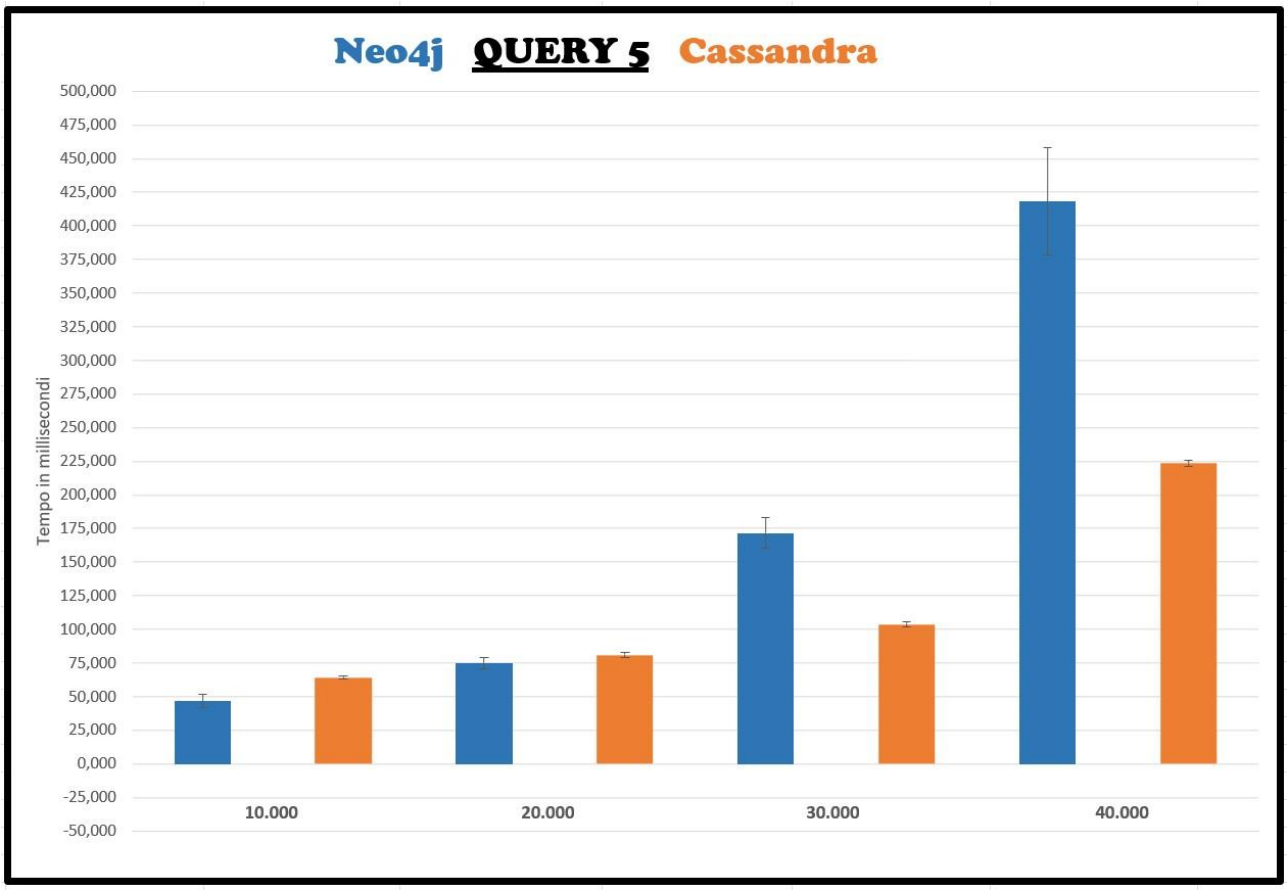

Primi tempi --- Neo4j QUERY 4 Cassandra
Tempo in millisecondi
4500
4250
4000
3750
3500
3250
3000
2750
2500
2250
2000
1750
1500
1250
1000
750
500
250
0
Neo4j (10mila)
Cassandra(10mila)
Neo4j (20mila)
Cassandra(20mila)
Neo4j (30mila)
Cassandra(30mila)
Neo4j (40mila)
Cassandra(40mila)


Neo4j QUERY 4 Cassandra
Tempo in millisecondi
300,000
275,000
250,000
225,000
200,000
175,000
150,000
125,000
100,000
75,000
50,000
25,000
0,000
-25,000
-50,000
10.000
20.000
30.000
40.000

Query 5

QUERY 5	Neo4j (10mila)	Cassandra(10mila)	Neo4j (20mila)	Cassandra(20mila)	Neo4j (30mila)	Cassandra(30mila)	Neo4j (40mila)	Cassandra(40mila)
Primi tempi	4428	74	4344	85	4322	114	4465	235
MEDIA	46,500	64,300	74,767	80,767	171,700	103,633	418,100	223,533
Dev.Standard	13,945	3,807	11,407	5,817	31,795	4,810	111,659	7,065
95% Conf	4,990	1,362	4,082	2,082	11,377	1,721	39,956	2,528





6) CONCLUSIONI

Fintanto che le query utilizzate presentano un grado di complessità relativamente minimo, si ottengono prestazioni simili tra i database Neo4j e Cassandra anche incrementando la dimensione del dataset. Quando però le query diventano più complesse, con l'aumentare delle dimensioni del dataset si ha un decremento delle prestazioni soprattutto per quanto riguarda Neo4j; infatti tendenzialmente Cassandra mostra prestazioni migliori della sua controparte con l'aumentare della dimensione del dataset.

Cassandra è ideale quando si ha a che fare con dataset di grandi dimensioni ma la cui complessità generale è bassa e nel caso vi siano query semplici. Viceversa, Neo4j è più indicato per database con numerosi relazioni che richiedono query complesse.