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

CORSO DI LAUREA TRIENNALE IN INFORMATICA (L-31)

Progetto di Basi di Dati II

Benchmark tra due DBMS: Neo4J e Cassandra

Caso di studio:

INSURANCE FRAUD INVESTIGATION

A.A 2021/2022

Docente: Antonio Celesti

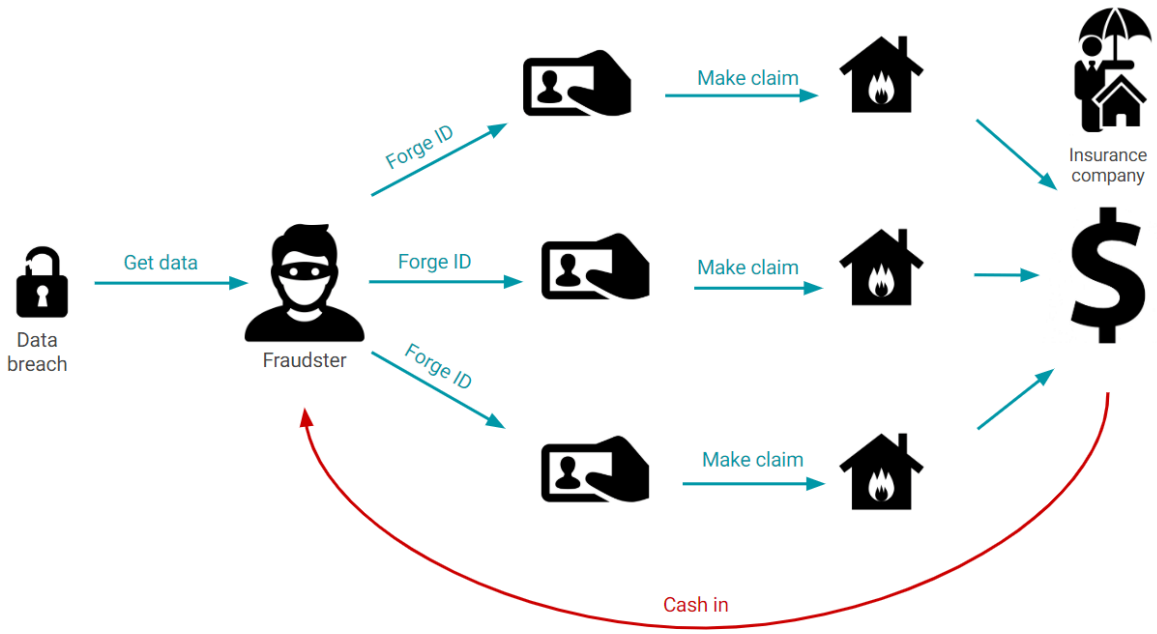
Studenti: Gianfranco Iaria
Giovanni Domenico Tripodi

1. INTRODUZIONE

Per frode assicurativa, si intende che chiunque, al fine di conseguire per sé o per altri l'indennizzo di un'assicurazione o comunque un vantaggio derivante da un contratto di assicurazione, distrugge, disperde, deteriora od occulta cose di sua proprietà, falsifica o altera una polizza o la documentazione richiesta per la stipulazione di un contratto di assicurazione. La frode assicurativa è raramente opera di un individuo isolato. I truffatori di solito formano reti complesse che sono difficili da individuare per le istituzioni assicurative e finanziarie. Una delle tecniche più comuni utilizzate dai truffatori è la seguente:

• Falsificare identità false,
• Presentare diversi reclami,
• Incassare gli assegni assicurativi.

La creazione di identità false, richiede di falsificare informazioni personali, come: indirizzi, carte di credito, ed altri documenti, da presentare alle compagnie assicurative per diventare clienti. Forgiare nuove informazioni per ogni identità falsa che creano ha un costo elevato per i truffatori. Questo è il motivo per cui i truffatori spesso riciclano questi dati per creare diverse identità false.



Studieremo questa problematica su due diversi database:
- Neo4j
- Cassandra

- Neo4j:

E’ un database NoSql appartenente alla famiglia dei “Graph Database”. I Graph Database, sfruttano dei grafi per la memorizzazione dei dati, e degli strumenti per lo studio di essi, con la possibilità di implementare query molto complesse per l’individuazione di nodi e archi.

Andando più nello specifico, il grafo che è possibile creare è costituito da nodi, archi e proprietà. Ogni nodo ha delle proprietà, che rappresentano i nostri record e dati da memorizzare.

Gli archi invece, hanno una direzione e rappresentano le relazioni trai i nodi. Neo4j ha sviluppato un query language per essere il più intuitivo possibile, chiamato Cypher, il quale è ottimizzato per lo studio dei grafi. Ad oggi viene usato anche in altri database.

- Cassandra:

Apache Cassandra è un database opensource NoSQL ottimizzato per la gestione di grandi quantità di dati dislocati in diversi server, fornendo un servizio orientato alla disponibilità. Inizialmente, Cassandra fu sviluppata da Facebook per potenziare la ricerca all'interno del sistema di posta.

Essa fornisce una struttura di memorizzazione chiave-valore. Alle chiavi corrispondono dei valori, raggruppati in famglie di colonne: una famiglia di colonne è definita quando il database viene creato. Tuttavia le colonne possono essere aggiunte a una famiglia in qualsiasi momento.

Le colonne sono aggiunte solo specificando le chiavi, così differenti chiavi possono avere differenti numeri di colonne in una data famiglia. I valori di una famiglia di colonne sono memorizzati insieme, in quanto Cassandra adotta un approccio ibrido tra DBMS orientato alle colonne e la memorizzazione orientata alle righe.

Cassandra presenta delle caratteristiche:

• È decentralizzato, i nodi del cluster sono identici e non esiste nessun single point of failure

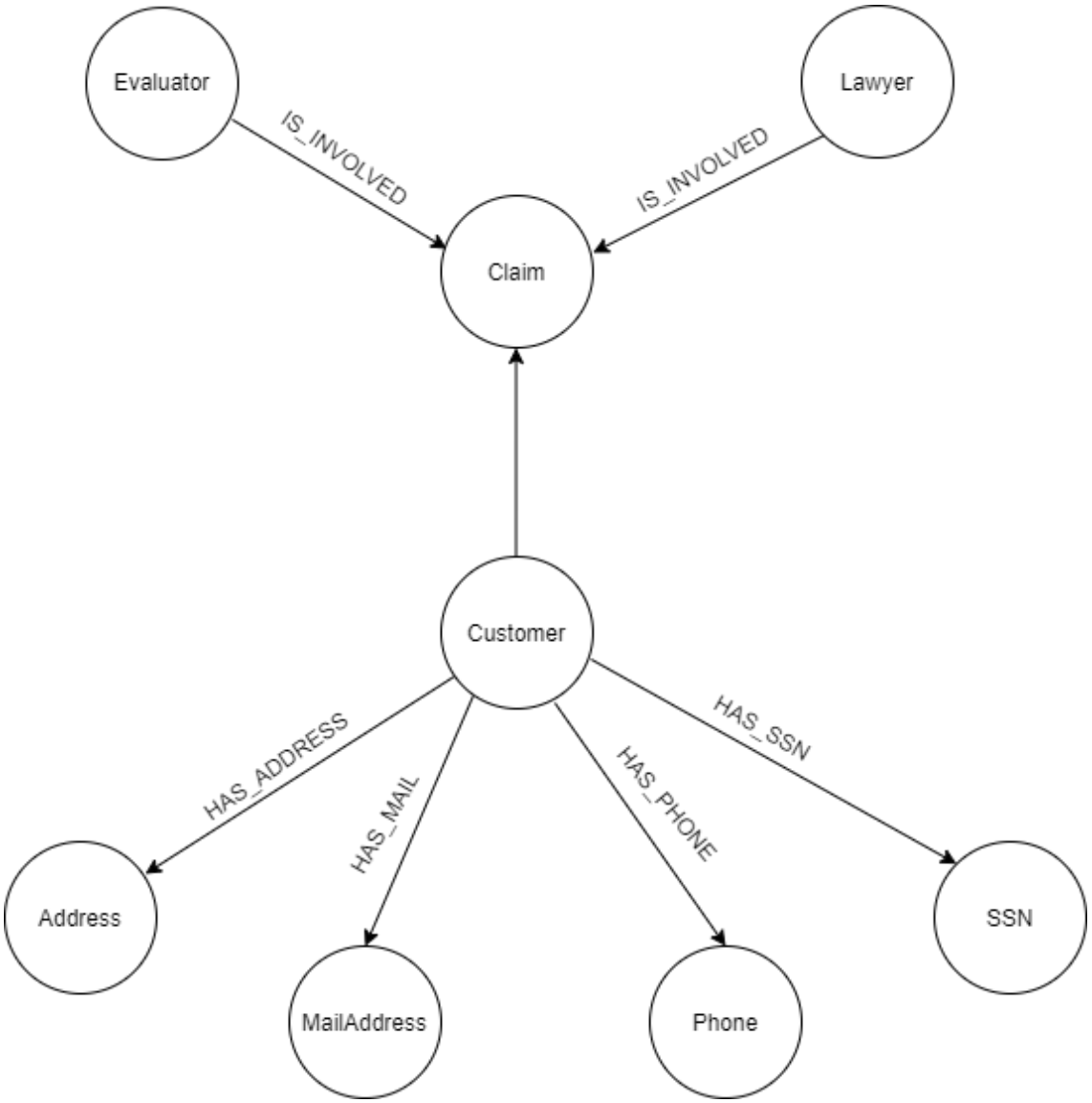
• Fault-tolerance: i dati vengono replicati automaticamente su più nodi. È supportata la replica mediante diversi data center, e la sostituzione dei nodi può essere effettuata senza alcun downtime

• Tunable consistency: il livello di coerenza (sia in scrittura che in lettura) può essere modificato (ad esempio da writes never fail a block for all replicas to be readable).

• Elasticità: il throughput di lettura o scrittura scala linearmente con l'aggiunta di nuove macchine (nodi), senza downtime e senza interruzione di alcun applicativo.

2. PROGETTAZIONE

Il grafo che andremo a considerare, si baserà sulle relazioni tra il cliente, con i rispettivi dati e il reclamo dovuto alla frode, che verrà esaminato dal valutatore e gestito dal giudice, il quale andrà ad eseguire il processo.



L'entità **Customer**, possiede:
- "**Address**" e "**MailAddress**", che corrispondono rispettivamente all'indirizzo personale e alla mail di posta elettronica, rappresentati entrambi da una stringa.
- "**Phone**", che corrisponde al numero di telefono, rappresentato da un numero intero.
- "**SSN**", che corrisponde al **social security number**, univoco per tutti i cittadini, rappresentato da tre coppie di numeri interi.

Per ogni "**Customer**", verrà identificato un **claim** (reclamo), che coinvolge un **Lawyer** e un **Evaluator**

3. DATABASE:

Per creare il file “customer.csv”, che comprende tutti i clienti, abbiamo utilizzato il linguaggio di programmazione “Python 3.9” e la sua libreria “Faker”, in modo da generare: **nomi, indirizzi personali, email, numeri di telefono, codici ssn e reclami** (nel caso in cui ci fossero) dei clienti in modo casuale. Se il cliente non ha alcun reclamo, la stringa sarà vuota. Quando il reclamo è presente, esso sarà collegato ad un **lawyer** ed un **evaluator**.

File “customer.py”:

```
1.  import csv
2.  import random
3.  from faker import Faker
4.  fake = Faker(['en-US'])
5.
6.  reclami = ["Property damage", "Vehicle damage", "Health damage"]
7.
8.  #Creo ed apro il file csv
9.  with open('customer40000.csv', 'w', newline="") as customer:
10.     fieldnames = ['ID','NAME', 'ADDRESS','EMAIL','PHONE','SSN', 'CLAIM', 'LAWYER', 'EVALUATOR']
11.     writer = csv.DictWriter(customer, fieldnames=fieldnames)
12.     writer.writeheader()
13.     id = 1
14.
15.     #Creo ed inserisco i dati all'interno del file .csv
16.     for x in range(40000):
17.
18.         #I numeri pari avranno un claim
19.         if id % 2 == 0:
20.             k = random.randint(0,2)
21.             name = fake.name()
22.             address = fake.street_address()
23.             email = fake.email()
24.             phone = fake.phone_number()
25.             ssn = fake.ssn()
26.             claim = reclami[k]
27.             lawyer = fake.name()
28.             evaluator = fake.name()
29.             writer.writerow({'ID' : id, 'NAME': name, 'ADDRESS' : address, 'EMAIL' : email, '
    PHONE' : phone, 'SSN': ssn, 'CLAIM': claim, 'LAWYER': lawyer, 'EVALUATOR': evaluator})
30.             id += 1
31.
32.         #I numeri dispari non avranno claim
33.         else:
34.             name = fake.name()
35.             address = fake.street_address()
36.             email = fake.email()
37.             phone = fake.phone_number()
38.             ssn = fake.ssn()
39.             claim = None
40.             lawyer = None
41.             evaluator = None
42.             writer.writerow({'ID' : id, 'NAME': name, 'ADDRESS' : address, 'EMAIL' : email, '
    PHONE' : phone, 'SSN': ssn, 'CLAIM': claim, 'LAWYER': lawyer, 'EVALUATOR': evaluator})
43.             id += 1
```

Per creare i legami tra i customer, abbiamo creato un secondo file “legami.py”, che ci fornisce coppie di utenti con stesso address o stesso ssn e gruppi di tre utenti con email e phone uguali.

File “legami.py”:

```
from tempfile import NamedTemporaryFile
import shutil
import csv
import random
from faker import Faker
fake = Faker(['en-US'])

reclami = ["Property damage", "Car damage", "Health damage"]
num_casuali_email_phone = []
num_casuali_ssn = []
num_random = 0

#Creo due liste di numeri manualmente per i legami.
#Lista numeri per email e phone
for x in range(5,40000,10):
    num_random = x
    num_casuali_email_phone.append(num_random)

#Lista numeri per gli ssn
for x in range(32,40000,10):
    num_random = x
    num_casuali_ssn.append(num_random)

filename = 'customer40000.csv'
tempfile = NamedTemporaryFile('w+t', newline="", delete=False)

fields = ['ID','NAME','ADDRESS','EMAIL','PHONE','SSN', 'CLAIM', 'LAWYER', 'EVALUATOR']

with open(filename,'r', newline="") as csv_file, tempfile:
    reader = csv.DictReader(csv_file, fieldnames=fields)
    writer = csv.DictWriter(tempfile, fieldnames=fields)

    #Inizializzo le variabili che mi serviranno
    first_row = 0 #Serve per creare la prima riga con i fields
    p = 0
    q1 = 0
    q2 = 0
    v = 0
    t = 0
    k = 0

    for row in reader:

        #Creo i fields
        if first_row == 0:
            id = "ID"
            name = "NAME"
            address = "ADDRESS"
            email = "EMAIL"
            phone = "PHONE"
            ssn = "SSN"
            claim = "CLAIM"
            lawyer = "LAWYER"
            evaluator = "EVALUATOR"
            row = {'ID': id, 'NAME': name, 'ADDRESS': address, 'EMAIL': email, 'PHONE': phone, 'SSN': ssn
, 'CLAIM': claim, 'LAWYER': lawyer, 'EVALUATOR': evaluator}
            writer.writerow(row)
            first_row += 1
            continue

        #Trasformo l'id da str a int in modo da poter creare i legami
        id = int(row['ID'])

        if k >= 3999:
            k = 0

```

```
        if t >= 3997:
            t = 0

        #Se id multiplo di 10, crea due utenti con stesso indirizzo, che quindi condivideranno lo stesso claim
        if id % 10 == 0:
            name = row['NAME']
            address = row['ADDRESS']
            y = address
            email = row['EMAIL']
            phone = row['PHONE']
            ssn = row['SSN']
            claim = row['CLAIM']
            u = claim
            lawyer = row['LAWYER']
            w = lawyer
            evaluator = row['EVALUATOR']
            e = evaluator
            row = {'ID': id, 'NAME': name, 'ADDRESS': address, 'EMAIL': email, 'PHONE': phone, 'SSN': ssn, 'CLAIM': claim, 'LAWYER': lawyer, 'EVALUATOR': evaluator}
            writer.writerow(row)
            p = id + 1

        elif id == p:
            name = fake.name()
            address = y
            email = fake.email()
            phone = fake.phone_number()
            ssn = fake.ssn()
            claim = u
            lawyer = w
            evaluator = e
            row = {'ID': id, 'NAME': name, 'ADDRESS': address, 'EMAIL': email, 'PHONE': phone, 'SSN': ssn, 'CLAIM': claim, 'LAWYER': lawyer, 'EVALUATOR': evaluator}
            writer.writerow(row)
            p = 0

        #Utente casuale con dei dati, utente successivo con stessa email e terzo utente con stesso numero di telefono, condividono dati unici e due di loro aprono un claim, probabile frode
        elif id == num_casuali_email_phone[k]:
            var = random.randint(0,2)
            name = row['NAME']
            address = row['ADDRESS']
            email = row['EMAIL']
            y = row['EMAIL']
            phone = row['PHONE']
            z = phone
            ssn = row['SSN']
            if id % 2 == 0:
                claim = reclami[var]
                u = claim
                lawyer = fake.name()
                w = lawyer
                evaluator = fake.name()
                e = evaluator
            else:
                claim = None
                lawyer = None
                evaluator = None
            row = {'ID': id, 'NAME': name, 'ADDRESS': address, 'EMAIL': email, 'PHONE': phone, 'SSN': ssn, 'CLAIM': claim, 'LAWYER': lawyer, 'EVALUATOR': evaluator}
            writer.writerow(row)
            q1 = id + 1

        elif id == q1:
            var = random.randint(0,2)
            name = fake.name()
            address = fake.street_address()
            email = y #Prendo l'email che mi servirà per il successivo
            phone = fake.phone_number()
            ssn = fake.ssn()
            if id % 2 == 0 : #Dato che in "customer" avevamo settato che i numeri pari hanno un claim, facciamo la stessa cosa qui e salviamo tutto nelle variabili
                claim = reclami[var]
                u = claim
                lawyer = fake.name()
```

```
                w = lawyer
                evaluator = fake.name()
                e = evaluator
            else:
                claim = None
                lawyer = None
                evaluator = None
            row = {'ID': id, 'NAME': name, 'ADDRESS': address, 'EMAIL': email, 'PHONE': phone, 'SSN': ssn
    , 'CLAIM': claim, 'LAWYER': lawyer, 'EVALUATOR': evaluator}
            writer.writerow(row)
            q2 = id + 1

        elif id == q2:
            var = random.randint(0,2)
            name = fake.name()
            address = fake.street_address()
            email = fake.email()
            phone = z
            ssn = fake.ssn()
            claim = u
            lawyer = w
            evaluator = e
            row = {'ID': id, 'NAME': name, 'ADDRESS': address, 'EMAIL': email, 'PHONE': z, 'SSN': ssn, 'C
    LAIM': claim, 'LAWYER': lawyer, 'EVALUATOR': evaluator}
            writer.writerow(row)
            q1 = 0
            q2 = 0
            k = k + 1

        elif id == num_casuali_ssn[t]:
            name = row['NAME']
            address = row['ADDRESS']
            email = row['EMAIL']
            phone = row['PHONE']
            ssn = row['SSN']
            y = row['SSN']
            claim = row['CLAIM']
            lawyer = row['LAWYER']
            evaluator = row['EVALUATOR']
            row = {'ID': id, 'NAME': name, 'ADDRESS': address, 'EMAIL': email, 'PHONE': phone, 'SSN': ssn
    , 'CLAIM': claim, 'LAWYER': lawyer, 'EVALUATOR': evaluator}
            writer.writerow(row)
            v = id + 1

        elif id == v:
            var = random.randint(0,2)
            name = fake.name()
            address = fake.street_address()
            email = fake.email()
            phone = fake.phone_number()
            if id % 2 == 0:
                claim = reclami[var]
                lawyer = fake.name()
                evaluator = fake.name()
            else:
                claim = None
                lawyer = None
                evaluator = None
            row = {'ID': id, 'NAME': name, 'ADDRESS': address, 'EMAIL': email, 'PHONE': phone, 'SSN': ssn
    , 'CLAIM': claim, 'LAWYER': lawyer, 'EVALUATOR': evaluator}
            writer.writerow(row)
            v = 0
            t = t + 1

        else:
            writer.writerow(row)

shutil.move(tempfile.name, filename) #Sostituisco i valori del file temporaneo all'interno del file custo
    mer.csv
```

4. CONNESSIONE:

Il linguaggio di programmazione utilizzato per la gestione dei database **Neo4j** e **Cassandra**, è stato, anche quì, “Python 3.9”.

- Neo4j:

Per poter far interagire Python e **Neo4j**, abbiamo innanzitutto installato il driver ufficiale tramite il prompt dei comandi, utilizzando:
“**python -m pip install neo4j**”.

Successivamente abbiamo importato nel compilatore il comando:
“**from neo4j import GraphDatabase**”
Questo ci permette di poter eseguire **Cypher** al suo interno, tramite delle “**session**”:

```
1.  from neo4j import GraphDatabase
2.
3.  uri = "bolt://localhost:7687"
4.  user = "neo4j"
5.  psw = "admin"
6.
7.  driver = GraphDatabase.driver(uri, auth=(user, psw))
8.  session = driver.session()
```

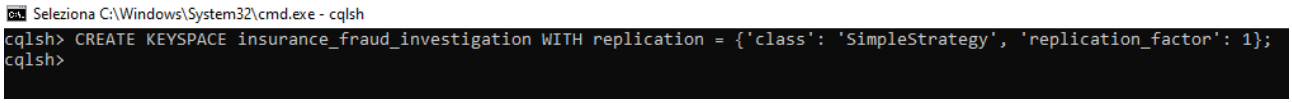
- Cassandra:

Per poter far interagire Python e **Cassandra,** abbiamo, anche qui installato il driver ufficiale tramite il prompt dei comandi, utilizzando:
“**pip install cassandra-driver**”

Successivamente abbiamo importato nel compilatore il comando:
“**from cassandra.cluster import Cluster**”

Prima di poter iniziare a eseguire qualsiasi query su un cluster Cassandra, è necessario configurare un'istanza di Cluster. Come suggerisce il nome, in genere si avrà un'istanza Cluster per ogni cluster Cassandra con cui vuoi interagire.
Il Cluster è stato creato tramite il comando: **cluster = Cluster()**.

Il passo successivo sarà creare il KEYSPACE, direttamente da **cqlsh** tramite:



Mentre la connessione al KEYSPACE verrà eseguita tramite una session, seguita da una session.execute che ci permette di utilizzare il KEYSPACE appena creato.

```
1.  from cassandra.cluster import Cluster
2.
3.  cluster = Cluster()
4.
5.  session = cluster.connect('insurance_fraud_investigation')
6.  session.execute("USE insurance_fraud_investigation")
```

5. IMPORTAZIONE:

Importazione tramite “Python 3.9“

- Neo4j:

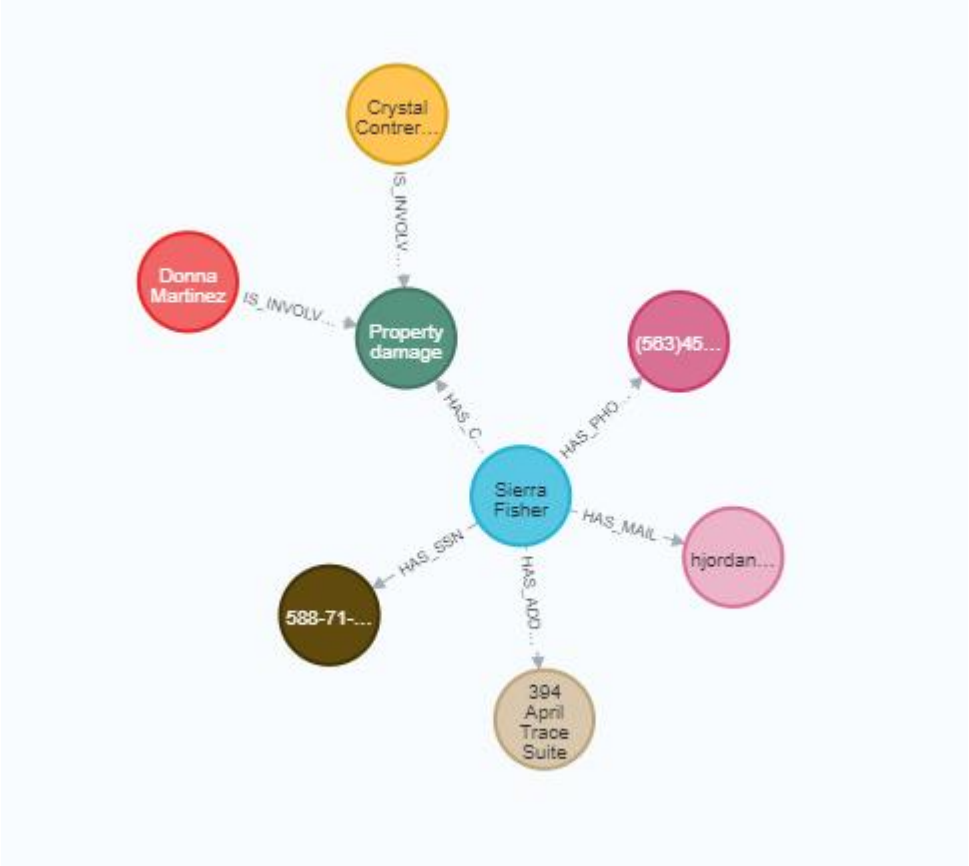
In **Neo4j**, dopo esserci collegati al db, abbiamo diviso le importazioni in due session:

- Nella prima **session.run**, saranno caricati i customer con i rispettivi attributi e le relazioni presenti tra di essi, quando il claim risulta nullo.

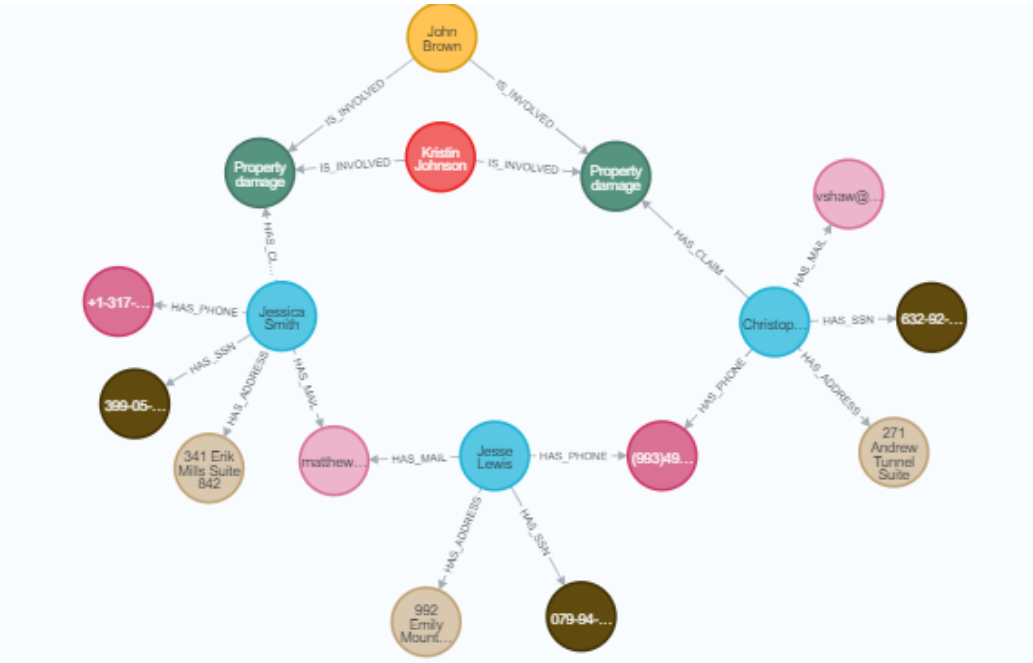
- Nella seconda **session.run** verranno invece caricati i customer in cui è presente un claim, insieme ai rispettivi lawyer ed evaluator e successivamente create le relazioni sia tra i vari customer che tra lawyer ed evaluator.

```
1.  from neo4j import GraphDatabase
2.
3.  uri = "bolt://localhost:7687"
4.  user = "neo4j"
5.  psw = "admin"
6.
7.  driver = GraphDatabase.driver(uri, auth=(user, psw))
8.  session = driver.session()
9.
10. session.run("""USING PERIODIC COMMIT 1000
11.                 LOAD CSV WITH HEADERS FROM
                    "file:///customer40000.csv" AS line WITH line WHERE line.CLAIM IS null
12.                 CREATE (n:Name {NAME: line.NAME})
13.                 MERGE (a:Address {ADDRESS: line.ADDRESS})
14.                 MERGE (e:Email {EMAIL: line.EMAIL})
15.                 MERGE (p:Phone {PHONE: line.PHONE})
16.                 MERGE (s:Ssn {SSN: line.SSN})
17.                 MERGE (n)-[:HAS_ADDRESS]->(a)
18.                 MERGE (n)-[:HAS_MAIL]->(e)
19.                 MERGE (n)-[:HAS_PHONE]->(p)
20.                 MERGE (n)-[:HAS_SSN]->(s)
21.                 """)
22.
23. session.run("""USING PERIODIC COMMIT 1000
24.                 LOAD CSV WITH HEADERS FROM
                    "file:///customer40000.csv" AS line WITH line WHERE line.CLAIM IS NOT null
25.                 CREATE (n:Name {NAME: line.NAME})
26.                 CREATE (cl:Claim {Claim: line.CLAIM})
27.                 MERGE (a:Address {ADDRESS: line.ADDRESS})
28.                 MERGE (e:Email {EMAIL: line.EMAIL})
29.                 MERGE (p:Phone {PHONE: line.PHONE})
30.                 MERGE (s:Ssn {SSN: line.SSN})
31.                 MERGE (l:Lawyer {LAWYER: line.LAWYER})
32.                 MERGE (ev:Evaluator {EVALUATOR: line.EVALUATOR})
33.                 MERGE (n)-[:HAS_ADDRESS]->(a)
34.                 MERGE (n)-[:HAS_MAIL]->(e)
35.                 MERGE (n)-[:HAS_PHONE]->(p)
36.                 MERGE (n)-[:HAS_SSN]->(s)
37.                 MERGE (n)-[:HAS_CLAIM]->(cl)
38.                 MERGE (l)-[:IS_INVOLVED]->(cl)
39.                 MERGE (ev)-[:IS_INVOLVED]->(cl)
40.                 """)
```

Di seguito è riportato un esempio della rappresentazione base dei dati in Neo4j con le relazioni tra i vari nodi:



Oltre la rappresentazione base, sono presenti anche altri nodi in cui abbiamo collegamenti tra phone, ssn o email, soprattutto nei casi di frode:



- Cassandra:

In Cassandra abbiamo utilizzato un session execute per la creazione della tabella, dove inseriamo anche l'id che sarà una chiave primaria. Successivamente abbiamo aperto il file .csv in lettura e per ogni riga, abbiamo inserito i valori all'interno della tabella. Tutti i dati sono "str", appunto per questo abbiamo deciso di togliere i trattini all'interno del phone e dell'ssn, in modo da poterli inserire senza problemi.

```
session.execute("""
                CREATE TABLE IF NOT EXISTS Customer (
                ID text,
                Name text,
                Address varchar,
                Email varchar,
                Phone text,
                Ssn text,
                Claim text,
                Lawyer text,
                Evaluator text,
                PRIMARY KEY (ID)
                )
                """)

fields = ['ID','NAME','ADDRESS','EMAIL','PHONE','SSN','CLAIM','LAWYER','EVALUATOR']

with open('customer1000.csv', 'rb') as customer:
    reader = csv.DictReader(customer, fieldnames=fields)
    next(reader)  #Salto la prima riga con i fields
    for row in reader:
        #Sostituiamo il trattino con uno spazio vuoto altrimenti non ci stampa i dati
        phone = row['PHONE'].replace("-"," ")
        ssn = row['SSN'].replace("-"," ")
        CQLString = ("INSERT INTO Customer
        (ID,Name,Address,Email,Phone,Ssn,Claim,Lawyer,Evaluator)
        VALUES (%s,%s,%s,%s,%s,%s,%s,%s,%s)")
        session.execute(CQLString, (row['ID'],row['NAME'],row['ADDRESS'],row['EMAIL'],phone,
        ssn,row['CLAIM'],row['LAWYER'],row['EVALUATOR']))
```

Una volta caricati i dati, calcoliamo i tempi di esecuzione con:

```
tempo = lambda: int(round(time.time() * 1000))

for x in range(31):
    a1 = tempo()

 #Qui ci va la query di inserimento tramite sessionrun in neo4j o sessionexecute in Cassandra

    a2 = tempo()
    print("Il risultato di " ,x+1, " e'" ,a2-a1, " millesecondi\n")
```

Di seguito sono mostrati i tempi di esecuzione dei due database:

Import	Neo4j	Cassandra
10.000	113301	155376
20.000	457368	310293
30.000	955216	464637
40.000	1820855	616878

6. QUERY:

In informatica, il termine **query** indica l'interrogazione di una base di dati da parte di un utente. Il database è in genere strutturato secondo il modello relazionale, che permette di compiere determinate operazioni sui dati (selezione, inserimento, cancellazione, aggiornamento, ecc.).

Esistono diversi tipi di query:

- Query di selezione: è possibile estrarre i dati (**SELECT**) da una o più tabelle e visualizzarli in una nuova tabella.

- Query di accodamento: consente di aggiungere (**INSERT**) a tabelle già esistenti un gruppo di record in base a dei criteri specifici.

- Query di aggiornamento: consente di modificare (**UPDATE**) il valore di uno o più campi in corrispondenza di un intervallo selezionato di records esistenti.

- Query di eliminazione: consente di cancellare (**DELETE**) uno o più record dipendentemente dai criteri inseriti nella creazione della query.

- Query a campi incrociati: permette di "incrociare" i campi di più tabelle in modo tale da ottenere una matrice in cui le righe corrispondono normalmente a campi di tipo descrittivo e riepilogativo, mentre le colonne corrispondono a totali o conteggi. Vengono chiamate anche query pivot, poiché i dati che risultano alla fine della creazione sono simili alle tabelle pivot di un foglio di calcolo.

- Query di creazione tabella: se lo si desidera, le righe prodotte da una query possono anche alimentare i record di una nuova tabella mediante una query di creazione tabella, facendo sempre riferimento alla scheda Struttura, gruppo Tipo di query.

Esistono inoltre le query annidate dove il filtro della query è dato da un'altra query.

Di seguito, è stato creato un elenco di query, utilizzate in entrambi i database, accompagnati da una tabella che descrive, in base alla query:

- Media della query eseguita 31 volte, escludendo il primo

- Deviazione standard

- Intervallo di confidenza calcolato al 95%

- Istogramma che mostra la latenza media di esecuzione con la rappresentazione dell’intervallo di confidenza entro il quale i risultati vengono considerati accettabili.

QUERY 1:

Con la prima query, andremo a selezionare tutti i name il cui claim è “Car damage”

- Neo4j:

```
import time
import xlsxwriter
from neo4j import GraphDatabase

uri = “bolt://localhost:7687”
user = “neo4j”
psw = “admin”

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

tempo = lambda: int(round(time.time() * 1000))
workbook = xlsxwriter.Workbook(‘query1__neo4j_10000.xlsx’)
worksheet = workbook.add_worksheet()
row = 1
col = 0

def query1():
    with driver.session() as session:
        session.run(“””MATCH(n:Name)
                       WHERE n.CLAIM = ‘Car damage’
                       RETURN n”””)

for x in range(31):
    a1 = tempo()
    query1()
    a2 = tempo()
    worksheet.write(row, col, a2-a1)
    row += 1
    print(“Il risultato di “ ,x+1, “ e’” ,a2-a1, “ millesecondi\n”)

worksheet.write(row, col, a2-a1)
workbook.close()
```

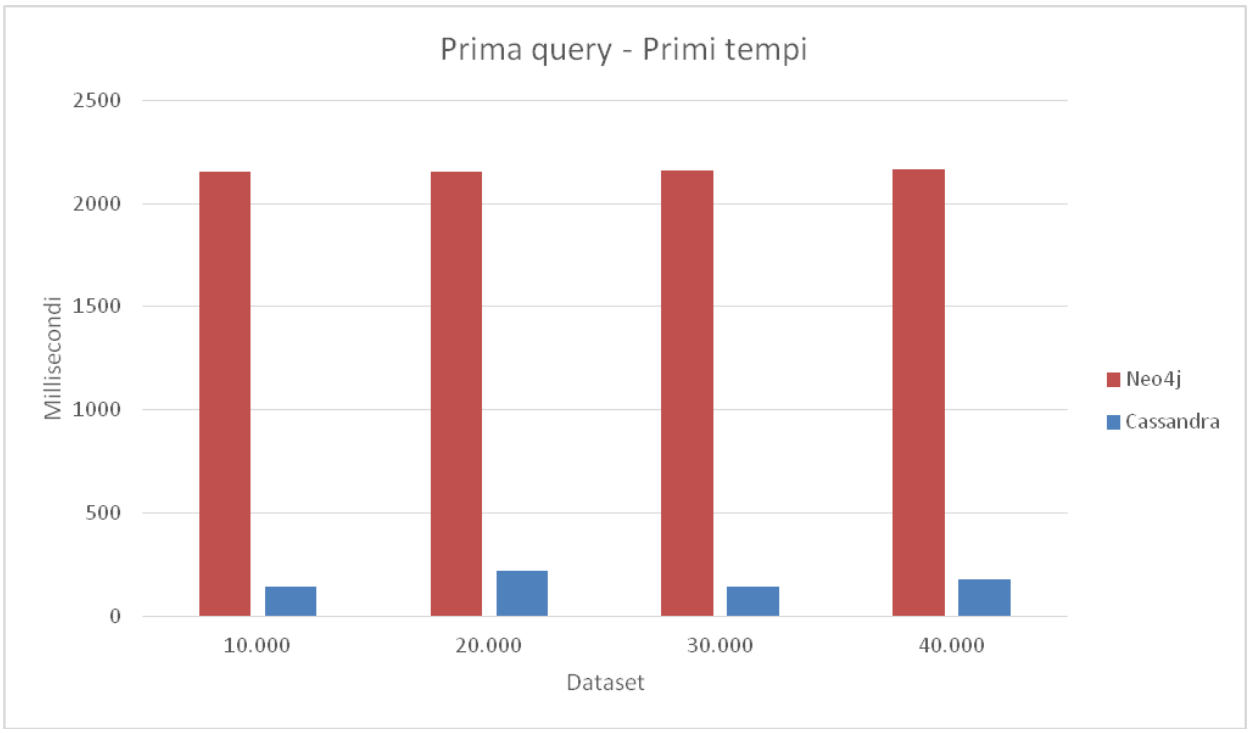
- Cassandra:

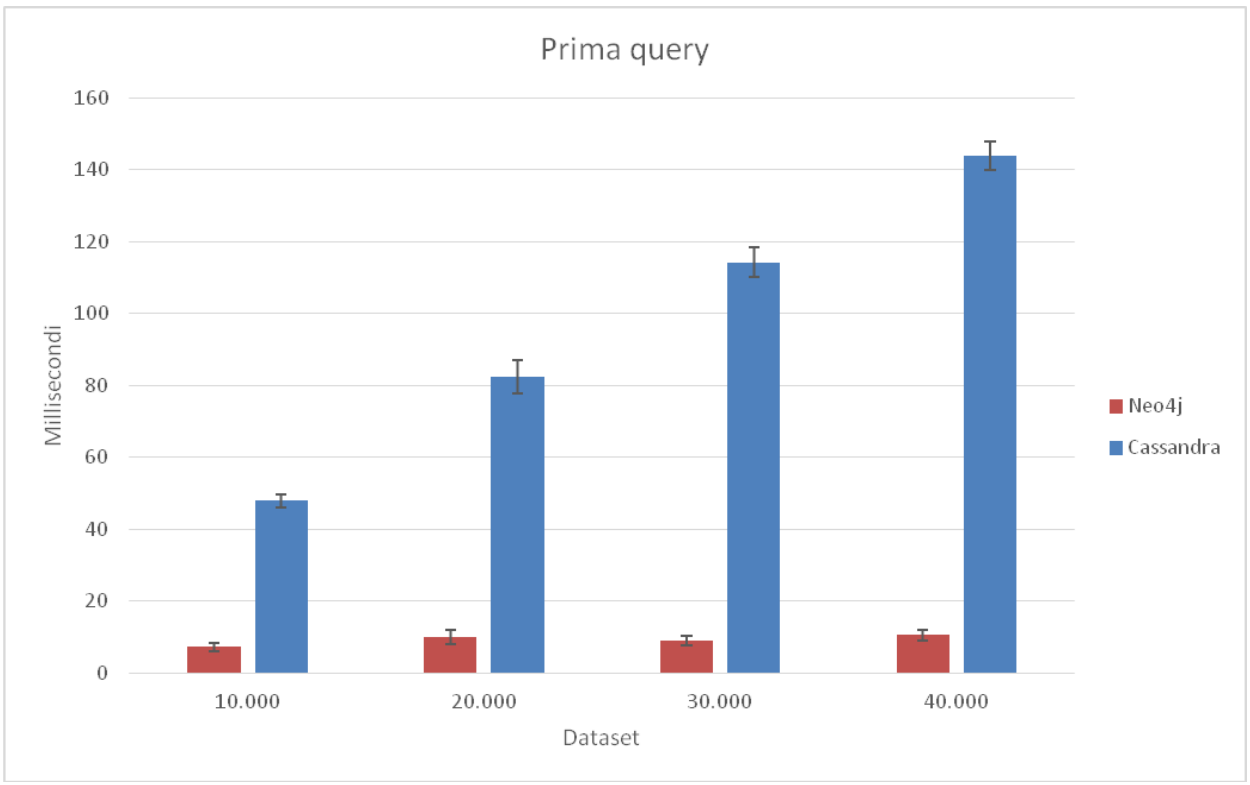
```
import time
import xlsxwriter
from cassandra.cluster import Cluster

Cluster = Cluster()

tempo = lambda: int(round(time.time() * 1000))
workbook = xlsxwriter.Workbook(‘query1__neo4j_10000.xlsx’)
worksheet = workbook.add_worksheet()
row = 1
col = 0

def query1():
    session.execute(“””SELECT name,claim FROM Customer WHERE claim = ‘Car damage’
                       ALLOW FILTERING;”””)

for x in range(31):
    a1 = tempo()
    query1()
    a2 = tempo()
    worksheet.write(row, col, a2-a1)
    row += 1
    print(“Il risultato di “ ,x+1, “ e’” ,a2-a1, “ millesecondi\n”)

worksheet.write(row, col, a2-a1)
workbook.close()
```

Risultati:

Prima Query	Neo4j 10.000	Cassandra 10.000	Neo4j 20.000	Cassandra 20.000	Neo4j 30.000	Cassandra 30.000	Neo4j 40.000	Cassandra 40.000
Primi Tempi	2152	143	2151	220	2158	142	2165	177
Media	7,3	48,36666	10,43333	82,4	8,862	114,3	10,6	144,13333
Deviazione Standard	3,4	5,18	5,72	13,22	3,71	11,65	4,12	11,32
95%conf	1,19686774	1,823463203	2,013553962	4,653703387	1,30599392	4,101032108	1,45032208	3,984865533

Istogramma:





QUERY 2:

Con la seconda query, andremo a selezionare tutti i name il cui claim è “Car damage” e i la cui iniziale è la lettera “M”

- Neo4j:

```
1.  def query2(valueClaim, valueName):
2.      with driver.session() as session:
3.          session.run(“””
4.                      MATCH (n:Name)-[r:HAS_CLAIM]->(c:Claim)
5.                      WHERE c.Claim = $valueClaim AND n.NAME STARTS WITH $valueName
6.                      RETURN n,r,c”””, valueClaim = valueClaim, valueName = valueName )
7.
8.  for x in range(31):
9.      a1 = tempo()
10.     query2(‘Car damage’,’M’)
11.     a2 = tempo()
12.     worksheet.write(row, col, a2-a1)
13.     row += 1
14.     print(“Il risultato di “ ,x+1, “ e’” ,a2-a1, “ millesecondi\n”)
```

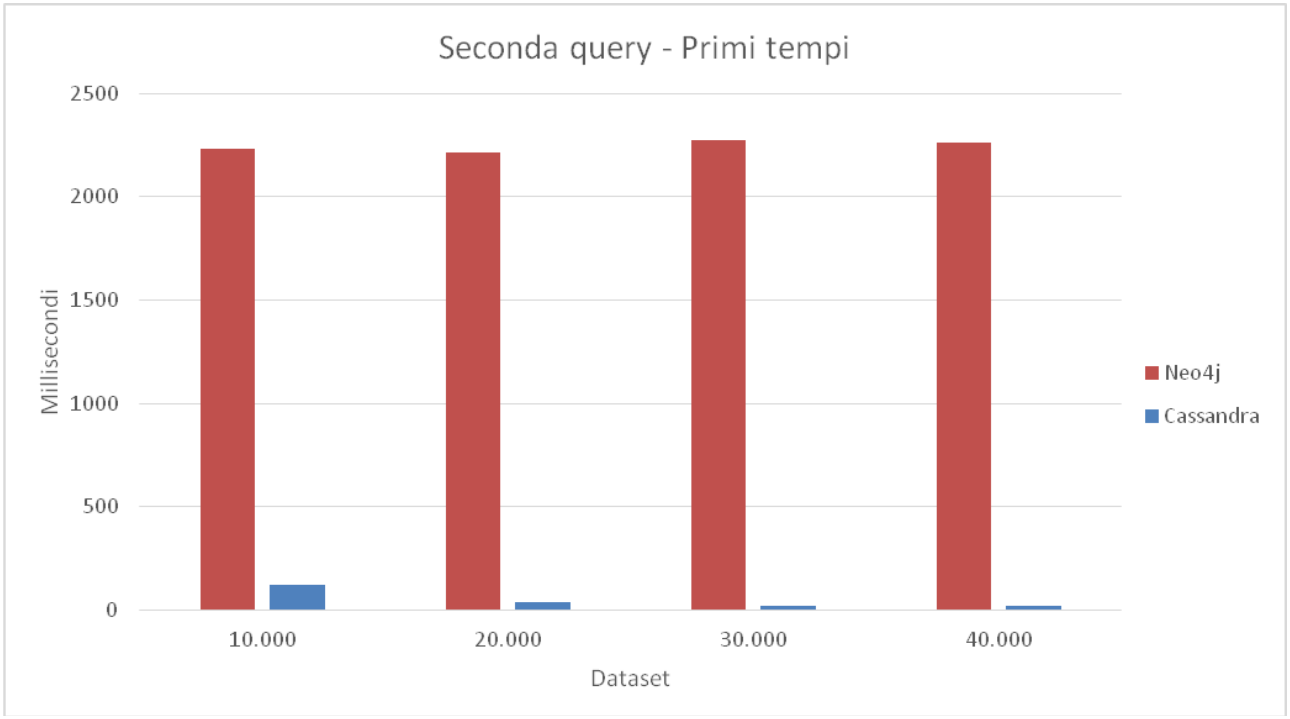
- Cassandra:

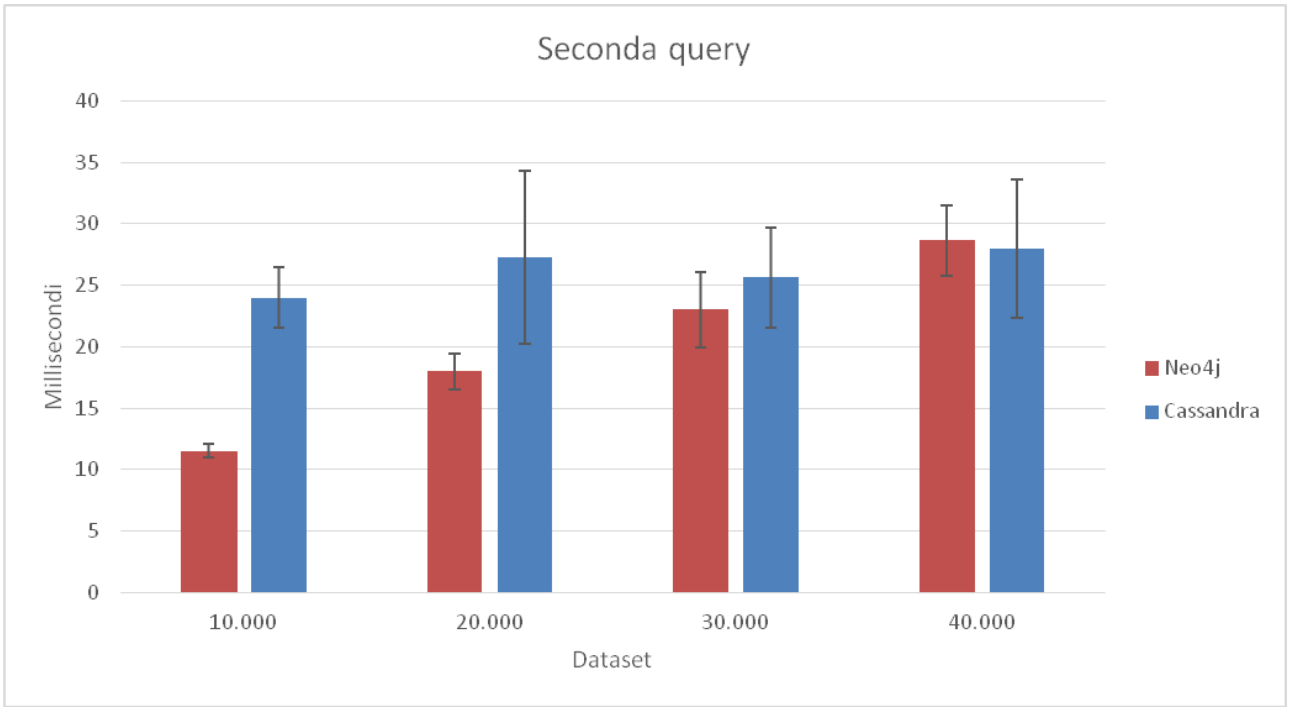
```
1.  #Va eseguita una volta sola
2.  session.execute(“””CREATE CUSTOM INDEX name_index ON Customer (name)
                    USING ‘org.apache.cassandra.index.sasi.SASIIndex’;”””)
3.
4.  def query2():
5.      session.execute(“””SELECT name,claim FROM Customer WHERE claim = ‘Car damage’
                        AND name LIKE ‘M%’ ALLOW FILTERING;”””)
6.
7.  for x in range(31):
8.      a1 = tempo()
9.      query2()
10.     a2 = tempo()
11.     worksheet.write(row, col, a2-a1)
12.     row += 1
13.     print(“Il risultato di “ ,x+1, “ e’” ,a2-a1, “ millesecondi\n”)
```

Risultati:

Seconda Query	Neo4j 10.000	Cassandra 10.000	Neo4j 20.000	Cassandra 20.000	Neo4j 30.000	Cassandra 30.000	Neo4j 40.000	Cassandra 40.000
Primi Tempi	2234	120	2214	40	2273	19	2265	19
Media	11,51612903	23,53	17,70967742	27,3	23,258065	25,63	28,64	27,73
Deviazione Standard	1,5631021	6,89798682	4,12852819	19,92845536	8,7840314	11,53106914	8,07843959	16,0269217
95%conf	0,54915108	2,42822879	1,45332418	7,0152132	3,09215406	4,05916608	2,84377169	5,64179575

Istogramma:


Seconda query - Primi tempi
2500
2000
1500
1000
500
0
Millisecondi
10.000
20.000
30.000
40.000
Dataset
Neo4j
Cassandra


Seconda query
40
35
30
25
20
15
10
5
0
Millisecondi
10.000
20.000
30.000
40.000
Dataset
Neo4j
Cassandra

QUERY 3:

Con la seconda query, andremo a selezionare tutti i name il cui claim è “Car damage” , i la cui iniziale è la lettera “M” e gli ssn che iniziano con “4”

- Neo4j:

```
def query3(valueClaim,valueName,valueSSN):
    with driver.session() as session:
        session.run("""
                    MATCH (n:Name)-[r:HAS_CLAIM]->(c:Claim)
                    MATCH (n:Name)-[r1:HAS_SSN]->(s:Ssn)
                    WHERE c.Claim = $valueClaim AND
                    n.NAME STARTS WITH $valueName AND s.SSN STARTS WITH $valueSSN
                    RETURN n,r,c,s,r1
                    """, valueClaim = valueClaim, valueName = valueName, valueSSN = valueSSN)

for x in range(31):
    a1 = tempo()
    query3('Car damage','M','4')
    a2 = tempo()
    worksheet.write(row, col, a2-a1)
    row += 1
    print("Il risultato di " ,x+1, " e'" ,a2-a1, " millesecondi\n")
```

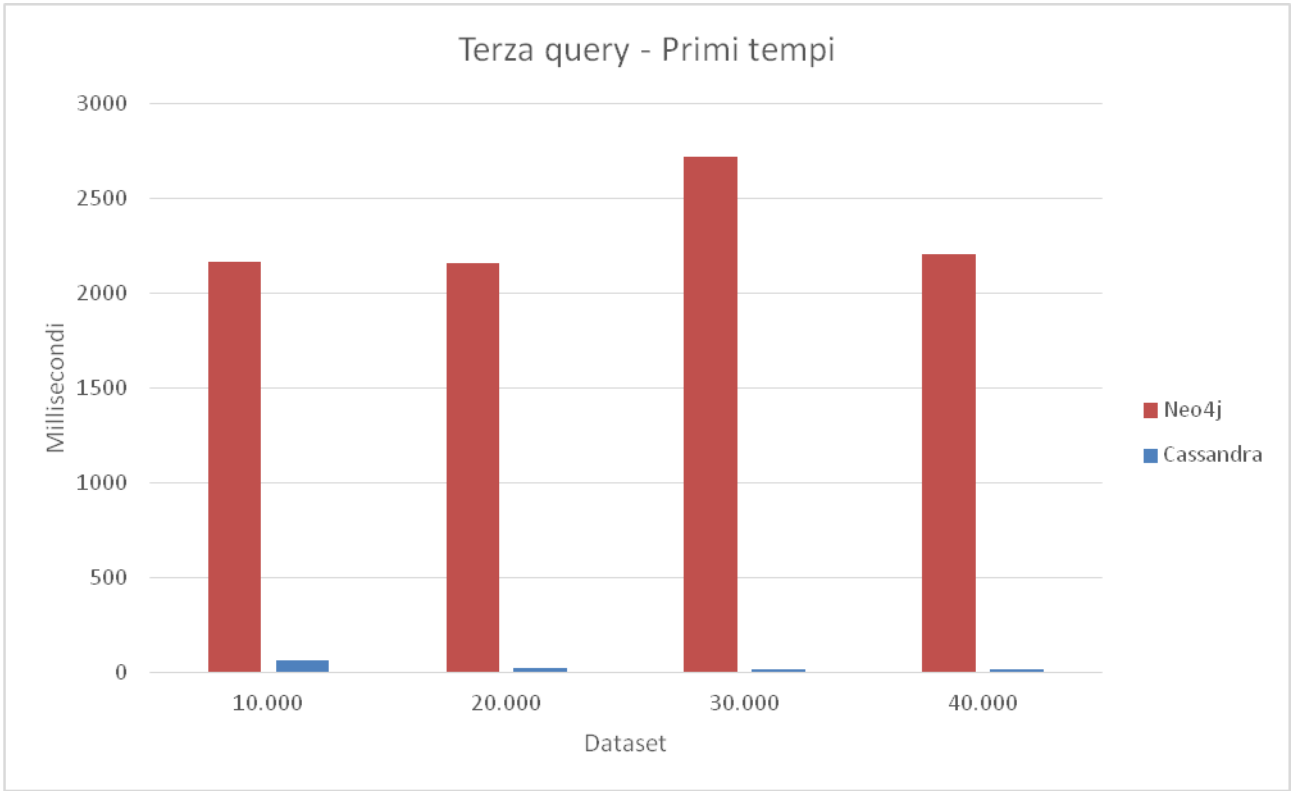
- Cassandra:

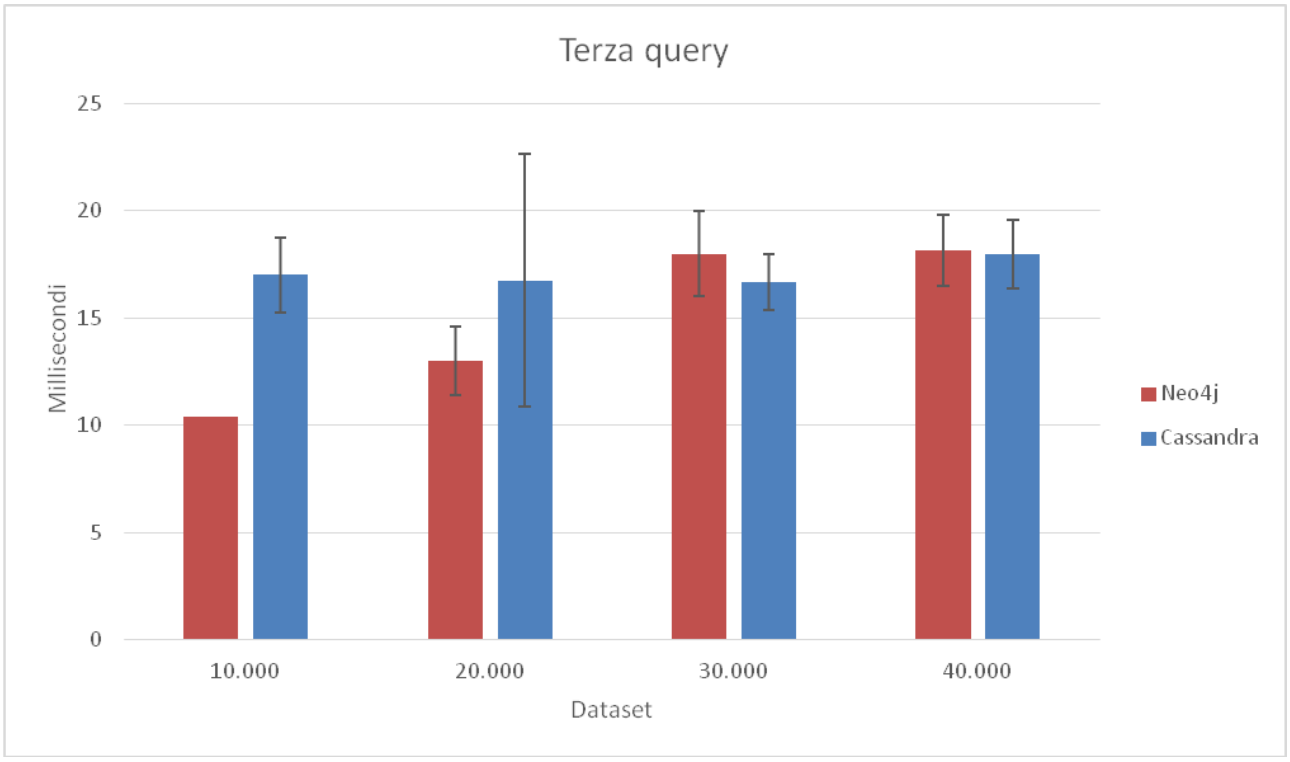
```
#Va eseguita una volta sola
session.execute("""CREATE CUSTOM INDEX ssn_index ON Customer (ssn)
                   USING 'org.apache.cassandra.index.sasi.SASIIndex';""")

def query3():
    session.execute("""SELECT name,claim,ssn FROM Customer WHERE claim = 'Car damage'
                       AND name LIKE 'M%' AND ssn LIKE '4%' ALLOW FILTERING;""")

for x in range(31):
    a1 = tempo()
    query3()
    a2 = tempo()
    worksheet.write(row, col, a2-a1)
    row += 1
    print("Il risultato di " ,x+1, " e'" ,a2-a1, " millesecondi\n")
```

Risultati:

Terza Query	Neo4j 10.000	Cassandra 10.000	Neo4j 20.000	Cassandra 20.000	Neo4j 30.000	Cassandra 30.000	Neo4j 40.000	Cassandra 40.000
Primi Tempi	2164	65	2155	22	2176	20	2207	20
Media	10,38709677	17	13,32258065	16,73333333	17,6129032	16,66666667	18,12903226	17,8
Deviazione Standard	2,611110496	4,983193978	4,525106285	4,760485503	5,64930723	3,681787006	4,729740738	4,53431362
95%conf	0.81355694	1,754183562	1,592927568	5,890466717	1,9886687	1,29606238	1,664962972	1,596168733

Istogramma:


Terza query - Primi tempi
3000
2500
2000
1500
1000
500
0
Millisecondi
10.000
20.000
30.000
40.000
Dataset
Neo4j
Cassandra


Terza query
25
20
15
10
5
0
Millisecondi
10.000
20.000
30.000
40.000
Dataset
Neo4j
Cassandra

QUERY 4:

Con la seconda query, andremo a selezionare tutti i name il cui claim è “Car damage” , la cui iniziale è la lettera “M”, gli ssn che iniziano con “4” e i lawyer

- Neo4j:

```
def query4(valueClaim,valueName,valueSSN):
    with driver.session() as session:
        session.run("""
                    MATCH (n:Name)-[r:HAS_CLAIM]->(c:Claim)<-[r2:IS_INVOLVED]-(l:Lawyer)
                    MATCH (n:Name)-[r1:HAS_SSN]->(s:Ssn)
                    WHERE c.Claim = $valueClaim AND n.NAME STARTS WITH $valueName
                    AND s.SSN STARTS WITH $valueSSN
                    RETURN n,r,c,s,r1,r2,l
                    """, valueClaim = valueClaim, valueName = valueName, valueSSN = valueSSN)

for x in range(31):
    a1 = tempo()
    query4('Car damage','M','4')
    a2 = tempo()
    worksheet.write(row, col, a2-a1)
    row += 1
    print("Il risultato di " ,x+1, " e'" ,a2-a1, " millesecondi\n")
```

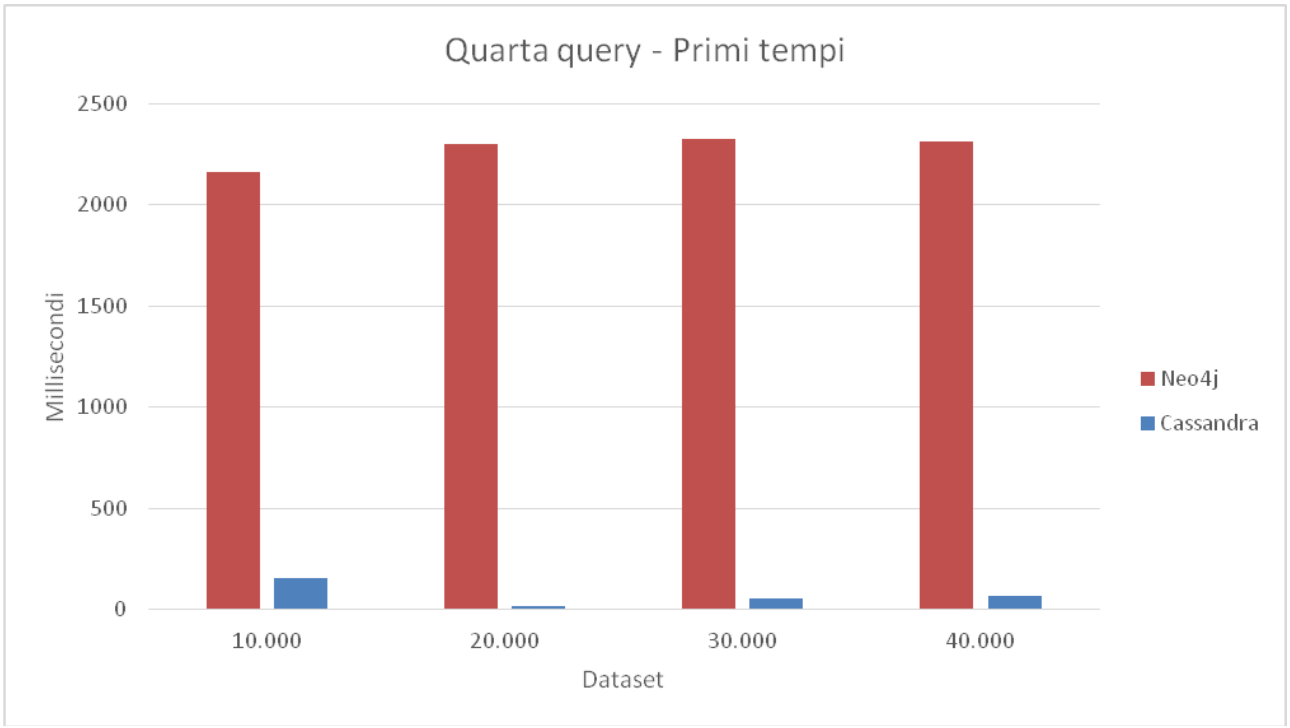
- Cassandra:

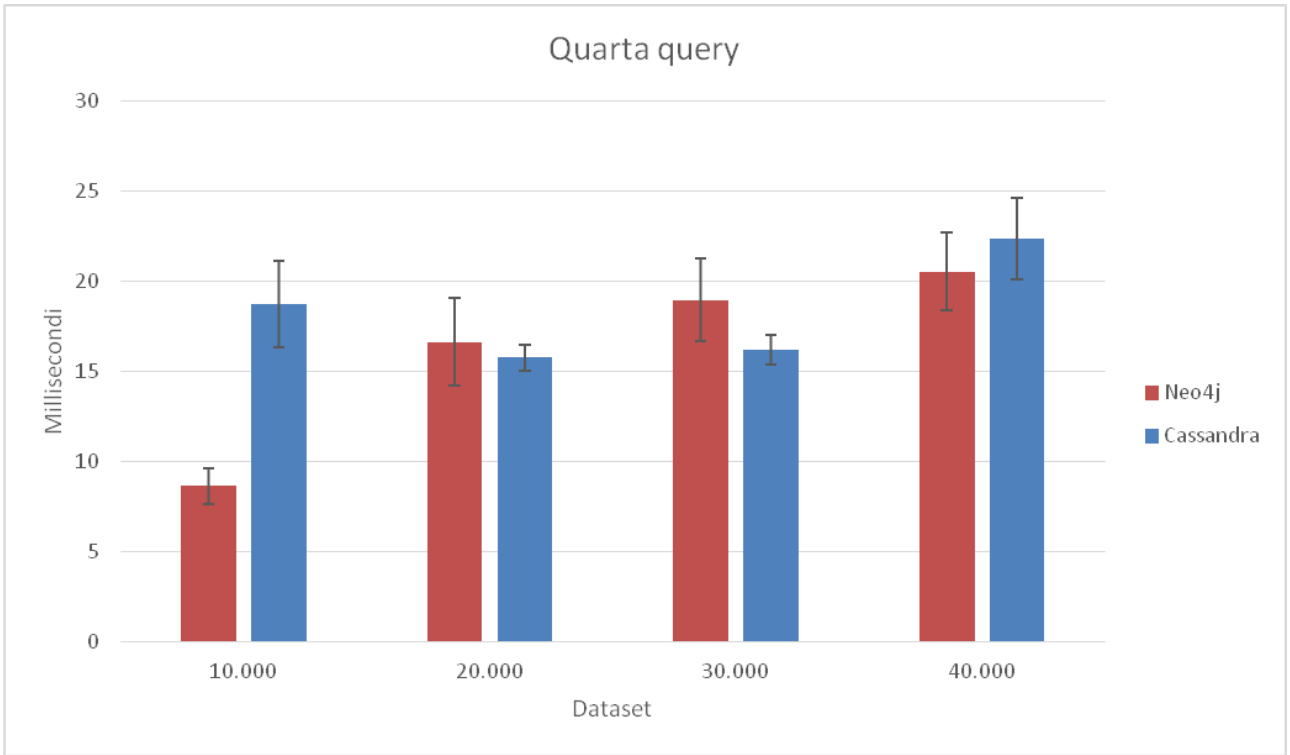
```
#Va eseguita una volta sola
session.execute("""CREATE CUSTOM INDEX ssn_index ON Customer (ssn)
                   USING 'org.apache.cassandra.index.sasi.SASIIndex';""")

def query4():
    session.execute("""SELECT name,claim,ssn,lawyer FROM Customer WHERE claim = 'Car damage'
                       AND name LIKE 'M%' AND ssn LIKE '4%' ALLOW FILTERING;""")

for x in range(31):
    a1 = tempo()
    query4()
    a2 = tempo()
    worksheet.write(row, col, a2-a1)
    row += 1
    print("Il risultato di " ,x+1, " e'" ,a2-a1, " millesecondi\n")
```

Risultati:

Quarta Query	Neo4j 10.000	Cassandra 10.000	Neo4j 20.000	Cassandra 20.000	Neo4j 30.000	Cassandra 30.000	Neo4j 40.000	Cassandra 40.000
Primi Tempi	2159	158	2298	20	2327	57	2310	69
Media	8,612903226	18,7333333333	16,61290323	15,73333333	18,9354839	16,2	20,51612903	22,33333
Deviazione Standard	2,755375033	6,821208756	6,856489365	1,99888858	6,53494568	2,343786111	6,158503001	6,4670103
95%conf	0,969946908	2,401201384	2,413620861	0,703648605	2,30043108	0,825059407	2,167915755	2,276516471

Istogramma:


Quarta query - Primi tempi
2500
2000
1500
1000
500
0
Millisecondi
10.000
20.000
30.000
40.000
Dataset
Neo4j
Cassandra


Quarta query
30
25
20
15
10
5
0
Millisecondi
10.000
20.000
30.000
40.000
Dataset
Neo4j
Cassandra

QUERY 5:

Con la seconda query, andremo a selezionare tutti i name il cui claim è “Car damage” , la cui iniziale è la lettera “M”, gli ssn che iniziano con “4”, i lawyer e gli evaluator

- Neo4j:

```
def query5(valueClaim,valueName,valueSSN):
    with driver.session() as session:
        session.run("""
                    MATCH (n:Name)-[r:HAS_CLAIM]->(c:Claim)<-[r2:IS_INVOLVED]-(l:Lawyer)
                    MATCH (c:Claim)<-[r3:IS_INVOLVED]-(e:Evaluator)
                    MATCH (n:Name)-[r1:HAS_SSN]->(s:Ssn)
                    MATCH (n:Name)-[r4:HAS_PHONE]->(p:Phone)
                    MATCH (n:Name)-[r5:HAS_MAIL]->(m:Email)
                    MATCH (n:Name)-[r6:HAS_ADDRESS]->(a:Address)
                    WHERE c.Claim = $valueClaim AND n.NAME STARTS WITH $valueNam
                    AND s.SSN STARTS WITH $valueSSN
                    RETURN n,r,c,r2,l,r3,e,r1,s,r4,p,r5,m,r6,a
                    """, valueClaim = valueClaim, valueName = valueName, valueSSN = valueSSN)

for x in range(31):
    a1 = tempo()
    query5('Car damage','M','4')
    a2 = tempo()
    worksheet.write(row, col, a2-a1)
    row += 1
    print("Il risultato di " ,x+1, " e'" ,a2-a1, " millesecondi\n")
```

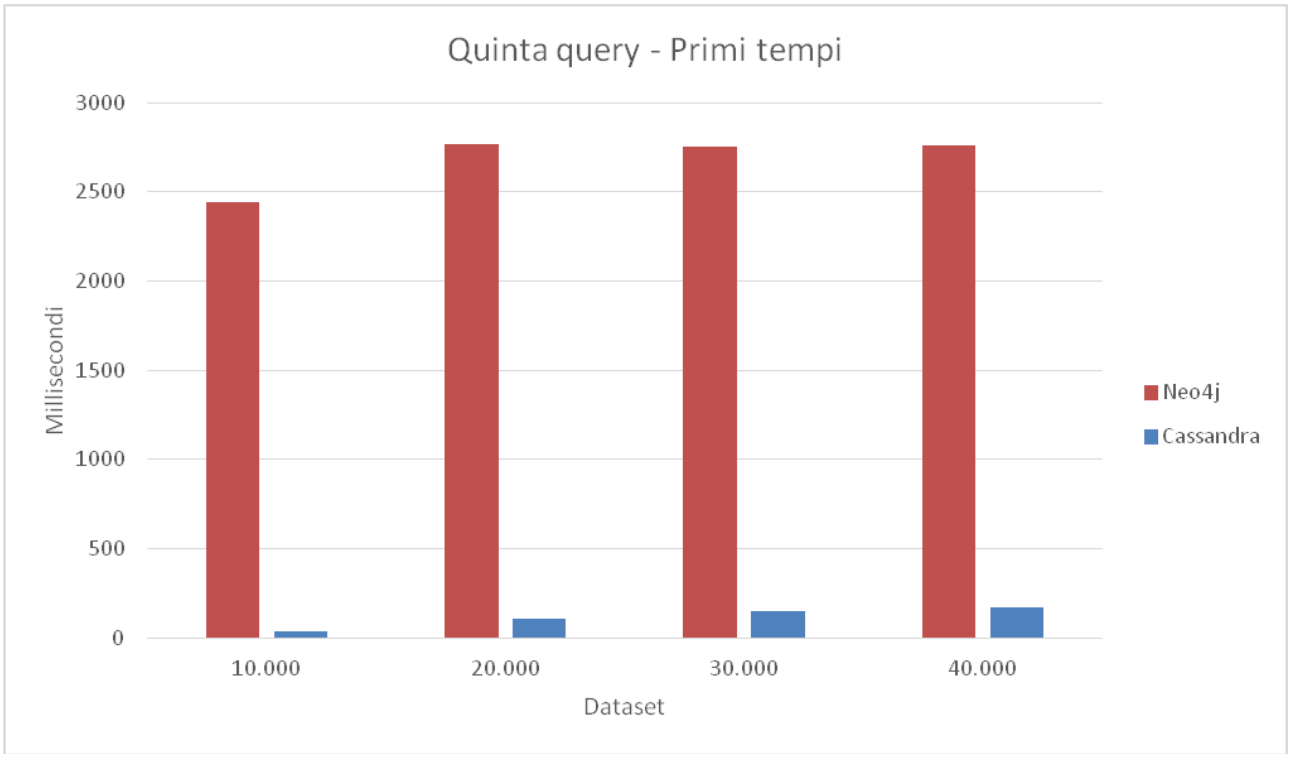
- Cassandra:

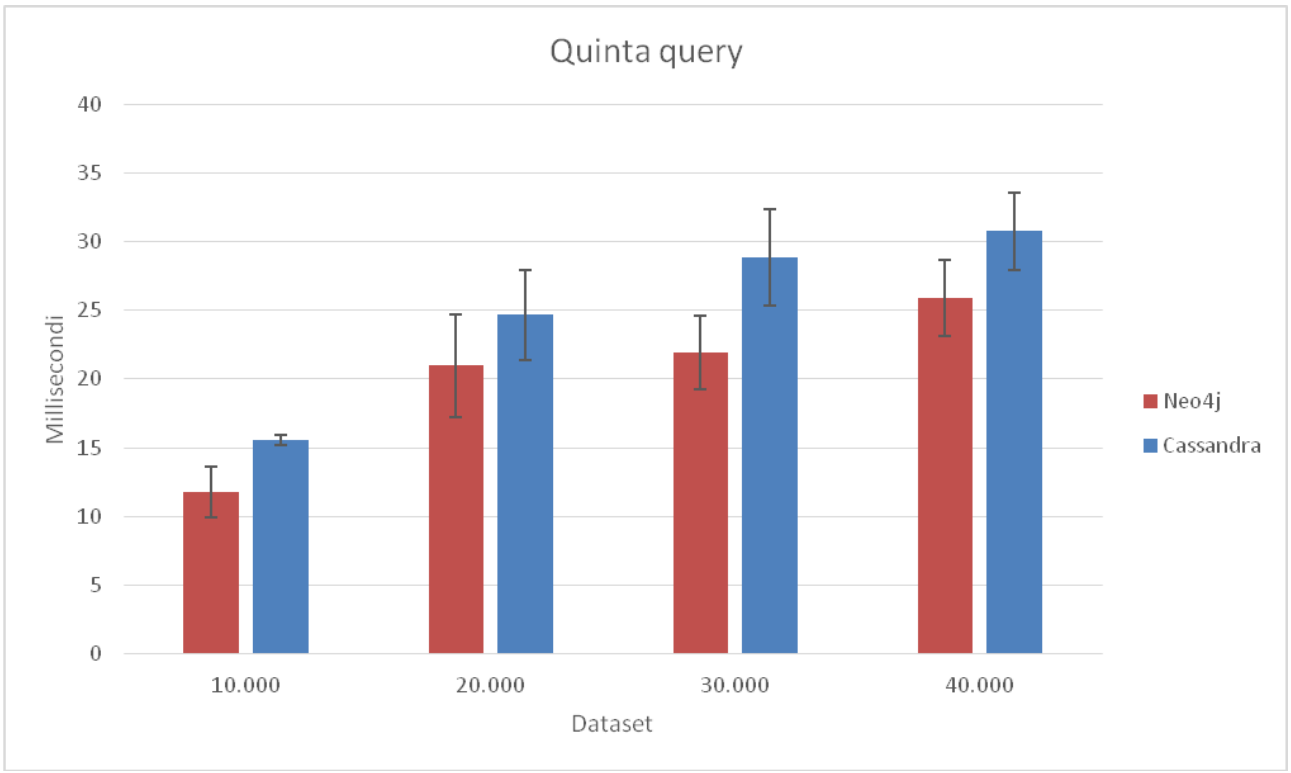
```
#Va eseguita una volta sola
session.execute("""CREATE CUSTOM INDEX ssn_index ON Customer (ssn)
                    USING 'org.apache.cassandra.index.sasi.SASIIndex';""")

def query5():
    session.execute("""SELECT * FROM Customer
                        WHERE claim = 'Car damage' AND name LIKE 'M%'
                        AND ssn LIKE '4%' ALLOW FILTERING;""")

for x in range(31):
    a1 = tempo()
    query5()
    a2 = tempo()
    worksheet.write(row, col, a2-a1)
    row += 1
    print("Il risultato di " ,x+1, " e'" ,a2-a1, " millesecondi\n")
```

Risultati:

Quinta Query	Neo4j 10.000	Cassandra 10.000	Neo4j 20.000	Cassandra 20.000	Neo4j 30.000	Cassandra 30.000	Neo4j 40.000	Cassandra 40.000
Primi Tempi	2444	35	2768	109	2755	151	2764	169
Media	11,74193548	15,5483871	20,96774194	24,67741935	21,9354839	28,83870968	25,90322581	30,77419355
Deviazione Standard	5,223814834	1,131334059	10,56923066	9,2958093117	7,48303666	9,974473142	7,863163767	7,95434422763
95%conf	1,838886898	0,398252129	3,720579769	3,272310082	2,63417799	3,511213272	2,767990307	2,800087646

Istogramma:


Quinta query - Primi tempi
3000
2500
2000
1500
1000
500
0
Millisecondi
10.000
20.000
30.000
40.000
Dataset
Neo4j
Cassandra


Quinta query
40
35
30
25
20
15
10
5
0
Millisecondi
10.000
20.000
30.000
40.000
Dataset
Neo4j
Cassandra

7. CONCLUSIONI:

Facendo riferimento ai test eseguiti sui database con grandezza crescente, rispettivamente: 10.000,20.000,30.000 e 40.000, si può dedurre che la struttura a grafo di Neo4j permette una esecuzione delle query in media più veloce rispetto a Cassandra, anche al crescere della difficoltà della query e delle dimensioni del dataset.

Il dataset da noi utilizzato possiede intrinsicamente un modello di dati a grafo e ciò facilità l'esecuzione delle query su Neo4j, in quanto è un database graph oriented e inoltre utilizza il linguaggio Cypher creato appositamente per le interrogazioni. Esso ammette una gestione di nodi e di relazioni notevolmente più efficiente. Neo4j, quindi, utilizzando nodi e relazioni, permette una facilità di navigazione all'interno del grafo per ricerche semantiche, come nel nostro caso di rilevazione di frodi.

Cassandra, per via della sua struttura column-oriented, in questo caso risulta più lento di Neo4j nella maggior parte delle query anche al crescere delle dimensioni del dataset. Aumentando il grado di complessità della query e la grandezza del dataset, Neo4j risulta in ogni caso quindi più performante rispetto a Cassandra.

Possiamo concludere dicendo che, Neo4j, per questo tipo di dataset che possiede una struttura a grafo intrinseca, fornisce delle prestazioni nettamente miglior rispetto a Cassandra, anche con query più complesse e con dataset di grandezza crescente. Neo4j ci permette, in questo caso, di avere un sistema che possiede una maggiore scalabilità rispetto a un sistema basato su Cassandra.

Neo4j rappresenta una prima scelta per database con numerose relazioni e che richiedono query di una certa complessità, come nel nostro caso di rilevazione delle frodi assicurative. Viceversa, con un database che possiede molti dati e in schemi molto omogenei, Neo4j non rappresenta la soluzione più ottimale.

Cassandra rappresenta la migliore scelta quando possediamo un grossa mole dati in schemi omogenei e abbiamo bisogno di un sistema facilmente scalabile, che quindi può mantenere delle prestazioni ottimali anche al variare della grandezza dei dati.