Università degli studi di Messina

Dipartimento di Scienze matematiche e informatiche,
scienze fisiche e scienze della terra

Corso di Laurea Triennale in INFORMATICA

Progetto di

Basi di dati II

"Carte di credito rubate e rilevamento frodi"

Realizzato da

DE PIETRO BERNARDO

SICLARI SERGIO

Problematica affrontata

Con quasi 3 milioni di reclami dei consumatori nel 2017 negli Stati Uniti, ora è uno scenario abbastanza comune in cui una persona malintenzionata ottiene i dati di una carta di credito e procede a svuotare il conto a cui è collegata.
Per gli analisti delle frodi è fondamentale ridurre i tempi di rilevamento di queste situazioni, che possono portare a gravi perdite finanziare per le organizzazioni.
Come è solito fare, per combattere i criminali, è importante capire come operano.
Le violazioni dei dati personali si verificano sempre più frequentemente, il che significa che le opportunità per gli hacker di mettere le mani sui dati delle carte di credito sono numerose. Il truffatore potrebbe aver acquistato i numeri della carta rubata da un sito web dubbio o semplicemente aver ottenuto i dettagli della carta tramite l'uso di un bancomat o di una pompa di benzina.
All'inizio di ottobre 2018, l'ufficio dello sceriffo di una città californiana ha trovato cinque skimmer di carte di credito in due stazioni di servizio. Hanno identificato più di una dozzina di casi di furto di carte di credito, per un importo di 20.000 dollari. I ladri avevano creato carte di credito false codificate con informazioni rubate dai clienti. Li avevano usati per acquistare più carte regalo di basso valore in diversi supermercati, sapendo che le banche in genere non si insospettiscono delle transazioni di basso valore nei negozi di alimentari.
Ciò che ha portato la polizia a svelare la frode è che per tutti i casi di carte di credito rubate su cui ha indagato, c'era qualcosa in comune. Ad un certo punto, le vittime avevano usato la loro carta di credito per il serbatoio dell'auto alle stazioni di servizio. Da lì, la polizia è stata in grado di identificare i criminali e arrestarli.
Forse leggendo questa storia inizierai a capire cosa hanno in comune le carte di credito rubate e i grafici. Lo scopo del progetto è quello di metterci nei panni di una società di carte di credito che cerca di rilevare le frodi.

Soluzioni considerate

Per analizzare la problematica descritta nel precedente capitolo, sono state utilizzate due soluzioni NoSQL:

- **Neo4j**
- **HBase**

Neo4j

Neo4j è un database NoSQL di tipo Graph. A differenza dei database tradizionali, che organizzano i dati in righe, colonne e tabelle, Neo4j ha una struttura flessibile definita dalle relazioni memorizzate tra i record di dati. Con Neo4j, ogni record di dati, o nodo, memorizza puntatori diretti (archi) a tutti i nodi a cui è connesso, e, sia nodi che archi, possono avere proprietà che li caratterizzano. Poiché Neo4j è progettato attorno a questa semplice e potente ottimizzazione, esegue query con connessioni complesse, ma con un volume di dati basso, più velocemente rispetto ad altri database. Grazie a questa caratteristica, Neo4j è ottimo, ad esempio, nel campo dell'intelligenza artificiale (IA) o nel machine learning, perché ci consente di analizzare i dati direttamente a livello database senza l'overhead di doverli trasferire e gestire a livello applicativo. Inoltre, Neo4j fornisce supporto completo per le transazioni e rispetta le proprietà ACID. La struttura a grafo di Neo4j si mostra estremamente comoda ed efficiente nel trattare strutture come gli alberi estratte, ad esempio, da file XML, filesystem e reti, che, ovviamente, vengono rappresentate con naturalezza da un grafo, poiché sono esse stesse dei grafi. L'esplorazione di queste strutture risulta in genere più veloce rispetto a un database a tabelle, perché la ricerca di nodi in relazione con un certo nodo è un'operazione primitiva e non richiede più passaggi su tabelle diverse. Neo4j ha, inoltre, un'interfaccia web grafica (Neo4j Desktop), tramite cui possiamo manipolare i dati ottenendo anche una visualizzazione in formato JSON o tabellare.

HBase

HBase è un database NoSQL, distribuito, di tipo Column-Oriented, open source, modellato su BigTable di Google e scritto in linguaggio Java. I database di tipo Column-Oriented riescono a gestire in maniera ottimale ampi volumi di dati, ma con complessità non molto elevate. HBase memorizza i dati in tabelle che mantengono il formato chiave-valore. Ogni riga contiene: una chiave univoca detta "row key" e una o più "column families", ovvero famiglie di colonne che contengono colonne figlie con tipi di dato simili tra loro. Ad ogni colonna corrisponde un valore, a cui viene associato un timestamp per gestire il versioning dei dati (utile per le repliche e la consistenza). È possibile aggiungere o rimuovere colonne quando e come necessario, in quanto HBase è schema free. Dal punto di vista fisico, tuttavia, le colonne nulle non vengono memorizzate in modo da

risparmiare spazio. HBase è una soluzione che fa parte della suite di framework di Apache Hadoop; si basa su HDFS (Hadoop Distributed File System), ovvero il file system di Hadoop, e utilizza un middleware di comunicazione chiamato Zookeeper. Quando si ha a che fare con progetti di applicazioni complesse, HBase risulta più efficiente perché, facendo parte di una suite di framework, è facilmente integrabile con altri moduli software per mezzo di altrettanti framework di Hadoop. Inoltre, fornisce API che consentono lo sviluppo in qualsiasi linguaggio di programmazione. Un altro vantaggio di HBase è che non ha necessità di occuparsi in prima persona della replicazione e manutenzione dei dati, in quanto sarà HDFS ad occuparsene. Uno degli svantaggi di HBase è che l'operazione join viene gestito a livello MapReduce. Altri svantaggi sono legati all'indicizzazione e all'ordinamento, che possono essere effettuati nativamente solo sulla row key, ed anche l'assenza delle funzionalità di aggregazione dei dati, che rendono HBase più difficile da interrogare.

Progettazione

Nella progettazione andiamo ad analizzare i dati forniti dal problema. Per capire al meglio la struttura dei dati ci viene fornita una tabella di esempio. La tabella sottostante ci fornisce una serie di transazioni: esse sono state fatte dai criminali che hanno rubato le informazioni delle carte di credito.
Analizzando la tabella d'esempio notiamo che:

- Quando andremo a generare il file .csv dovremo prendere in considerazione che i **customers** possano avere più transazioni;
- Vengono selezionati i record nei quali i **customers** abbiano come **amount** un valore compreso tra 1000 e 2000;
- Tutte le transazioni effettuate devono avere uno **status** che può essere "Disputed" o "Undisputed".

Customer name	Store name	Amount	Transaction Date	Status
Madison	Macys	1790	12/20/2014	Disputed
Madison	Macys	1003	12/20/2014	Disputed
Madison	Macys	1849	12/20/2014	Disputed
Madison	Macys	1816	12/20/2014	Disputed
Marc	Urban Outfitters	1152	5/10/2014	Disputed
Marc	Urban Outfitters	1424	5/10/2014	Disputed
Marc	Urban Outfitters	1732	5/10/2014	Disputed
Marc	Urban Outfitters	1374	5/10/2014	Disputed
Olivia	Apple Store	1149	7/18/2014	Disputed
Olivia	Apple Store	1914	7/18/2014	Disputed
Olivia	Apple Store	1021	7/18/2014	Disputed
Olivia	Apple Store	1925	7/18/2014	Disputed
Paul	RadioShack	1884	4/1/2014	Disputed
Paul	RadioShack	1721	4/1/2014	Disputed
Paul	RadioShack	1415	4/1/2014	Disputed
Paul	RadioShack	1368	4/1/2014	Disputed

A questo punto, definiamo una nuova tabella per descrivere al meglio la struttura che dovrà avere il file .csv che andremo a generare.

Colonna	**Descrizione**	**Esempio**
CUSTOMER_NAME	Contiene il nome e il cognome del customer.	Carlo Vanzina
STORE_NAME	Contiene il nome dello store e la località in cui si trova.	Hard Rock, Roma
AMOUNT	È l'ammontare della spesa effettuata.	1903
TRANSACTION_DATE	È la data in cui è stata effettuata la transazione.	14/07/2021
STATUS	Contiene lo stato della transazione. Può assumere solo due valori: "Disputed" o "Undisputed"	Disputed

Data Model in Neo4j

I primi da prendere in considerazione sono i **customers**. I customers effettuano degli acquisti in determinati **store** (o anche chiamati **merchant**). Quindi ogni customer sarà in relazione con gli store in cui ha effettuato degli acquisti. L'immagine sottostante fornisce una rappresentazione grafica di quanto detto in precedenza.

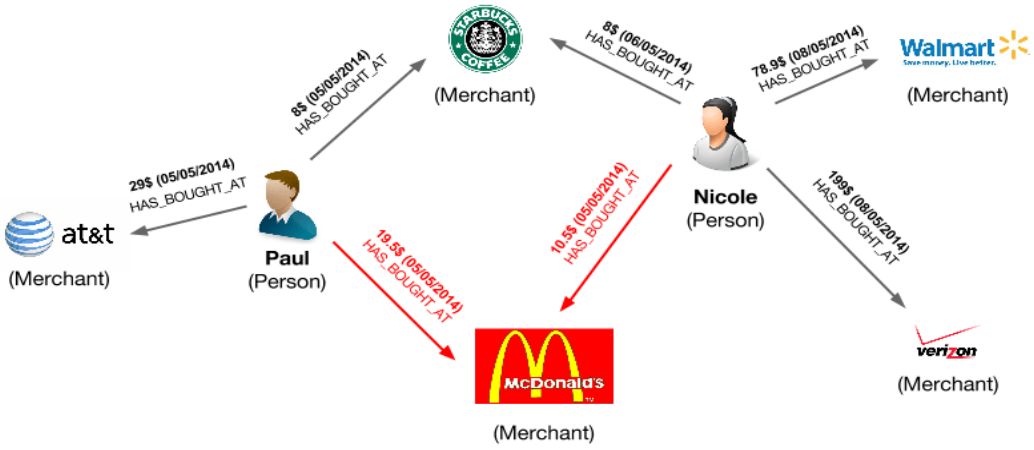


Figura 1: Rappresentazione concettuale d'esempio

Ogni transazione che viene effettuata avrà alcune proprietà:

1. **Date**: ogni transazione avrà la data in cui è stata effettuata;
2. **Amount**: conterrà il valore dell'acquisto effettuato;
3. **Status**: ogni transazione avrà uno stato, in modo da differenziare le transazioni fraudolente da quelle legittime.

Nella figura 2 è presente la rappresentazione di due nodi (customers) e di alcuni store in cui i customers hanno effettuato acquisiti. La relazione è stata chiamata **HAS_BOUGHT** e come detto in precedenza contiene al suo interno degli attributi che sono la data, l'ammontare della spesa e infine lo stato della transazione.

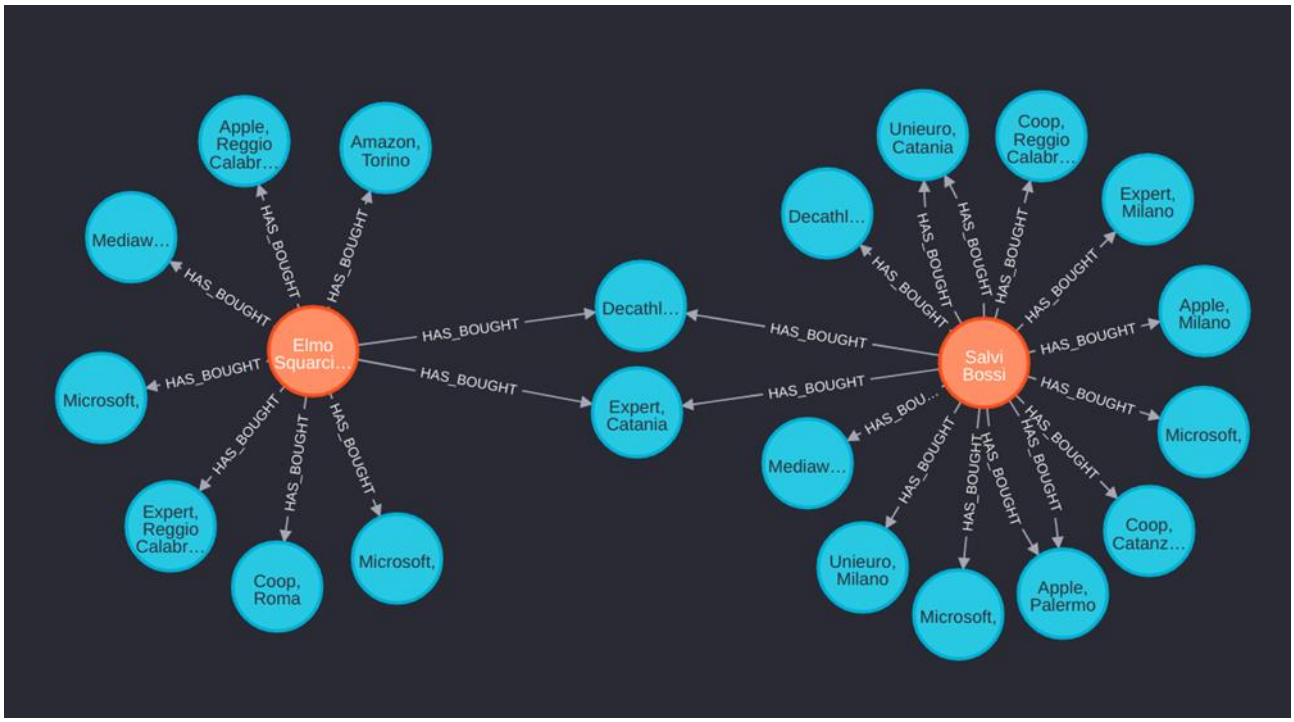


Figura 2: Rappresentazione nodi Neo4j

Data Model in HBase

Il data model è stato progettato ad hoc per ogni query, in quanto, HBase non supporta l'operazione di join. Si è evitata, così, l'elaborazione dei dati a livello di linguaggio di programmazione, ottenendo anche un miglioramento delle prestazioni.

Schema prima query

La figura 3 rappresenta la prima query, essa cerca tutti i **customers** e tutti gli **store** nei quali sono state effettuate delle transazioni con un valore di **amount** maggiore o uguale a 1500. Per ogni transazione viene stampato il nome del customer, il nome dello store in cui è stata effettuata la transazione e il valore della spesa, cioè l'amount.

Row Key	Transaction_Data		
	Customer_name	Store_name	Amount
000001	GIANMARCO MELLINO	AMAZON ROMA	1973

100000	VALERIA PEZZELLA	DECATHLON, REGGIO CALABRIA	1324

Figura 3: Prima tabella di HBase

Schema seconda query

Nella seconda query vengono cercate tutte le transazioni che sono state effettuate con un valore di **amount** maggiore o uguale a 1500 e con data successiva al 2015. Il risultato stamperà il nome del **customer**, il nome dello **store**, l'**amount** e la data della transazione.

Row Key	Transaction_Data			
	Customer_name	Store_name	Amount	Date
000001	GIAMPAOLO CARULLO	COOP, TORINO	1384	2012-01-18

100000	GAETANO BATAGLIA	IKEA, MESSINA	1587	2015-01-03

Figura 4: Seconda tabella HBase

Schema terza query

In figura 5 abbiamo la tabella per la terza query, in cui vengono cercate tutte le transazioni che sono state effettuate con un valore di **amount** maggiore o uguale a 1500, con data successiva al 2015 e con un valore di **status** "Disputed", ovvero transazioni contestate. Il risultato stamperà il nome del **customer**, il nome dello **store**, l'**amount** e la data della transazione.

Row Key	Transaction_Data				
	Customer_name	Store_name	Amount	Date	Status
000001	ANGELICA PIAZZI	IKEA, GENOVA	1774	2018-08-10	UNDISPUTED

100000	LUCIANO GIGLI	APPLE, MILANO	1344	2018-01-09	UNDISPUTED

Figura 5: Terza, Quarta e Quinta tabella HBase

Schema quarta query

Nel caso della quarta query abbiamo la stessa tabella della figura 5. In questo caso vengono cercate tutte le transazioni effettuate in un determinato **store**, con un **amount** maggiore o uguale a 1500, con data successiva al 2015 e con un valore di **status** "Disputed".

Schema quinta query

Anche in questo caso, riutilizziamo la tabella in figura 5. Vengono cercate tutte le transazioni effettuate da un determinato **customer**, in uno specifico **store**, con un **amount** maggiore o uguale a 1500, con data successiva al 2015, con un valore di **status** "Disputed" e infine ordinati per AMOUNT decrescente.

Implementazione

In questa sezione sono analizzati i metodi più importanti usati nel programma. Il linguaggio di programmazione usato è Python e, per il collegamento con i due database, sono state usate le librerie:

- Neo4j Python Driver 4.3: driver ufficiale di Neo4j per Python;
- Happybase 1.2.0: libreria di Python per interagire con HBase. Happybase usa la libreria Python Thrift2 per connettersi e comunicare con HBase usando il Thrift gateway.
- Faker 8.13.1: un pacchetto Python che genera dati falsi. E' stata utilizzata per creare il dataset iniziale, dal quale derivano i dataset di dimensione inferiore.

Neo4j

Di seguito sono presenti tutti i metodi più importanti utilizzati per Neo4J.

Importazione dataset da CSV

CSVImportCustomer

Il metodo **CSVImportCustomer** permette di importare i nodi riguardanti i **customers** in Neo4J. Prende in ingresso la connessione al database e il percorso del file CSV dai cui leggere i record.

```
def CSVImportCustomer(self, pathCSV):
    session = driver.session()
    try:
        session.run("CREATE (c:CUSTOMER_NAME);")
    except:
        print("Errore")
    print("Importing customer...")
    result = session.run("""USING PERIODIC COMMIT 2000
    LOAD CSV WITH HEADERS FROM \"""" + str(pathCSV) + """\" AS line
    FIELDTERMINATOR ','
    MERGE (c:CUSTOMER_NAME {CUSTOMER_NAME: line.CUSTOMER_NAME});""")
    print("added " + str(result.consume().counters.nodes_created) + " nodes.")
    session.close()
```

Dopo aver creato il customer con la label CUSTOMER_NAME, viene letto il file CSV in cui è contenuto il nome del customer e viene inserito un attributo al nodo che chiamiamo CUSTOMER_NAME.

CSVImportStore

Il metodo **CSVImportStore** è analogo al metodo precedente. Si occupa di importare i nodi riguardanti gli **store** in Neo4j.

```
def CSVImportStore(self, pathCSV):
    session = driver.session()
    try:
        session.run("CREATE (s:STORE_NAME);")
    except:
        print("Errore")
    print("Importing store...")
    result = session.run("""USING PERIODIC COMMIT 2000
    LOAD CSV WITH HEADERS FROM \"""" + str(pathCSV) + """\" AS line
    FIELDTERMINATOR ','
    MERGE (s:STORE_NAME {STORE_NAME: line.STORE_NAME});""")
    print("added " + str(result.consume().counters.nodes_created) + " nodes.")
    session.close()
```

Dopo aver creato il customer con la label STORE_NAME, viene letto il file CSV in cui è contenuto il nome del customer e viene inserito un attributo al nodo che chiamiamo STORE_NAME.

CSVImportTransaction

Infine, nel metodo **CSVImportTransaction** vengono messi in relazione i customers con gli store, prende in ingresso la connessione al database e il percorso del file CSV.

```
def CSVImportTransaction(self, pathCSV):
    session = driver.session()
    try:
        print("Importing transactions to store from customer...")
        result = session.run("""USING PERIODIC COMMIT 1000
        LOAD CSV WITH HEADERS FROM \"""+ str(pathCSV) + """\" AS line
        FIELDTERMINATOR ','
        MATCH (c:CUSTOMER_NAME {CUSTOMER_NAME: line.CUSTOMER_NAME})
        MATCH (s:STORE_NAME {STORE_NAME: line.STORE_NAME})
        CREATE (c)-[:HAS_BOUGHT {
            TRANSACTION_ID: line.ID,
            DATE: line.TRANSACTION_DATE,
            AMOUNT: line.AMOUNT,
            STATUS: line.STATUS
        }]->(s);""")
        print("added " + str(result.consume().counters.relationships_cre-
ated) + " relations.")
    except:
        print('errore')
    session.close()
```

La transazione avrà come attributi la data di transazione, l'ammontare della spesa e stato in cui essa si trova.
Ogni riga rappresenta una transazione, di conseguenza è legata ad un *customer* e ad uno *store*. Viene effettuato un match del nodo CUSTOMER_NAME e del nodo STORE_NAME e si crea una relazione identificata dalla label "HAS_BOUGHT".
Ciascuna relazione avrà le seguenti proprietà:

- TRANSACTION_ID
- DATE
- AMOUNT
- STATUS

E' possibile visualizzare il file integrale al seguente link: insert_data.py.

Esecuzione queries

Verranno eseguite le seguenti queries:

1. La prima query ci permette di estrapolare dal database i dati relativi a tutti i **CUSTOMERS** e a tutti gli **STORES** nei quali le transazioni effettuate hanno in **AMOUNT** un valore maggiore o uguale a 1500.
2. La seconda query ci permette di estrapolare dal database i dati relativi a tutti i **CUSTOMERS** e a tutti gli **STORES** nei quali le transazioni effettuate hanno in **AMOUNT** un valore maggiore o uguale a 1500 e in **DATE** una data successiva al 2015-12-31.
3. La terza query ci permette di estrapolare dal database i dati relativi a tutti i **CUSTOMERS** e a tutti gli **STORES** nei quali le transazioni effettuate hanno in **AMOUNT** un valore maggiore o uguale a 1500, in **DATE** una data successiva al 2015-12-31 e in **STATUS** il valore "Disputed".
4. La quarta query ci permette di estrapolare dal database i dati relativi a tutti i **CUSTOMERS**, dove le transazioni effettuate hanno in **AMOUNT** un valore maggiore o uguale a 1500, in **DATE** una data successiva al 2015-12-31, in **STATUS** il valore "Disputed" e filtrando per uno specifico **STORE_NAME**.
5. La quinta query ci permette di estrapolare dal database i dati relativi alle transazioni effettuate che hanno in **AMOUNT** un valore maggiore o uguale a 1500, in **DATE** una data successiva al 2015-12-31, in **STATUS** il valore "Disputed" e filtrando per uno specifico **STORE_NAME** ed uno specifico **CUSTOMER_NAME**, una volta prelevati i dati, vengono stampati in ordine decrescente in base al valore di **AMOUNT**.

query.py

Contiene le funzioni che eseguono le cinque queries esplicate in precedenza. Ai fini di calcolare correttamente i tempi di esecuzione, ciascuna query viene eseguita esattamente 31 volte.

Le funzioni sono molto somiglianti tra loro, in sostanza il funzionamento di ciascuna funzione è il seguente:

- Quando viene richiamata, viene definita una variabile "start" che contiene un timestamp generato tramite la funzione time( ).
- Successivamente, avviamo la connessione con il database e inviamo la query da eseguire.
- Infine, viene definita una variabile "exec_time", che conterrà il tempo di esecuzione, calcolato con una sottrazione tra un ulteriore timestamp generato al termine della query e la variabile start iniziale, questo valore viene moltiplicato per 1000 ottenendo cosi il tempo in millisecondi.

Di seguito, le funzioni per ciascuna query:

query1

```
def query1():
    start = time.time()
    with driver.session() as session:
        session.run("""
        MATCH (c:CUSTOMER_NAME)-[r:HAS_BOUGHT]->(s:STORE_NAME)
        WHERE r.AMOUNT >= '1500'
        RETURN c.CUSTOMER_NAME, s.STORE_NAME, r.AMOUNT""")
    exec_time = "%.2f" % round((time.time()-start) * 1000, 2)
    return exec_time
```

query2

```
def query2():
    start = time.time()
    with driver.session() as session:
        session.run("""
        MATCH (c:CUSTOMER_NAME)-[r:HAS_BOUGHT]->(s:STORE_NAME)
        WHERE r.AMOUNT >= '1500' AND r.DATE > '2015-12-31'
        RETURN c.CUSTOMER_NAME, s.STORE_NAME, r.AMOUNT""")
    exec_time = "%.2f" % round((time.time()-start) * 1000, 2)
    return exec_time
```

query3

```
def query3():
    start = time.time()
    with driver.session() as session:
        session.run("""
        MATCH (c:CUSTOMER_NAME)-[r:HAS_BOUGHT]->(s:STORE_NAME)
        WHERE r.AMOUNT >= '1500' AND r.DATE > '2015-12-31' AND r.STATUS = 'Disput-
ed'
        RETURN c.CUSTOMER_NAME, s.STORE_NAME, r.AMOUNT""")
    exec_time = "%.2f" % round((time.time()-start) * 1000, 2)
    return exec_time
```

query4

```
def query4():
    start = time.time()
    with driver.session() as session:
        session.run("""
        MATCH (c:CUSTOMER_NAME)-[r:HAS_BOUGHT]->(s:STORE_NAME {STORE_NAME: 'Pan-
zera SPA'})
        WHERE r.AMOUNT >= '1500' AND r.DATE > '2015-12-31' AND r.STATUS = 'Disput-
ed'
        RETURN c.CUSTOMER_NAME, s.STORE_NAME, r.AMOUNT, r.DATE, r.STATUS""")
    exec_time = "%.2f" % round((time.time()-start) * 1000, 2)
    return exec_time
```

query5

```
def query5():
    start = time.time()
    with driver.session() as session:
        session.run("""
        MATCH (c:CUSTOMER_NAME {CUSTOMER_NAME: 'Amanda Pietrangeli'})-
[r:HAS_BOUGHT]->(s:STORE_NAME {STORE_NAME: 'Panzera SPA'})
        WHERE r.AMOUNT >= '1500' AND r.DATE > '2015-12-31' AND r.STATUS = 'Disput-
ed'
        RETURN c.CUSTOMER_NAME, s.STORE_NAME, r.AMOUNT, r.DATE, r.STATUS
        ORDER BY r.AMOUNT DESC""")
    exec_time = "%.2f" % round((time.time()-start) * 1000, 2)
    return exec_time
```

HBase

Di seguito sono riportati i metodi utilizzati per la gestione dei dataset in HBase.

Connessione al database

La libreria Happybase, ci permette in modo semplice e veloce di aprire o chiudere le connessioni con HBase:

```
def connection_db():
    try:
        conn = happybase.Connection()
        conn.open()
        return conn
    except Exception as e:
        print("Error: " + e)
```

Definizione delle tabelle

I seguenti metodi hanno il compito di generare esattamente tre tabelle:

- TRANSACTION_Q1: contiene la column family Transaction_Data, con le column CUSTOMER_NAME, STORE_NAME e AMOUNT
- TRANSACTION_Q2: contiene la column family Transaction_Data, con le column CUSTOMER_NAME, STORE_NAME, AMOUNT e DATE
- TRANSACTION_Q3: contiene la column family Transaction_Data, con le column CUSTOMER_NAME, STORE_NAME, AMOUNT, DATE e STATUS

Le tabelle sono ottimizzate per ogni query, infatti:

- sulla tabella Q1 verrà eseguita la funzione query1( )
- sulla tabella Q2 verrà eseguita la funzione query2( )
- sulla tabella Q3, verranno eseguite query3( ), query4( ) e query5( )

```
def create_table_q1():
    try:
        conn = connection_db()
        print("\n#####################################")
        print("# Creating table TRANSACTION_Q1_100 #")
        print("#####################################\n")
        conn.create_table('TRANSACTION_Q1_100', { 'Transaction_Data':dict() })
        print("Table created.")
```

```
def create_table_q2():
    try:
        conn = connection_db()
        print("\n#####################################")
        print("# Creating table TRANSACTION_Q2_100 #")
        print("#####################################\n")
        conn.create_table('TRANSACTION_Q2_100', { 'Transaction_Data':dict() })
        print("Table created")

def create_table_q3():
    try:
        conn = connection_db()
        print("\n#####################################")
        print("# Creating table TRANSACTION_Q3_100 #")
        print("#####################################\n")
        conn.create_table('TRANSACTION_Q3_100', { 'Transaction_Data':dict() })
        print("Table created")
```

Importazione dataset da CSV

I metodi push_data, permettono di creare e popolare le column all'interno della column family Transaction_Data.

Prima di tutto viene avviata la connessione al database, successivamente definiamo la variabile "table" che recupera la tabella sulla quale memorizzare i dati e la variabile "csvfile" che accede al dataset in modalità read. Infine, definiamo un'ulteriore variabile "csvreader" che legge tutte le righe presenti nel dataset e le inseriamo ciclicamente nella tabella interessata. Una volta terminato il procedimento viene chiusa la connessione.

Query 1:

```
def push_data_q1():
    try:
        conn = connection_db()
        print("\n#####################################################")
        print("# Importing dataset into table TRANSACTION_Q1_100 #")
        print("#####################################################\n")
        table = conn.table('TRANSACTION_Q1_100')
        csvfile = open("dataset_100.csv", "r")
        csvreader = csv.reader(csvfile)
        for row in csvreader:
            table.put(row[0], {
            'Transaction_Data:CUSTOMER_NAME':row[1],
            'Transaction_Data:STORE_NAME':row[2],
            'Transaction_Data:AMOUNT':row[3]
            })
```

Query 2:

```
def push_data_q2():
    try:
        conn = connection_db()
        print("\n######################################################")
        print("# Importing dataset into table TRANSACTION_Q2_100 #")
        print("######################################################\n")
        table = conn.table('TRANSACTION_Q2_100')
        csvfile = open("dataset_100.csv", "r")
        csvreader = csv.reader(csvfile)
        for row in csvreader:
            table.put(row[0], {
            'Transaction_Data:CUSTOMER_NAME':row[1],
            'Transaction_Data:STORE_NAME':row[2],
            'Transaction_Data:AMOUNT':row[3],
            'Transaction_Data:DATE':row[4]
            })
```

Queries 3, 4 e 5:

```
def push_data_q3():
    try:
        conn = connection_db()
        print("\n######################################################")
        print("# Importing dataset into table TRANSACTION_Q3_100 #")
        print("######################################################\n")
        table = conn.table('TRANSACTION_Q3_100')
        csvfile = open("dataset_100.csv", "r")
        csvreader = csv.reader(csvfile)
        for row in csvreader:
            table.put(row[0], {
            'Transaction_Data:CUSTOMER_NAME':row[1],
            'Transaction_Data:STORE_NAME':row[2],
            'Transaction_Data:AMOUNT':row[3],
            'Transaction_Data:DATE':row[4],
            'Transaction_Data:STATUS':row[5]
            })
```

Esecuzione queries

Le queries da eseguire sul sistema HBase sono identiche a quelle eseguite su Neo4j, in modo da ottenere un confronto reale sui tempi di esecuzione.

La seguente porzione di codice genera un file csv sotto la cartella “results”, sul quale verranno scritti i risultati in millisecondi della query interessata, eseguita esattamente 31 volte.

```
with open('results/time_q5_100.csv', mode='w', newline='') as csv_file:
    fieldnames =['Iterazione', 'Tempo']

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


    writer.writeheader()
    for i in range(31):
        temp = query5()
        writer.writerow({
            'Iterazione': i+1,
            'Tempo': temp,
        })
```

Per quanto riguarda l’esecuzione delle queries, abbiamo delle funzioni simili.

Utilizziamo il metodo time( ) per generare un timestamp immediatamente prima alla connessione al database, e lo salviamo nella variabile “start”. Avviamo la connessione al database, e recuperiamo nella variabile “table” la tabella interessata.

Definiamo una o più variabili, in base alla query da eseguire, che conterrà le stringhe con i filtri da applicare. Le stringhe verranno passate come parametri alla funzione scan( ), dopo essere state concatenate.

Infine, salviamo all’interno della variabile “exec_time” il tempo di esecuzione della query sottraendo ad un timestamp generato sul momento, il valore di “start”.

Al termine della procedura viene chiusa la connessione al database.

Query 1:

```
def query1():
    start = time.time()
    connection = happybase.Connection()
    table = connection.table("TRANSACTION_Q1_100")
    filter = "SingleColumnValueFilter('Transaction_Data', 'AMOUNT', >=, 'bi-
nary:1500')"
    table.scan(filter=filter)
    exec_time = "%.2f" % round((time.time()-start) * 1000, 2)
    print(str(exec_time) + "ms")
    connection.close()
    return exec_time
```

Query 2:

```
def query2():
    start = time.time()
    connection = happybase.Connection()
    table = connection.table("TRANSACTION_Q2_100")
    filter_a = "SingleColumnValueFilter('Transaction_Data', 'AMOUNT', >=,
               'binary:1500')"
    filter_b = "SingleColumnValueFilter('Transaction_Data', 'DATE', >,
               'binary:2015-12-31')"
    filter = filter_a + " AND " + filter_b
    table.scan(filter=filter)
    exec_time = "%.2f" % round((time.time()-start) * 1000, 2)
    print(str(exec_time) + "ms")
    connection.close()
    return exec_time
```

Query 3:

```
def query3():
    start = time.time()
    connection = happybase.Connection()
    table = connection.table("TRANSACTION_Q3_100")
    filter_a = "SingleColumnValueFilter('Transaction_Data', 'AMOUNT', >=,
               'binary:1500')"
    filter_b = "SingleColumnValueFilter('Transaction_Data', 'DATE', >,
               'binary:2015-12-31')"
    filter_c = "SingleColumnValueFilter('Transaction_Data', 'STATUS', =,
               'binary:Disputed')"
    filter = filter_a + " AND " + filter_b + " AND " + filter_c
    table.scan(filter=filter)
    exec_time = "%.2f" % round((time.time()-start) * 1000, 2)
    print(str(exec_time) + "ms")
    connection.close()
    return exec_time
```

Query 4:

```
def query4():
    start = time.time()
    connection = happybase.Connection()
    table = connection.table("TRANSACTION_Q3_100")
    filter_a = "SingleColumnValueFilter('Transaction_Data', 'AMOUNT', >=, 'bi-
nary:1500')"
    filter_b = "SingleColumnValueFilter('Transaction_Data', 'DATE', >, 'bi-
nary:2015-12-31')"
    filter_c = "SingleColumnValueFilter('Transaction_Data', 'STATUS', =, 'bi-
nary:Disputed')"
    filter_d = "SingleColumnValueFilter('Transaction_Data', 'STORE_NAME', =, 'bi-
nary:Panzera SPA')"
    filter = filter_a + " AND " + filter_b + " AND " + filter_c + " AND " + fil-
ter_d
    table.scan(filter=filter)
    exec_time = "%.2f" % round((time.time()-start) * 1000, 2)
    print(str(exec_time) + "ms")
    connection.close()
    return exec_time
```

Query 5:

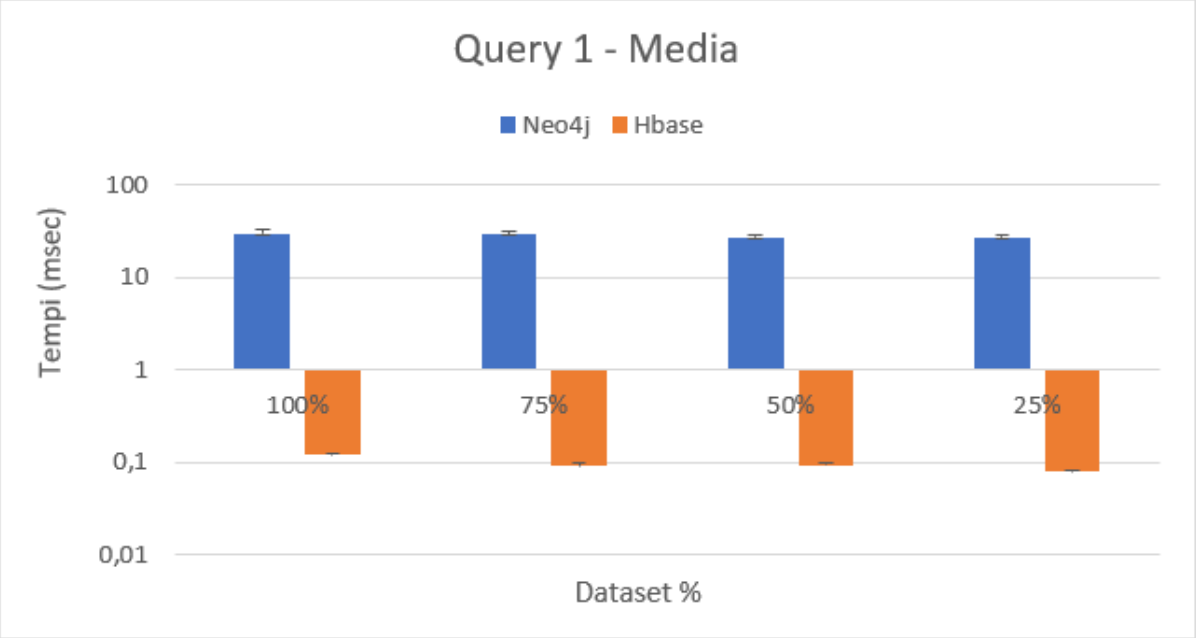
```
def query5():
    start = time.time()
    connection = happybase.Connection()
    table = connection.table("TRANSACTION_Q3_100")
    table_dict = {}
    filter_a = "SingleColumnValueFilter('Transaction_Data', 'AMOUNT', >=, 'bi-
nary:1500')"
    filter_b = "SingleColumnValueFilter('Transaction_Data', 'DATE', >, 'bi-
nary:2015-12-31')"
    filter_c = "SingleColumnValueFilter('Transaction_Data', 'STATUS', =, 'bi-
nary:Disputed')"
    filter_d = "SingleColumnValueFilter('Transaction_Data', 'STORE_NAME', =, 'bi-
nary:Panzera SPA')"
    filter_e = "SingleColumnValueFilter('Transaction_Data', 'CUS-
TOMER_NAME', =, 'binary:Amanda Pietrangeli')"
    filter = filter_a + " AND " + filter_b + " AND " + filter_c + " AND " + fil-
ter_d + " AND " + filter_e
    for (key, data) in table.scan(filter=filter):
        table_dict[key.decode('utf-8')] = int(data[b'Transaction_Data:AMOUNT'].de-
code('utf-8'))
    table_dict = sorted(table_dict.items(), key=lambda x: x[1], reverse=True)
    exec_time = "%.2f" % round((time.time()-start) * 1000, 2)
    print(str(exec_time) + "ms")
    connection.close()
    return exec_time
```

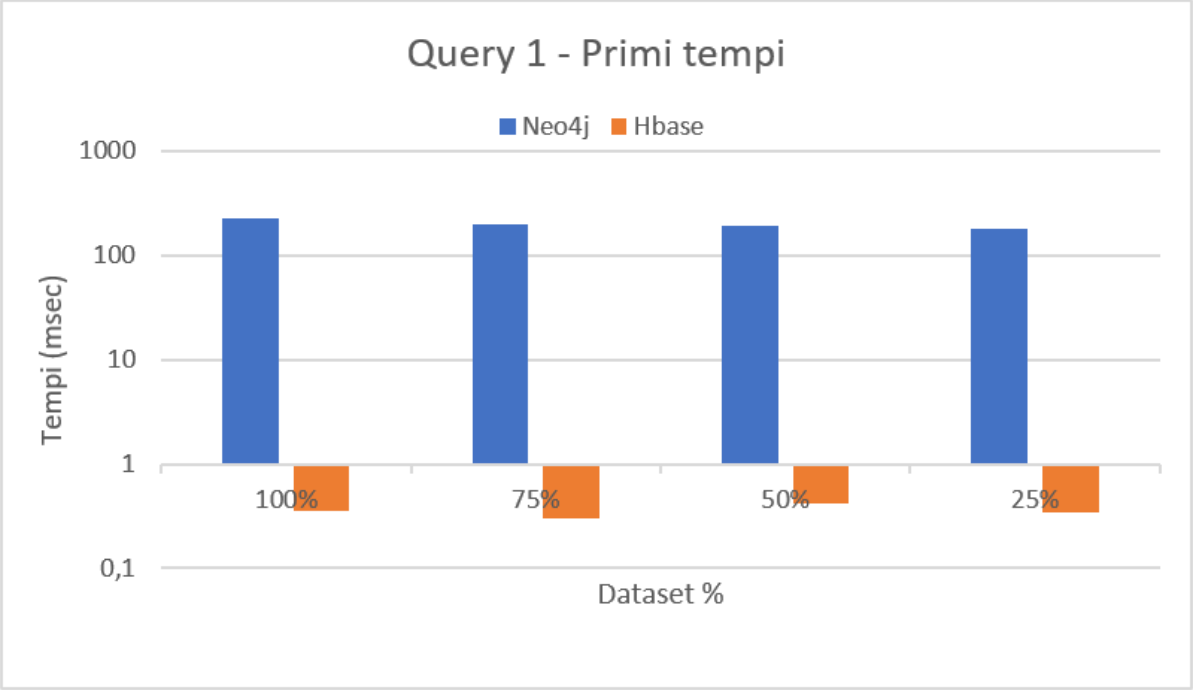
In quest'ultima query, poiché HBase non possiede funzionalità di ordinamento, abbiamo predisposto a livello di linguaggio di programmazione l'ordinamento dei dati, in modo decrescente in base al valore di AMOUNT.

Esperimenti

Gli esperimenti effettuati mirano a confrontare le prestazioni di Neo4j e HBase riguardo i tempi di esecuzione delle query al crescere del dataset. Sono stati considerati quattro dataset di dimensioni crescenti.
Il partizionamento è stato effettuato sui file csv considerando il 25%, il 50%, il 75% e il 100% delle righe totali di ognuno.
Nel valutare i tempi si è tenuto in considerazione che la prima esecuzione di una query impiega tendenzialmente più tempo rispetto a quelle successive, poiché sia in Neo4j che in HBase è presente un sistema di caching che permette di memorizzare dati o metadati in modo da ottenere più velocemente i risultati nelle esecuzioni successive.
Nel realizzare i benchmark sono stati cancellati i database e le tabelle ad ogni cambio di dataset, in modo da essere certi che il primo tempo sia quello relativo all'assenza di cache.
Per ogni query è stata inserita una tabella contenente il primo tempo di esecuzione, la media dei tempi delle 30 esecuzioni successive e l'errore standard per calcolare l'intervallo di confidenza al 95%. Inoltre, per ogni query sono anche presenti un istogramma che mostra l'andamento della media dei tempi, con il relativo intervallo di confidenza, e un istogramma che mostra l'andamento del primo tempo di esecuzione, al diminuire del dataset.

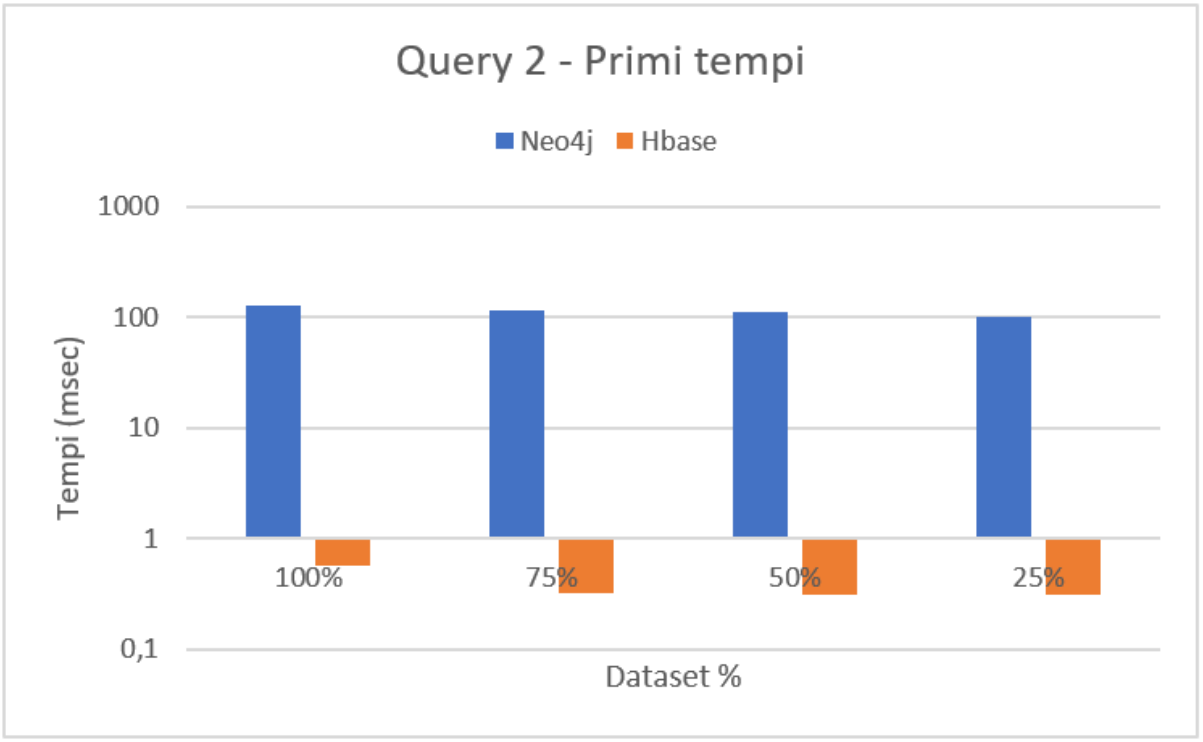
Query 1

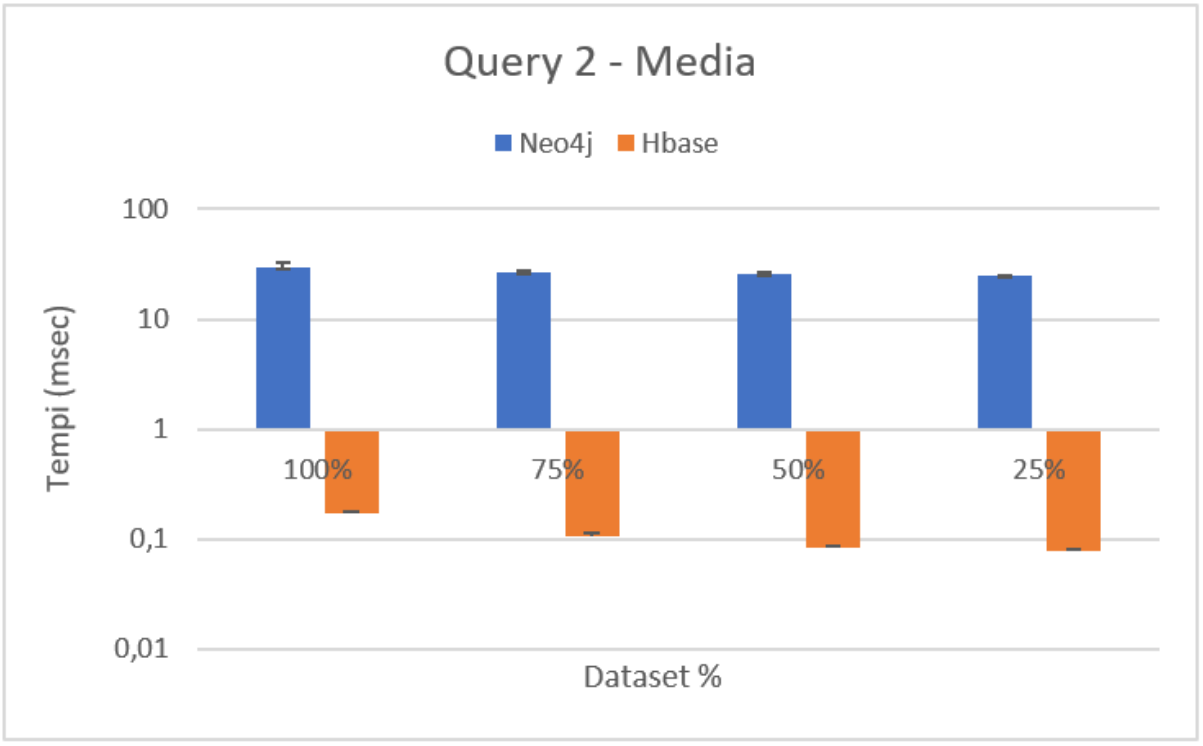




Query1	Dataset 100%		Dataset 75%		Dataset 50%		Dataset 25%	
	Neo4j	Hbase	Neo4j	Hbase	Neo4j	Hbase	Neo4j	Hbase
Primi tempi	228,97	0,36	196,93	0,3	189,36	0,42	181,71	0,34
Media	30,12	0,124	29,48	0,091	27,44	0,094	27,33	0,08
Errore	2,58	0,0032	2,14	0,0089	1,33	0,0073	1,69	0,0025
Dev. Standard	7,20	0,0092	5,98	0,0252	3,72	0,0206	4,73	0,0071
Confidenza 95%	2,58	0,0032	2,14	0,0089	1,33	0,0073	1,69	0,0025

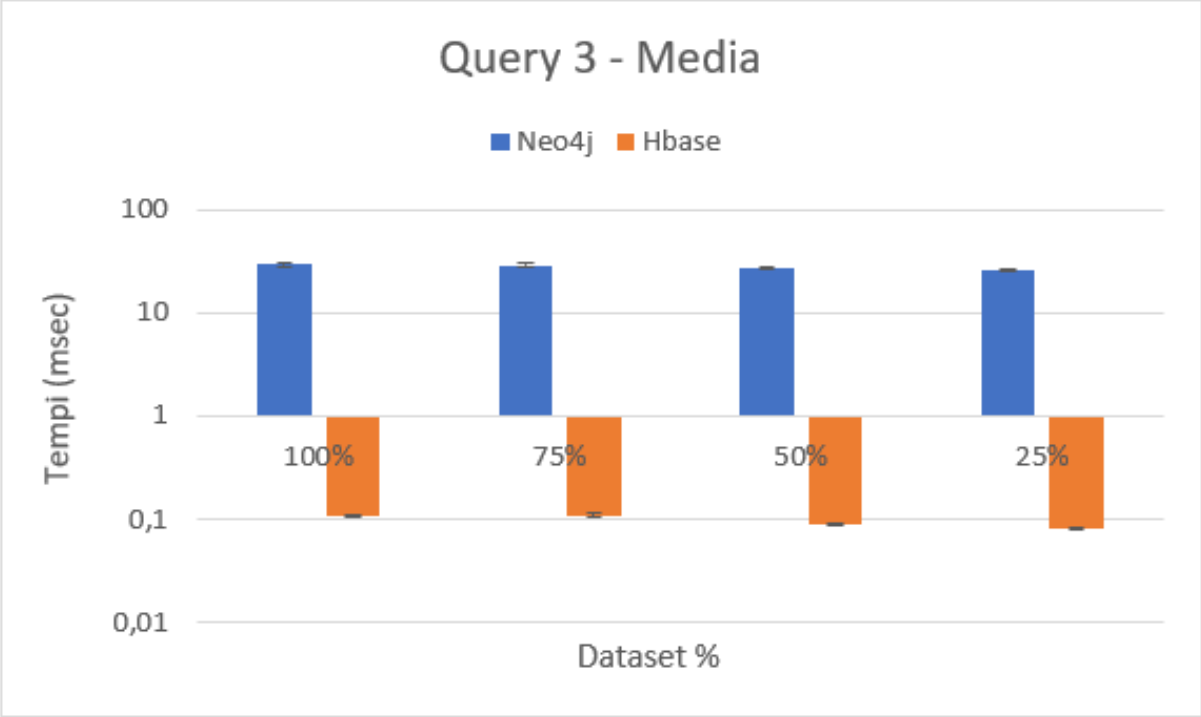
Query 2

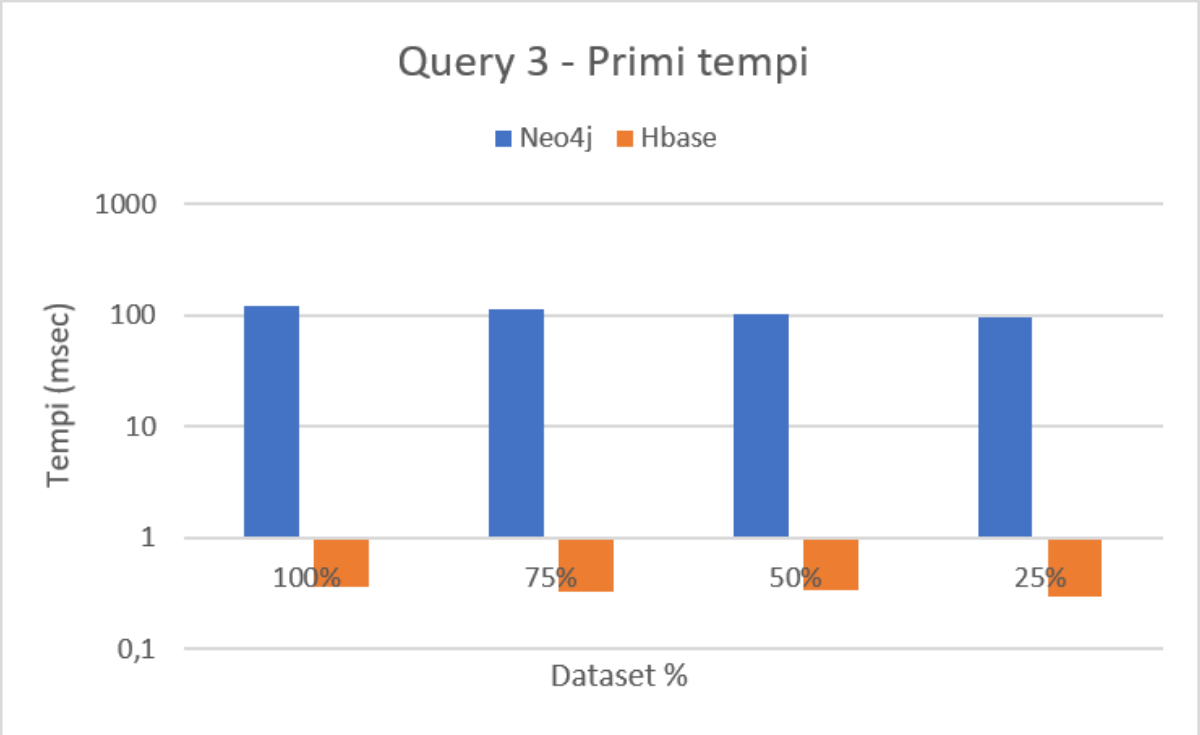




Query 2	Dataset 100%		Dataset 75%		Dataset 50%		Dataset 25%	
	Neo4j	Hbase	Neo4j	Hbase	Neo4j	Hbase	Neo4j	Hbase
Primi tempi	129,03	0,57	115,85	0,32	109,38	0,31	102,03	0,31
Media	30,04	0,172	27,04	0,108	25,80	0,083	24,74	0,077
Errore	2,59	0,0069	1,01	0,0056	0,45	0,0031	0,17	0,029
Dev. Standard	7,23	0,0192	2,82	0,0157	1,27	0,0087	0,46	0,0082
Confidenza 95%	2,59	0,0069	1,01	0,0056	0,45	0,0031	0,17	0,0029

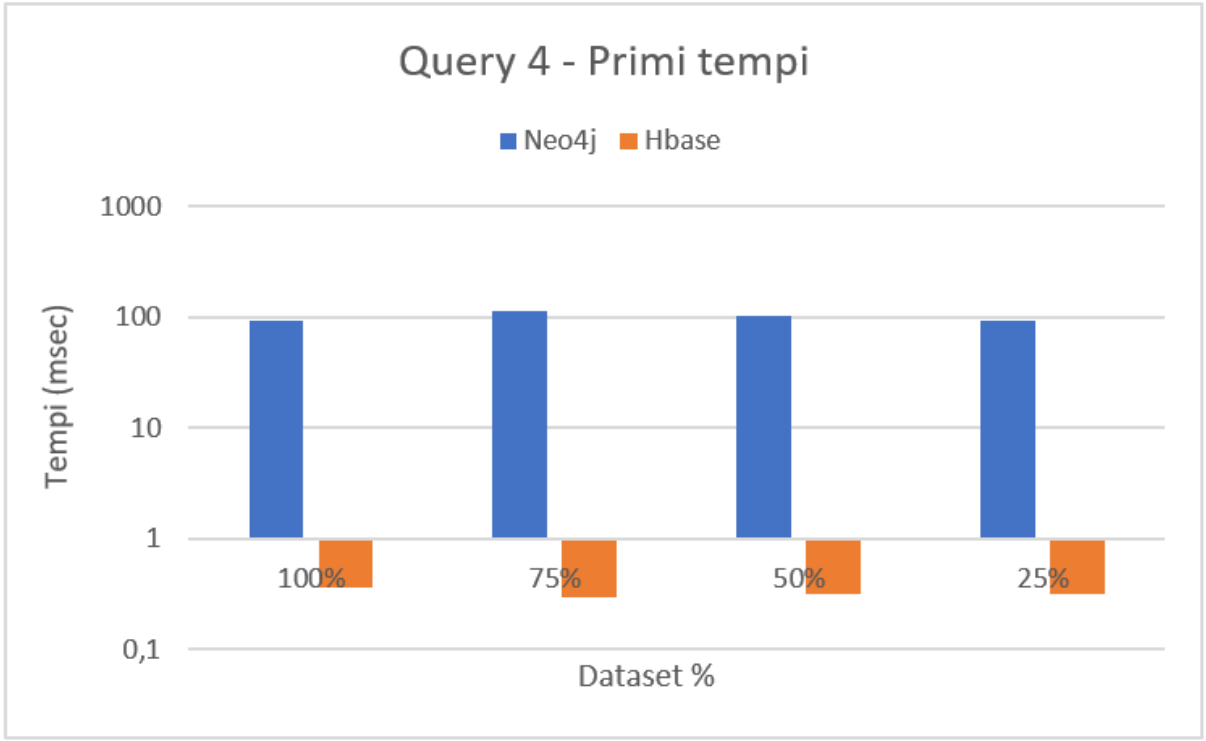
Query 3

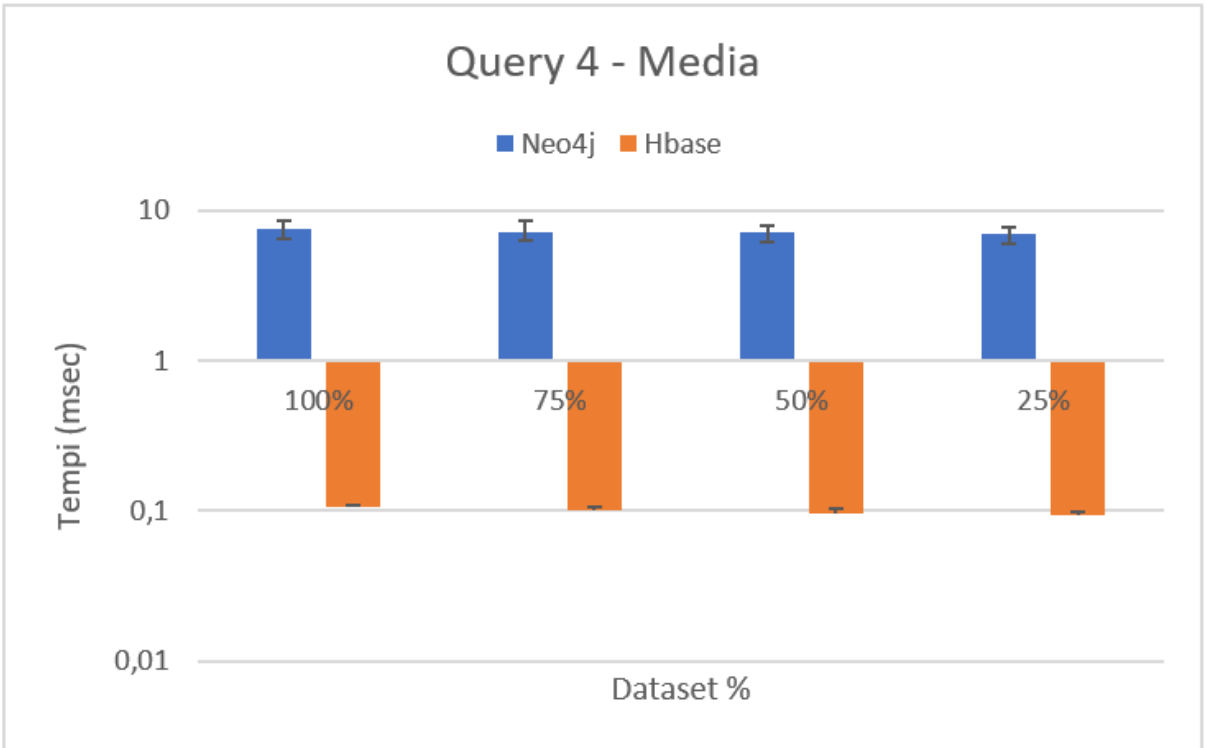




Query 3	Dataset 100%		Dataset 75%		Dataset 50%		Dataset 25%	
	Neo4j	Hbase	Neo4j	Hbase	Neo4j	Hbase	Neo4j	Hbase
Primi tempi	120,2	0,36	114,54	0,33	101,41	0,34	95,21	0,3
Media	29,19	0,106	28,40	0,108	26,82	0,088	26,13	0,082
Errore	1,66	0,005	1,7	0,009	1,3	0,004	0,56	0,004
Dev. Standard	4,64	0,014	4,75	0,026	3,63	0,012	1,55	0,01
Confidenza 95%	1,66	0,005	1,70	0,009	1,30	0,004	0,56	0,004

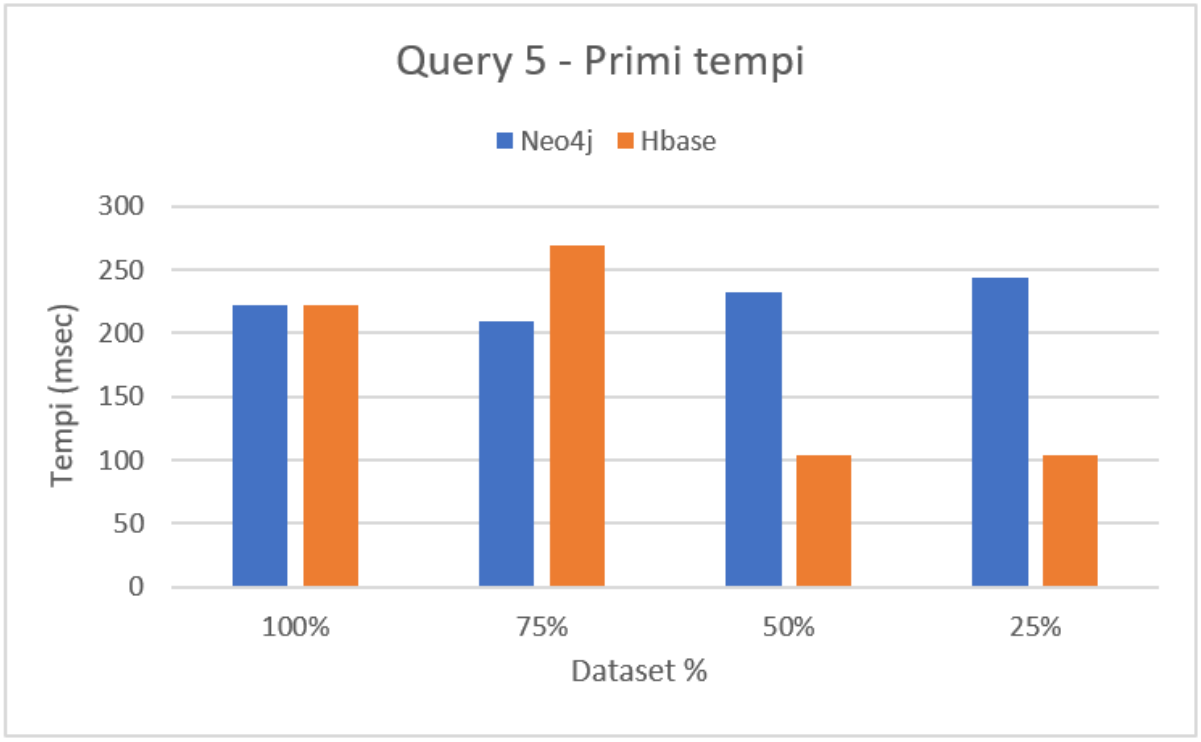
Query 4

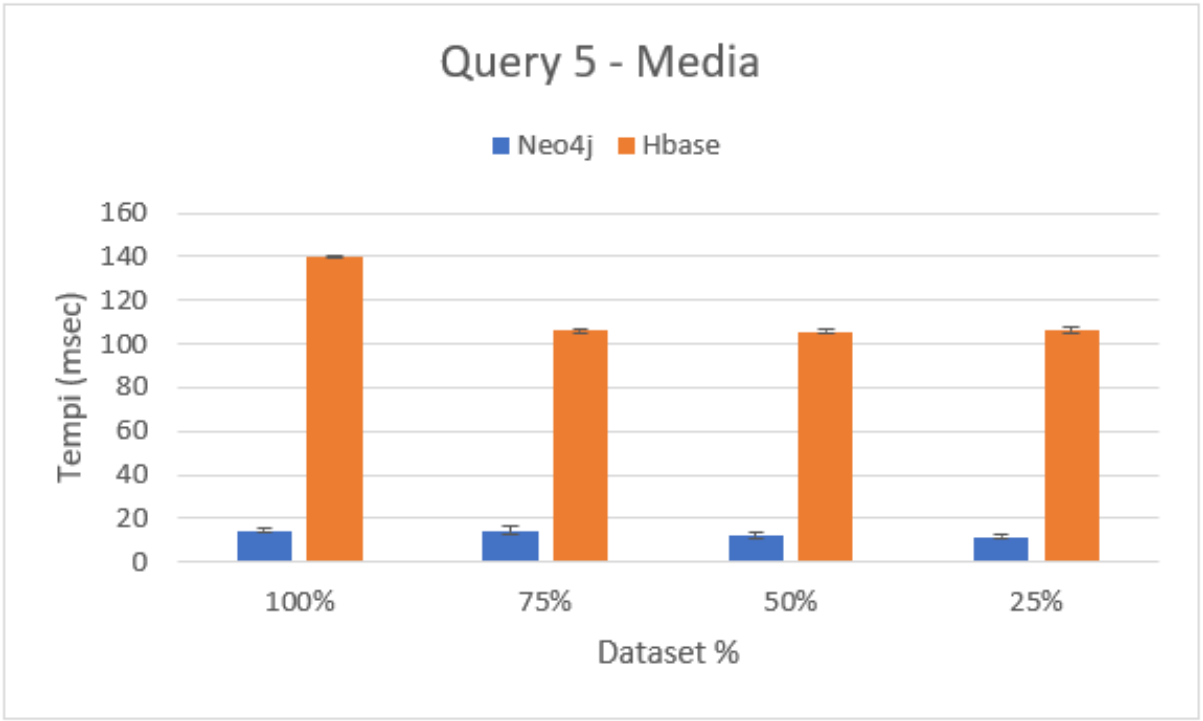




Query 4	Dataset 100%		Dataset 75%		Dataset 50%		Dataset 25%	
	Neo4j	Hbase	Neo4j	Hbase	Neo4j	Hbase	Neo4j	Hbase
Primi tempi	92,8	0,36	114,21	0,3	102,75	0,32	91,63	0,32
Media	7,53	0,105	7,23	0,101	7,09	0,097	7,03	0,094
Errore	0,97	0,0039	1,2	0,006	0,83	0,0062	0,59	0,0053
Dev. Standard	2,71	0,0109	3,35	0,0169	2,33	0,0172	1,66	0,0147
Confidenza 95%	0,97	0,0039	1,20	0,0060	0,83	0,0062	0,59	0,0053

Query 5





Query 5	Dataset 100%		Dataset 75%		Dataset 50%		Dataset 25%	
	Neo4j	Hbase	Neo4j	Hbase	Neo4j	Hbase	Neo4j	Hbase
Primi tempi	221,71	222,69	209,77	269,6	231,8	104,14	243,44	104,5
Media	14,34	140,14	13,75	106,10	12,14	105,46	11,40	106,03
Errore	1,26	0,75	2,79	1,05	1,51	0,94	1,53	1,59
Dev. Standard	3,51	2,10	7,79	2,93	4,23	2,62	4,28	4,45
Confidenza 95%	1,26	0,75	2,79	1,05	1,51	0,94	1,53	1,59

Conclusioni

Dagli esperimenti effettuati risulta che, quasi in tutti i casi, HBase risponde molto più velocemente rispetto a Neo4j.

HBase è un database di tipo *Column Oriented* e riesce a gestire meglio ampi volumi di dati con complessità relativamente bassa.

Neo4j, invece, è un database di tipo *Graph* e riesce a gestire meglio dati con complessità elevata ma con un volume di dati minore.

HBase risulta più efficiente, principalmente, perché il data model è stato ottimizzato per l'esecuzione di ogni query. Questo ha permesso che venissero memorizzate nelle tabelle solo i dati richiesti dalle query e che non venissero effettuati join durante l'esecuzione delle stesse.

In Neo4j, invece, le query sono state eseguite sempre sullo stesso data model che include tutti i nodi, le relazioni e le proprietà recuperate dai file csv.

Un'eccezione si ha solo nell'ultima query, che presenta una funzione di ordinamento dei dati. HBase, infatti, non presenta alcuna funzionalità riguardante l'ordinamento e, nonostante nella tabella fossero presenti solo i dati risultanti dalla query, è stato necessario effettuare l'ordinamento con un linguaggio di programmazione, dopo averli recuperati.

Neo4j, di contro, esegue l'ordinamento direttamente a livello di database.

In conclusione, è possibile affermare che HBase si comporta meglio, ma per poterne apprezzare i vantaggi prestazionali, è necessaria un'importante fase di progettazione del data model.

Neo4j, al contrario, non richiede un'importante fase di progettazione del data model poiché supporta già a livello di database funzionalità quali: le aggregazioni, l'ordinamento e le relazioni fra i nodi.