Towards Emulation of Large Scale Complex Network Workloads on Graph Databases with XGDBench

Miyuru Dayarathna
School of Computer Engineering
Nanyang Technological University, Singapore
/ JST CREST
miyurud@ntu.edu.sg

Toyotaro Suzumura
IBM Research /
University College Dublin / JST CREST
suzumura@acm.org

***Abstract*—Graph database systems are getting a lot of attention in recent times from the big data management community due to their efficiency in graph data storage and powerful graph query specification abilities. In this paper we present a methodology for modeling workload spikes in a graph database system using a scalable benchmarking framework called XGDBench. We describe how two main types of workload spikes called data spikes and volume spikes can be implemented in the context of graph databases by considering realworld workload traces and empirical evidence. We implemented these features on XGDBench which we developed using X10. We validated these features by running workloads on Titan which is a popular open source distributed graph database server. We observed the ability of XGDBench in generating realistic workload spikes on Titan. The distributed architecture of XGDBench promotes implementation of such techniques efficiently through utilization of computing power offered by distributed memory compute clusters.**

***Keywords*-Database management; Distributed databases; Graph databases; Graph theory; Benchmark testing; Performance analysis; Cloud computing;**

I. Introduction

Scalable big graph data mining has become of utmost importance in recent years due to the prevalence of massive graph data sets in multiple domains. For example, Facebook social network had 1.23 billion active users[1] (i.e., vertices) as of December 31, 2013 while Twitter has 241 million monthly active users with more than 500 million tweets sent per day by the whole system[2]. Modern sequencers generate large graph data sets with millions (and even billions) of vertices de Bruijn graphs [8] for which scientists are still searching for efficient algorithms for finding Hamiltonian cycles in such massive graph data sets.

Graph data storage has attracted lot of attention in recent years due to such increasing numbers of applications that require storage and processing of huge amounts of information in the form of graphs [19][26]. Graph database model is a no-sql storage model which has proven to be more successful in applications that store and process graph information [29]. Graph Database Management Systems [16] which follow a networked data model are increasingly deployed in application areas such as social networks, bioinformatics, cyber security, automotive traffic modeling, etc. Graph databases are intended for addressing the challenge of low latency online query processing on graph data sets [7][30]. In recent times graph databases related applications have started appearing in clouds [22].

Number of graph database systems have recently appeared in order to cater the needs of graph data management and online graph analysis [20]. Neo4j [29], DEX [23], OrientDB [28], Trinity [30], Titan [3], etc. are examples for some popular graph database systems. The field of graph databases is still relatively young and it lacks standards for benchmarking and performance comparisons [7].

In general benchmarks should be scalable in such a way that the benchmark does not become the bottleneck during the benchmarking process. Most of the current graph database benchmarks are intended for run in single compute node and are vertically scalable by adding compute resources such as CPUs, RAM, etc. Ability for horizontal scaling is an important aspect present in cloud computing systems and should be exploited in graph database benchmarks targeted for cloud systems. By considering the aforementioned prevalent issues in the area of graph database benchmarking, we introduced a new graph database benchmark called XGDBench which can realistically model property graph data and support creating different benchmarking scenarios with different workloads [11][12]. XGDBench is implemented as a distributed framework which enables it to execute workloads in a concurrent and distributed fashion.

Workload spikes are prevalent in data intensive systems such as social networking sites (Facebook, Twitter, etc.) reference content sites such as Wikipedia, search engines such as Yahoo!, etc. [5]. Since graph data storage systems are becoming an important back-end component in such systems it is important to measure the capabilities of graph storage systems to withstand such workload spikes.

In this work we describe how we can leverage distributed execution infrastructure in the context of conducting real

[1] http://newsroom.fb.com/Key-Facts
[2] https://about.twitter.com/company

978-1-4799-5057-7/14 $31.00 © 2014 IEEE
DOI 10.1109/BigData.Congress.2014.140

world graph database benchmarking with workload spikes. We describe a methodology for generation of workloads with spikes by following a workload multiplication based technique. We observed XGDBench's ability for generating workload spikes through running a spikeful workload on a Titan graph database server instance running on a cluster of twelve compute nodes.

The remainder of this paper is organized as follows. Section II lists previous work related to graph database benchmarking and workload spike generation. The architecture of XGDBench is described in section III. Workload spike generation methodology is described in Section IV. The evaluation conducted on workload spike generation is presented in Section V. We discuss the results in Section VI. Finally, Section VII summarizes the paper and provides some future directions.

II. Related Work

Graph database benchmarking evolves from the area of database benchmarking. The famous database benchmarks such as TPC-C are built around the concept of emulating a computing environment similar to the real world operation of the target database system [32].

One of the earliest efforts in creating a graph database benchmark was made by Bader *et al.* and is called HPC Scalable Graph Analysis Benchmark [4]. However, this benchmark did not evaluate some features that are inherent to most of the graph databases such as attribute management, object labeling, etc. There have been multiple previous work on performance evaluation of graph databases. Dominguez-Sal *et al.*, made a survey of graph database performance on the HPC Scalable Graph Analysis Benchmark [14]. They evaluated the performance of four graph databases/stores: Neo4j, Jena, HypergraphDB, and DEX. In a different work Abreu *et al.* evaluated performance of RDF-3x, Neo4j, DEX, and HypergraphDB [13].

Recently, Guo *et al.* discussed challenges in benchmarking graph processing systems [18]. McColl *et al.* made a comparison of graph databases [24]. Angles *et al.* developed a microbenchmark based on social networks [1]. Similar to XGDBench their benchmark has a synthetic graph data generator. Their benchmark includes a set of low level atomic queries that model the behavior of social network users. Different from XGDBench, they use R-MAT generator for graph data generation. Furthermore, their benchmarking framework cannot operate distributed like XGDBench does. LinkBench is a database benchmark developed following the Facebook social graph. However, LinkBench depends on the ability of vertical scaling because the benchmark execution cannot be distributed like done in XGDBench [2]. BigBench is a proposal for an end-to-end big data benchmark [15]. Similar to XGDBench their work covers a data model, synthetic data generator, and workload description. However, their focus is not specifically on graph databases.

Holzschuher *et al.* made a performance comparison of graph query languages using Cypher [29], Gremlin [17], and native access of Neo4j [21]. Their work is different from our's because our focus is on creation of a graph database benchmarking platform rather than characterizing the performance of graph query languages.

Bodik *et al.* analyzed five workload spikes from four real web server traces [5]. Based on the analysis results they created a simple workload model that has important aspects of volume and data spikes. We implement a similar workload model for XGDBench to emulate scenarios of volume spikes. It should be noted that none of these works study about graph database server behavior during workload spike situations.

YCSB [10] framework was released by Yahoo! Inc. in 2010, with the motivation of creating a standard benchmark and benchmarking framework to assist evaluation of cloud data serving systems. One of the key goals of YCSB is its extensibility. The framework is composed of a workload generator client and a package of standard workloads that cover interesting parts of the performance space. The workload generator of YCSB supports definition of new workload types which motivated us for following YCSB's approach for implementing XGDBench.

III. Architecture of XGDBench

Our focus in XGDBench is to create a scalable graph database benchmark that realistically models graph database application scenarios. We use a synthetic graph data generator model called Multiplicative Attribute Graphs (MAG) as the data generator of XGDBench [25].

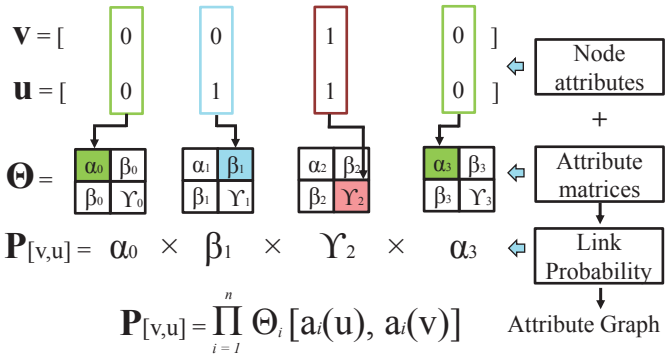
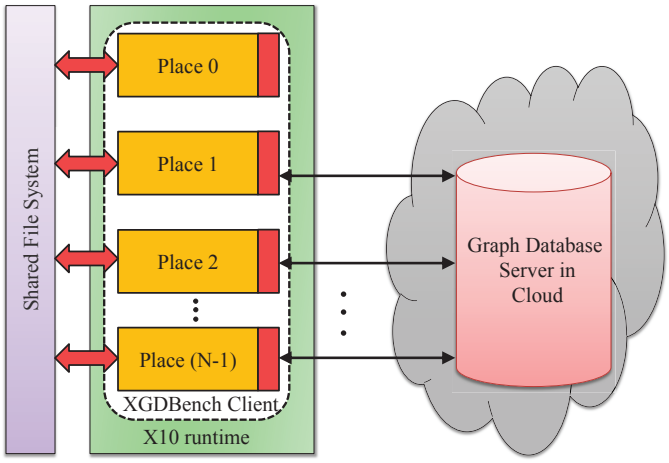


Figure 1. Multiplicative Attribute Graphs model.

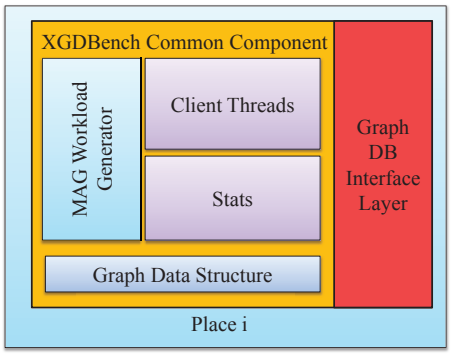
Figure 1 shows two vertices v and u each having a vector of n categorical attributes and each attribute has a cardinality d_i for i = 1,2,..., n. There are also n matrices denoted by Θ_i, $\Theta_i \in d_i \times d_i$ for i = 1,2,..., n. Each entry of Θ_i is the affinity of a real value between 0 and 1. Values α, β, and γ are floating point values between 0 and 1. Given these conditions, probability of an edge (u,v), P[u,v], is defined as the multiplication of affinities corresponding

to individual attributes as shown in Figure 1. Here $a_i(u)$ and $a_i(v)$ represent the value of i-th attribute of nodes u and v. More details of the XGDBench's data generator and evaluation of its properties are available from [11][12].

The software architecture details of XGDBench is shown in Figure 2. XGDBench Client is the software application that executes the benchmark's workloads. It consists of a MAG data generator described previously for generating the data to be loaded to the database. The workload executor drives multiple client threads. A sequential series of operations are executed by the client threads. Graph database interface layer (See Figure 2(b)) translates simple requests from client threads into calls against the graph database.



(a)



(b)

Figure 2. Architecture of XGDBench Client. (a) Abstract view of the client architecture. (b) Organization of components within single place.

XGDBench has two phases of execution called *loading phase* and *transaction phase*. The loading phase generates a property graph by using the MAG data generator and uploads the graph dataset onto the target database server. The transaction phase of XGDBench calls a method in *CoreWorkload* called *doTransaction()*, which invokes the basic operations such as database read, update, insert, scan (See Table II). The general workloads implemented on XGDBench are listed in Table I. We implemented XGDBench's workload spike generation feature only in the transaction phase because spikeful situations might not occur during a bulk data loading session.

Table I
CORE WORKLOADS OF XGDBENCH.

A : Update heavy
Workload A is a mix of 50/50 read/update workload. Read operations query a vertex V and read all the attributes of V. Update operation changes the last login time attribute of vertices. Attributes related to vertex affinity are not changed.
B : Read mostly
A mix of 95/5 read/update workload. Read/update operations are similar to A.
C : Read only
Consists of 100% read operations. The read operations are similar to A.
D : Read latest
This workload inserts new vertices to the graph. The inserts are made in such a way that the power-law relations of the original graph are preserved.
E : Short Range Scan
This workload reads all the neighbor vertices and their attributes of a Vertex A. This represents the scenario of loading the friend list of person A onto an application.
F : Traverse heavy
Consists of 45/55 mix of traverse/read operations.
G : Traverse only
Consists of 100% traverse operations.

Table II
BASIC OPERATIONS OF GRAPH DATABASES.

Operation	Description
Read	Read a vertex and its properties
Insert	Inserts a new vertex
Update	Update all the attributes of a vertex
Delete	Delete a vertex from the DB
Scan	Load the list of neighbors of a vertex
Traverse	Traverses the graph from a given vertex using BFS. This represents finding friends of a person in social networks.

XGDBench has been implemented by using the X10 programming language which is a new programming language intended for concurrent and distributed programing [6]. X10 is an open source programming language that is aimed for providing a robust programming model that can withstand the architectural challenges posed by multi-core systems, hardware accelerators, clusters, and supercomputers. The latest major release of X10 is X10 2.4 of which the applications are developed by source-to-source compilation to either C++ (Native X10) or Java (Managed X10).

We used Managed X10 when developing XGDBench. X10 has the notion of a place which resembles an individual application process (with its own address space) running in a compute node. We use the notion of place for emulating the workload spikes in XGDBench.

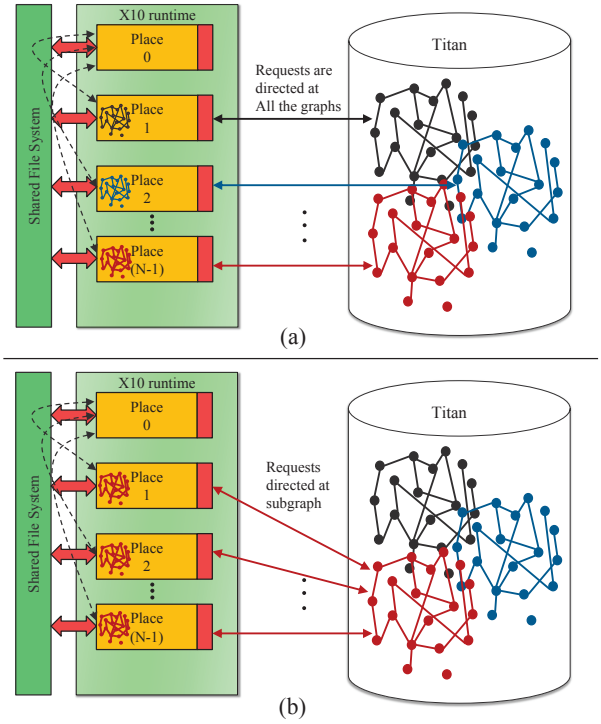


Figure 3. Methodology used for workload spike generation on XGDBench. (a) Scenario of volume spike generation (b) Scenario of data spike generation.

The workload generator of XGDBench is based on Multiplicative Attribute Graphs. Since this model is a stochastic probabilistic model there is a tendency for production of graphs with different numbers of edges (Even though there is a fixed vertex count) if a true random number generator was employed. This is because the edge probabilities that are determined by MAG algorithm are random due to the random initialization of the vertex attributes at the beginning of the graph generation process [11]. However, this issue can be easily overcome through use of Pseudo-random number generators present with X10.

IV. Workloads with Spikes

As other compute systems such as web servers, relational database servers, etc.; graph database systems are susceptible for workload spikes. Out of the different spike types, "Volume Spike" refers to an unexpected sustained increase in aggregate workload volume [5]. Another important category of spikes is called a "Data Spike" which is an unexpected increase in demand for certain data objects. For example, during an important event happening to a famous person, the data objects corresponding to that person will be accessed frequently compared to the other objects. Even though the aggregated workload volume will remain the same during such spike, we can observe a significant increase in demand for few objects which are highly sought during that time period.

The original design of XGDBench described in [11] did not have support for modeling workload spikes. This is because the XGDBench client just accepted a fixed number of threads (E.g., N) to be used during a benchmarking session. Then each thread simultaneously executes portion of the total number of operations that need to be executed on the database (i.e., `(Total number of operations)/N`). By changing the number of client threads dynamically, we can emulate a volume spike during a benchmarking session. Since XGDBench is a distributed benchmarking framework, each X10 place running in the distributed cluster can dynamically orchestrate its own set of threads. Therefore, if there are N threads and M number of places a total N×M maximum number of threads could involve in generation of the workload spike. Furthermore, we dynamically orchestrate places in different nodes (along with threads) during the workload spike which provides us a finer grain control of the shape of the workload spike compared to a technique that relies only on threads on a single node. The methodology used for generating workload spikes on XGDBench is shown in Figure 3.

Figure 3 (a) shows how a volume spike is generated. The client activities running at each place generates the same graph structure (shown in blue, black, and red colors) but with different properties (i.e., Vertex attribute labels that do not correspond to graph generation such as vertex name). During a volume spike each activity directs requests to its corresponding subgraph located on the graph database server. When we run XGDBench in a large number of places it will stress all the components of the graph database server (especially in the case of distributed servers such as Titan).

How a data spike is generated is shown in Figure 3 (b). In this case all the activities running in different places generated the same graph structures with same property values. However, during emulation of a data spike, the query requests for the graph database server will be focused only onto a particular subgraph of the entire graph database server. This is because each activity has the same contents for formulating queries; hence the graph database server needs to use the same section of its data storage to answer the queries. This is similar to a scenario that requests information of a famous person on a social graph by multiple applications simultaneously.

A. Workload Spike Emulation

In order to implement workload spikes on XGDBench we employ a multiplication factor based technique as described below. We take a general workload (See Table I) created by XGDBench and augment it by multiplying with an integer value sequence which changes over time (See Figure 4 (a)). When we apply such transformation to the original workload the end result is a workload spike as shown in Figure 4 (b). Here, we applied the same multiplication factor

based workload emulation on a real world workload trace (Yahoo! Webscope data set [33]) obtained from Yahoo!'s PNUTS storage system [9] with Workload A of YCSB [10] to demonstrate how such technique can be used to generate workload spikes.

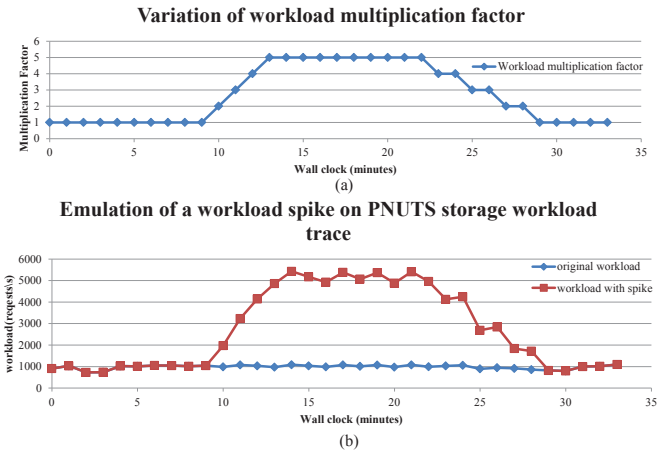


Figure 4. Emulation of a workload spike on Yahoo! PNUTS workload trace.

We implemented this multiplication factor based workload spike generation method in XGDBench as shown in Figure 5. During execution of a spiky workload with M places, at first k places (k $<$ M) start execution of the requests (E.g., when k=2, places 1 and 2 start execution of the workload). Once a time period T_i has elapsed, the remaining activities start issuing requests to the database server. All the activities will be issuing queries for a plateau time period (T_p). Next, each place completes execution of its workload as shown in Figure 5 with (T_s) spike decremental interval which results in a workload spike. By varying the number of X10 places the number of simultaneous requests issued to the graph database server can be controlled. This is demonstrated in the experiments conducted in Section V.

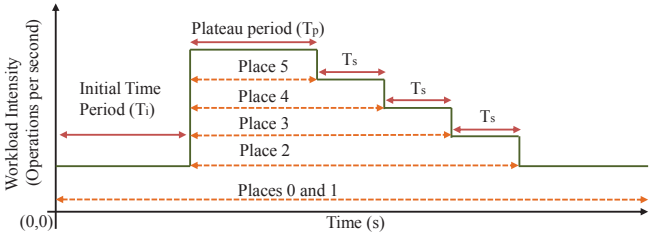


Figure 5. Model of the workload spike.

A sample X10 code fragment that implements part of the workload spike generation is shown in Figure 6. The main thread waits till the initial time T_i elapses to initiate the workload spike. The code within the `finish` statement conducts the workload spike generation. The value `spkDuration` variable corresponds to the T_s shown in Figure 5.

```
var spkDuration:Int=-1;          <- This variable corresponds to Ts of the workload spike

//Currently we create workload spikes only during the transaction phase.
if(dotransactions){
        spkDuration=Int.parseInt(props.getProperty("spikeduration","-1"));

        while((spkDuration != -1)&&(!getFlag())){   <- Wait till the Ti elapses
                System.sleep(100);//In milliseconds
        }
}

finish for (i in 0 .. (len -1))
{
        async{
                t(0) = threads.get(i) as __ClientThread;   <- Run each and every thread corresponding to
                t(0).run(spkDuration * 1000);                 the specified spike duration
        }
}
```

Figure 6. An example code fragment in XGDBench Titan client corresponding to the process of workload spike generation.

The `getFlag()` method is used to obtain a flag value kept in the shared file system. Place 0 instructs other places about the elapse of the initial time T_i by setting file flag on the shared file system. We used Network File System (NFS) as the file system for coordination between different places.

V. Evaluation of Workload Spike Generation

We evaluated workload spike generation performance of a graph data set generated by XGDBench's MAG graph generator with 3206 vertices and 18895 edges. We used Titan [3] distributed graph database server as the representative graph database server for this purpose.

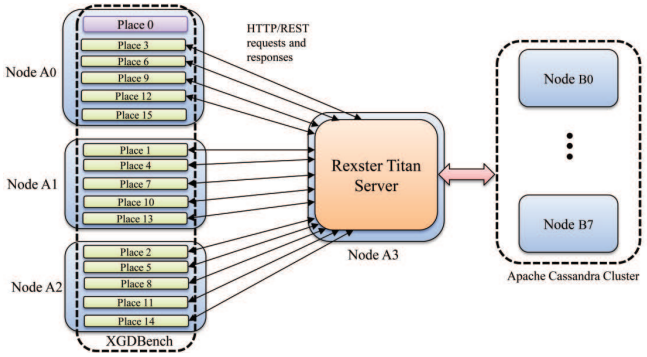


Figure 7. How XGDBench, Titan Rexster server, and Apache Cassandra are deployed in the experiment node cluster.

Titan server was exposed as a HTTP/REST server through REXSTER 2.1.0 [31]. Rexster was configured to use Apache Cassandra 1.1.6 backend. The Cassandra cluster we used consisted of eight nodes each with AMD Phenom^{TM}Processor 9850, 8GB RAM, 4 cores, 1600MHz 512KB L2 cache per core, 160GB hard drive. These nodes are denoted as nodes $B_i(i \in \{0...7\})$ in Figure 7. XGDBench client (on three nodes) and Titan server (on one node) were deployed on a cluster of four nodes (Nodes denoted as

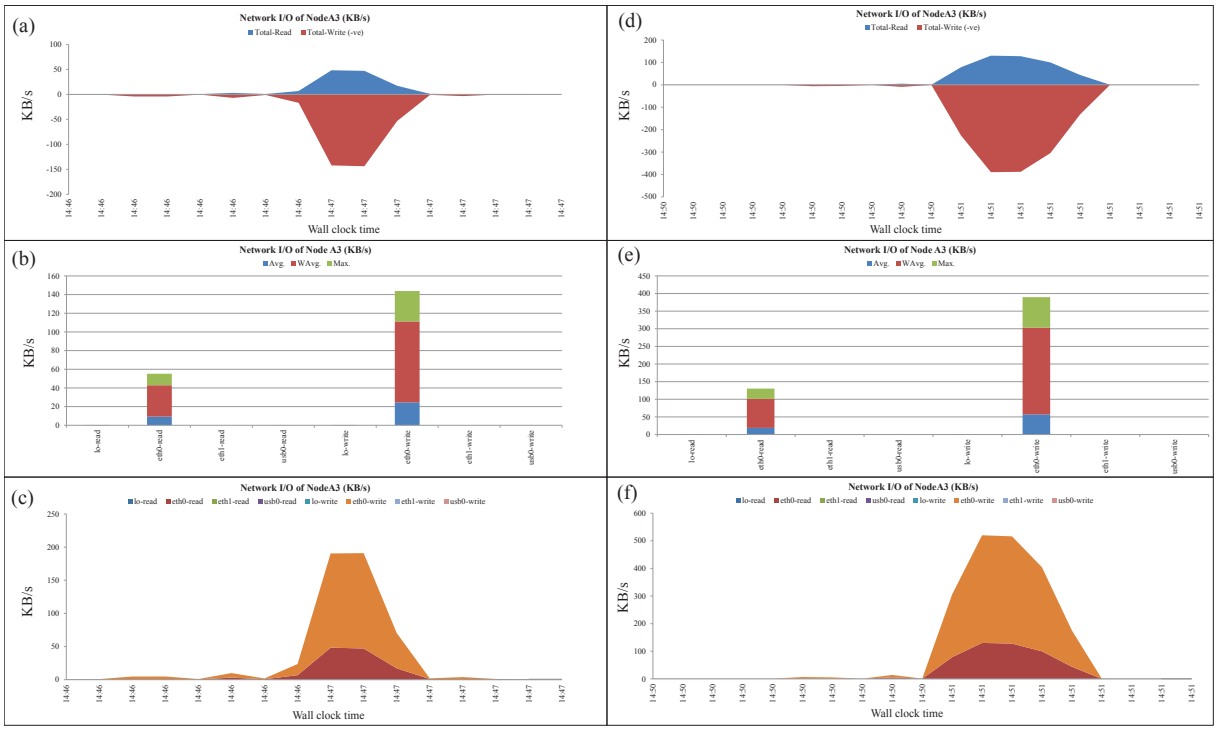


Figure 8. Network I/O of the node running Titan Rexster server during the experiment with sampling frequency of 5s.

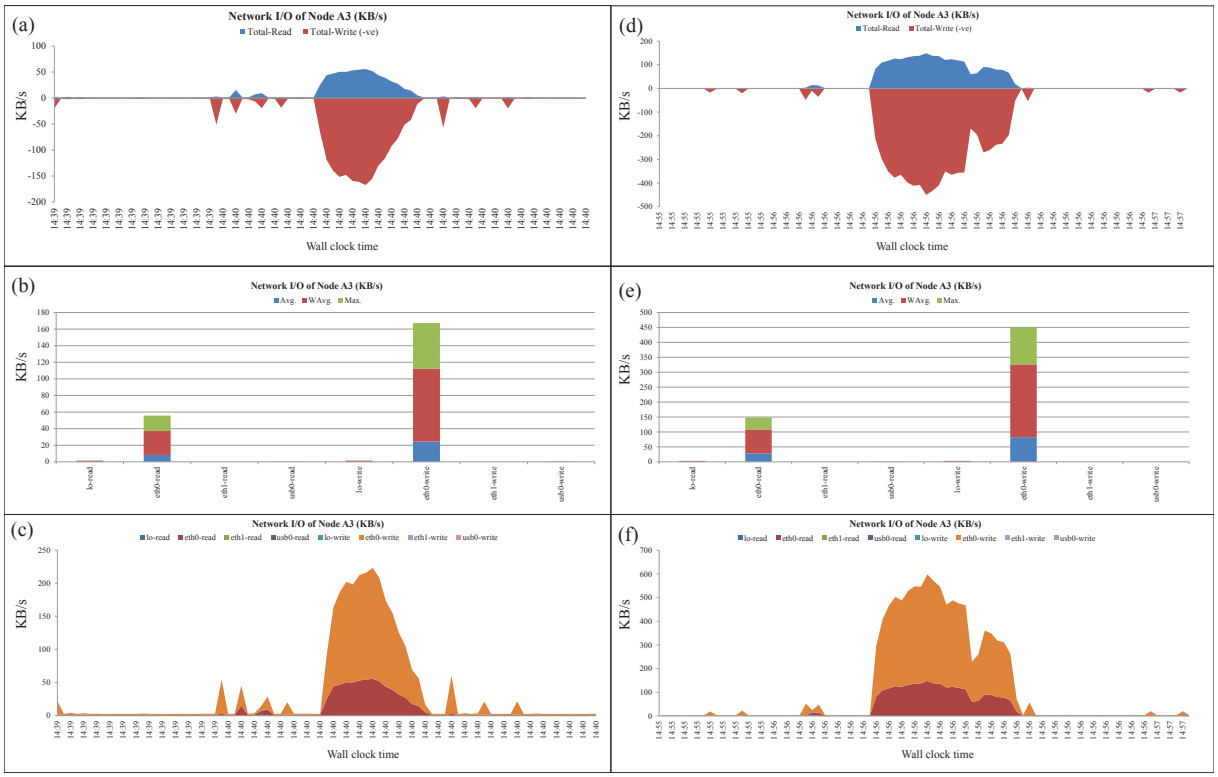


Figure 9. Network I/O of the node running Titan Rexster server during the experiment with sampling frequency of 1s.

$A_j (j \in \{0...3\})$ in Figure 7) each with Intel Core™i7-2600K CPU @ 3.40GHz, 8 cores, 16GB RAM, 8MB L2 cache, 200GB hard drive. All the nodes were installed with Linux CentOS (kernel 2.6.18), Oracle JDK 1.6.0_43, and X10 2.4.0. We used 4GB JVM heap size settings for Titan Rexster server. For XGDBench we used a heap of 8GB and a stack of 1GB.

We used workload C which is a read-only workload of XGDBench (Described in Table I) for this evaluation. We set up the plateau period (T_p) of the spike to 5 seconds and activities terminated their workload execution with in a time gap (i.e., spike decremental interval T_s) of 2 seconds. The non-spike activity count was set to two, hence at the beginning of the spikeful workload execution only two activities executed their workloads. We employed total 16 places during the experiment with each place running one activity. Hence 13 activities ignited the workload spike after the initial time period T_i (Activity running at place 0 did not run workloads hence it was kept for workload coordination). Figure 7 shows the arrangement of the places during the spikeful workload execution. Each and every place in Figure 7 can be considered as a real world analogy for an individual client executing queries over the Titan Rexster server simultaneously (i.e., this emulates clients with a factor of 15).

We used Nmon [27] which is a famous profiling tool to observe the behavior of the Titan Rexster server during the workload spike. The results are shown in Figures 8 and 9. The results on Figures 8 and 9 (a), (b), and (c) correspond to an experiment conducted with 8 places (Emulating a scenario of 7 clients executing a workload spike simultaneously). The sub figures (d), (e), and (f) of Figures 8 and 9 correspond to an experiment conducted with 16 places (Emulating a scenario of 15 clients executing a workload spike simultaneously). The difference between Figures 8 and 9 is that Figure 8 shows results sampled in the intervals of 5s while 9 shows results sampled in the intervals of 1s. The finer grained sampling on Figure 9 allows us to observe more details of the workload spike while coarser grained sampling in Figure 8 shows the abstract structure of the workload spike.

VI. Results and Discussion

From the results shown on Figures 8 and 9 we observed that XGDBench is able to generate realistic workload spikes for graph database benchmarking. Furthermore, by comparing the results shown in the left side and the right side of the Figures 8 and 9 we conclude that by varying the number of places we are able to control the magnitude of the workload spike. With use of Nmon we observed that most of the network I/O data from the Titan Rexster server was on write. This indicates that the response format of the Titan graph database server consumes more network bandwidth compared to the little bandwidth consumed by the queries issued by the XGDBench on the Titan Rexster server during the workload spike emulation.

Current evaluation of the workload generation was conducted on a graph data set with ≈19K edges. We have demonstrated the ability of XGDBench for generating much larger graph sizes in our previous work [12]. While we report the methodology for emulating a workload spike in this paper using XGDBench framework, we keep evaluation of the graph database's ability to withstand much larger workload spikes as a future work.

VII. Conclusion

XGDBench is a distributed, extensible graph database benchmarking framework aimed for benchmarking graph databases which store property graphs. In this paper we described a methodology for implementing workload spikes generation with XGDBench through a multiplication factor based workload spike emulation technique. We observed that XGDBench is capable of generating workload spikes through preliminary experiments conducted with Titan graph database server with a cluster of twelve compute nodes. We demonstrated the scalability of the benchmark by varying the number of X10 places. Increasing the number of places from 8 to 16 increased the magnitude of the workload spike by a factor of 2.7.

In future we hope to investigate more on analysis of real world workload traces of graph databases. We are working on implementing workload spikes emulation for other popular open source graph databases. We hope to experiment with larger graph data set sizes. Another future direction we hope to take this research is random evolution of property graphs which will enable creation of more efficient benchmarking sessions by eliminating the need for storing the entire benchmarking graph data set in memory.

VIII. Acknowledgments

This research was supported by the Japan Science and Technology Agency's CREST project titled "Development of System Software Technologies for post-Peta Scale High Performance Computing".

References

[1] R. Angles, A. Prat-Pérez, D. Dominguez-Sal, and J.-L. Larriba-Pey. Benchmarking database systems for social network applications. GRADES '13, pages 1–7. ACM, 2013.

[2] T. G. Armstrong, V. Ponnekanti, D. Borthakur, and M. Callaghan. Linkbench: a database benchmark based on the facebook social graph. SIGMOD '13, pages 1185–1196, New York, NY, USA, 2013. ACM.

[3] Aurelius. Titan: Distributed graph database. URL: http://thinkaurelius.github.com/titan/, 2012.

[4] D. A. Bader, J. Feo, J. Gilbert, J. Kepner, D. Koester, E. Loh, K. Madduri, B. Mann, T. Meuse, and E. Robinson. Hpc scalable graph analysis benchmark. Feb 2009.

[5] P. Bodik, A. Fox, M. J. Franklin, M. I. Jordan, and D. A. Patterson. Characterizing, modeling, and generating workload spikes for stateful services. In *Proceedings of the 1st ACM symposium on Cloud computing*, SoCC '10, pages 241–252. ACM, 2010.

[6] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: An object-oriented approach to non-uniform cluster computing. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '05, pages 519–538. ACM, 2005.

[7] M. Ciglan, A. Averbuch, and L. Hluchy. Benchmarking traversal operations over graph databases. *Data Engineering Workshops, 22nd International Conference on*, 0:186–189, 2012.

[8] P. E. Compeau, P. A. Pevzner, and G. Tesler. How to apply de bruijn graphs to genome assembly. *Nature biotechnology*, 29(11):987–991, 2011.

[9] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni. Pnuts: Yahoo!'s hosted data serving platform. *Proc. VLDB Endow.*, 1(2):1277–1288, Aug. 2008.

[10] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM symposium on Cloud computing*, SoCC '10, pages 143–154, New York, NY, USA, 2010. ACM.

[11] M. Dayarathna and T. Suzumura. XGDBench: A benchmarking platform for graph stores in exascale clouds. In *Cloud Computing Technology and Science (CloudCom), 2012 IEEE 4th International Conference on*, pages 363–370, 2012.

[12] M. Dayarathna and T. Suzumura. Graph database benchmarking on cloud environments with XGDBench. *Automated Software Engineering*, pages 1–25, 2013.

[13] D. De Abreu, A. Flores, G. Palma, V. Pestana, J. Pinero, J. Queipo, J. Sánchez, and M.-E. Vidal. Choosing between graph databases and RDF engines for consuming and mining linked data. 1034, 2013.

[14] D. Dominguez-Sal, P. Urbón-Bayes, A. Giménez-Vañó, S. Gómez-Villamor, N. Martínez-Bazán, and J. L. Larriba-Pey. Survey of graph database performance on the hpc scalable graph analysis benchmark. WAIM'10, pages 37–48. Springer-Verlag, 2010.

[15] A. Ghazal, T. Rabl, M. Hu, F. Raab, M. Poess, A. Crolotte, and H.-A. Jacobsen. BigBench: Towards an Industry Standard Benchmark for Big Data Analytics. SIGMOD '13, pages 1197–1208. ACM, 2013.

[16] M. Graves, E. Bergeman, and C. Lawrence. Graph database systems. *Engineering in Medicine and Biology Magazine, IEEE*, 14(6):737 –745, nov/dec 1995.

[17] Gremlin. Gremlin. URL: https://github.com/tinkerpop/gremlin/wiki/, 2013.

[18] Y. Guo, A. L. Varbanescu, A. Iosup, C. Martella, and T. L. Willke. Benchmarking graph-processing platforms: A vision. In *Proceedings of the 5th ACM/SPEC International Conference on Performance Engineering*, ICPE '14, pages 289–292. ACM, 2014.

[19] L.-Y. Ho, J.-J. Wu, and P. Liu. Distributed graph database for large-scale social computing. In *Cloud Computing (CLOUD), 2012 IEEE 5th International Conference on*, pages 455 –462, june 2012.

[20] L.-Y. Ho, J.-J. Wu, and P. Liu. Data replication for distributed graph processing. In *Cloud Computing (CLOUD), 2013 IEEE Sixth International Conference on*, pages 319–326, June 2013.

[21] F. Holzschuher and R. Peinl. Performance of graph query languages: comparison of cypher, gremlin and native access in neo4j. In *Proceedings of the Joint EDBT/ICDT 2013 Workshops*, EDBT '13, pages 195–204, 2013.

[22] K. Lee, L. Liu, Y. Tang, Q. Zhang, and Y. Zhou. Efficient and customizable data partitioning framework for distributed big RDF data processing in the cloud. In *Cloud Computing (CLOUD), 2013 IEEE Sixth International Conference on*, pages 327–334, June 2013.

[23] N. Martinez-Bazan, S. Gomez-Villamor, and F. Escale-Claveras. DEX: A high-performance graph database management system. In *Data Engineering Workshops (ICDEW), 2011 IEEE 27th International Conference on*, pages 124–127, 2011.

[24] R. C. McColl, D. Ediger, J. Poovey, D. Campbell, and D. A. Bader. A performance evaluation of open source graph databases. PPAA '14, pages 11–18, 2014.

[25] K. Myunghwan and J. Leskovec. Multiplicative attribute graph model of real-world networks. *Internet Mathematics*, 8(1-2):113–160, 2012.

[26] M. Nisar, A. Fard, and J. Miller. Techniques for graph analytics on big data. In *Big Data (BigData Congress), 2013 IEEE International Congress on*, pages 255–262, June 2013.

[27] Nmon. *nmon for Linux*. June 2011. URL: http://nmon.sourceforge.net.

[28] Orient Technologies LTD. Orientdb. URL: http://www.orientdb.org/, 2014.

[29] J. Partner, A. Vukotic, and N. Watt. In *Neo4j in Action*. Manning Publications Co., 2012.

[30] B. Shao, H. Wang, and Y. Xiao. Managing and mining large graphs: systems and implementations. SIGMOD '12, pages 589–592. ACM, 2012.

[31] Tinkerpop. Rexster. URL: https://github.com/tinkerpop/rexster/wiki, 2013.

[32] TPC. *TPC - Benchmarks*. October 2013. URL: http://www.tpc.org/information/benchmarks.asp.

[33] Yahoo! Webscope. *ydata-sherpa-database-platform-system-measurements-v1_0*. April 2014. URL: http://research.yahoo.com/Academic_Relations.