Workload Analysis on Distributed Graph Databases

CS5352 Course Project, Spring 2017

Manisha Siddartha Nalla
Masters in Computer Science
Department of Computer Science
Texas Tech University
Lubbock, USA
manisha-siddartha.nalla@ttu.edu

Bharath Kumar Kande
Masters in Computer Science
Department of Computer Science
Texas Tech University
Lubbock, USA
bharath.kumar.kande@ttu.edu

***Abstract*—This electronic document is a detailed report on distributed graph databases, characteristics that make graph databases a good replacement for traditional relational models. Generation of Traversals, workload distributions generation for graph dataset to mimic real world datasets. Also, Analysis on those workloads to see interesting properties that can be found using graph datasets. And, finally evaluate the performance of graph query language against native coding with the results of running times of, traversal and workload generation, at hand.**

Keywords - graph database; workloads; ; traversal; gremlin; titan; tinkerpop;

I. INTRODUCTION

Large-scale graphs become increasingly important in many applications and domains such as social network and scientific computing. Many of these use cases require high-performance distributed graph databases, designed for serving continuous updates from clients and, at the same time, answering complex queries regarding the current graph in an online manner. Graphs already obtain connectivity information about various vertices. However, this is not enough as the graphs may not be accessed in the same way as they were formed. Hence, it is critical to analyze the real workloads and facilitate later processing.

Increasing need for fast distributed processing of large-scale graphs such as the web graph and various social networks. Graph databases are more and more used to address the increase of data complexity. Indeed, graphs can be used to model many interesting problems. For example, it is quite natural to represent a social network as a graph. Large-scale property graph traversals are particularly challenging for distributed graph storage systems. Most existing graph systems implement a "level-synchronous" breadth-first search algorithm that relies on global synchronization in each traversal step. This performs well in many problem domains; but a rich metadata management system is characterized by imbalanced graphs, long traversal lengths, and concurrent workloads, each of which has the potential to introduce or exacerbate stragglers that lead to low overall throughput for synchronous traversal algorithms. Previous research indicated that the straggler problem can be mitigated by using asynchronous traversal algorithms, and many graph-processing frameworks have successfully demonstrated this approach. Such systems require the graph to be loaded into a separate batch-processing framework instead of being iteratively accessed. Unlike other databases, relationships take priority in graph databases. This means your application doesn't have to infer data connections using things like foreign keys or out-of-band processing, such as MapReduce. The data model for a graph database is also significantly simpler and more expressive than those of relational or other NoSQL databases. Graph databases are built for use with transactional (OLTP) systems and are engineered with transactional integrity and operational availability in mind.

For this project, we deal with one such graph database framework, "Titan", an open source Java based framework, that supports for very large graphs which scale with number of machines in the cluster. Titan like every other graph database supports for many concurrent transactions and operational graph processing. Vertex centric indices provide vertex-level querying to solve infamous super node problem. Provides an optimized representation to allow for efficient use of storage and speed of access.

Characteristics of titan that make it unique are, titan possess linear-scalability, a non-redundant storage, easy to model architecture, and distributed in nature. Titan supports various storage backends like Apache Cassandra, Apache HBase, Oracle BerkeleyDB. Support for global graph data analytics, reporting, and ETL through integration with big data platforms: Apache Spark, Apache Giraph, Apache Hadoop. Support for geo, numeric range, and full-text search via: ElasticSearch, Solr, Lucene. Native integration with the TinkerPop graph stack: Gremlin graph query language, Gremlin graph server, Gremlin applications.

Here we aim at finding all Critical nodes in a graph, i.e., we try to find cut vertices in a connected graph. In the graph, every node is connected to every other node by some path. We calculate 1st-order and 2-nd order connectedness by generating 1st and 2nd order traversals and workloads respectively. We use Gremlin, Titan's query language to retrieve data from and modify data in the graph. Gremlin is a path-oriented language which succinctly expresses complex graph traversals and mutation operations. It is a functional language whereby traversal operators are chained together to form path-like expressions.

Identify applicable sponsor/s here. *(sponsors)*

II. RELATED WORK

A. *Titan Tinkerpop Driver*

Tinkerpop Driver is a starter project that demonstrates how a basic graph application can communicate with a Titan Server.

Titan implements the graph APIs defined in Apache TinkerPop, so much of that project code can work with any Gremlin Server. This project and its documentation demonstrates how to connect to Gremlin Server through Java using the Gremlin Driver that is distributed by TinkerPop.

B. *Work on Centrality*

There are many works on centrality that find the most popular nodes based on closeness between nodes, betweenness between nodes and also few degree based approaches.

The Research we carry out now is very much similar to degree based centrality, but unlike the degree centrality where you calculate the popularity of nodes based on 1st order degree, we now calculate popularity considering 1st and 2nd order degree values. Also, traversals get much accurate with the level of order of degree we consider. Eigenvector centrality is a common measure of the importance of nodes in a network. Here we show that under common conditions the eigenvector centrality displays a localization transition that causes most of the weight of the centrality to concentrate on a small number of nodes in the network. In this regime, the measure is no longer useful for distinguishing among the remaining nodes and its efficacy as a network metric is impaired. As a remedy, we propose an alternative centrality measure based on the nonbacktracking matrix, which gives results closely similar to the standard eigenvector centrality in dense networks where the latter is well behaved but avoids localization and gives useful results in regimes where the standard centrality fails.

Each Centrality measures for some sort of importance of nodes. Like degree centrality maintains numerous contacts with other network nodes. Nodes have higher centrality to the extent they can gain access to and/or influence over others. A central node occupies a structural position (network location) that serves as a source or conduit for larger volumes of information exchange and other resource transactions with other nodes. Central nodes are located at or near the center in network diagrams of social space. In contrast, a peripheral node maintains few or no relations and thus is located spatially at the margins of a network diagram.

While the closeness centrality a central ego node has minimum path distances alters. A node that is close to many others can quickly interact and communicate with them without going through many intermediaries. Thus, if two nodes are not directly tied, requiring only a small number of steps to reach one another is important to attain higher closeness centrality.

On the other hand, A central node occupies a “between” position on the geodesics connecting many pairs of other nodes in the network. As a cut point in the shortest path connecting two other nodes, a between node might control the flow of information or the exchange of resources, perhaps charging a fee or brokerage commission for transaction services rendered. If more than one geodesic links a pair of nodes, assume that each of these shortest paths has an equal probability of being used.

In Consideration of all these centrality measures our work moves similar to degree based centrality but with variation of extending the property of degree to next order traversals.

III. RESOURCES

A. *Dataset*

This file contains data from Wikipedia voting on promotion to administrator-ship. This file wikivote.txt contains information of 7115 nodes and 103,689 edges (voting relationship).

This dataset is available from-
http://snap.stanford.edu/data/wiki-Vote.html

Snapshot of the dataset is as follows-

```
# FromNodeId    ToNodeId
30  1412
30  3352
30  5254
30  5543
30  7478
3   28
3   30
3   39
3   54
3   108
3   152
3   178
3   182
3   214
```

Figure 1. wiki-vote.txt dataset

B. *Downloads*

a) Java8

Latest version of java, Java8 is required.

b) Hadoop-titan-1.0.0

Titan 1.0.0 itself is compatible with both Hadoop 1 and 2, but TinkerPop’s Hadoop - gremlin requires Hadoop 1. Hadoop - gremlin contains graph computer implementations for running traversals on Spark and Giraph, among other bits.

Available for download at-

Identify applicable sponsor/s here. *(sponsors)*

http://s3.thinkaurelius.com/downloads/titan/titan-1.0.0-hadoop1.zip

c) Tinkerpop driver

Titan implements the graph APIs defined in Apache TinkerPop, so much of this example code can work with any Gremlin Server. This project demonstrates how to connect to Gremlin Server through Java using the Gremlin Driver that is distributed by TinkerPop.

Available for download at-
https://github.com/pluradj/titan-tp3-driver-example

IV. EXPERIMENTS

A. Environment Setup

a) Configuring Gremlin server

Configure gremlin-server. yaml file with the following settings-

i) Set host: Set host address of underlying server. In our case, host value is localhost.

ii) Set port: Set port address on which the server runs.

iii) Write ScriptEngines: Script engines has a set of properties like gremlin-groovy, serializers and metrics. Gremlin-groovy in turn has another property called Scripts. Scripts has a list of scripts to execute when this configuration file is used to activate server. Here we input a script, 'generate-modern.groovy'.

b) Configuring a Groovy script

In this Groovy Script, we write some source code to import a graph.

i) Define a Global variable to map all variables to tinkerpop server.

ii) Create a graph traversal, assigned to some variable, then load entire graph into it.

iii) Read through each line of file, check if the node already exists, else create a node and assign an id. Thus, reading entire graph into some variable 'g'.

c) Running Gremlin server

i) Run gremlin server passing the configuration file just built. The command is as follows-

$ bin\gremlin-server.bat conf\gremlin-server.yaml

ii) Server starts running on the given server IP address and port number given in configuration file.

d) Configuring Driver-settings

We configure the driver settings to use gremlin server from client program. Driver settings file has the following fields-

i) Set hosts: Set hosts field with IP address of server running gremlin server component.

ii) Set port: Set port field with port address of port running gremlin server process.

iii) Set serializer: Set 'serializeResultToString' to false. If this property is set to False, the results of queries can be accessed as objects thereby we can use the object's properties. Else if, this property when set to True, passes results as list with strings.

e) Running a client

To run client from console, one need to run the following command-

$ bin\gremlin.bat

To connect to remote server, run the following command-

$:remote connect tinkerpop.server conf\remote-objects.yaml

To run remote queries, append ': >' to query. One good example is as follows-

$: > g.V( )

Once all these steps have been performed, distributed graph database is setup to query on the given dataset.

B. Workload Generation

Preprocessing done on the input in order to identify the traversal of graph. Each node on traversal counts to workload. We have two types of distributions, and 6 distributions in whole. They are as follows-

a) Uniform distribution step0

i) It takes voters.txt as the input file.

ii) Selects a number using random generator. Each selection is counted as one client request.

iii) The above step is repeated for 10000 times.

iv) All the selected nodes from the input file using the random generator is stored in a separate file.

b) Uniform distribution step1

i) It takes voters.txt as the input file.

ii) Selects a number using random generator. Each selection is counted as one client request.

Identify applicable sponsor/s here. *(sponsors)*

iii) Calculate the neighbors by querying the graph database

neighbouring_userIds.add(rs.getVertex().property("userId").value().toString())
Checks the given neighbor id(user id) is present in the graph database and adds its neighbors into the neighbouring_userIds arraylist.
both() – retrieves neighbors

iv) The graph uses connective and match predicate strategies to search neighbors if it finds given userId in the list of nodes. Time complexity when it uses these strategies will be log(n).
v) The selected userId and its corresponding neighbors are placed one after the other in a column.
vi) The above steps are repeated 10000 times.
vii) It uses different search strategies when the nodeId is not present like pattern matching.

c) Uniform distribution step2

i) It takes voters.txt as the input file.
ii) Selects a number using random generator. Each selection is counted as one client request.
iii) Calculate the neighbors by querying the graph database

neighbouring_userIds.add(rs.getVertex().property("userId").value().toString());

Checks the given neighbor id(user id) is present in the graph database and adds its neighbors into the neighbouring_userIds arraylist.

iv) Also calculates second level of neighbors (neighbors of neighbors) using the following query.

```
client3.submit("g.V().has('userId',userId).as('neighbours','neighboursOfneighbours','neighbours^2')"+
".select('neighbours','neighboursOfneighbours','neighbours^2').by(__.both().values('userId').dedup().fold())"+
".by(__.both().both().values('userId').dedup().fold())"
```

.both().both() – retrieves neighbors of neighbors
dedup() – avoids visiting a node more than once

v) The selected userId and its corresponding neighbors and their neighbors are placed one after the other in a column.
vi) The above steps are repeated 10000 times.
vii) The graph uses connective and match predicate strategies to search neighbors if it finds given userId in the list of nodes. Time complexity when it uses these strategies will be log(n).
viii) It uses different search strategies when the nodeId is not present like pattern matching.

d) Degree based distribution step0

i) It takes voters.txt as the input file.
ii) Selects a number using random generator, probability of each number being picked is proportional to its degree.
iii) The above step is repeated for 10,000 times.
iv) All the selected nodes from the input file using the random generator is stored in a separate file.

e) Degree based distribution step1

i) It takes voters.txt as input file.
ii) Selects a number using random generator, probability of each number being picked is proportional to its degree.
iii) Calculate the neighbors by querying the graph database same as like uniform distribution.
iv) The graph uses connective and match predicate strategies to search neighbors if it finds given userId in the list of nodes. Time complexity when it uses these strategies will be log(n).
v) The selected userId and its corresponding neighbors are placed one after the other in a column.
vi) The above steps are repeated 10000 times.
vii) It uses different search strategies when the nodeId is not present like pattern matching.

f) Degree based distribution step2

i) It takes voters.txt as input file.
ii) Selects a number using random generator, probability of each number being picked is proportional to its degree.
iii) Calculate the neighbors by querying the graph database same as in uniform distributions.
iv) Also calculates second level of neighbors using the query as in uniform distribution.
v) The selected usedId and its corresponding neighbors and their neighbors are placed one after the other in a column.
vi) The above steps are repeated 10,000 times.
vii) The graph uses connective and match predicate strategies to search neighbors if it finds given userId in the list of nodes. Time complexity when it uses these strategies will be log(n).
viii) It uses different search strategies when the nodeId is not present like pattern matching.

Identify applicable sponsor/s here. *(sponsors)*

C. Workload Analysis

a) Uniform distribution step0

In this line chart, we observe only lower workloads to due to the low probability of picking the nodes. In this line chart, X-axis represents node Ids and Y-axis represents the workload.

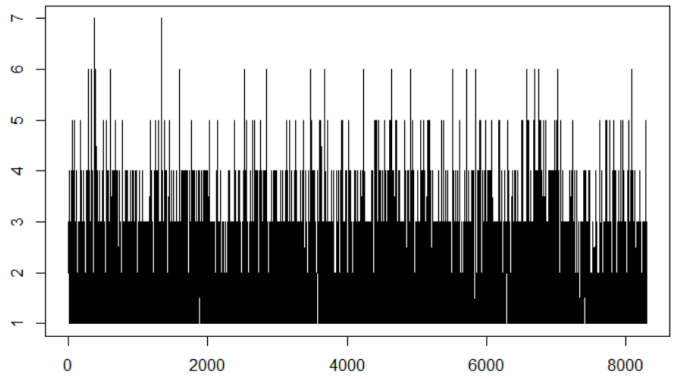

Figure 2. uniform distribution step0 line chart

This curve is a clear indication of fewer workload levels and imply that large number of nodes have very lower workload of just one visit. In this curve, X-axis represents Ids sorted in decreasing order of workloads and Y-axis represents workloads.

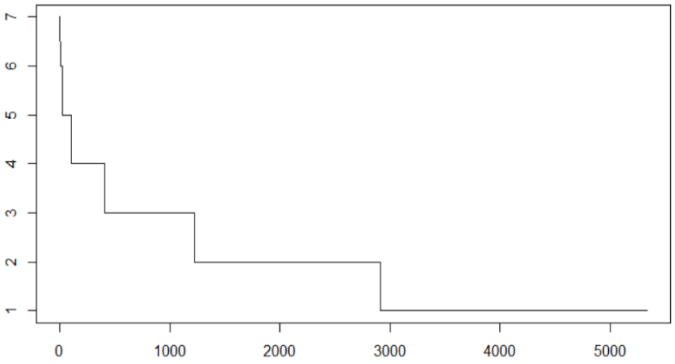

Figure 3. uniform distribution step0 curve

b) Uniform distribution step1

In this line chart, we observe only maximum workload reaches around 1400 visits and only one node reaches huge visits. Thus, implying the popularity of this node in 1st order traversal. In this line chart, X-axis represents node Ids and Y-axis represents the workload.

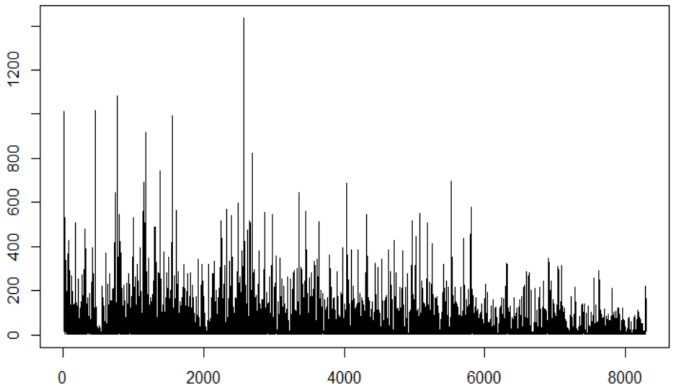

Figure 4. uniform distribution step1 line chart

This curve indicates many workload levels than uniform 0. Many nodes have near equal 0 workload and fewer nodes have very large workloads of around 1400 visits. In this curve, X-axis represents Ids sorted in decreasing order of workloads and Y-axis represents workloads.

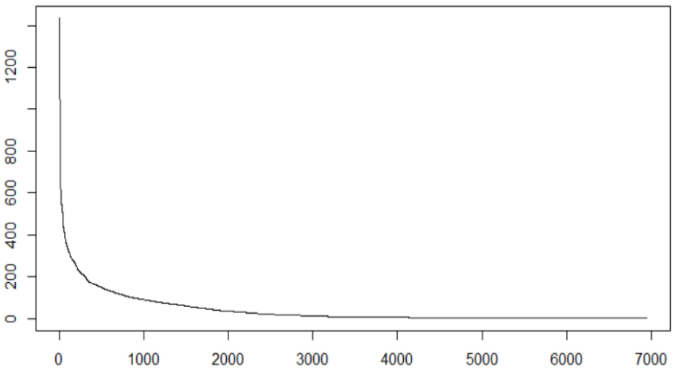

Figure 5. uniform distribution step1 curve

c) Uniform distribution step2

In this line chart, we observe only maximum workload reaches around 10000 visits and only one node reaches huge visits. Also, many other nodes have near equal large workloads. The darker the chart, larger the workloads. Thus, implying the popularity of nodes in 2nd order traversal. In this line chart, X-axis represents node Ids and Y-axis represents the workload.

Identify applicable sponsor/s here. *(sponsors)*

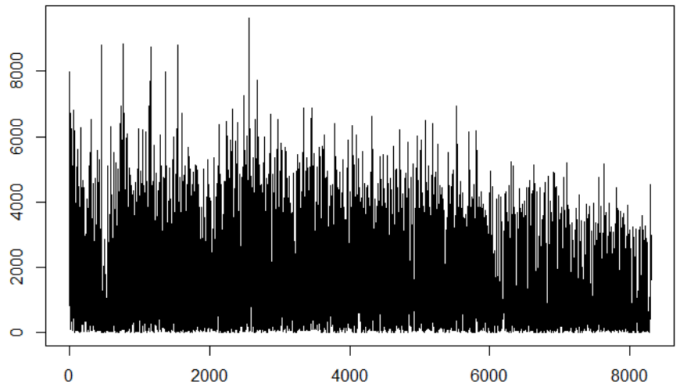

Figure 6. uniform distribution step2 line chart

This curve indicates many workload levels than uniform 0 and uniform 1. Only few nodes have lower workload and fewer nodes have very large workloads of around 10000 visits. Rest all the nodes have optimum workload, thus distributing near proportionately. In this curve, X-axis represents Ids sorted in decreasing order of workloads and Y-axis represents workloads.

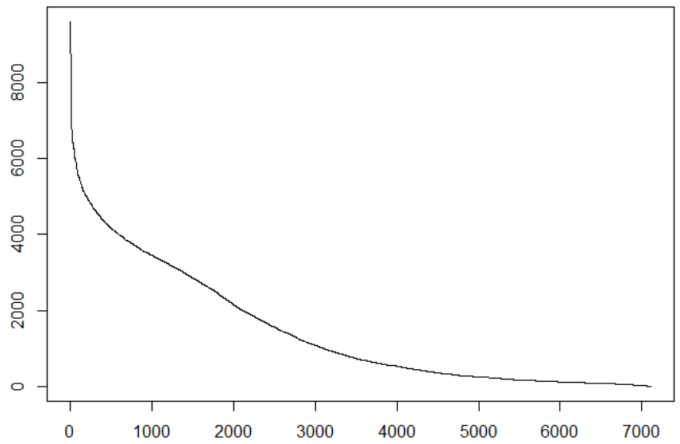

Figure 7. uniform distribution step2 curve

d) Degree based distribution step0

In this line chart, we observe good number of workload levels compared to uniform 0 distribution. Even in this chart, same as in uniform, there exists a node with Id between 2000 and 3000 that is more popular. In this line chart, X-axis represents node Ids and Y-axis represents the workload.

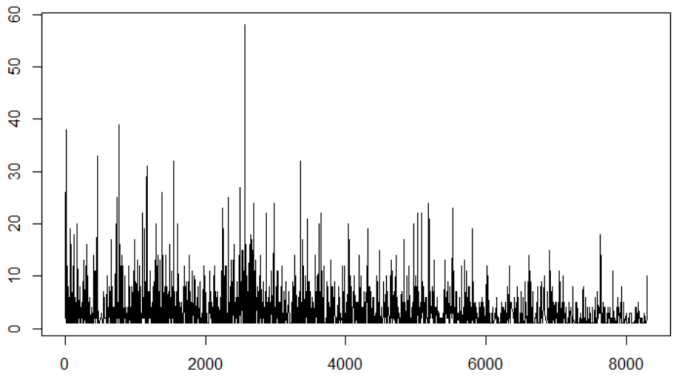

Figure 8. degree distribution step0 line chart

This curve indicates many workload levels than uniform 0. Large number of nodes have lower workload and fewer nodes have very large workloads of around 50 to 60 visits. In this curve, X-axis represents Ids sorted in decreasing order of workloads and Y-axis represents workloads.

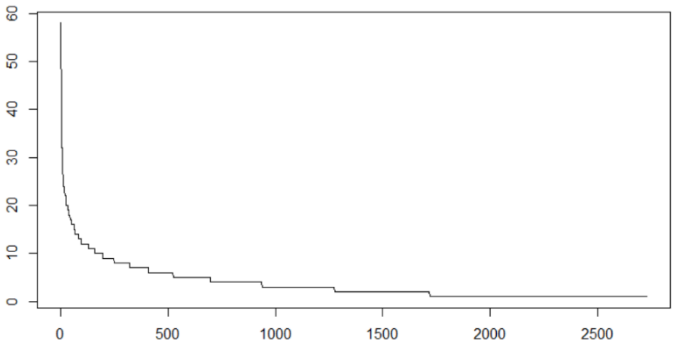

Figure 9. degree distribution step0 curve

e) Degree based distribution step1

In this line chart, we observe good number of workload levels compared to uniform 0 distribution. Even in this chart, same as in uniform, there exists a node with Id between 2000 and 3000 that is more popular with 5000 visits. In this line chart, X-axis represents node Ids and Y-axis represents the workload.

Identify applicable sponsor/s here. *(sponsors)*

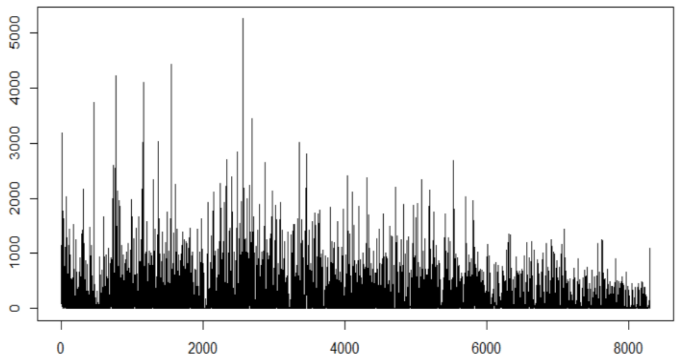

Figure 10. degree distribution step1 line chart

This curve indicates many workload levels than uniform 0 and degree 0. Large number of nodes have lower workload and fewer nodes have very large workloads of around 4000 to 5000 visits. In this curve, X-axis represents Ids sorted in decreasing order of workloads and Y-axis represents workloads.

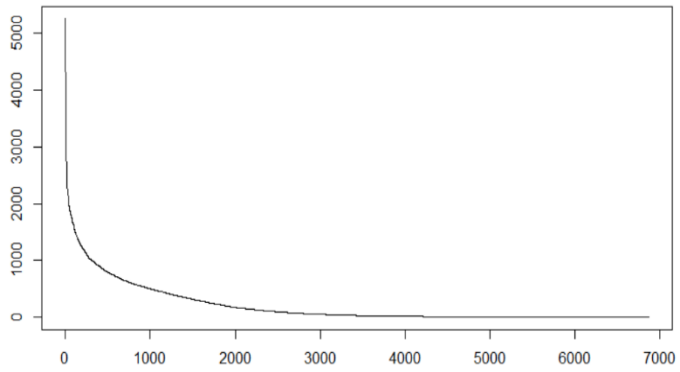

Figure 11. degree distribution step1 curve

f) Degree based distribution step2

In this line chart, we observe good number of workload levels compared to all uniform and degree distributions. Even in this chart, same as in uniform, there exists a node with Id between 2000 and 3000 that is more popular with around 15000 visits. And darker this chart region, more the workload density. In this line chart, X-axis represents node Ids and Y-axis represents the workload.

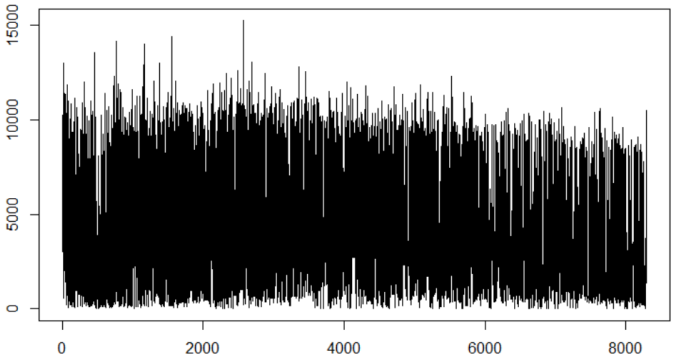

Figure 12. degree distribution step2 line chart

This curve indicates different behavior than usual. Fewer nodes have large workloads between 10000 to 15000. Rest large number of nodes possess a up and down curve implying special behavior. In this curve, X-axis represents Ids sorted in decreasing order of workloads and Y-axis represents workloads.

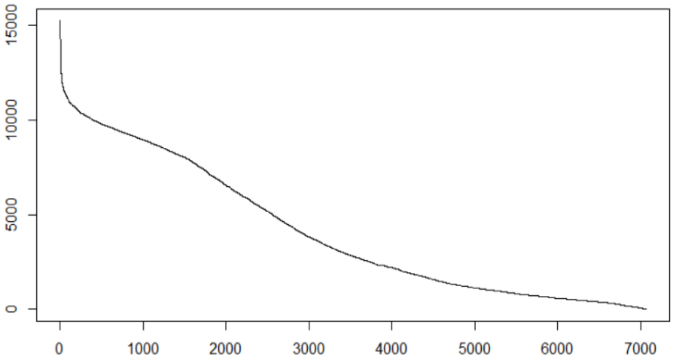

Figure 13. degree distribution step2 curve

D. Performance Analysis

Running time for workload generation is calculated on both titan using gremlin and native java program.

Distribution	Workload Generation Time in Titan (in sec)	Workload Generation Time in Java (in sec)
Uniform step 0	1	2
Uniform step 1	5	2
Uniform step 2	20	3068
Degree step 0	1	5
Degree step 1	16	72
Degree step 2	110	4842

Table 1. Performance Analysis Table

Identify applicable sponsor/s here. *(sponsors)*

V. CONCLUSION

Results clearly emphasize the performance step-up from traditional storage systems to graph database systems. We learnt to work with graph databases, generated our own traversals to mimic real world traversals thereby counting to real world workloads. Identified popular and least popular nodes from large set of nodes. Also. Able to identify the dense regions of graph. Performance Analysis show that titan has a time complexity of O(log V).

VI. FUTURE WORKS

- Can conduct experiments on larger graphs with sizes of few million nodes.
- Can generate different distributions to create more realistic traversals and workloads that mimic real workloads.
- Optimize the traversals by analyzing traversal strategies like ConnectiveStrategy, MatchPredicateStrategy and many other. (Use explain() on traversal to see a list of strategies)
- Learn the patterns and predict the traversal based on prediction algorithms.

VII. ROLES AND RESPONSIBILITIES

Manisha Siddartha Nalla was mainly involved in program development in java and program development in titan with gremlin queries.

Bharath Kumar Kande, was involved in setting up distributed graph databases and program design for both uniform and degree based approaches.

Overall, we had a privilege of sharing the responsibilities equally and complete the project successfully.

ACKNOWLEDGMENT

I would like to extend my sincere thanks to Dr. Yong Chen. I am highly indebted to Dr. Dong Dai, for his guidance and constant supervision as well as for providing necessary information regarding the project & also for his support in completing the project.

REFERENCES

[1] Rodriguez, M.A., "Gremlin's Graph Traversal Machinery," Cassandra Summit, September 2016.

[2] Matthias Broecheler, http://s3.thinkaurelius.com/docs/titan/1.0.0: Lead developer and theorist.

[3] Pluradj, Titan Tinkerpop Driver (An Open Source Project)

[4] Salim Jouili, An Empirical Comparision of graph databases,2010.

[5] M. I. Jordan (ed). (1998) ."Learning in Graphical Models". MIT Press.

[6] Ladialav Hluchy Marek Ciglan, Alex Averbuch. Benchmarking traversal operations over graph databases. 2012 IEEE 28th International Conference on Data Engineering Workshops, 2012.

GitRepo: https://bharathkande.github.io/AOS_Course_Project/

Identify applicable sponsor/s here. *(sponsors)*