Università degli Studi di Messina

Dipartimento di Scienze Matematiche, Informatiche, Fisiche e Scienze della Terra

Relazione di Basi di Dati 2
A. A. 2021/2022

"Casi d'uso di antiriciclaggio di denaro"

Studenti:
Buemi Emanuele – Giambò Filippo

Docente:
Prof. Antonio Celesti

Sommario

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.

I due DBMS presi in considerazione sono:

- **Neo4j**
- **Hbase**

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 tutte le persone coinvolte nell'**anti-riciclaggio di denaro** attraverso l'analisi delle transazioni da loro effettuate.
Il riciclaggio di denaro *"sporco"*, noto internazionalmente come money laundering, secondo le definizioni giuridiche consiste nella sostituzione o trasferimento di denaro, beni o altre utilità provenienti da attività criminose con lo scopo di ostacolare la possibilità di identificare la loro provenienza illecita.

In particolare i casi di studio in cui è possibile intravedere dei casi sospetti di anti-riciclaggio di denaro riguardano:

- l'identificazione degli UBO, cioè un individuo che detiene più del 25% del capitale (quote) riguardante una determinata entità (società).
- il problema dello smurfing che consiste nell'identificare tutte le transazioni che vengono effettuate su uno stesso conto bancario da parte di più persone.
- il problema del round trip in cui i fondi vengono restituiti dopo essere stati trasferiti a delle entità collegate tra di loro.

Il contrasto al fenomeno permette di combattere efficacemente la criminalità organizzata in quanto si toglie alla stessa la possibilità di utilizzare effettivamente il denaro proveniente da attività illecite. Seguendo le tracce lasciate dalle transazioni per riciclare il denaro è possibile risalire ai reati e quindi ai colpevoli.

Soluzioni DBMS considerate

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

- Neo4j
- Apache HBase

Neo4j

Neo4j è un database NoSQL che appartiene alla categoria dei "***Graph database***".

Il data model è un grafo orientato con proprietà nome-valore, che prevede:

- **nodi**, che rappresentano le entità,
- **archi** (relazioni), che esprimono le associazioni tra le entità (tra nodi).

Poiché Neo4j è progettato attorno a questa semplice e potente ottimizzazione (fatta da nodi e archi), esegue delle query con connessioni molto complesse più velocemente rispetto ad altri database (ma con un volume di dati più basso). Grazie a questa caratteristica, Neo4j viene ottimamente utilizzato 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.

Neo4j è stato sviluppato in Java e presenta le seguenti caratteristiche:

- È robusto e scalabile.
- Fornisce alte prestazioni.
- Fornisce un'alta velocità di interrogazione tramite gli attraversamenti.
- Dispone di numerosi driver che gli permettono di interfacciarsi con numerosi linguaggi di programmazione.
- Supporta un linguaggio di querying dichiarativo chiamato **Cypher,** creato dal team di Neo4j allo scopo di interagire con il database in modo semplice e veloce.
    - Cypher è ottimizzato per lo studio dei grafi e ci consente di indicare cosa vogliamo selezionare, inserire, aggiornare o eliminare dai nostri dati senza il bisogno di fornire una descrizione su come farlo.

Hbase

HBase è un database NoSQL, distribuito, di tipo "***Column-Oriented***", open source e scritto in linguaggio Java.

Memorizza i dati in tabelle che mantengono il formato chiave-valore.
Ogni riga contiene:

- una chiave univoca detta "***row key***"
- una o più "***column families***", ovvero famiglie di colonne che contengono colonne figlie con tipi di dato simili tra loro.

Ad ogni colonna invece corrisponde un value (valore), a cui viene associato un timestamp che servirà per gestire il versioning dei dati (utile per le repliche).

HBase è una soluzione di database NoSQL che fa parte del framework di tool di Hadoop.
L'architettura di Hbase al livello più basso si basa su ***HDFS*** (*Hadoop Distributed File System*), ovvero il file system di Hadoop che permette di memorizzare i nostri dati su un sistema distribuito e parallelo mentre al livello più alto si basa su **HBaseMaster**, che comunica con altri server slave di nome **HRegionServer**. Inoltre utilizza un middleware di comunicazione chiamato **Zookeeper** per coordinare la comunicazione tra il client e il nodo master.

HBase risulta più efficiente quando si ha a che fare con dei progetti di applicazioni molto complesse, fornisce API che consentono lo sviluppo in qualsiasi linguaggio di programmazione e inoltre non necessità di occuparsi in prima persona della replicazione e manutenzione dei dati, in quanto è HDFS ad occuparsene.

Progettazione

Per lo studio delle problematiche descritte nella sezione precedente sono stati generati dei datasets fittizi che simulano delle situazioni legate all'anti-riciclaggio di denaro.

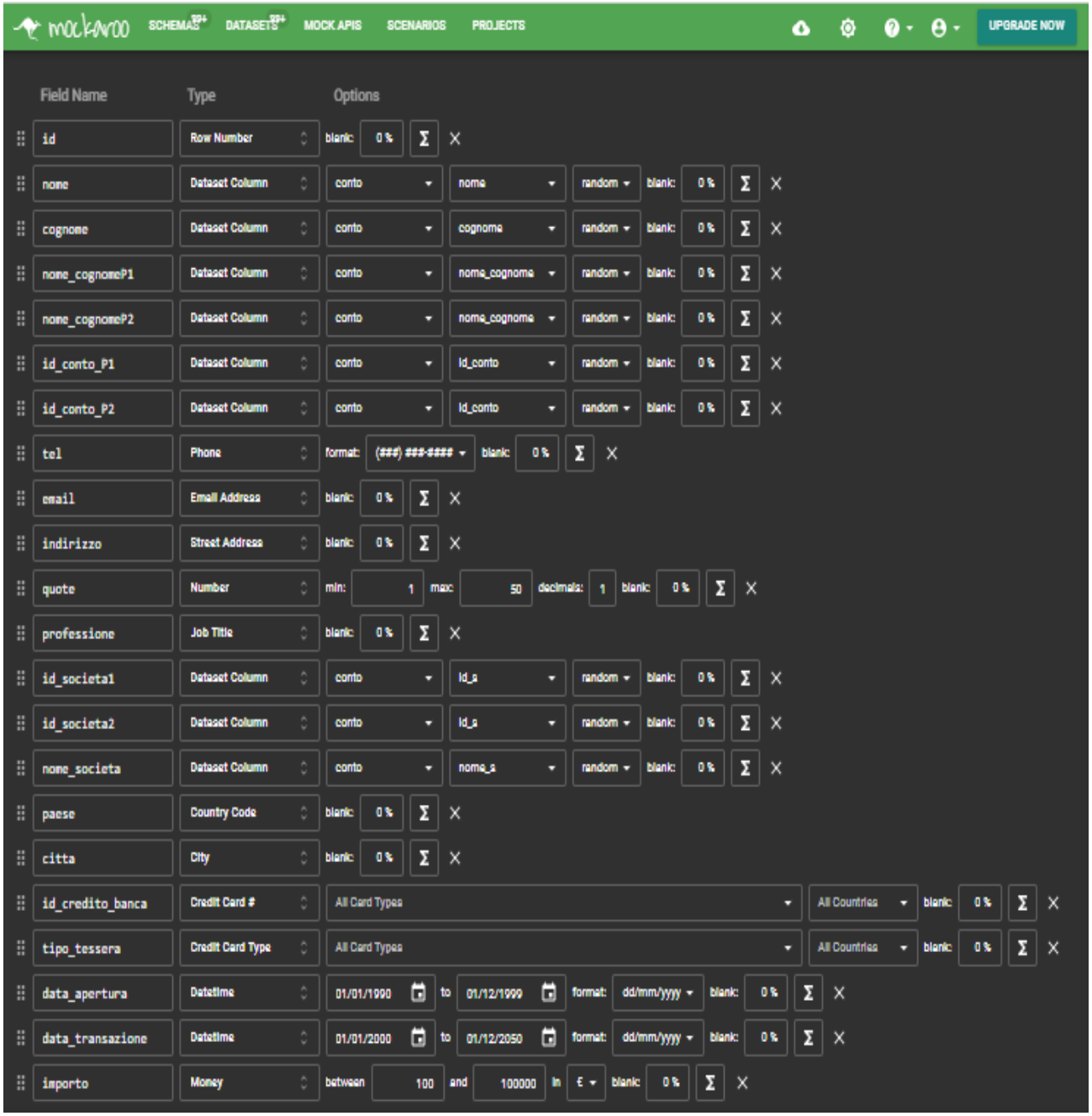
Gli attributi che abbiamo assegnato alle varie entità sono:

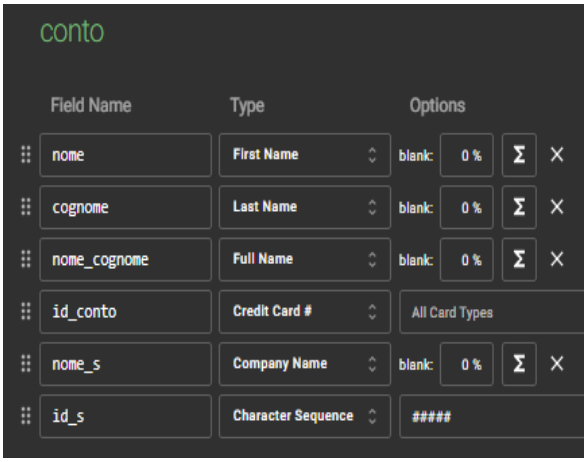
- **id**
  Identificativo univoco assegnato ad ogni riga del dataset.
- **nome**
  Nome della persona che effettua e/o riceve la transazione.
- **cognome**
  Cognome della persona che effettua e/o riceve la transazione.
- **nome_cognomeP1**
  Nome completo della persona che effettua la transazione.
- **nome_cognomeP2**
  Nome completo della persona che riceve la transazione.
- **id_conto_P1**
  Identificativo della persona che effettua la transazione.
- **id_conto_P2**
  Identificativo della persona che riceve la transazione.
- **tel**
  Numero di telefono della persona.
- **email**
  Indirizzo email della persona.
- **indirizzo**
  Indirizzo civico della persona.
- **quote**
  Specifica la percentuale di possesso della società da parte della persona.
- **professione**
  Titolo di lavoro della persona.
- **id_societa1**
  Identificativo della società che effettua la transazione.
- **id_societa2**
  Identificativo della società della quale la persona possiede delle quote.

- **nome_societa**
  Nome della società riferito sia alla società che effettua la transazione e sia a quella in possesso della persona.
- **paese**
  Prefisso della nazione dove si trova la società.
- **citta**
  Città riferita della sede dove si trova la società.
- **id_credito_banca**
  Identificativo del conto bancario appartenente alla persona ricevente la transazione.
- **tipo_tessera**
  Tipo di carta di credito.
- **data_apertura**
  Data di apertura del conto bancario.
- **data_transazione**
  Data di quado è stata effettuata la transazione.
- **Importo**
  Somma di denaro trasferita durante la transazione.

Per questo progetto, sono stati preparati dei sei di dati fittizi utilizzando **Mockaroo**.
I dataset sono stati salvati in formato csv (comma-separated values).

Qui sotto è ripoto lo screen della struttura dati in Mockaroo:





Per fare in modo che una stessa persona possa effettuare e ricevere una transazione, e fare in modo che una persona o una società effettui più transazioni su più conti bancari (creando così delle relazioni 1 – N) è stato utilizzato un dataset contenente i soli attributi: nome, cognome, nome_cognome, id_conto, nome_s e id_s e abbiamo importato questo dataset come ripetizione random all'interno del dataset precedente tramite le funzioni di Mockaroo.

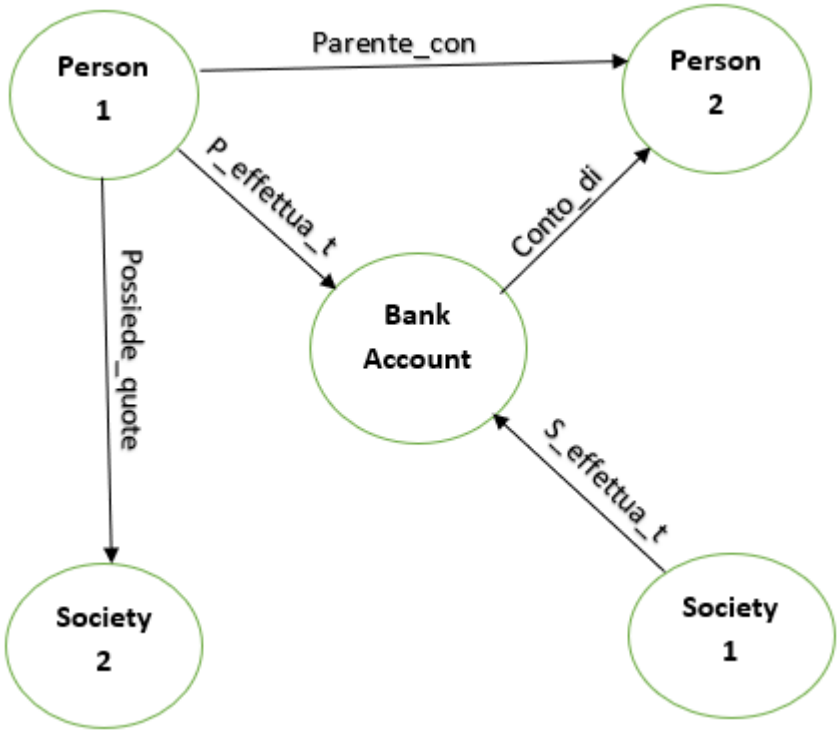
Data model Neo4j

Il data model di Neo4j si focalizza su 5 entità che sono:

- 2 entità **Person**,
- 1 entità **BankAccount**,
- 2 entità **Society**,

e 5 relazioni che sono:

- 1 relazione "**P_effettua_t**" → che fa riferimento alla persona che effettua la transazione,
- 1 relazione "**Conto_di**" → che fa riferimento al conto bancario posseduto da una determinata persona.
- 1 relazione "**S_effettua_t**" → che fa riferimento alla società che effettua la transazione,
- 1 relazione "**Possiede_quote**" → che fa riferimento alla persona che possiede la società,
- 1 relazione "**Parente_con**" → che fa riferimento alla relazione di parentela tra due persone.



Secondo questo modello a grafo da noi costruito una persona può effettuare una o più transazioni (Person 1) sul conto bancario (Bank Account) di una seconda persona e nello stesso tempo ricevere una o più transazioni (Person 2) su un proprio conto bancario.
Allo stesso modo una società può effettuare una o piu transazioni (Society 1) e può essere posseduta da diverse persone (Society 2).

Data model Hbase

Il data model di Hbase è costituito da una tabella costituita da **3 Column Family** che sono:

- **Person** → composta da 7 colonne (id_conto_P1, nome, cognome, nome_cognomeP1, tel, email, indirizzo);
- **Society** → composta da 5 colonne (id_societa2, nome_societa, paese, citta, quote);
- **BankAccount** → composta da 8 colonne (id_conto_P2, id_credito_banca, nome_cognomeP2, professione, tipo_tessera, data_transazione, data_apertura, importo).

DB		
FAMILY:	**Person:**	**Columns:** id_conto_P1, nome, cognome, nome_cognomeP1, tel, email, indirizzo
	Society:	**Columns:** id_societa2, nome_societa, paese, citta, quote
	BankAccount	**Columns:** id_conto_P2, id_credito_banca, nome_cognomeP2, professione, tipo_tessera, data_transazione, data_apertura, importo

Implementazione

In questa sezione verranno analizzati i metodi più importanti usati nel programma. Per l'implementazione di questo progetto abbiamo utilizzato un sistema operativo Ubuntu, in quanto l'installazione di Hbase su ambiente Windows risulta più problematica.

Il linguaggio di programmazione usato è Java e per il collegamento con i due database sono stati aggiunti al file "***pom.xml***" i seguenti **driver**:

- Per Hbase:

```
<groupId>org.apache.hbase</groupId>
<artifactId>hbase-client</artifactId>
 <version>2.4.8</version>
```

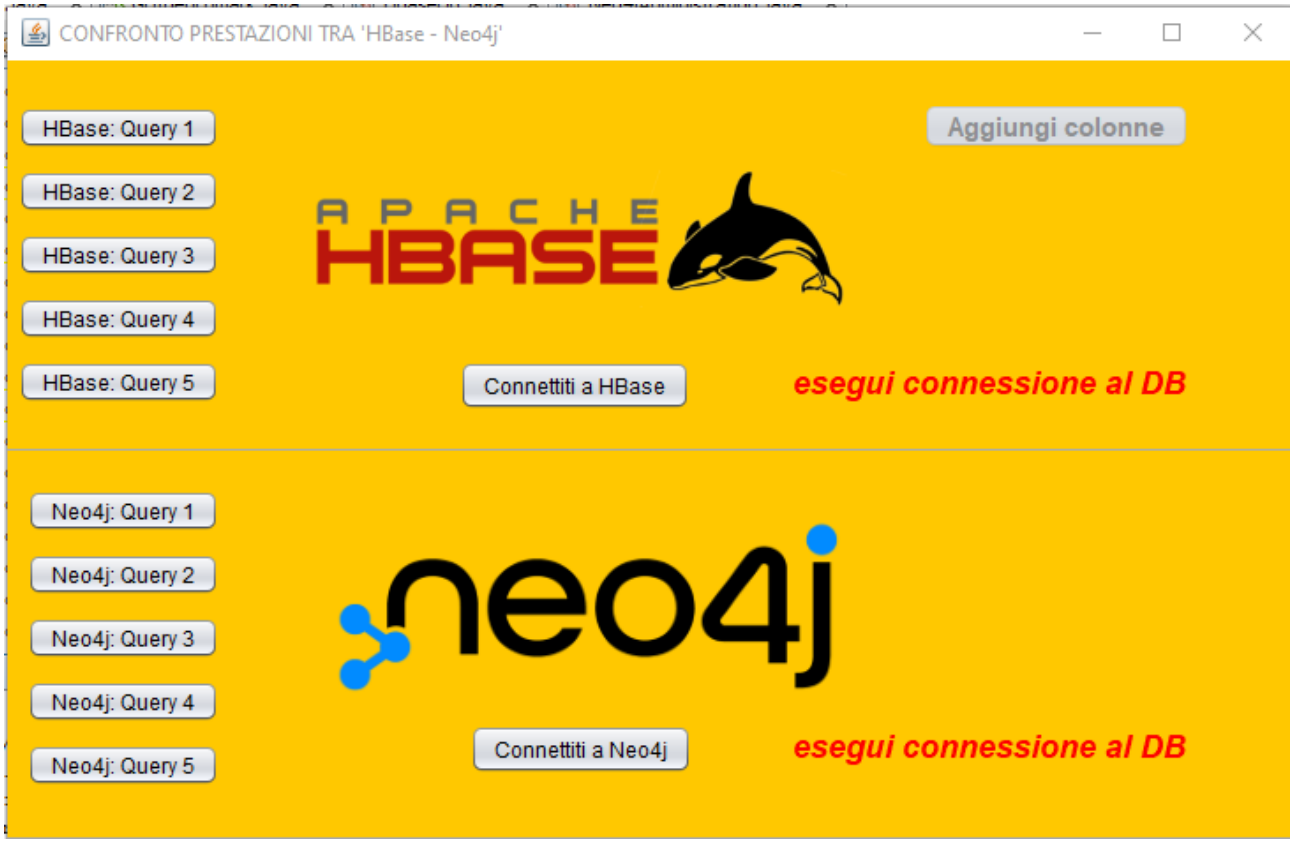
Maven Repository: org.apache.hbase » hbase-client » 2.4.8 (mvnrepository.com)

- Per Neo4j:

```
<groupId>org.neo4j.driver</groupId>
<artifactId>neo4j-java-driver</artifactId>
<version>4.4.2</version>
```

Maven Repository: org.neo4j.driver » neo4j- java-driver » 4.4.2 (mvnrepository.com)

Per rendere il progetto più dinamico abbiamo creato una **GUI** che ci permette di connetterci ai due database e di eseguire le 5 query cliccando sui bottoni.



A ogni tasto è associato un evento gestito mediante un gestore di eventi **ActionListener** che implementa un metodo chiamato actionPerformed; questo metodo in base all'evento che si verifica, risponde a quel determinato evento associando il rispettivo comportamento cioè l'esecuzione delle query.
Di seguito è riportato un esempio del metodo **ActionPerformed** associato al bottone che esegue la query 1 di Hbase.

```
private void buttonHbaseQuery1ActionPerformed(java.awt.event.ActionEvent evt) {
    // TODO add your handling code here:
    HbaseAdministration hbase;
    try {
        hbase = new HbaseAdministration();
        hbase.timeQuery1();// eseguo query 1
        buttonHbaseQuery1.setText("Query 1 Hbase eseguita");
    } catch (IOException ex) {
        Logger.getLogger(GUIbenchmark.class.getName()).log(Level.SEVERE, null, ex);
    }
}
```

Implementazione in Neo4J

Per quanto riguarda Neo4j, inizialmente abbiamo importato il dataset creato in formato CSV tramite apposita operazione di import.

Successivamente abbiamo creato il grafo utilizzando i seguenti **comandi Cypher**:

- **CREATE CONSTRAINT** per creare dei vincoli di univocità per ciascuna entità sulle chiavi primarie.

```
CREATE CONSTRAINT ON (p:Person) assert p.number is unique;
CREATE CONSTRAINT ON (s:Society) assert s.id is unique;
CREATE CONSTRAINT ON (c:BankAccount) assert c.id is unique;
```

- **LOAD CSV** per importare il dataset in formato csv all'interno del database

```
LOAD CSV WITH HEADERS FROM "file:///100000_records.csv" AS row
```

- **MERGE** che rappresenta una combinazione tra il comando CREATE e MATCH, in particolare se il nodo esiste restituisce i risultati, altrimenti se il nodo non esiste lo crea e restituisce i risultati.
  ON CREATE SET che ci permette di stabilire i campi da estrarre dal file csv e da passare ai vari nodi.

Nel nostro caso il comando MERGE è stato utilizzato per creare i nodi Person, Society, BankAccount mentre il comando ON CREATE SET ci ha permesso di associare gli attributi corretti ai nodi.

```
MERGE (p1:Person {number: row.id_conto_P1})
ON CREATE SET p1.nome=row.nome, p1.cognome=row.cognome, p1.nome_cognome=row.nome_cognomeP1, p1.tel=row.tel, p1.email=row.email, p1.indirizzo=row.indirizzo
ON MATCH SET p1.nome=row.nome, p1.cognome=row.cognome, p1.nome_cognome=row.nome_cognomeP1, p1.tel=row.tel, p1.email=row.email, p1.indirizzo=row.indirizzo
MERGE (p2:Person {number: row.id_conto_P2})
ON CREATE SET p2.nome_cognome=row.nome_cognomeP2, p2.professione=row.professione
ON MATCH SET p2.nome_cognome=row.nome_cognomeP2, p2.professione=row.professione
MERGE (s1:Society {id: row.id_societa1})
ON CREATE SET s1.nome_societa=row.nome_societa, s1.paese=row.paese, s1.citta=row.citta
ON MATCH SET s1.nome_societa=row.nome_societa, s1.paese=row.paese, s1.citta=row.citta
MERGE (s2:Society {id: row.id_societa2})
ON CREATE SET s2.nome_societa=row.nome_societa, s2.paese=row.paese, s2.citta=row.citta
ON MATCH SET s2.nome_societa=row.nome_societa, s2.paese=row.paese, s2.citta=row.citta
MERGE (c:BankAccount {id: row.id_credito_banca})
ON CREATE SET c.tipo_tessera=row.tipo_tessera, c.data_apertura=row.data_apertura, c.data_transazione=row.data_transazione, c.importo=row.importo;
```

- **DROP CONSTRAINT** per eliminare i precedenti vincoli.

```
DROP CONSTRAINT ON (p:Person) ASSERT p.number IS UNIQUE;
DROP CONSTRAINT ON (s:Society) ASSERT s.id IS UNIQUE;
DROP CONSTRAINT ON (c:BankAccount) ASSERT c.id IS UNIQUE;
```

- **CREATE INDEX** per aggiungere degli indici a number di Person e a id di Society e Bank Account.

```
CREATE INDEX ON :Person(number);
CREATE INDEX ON :Society(id);
CREATE INDEX ON :BankAccount(id);
```

- **MATCH** utilizzato nella creazione delle relazioni e ci permette di stabilire dei legami tra le chiavi primarie delle entità coinvolte nella relazione.
  Le relazioni che abbiamo creato sono le seguenti:
  - La relazione P_effettua_t → tra la persona che effettua la transazione e il conto corrente della persona ricevente.
    La relazione Conto_di → tra il conto corrente e la persona posseditore del conto.

```
:auto USING PERIODIC COMMIT 100000
LOAD CSV WITH HEADERS FROM "file:///100000_records.csv" AS row
MATCH (p1:Person {number: row.id_conto_P1}), (c:BankAccount {id: row.id_credito_banca}), (p2:Person {number: row.id_conto_P2})
CREATE (p1)-[:P_effettua_t{data_transazione:row.data_transazione, importo:row.importo}]->(c)-[:Conto_di]->(p2);
```

  - La relazione S_effettua_t → tra la società che effettua la transazione e il conto corrente della persona ricevente.

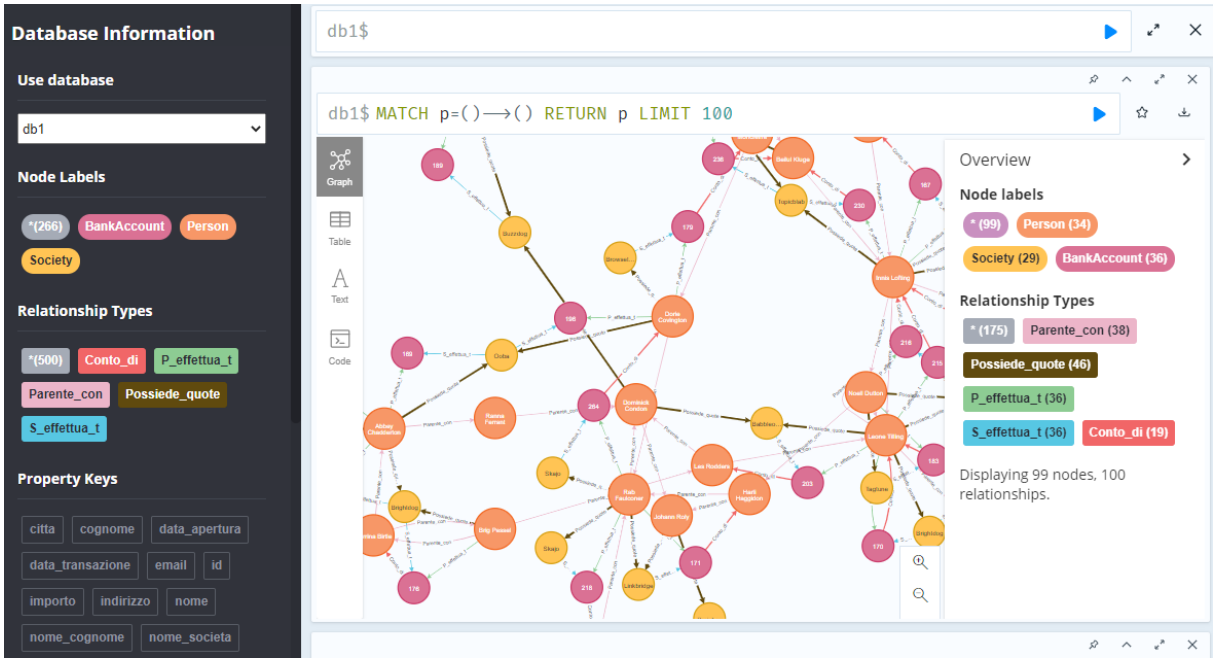
```
:auto USING PERIODIC COMMIT 100000
LOAD CSV WITH HEADERS FROM "file:///100000_records.csv" AS row
MATCH (s2:Society {id: row.id_societa2}), (c:BankAccount {id: row.id_credito_banca})
CREATE (s2)-[:S_effettua_t{data_transazione:row.data_transazione, importo:row.importo}]->(c);
```

  - La relazione Possiede_quote → tra la persona e la società posseduta da quella persona.

```
:auto USING PERIODIC COMMIT 100000
LOAD CSV WITH HEADERS FROM "file:///100000_records.csv" AS row
MATCH (p1:Person {number: row.id_conto_P1}), (s2:Society {id: row.id_societa2})
CREATE (p1)-[:Possiede_quote{quote:row.quote}]->(s2);
```

- La relazione Parente_con → tra due entità Person che sono parenti.

```
:auto USING PERIODIC COMMIT 100000
LOAD CSV WITH HEADERS FROM "file:///100000_records.csv" AS row
MATCH (p1:Person {number: row.id_conto_P1}), (p2:Person {number: row.id_conto_P2})
CREATE (p2)-[:Parente_con]->(p1);
```



Esempio di visualizzazione del grafo sull'interfaccia Neo4j utilizzando il linguaggio di querying Cypher.

Dopo aver creato il grafo attraverso il linguaggio di programmazione Java abbiamo definendo all'interno del costruttore della classe Neo4jAdministration() l'uri (corrispondente alla bolt port), lo user, la password e un driver per un'istanza di neo4j.

```
public class Neo4jAdministration implements AutoCloseable{
    private final Driver driver;
    String uri = "bolt://localhost:7687";
    String user = "neo4j";
    String password = "3647";

    public Neo4jAdministration(){
        this.uri = uri;
        this.user = user;
        this.password = password;
        driver = GraphDatabase.driver(this.uri, AuthTokens.basic(this.user, this.password));
    }
```

Successivamente con il metodo connectionNeo4j() abbiamo eseguito la connessione a Neo4j.

```
public boolean connectionNeo4j(){
    try(Session session = driver.session()) {
        Neo4jAdministration db = new Neo4jAdministration();
        System.out.println("\n*** CONNESSIONE STABILITA CON NEO4J ***\n");
        return true;
    } catch(Exception e){
        System.out.println("\n*** Errore di connessione ***\n");
        return false;
    }
}
```

Implementazione in Hbase

Per quanto riguardata HBase è stata scelta una **configurazione standalone** 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 abbiamo caricato i nostri dati sperimentali utilizzando la libreria Java "commons.csv" per la lettura dei file in formato csv, e in seguito, tramite API Java "hbase.client" con il comando "put", sono stati caricati i dati nelle corrispondenti Column Family.

```
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) {
            String rowKey = csvRecord.get("id");
            Put put = new Put(Bytes.toBytes(rowKey));

            for(int i=1;i<column.length;i++){
                put.addColumn(columnFamily, Bytes.toBytes(column[i]),Bytes.toBytes(csvRecord.get(column[i])));
            }
            try {
               table.put(put);
            }
            catch (IOException e) {
            }
```

Per la creazione del Namespace e della tabella contenente le 3 Column family (Person, Society, BankAccount) in HBase, sono stati utilizzati i seguenti comandi:

```
public void createNamespaceAndTable(final Admin admin, String name_space, TableName table_name,
        byte[] columnFamilyPerson, byte[] columnFamilySociety, byte[] columnFamilyBankAccount) throws IOException {
    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(columnFamilySociety) +  ", "
            + Bytes.toString(columnFamilyBankAccount) + "].");
        TableDescriptor desc = TableDescriptorBuilder.newBuilder(table_name)
            .setColumnFamily(ColumnFamilyDescriptorBuilder.of(columnFamilyPerson))
            .setColumnFamily(ColumnFamilyDescriptorBuilder.of(columnFamilySociety))
            .setColumnFamily(ColumnFamilyDescriptorBuilder.of(columnFamilyBankAccount))
            .build();
        admin.createTable(desc);
    }
}
```

Per creare la connessione con HBasemaster tramite API Java abbiamo definito il seguente metodo.

```
public boolean connectionHBase() throws IOException{
    try{
        Configuration config = HBaseConfiguration.create();
        Connection connection = ConnectionFactory.createConnection(config);
        Admin admin = connection.getAdmin();
        System.out.println("\n*** CONNESSIONE STABILITA CON HBASE ***\n");
        hbase.createNamespaceAndTable(admin,MY_NAMESPACE_NAME, MY_TABLE_NAME, MY_COLUMN_FAMILY_NAME_PERSON,
                MY_COLUMN_FAMILY_NAME_SOCIETY,MY_COLUMN_FAMILY_NAME_BANKACCOUNT);
        this.table = connection.getTable (MY_TABLE_NAME);
        return true;
    }
    catch(IOException e){
        return false;
    }
}
```

Implementazione in query

Una volta inseriti i dati vengono eseguite per entrambi i DB cinque query. Per ciascuna query, inoltre, abbiamo implementato un metodo che calcola i tempi di esecuzione (della rispettiva query) 31 volte e che stampa il risultato della query.

```
public void timeQuery1() throws IOException, Exception{
    double millisecondi = Math.pow(10,6);
    FileWriter w;
    //String percorso = System.getProperty("user.dir") + File.separator + "/Query_Time/Neo4j/100_records/Query1";
    //String percorso = System.getProperty("user.dir") + File.separator + "/Query_Time/Neo4j/1000_records/Query1";
    //String percorso = System.getProperty("user.dir") + File.separator + "/Query_Time/Neo4j/10000_records/Query1";
    String percorso = System.getProperty("user.dir") + File.separator + "/Query_Time/Neo4j/100000_records/Query1";
    w = new FileWriter(percorso);
    System.out.println("Esecuzione query 1 ---> Neo4j");
    for(int i=0; i<31; i++){
        long start = System.nanoTime();
        esecuzioneQuery1();
        long end = System.nanoTime();
        String total = String.valueOf((end - start) / millisecondi);
        w.write(total + " \n");
    }
    query1();
    System.out.println("Fine query 1 ---> Neo4j\n");
    w.flush();
    w.close();
    close(); //driver.close
}
```

Qui possiamo vedere un esempio di tale metodo per calcolare i tempi sulla query 1 di Neo4j.

QUERY 1

Trovare tutti i conti bancari posseduti da "Harli Haggidon".

- NEO4J

```
public void esecuzioneQuery1(){
    try(Session session = driver.session()){
        Result result = session.run("MATCH (c:BankAccount) - [:Conto_di] -> (p:Person{nome_cognome: 'Harli Haggidon'})\n" +
                                    "RETURN p, c");
    }catch(Exception ex){
        System.out.println("query non eseguita");
    }
}
```

- HBASE

```
public void esecuzioneQuery1() throws IOException{
    Scan scan = new Scan();
    SingleColumnValueFilter filter = new SingleColumnValueFilter(HbaseAdministration.MY_COLUMN_FAMILY_NAME_BANKACCOUNT,
        Bytes.toBytes("nome_cognomeP2"), CompareOp.EQUAL, Bytes.toBytes("Harli Haggidon"));
    scan.setFilter(filter);
    ResultScanner scanner = table.getScanner(scan);
}
```

QUERY 2

Trovare tutte le società possedute da "Torie Pickersgill" con sede in Russia (RU).

- NEO4J

```
public void esecuzioneQuery2(){
    try(Session session = driver.session()){
        Result result = session.run("MATCH (p:Person{nome_cognome:'Torie Pickersgill'})- [:Possiede_quote] -> (s:Society {paese:'RU'})\n" +
                                    "RETURN p, s");
    }catch(Exception ex){
        System.out.println("query non eseguita");
    }
}
```

- HBASE

```
public void esecuzioneQuery2() throws IOException{
    List<Filter> filters = new ArrayList<>();
    Scan scan = new Scan();
    SingleColumnValueFilter filter1 = new SingleColumnValueFilter(HbaseAdministration.MY_COLUMN_FAMILY_NAME_PERSON,
        Bytes.toBytes("nome_cognomeP1"), CompareOp.EQUAL, Bytes.toBytes("Torie Pickersgill"));
    filters.add(filter1);
    SingleColumnValueFilter filter2 = new SingleColumnValueFilter(HbaseAdministration.MY_COLUMN_FAMILY_NAME_SOCIETY,
        Bytes.toBytes("paese"), CompareOp.EQUAL, Bytes.toBytes("RU"));
    filters.add(filter2);
    FilterList filterList = new FilterList(FilterList.Operator.MUST_PASS_ALL, filters);
    scan.setFilter(filterList);
    ResultScanner scanner = HbaseAdministration.table.getScanner(scan);
}
```

QUERY 3

Trovare tutte le società in cui "Rab Faulconer" possiede delle quote sociali superiori al 25%.

- NEO4J

```
public void esecuzioneQuery3(){
    try(Session session = driver.session()){
        Result result = session.run("MATCH (p:Person{nome_cognome: 'Bret Van Schafflaer'}) - [possesso:Possiede_quote] -> (s:Society)\n" +
                                    "WHERE possesso.quote > '25.0'\n" +
                                    "RETURN s, p");
    }catch(Exception ex){
        System.out.println("query non eseguita");
    }
}
```

- HBASE

```
public void esecuzioneQuery3() throws IOException{
    List<Filter> filters = new ArrayList<>();
    Scan scan = new Scan();
    SingleColumnValueFilter filter1 = new SingleColumnValueFilter(HbaseAdministration.MY_COLUMN_FAMILY_NAME_PERSON,
        Bytes.toBytes("nome_cognomeP1"), CompareOp.EQUAL, Bytes.toBytes("Rab Faulconer"));
    filters.add(filter1);
    SingleColumnValueFilter filter2 = new SingleColumnValueFilter(HbaseAdministration.MY_COLUMN_FAMILY_NAME_SOCIETY,
        Bytes.toBytes("quote"), CompareOp.GREATER, Bytes.toBytes("25.0"));
    filters.add(filter2);
    FilterList filterList = new FilterList(FilterList.Operator.MUST_PASS_ALL, filters);
    scan.setFilter(filterList);
    ResultScanner scanner = HbaseAdministration.table.getScanner(scan);
}
```

QUERY 4

Trovare la persona di nome "Dominick Condon" che ha effettuato delle transazioni con un importo > di 45000 euro sul conto corrente di "Les Rodders".

- NEO4J

```
public void esecuzioneQuery4(){
    try(Session session = driver.session()){
        Result result = session.run("MATCH (p1:Person{nome_cognome: 'Dominick Condon'}) - [soldi:P_effettua_t] -> (c:BankAccount), \n" +
                                    "(c:BankAccount) - [:Conto_di] -> (p2:Person{nome_cognome: 'Les Rodders'})\n" +
                                    "WHERE soldi.importo>'€45000'\n" +
                                    "RETURN c, p1, p2");
    }catch(Exception ex){
        System.out.println("query non eseguita");
    }
}
```

- HBASE

```
public void esecuzioneQuery4() throws IOException{
    List<Filter> filters = new ArrayList<>();
    Scan scan = new Scan();
    SingleColumnValueFilter filter1 = new SingleColumnValueFilter(HbaseAdministration.MY_COLUMN_FAMILY_NAME_PERSON,
        Bytes.toBytes("nome_cognomeP1"), CompareOp.EQUAL, Bytes.toBytes("Dominick Condon"));
    filters.add(filter1);
    SingleColumnValueFilter filter2 = new SingleColumnValueFilter(HbaseAdministration.MY_COLUMN_FAMILY_NAME_BANKACCOUNT,
        Bytes.toBytes("nome_cognomeP2"), CompareOp.EQUAL, Bytes.toBytes("Les Rodders"));
    filters.add(filter2);
    SingleColumnValueFilter filter3 = new SingleColumnValueFilter(HbaseAdministration.MY_COLUMN_FAMILY_NAME_BANKACCOUNT,
        Bytes.toBytes("importo"), CompareOp.GREATER, Bytes.toBytes("€45000"));
    filters.add(filter3);
    FilterList filterList = new FilterList(FilterList.Operator.MUST_PASS_ALL, filters);
    scan.setFilter(filterList);
    ResultScanner scanner = HbaseAdministration.table.getScanner(scan);
}
```

QUERY 5

Trovare la società che ha effettuato delle transazioni il 15 Gennaio del 2044 sul conto corrente posseduto da "Dorotea Menure" che è parente con delle persone che possiedono quote sociali maggiori al 25% riguardanti la società stessa.

- NEO4J

```
public void esecuzioneQuery5(){
    try(Session session = driver.session()){
        Result result = session.run("MATCH (s:Society) - [data:S_effettua_t] -> (c:BankAccount), \n" +
                                    "(c:BankAccount) - [:Conto_di] -> (p1:Person{nome_cognome: 'Dorotea Menure'}), \n" +
                                    "(p1:Person) - [:Parente_con] -> (p2:Person), \n" +
                                    "(p2:Person) - [possiede:Possiede_quote] -> (s:Society)\n" +
                                "WHERE data.data_transazione = '15/01/2044' AND possiede.quote > '25.0'\n" +
                                "RETURN s, c, p1, p2");
    }catch(Exception ex){
        System.out.println("query non eseguita");
    }
}
```

- HBASE

```
public void esecuzioneQuery5() throws IOException{
    List<Filter> filters = new ArrayList<>();
    Scan scan = new Scan();
    SingleColumnValueFilter filter1 = new SingleColumnValueFilter(HbaseAdministration.MY_COLUMN_FAMILY_NAME_BANKACCOUNT,
        Bytes.toBytes("data_transazione"), CompareOp.EQUAL, Bytes.toBytes("15/01/2044"));
    filters.add(filter1);
    SingleColumnValueFilter filter2 = new SingleColumnValueFilter(HbaseAdministration.MY_COLUMN_FAMILY_NAME_BANKACCOUNT,
        Bytes.toBytes("nome_cognomeP2"), CompareOp.EQUAL, Bytes.toBytes("Dorotea Menure"));
    filters.add(filter2);
    SingleColumnValueFilter filter3 = new SingleColumnValueFilter(HbaseAdministration.MY_COLUMN_FAMILY_NAME_SOCIETY,
        Bytes.toBytes("quote"), CompareOp.GREATER, Bytes.toBytes("25.0"));
    filters.add(filter3);
    FilterList filterList = new FilterList(FilterList.Operator.MUST_PASS_ALL, filters);
    scan.setFilter(filterList);
    ResultScanner scanner = HbaseAdministration.table.getScanner(scan);
}
```

Esperimenti

Gli esperimenti effettuati mirano a confrontare le prestazioni riguardanti i tempi di esecuzione delle query (con gradi di complessità crescenti) al crescere del dataset per **Neo4j** e **Hbase**.
Per effettuare gli esperimenti, sono stati valutati dataset crescenti considerando una dimensione variabile del 25%, 50%, 75% e 100%.

In particolare sono stati considerati 4 dataset con:

- **100 record**,
- **1000 record**,
- **10000 record**,
- **100000 record**.

Per ogni query è stata inserita una tabella contenente:

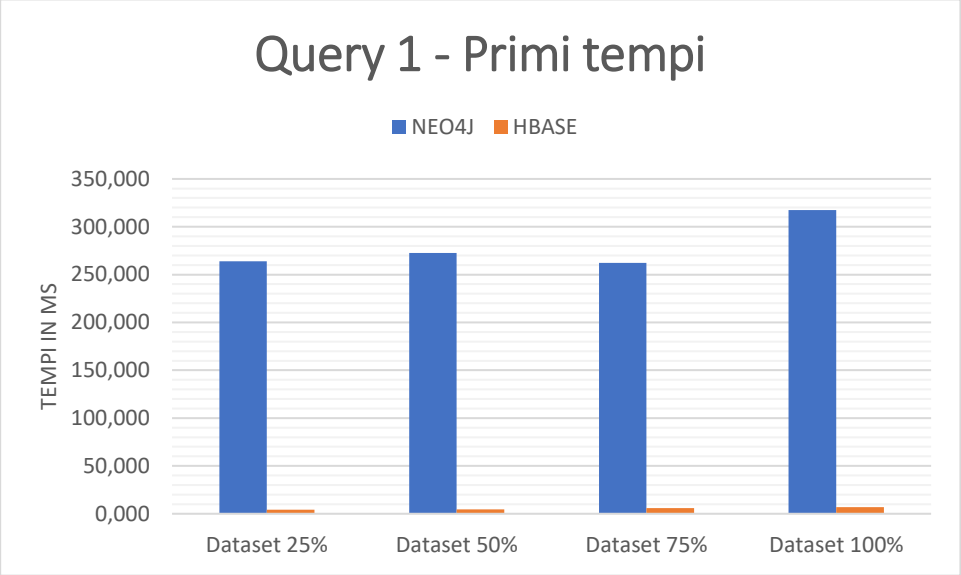
- il 1° tempo di esecuzione che viene preso a parte in quanto la prima esecuzione di una query impiega tendenzialmente più tempo rispetto a quelle successive,
- la media dei tempi delle 30 esecuzioni successive,
- la deviazione standard,
- l'intervallo di confidenza al 95%.

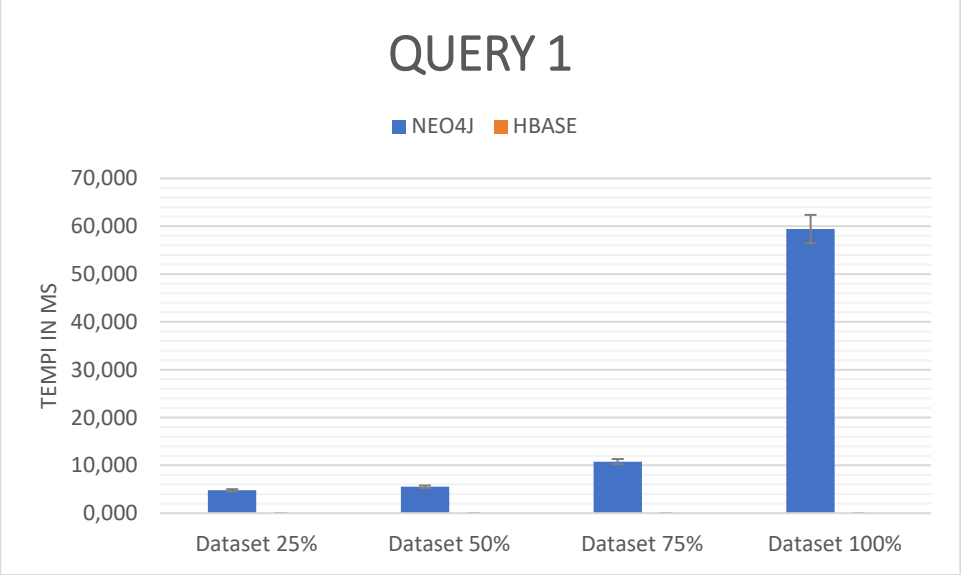
Inoltre, per ogni query sono anche presenti due istogrammi:

- il primo che mostra l'andamento del **1° tempo di esecuzione** al crescere del dataset,
- il secondo che mostra l'andamento della **media dei tempi**, con il relativo intervallo di confidenza al crescere del dataset.

Query 1

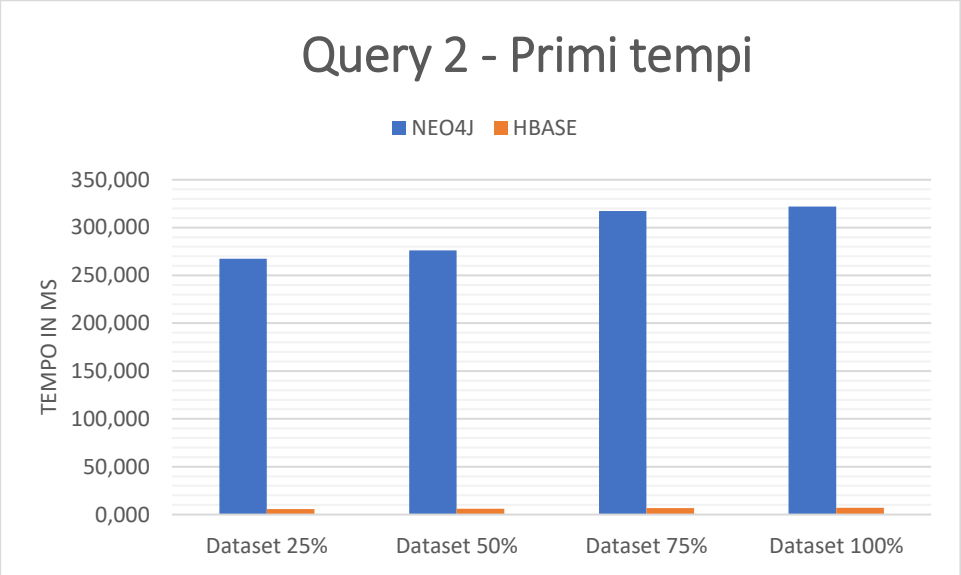
	NEO4J 100	HBASE 100	NEO4J 1000	HBASE 1000	NEO4J 10000	HBASE 10000	NEO4J 1000000	HBASE 1000000
PRIMI TEMPI	263,813727	4,244301	272,647491	4,566863	262,31957	5,993177	317,501995	6,803616
MEDIA	4,786439	0,0205902	5,518166833	0,026567267	10,79354603	0,0252954	59,3998598	0,029161467
DEVIAZIONE STANDARD	1,34121606	0,006992546	1,377197852	0,008928468	1,154860617	0,006381717	4,508378032	0,013416579
INTERVALLO DI CONFIDENZA 95%	0,479939184	0,002502204	0,492814866	0,003194952	0,413253971	0,002283626	1,613272715	0,004800973

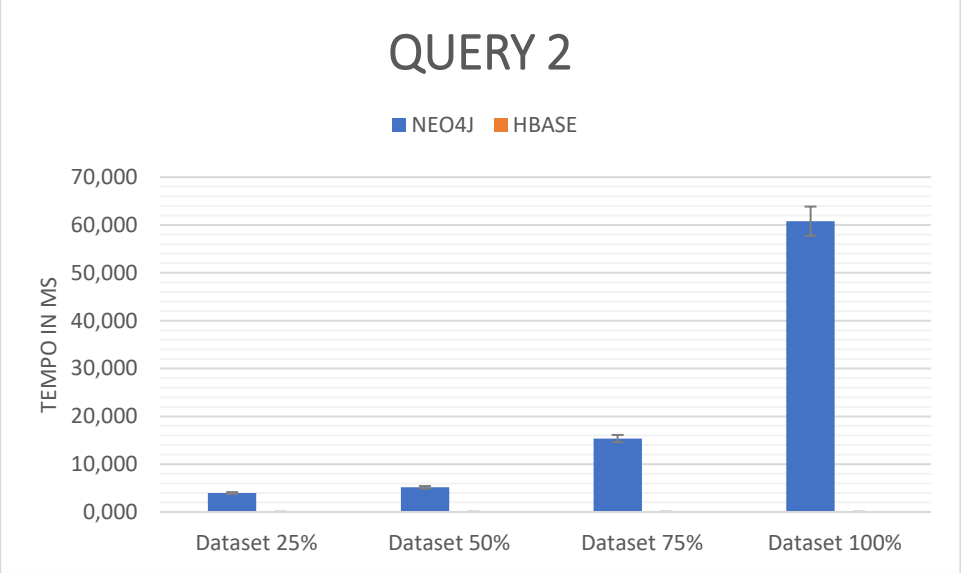




Query 2

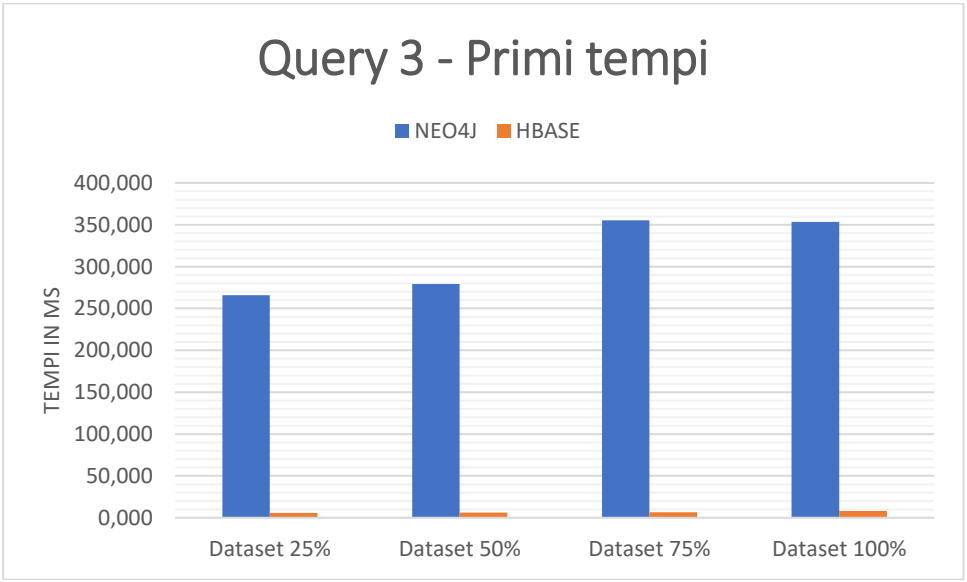
	NEO4J 100	HBASE 100	NEO4J 1000	HBASE 1000	NEO4J 10000	HBASE 10000	NEO4J 1000000	HBASE 1000000
PRIMI TEMPI	267,552703	5,797834	276,168923	6,101992	317,426028	6,8313	321,990618	6,988161
MEDIA	3,9602564	0,029497733	5,1759557	0,028801333	15,34505067	0,0289937	60,81124903	0,028427867
DEVIAZIONE STANDARD	0,84426392	0,010129673	1,119254336	0,007973672	4,070282531	0,006844469	4,747516389	0,007608393
INTERVALLO DI CONFIDENZA 95%	0,302110412	0,00362479	0,400512661	0,00285329	1,456505134	0,002449217	1,698845704	0,002722578

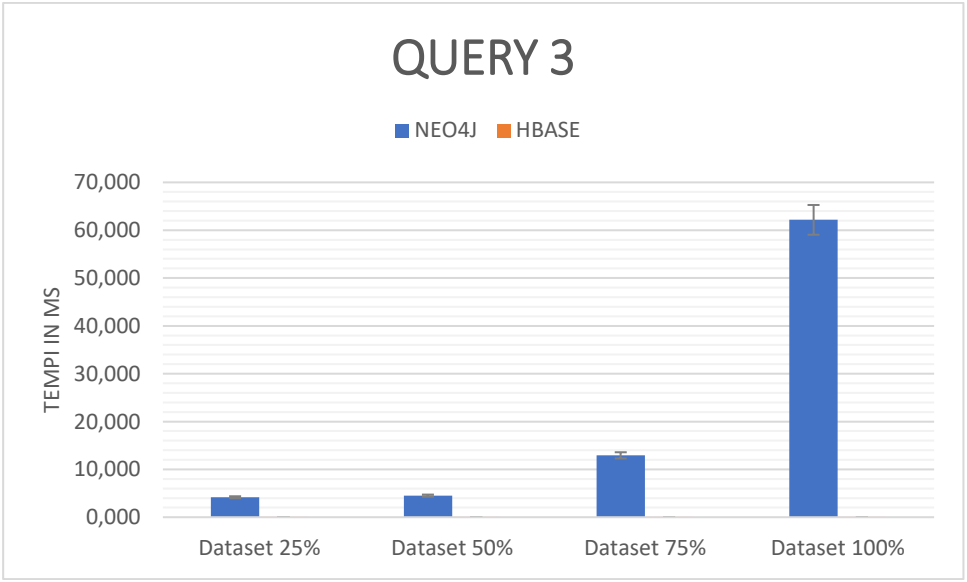




Query 3

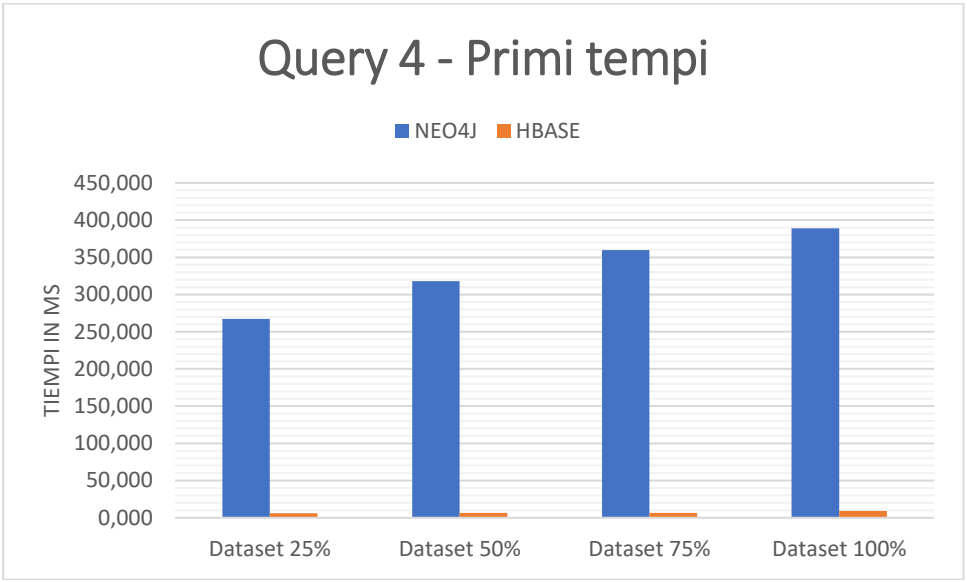
	NEO4J 100	HBASE 100	NEO4J 1000	HBASE 1000	NEO4J 10000	HBASE 10000	NEO4J 1000000	HBASE 1000000
PRIMI TEMPI	265,724517	5,745856	279,297819	6,036259	355,235929	6,566041	353,267768	8,005782
MEDIA	4,168580367	0,0287878	4,5032409	0,034397533	12,9370085	0,034342667	62,17057247	0,034409667
DEVIAZIONE STANDARD	0,991365412	0,008374141	1,090022861	0,010916185	3,615556453	0,009065587	4,421574263	0,008787202
INTERVALLO DI CONFIDENZA 95%	0,354749038	0,002996593	0,390052504	0,003906235	1,293786486	0,003244019	1,582210956	0,003144402

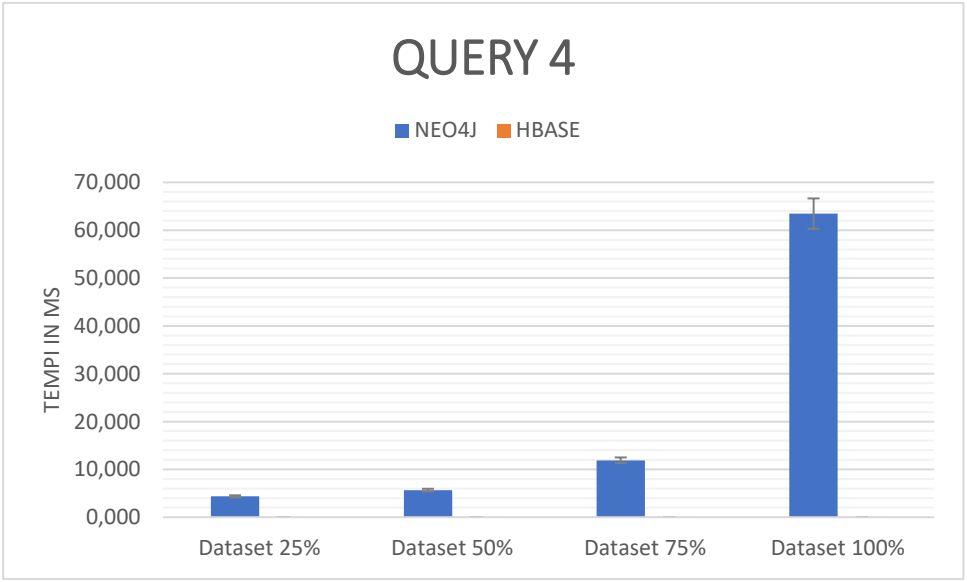




Query 4

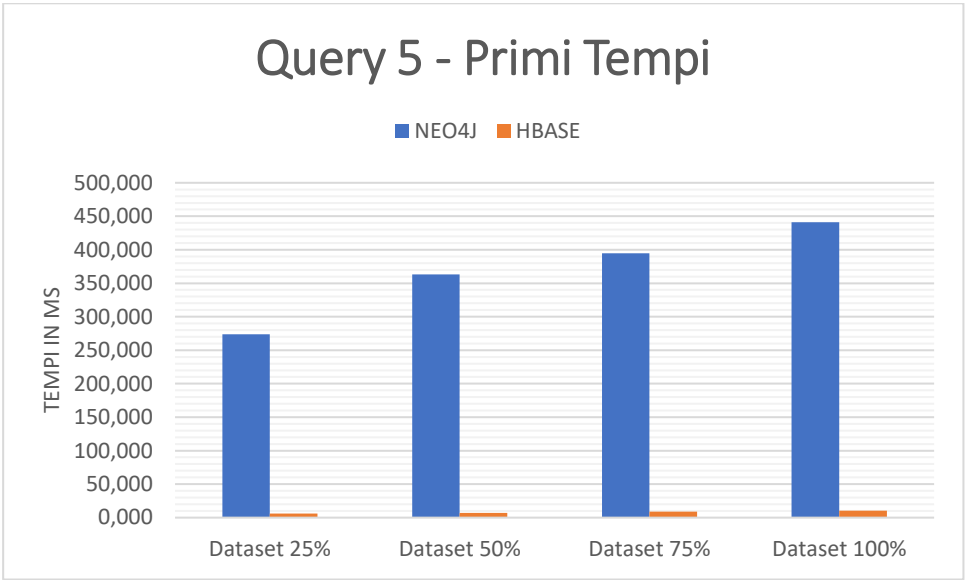
	NEO4J 100	HBASE 100	NEO4J 1000	HBASE 1000	NEO4J 10000	HBASE 10000	NEO4J 1000000	HBASE 1000000
PRIMI TEMPI	267,353364	6,120904	317,893334	6,576802	359,559338	6,650602	388,803169	9,163724
MEDIA	4,362516567	0,031484733	5,660806767	0,0395623	11,9026599	0,0389162	63,47404537	0,030347933
DEVIAZIONE STANDARD	0,977403377	0,009582069	0,965217923	0,010942386	1,179997224	0,010532049	5,567329243	0,006805731
INTERVALLO DI CONFIDENZA 95%	0,34975288	0,003428836	0,345392451	0,003915611	0,422248825	0,003768776	1,992206576	0,002435355

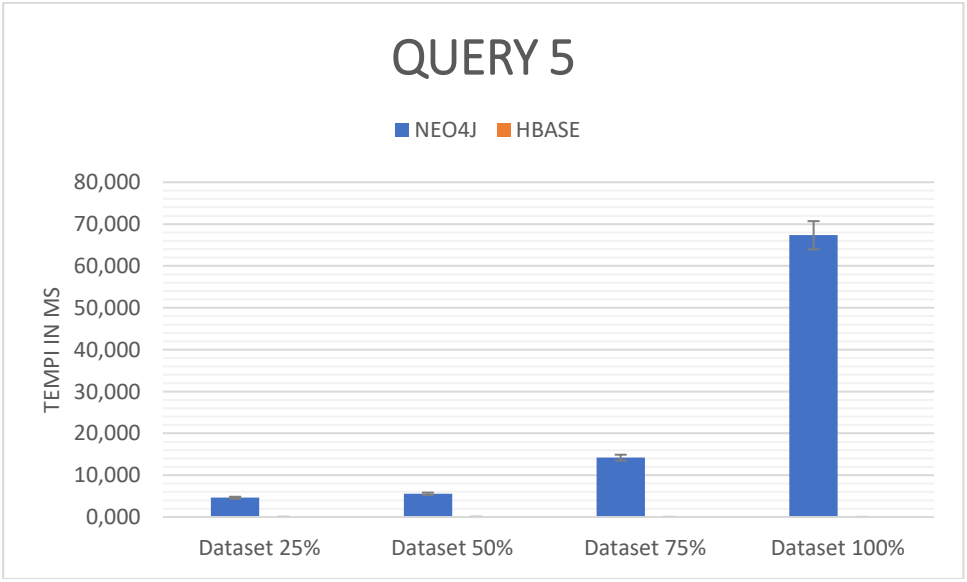




Query 5

	NEO4J 100	HBASE 100	NEO4J 1000	HBASE 1000	NEO4J 10000	HBASE 10000	NEO4J 1000000	HBASE 1000000
PRIMI TEMPI	273,949639	6,049391	363,092547	6,932614	394,629763	8,780198	441,16685	10,520728
MEDIA	4,625556033	0,041886367	5,567804867	0,049475867	14,19332933	0,031896933	67,3422322	0,028828967
DEVIAZIONE STANDARD	1,149764634	0,014258961	1,294449534	0,021561257	8,145239035	0,0060142	12,12669013	0,005334378
INTERVALLO DI CONFIDENZA 95%	0,41143043	0,00510241	0,463204305	0,007715455	2,914682796	0,002152114	4,339400592	0,001908848





Conclusioni

Dagli esperimenti effettuati possiamo dire di avere un vincitore in quanto risulta che in tutti i casi, **HBase risponde più velocemente rispetto a Neo4j per tutte le query esaminate**.

- 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.

In conclusione da questa indagine possiamo concludere che nonostante la complessità delle query aumenta il DBMS Hbase è senza dubbio migliore per modellare delle situazioni come quelle prese in carico secondo questo studio sull'antiriciclaggio di denaro.