Università degli Studi di Messina

Dipartimento MIFT

Tesina Basi di Dati 2

"Usare le chiamate telefoniche per indentificare dei criminali"

Studente
Marco Puliafito

Docente
Prof. Antonio Celesti

Problematica affrontata

Lo scopo di questo progetto è effettuare un confronto in termini di prestazioni tra due diversi DBMS NoSQL, mettendo a confronto una problematica reale comune. Andremo a valutare i tempi di risposta di ciascun DBMS, al variare della dimensione del dataset e andando a considerare delle query di difficoltà crescente,al fine di stabilire quale delle due soluzioni prese in esame ci permette di avere delle migliori performance.

La problematica che abbiamo affrontato riguarda l'identificazione di un potenziale criminale attraverso l'analisi delle chiamate telefoniche effettuate nei pressi dell'area incriminata dove è avvenuto il reato, mediante tecniche realmente utilizzate dagli investigatori di polizia per dare una svolta alle indagini. Ogni operatore telefonico è, infatti, autorizzato a raccogliere dati sulle chiamate effettuate dai suoi utenti e questi possono essere richiesti dalle forze dell'ordine per effettuare indagini.

Nello specifico prendiamo in considerazione la scenario in cui in un determinato luogo e ora è avvenuto un crimine, una rapina ad un negozio compiuta da quattro criminali. I criminali erano mascherati, hanno usato un veicolo rubato per scappare via e non hanno lasciato impronte digitali. L'unica informazione che abbiamo è che un testimone ha notato uno dei malviventi parlare al cellulare pochi minuti prima di effettuare la rapina. Gli agenti di polizia possono quindi richiedere agli operatori telefonici i dati relativi alle chiamate per scoprire chi ha effettuato o ricevuto chiamate in quella zona ed in quella frazione di tempo.

I tabulati forniti dagli operatori sono solitamente lunghi e complessi e fare delle ricerche manualmente è impensabile, per questo vengono spesso modellati in soluzioni DBMS ed interrogati tramite query per ottenere I risultati in modo rapido e preciso.

Cosa significa NoSQL?

Prima di andare a vedere nel dettaglio le due soluzioni prese in esame, analizziamo rapidamente cosa si intende con database NOSQL.

NoSQL è l'acronimo di "Not only SQL" e viene utilizzato generalmente per indicare i database che non si basano sul tradizionale modello di dati relazionale e che quindi potrebbero non avere SQL come linguaggio di interrogazione. Questa generica definizione può essere ampliata dalle caratteristiche che hanno in comune:

- **Memorizzano i dati in formati diversi**. I RDBMS memorizzano i dati in tabelle, formate da righe e colonne. I database NoSQL possono utilizzare diversi formati come archivi di documenti, grafi, archivi chiave-valore e cosi via
- **Non utilizzano le join**. I database NoSQL sono in grado di estrarre i dati utilizzando semplici interfacce orientate ai documenti senza utilizzare join SQL.
- **Rappresentazione dei dati senza uno schema fisso** (schemaless).

Le implementazioni NoSQL si basano su una rappresentazione di dati schemaless.
Con questo approccio non è necessario definire i dati in anticipo, e questi possono quindi continuare a cambiare nel tempo.

Classificazione dei sistemi NoSQL

I sistemi NoSql si suddividono:

- **Key-Values stores :** Il modello a chiave-valore si basa su una API analoga ad una mappa, accessibile tramite la chiave. Il valore può contenere sia dati elementari, che dati avanzati. Può convenire utilizzarli quando non è possibile definire uno schema sui dati ed è necessario un accesso rapido alle singole informazioni. Le interrogazioni si effettuano sulle chiavi (che possono essere indicizzate) e si ottiene il valore.
Viene spesso utilizzato per memorizzare informazioni che non presentano correlazioni, ad esempio per il salvataggio delle sessioni degli utenti in ambito web.
- **Column-oriented** : Vengono chiamati in questo modo perché organizzano memorizzano i dati per colonne. Ogni riga può avere un insieme diverso di *colonne, poiché possono essere aggiunte quelle necessarie e tolte quelle inutilizzate, evitando cos*ì la presenza di valori null. Consente la compressione delle informazioni e il versioning. Un utilizzo tipico è l'indicizzazione di pagine web: possiedono un testo, che può essere compresso, e cambiano nel tempo, beneficiando così del versioning.
Document database: I database orientati ai documenti sono caratterizzati da una struttura fondamentale, detta document, di solito scritta in JSON, costituita da un identificatore univoco e da un qualsiasi altro numero di attributi di qualunque tipo (purché esprimibili come un documento), anche nidificato. Sono utili quando i dati variano nel tempo, e possono mappare correttamente gli oggetti nel modello OOP.
- **Graph database** : I database a grafo memorizzano grafi (nodi e collegamenti tra nodi) e sono adatti a rappresentare dati fortemente interconnessi tra loro e possono effettuare interrogazioni mediante un attraversamento efficiente della struttura. Rispetto ad una normale query di altri tipi di database, si può velocizzare il cammino da un nodo ad un altro aggiungendo un collegamento diretto tra i due (con costo unitario dell'operazione)

Soluzioni DBMS considerate

Le soluzioni di database NOSQL che sono state considerate sono le seguenti:

- Neo4j
- Apache HBase

Neo4j

Neo4j appartiene alla categoria dei graph database, è open source ed prodotto dalla software house Neo Technology.
Il data model è un grafo orientato con proprietà chiave-valore, che prevede:

- **nodi**, che rappresentano le entità
- **relazioni** dette anche archi,che esprimono le associazioni tra entità

Per quanto riguarda le **caratteristiche dei nodi**, il modello prevede che ognuno di essi:

- Abbia un ID univoco assegnato automaticamente da Neo4j al momento della creazione.
- Possa avere una o più Label (etichette) che servono a classificare i nodi e indicizzarli.

Ogni relazione invece :
- Ha necessariamente un tipo(etichetta), che deve essere specificata dall'utente al momento della creazione.
- Ha necessariamente una direzione, ossia una relazione va da un nodo ad un altro e non è possibile creare relazioni senza verso. È possibile però fare query su relazioni indipendentemente dalla direzione delle relazioni, ad esempio per trovare i nodi che hanno la relazione "AMICO_DI" un certo nodo, indipendentemente dal fatto che le relazioni arrivano o partono da tale nodo. Il modello si presta a molti scenari di utilizzo.

Generalmente, le Label si usano per raggruppare entità dello stesso tipo: ad esempio, se uso Neo4j come database di un social network, potrei avere dei nodi con Label "Utente", altri con label "Gruppi", "Messaggi", etc. Un nodo può avere più Label, aprendo la strada ad applicazioni interessanti, implementando una sorta di polimorfismo. I nodi con una certa Label possono essere indicizzati su certe proprietà, per velocizzarne la ricerca.

E' stato sviluppato interamente in java ed presenta le seguenti caratteristiche:

- robusto
- scalabile

- alte prestazioni
- dotato di transazioni ACID
- alta disponibilità
- tecniche di memorizzazione per miliardi di nodi e relazioni,
- alta velocità di interrogazione tramite attraversamenti

Inoltre è dotato linguaggio di interrogazione dichiarativo e grafico (**Cypher**), dispone di numerosi driver che gli permettono di interfacciarsi con numerosi linguaggi di programmazione.
E' un DBMS schemaless, ciò sta a significare che i suoi dati non devono attenersi ad alcuna struttura di riferimento prefissata.
Innanzitutto, esso si presta a modellare situazioni che hanno intrinsecamente un modello a grafo, come ad esempio un'infrastruttura di una rete aziendale. Inoltre, grazie alla facilità di navigazione all'interno del grafo, questo modello è adatto a casi in cui siano necessarie ricerche semantiche, ad esempio nei sistemi ***di rilevazione di frodi.***

HBase è un **database column oriented.**

HBase è una base di dati distribuita open source modellata su BigTable di Google e scritta in Java.

Fu sviluppato come parte del progetto Hadoop dell'Apache Software Foundation ed eseguito su HDFS (Hadoop Distributed File System), fornendo capacità simili a quelle di BigTable per Hadoop.

Dal punto di vista architetturale è molto più complesso rispetto Cassandra (altro database column oriented). Per un numero di nodi ristretto infatti non è conveniente.
Si basa a basso livello su hadoop HDFS.
Hadoop HDFS (contiene un nodo principale chiamato NameNode che tiene traccia dei server Datanode in cui sono memorizzati dati).

Ad alto livello invece si basa su **HbaseMaster**, che comunica con altri server slave **HRegionServer**. Il client comunicherà con il master tramite un middleware di comunicazione: **zookeeper**.

Progettazione

Per la risoluzione della problematica per prima cosa sono stati generati dei datasets che simulano un effettivo tabulato telefonico.

Ogni riga del dataset descrive, univocamente, una chiamata effettuata. Gli attributi assegnati ad ogni chiamata sono I seguenti:

1. ID
Identificativo univoco assegnato ad ogni chiamata.

2. FULL_NAME
Nome completo, nome e cognome, di chi ha effettuato la chiamata, il chiamante.

3. FIRST_NAME
Nome del chiamante.

4. LAST_NAME
Cognome del chiamante.

5. CALLING_NBR
Numero telefonico del chiamante.

6. FULL_NAME_CALLED
Nome completo di chi ha ricevuto la chiamata.

7. CALLED_NBR
Numero telefonico del chiamato.

8. START_DATE
Data ed ora di inizio della chiamata, espressa in datevalue.

9. END_DATE
Data ed ora di fine della chiamata sempre espressa in datevalue.

10. DURATION
Durata della chiamata

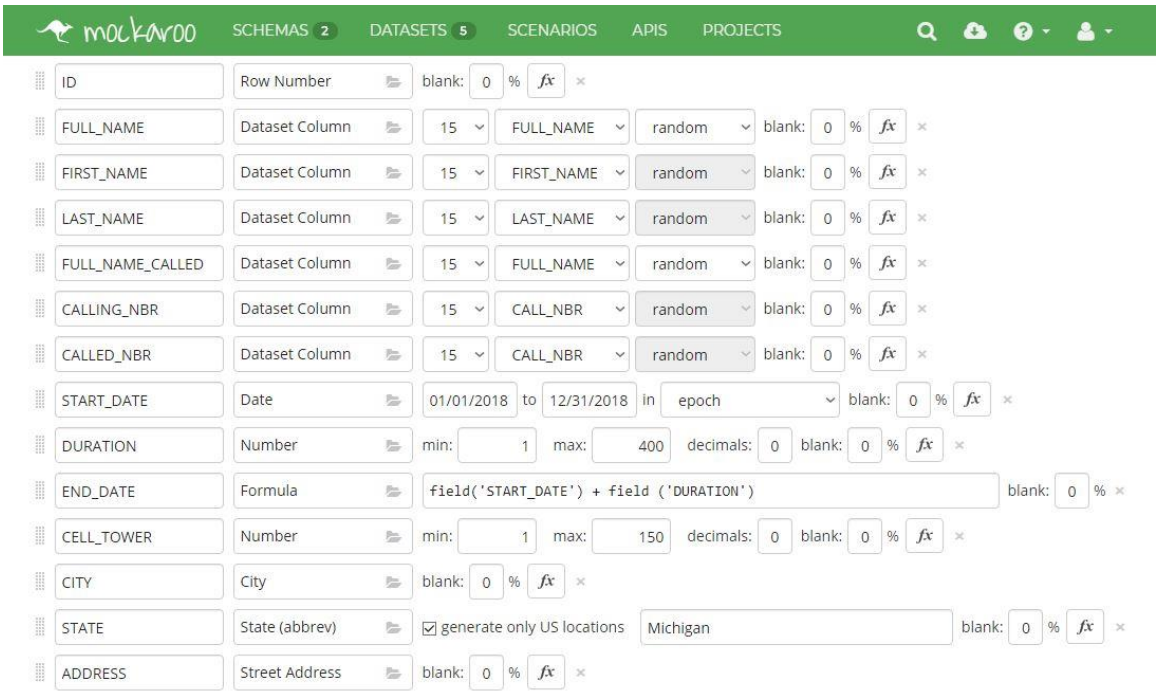
11. CELL_TOWER
Numero della cella telefonica a cui era agganciato il segnale del dispositivo da cui è stata effettuata la chiamata.

12. CITY
Città da cui è stata effettuata la chiamata.

13. STATE
Stato di appartenenza della città.

14. ADDRESS
Indirizzo da cui è stata effettuata la chiamata.

Per questo progetto, sono stati preparati dei sei di dati fittizi utilizzando **Mockaroo**.
I dataset sono stati salvati in formato csv (comma-separated values).
Di seguito ripoto lo screen di esempio della struttura dati in Mockaroo:

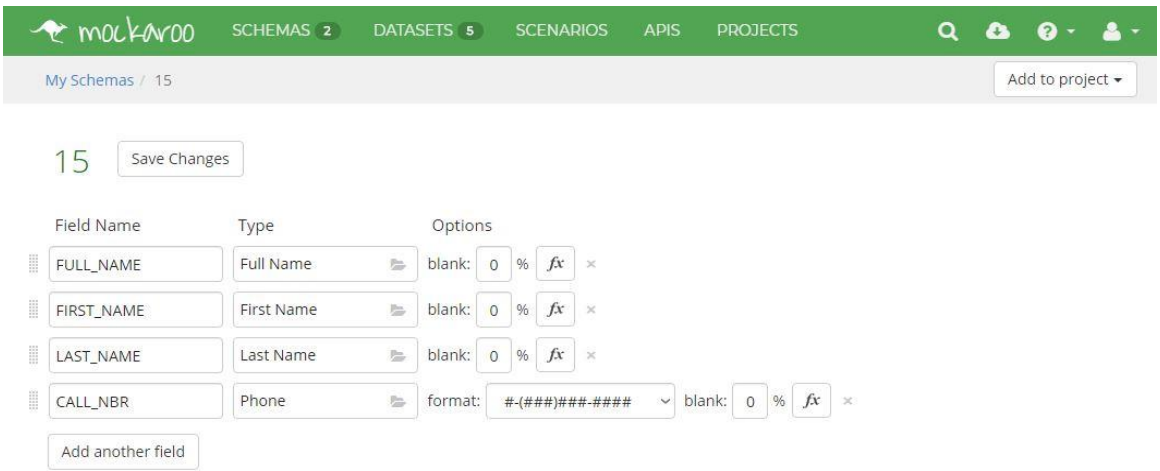


Per ottenere uno stesso chiamante (nome e numero telefonico) più volte nel nostro dataset, quindi lo stesso chiamante che ha effettuato più

chiamate e non solo una, è stato utilizzato un dataset contenente i soli attributi: full_name, first_name, last_name e call_nbr.

Esso è stato importato come ripetizione random all'interno del dataset effettivo tramite le funzioni di Mockaroo. Nello stesso modo abbiamo impostato che un chiamante può essere anche un chiamato all'interno dello stesso dataset.

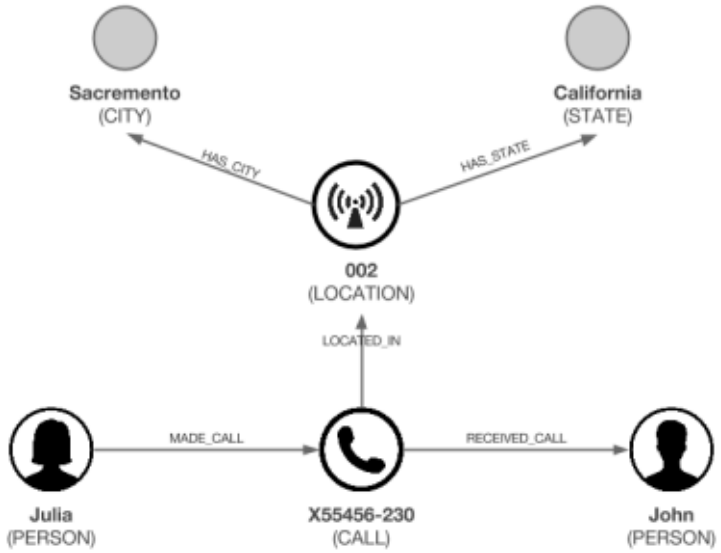
Di seguito lo schema del dataset di supporto per la creazione del dataset effettivo:



Un esempio in formato csv del nostro dataset:

DATA MODEL NEO4J

Su Neo4J utilizzeremo i dati memorizzati nei file csv per costruire un grafico. Per fare ciò, dobbiamo definire un **modello a grafo.**



Come si può vedere sopra, il data model costruito è incentrato sulle chiamate. Una singola telefonata collega insieme 4 entità:

- 2 proprietari di telefoni
- 1 posizione
- 1 stato
- 1 città

DATA MODEL HBASE

Il data model in HBase è progettato per supportare dati semistrutturati; che possono variare in termini di dimensioni del campo, tipo di dato e colonne. Il data model in HBase è costituito da diversi componenti logici:

- Tabelle: le tabelle HBase sono più simili alla raccolta logica di righe archiviate in partizioni separate denominate aree.

- Row: una riga è un'istanza di dati in una tabella ed è identificata da una **rowkey**. Le chiavi di riga sono univoche in una tabella e vengono sempre considerate come un byte[].

- Column Family: i dati in una riga vengono raggruppati come famiglie di colonne. Ogni famiglia di colonne ha un'altra colonna e queste colonne in una famiglia vengono archiviate insieme in un file di archiviazione di basso livello noto come HFile.

- Colonne: una **column family** è composta da una o più colonne. Una colonna è identificata da un qualificatore di colonna costituito dal nome della famiglia di colonne concatenato con il nome della colonna utilizzando i due punti, ad esempio: columnfamily:columnname. Possono essere presenti più colonne all'interno di una famiglia di colonne e le righe all'interno di una tabella possono avere un numero di colonne vario.

- Cell: una cella memorizza i dati ed è essenzialmente una combinazione univoca di rowkey, Column Family e Column (Column Qualifier).

- Version: i dati memorizzati in una cella sono con controllo delle versioni e le versioni dei dati sono identificate dal **timestamp**.

In particolare per questo progetto è stato utilizzato un data model così sintentizzato:

3 Column Family:

- **‘Person’** con 4 colonne : first_name, last_name, full_name, calling_number;
- **‘Call’** con 5 colonne: called_number, full_name_called, start_date, duration, end_date;
- **‘Location’** con 4 colonne: cell_tower, city, state, address;

CRIMINAL_DATA		
Family:	person:	Columns: first_name, last_name, full_name, calling_nbr
	call:	Columns: called_nbr, full_name_called, start_date, duration, end_date
	location:	Columns: cell_tower, city, state, address

Nel prossimo paragrafo illustreremo come sono stati implementati questi due data model.

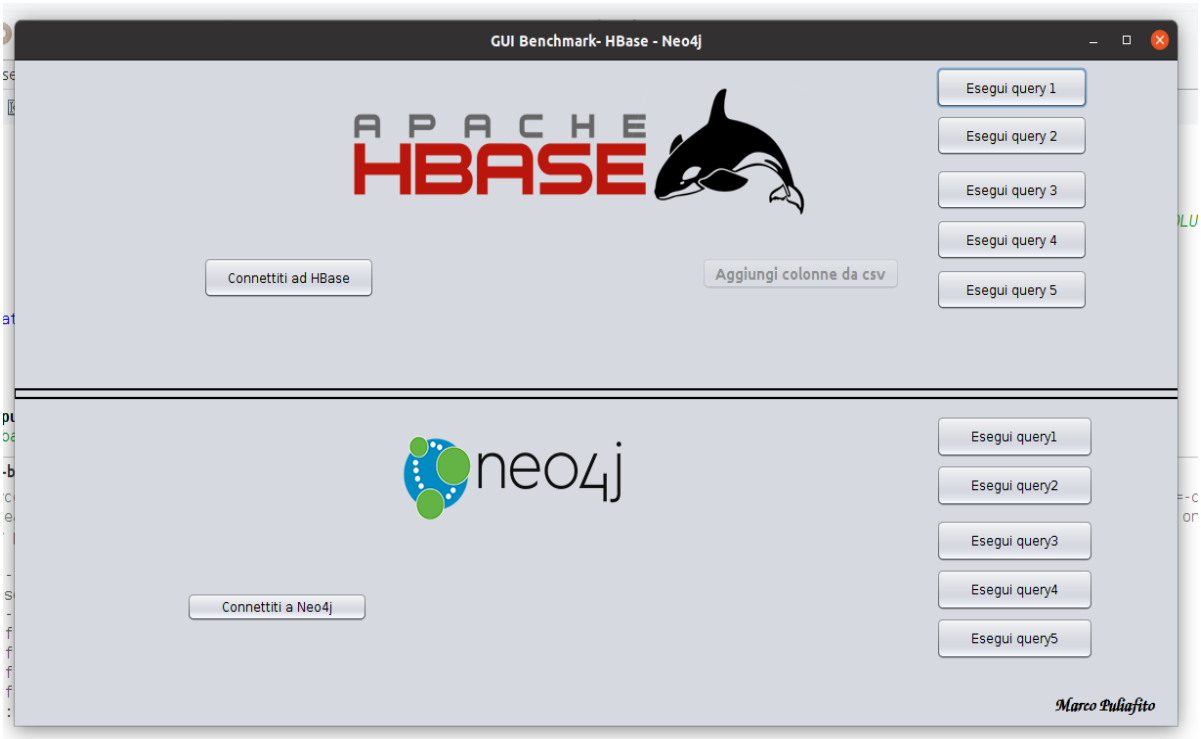
Implementazione

Per l'implementazione di questo progetto è stata utilizzato Oracle VM con installazione di un ambiente Linux, in quanto l'installazione di HBase su ambiente windows risulta più problematica.

Il dataset creato come visto nel precedente paragrafo in formato CSV è stato poi importato su **Neo4J**, tramite apposita operazione di import.

Su **HBase** mi sono servito di un linguaggio di programmazione esterno, Java, e delle API per il suddetto DBMS: **Hbase-Client API**, che permettono l'accesso e l'esecuzione di operazioni sul database.

È stata creata una **GUI** per entrare maggiormente nel vivo del progetto ed avere una smart e piacevole user interface.
Di seguito la finestra di comando di questo progetto:



Per quanto riguarda il grafo creato in **Neo4J** sono stati utilizzati I seguenti comandi di Cypher:

//Il comando CONSTRAINT ci permette di definire dei vincoli di unicità sulle chiavi primarie
//Il comando LOAD CSV ci permette di importare il dataset in formato csv.
//Il comando MERGE può essere immaginato come un tentativo di effettuare una corrispondenza sulla struttura dati, e se questa corrispondenza non esiste la crea;quindi può essere considerata una combinazione dei comandi MATCH e CREATE.
//Il comando ON CREATE SET ci permette di stabilire i campi da estrarre dal file csv.
//Il comando MATCH utilizzato nella creazione delle relazioni,ci permette di stabilire un legame tra le chiavi delle entità coinvolte nella relazione.

```
CREATE CONSTRAINT ON (a:PERSON) assert a.number is unique;
CREATE CONSTRAINT ON (b:CALL) assert b.id is unique;
CREATE CONSTRAINT ON (c:LOCATION) assert c.cell_tower is unique;
CREATE CONSTRAINT ON (d:STATE) assert d.name is unique;
CREATE CONSTRAINT ON (e:CITY) assert e.name is unique;
```

Per creare dei vincoli di univocità all'interno del grafo, il numero di una persona deve essere univoco così come l'id di una chiamata, la cella ripetitrice per una zona, il nome di uno stato e di una città.

```
USING PERIODIC COMMIT 1000
LOAD CSV WITH HEADERS FROM "file:///call.csv" AS line
```

Comando utilizzato per importare il dataset creato con Mockaroo.

```
MERGE (a:PERSON {number: line.CALLING_NBR})
ON CREATE SET a.first_name = line.FIRST_NAME, a.last_name = line.LAST_NAME,
a.full_name = line.FULL_NAME
ON MATCH SET a.first_name = line.FIRST_NAME, a.last_name = line.LAST_NAME,
a.full_name = line.FULL_NAME
MERGE (b:PERSON {number: line.CALLED_NBR})
ON CREATE SET b.full_name = line.FULL_NAME_CALLED
ON MATCH SET b.full_name = line.FULL_NAME_CALLED
```

Con questa parte di codice andiamo a creare I nodi Person come chiamanti e chiamati, impostando loro come proprietà gli attributi scanditi dal file csv.
Il comando Merge viene utilizzato come Create per la creazione di porzioni di grafo, solo che a differenza di Create però non crea un nodo già esistente ma

permette di specificare cosa fare se il nodo che si voleva creare esiste già oppure no.

```
MERGE (c:CALL {id: line.ID})
ON CREATE SET c.start = line.START_DATE, c.end= line.END_DATE, c.duration =
line.DURATION
MERGE (d:LOCATION {cell_tower: line.CELL_TOWER})
ON CREATE SET d.address= line.ADDRESS, d.state = line.STATE, d.city = line.CITY
MERGE (e:CITY {name: line.CITY})
MERGE (f:STATE {name: line.STATE})
```

Allo stesso modo qui aniamo a creare I nodi Call, Location, City e State.

```
DROP CONSTRAINT ON (a:PERSON) ASSERT a.number IS UNIQUE;
DROP CONSTRAINT ON (a:CALL) ASSERT a.id IS UNIQUE;
DROP CONSTRAINT ON (a:LOCATION) ASSERT a.cell_tower IS UNIQUE;
CREATE INDEX ON :PERSON(number);
CREATE INDEX ON :CALL(id);
CREATE INDEX ON :LOCATION(cell_tower);
```

Eliminiamo I precedenti vincoli ed aggiungiamo degli indici a number di Person, id di Call e cell_tower di Location.

```
USING PERIODIC COMMIT 1000
LOAD CSV WITH HEADERS FROM "file:///dataset_criminals.csv" AS line
MATCH (a:PERSON {number: line.CALLING_NBR}),(b:PERSON {number:
line.CALLED_NBR}),(c:CALL {id: line.ID})
CREATE (a)-[:MADE_CALL]->(c)-[:RECEIVED_CALL]->(b)
```

Vengono cercati I nodi person chiamanti e chiamati, I nodi call e viene creata una relazione tra loro.

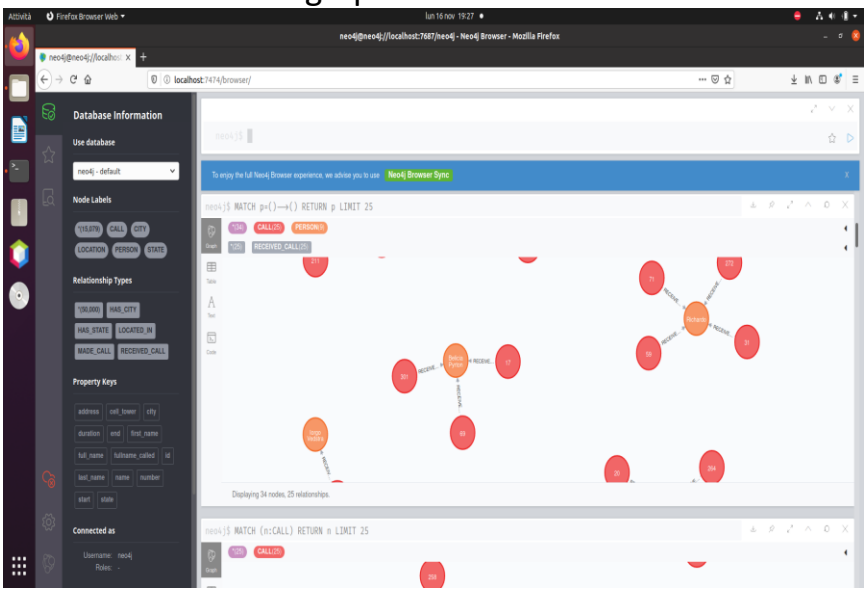
```
USING PERIODIC COMMIT 1000
LOAD CSV WITH HEADERS FROM "file:///dataset_criminals.csv" AS line
MATCH (a:CALL {id: line.ID}), (b:LOCATION {cell_tower: line.CELL_TOWER})
CREATE (a)-[:LOCATED_IN]->(b)
```

Viene creata la relazione tra call e cell_tower di appartenenza.

```
USING PERIODIC COMMIT 1000
LOAD CSV WITH HEADERS FROM "file:///dataset_criminals.csv" AS line
MATCH (a:LOCATION {cell_tower: line.CELL_TOWER}), (b:STATE {name:
line.STATE}), (c:CITY {name: line.CITY})
CREATE (b)<-[:HAS_STATE]-(a)-[:HAS_CITY]->(c)
```

Viene creata la relazione tra cell_tower, città e stato per sapere dove effettivamente si trova la cella ripetitrice.

Esempio di visualizzazione graph database e interfaccia Neo4J browser:



Da qui possiamo vedere anche il forte potere espressivo di neo4j.

Per quanto riguardata **HBase** è stata scelta una configurazione **standalone** di HBase per la realizzazione di questo progetto.
Sebbene HBase disponga di una propria Shell per l'invio di interrogazioni, si è scelto di creare il tutto con Java attraverso i driver messi a disposizione per l'interfacciamento con HBase, in particolare sono state utilizzate delle API Java.

Per prima cosa andiamo a creare una connessione con HBase master tramite API Java. Di seguito il codice:

```
boolean connectionHBase() throws IOException{
    try{
        Configuration config = HBaseConfiguration.create();

        /**
        * ConnectionFactory#createConnection() automatically looks for
        * hbase-site.xml (HBase configuration parameters) on the system's
        * CLASSPATH, to enable creation of Connection to HBase via ZooKeeper.
        */
        Connection connection = ConnectionFactory.createConnection(config);
        Admin admin = connection.getAdmin();
        //admin.getClusterMetrics(); // assure connection successfully established
        System.out.println("\n*** CONNESSIONE STABILITA ***\n");
```

Per la creazione della table e delle 3 Column family (Person, Call, Location) in **HBase**, sono stati utilizzati i seguenti comandi in codice Java (in termini di sperimentazione è stata utilizzata anche la shell di Hbase per lanciare alcuni comandi):

```
public void createNamespaceAndTable(final Admin admin, String name_space, TableName table_name, byte[] columnFamilyPerson,

    if (!namespaceExists(admin, name_space)) {
      System.out.println("Creating Namespace [" + name_space + "].");

        admin.createNamespace(NamespaceDescriptor
            .create(name_space).build());
    }
    if (!admin.tableExists(table_name)) {
        System.out.println("Creating Table [" + table_name.getNameAsString()
            + "], with Column Family ["
            + Bytes.toString(columnFamilyPerson) + ", "
            +Bytes.toString(columnFamilyCall) +  ", "
            + Bytes.toString(columnFamilyLocation) + "].");
        TableDescriptor desc = TableDescriptorBuilder.newBuilder(table_name)
            .setColumnFamily(ColumnFamilyDescriptorBuilder.of(columnFamilyPerson))
            .setColumnFamily(ColumnFamilyDescriptorBuilder.of(columnFamilyCall))
            .setColumnFamily(ColumnFamilyDescriptorBuilder.of(columnFamilyLocation))
            .build();
        admin.createTable(desc);
    }
}
```

Per il caricamento dei nostri dati sperimentali è stata utilizzata la libreria Java "commons.csv" per la lettura dei file in formato csv, e di seguito, tramite API Java "**hbase.client**" con il comando "put", sono stati caricati i dati nelle corrispondenti Column Family.
Di seguito una parte di codice:

```
public void importLocalFileToHBase(String fileName, Table table, byte[] columnFamily, String[] column) {
    long st = System.currentTimeMillis();
    try{
        int count = 0;
        Reader csvData = new FileReader(fileName);
        CSVParser parser = CSVFormat.RFC4180.withFirstRecordAsHeader().parse(csvData);
        for (CSVRecord csvRecord : parser) { //scorre le righe del csv
            String rowKey = csvRecord.get("ID");//prendo la colonna con header ID
            Put put = new Put(Bytes.toBytes(rowKey));//creo un inserimento a partire dalla row key
            //scorro la riga in base alle colonne interessate
            for(int i=1;i<column.length;i++){
                //inserimento nella columnfamily delle colonne con lo scorrere dei valori in csv
                put.addColumn(columnFamily, Bytes.toBytes(column[i]),Bytes.toBytes(csvRecord.get(column[i])));
            }
            try {
               table.put(put); // put to server
            }
            catch (IOException e) {
               e.printStackTrace();
            }

            //stampo il contenuto della table--> CF : [column == value,..]
            Result row = table.get(new Get(Bytes.toBytes(rowKey)));
```

Implementazione Query

Una volta inseriti i dati vengono eseguite per entrambi i DB cinque query. Riportiamo in basso il codice di Cypher per tutte le query mentre per quanto riguarda HBase inseriamo un piccolo esempio in quanto il codice Java di alcune query potrebbe essere troppo lungo e di difficile comprensione (alleghiamo il codice sorgente alla relazione).

E' stata realizzata un interfaccia user friendly per effettuare i benchmark anche su Neo4j:
che una volta inserite le credenziali permette di effettuare le query. (GUI mostrata all'inizio del paragrafo "Implementazione").

Query 1

//Cercare il proprietario del numero: 98(775)873-6474

Neo4j

MATCH(c:PERSON {number:'86(644)491-7854'}) RETURN c

HBase

```
//Cercare il proprietario del numero: 98(775)873-6474
void query1() throws IOException{
    Scan scan = new Scan();
    SingleColumnValueFilter filter = new SingleColumnValueFilter(this.MY_COLUMN_FAMILY_NAME_PERSON,
        Bytes.toBytes("CALLING_NBR"), CompareOp.EQUAL, Bytes.toBytes("86(644)491-7854"));
    scan.setFilter(filter);
    ResultScanner scanner = table.getScanner(scan);
    Result result = scanner.next();
    //stampo il risultato di scan
```

Query 2

//Filtrare le chiamate effettuate in un determinato momento

Neo4j

MATCH(a:CALL {start:'1537178627'}) RETURN a

HBase

```
//Filtrare le chiamate effettuate in un determinato momento
void query2() throws IOException{
    Scan scan = new Scan();
    SingleColumnValueFilter filter = new SingleColumnValueFilter(this.MY_COLUMN_FAMILY_NAME_CALL,
        Bytes.toBytes("START_DATE"), CompareOp.EQUAL, Bytes.toBytes("1537178627"));
    filter.setFilterIfMissing(true); //non contare se manca la colonna
    scan.setFilter(filter);
    ResultScanner scanner = this.table.getScanner(scan);
    Result result = scanner.next();
    //stampo il risultato di scan
    /*for (Result result : scanner) {
```

Query 3

//Cercare le chiamate fatte al numero : 380(486)299-6217 o al numero : 389(365)470-8680

Neo4j

```
MATCH (c:PERSON)-[:MADE_CALL]->(a)-[:RECEIVED_CALL]->(d:PERSON)
WHERE d.number='380(486)299-6217' OR d.number = '389(365)470-8680'
RETURN a
```

HBase

```
//Cercare le chiamate fatte al numero : 380(486)299-6217 or numero : 389(365)470-8680
void query3() throws IOException{
    List<Filter> filters = new ArrayList<Filter>(); //array di filtri (per fare condizioni concatenate)
    Scan scan = new Scan();
    SingleColumnValueFilter filter1 = new SingleColumnValueFilter(this.MY_COLUMN_FAMILY_NAME_CALL,
        Bytes.toBytes("CALLED_NBR"), CompareOp.EQUAL, Bytes.toBytes("380(486)299-6217"));
    filters.add(filter1);
    SingleColumnValueFilter filter2 = new SingleColumnValueFilter(this.MY_COLUMN_FAMILY_NAME_CALL,
        Bytes.toBytes("CALLED_NBR"), CompareOp.EQUAL, Bytes.toBytes("389(365)470-8680"));
    filters.add(filter2);
     //creo un istanza di FIlterList,FilterList.Operator.MUST_PASS_ONE: corrisponde ad OR
    FilterList filterList = new FilterList(FilterList.Operator.MUST_PASS_ONE, filters);
    scan.setFilter(filterList);
    ResultScanner scanner = this.table.getScanner(scan);
    Result result = scanner.next();
    //stampo il risultato di scan
```

Query 4

//Filtrare le chiamate avvenute in una determinata ora e data

Neo4j

```
MATCH (a:CALL)-[:LOCATED_IN]->(b:LOCATION) WHERE b.cell_tower
='115'
AND '1578753860' < a.start AND a.start < '1589725288' WITH a, b MATCH
(c:PERSON)-[:MADE_CALL]->(a)-[:RECEIVED_CALL]->(d:PERSON)
RETURN c.full_name
```

HBase

```
//Filtrare le chiamate avvenute in una determinata zona e data
void query4() throws IOException{
    List<Filter> filters = new ArrayList<Filter>(); //array di filtri (per fare condizioni concatenate)
    Scan scan = new Scan();
    //data zona (cell tower)
    SingleColumnValueFilter filter1 = new SingleColumnValueFilter(this.MY_COLUMN_FAMILY_NAME_LOCATION,Bytes.toBytes("CELL_TOWER"),
        CompareOp.EQUAL, Bytes.toBytes("115"));
    filters.add(filter1);
    //chiamata effettuata dopo il time indicato come value
    SingleColumnValueFilter filter2 = new SingleColumnValueFilter(this.MY_COLUMN_FAMILY_NAME_CALL,Bytes.toBytes("START_DATE"),
        CompareOp.GREATER, Bytes.toBytes("1578753860"));
    filters.add(filter2);
    //chiamata effettuata prima del time indicato come value
    SingleColumnValueFilter filter3 = new SingleColumnValueFilter(this.MY_COLUMN_FAMILY_NAME_CALL,Bytes.toBytes("START_DATE"),
        CompareOp.LESS, Bytes.toBytes("1589725288"));
    filters.add(filter3);
    //creo un istanza di FIlterList,FilterList.Operator.MUST_PASS_ALL: corrisponde ad AND
    FilterList filterList = new FilterList(FilterList.Operator.MUST_PASS_ALL, filters);
    scan.setFilter(filterList);
    ResultScanner scanner = this.table.getScanner(scan);
    Result result = scanner.next();
```

Query 5

//Cercare le chiamate effettuate dalla città di Albany e dalla città di New York City

Neo4j

MATCH (a:CALL)-[:LOCATED_IN]->(b:LOCATION) WHERE b.city = 'Albany' OR b.city = 'New York City' Return a

HBase

```
//Cercare le chiamate effettuate dalla città di Albany e da New York city
void query5() throws IOException{
    List<Filter> filters = new ArrayList<Filter>(); //array di filtri (per fare condizioni concatenate)
    Scan scan = new Scan();
    //data zona (cell tower)
    SingleColumnValueFilter filter1 = new SingleColumnValueFilter(this.MY_COLUMN_FAMILY_NAME_LOCATION, Bytes.toBytes("CITY"),
        CompareOp.EQUAL, Bytes.toBytes("Albany"));
    filters.add(filter1);
    //chiamata effettuata dopo il time indicato come value
    SingleColumnValueFilter filter2 = new SingleColumnValueFilter(this.MY_COLUMN_FAMILY_NAME_LOCATION, Bytes.toBytes("CITY"),
        CompareOp.EQUAL, Bytes.toBytes("New York City"));
    filters.add(filter2);
    //creo un istanza di FIlterList,FilterList.Operator.MUST_PASS_ONE: corrisponde ad OR
    FilterList filterList = new FilterList(FilterList.Operator.MUST_PASS_ONE, filters);
    scan.setFilter(filterList);
    ResultScanner scanner = this.table.getScanner(scan);
```

ESPERIMENTI

Premessa:

Per effettuare gli esperimenti, sono stati valutati dataset di differenti dimensioni:

100 record

1.000 record

10.000 record

Dal momento che molti DBMS NoSQL utilizzano meccanismi di caching, a parità di dimensione del dataset, si è considerato, a parte il primo tempo di esecuzione, **e il valor medio delle successive 30 esecuzioni per un totale di 31 tests**.

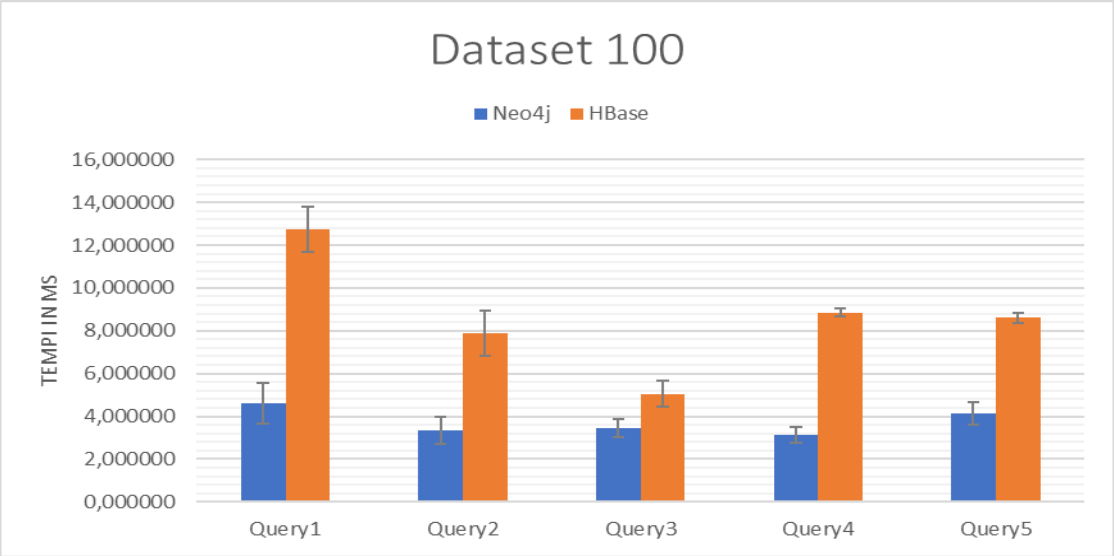
Per ogni dataset sono stati rappresentati due **istogrammi** :

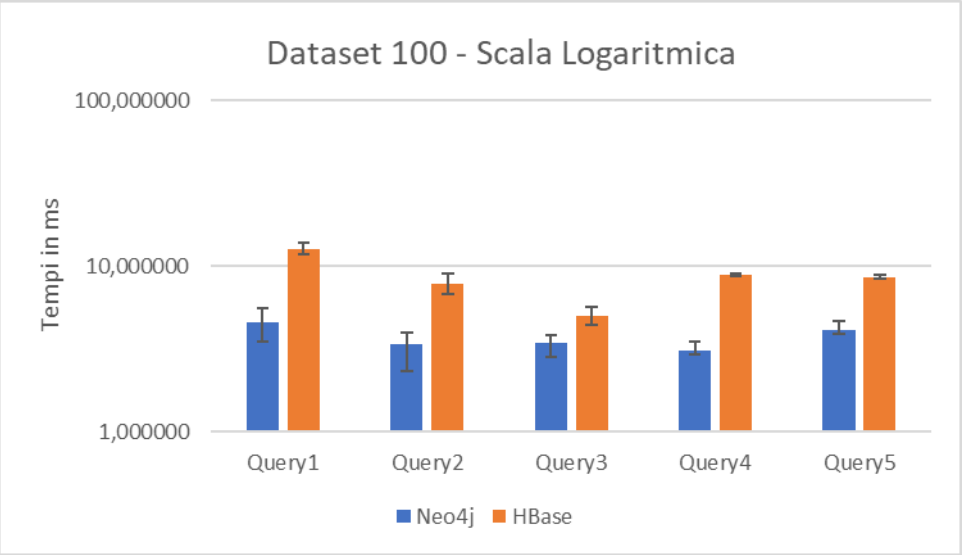
- Il primo raffigura i tempi medi di ogni query, con un intervallo di confidenza del 95%
- Il secondo raffigura i tempi medi di ogni query rappresentati su scala logaritmica e con un intervallo di confidenza del 95%

DATASET da 100

HBase					
Query1	Query2	Query3	Query4	Query5	
12,7609742	7,895675	5,037693	8,850974	8,595509	MEDIA
2,83395001	2,818012	1,619672	0,50037	0,677049	DEVIAZIONE STANDARD
1,05821432	1,052263	0,604795	0,186841	0,252814	INTERVALLO DI CONFIDENZA

Neo4j					
Query1	Query2	Query3	Query4	Query5	
4,591726	3,357361	3,4504555	3,113337	4,132584	MEDIA
2,549993	1,674399	1,09900048	0,99697	1,414986	DEVIAZIONE STANDARD
0,9521832	0,625231	0,41037352	0,372275	0,528364	INTERVALLO DI CONFIDENZA

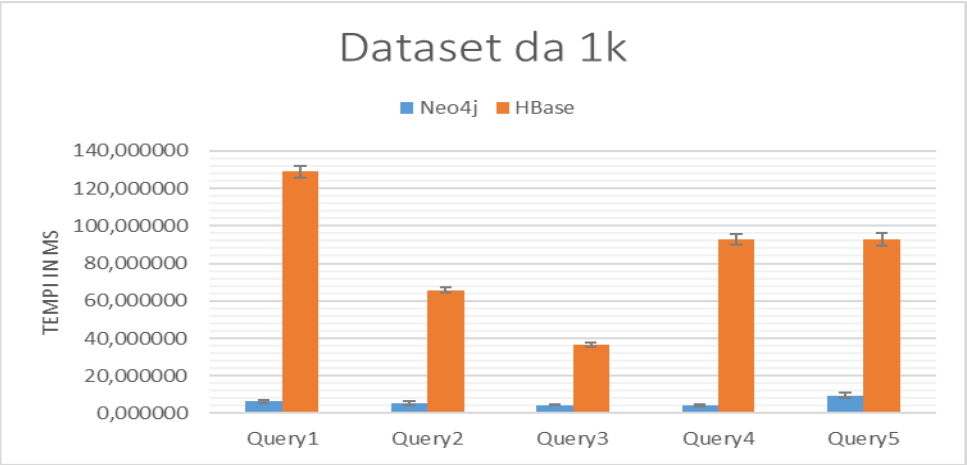


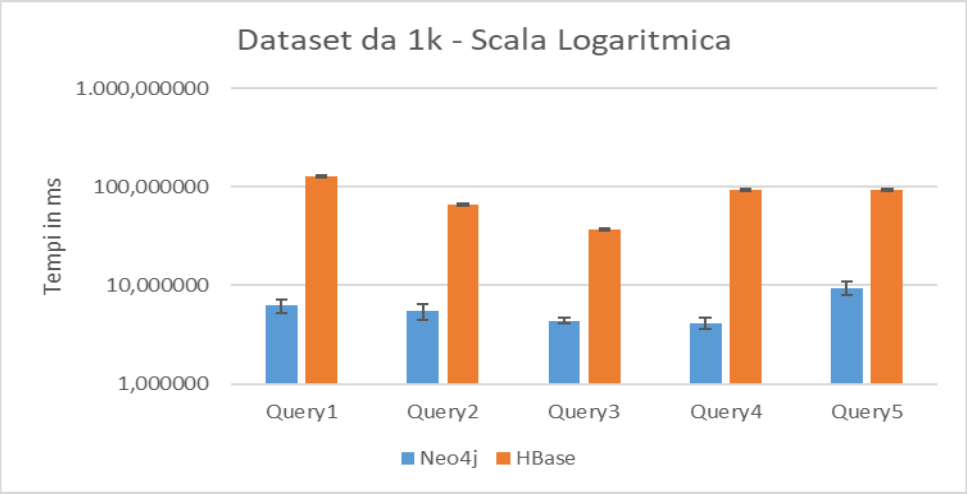


DATASET da 1k

HBase					
Query1	Query2	Query3	Query4	Query5	
128,976075	65,59253	36,6595262	92,87693	92,85949	MEDIA
8,159795	3,923884	3,062348	7,578487	8,553569	DEVIAZIONE STANDARD
3,046917519	1,465202	1,143499721	2,829854	3,193955	INTERVALLO DI CONFIDENZA

Neo4J					
Query1	Query2	Query3	Query4	Query5	
6,232162	5,511872	4,3784306	4,163135033	9,4177504	MEDIA
2,563880	2,764552	0,741628	1,468860	4,042448	DEVIAZIONE STANDARD
0,9573685	1,032300865	0,276928599	0,548481459	1,509474718	INTERVALLO DI CONFIDENZA

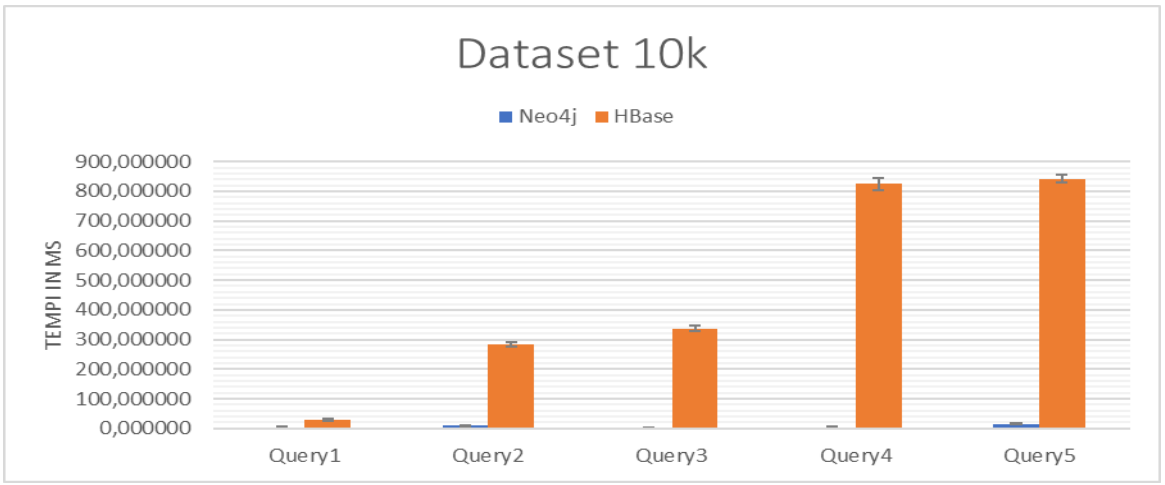


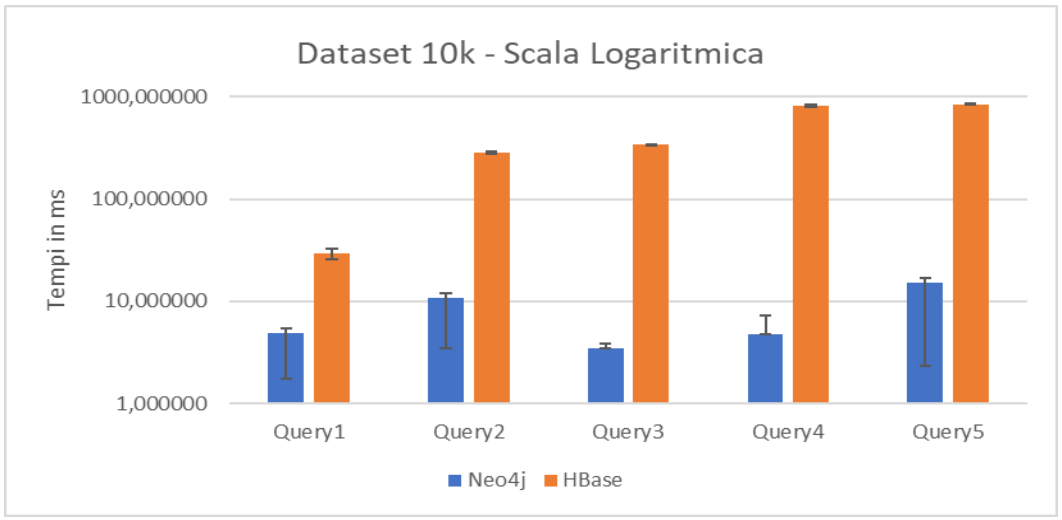


DATASET da 10k

HBase					
Query1	Query2	Query3	Query4	Query5	
29,060271	285,196486	337,541930	824,053131	842,341651	MEDIA
8,562308	19,471559	21,893574	54,003725	33,946845	DEVIAZIONE STANDARD
3,197218329	7,270799582	8,1751948	20,1653225	12,67596021	INTERVALLO DI CONFIDENZA

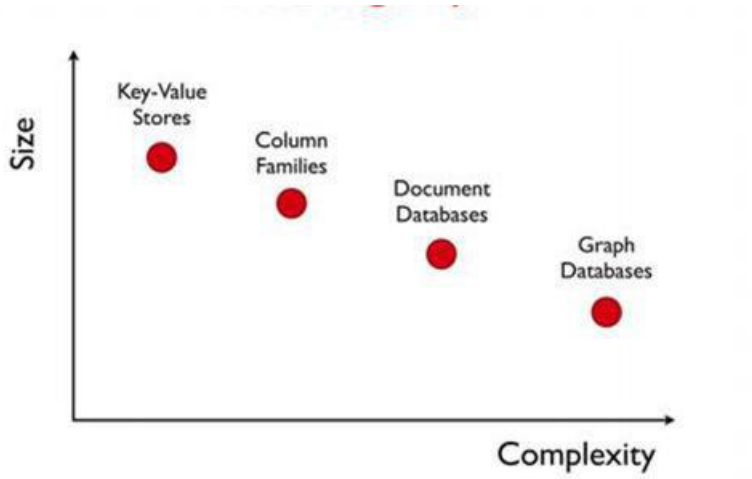
NEO4j					
Query1	Query2	Query3	Query4	Query5	
4,925596	10,689597	3,475586	4,747012	14,992874	MEDIA
1,460634	3,134355	0,896635	6,813430	4,558989	DEVIAZIONE STANDARD
0,5454098	1,17038736	0,334809103	2,544176725	1,70235444	INTERVALLO DI CONFIDENZA





CONCLUSIONI

Da questa indagine possiamo concludere che il DBMS Neo4J è senza dubbio migliore per modellare una situazione come quella qui presa in carico, il suo modello che mette in risalto le relazioni ed il linguaggio Cypher rendono la modellazione e l'interrogazione semplice ed agile a differenza di HBase per cui avremmo dovuto utilizzare un linguaggio esterno per effettuare le stesse query.
Questo è un risultato che non ci sorprende dato che I Graph database, nella famiglia dei NoSQL sono I DBMS che riescono agestire meglio la complessità di operazioni e schema.



In base ai tempi di risposta raccolti dalle cinque query possiamo constatare che per piccoli set di dati sia Neo4J che HBase offrono ottimi risultati.

Al crescere del set di dati invece possiamo notare come le prestazioni di HBase degradino velocemente, aumentando anche di molto I tempi di risposta. Al contrario Neo4J mantiene quasi inalterati I suoi tempi di risposta, dimostrandosi un DBMS incredibilmente efficiente.