# Graph Data Modeling for Political Communication on Twitter

Greeshma Reddy Padiri (greeshma@iastate.edu)
Shruti Biswal (sbiswal@iastate.edu)
Boudhayan Banerjee (bbanerji@istate.edu)

***Abstract*— It is well known that graph databases have been constantly gaining popularity because of their reliability and high scalability, especially when there is huge amount of highly interconnected data to handle. Performance of the graph database depends on the underlying graph model used. Query processing time is one of the important metrics for the performance evaluation of any graph model. It is of vital importance to fine tune the underlying graph model so that the average query response time is minimal. Titan, being one of the best graph databases providing low-latency, for complex graph traversals is used in the project. Our project focuses on importing the Twitter dataset into Titan from Neo4j and creating two different graph models that use indexing and reducing hop methods on the given reference data model and testing them by executing a set of queries, comparing their average response times and also increase in the databases size. Reduced hop model showed better performance than the reference model but the indexed model proved to be the best one with minimal response time and considerable increase in the database size. The report presents the implementation details for importing data into Titan from Neo4j, description of the other two data models and rationale behind creating them, our experiments with results and conclusion.***

***Keywords—Graph databases; neo4j; titan; query optimization; indexing***

## I. INTRODUCTION

Graph databases are gaining traction in recent years, as compared to the traditional Relational Database Management Systems that are being used for several years. The reason behind this rapid shift to graph database can be attributed to the fact that most of the data that we use today has high degree of connectedness. The relational databases have been the powerhouse for storing and manipulating data for decades. In traditional relational databases data are kept in a well-structured manner in the form of tables. Graph databases differ in that they use graph structures to represent the data for executing queries over the data. According to Neo4j documentation "relationships" are the first-class citizens in graph database. The three main components of graph databases used for representing and storing the data as nodes, edges and properties. The edges or relationships are the most important component of the graph database in a sense that they allow the data in the storage to be linked with each other. This relationship, implemented through edges between the nodes, helps in faster retrieval of queries in many cases. Graph databases are well suited to the cases where the dataset has deep links. One such application of graph databases can be found in social networking systems, where the "friends" relationship is essentially unbounded [1]. And opposed to relational databases, graph databases can easily execute recursive queries and traversal queries that include a full traversal or few hops like *friend of friend of friend*. Other places where graph databases would naturally perform better can be the Online Search Systems and also BigData environments. Keeping in view the popularity of graph databases, several graph databases and different graph query languages have been introduced in the market in recent past, for example, Neo4j- Cypher, Titan-Gremlin, Graphbase, GraphDB, Oracle Spatial and Graph, OrientDB, ArangoDB and so on. Graph query languages include Cypher, Gremlin, GraphQL, SPARQL and so on.

In this project, a graph database comprising of political communications tweets between political figures and reporters pertaining to the US election 2016, in Neo4j is provided as initial dataset. The goal is to come up with a data model that reduces the query response time in the graph database. This can be accomplished in many ways. Here, we focus on two of them. First, we create indexes on the key properties. Second, we add new edges between the nodes that are often queried together to reduce the number of hops during graph traversal. Importing data into Titan DB can sometime be challenging. Titan provides lots of configuration options and tools that make importing graphs with billions of nodes into Titan easily and efficiently. This kind of large amount of ingestion is called bulk loading. This is different from transactional loading where only small amount of data are added through smaller single transactions. [5] One of the popular ETL tool that extracts data from Neo4j and loads in Titan is gremlin-importer. Gremlin- importer can be time-consuming and inconsistent, therefore we used Groovy and Gremlin itself to parse the generated Comma Separated Value (CSV) formatted file into JavaScript Object Notation (JSON) format.

Initiation of the project and experiment was with a Neo4j Database containing Model I. Figure 1 represents Model I available in Neo4j. We extract the information about nodes and relationships as separate CSV files from Neo4j database. The gathered CSV files are parsed using Groovy and Gremlin to create a JSON file which is readable by Titan DB. Loading the data into Titan is fairly simple process. The created JSON files should to be placed in the data directory of Titan server. Gremlin's GraphSON is used to read the graph structure from the JSON file placed in data directory. In the reference data model that we have, there are nodes with labels User, Tweet, State, Url, Hashtag, Day, Month, Year, and Century. Properties in a graph database model have the pertinent information related to the nodes.

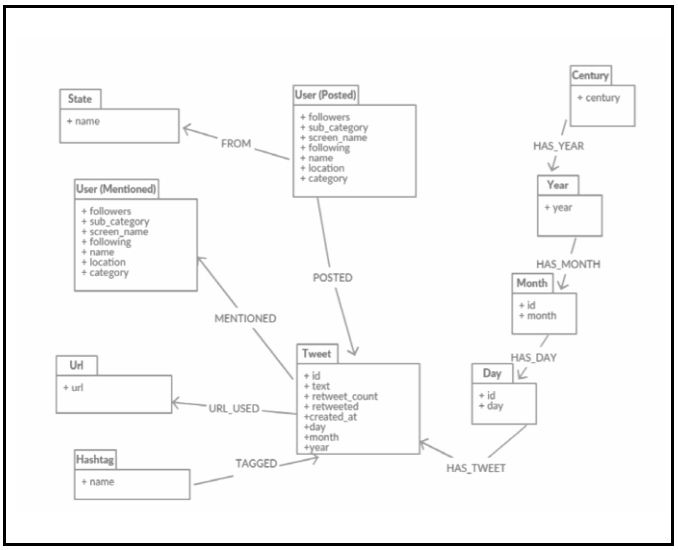


**Figure 1: Reference Data Model**

In our experiment, we have observed that Model II has outperformed the other two models in terms of average query response time. The main idea behind Model II was to build index on primary properties and other attributes that are frequently referred for filtering. In this model, we applied two types of indexes to address different optimization required in the query, namely mixed index, and composite index. We will discuss in detail how the indexes are applied in our model in later sections. In case of Model III, we took a different approach. If there are two different types of nodes with are queried together multiple times and if the two nodes do not have a direct edge between them, then adding edges between them will decrease average query response time.

## II. RELATED WORK

Social media has been most important and popular form of communication from past decade. We use social media like Twitter for day to day communication with global audience. It is also a great platform for various companies for product promotions. In this paper, we have focused on the political communication through Twitter during 2016 presidential election campaign by presidential candidates, state's senators, state's house representatives, journalists, and state's senates. Our project starts with understanding the importance of graph databases and corresponding graph query languages. [3] gives a performance comparison of Cypher, Gremlin, and native access in Neo4j and information presents test results using different performance metrics like average query response times. It also provides a valuable scope to understand how the efficiency of a data model can play key role in productive results. And, more information about the test setup and performance metrics. The paper also compared and evaluated the performance of different graph query languages like Cypher, Gremlin and Native access using Neo4j when used for OpenSocial, a social web application that contains the interconnected data from Facebook and Twitter. The paper provides introduction to graph query languages. [1] aims at devising a technique that can allow fast search of queries in large graphs without being aware of the actual structure of the underlying graph. This method involves use of graph matching (not subgraph isomorphism) in terms of finding the query graph embedding in the target graph by analyzing the neighboring node labels and converting them to vectors, which in turn provides indexing to be used to search efficiently. Since the actual large networks are prone to errors and outliers, this technique provides a robust and quick method of querying the graphs. This technique helped us in understanding underlying search technique. The query time can be significantly improved without requiring to make much changes to the design of the database that would be otherwise made to fit the probable queries. Also, the method allows the index structure to be updated more efficiently rather than requiring re-indexing of the graph when changes occur in the graph. In short, this paper aims to modify the search technique used rather than designing the data which uses the default search technique. Due to the large size of the data and restructuring data model being a tedious and time consuming process, this method of designing the search technique can improve performance of database and can perform better with a nicely designed model.

[2] describes how to convert a relational database to graph database. This technique can create a graph database from a given relational database. We have more relational database than graph databases because graph database is newer concept. Relational databases have matured a lot over time and being used everywhere. But in some cases, especially in the context of social media, graph databases make more sense and it is easier to construct. [2] describes building of a system called GraphGEN which can extract variety of different graphs from a relational database. As researchers are trying to find out the underlying graph relation from the underlying relational database GraphGEN can be an excellent technique to convert the RDBMS to GDBMS fairly easily. The paper describes interconnected structure among data using graph algorithms and graph analytics. GraphGEN can also allow user to specify graph extraction query and to interactively explore the generated graphs. GraphGEN supports an expressive domain specific language based on Datalog, to specify graphs to be extracted from the relational data. GraphGen uses a translation layer to generate SQL queries which will be used in the database. We decided to test the performance of the created models by running 25 different queries 30 times, ignoring first 5 runs. The rationale behind running each query 30 times is to nullify any kind of interference that may occur during query run time. And average value of the 30 runs is considered. [6] outlines how to read data from a CSV file using Gremlin. [7] describes the process to create mixed and composite indexes on the database.

## III. PROPOSED WORK

In this section, we describe in detail the rationale and design of each model created, the implementation of each model and the advantages of each design along with how the reference data

model is loaded into Titan. The same process is used to load the other two models also. Reference data model can be seen in Figure 1.

### A. *Indexed Data Model*

Creation of indexes on the reference data model improves the speed of retrieval of data retrieval operations and generates better response times for queries. Titan supports two kinds of graph indexing which are used over the entire graph for efficient traversals or retrieval of vertices and edges namely, Composite index and Mixed index. While composite indexes are faster and suitable for setting unique constraints & equality comparisons only, Mixed indexes are more flexible and useful for non-equality comparisons, ordering operations, range operations. The proposed data model with indexing contains both mixed and composite indexes over various node properties to allow improved performance of the queries given in [4].

Each node has unique property that distinguishes it from other nodes such as *name* property of *State* node, *name* property of *Hashtag* node, *screen_name* of *User* node, id property of *Tweet* node, *url* property of *Url* node. Composite indexes on these unique properties are created. From [4], we have observed that most of the queries have an equality condition check on the *sub_category* property of *User* nodes. Though there is one query among twenty five queries that has a range check on *month* and *sub_category* properties, majority of the queries have an equality check. So, we chose to create composite index itself on *sub_category* and also *month*. Mixed indexes are created on the properties which are constantly checked for non-equality comparisons. These properties include *screen_name*, *sub_category* and *category* of the *User* node and *name* of the *State* node and the *Hashtag* node. Creation of mixed index on the *day* property of *Tweet* node has improved the response times of queries given in [4] which involve ordering based on the day of a month. Table. 1 lists the indexes created on different properties, the nodes which they belong to, along with type of the index created.

And Figure 2 shows the data model on which indexes are created.

| Property | Node | Index type |
|---|---|---|
| screen_name, sub_category | User | Composite, Mixed |
| id, month | Tweet | Composite |
| name | State, Hashtag | Composite, Mixed |
| url | Url | Composite |
| day | Tweet | Mixed |
| category | User | Mixed |

**Table 1: Index Types on Node Properties**

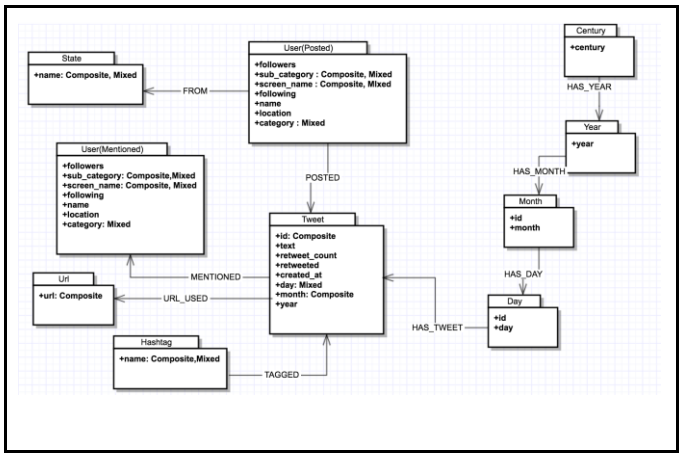


**Figure 2: Indexed Data Model**

### B. *Reduced Hop Data Model*

Average response time of certain queries can be significantly improved when the number of edges visited along the path of traversal are decreased. This reduction can be ensured by introducing direct edges between few nodes which are otherwise indirectly connected. While many queries require at least two hops, reducing the hop count to 1 can cause the graph size to increase by approximately 10 million more edges. Among the queries given in [4], the longest path traversed is that from node *Hashtag* to *State*, which involves three hops. The proposed model aims at reducing this hop count to 1 by introducing new edges from *Hashtag* node to *State* node. This new edges between Hashtag and State always represents- an "User" from a "State" posts a "Tweet" containing the "Hashtag". The edge label of the newly created edge is given as *TAGFROM* . It has edge properties - *date* of *Tweet* and the *sub_category* of *User*. This simple modification with addition of edges improved the response time of the queries involving more hops. Figure 3 shows the schema for the reduced hop model. And performance of this model can be induced from Figure 7 & Figure 8.

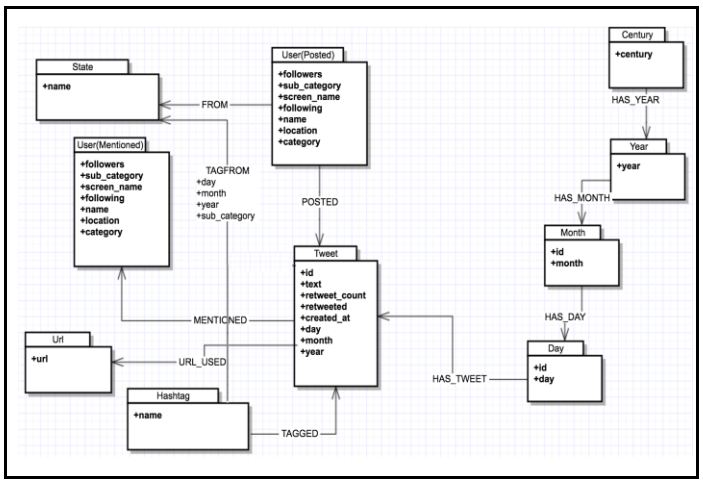


**Figure 3: Reduced Hop Data Model**

## IV. EXPERIMENTS

In this section, we present the experimental procedure and results to demonstrate the improvement on query response time over reference data model, indexed data model and reduced hop data model. All experiments are performed using- Titan 1.0.0 version with Berkeley DB as storage backend, ElasticSearch as index backend and 3.0.1-incubating Gremlin version on single core 8GB, 2.7 GHz processor.

### *A. Graph Datasets*

**Twitter Graph:** Twitter Graph is a large dataset depicting the political activity on Twitter which was collected in [4] during January-May, 2016 before the Presidential Election. The dataset contains 137074 nodes and 446701 edges. The nodes represent various entities/ objects of the graph dataset such as *User*, *Tweet*, *Hashtag*, *Url*, *State*, *Day*, *Month* and *Year*. The edges represent the relation between various nodes as per Twitter activity.

### *B. Database Creation*

The Twitter dataset from [4] was available in a reference database compatible in Neo4j 2.3.7. In order to design new data models in Titan, data from Neo4j were exported into Comma Separated Values (CSV) format. Data from the CSV files were read using Gremlin to import the reference model into a graph variable in Titan which was further written into a JavaScript Object Notation (JSON) file for future usage. In order to create the Reduced Hop Data model in Titan, the reference data model in Neo4j was modified by adding the extra edges to it. This modified model was exported into CSV format and the above mentioned steps were followed. In order to create the Indexed Data model, the reference data model was loaded into Titan and indexes were created using the Titan Management system. Figure 4 describes the process of creating data model in Titan which uses commands as shown in Figure 5.

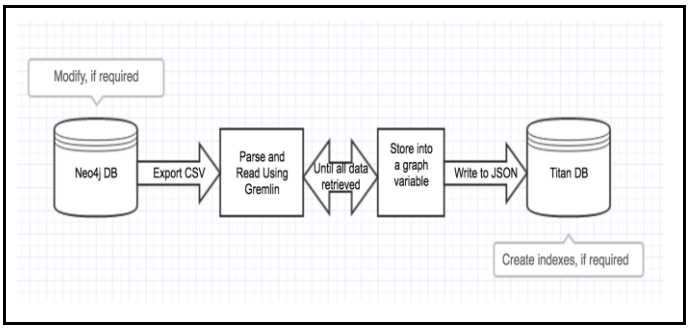


**Figure 4: Process of importing data into Titan database**

The three models created are different in terms of number of nodes, edges and space occupied by indexes and number of hops on the traversal for each query. Hence, each model has a different set of queries written in Gremlin. Table 2 describes the size details of each of the model.

```
g=TinkerGraph.open();
g.io(graphson()).readGraph('data/Tweets.json');
vs=[] as List;
new File("../Nodes/State.txt").eachLine{l->p=l.split(":");vs<<p[1];}
vs.eachWithIndex{v,index->g.addVertex(T.id, "4"+index,'label','STATE','name',v)};
vs.clear();
g.io(graphson()).writeGraph('data/StateTweets.json');
```

**Figure 5: Gremlin command for data import**

| | Ref Model | Indexed Model | | Reduced Hop Model | |
|---|---|---|---|---|---|
| | **Size** | **Size** | **% Change** | **Size** | **% Change** |
| Total Size | 251.8 MB | 281.6 MB | 11.83 | 260.2 MB | 3.33 |
| Number of Nodes | 137074 | 137074 | 0 | 137074 | 0 |
| Number of Edges | 446701 | 446701 | 0 | 474777 | 6.28 |

**Table 2: Size of Data Models**

### *C. Titan Configuration*

By default, Titan uses Cassandra DB. In order to change the storage backend to Oracle Berkeley DB, certain configuration changes were made using commands as shown in Figure 6.

```
conf = new BaseConfiguration();
Configuration conf = new BaseConfiguration();
config = TitanFactory.build();
config.set("storage.backend", "berkeleyje");
config.set("storage.directory","../dbNoIdx");
g = config.open();
g.io(graphson()).readGraph('data/Twitter.json');
```

**Figure 6: Gremlin command to configure storage backend for Reference Model and Reduced Hop Model**

In order to enable mixed indexing, index backend was setup using the commands shown in Figure 7.

```
conf = new BaseConfiguration();
Configuration conf = new BaseConfiguration();
config = TitanFactory.build();
config.set("storage.backend", "berkeleyje");
config.set("storage.directory","../dbIdx");
config.set("index.search.backend","elasticsearch")
config.set("index.search.local-mode",true)
config.set("index.search.client-only",false)
config.set("index.search.directory","../dbIdx"+File.separator+"es");
g = config.open();
g.io(graphson()).readGraph('data/Twitter.json');
```

**Figure 7: Gremlin command to configure storage backend and index backend for Indexed Model**

## V.

### D. *Performance Metrics*

Performance metric used is average response time of each query. Each query was run over 35 times, where the first 5 runs were not considered for calculation due to initial warm up. The initial 5 runs showed higher values which decreased consequently and become stable thereafter. In order to calculate the execution time, the sample code snippet as shown in Figure 8 was run, where the variable *vquery* returned the result of the concerned query whose execution time is being evaluated. For each of the three database models, the 25 queries mentioned in [4] were run to calculate the average execution time. All the queries were run on Intel 2.7 GHz CPU with 8GB RAM running on MAC OS X.

```
t=System.currentTimeMillis();
vquery();
query_time=System.currentTimeMillis()-t;
```

**Figure 8: To calculate query execution time**

### E. *Experimental Results*

The average query response time for each query in each database model is presented in Table 3 and graphically shown in Figure 9. The box-plot for the best model is shown in Figure 10.

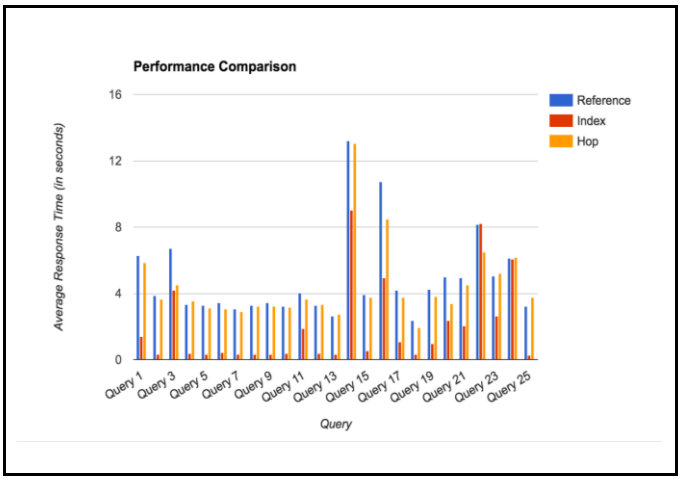


**Figure 9: Performance Comparison of average query response time**

### F. *Important Findings*

The results show that there is considerable improvement in query response time for the indexed data model with respect to reference data model and the reduced hop data model. Few of the observations are presented as follows. There is 77.6% improvement for the Q1 in indexed model from reference

| | Ref. Model | Indexed Model | | Reduced Hop Model | |
|---|---|---|---|---|---|
| Query | Exec. time (s) | Exec. time (s) | % Change | Exec. time (s) | % Change |
| Q1 | 6.28 | 1.4 | -77.66 | 5.86 | -6.81 |
| Q2 | 3.84 | 0.33 | -91.52 | 3.65 | -4.97 |
| Q3 | 6.71 | 4.21 | -37.33 | 4.49 | -33.15 |
| Q4 | 3.33 | 0.37 | -88.94 | 3.53 | 6.11 |
| Q5 | 3.29 | 0.31 | -90.57 | 3.11 | -5.64 |
| Q6 | 3.43 | 0.41 | -87.92 | 3.07 | -10.54 |
| Q7 | 3.04 | 0.31 | -89.91 | 2.9 | -4.61 |
| Q8 | 3.26 | 0.34 | -89.58 | 3.2 | -1.74 |
| Q9 | 3.41 | 0.31 | -90.8 | 3.23 | -5.32 |
| Q10 | 3.22 | 0.36 | -88.71 | 3.17 | -1.78 |
| Q11 | 4 | 1.89 | -52.87 | 3.65 | -8.8 |
| Q12 | 3.28 | 0.38 | -88.29 | 3.31 | 0.76 |
| Q13 | 2.63 | 0.34 | -87.19 | 2.73 | 3.9 |
| Q14 | 13.22 | 9.03 | -31.67 | 13.03 | -1.41 |
| Q15 | 3.91 | 0.56 | -85.59 | 3.76 | -3.85 |
| Q16 | 10.75 | 4.93 | -54.11 | 8.46 | -21.36 |
| Q17 | 4.2 | 1.05 | -75.06 | 3.75 | -10.69 |
| Q18 | 2.34 | 0.33 | -85.8 | 1.96 | -16.32 |
| Q19 | 4.22 | 0.94 | -77.75 | 3.79 | -10.29 |
| Q20 | 5.02 | 2.35 | -53.2 | 3.41 | -32.1 |
| Q21 | 4.96 | 2.06 | -58.52 | 4.51 | -8.99 |
| Q22 | 8.17 | 8.21 | 0.53 | 6.47 | -20.73 |
| Q23 | 5.06 | 2.62 | -48.18 | 5.19 | 2.52 |
| Q24 | 6.14 | 6.06 | -1.38 | 6.17 | 0.55 |
| Q25 | 3.2 | 0.28 | -91.21 | 3.78 | 18.17 |

**Table 3: Average query response time in seconds**

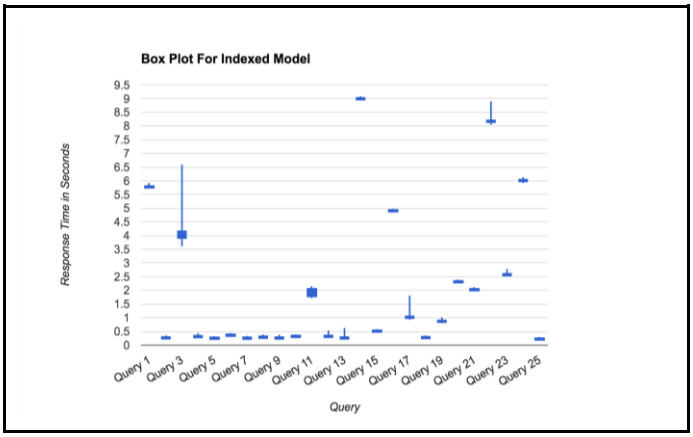


**Figure 10: Box-Plot for Query Performance of Indexed Model**

model. Because there is an index on *month* property of Tweet node for filtering and another two indexes on *screen_name* and *category* of *User* node for fetching the query results. Similarly,

for Q1, Q2, Q4, Q5, Q6, Q7, Q8, Q9, Q10, Q12, Q13, Q15, Q17, Q18, Q19 and Q25 the improvement ranges from 75 - 90% over reference model when indexed model is used. Other queries like Q3, Q11, Q14, Q16, Q20, Q21, Q23 we see an improvement of less than around 50 % in indexed model. On overall, it is evident that the indexed model is the best among the two proposed models in terms of performance with respect to the reference model.

## V. CONCLUSION AND FUTURE WORK

Graph Databases are optimized for storing and querying graphs containing large number of vertices and edges distributed across multi machine cluster. Graph database has advantage over traditional databases for large connected network. Our experiment has two major parts. First load the data from Neo4j to Titan and then create two other model so that the query performance is better than the reference models. In our experiment, we started with a Neo4j graph database. We extracted the node and relation information from Neo4j as CSV file. We parsed the CSV file into Titan readable JSON using Groovy and Gremlin. Once the JSON file is created it is placed in the data directory of the Titan server. The GraphSON reader utility is then used to read the information of the graph from the stored JSON file in data directory. After Data importing we designed the schema of our new models. The reference model had no index implemented. So, for our first model we created a mixed index on name attribute of State Node, screen_name attribute of User node, name attribute of Hashtag node, category attribute of User node and day attribute of Tweet node. The mixed index was created to address inequality predicates in the queries. We then created composite indexes on all unique key attributes of the nodes e.g. screen_name, tid, name (*State*), name (*Hashtag*), url along with hashtag, category, state and sub-category. For the other model, we tried to reduce hop in graph traversal. Therefore, we have added an edge between Hashtag and State named *TAGFROM* with properties day, month, year, sub_category. The best model in our experiment has turn out to be the model with index. As part of this project we were required to keep two separate models, one with applied indexing and another with reduced hops. As future work, we can create a model that has both index applied and hops reduced. In that way, we can understand the effect of both the changes together.

## REFERENCES


[1] Arijit Khan *et al.*, "Neighborhood Based Fast Graph Search in Large Networks" in ACM SIGMOD, Greece, pp. 901-912, June 12-16, 2011

[2] Konstantinos Xirogiannopoulos *et al.*, "GraphGen: Exploring Interesting Graphs in Relational Data" in VLDB, Hawaii, Vol 8. pp. 2032-2035, August 31- September 4, 2015.

[3] Rene Peinl, Florian Holzschuher, "Performance of graph query languages: Comparison of cypher, gremlin and native access in Neo4j", in EDBT, Italy, pp. 195-204, March 18-22, 2013.

[4] Prashant Kumar, "Graph Data Modeling for Political Communication on Twitter", M.S. thesis, Dept. Comp Sc, Iowa State Univ., Ames, Iowa, 2016.

[5] https://www.npmjs.com/package/gremlin-importer

[6] http://gremlindocs.spmallette.documentup.com