Team Neo4j

# Members

**Rohit Zawar** (rzawar@indiana.edu)

**Chintan Gosalia** (cgosalia@indiana.edu)

**Aishwarya Iyer** (aishiyer@indiana.edu)

**Abdul Mudabir Patel** (abdpatel@indiana.edu)

# Github repository

https://github.iu.edu/cgosalia/z517\_advn\_track\_Neo4J.git

# Project Objective

To create a graph database from the given twitter dataset - obama\_20121015\_20121115.txt so that the database can be queried in order to dynamically visualize a tweet-based diffusion over some specific time intervals.

# High level design

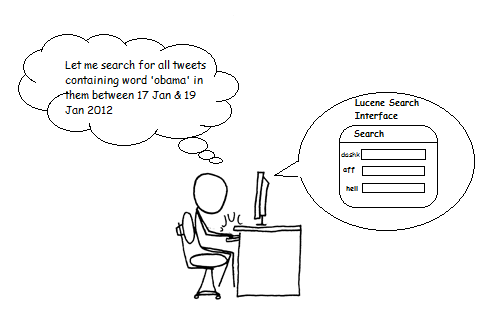
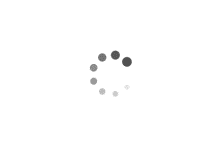
A Quick Operational View

The Neo4j team worked in conjunction with teams – Lucene and D3.js. This section unveils the functional view of the entire project that the 3 teams collaborated for.

*(a detailed description can be found under the ‘Low Level design’ section)*

* Team Lucene initiates an interface where a user may choose to track a set of tweets matching his/her search criteria. The fields on the basis of which the tweets are to be identified comprise of a tweet ***Location***, tweeter’s ***Username***, ***Hashtag***s contained, ***Tweet*** content, ***Date*** *range* of the tweets.
* In the backend, a list of the located tweets’ IDs is passed over to our Neo4j team in order that those tweets may be located in the Neo4j graph database. All form of data in the graph database is represented in the form of **nodes** and **edges**. Node relationships are indicated in the graph by arrows pointing from one node to the other. Thus, the list of tweet IDs fetched from Lucene are in fact located as nodes in the graph database. These nodes are henceforth referred to as **seed tweets** or **seed nodes**.
* The seed nodes are to be diffused over time. Consider a use case here in layman terms – if the seed nodes correspond to all the tweets in which the name *‘obama’* is mentioned between the dates 17 January, 0000 hours and 19 January, 2400 hours (let us note this as ‘*base time’*); then a user can further see how many users retweeted those seed tweets over say, the next 1 hour, 3 hours or maybe even a day. This act of the seed tweets branching out to indicate the number of times they were retweeted in the chosen time interval is called **tweet-based time diffusion**.
* Let’s take a look at how the diffusion works. Team D3.js provides an interface where a time slot may be chosen (e.g. 1 hour, 3 hours after the base time). This time slot is passed by D3.js’s interface as an argument to a *Search* function that queries the graph database with the previously tracked seed node information in the backend. This is the point of integration between the Neo4j team and the Lucene and D3.js teams.
* Our Neo4j team’s Search function passes the relevant seed nodes’ time diffusion information back to the D3.js team in order for the D3.js’s interface to visualize the information in a fancy manner.
* D3.js presents two kinds of visualization to the user:
  + a *location-based* seed nodes’ diffusion over the states in the US from where the tweets originate over time.
  + an elaborate tweet-based time diffusion network of connected tweets (nodes).

system architecture diagram

****

D3.js interface

Generates a list   
of tweet IDs and passes them to   
the Neo4j team in order for them to be identified as **seed nodes**

****

Time diffused seed nodes information in the visualization area is displayed back to the user

**Neo4j graph database**

Visualization Area

3 hours later

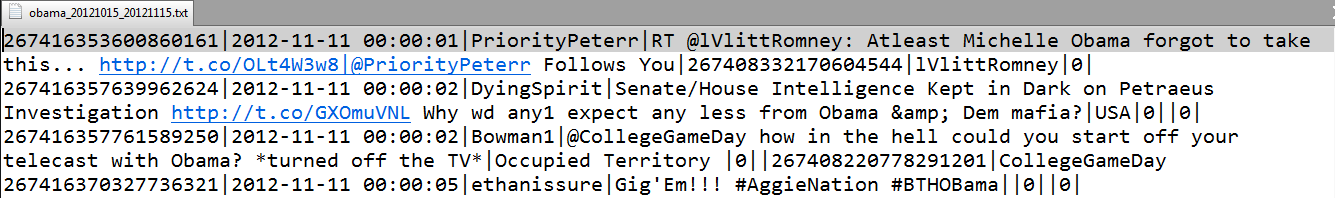
1 hour later

Time

d3.js’s interface passes time slot information to graph DB

neo4j graph Database design

The teams were provided with a twitter dataset text file - obama\_20121015\_20121115.txt (Size: 974 MB). The dataset comprises of all tweets generated during President Obama’s re-election. Here’s a snapshot of a few lines of the twitter dataset:



A naked eye view of the dataset brought out the following inferences:

* Each line in the file corresponds to information related to a tweet posted by some twitter user.
* Each tweet further contains fields separated by ‘|’
* There are either 8 fields or 9 fields. **8** in case of normal **Tweet**s and **Retweet**s (RTs) and **9** fields when the tweet is a **Reply** to another tweet.
* The fields in the dataset can be classified as below:

tweet-id | Unix-timestamp | tweeter-username | tweet | geolocation | original-tweetID-retweeted | original-tweeter-username | tweetID-being-replied-to | username-being-replied-to

* Some trivia about the dataset:
  + In the case that a tweet line in the dataset contains **8 fields** i.e. for normal Tweets and Retweets, the last field contains '0'.
  + Whenever there are 9 fields in a tweet line in the dataset, it should be inferred that the tweet is a *Reply* to some other tweet. The last column in such cases does not contain '0' but the *username* of the user whose tweet is being replied to.

Knowing the above characteristics of the twitter dataset at hand, the dataset was then programmatically parsed optimally to create a graph database. Creating the graph database by parsing the dataset essentially means creating nodes that are connected to each other with edges that have arrows pointing either outward from a node or inward. Each node signifies specific information related to a tweet whereas each edge signifies a twitter relationship type (e.g. *IS\_A\_RETWEETOF*, *IS\_A\_REPLYTOTWEET*, etc.) between two nodes.

(More details about this in the Low Level Design section ahead)

# Low level design

This section emphasizes on details not elaborated in the *High level design* section.

Components of system

To build a working system, Team Neo4j wrote Java code with Neo4j’s default set of class libraries along with a few other supplementary binaries like the **Jersey** and **JSON-simple** libraries. Neo4j provides an extensive API that can be readily embedded in Java code. Let’s look at each program class that is a part of the Java program:

Dataset

**class Dataset**

# retweet\_original\_message\_id : long

# tweet\_id : long

# username : String

# tweet : String

# Location : String

# retweet\_username : String

# reply\_username : String

# hashtags\_list : ArrayList<StringBuffer>

# username\_list : ArrayList<StringBuffer>

# unix\_time : long

# isRetweet : boolean

This class is used to embody the basic

characteristics of a tweet.

These characteristics are retrieved by

parsing each line of the dataset provided.

# callThread : boolean

# service : GraphDatabaseService

- path : String

# tx : Transaction

# createdNodeMap : Map<String, Long>

# createdTweetNodeMap : Map<String, Long>

# dataset : ArrayList<Dataset>

# datasetTweetCount : int

# numberofBatches : int

# batchsize : int

RegexDB

**class RegexDB extends Thread**

It is clear that each line in the dataset corresponds

to a tweet and that the fields in the tweet line

bear all information that vitally describes

any tweet.

One of the key functions of the **RegexDB** class

is to parse the dataset. As a result of parsing the

dataset by iteratively finding regular expressions

occurring in it, each node in the graph database is

- <<enumeration>> RelationType

- configureDatabase()

+ AnalyzeDataset(dataSetPath : String)

+ parseAndCreateDatabase()

+ run()

+ createTweetNode\_New(messageID : long, message : String, timestamp : long, location : String) : Node

+ checkNode(value : Long) : Node

+ createUserNode(username : String) : Node

+ createHashTag(hashTag : String) : Node

+ createRelationShip(node1 : Node, node2 : Node, relationshipname : String)

+ connectTweets(originalNode : Node, Tweet : Node, string : String)

- registerShutdownHook(graphDb : GraphDatabaseService)

constructed. The second key function of the RegexDB

class is to create every node needed for every piece of

information related to a node (e.g. tweet node,

username node, hash tag node, etc) and to

create every required relationship between

nodes (e.g. a ‘*MENTIONS*’ relationship between

a *tweet* node and a *user* node).

We describe each method of the RegexDB class in the following section:

* RelationType
  + This is an *enum* which defines all the various relationship types that have been designated as possible for connecting two nodes. The relationship types are:

*TWEETS, RETWEETS, CONTAINS, IS\_A\_RETWEETOF, MENTIONS, IS\_A\_REPLYTOTWEET, REPLIES*

* configureDatabase()
  + This method sets stage for our graph database creation. It creates an instance of a new graph database.
  + It implements *Label Indexing* on *User*, *Tweet* and *Hash tag* nodes.
  + It registers a shutdown hook for the Neo4j graph database to avoid abrupt database shutdowns.
* AnalyzeDataset(dataSetPath : String)
  + This method reports the status of the amount of data created and indexed in the graph database.
  + The tweet lines are being parsed in batches and each individual batch is being processed to create respective nodes and relationships. Thus, details such as which batch of dataset lines is being processed form a part of the status report.
  + The method accepts the location of the twitter dataset file as its argument.
* parseAndCreateDatabase()
  + This method is the highlight of the RegexDB class. This is where the dataset is being read line by line to examine it based on regular expressions occurring in it.
  + Some ‘regular expressions’ examples
    - A user’s *username* is being identified in the tweet’s message field if a *String* has a *‘@’* preceding it. Likewise, a *hash tag* if a *String* is preceded by a *‘#’*
    - A tweet is being classified as a *Retweet* if its *message* field contains ‘*RT @’* in it.
    - A tweet is being classified as a *Reply* to some tweet if that tweet line’s 9th field contains a value other than ‘0’.
* run()
  + This method implements the multi-threaded nature of nodes and relationships creation.
  + It instantiates a thread for every line of the dataset in order to analyze the tweets’ related information contained in each line and subsequently, creating nodes like *User*, *Tweet, HashTag* andrelationships between the nodes like *CONTAINS, IS\_A\_RETWEETOF, MENTIONS*, etc.
* createTweetNode\_New(messageID : long, message : String, timestamp : long, location : String) : Node
  + This method creates a node of the tweet type by setting all its respective properties like its *messageID, TimeStamp, Location, Message,* etc.
  + It accepts arguments that are parsed from the dataset - *messageID, message, timestamp, location.*
  + It returns the tweet node that is created.
* checkNode(value : Long) : Node
  + This method checks whether a node that can be uniquely mapped with *value* (method argument), is already present in our graph database.
  + If a node with such a *value* is present, that same node is returned by the method, else ‘null’ is returned.
* createUserNode(username : String) : Node
  + This method creates a node that can be identified as one that corresponds to a *User* in the twitter dataset.
  + It accepts a *username* tag to the *User.*
  + The *User* node is returned post its creation.
* createHashTag(hashTag : String) : Node
  + This method is just like the createUserNode() method except this one creates a node for a ‘hash tag’ encountered in the tweet message.
* createRelationShip(node1 : Node, node2 : Node, relationshipname : String)
  + This method appropriately assigns the correct relationship type (*TWEETS*, *MENTIONS, etc.*) to two connecting nodes.
  + It accepts 2 nodes between which a relationship is to be made and a relationship property *relationshipname* to decipher the relationship type to be created between the nodes.
* connectTweets(originalNode : Node, Tweet : Node, string : String)
  + This method is exclusively designed only to handle the Retweet and Reply relationship creation. It calls the *createRelationShip* method in turn to create the relationships in both cases.
  + It accepts 2 nodes and a string just like the *createRelationShip* method.
* registerShutdownHook(graphDb : GraphDatabaseService)
  + This method creates the shutdown hook for the Neo4j graph database to avoid abrupt database shutdowns.
  + It accepts an instance of the GraphDatabaseService object.

SearchQuery

**public class SearchQuery**

- tweetIdList : List<Long>

The function of this class is primarily to query a **REST**

+ <<constructor>> SearchQuery()

+ <<constructor>> SearchQuery(tweetId : List<Long>)

+ query\_new(originalQuery : String) : Map<JSONObject,JSONArray>

+ getJsonFromMessageList(relType : String, startTime : long, endTime : long) : JSONArray

+ writeJsonToFile(jObj: JSONArray)

service using **Cypher** as the querying language. The

server’s response needs to be interpreted and written

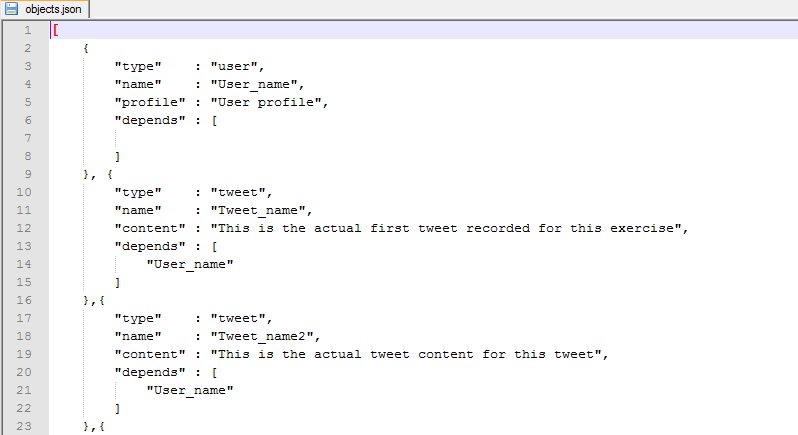
in JSON format. This JSON format is then parsed by the

d3.js team to finally visualize it for the user.

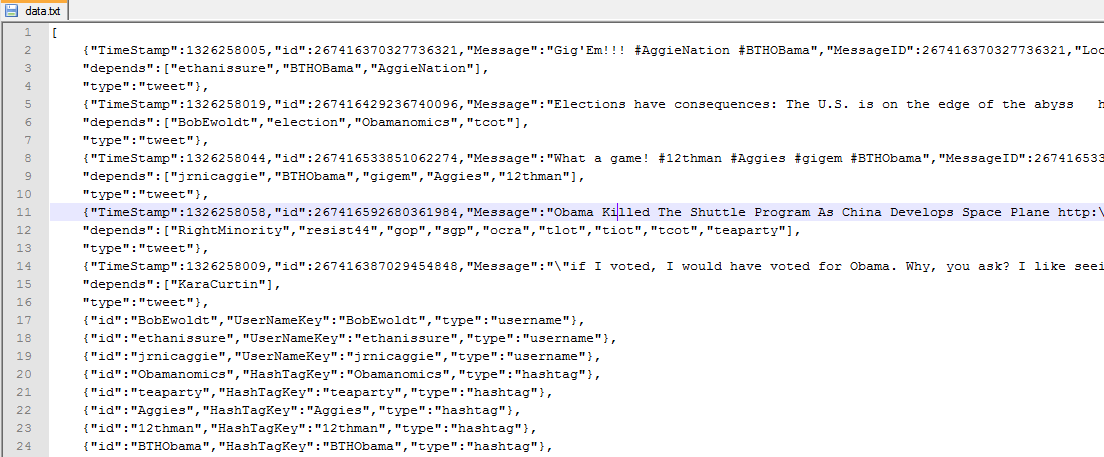
We describe each method of the SearchQuery class

in the following section:

* SearchQuery()
  + This is the class’s default empty constructor method.
* SearchQuery(tweetId : List<Long>)
  + This is a parameterized constructor of the SearchQuery class.
  + It instantiates and defines the class’s member variable in order to store all the seed tweet IDs passed over from the Lucene team.
* query\_new(originalQuery : String) : Map<JSONObject,JSONArray>
  + This method sets up the **REST client-server environment** and the REST service is then queried using **cypher** queries.
  + It accepts the argument *originalQuery* which is written to query for information about all the dependent nodes connected to a single seed tweet over a specific time slot.
  + The method returns a Map data structure. The Map stores a tweet node as a JSONObject which is mapped to a JSONArray. The JSONArray contains a set of JSONObjects corresponding to the time dependent nodes of the seed tweet.
* getJsonFromMessageList(messgIDList : List<Long>, relType : String, startTime : long, endTime : long) : JSONArray
  + This method accepts the Map<JSONObject,JSONArray> returned by the query() method for each seed tweet and then further adds each one of the seed tweets’ dependent nodes to the final JSONArray being constructed to pass over to the d3.js team.
  + We make sure that none of the dependent nodes of the seed tweets are ever repeated in the final JSONArray. This is a crucial requirement for the parsing of the JSON file for d3.js team. The format of the final JSON file for the d3.js team is as follows:



* + The method returns the final JSONArray that is needed to be parsed by the d3.js team. As per the JSON format, our team’s JSON format construction was as follows:



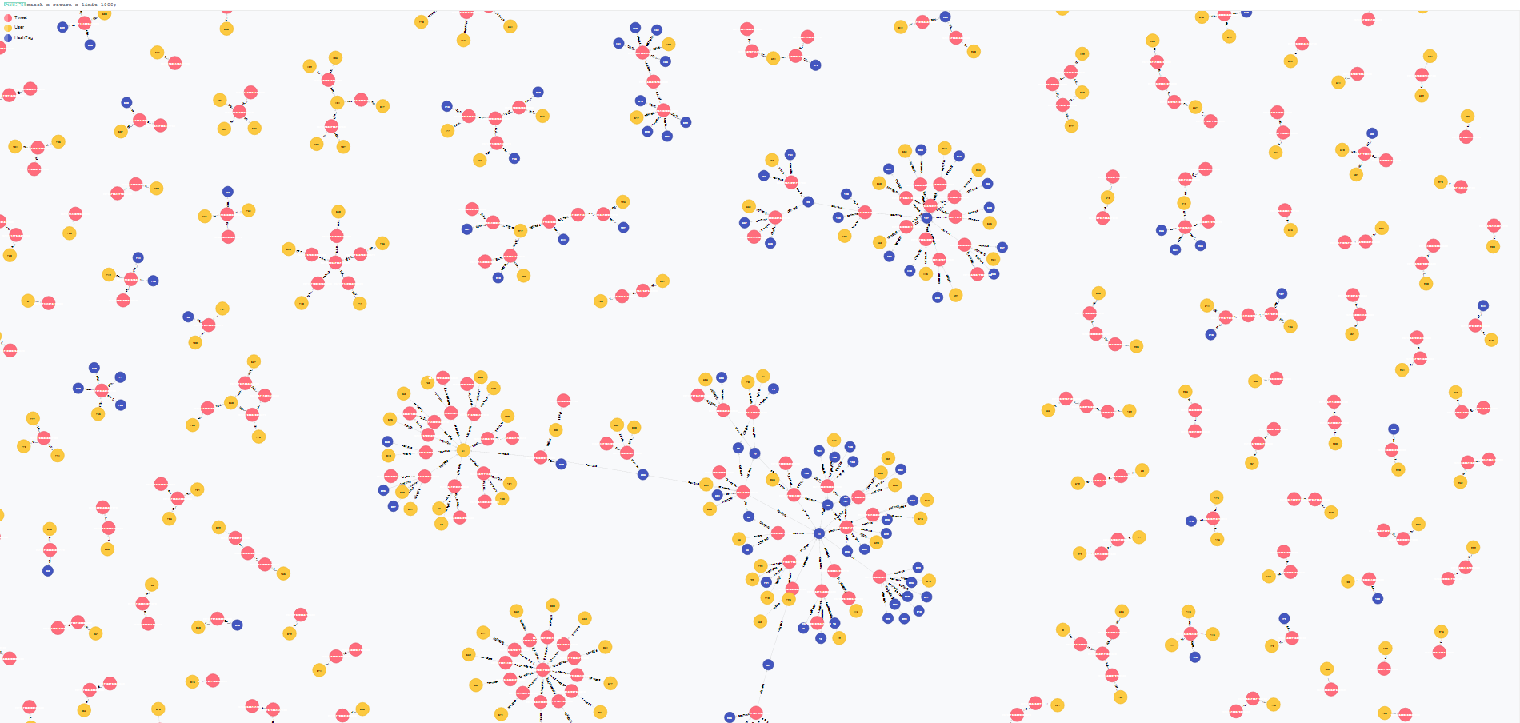
* writeJsonToFile(JSONArray jObj)
  + This method writes the final JSONArray returned by the getJsonFromMessageList() to a text file called *‘data.txt’.*

use case cypher queries

The graph database was queried using **Cypher** to get sample responses that needed to be verified for correctness of the information needed for constructing the final JSON output. Here are a few sample queries that were useful:

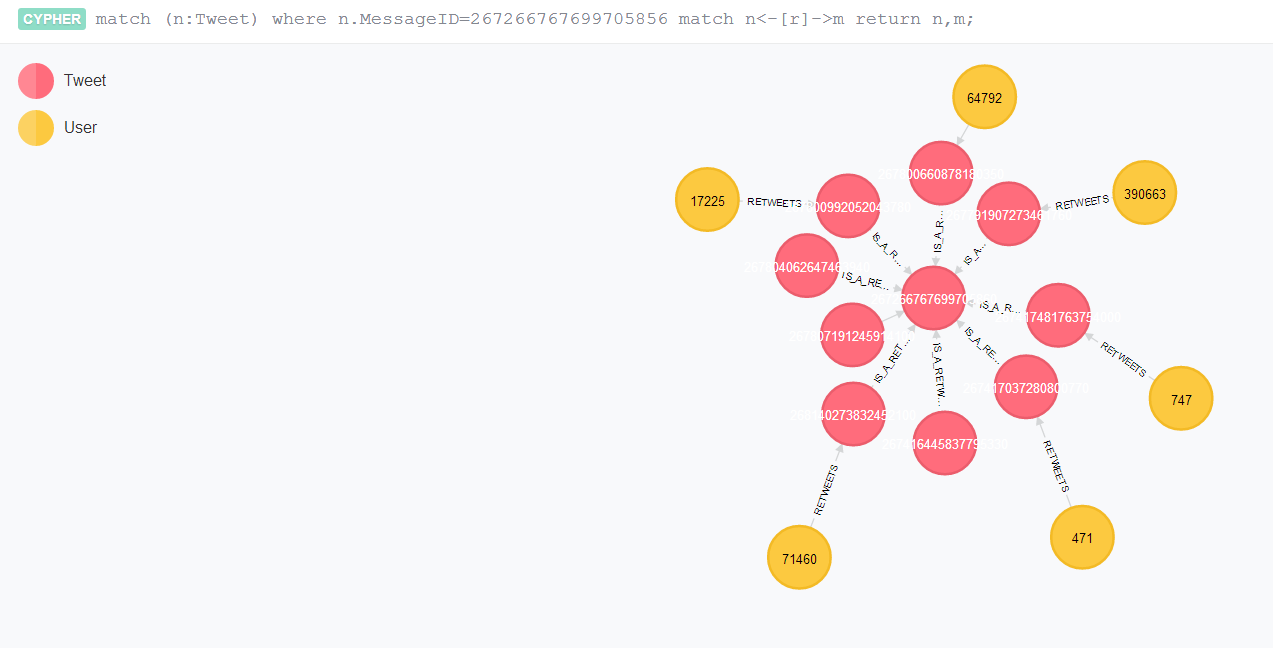
* To display the first 1000 nodes in the graph database

*Cypher Query* : **MATCH x RETURN x LIMIT 1000;**

 *Neo4j web browser visualization*:

* To display a single node and all its dependent nodes

*Cypher Query* : **MATCH (n:Tweet) WHERE n.MessageID=<sample Message ID> MATCH (n)<-[r]->m RETURN n,m;**

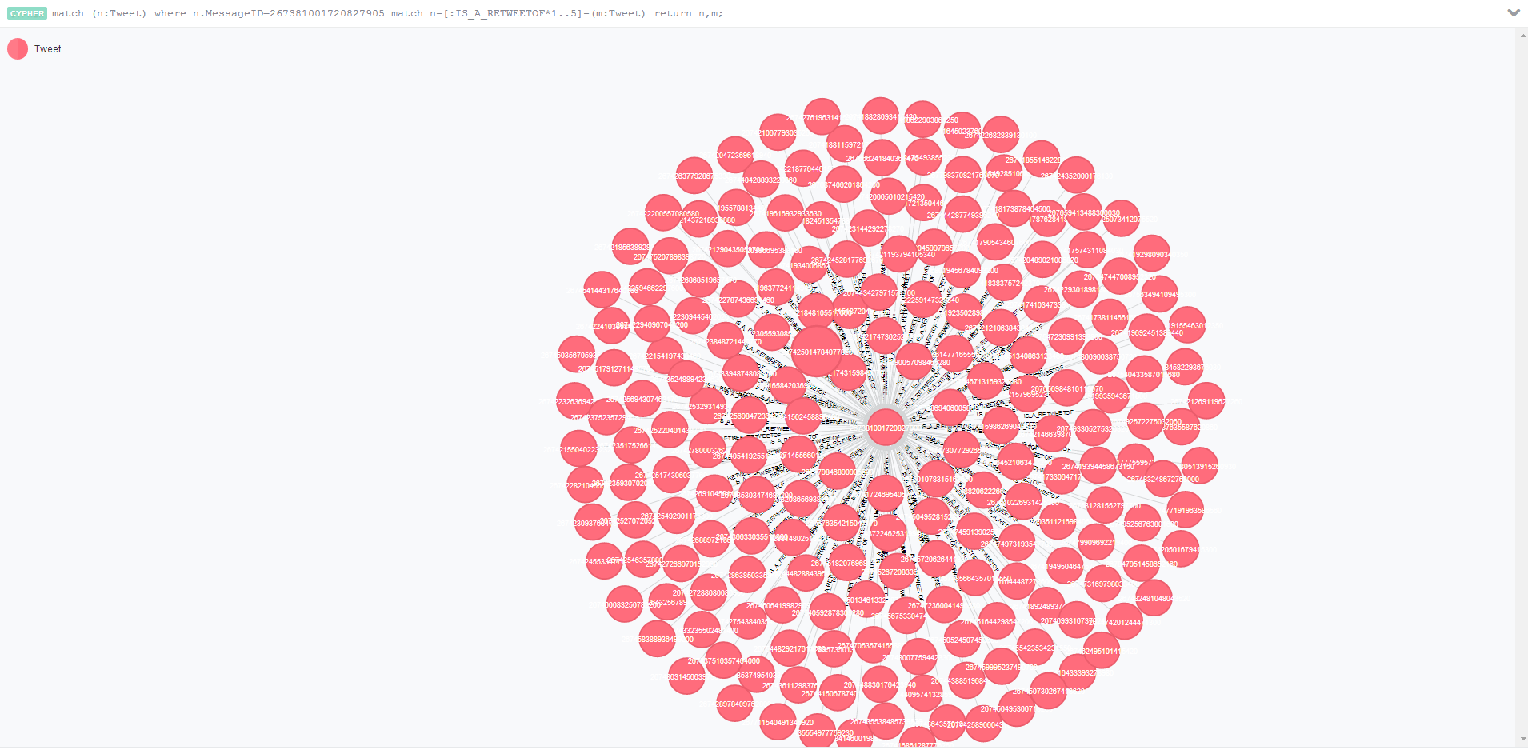
 *Neo4j web browser visualization*:

# 

* To display a sample node chosen from the graph database which can be diffused over a specific time interval

*Cypher Query* : **MATCH (n:Tweet) WHERE n.MessageID=<sample Message ID> MATCH (n)<-[:<relationship type>\*1..5]-(x:Tweet) RETURN n,x;**

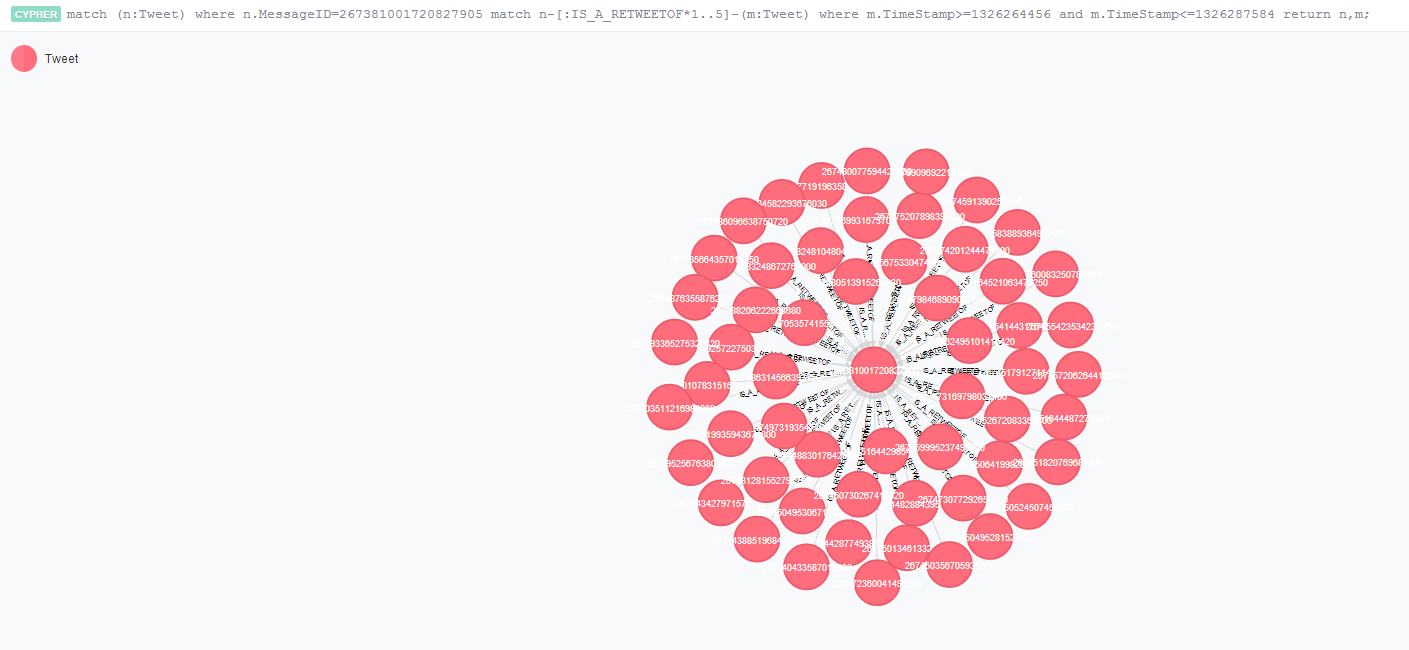
*Neo4j web browser visualization*:



* Diffusing the sample seed node by filtering in only those dependent nodes that fall within the specified time slot

*Cypher Query* : **MATCH (n:Tweet) WHERE n.MessageID=<sample Message ID> MATCH (n)<-[:<relationship type>\*1..5]-(x:Tweet) WHERE x.TimeStamp>=<startTime> AND x.TimeStamp<=<endTime> RETURN n,x;**

*Neo4j web browser visualization*:



# Auxiliary information

challenges accomplished

* Parsed the twitter dataset optimally using relatively faster functions in the Neo4j API.
* Used multi-threading for efficient system memory management while creating the graph database.
* Used the REST service to fire Cypher queries to it.
* Interpreted the REST service’s crude query response into a parseable JSON format for the d3.js team.

difficulties encountered

* We could not complete the final integration with the d3.js team because the *Jersey client jar* used in our code was not compatible with *Apache Tomcat server*.
* The graph database we created could not generate multi-level node relationships because of incomplete or inconsistent data present in the twitter dataset.

Things we learned from the project

* Parsing a huge dataset with regular expressions efficiently – use of appropriate data-structures (*Hashmap*s, *StringBuffer*s, *List*s, etc.), use of multi-threading to optimally manage memory.
* Neo4j graph database concepts and the Cypher query language.
* Various applications of graph database with the evolution and rise of complex data to be analyzed on the internet. Understanding that an offshoot of the project may be to visualize twitter dataset for *sentiment analysis* and *popular trends*.
* Embedding Neo4j functionality in Java.
* Use of REST service for Neo4j cypher query responses in Java code.
* Interpreting a complex REST service’s response into an understandable and useful data format like JSON.
* Version controlling system – *Github*.
* Working in a team harmoniously towards a large scale project.