RELAZIONE TECNICA

SVILUPPO E SPERIMENTAZIONE DELLE QUERY SU GRAFI: SPARK, CYPHER, MORPHEUS, GRAPHX

Giovanni Martucci

M: 1000012435

Table of Contents

1. CENNI SULLA TEORIA

1.1. Spark

Apache Spark è una piattaforma open source per l'elaborazione di analisi dei dati su larga scala, progettata per essere veloce e generica.
Non include un sistema di gestione dei dati e pertanto viene solitamente distribuito su Hadoop o su altre piattaforme di archiviazione.
La sua caratteristica più importante è il suo cluster computing in memoria che è responsabile di aumentare la velocità di elaborazione dei dati. Le applicazioni Spark vengono eseguite come serie indipendenti di processi su un cluster, coordinate dall'oggetto SparkContext (Master) nel programma principale. Permette di schedulare il problema su diversi nodi del cluster, rendendo l'elaborazione in parallelo, distribuendo i vari carichi di lavoro tra i vari processori del cluster.
Fornisce due modalità di esecuzione: Cluster Mode vs Client Mode.
Fornisce API scritte in Scala, Java, Python o R.

1.2. Cypher

Cypher è un linguaggio di interrogazione su grafo dichiarativo che consente query espressive ed efficienti nel grafo. È progettato per essere adatto sia agli sviluppatori che ai professionisti delle operazioni. Cypher è progettato per essere semplice, ma potente; query di database altamente complicate possono essere facilmente espresse.
Originariamente concepito per essere utilizzato con il database grafico Neo4j, è stato poi aperto tramite il progetto openCypher nell'ottobre 2015.

```
//node
(variable:Label {propertyKey: 'propertyValue'})

//relationship
-[variable:RELATIONSHIP_TYPE]->

//Cypher pattern
(node1:LabelA)-[rel1:RELATIONSHIP_TYPE]->(node2:LabelB)
```

Figura X. Esempio query Cypher.

1.3. Morpheus

Morpheus estende Apache Spark™ con Cypher , il linguaggio di query di grafici di proprietà più utilizzato del settore, definito e gestito dal progetto openCypher, come discusso nel paragrafo precedente. Consente l'integrazione di molte origini dati e supporta l'interrogazione di più grafici. Consente di utilizzare il cluster Spark per eseguire query di grafici analitici. Le query possono anche restituire grafici per creare pipeline di elaborazione.

1.4. GraphX

Come detto in precedenza GraphX è uun componente di Spark, una libreria per manipolare grafici (ad es. Un grafico di un social network) ed eseguire calcoli grafico-paralleli. È un potente tool che permette di unificare il processo ETL (Estrai, Trasforma e Carica), l'analisi esplorativa e il calcolo iterativo dei grafi all'interno di un singolo sistema. Inoltre, GraphX include una raccolta crescente di algoritmi e costruttori di grafi per semplificare le attività di analisi dei grafi, nonché tutta una serie di operazioni (ad esempio Subgraph, joinVertices e aggregateMessages) per operare e rappresentare al meglio i grafi.

2. OBIETTIVO

Questo progetto si prepone l'obiettivo di testare la compatibilità, l'utilizzo, l'efficienza di Cypher in Spark >= 3.0; l'applicazione di query scritte in Cypher sulla rete presa in esame ("Myeloma"); in aggiunta calcolare il degree più alto della rete con GraphX ed infine realizzare uno script che converte le query implementate per Neo4j in formato Spark.

3. SPECIFICHE TECNICHE

Di seguito vengono elencate tutte le specifiche tecniche utilizzate per questo progetto:

- Distro: Linux 18.04;
- Linguaggio di programmazione: Scala, Java e Python (quest'ultimo non supporta GraphX);
- Gestore di pachetti Java, Scala: Maven;
- Cypher 3.0.3, 3.1.2;
- Morpheus 0.4.2.

File pom.xml Cypher/Morpheus:

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-
v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>edu.unict.BigData</groupId>
  <artifactId>my-app</artifactId>
  <packaging>jar</packaging>
  <version>1.0-SNAPSHOT</version>
  <name>my-app</name>
  <url>http://maven.apache.org</url>
  <properties>
    <maven.compiler.source>1.6</maven.compiler.source>
    <maven.compiler.target>1.6</maven.compiler.target>
  </properties>
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</version>
      <scope>test</scope>
    </dependency>
    <!-- https://mvnrepository.com/artifact/org.apache.spark/spark-core -->
    <dependency>
      <groupId>org.apache.spark</groupId>
      <artifactId>spark-core_2.12</artifactId>
      <version>3.0.3</version>
```

```
</dependency>
<!-- https://mvnrepository.com/artifact/org.apache.spark/spark-sql -->
<dependency>
  <groupId>org.apache.spark</groupId>
  <artifactId>spark-sql_2.12</artifactId>
  <version>3.0.3</version>
  <scope>provided</scope>
</dependency>
<dependency>
  <groupId>org.scala-lang</groupId>
  <artifactId>scala-reflect</artifactId>
  <version>2.12.8</version>
</dependency>
<!-- https://mvnrepository.com/artifact/org.scala-lang/scala-compiler -->
<dependency>
  <groupId>org.scala-lang</groupId>
  <artifactId>scala-compiler</artifactId>
  <version>2.12.8</version>
</dependency>
<dependency>
  <groupId>org.opencypher</groupId>
  <artifactId>okapi-relational</artifactId>
  <version>0.4.2</version>
</dependency>
<dependency>
  <groupId>org.opencypher</groupId>
  <artifactId>okapi-relational</artifactId>
  <version>0.4.2</version>
</dependency>
<dependency>
  <groupId>org.opencypher</groupId>
  <artifactId>okapi-ir</artifactId>
  <version>0.4.2</version>
</dependency>
<dependency>
  <groupId>org.apache.commons</groupId>
  <artifactId>commons-lang3</artifactId>
  <version>3.7</version>
</dependency>
<!-- https://mvnrepository.com/artifact/org.atnos/eff -->
<dependency>
  <groupId>org.atnos</groupId>
  <artifactId>eff_2.12</artifactId>
  <version>5.0.0</version>
```

```
 </dependency>
 <dependency>
  <groupId>org.scala-tools</groupId>
  <artifactId>maven-scala-plugin</artifactId>
  <version>2.11</version>
 </dependency>
 <!-- https://mvnrepository.com/artifact/org.neo4j/neo4j -->
 <dependency>
  <groupId>org.neo4j</groupId>
  <artifactId>neo4j</artifactId>
  <version>4.3.2</version>
 </dependency>
 <!-- https://mvnrepository.com/artifact/org.neo4j/neo4j-cypher -->
 <dependency>
  <groupId>org.neo4j</groupId>
  <artifactId>neo4j-cypher</artifactId>
  <version>4.3.2</version>
 </dependency>
 <!-- https://mvnrepository.com/artifact/org.opencypher/spark-cypher -->
 <dependency>
  <groupId>org.opencypher</groupId>
  <artifactId>spark-cypher</artifactId>
  <version>1.0.0-beta7</version>
 </dependency>
 <!-- https://mvnrepository.com/artifact/org.opencypher/morpheus-spark-cypher -->
 <dependency>
  <groupId>org.opencypher</groupId>
  <artifactId>morpheus-spark-cypher</artifactId>
  <version>0.4.2</version>
 </dependency>
</dependencies>
<build>
  <plugins>
      <plugin>
          <groupId>org.scala-tools</groupId>
          <artifactId>maven-scala-plugin</artifactId>
          <executions>
              <execution>
                  <goals>
                      <goal>compile</goal>
                      <goal>testCompile</goal>
                  </goals>
              </execution>
          </executions>
```

```
        </plugin>
    </plugins>
  </build>
</project>
```

File pom.xml GraphX:

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-
v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>edu.unict.BigData</groupId>
  <artifactId>my-app</artifactId>
  <packaging>jar</packaging>
  <version>1.0-SNAPSHOT</version>
  <name>my-app</name>
  <url>http://maven.apache.org</url>
  <properties>
    <maven.compiler.source>1.6</maven.compiler.source>
    <maven.compiler.target>1.6</maven.compiler.target>
  </properties>
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</version>
      <scope>test</scope>
    </dependency>
    <!-- https://mvnrepository.com/artifact/org.apache.spark/spark-core -->
    <dependency>
      <groupId>org.apache.spark</groupId>
      <artifactId>spark-core_2.12</artifactId>
      <version>3.0.3</version>
```

```
  </dependency>
  <!-- https://mvnrepository.com/artifact/org.apache.spark/spark-sql -->
  <dependency>
    <groupId>org.apache.spark</groupId>
    <artifactId>spark-sql_2.12</artifactId>
    <version>3.0.3</version>
    <scope>provided</scope>
  </dependency>
  <dependency>
    <groupId>org.scala-lang</groupId>
    <artifactId>scala-reflect</artifactId>
    <version>2.12.8</version>
  </dependency>
  <!-- https://mvnrepository.com/artifact/org.scala-lang/scala-compiler -->
  <dependency>
    <groupId>org.scala-lang</groupId>
    <artifactId>scala-compiler</artifactId>
    <version>2.12.8</version>
  </dependency>
  <dependency>
    <groupId>org.apache.commons</groupId>
    <artifactId>commons-lang3</artifactId>
    <version>3.7</version>
  </dependency>
  <!-- https://mvnrepository.com/artifact/org.atnos/eff -->
  <dependency>
    <groupId>org.atnos</groupId>
    <artifactId>eff_2.12</artifactId>
    <version>5.0.0</version>
  </dependency>
  <dependency>
    <groupId>org.scala-tools</groupId>
    <artifactId>maven-scala-plugin</artifactId>
    <version>2.11</version>
  </dependency>
  <!-- https://mvnrepository.com/artifact/org.apache.spark/spark-graphx -->
  <dependency>
    <groupId>org.apache.spark</groupId>
    <artifactId>spark-graphx_2.12</artifactId>
    <version>3.0.3</version>
  </dependency>
</dependencies>
<build>
  <plugins>
```

```
        <plugin>
            <groupId>org.scala-tools</groupId>
            <artifactId>maven-scala-plugin</artifactId>
            <executions>
                <execution>
                    <goals>
                        <goal>compile</goal>
                        <goal>testCompile</goal>
                    </goals>
                </execution>
            </executions>
        </plugin>
    </plugins>
  </build>
</project>
```

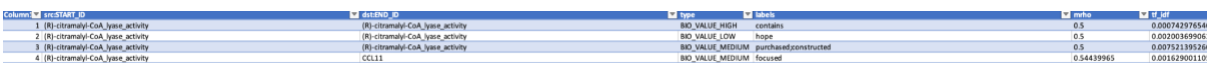
4. INPUT UTILIZZATO

In questo progetto è stata utilizzata la rete denominata “multiple myeloma” con le relative query implementate per Neo4j.
Nello specifico tale rete è rappresentata da due file csv:

- nodes_elab.csv: rappresenta i nodi della rete:

id:ID	names	size	labels	rho
ATF6	ATF6	1636	gene:protein	0.6766465465909092
Calcium	Calcium	8533	drug	0.2054015438301512

- edges_elab.csv: rappresenta le relazioni tra i nodi della rete:



Column	src:START_ID	dst:END_ID	type	labels	mrho	tf_idf
1	(R)-citramalyl-CoA_lyase_activity	(R)-citramalyl-CoA_lyase_activity	BIO_VALUE_HIGH	contains	0.5	0.0007429765
2	(R)-citramalyl-CoA_lyase_activity	(R)-citramalyl-CoA_lyase_activity	BIO_VALUE_LOW	hope	0.5	0.0020036990
3	(R)-citramalyl-CoA_lyase_activity	(R)-citramalyl-CoA_lyase_activity	BIO_VALUE_MEDIUM	purchased;constructed	0.5	0.0075213952
4	(R)-citramalyl-CoA_lyase_activity	CCL11	BIO_VALUE_MEDIUM	focused	0.54439965	0.0016290011

E’ possibile reperire tale input al seguente link:
https://www.dropbox.com/sh/dbjglnydl7ou7ks/AACe8pkRdubgw0WzOerszuoGa?dl=0

Di seguito le query testate:

- Relationships among multiple myeloma disease and generic drugs

```
MATCH (dis:disease {names:"multiple myeloma"})-[r:BIO_VALUE_HIGH]->(drg:drug)
RETURN dis, r, drg LIMIT 10;
```

- Relationships among a disease which name starts with "my"

```
MATCH p=(dis:disease)-[:BIO_VALUE_HIGH]->(gen:gene)
WHERE dis.names =~ '(?i)my.*'
RETURN p LIMIT 10;
```

- Relationships among a disease which name contains "io" and several genes

```
MATCH p=(dis:disease)-[:BIO_VALUE_HIGH]->(gen:gene)
WHERE dis.names CONTAINS 'io'
RETURN p LIMIT 10;
```

- Relationships among diseases and drugs where the names are into two vectors (diseases and drugs)

```
WITH ["multiple myeloma"] AS diseases, ["ixazomib", "auranofin"] AS drugs
MATCH (dis:disease)<-[r:BIO_VALUE_HIGH]-(drg:drug)
WHERE toLower(dis.names) IN diseases AND
      toLower(drg.names) IN drugs
RETURN dis, r, drg;
```

- Path of min length "1" and max one "3" amoug multiple myeloma disease and genes/proteins

```
MATCH p=(dis:disease {names:"multiple myeloma"})-[:BIO_VALUE_HIGH *1..3]->(gp:`gene:protein`)
RETURN p LIMIT 10
```

- Diseases and Drugs triangulation

```
MATCH  p=(dis1:disease {names:"multiple myeloma"})-[:BIO_VALUE_HIGH *1..3]->(drg1:drug),
    q=(dis1)-[:BIO_VALUE_HIGH *1..2]->(dis2:disease),
    r=(dis2)-[:BIO_VALUE_HIGH *1..3]->(drg2:drug),
    t=(drg2)-[:BIO_VALUE_HIGH *2..3]->(drg1)
RETURN p,q,r,t LIMIT 1;
```

- Shortest Path

```
MATCH (dis:disease {names:"multiple myeloma"}), (gen:gene {names: "PARTICL"}),
    p= shortestPath((dis)-[:BIO_VALUE_HIGH *]->(gen))
RETURN p;
```

- Query concatenation

```
MATCH (dis1:disease)<-[r1:BIO_VALUE_HIGH]-(gen1:gene)<-[r2:BIO_VALUE_HIGH]-(gen2:gene)<-
[r3:BIO_VALUE_HIGH]-(dis1)
WHERE gen1.names <> gen2.names
WITH dis1,r1,gen1,r2,gen2,r3 LIMIT 5
MATCH (gen1)-[r4:BIO_VALUE_HIGH]-(drg:drug)-[r5:BIO_VALUE_HIGH]-(gen2)
RETURN dis1,r1,gen1,r2,gen2,r3,drg,r4,r5
```

5. MODALITA' D'ESECUZIONE

Inizialmente si è proceduto con l'inizializzazione e configurazione dell'ambiente di sistema: Linux 18.04. Successivamente la seconda fase è stata l'installazione di Spark 3.1.2.
Sono stati svolti vari test sull'utilizzo di Spark sia da shell che da codice sorgente con la creazione di un progetto Maven: nello specifico sono state utilizzate sia la spark-shell che Pyspark in python; per il lancio, sia di spark-shell che di spark-submit, è stata utilizzata la seguente configurazione:

spark-shell --packages=org.opencypher:morpheus-spark-cypher:0.4.2 --conf
spark.sql.legacy.allowUntypedScalaUDF=True

Dopo aver testato tutto l'ambiente si è proceduto con l'argomento preso sotto esame in questo progetto, installando Cypher e Morpheus (disponibile solo per Scala e Java).

Creazione progetto in Maven:
mvn -B archetype:generate \ -DarchetypeGroupId=org.apache.maven.archetypes -
DgroupId=edu.unict.BigData -DartifactId=my-app

I test iniziali hanno preso in studio Cypher/Morpheus su Spark 3.1.2., ottenendo problemi di incompatibilità; Da qui si è proceduto, dunque, con il downgrade di Spark alla versione minima reperibile (3.0.3). Una volta configurato nuovamente tutto l'ambiente si è proceduto con la fase di testing, lanciando le query fornite sulla rete "Myeloma". Inizialmente vi erano problemi d'incompatibilità della sintassi, questo perché erano state scritte per Neo4j, infatti modificando le parti delle query con la sintassi accettata da Saprk sono tutte andate a buon fine. Da qui nacque il bisogno di dover creare uno script che convertisse le query di Neo4j per Spark: difatti è stato sviluppato uno script python in grado di soddisfare tale richiesta (dettagli dello script al capitolo 7).
Infine è stato installato GraphX per effettuare dei calcoli sulla rete, nel dettaglio si sono cercati i degree dei nodi corrispondenti agli Hub della rete (i nodi con i degree più alti).
Di seguito ecco i codici sorgenti eseguiti:

1. Codice per il lancio delle query Cypher in Spark 3.0.3:

```
package edu.unict.BigData
```

```
import org.apache.spark.{SparkConf,SparkContext}
import scala.math.random
import org.apache.spark.SparkContext._
import org.apache.spark.sql.{SparkSession, DataFrame}
import org.opencypher.morpheus.api.MorpheusSession
import org.opencypher.okapi.api.io.conversion.{NodeMappingBuilder,
RelationshipMappingBuilder}
import org.opencypher.okapi.api.value.CypherValue
import org.opencypher.morpheus.api.io.{MorpheusNodeTable,
MorpheusRelationshipTable, MorpheusElementTable}


object App {

    def main(args: Array[String]): Unit = {

        val spark = SparkSession .builder()
        .appName( name = "meyloma") .config("spark.master","local[*]")
        .getOrCreate()

        implicit val morpheus: MorpheusSession = MorpheusSession.local() import
        spark.sqlContext.implicits._

        val csvOptions = Map("header"->"true", "delimiter" -> ";", "inferSchema" ->
        "true")

        val nodesDF = spark.read.options(csvOptions).csv("nodes_elab.csv")
                val edgesDF = spark.read.options(csvOptions).csv("edges_elab.csv")

        val NodeMapping =
        NodeMappingBuilder.withSourceIdKey("id:ID").withImpliedLabel("Node").
        withPropertyKey("names", "names").withPropertyKey("size",
        "size").withPropertyKey("labels", "labels").withPropertyKey("rho",
        "rho").build

        val RelationMapping =
        RelationshipMappingBuilder.withSourceIdKey("identific").withSourceStartN
        odeKey("
        src:START_ID").withSourceEndNodeKey("dst:END_ID").withRelType("typ
        e").withPro pertyKey("labels", "labels").withPropertyKey("mrho",
        "mrho").withPropertyKey("tf_idf", "tf_idf").build

        val Node = MorpheusElementTable.create(NodeMapping, nodesDF)

        val Relation = MorpheusElementTable.create(RelationMapping, edgesDF)

        val Graph = morpheus.readFrom(Node, Relation)

        val results = Graph.cypher("MATCH (dis:disease {names:'multiple
        myeloma'})- [r:BIO_VALUE_HIGH]->(drg:drug) RETURN dis, r, drg LIMIT
        10;")
```

```
result.show

spark.stop() }}
```

2. Codice per il calcolo dei degree tramite GraphX in Spark 3.0.3

Per questa sezione di progetto sono state apportate delle modifiche ai file csv, inserendo degli identificativi numerici (int) corrispondenti alle colonne “SRC:START_ID” e “DST:END_ID”. Nello specifico si sono effettuate delle join tra gli identificativi dei nodi contenuti nel file nodes_elab.csv con le colonne sopra nominate. E’ stato implementato un ulteriore script adhoc :

```
package it.unict.BigData;

import org.apache.spark.{SparkConf,SparkContext}
import scala.math.random
import org.apache.spark.SparkContext._
import org.apache.spark.sql.{SparkSession, DataFrame}
import org.apache.spark._
import org.apache.spark.graphx._
import org.apache.spark.rdd.RDD

object App
{
   def main(args: Array[String])
   {
      //val spark =
SparkSession.builder().master("local[*]").appName("Graphx").getOrCreate()
         val conf = new SparkConf().setMaster("local[*]").setAppName("Graphx")
      val sc = new SparkContext(conf)

         sc.setLogLevel("ERROR")
         println("spark read csv files from a directory into RDD")
         val rddNodesFile = sc.textFile("nodes_elab.csv")
   val rddEdgesFile = sc.textFile("edges_elab.csv")
         println(rddNodesFile.getClass)
   println(rddEdgesFile.getClass)

         val rdd_nodes = rddNodesFile.map(f=>{f.split(";")})
         val rdd_edges = rddEdgesFile.map(f=>{f.split(";")})
```

```
        val rdd_nodes_ = rdd_nodes.mapPartitionsWithIndex((index, it) => if (index
== 0) it.drop(1) else it, preservesPartitioning = true)
        val rdd_edges_ = rdd_edges.mapPartitionsWithIndex((index, it) => if (index
== 0) it.drop(1) else it, preservesPartitioning = true)


        var rdd_nodes_final: RDD[(VertexId, (String, Int, String))] =
rdd_nodes_.map(node => (
            node(0).toLong, ( node(2), node(3).toInt, node(4) )
   ))

        var rdd_edges_final: RDD[Edge[String]] = rdd_edges_.map(node => Edge(
            node(1).toLong, node(3).toLong, ( node(5) +" "+ node(6) )
        ))


        val defaultNodes = ("",0,"")

        val graph = Graph(rdd_nodes_final, rdd_edges_final, defaultNodes)

        val degrees: VertexRDD[Int] = graph.degrees
        println("\n\n")
        println(degrees.take(10))
        println("\n\n")
        println(degrees)
        println("\n\n")

   }
}
```

3. Script add_identify.py:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
import pandas as pd


def main():
```

```
# Exctract dictionary from dataframe of nodes
nodes_elab_file = pd.read_csv("./nodes_elab.csv", sep=";")
nodes_elab_filtered = nodes_elab_file[["new_id","names"]]

dictionary_index = {}

for index, row in nodes_elab_filtered.iterrows():
    dictionary_index[ row['names'] ] = row['new_id']


# Create new column id into edges_elab.csv
edges_elab_file = pd.read_csv("./edges_elab.csv", sep=";")
edges_elab_filtered = edges_elab_file[["src:START_ID","dst:END_ID"]]

list_src_id = []
list_dst_id = []

for index, row in edges_elab_filtered.iterrows():
    # Create id for src:START_ID
    if row['src:START_ID'] in dictionary_index:
        list_src_id.append(dictionary_index[ row['src:START_ID'] ])
    else:
        id_ = len(dictionary_index)+1
        dictionary_index[ row['src:START_ID'] ] = id_
        list_src_id.append(dictionary_index[ row['src:START_ID'] ])

    # Create id for dst:END_ID
    if row['dst:END_ID'] in dictionary_index:
        list_dst_id.append(dictionary_index[ row['dst:END_ID'] ])
    else:
        id_ = len(dictionary_index)+1
        dictionary_index[ row['dst:END_ID'] ] = id_
        list_dst_id.append(dictionary_index[ row['dst:END_ID'] ])


edges_elab_file["src:START_ID_int"] = list_src_id
edges_elab_file["dst:END_ID_int"] = list_dst_id

print(edges_elab_file.head())

edges_elab_file.to_csv("./new/edges_elab.csv", sep=";")

return 0
```

```
if __name__ == "__main__":
    main()
```

6. RISULTATI

I risultati ottenuti confermano l'ipotesi di poter utilizzare Cypher, Morpheus in Spark 3.0.3. Durante la fase di test quasi tutte le query realizzate ottengono dei risultati. Solamente in due riportano delle implementazioni non ancora supportate da Spark. Inizialmente i test erano stati svolti su Spark 3.1.2 ottenendo errori di incompatibilità. Si è proceduti ad un downgrade di Spark alla minima versione, >= 3.0, reperibile dal sito ufficiale: 3.0.3.

Lo script implementato è stato testato sotto diverse tipologie di query, variando in grandezza e tipologia, ottenendo una corretta traduzione delle query per Spark (Video dimostrativo al capitolo 7).

Tramite l'utilizzo di GraphX sulla rete presa in esame è stato possibile calcolare il degree dei nodi, riportando il gene con il degree più alto; esso corrisponde all' "hub" della rete. Dai risultati si evince che il nodo 25 è l'hub con il degree più alto pari a 57.

7. SCRIPT QUERY_TRANSLATE

- Video utilizzo script: https://drive.google.com/file/d/1MIbktOPqobpocxUwrDVYPggoaAc6L_nN/view?usp=sharing

- Download: https://drive.google.com/file/d/18zMhjY11xCunt_7bGa_oL7LOHS-VZHtk/view?usp=sharing

Dall'analisi presa in esame è emerso un dettaglio interessante, la sintassi delle query scritte per Neo4j differisce da quella utilizzata in Spark. Vi sono presenti operazioni ancora non supportate, problemi di incompatibilità della sintassi utilizzata, portando quindi alla conversione della query nel formato supportato da Cypher.
Tale script è stato scritto in Python. Una volta lanciato lo script, con i comandi da terminale "python query_translate.py", verrà richiesta una stringa d'input. Tale stringa è la query da convertire in formato Cypher. Nello specifico traduce le molteplicità introdotte dalle query che presentano tale sotto-stringa **1..X]*; quindi lo script si aspetta in input una stringa in cui sia presente la sotto-stringa mostrata. Successivamente, una volta confermata la correttezza della stringa, lo script genererà la stessa query in formato Cypher-Spark, fornendo in output entrambe le versioni della query.

8. SITOGRAFIA

1. Mats Rydberg - Max Kießling, Graph Features in Spark 3.0: Integrating Graph Q.. , https://databricks.com/session_eu19/graph-features-in-spark-3-0-integrating-graph-querying-and-algorithms-in-spark-graph-continues, 10/07/21.
2. Databricks, Graph Features in Spark 3.0: Integrating Graph Q.. slide, https://www.slideshare.net/databricks/graph-features-in-spark-30-integrating-graph-querying-and-algorithms-in-spark-graph, 12/07/21.
3. GraphX in Spark >= 3.0, https://spark.apache.org/docs/latest/graphx-programming-guide.html, 07/08/21.

9. CONCLUSIONI

Tale progetto ha permesso l'acquisizione di diverse competenze pratiche, analizzate durante il corso di Big-Data. Si è preso consapevolezza dell'utilizzo e della logica di programmazione Spark. E' stata utilizzata una nuova modalità di query non SQL, ovvero query Cypher, utilizzate per effettuare query su grafi e non su database. Sono state analizzate e utilizzate diverse estensioni di Cypher, tra cui Morpheus, utilizzato per effettuare le query prese sotto esame nella fase di testing. Infine è stato utilizzato anche la libreria GraphX permettendo di effettuare dei calcoli su grafi, da quelle più comuni, come il calcolo del degree, a quelle più complesse.