2014 5th International Workshop on Data-Intensive Computing in the Clouds

Towards Scalable Distributed Graph Database Engine for Hybrid Clouds

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*—Large graph data management and mining in clouds has become an important issue in recent times. We propose Acacia which is a distributed graph database engine for scalable handling of such large graph data. Acacia operates between the boundaries of private and public clouds. Acacia partitions and stores the graph data in the private cloud during its initial deployment. Acacia bursts into the public cloud when the resources of the private cloud are insufficient to maintain its service-level agreements. We implement Acacia using X10 programming language. We describe how Top-K PageRank has been implemented in Acacia. We report preliminary experiment results conducted with Acacia on a small compute cluster. Acacia is able to upload 69 million edges LiveJournal social network data set in about 10 minutes. Furthermore, Acacia calculates the average out degree of vertices of LiveJournal graph in 2 minutes. These results indicate Acacias potential for handling large graphs.**

***Index Terms*—Graph databases; Database management; Distributed databases; Graph theory; Cloud computing;**

I. Introduction

In recent times graph data management and mining has become a key question faced by cloud computing community. Variety of applications in the areas of social network analysis [20], bioinformatics [6], cheminformatics [38], semantic web [17], geographic information systems [42], etc. have appeared that deal with large graph data in cloud environments. Currently there are two broad categories of graph processing application types: first, offline graph analytics which demand high throughput and second, online query processing that requires low latency [34]. The focus of this paper is on *Graph Database Systems* (Graph databases) which answer the latter question. Graph databases are a type of NoSQL data stores which follow networked data model for storing graph data. Neo4j [30], OrientDB [32], DEX [27], etc. are examples for some popular graph databases. Many of the current graph databases are intended for running only on single node. Therefore, they face multiple issues such as system performance and resource availability when handling large scale graphs [13][14]. However, there are few distributed graph database systems such as Trinity [33], Titan [4], G* [25], etc. currently exist to solve these issues. With the increasing demand for hosting graph related applications and services in the clouds, several graph database vendors have started their operations in public clouds. NuvolaBase [31], Dydra [16], CloudGraph [9], etc. are some examples for such services that operate on SaaS model.

Public and private clouds represent the two established ends of the cloud computing spectrum based on the ownership and efficiency of shared resources. Public cloud is a cloud computing model where the computing resources are made available for the general public over the Internet. Private cloud on the other hand is a cloud computing model where the computing resources are privately held and maintained within a corporate network. A third deployment model called hybrid cloud that incorporates characteristics of public and private clouds is currently emerging. Hybrid clouds provide the ability of extending an organization's private cloud infrastructure by providing elastic cloud computing resources in a cost-effective manner. There have been previous works on hybrid operation of different data analysis applications in the domains of stream computing [24][22], MapReduce, etc. In this work we study about the problem of graph database operation in hybrid cloud environments.

In achieving our aim, we create an efficient and effective operation model for hybrid cloud graph databases. We implement this method on a hybrid cloud graph database engine that we developed called Acacia. We developed Acacia system in X10 [7] which is an Asynchronous Partitioned Global Address Space (APGAS) language for programming future exacscale systems. Acacia's data loading phase is supported by Hadoop and Hadoop Distributed File System (HDFS) [37]. With experiments conducted on multiple graph data sets on a small scale compute cluster, we show the scalability of Acacia. By running experiments on this initial Acacia system we have observed multiple issues in scalability of the system which we list down in the discussion section. LiveJournal social network graph data set [26] with 69 million edges ($\approx$ 1GB) can be uploaded to the Acacia system in 10 minutes on four servers. We describe the details of why such performance behavior is occurring in our current system under the discussion section. Specifically we make the following contributions in this paper,

- *Graph data management in hybrid clouds* - We create a model of graph data distribution between the public and private clouds in the context of graph databases.
- *Distributed graph database engine* - We describe the architecture of a distributed graph database engine. We

978-1-4799-7034-6/14 $31.00 © 2014 IEEE
DOI 10.1109/DataCloud.2014.9

implement the proposed system using X10.
- *Evaluation* - We provide preliminary evaluation results of the Acacia system with experiments conducted using real data sets.

The paper is structured as follows. We provide related work in Section II and describe the system design of Acacia in Section III. Next, the implementation details of Acacia is given in Section IV. The hybrid cloud scheduling algorithm of Acacia is described in Section IV-B. How Top-K PageRank calculation on Acacia is implemented is described in Section IV-C. We describe the evaluation of Acacia in Section V. We discuss the results in Section VI, and conclude the paper in Section VII.

II. Related Work

Graph data storage and management has been a rigorously addressed issue by database, complex network analysis, and cloud computing communities in recent times. Many of these efforts have materialized in to solid implementations of graph database servers such as Neo4j [30], OrientDB [32], AllegroGraph [1], DEX [27], etc. But all of these run only on single workstations which limits their ability to handle large data sets that cannot be stored in a single system due to hardware limitations.

There are multiple recent work on development of distributed graph database systems. Trinity is a distributed graph engine for analysis of large scale graphs which incorporates graph storage functionality [33][34]. Trinity's in-memory distributed database supports both online query processing and offline analytics on graphs. However, our focus is more on graph storage aspects and we use secondary storage as well in Acacia. One of the closely related recent works to ours in graph data management area has been presented by Li-Yung Ho *et al.* [20]. They describe a distributed graph database architecture for large social computing. Their focus is on creation of a distributed graph database server that is completely run on either public or private clouds which is different from the focus of our work. Similar to us, they use Neo4j as their storage back-end. However, their underlying communication has been implemented using MPI while Acacia uses the socket back-end of Managed X10/Java sockets for communication. G* is another distributed graph database system for managing graph data in parallel [25]. The aforementioned graph databases operate as distributed database engines. However, no known studies exists for their capabilities in operating in hybrid cloud environments.

Graph partitioning is a challenging issue that needs to be addressed in distributed graph data management. Barguñó *et al.* described the architecture of ParallelGDB which is a graph database that uses random graph partitioning to avoid complex partitioning methods on graph topology [5]. Different from them we utilize METIS [23] graph partitioner to reduce the communication overhead between compute nodes. Mondal *et al.* proposed aggressive replication of the nodes in a graph for supporting low-latency querying and investigates on three novel techniques to minimize the communication bandwidth and the storage requirements [29]. Similar to Acacia, their proposal is to create a data management engine. However, they use CouchDB [3] for this purpose. Chen *et al.* developed a novel graph partitioning framework to improve the network performance of graph partitioning itself, partitioned graph storage, and vertex oriented graph processing [8]. Different from them, we use a popular graph partitioning library called METIS [23] to develop Acacia's graph partitioner.

There is substantial amount of research literature [19][22][24] devoted for stream processing in hybrid cloud environments. Gulisano *et al.* combined elasticity with dynamic load balancing to minimize the computational resources utilization [19]. Kleiminger *et al.* presented a combined stream processing system that adaptively balances workload between a dedicated local stream processor and a cloud stream processor [24]. Ishii *et al.* described a method and an architecture to use Virtual Machines (VMs) in a cloud environment elastically to satisfy real-time processing requirements [22]. All these works are closely related to our work. However, they are conducted on stream processing applications.

Multiple work have been conducted on deployment of databases in clouds. Yan-hua *et al.* described a cloud database route scheduling algorithm using a combination of the genetic algorithm and ant colony algorithm [40]. Minhas *et al.* described a highly available database management system that is based on VM replication, that leverages the capabilities of the underlying virtualization layer [28]. Different from all the aforementioned works our focus is on graph databases. TIRAMOLA is a cloud-enabled framework for automatic provisioning of elastic resources on any NoSQL platform [2]. They apply this framework on top of a fully distributed RDF store back-end by an elastic NoSQL database. However, our current Acacia system is not aimed for storing RDF graphs.

III. System Design

Acacia is a distributed graph database system designed to operate in hybrid cloud settings. In this section we describe the system architecture and the assumptions we made.

An overview of the system design of Acacia is shown in Figure 1. Acacia is designed based on Master-Worker pattern which provides us more control over the system's execution compared to alternative approaches such as P2P in the medium scale deployments (e.g., less than 100 workers). Acacia is designed to operate on a partitioned distributed graph. While graph partitioning requires sophisticated partitioning algorithms and imposes considerable processing overhead, it provides us the benefit of reducing the costly inter node communication happening among the nodes in both public and private clusters. Furthermore, Acacia is designed to operate in a hybrid cloud environment, where in general the system operates in a private cloud and bursts into the public cloud when the system runs on lower resources to maintain the system's integrity and Service Level Agreement (SLA). Note that Acacia is designed to maintain two SLA values. First, it needs to ensure low response latency. Second, it needs to

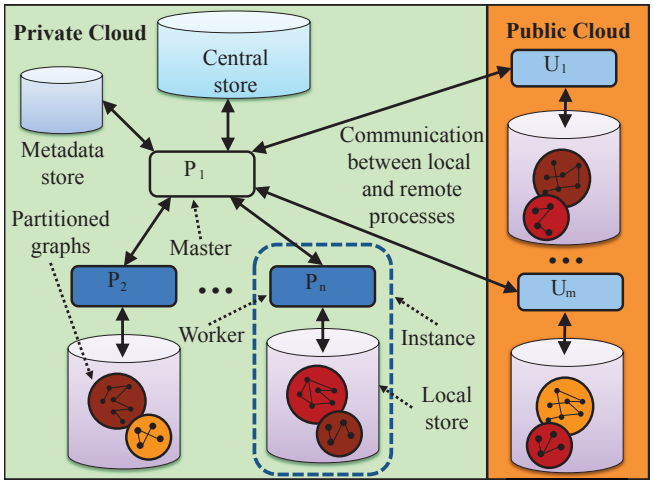

Fig. 1. Overview of Acacia

use minimum local disk storage based on the storage capacity allocation. In this paper we describe how the second feature is implemented in Acacia system.

Acacia comprises of a set of distributed processes and their associated local data stores. Each local data store has an assigned storage quota. Note that we assume each compute node imposes a limitation on its local data storage. When a new graph is added, Acacia first partitions the graph into number of subgraphs which is equal to the number of workers that it runs. Initially, the partitioned graphs are distributed to *Workers* (A worker is a standalone, non-distributed graph store) based on their available capacity. Acacia keeps on monitoring the Local store capacities as well as the CPU utilization of each node on the local cluster. When it finds the utilization violates the SLA values agreed before the execution, it considers of migrating portion of sub graphs into public cloud. Acacia calculates the required number of nodes to be allocated in the public cloud. Then Acacia starts the VMs on public cloud and instantiates new *Remote Workers*. As soon as the Public VMs are available, Acacia starts migrating the selected graph data. Note that in the current implementation we assume the storage in the public cloud is a persistent storage where the data stored in the public cloud do not get deleted even if the VMs in the public cloud are shutdown. When the local graph data storage quota is increased (due to addition of new storage or gaining of storage space due to deletion of graph data sets) Acacia will reorganize its graph data distribution which will enable shutting down the VMs run in the public cloud and releasing their resources.

If a graph can be partitioned with less minimum-cut between its partitions, it enables better graph query execution performance because such partitioning reduces communication between different processes on the graph database service. On the other hand we need to partition the graph data as evenly as possible among the nodes (Note that in this paper we use the term node for a server in the cloud environment while vertex is used to represent an entity in a graph.) so that the workload can be balanced among the nodes. In the current version of Acacia we allow one compute node to run only one worker.

IV. Implementation

In this section first we provide a necessarily brief introduction to X10 and then we provide the details on how the current Acacia architecture has been realized. Furthermore, we describe how the hybrid cloud scheduling algorithm and Top-K PageRank algorithm implementations on Acacia system have been conducted.

A. System Internals

In order to cope with the heterogeneity of cloud systems we have chosen to develop Acacia system with X10 programming language with use of X10's Managed backend. 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 [21][7]. The main role of X10 is to simplify the programming model in a way that it leads to increase in programmer productivity for future systems such as Extreme-scale computing systems [15]. X10 is a strongly typed, object-oriented language which emphasizes static type-checking and static expression of program invariants. The latest major release of X10 is X10 2.5 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 Acacia because it provides seamless interoperability with the existing Java libraries. We leverage the notion of *places*, language constructs for asynchronous execution (i.e., `async`), `GlobalReferences`, etc. available in X10 when developing the Acacia system. Furthermore, we can leverage the GPU integration features available in X10 for developing computation intensive graph algorithms. Moreover, X10's fault tolerant program execution features [10] can be leveraged to implement fault tolerance functionality on Acacia system. This is especially useful when improving Acacia to operate robustly in an error-prone environment.

The system architecture of Acacia is shown in Figure 2. Current version of Acacia accepts graphs stored as edgelists. We use METIS graph partitioner [23] to implement the graph partitioning functionality of Acacia. METIS partitions an unstructured graph into p number of parts specified by a user either using multilevel recursive bisection or the multilevel p-way partitioning paradigms. METIS provides high quality graph partitions with less edge cut values between subgraphs which allows for reducing the requirement of excessive communication between compute nodes. Graph partitioning allows us to do computational load balancing by running a graph processing algorithm concurrently on multiple workers. We implemented graph partitioning as a sequence of MapReduce jobs which get executed on Hadoop with support of METIS.

Since METIS accepts a specific type of file format we had to implement format conversion as MapReduce jobs. The partitioned graphs' edges are then stored on local Neo4j

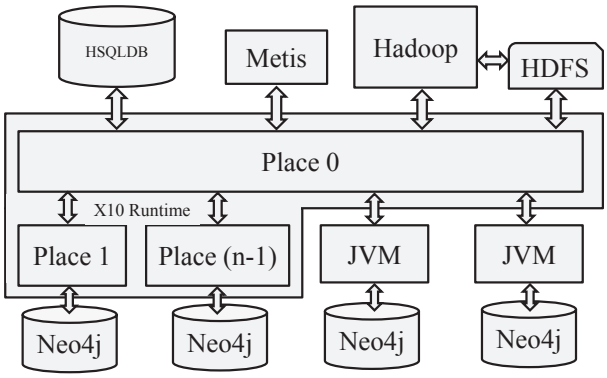

Fig. 2. Acacia System Architecture

instances that are run on each node on private/public cluster. The edges that lie on two partitions as well as unconnected vertices are stored on the central node. Note that, we run Hadoop and HDFS only in private cloud. Acacia keeps a metadata store of all operational information on a relational database and we use HSQLDB [36] for this purpose. On the public cloud the Remote Instances are represented by Java Virtual Machine (JVM) processes. Acacia deploys remote JVM instances when required. The components of Acacia system is shown in Figure 3.

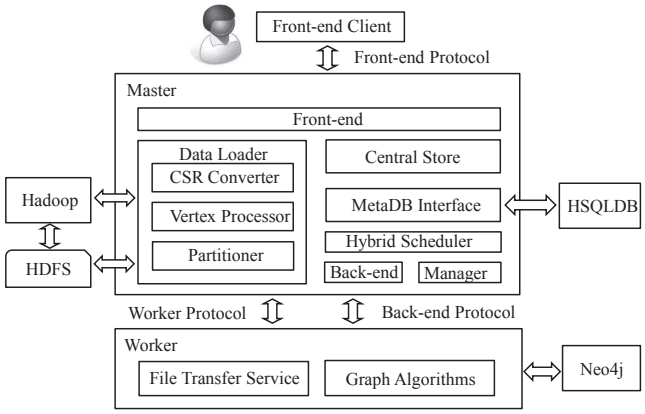

Fig. 3. Components of Acacia

There are two key components of Acacia system: Master and Worker. Front-end is the command-line user interface. Users issue custom commands to Acacia using the Front-end protocol. There are Front-end commands to list the system statistics, to upload/delete graphs, to get system statistics, commands to run graph algorithms, etc.

Master hosts a data loader component which converts a plain edge list file to METIS compatible file, partitions using METIS, and distributes the partitions among the workers and the central store. This process happens within the CSR Converter, Vertex Processor, and Partitioner components of the Data Loader of Acacia master with the support from Hadoop and HDFS. It should be noted that Hadoop and HDFS are used only during the data loading phase of Acacia. The normal operation (i.e., transaction phase) of Acacia (including graph querying) do not depend on Hadoop or HDFS. MetaDB Interface manages the standalone HSQLDB based meta data store while the Central Store component manages a fleet of HSQLDB embedded databases which are used as Acacia's central storage. In the current version of Acacia each of the embedded HSQLDB instance stores edges which cut across different partitions. There are two protocols that manage the communication happening between the masters and the workers. The Worker Protocol is used for the communication initiated from the master to worker. The communication initiated by workers to master is handled by Back-end protocol.

When a graph algorithm is run, the master instructs the workers to execute the graph algorithm on the specified data set in parallel. In current version of Acacia, all the graph algorithms are coded in as a hybrid of Java and X10 on the worker (See Figure 3). It should be noted that although the local graph storage is a Neo4j embedded server, graph algorithms are coded separately from Neo4j's own APIs which enables us to use heterogeneous hardware such as GPUs for optimizing our graph algorithm implementations.

B. Hybrid Cloud Scheduling Algorithm

We model the operation of Acacia to a scenario of solving the *Bin Packing* problem [35]. The set of x graphs stored in Acacia can be represented by $(G = G_1, G_2, ..., G_x)$ each of which is partitioned into r different pieces ($r \geq 0$). There are n compute nodes in the private cloud (denoted as B_i). Note that in the current version each node runs only one worker, therefore P_i shown in Figure 1 corresponds to the B_i which represents a bin in the private cloud. We consider each compute node in the private cloud as a bin (B_i) of different capacities $C_1, ..., C_n$. The size of the graph data stored in a particular moment in each bin is represented as $c_1, ..., c_n$. Furthermore, we assume that we have m set of compute nodes in public cloud each (U_j) of different sizes $D_1, ..., D_m$. Then the problem of Acacia bin packing can be stated as "store all the subgraph partitions of Acacia in an even manner". The occupied size of the bin B_i (c_i) at a particular moment is chosen to be of s times larger than the maximum amount of subgraphs' data size that can be stored in it. This measure is taken to ensure a sufficient amount of space is left in the bin to store intermediate results and data. Note that large social network graph data sets may produce highly skewed partitions. However, still the proposed approach can be utilized in such scenarios since it depends just on the disk capacity C_i allocated for each bin.

Since the bin packing problems are NP-Complete, we use a heuristic to solve the problem. Once we create a set of partitioned graph data with using Hadoop on HDFS, we sort the sub graphs based on their sizes (i.e., disk space utilization). Furthermore, we sort the set of available bins in the private cloud based on the available free space at that particular time $(C_1 - c_1), ..., (C_1 - c_n)$. Then we allocate the largest size

Algorithm 1: Subgraph Placement Algorithm

Input : Set of partitioned subgraphs (g), set of bins in private cloud (B), set of bins in public cloud (U), storage multiplier (s)
Output : A list of graph partition ids and their corresponding bin id (resultKV)
Description :

```
1:  resultKV ← {}
2:  for all gi in g do
3:      sortedB ← sortDescending(B)
4:      if (sortedB.top().size() * s) > gi.size() then
5:          sortedB.top().assign(gi)
6:          resultKV.append(sortedB.top(), gi)
7:      else
8:          sortedU ← sortDescending(U)
9:          if (sortedU.top().size() * s) > gi.size() then
10:             sortedU.top().assign(gi)
11:             resultKV.append(sortedU.top(), gi)
12:         else
13:             uNew ← allocateNewPublicVM()
14:             U.add(uNew)
15:             go to 3
16:         end if
17:     end if
18: end for
19: return resultKV
```

subgraph to the bin having the largest free space. After the assignment we sort the set of bins again based on their free capacity. Next, we assign the second largest sub graph to the bin with the largest free space. We continue this procedure until all the subgraphs get allocated to bins. If we find that the available free space $(C_i - c_i)$ is insufficient to store a subgraph, we allocate a new node in the public cloud and use it along with the remaining set of bins in the private cloud to run the scheduling algorithm. This subgraph placement algorithm for hybrid cloud operation of Acacia is shown in Algorithm 1. When a graph gets deleted from Acacia, we evaluate the feasibility of releasing a node allocated in the public cloud by migrating its data to private cloud following the same technique.

C. Top-K PageRank Calculation Algorithm

Acacia is designed to support multiple different graph processing algorithms. In the current version of Acacia we have implemented facility for calculating Top-K PageRank of a graph. Since the graph data set is partitioned across multiple nodes it is not possible to directly apply PageRank algorithm. Instead we developed an approximate PageRank calculation algorithm (ApproxRank) following the technique described by Wu *et al.* [39]. How Acacia's PageRank algorithm is abstracting a local graph from the global graph is illustrated in Figure 4.

In ApproxRank, the PageRank calculation happens at each and every subgraph of a global graph concurrently. Each instance of the algorithm running at each worker treats the local graph g_l it handles separate from the rest of the graphs. A summarized version of the graph is created by treating the local graph g_l separate from the rest of the graph g_e (See Figure 4(b)). Furthermore, in ApproxRank since the PageRank

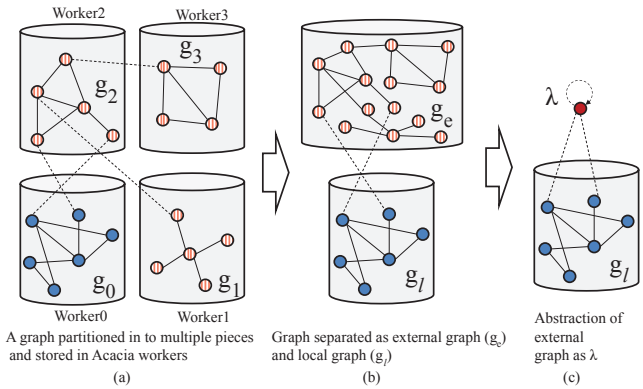

Fig. 4. Summarization of a graph stored in Acacia as local and global components.

scores of g_e is not known by the worker who executes the PageRank algorithm locally, the graph g_e is treated as an abstraction called λ (external vertex). The local graph g_l with λ is called the *extended local graph*. Then the PageRank vector R_{approx} can be defined as,

$$R_{approx} = \epsilon A^T_{approx} R_{approx} + (1-\epsilon) P_{ideal}, \quad (1)$$

where $(1-\epsilon)$ is the probability of a web surfer jumping to a page (We used $\epsilon = 0.85$ in our implementation [39]). A_{approx} is the transition matrix and P_{ideal} is the personalization vector. The values of the four quadrants of A_{approx} is calculated considering the edges within the local graph, edges from the local graph to λ, edges from λ to the local graph, and edges from λ to λ.

When running the Top-K PageRank on a graph, first Acacia runs ApproxRank algorithm on each and every worker which stores the subgraphs of the target graph. During this process workers communicate with each other as well as with the master. This is because the workers need in/out degree distributions for external graph (g_e) which is calculated at the master. Once PageRank scores were calculated by each and every worker those are aggregated at the master. Since user requests for only top k number of PageRank scores and their corresponding vertices, each worker need to send its top k number of PageRank scores and their corresponding vertices. The results sent by all the workers are used to identify top k number of PageRank scores for the entire graph. A summarized version of the Top-K PageRank algorithm of Acacia is shown in Algorithm 2.

V. Evaluation

In this paper we conduct preliminary level experiments to evaluate scalability of Acacia system during its main two phases of operation: data loading phase and transaction phase.

A. Experiment Setup

We used four compute nodes during the experiments described in this paper. Each node of the evaluation compute cluster had Intel ™Xeon™CPU @ 2.93GHz, 12 cores (24

Algorithm 2: Top-K PageRank Algorithm

Input : Local Graph (g_l), entire graph (G), k
Output : A vector of Top-K PageRank scores
Description :

1: Add external node λ to g_l
2: Create edges associated with λ and get g_e
3: Assign values to P_{ideal} and A_{approx} at each and every worker
4: Perform a random walk on the extended local graph following Equation 1 on each and every worker.
5: Aggregate the results at the master and return the Top-K values

hardware threads), 48GB RAM, 12MB L2 cache, 250GB hard disk drive. All the nodes were installed with Linux CentOS (kernel 2.6.18). We used X10-2.4.3 version, Oracle JDK 1.6.0_43, HSQLDB 2.2.9, Hadoop 1.2.0, and METIS version 5.0.2. X10 was configured to use 8GB heap size. All the nodes were connected to a 1 Gigabit Ethernet. We used five graph data sets of different social networks which are listed in Table I (Note that vertex and edge counts are shown in millions (M)). Four of the graphs (G_1, G_2, G_3, and G_5) were obtained from ASU Social Computing Data Repository [41] while G_4 was from SNAP repository [26]. All the compute nodes ran a worker each while the master was ran on the first node (i.e., node with ID=0).

TABLE I
GRAPH DATA SETS.

ID	Data set name	Vertices	Edges	File size
G_1	Hyves	1.40M	2.78M	36MB
G_2	Flickr	0.08M	5.90M	65MB
G_3	Flixster	2.52M	9.20M	112MB
G_4	LiveJournal	4.85M	69.00M	965MB
G_5	Twitter11M	11.32M	85.33M	1.23GB

In the first half of the evaluations we measured the elapsed time for loading multiple different graph data sets. In the second half of the experiments we ran average degree distribution calculation algorithm on the loaded graph data sets. Note that although we described the hybrid cloud execution algorithm of Acacia in Section IV-B we keep the evaluation of this feature as a future work.

B. Scalability of Data Uploading Phase

Data uploading is the task of reading a graph stored in some file format (such as text) and storing it in the internal storage of the graph database system. Unlike high performance graph libraries [18][12][11] which load graph data into memory, Acacia incurs significant overhead when uploading the graph data into the system because of the involvement of secondary storage access. Bulk uploading has been the norm for loading high volume data sets to relational databases while some graph databases such as Neo4j support such bulk uploading. Through use of such parallel and bulk loading techniques we were able to obtain significant performance gain during the data loading phase. In this section we evaluate the elapsed time for loading graph data into Acacia system. The results for loading the graph data sets listed in Table I (from G_1 to G_4) is shown in Figure 5.

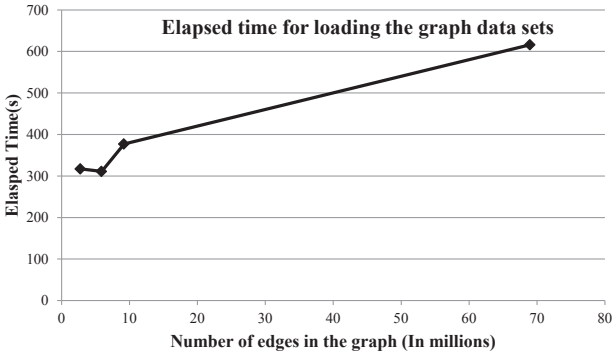

Fig. 5. Comparison of elapsed time for loading multiple graphs

It can be observed that the smallest graph (G_1) of 36MB loading took around 5 minutes and Flixster graph (G_2) of 112MB took 6 minutes, while the largest graph (G_4) which is 965MB size took about 10 minutes. This result indicates that the current Acacia system can upload large graphs in a fast manner despite the overhead of format conversion.

C. Scalability of Graph Statistics

Graph statistics such as number of vertices, number of edges, degree distribution provides insights about the structure of a graph data set. We ran average out degree calculation algorithm (which provides per vertex average outgoing degree) on the graphs listed in Table I to measure the scalability of such statistics calculation on Acacia system. The results are shown in Figure 6.

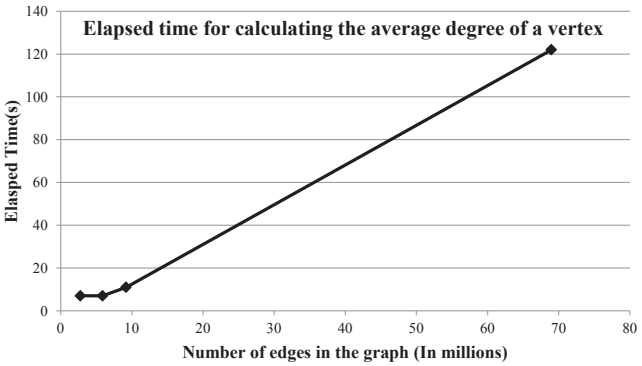

Fig. 6. Comparison of elapsed time for calculating the average out degree of multiple graphs

The average degree distribution calculation algorithm was run on each and every worker storing the intended graph concurrently. We observed interesting results for average degree calculation algorithm implemented on Acacia system.

VI. Discussion

From the experiments conducted with Acacia system it is clear that the system is able to handle medium size graph data sets efficiently. In the data loading process the bottleneck exists at the graph partitioning phase. METIS takes considerable amount of time for partitioning the graph data. Use of the parallel version of METIS (ParMETIS) also does not provide us a satisfactory solution. However, ParMETIS reduces the elapsed time by about 32% or more when creating more than eight graph partitions. A comparison of the elapsed time for partitioning Twitter11M (G_5 in Table I) data set is shown in Figure 7. Note that when running ParMETIS (version 4.0.3) we used mpich-3.0.4 and we used only one MPI place per partition. That means when there are 92 graph partitions created, we used 92 MPI places which is the maximum we could use since each compute node in the experiment environment had 24 hardware threads. We kept one thread on each node for OS and other back ground tasks. This is why there is no corresponding data point for 128 partitions for ParMETIS results shown in Figure 7.

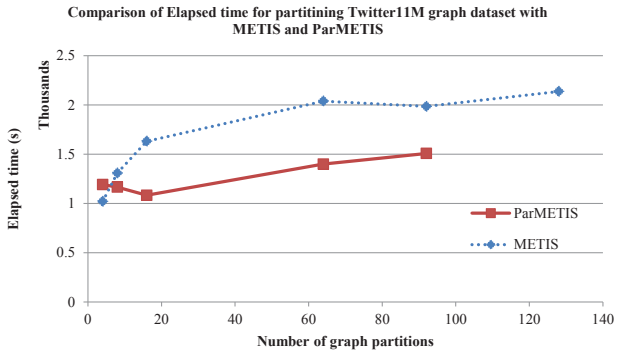

Fig. 7. Comparison of elapsed time for graph partitioning using METIS and ParMETIS

These results suggest that it is better to use ParMETIS with Acacia system when using more than eight graph partitions. Note that although the experiments described in this paper used METIS as the graph partitioner the number of graph partitions created did not exceed eight.

We have run Top-K PageRank algorithm on G_1 data set and obtained the top-4 PageRank values in 31 seconds (average of three consecutive runs).

The percentage of edges stored for each graph (from G_1 to G_4) on each worker as well as on the central store is shown in Figure 8. It can be observed that the graphs displayed different sizes of partitions. The Hyves graph had the largest percentage (43.29%) of edges stored in the central store while LiveJournal graph had only 11.30% of the edges stored in the central store.

In near future we plan to change to use ParMETIS as our graph partitioner as a solution for the graph partitioning scalability issue. Graphs do not present sufficient locality to allow balanced partitioning across multiple workers (as shown in Figure 8). Many of the graphs have considerable

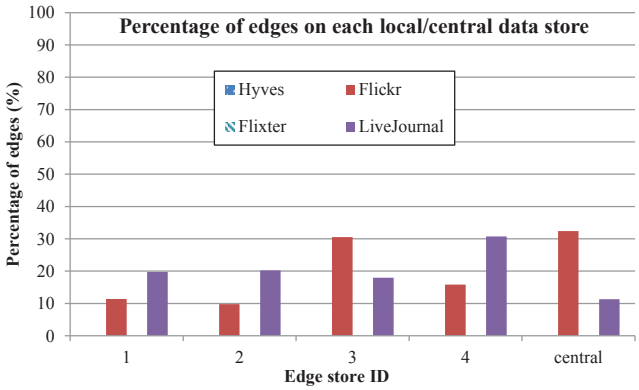

Fig. 8. Edge distribution percentages among different workers and the central store

amount of edges stored in the central store. We are working on a more efficient long term solution currently. While there exists challenges of subgraph extraction, migration, and query costs; we hope to optimize Acacia system to overcome these difficulties in future.

VII. Conclusion and Further Work

Scalable large scale graph data storage is a challenging issue which has attracted lot of attention in recent times. In this paper we described the system architecture and initial system implementation of Acacia which is a distributed large scale graph database engine intended for run in distributed cloud environments. Acacia is designed to store multiple different graph data sets partitioned across a distributed compute system that spans between the boundaries of private and public clouds. Acacia has been implemented with X10 (Managed X10) which is a new APGAS language for concurrent and distributed programming. Acacia system consists of multiple different types of operations such as data loading, querying graph statistics, and running complex graph queries such as Top-K PageRank, etc. LiveJournal social network data set of 69 million edges was uploaded to the Acacia system in about 10 minutes. From the experiments we conducted on medium size graphs we observed that graph partitioning is a challenging issue which consumes significant amount of time during data loading phase. This cannot be easily solved even using parallel graph partitioning software such as ParMETIS. We are currently working on a solution that reduces graph partitioning time. We hope to evaluate the trade-off between use of different other graph partitioning schemes such as hash partitioning, etc. with Acacia in future. We are also working on experimenting Acacia with large scale data sets (such as billion scale graphs) on large scale super computers. The current Acacia system is susceptible for single point of failure. We hope to implement fault tolerance features in Acacia system in future to solve this issue. We are planning to evaluate the hybrid cloud execution of Acacia using a public cloud environment such as Amazon EC2. We also hope to implement a graphical user interface for Acacia system in future.

ACKNOWLEDGMENT

This research was partly supported by Japan Science and Technology Agency's CREST Programs "Advanced Core Technologies for Big Data Integration" and "Development of System Software Technologies for post-Peta Scale High Performance Computing".

REFERENCES

[1] AllegroGraph. Allegrograph rdfstore web 3.0's database. URL: http://www.franz.com/agraph/allegrograph/, 2014.

[2] E. Angelou, N. Papailiou, I. Konstantinou, D. Tsoumakos, and N. Koziris. Automatic scaling of selective sparql joins using the tiramola system. In *Proceedings of the 4th International Workshop on Semantic Web Information Management*, SWIM '12, pages 1:1–1:8, New York, NY, USA, 2012. ACM.

[3] Apache. Apache couchdb. URL: http://couchdb.apache.org/, 2014.

[4] Aurelius. Titan: Distributed graph database. URL: http://thinkaurelius.github.io/titan/, 2014.

[5] L. Barguñó, V. Muntés-Mulero, D. Dominguez-Sal, and P. Valduriez. Parallelgdb: a parallel graph database based on cache specialization. In *Proceedings of the 15th Symposium on International Database Engineering & Applications*, IDEAS '11, pages 162–169, New York, NY, USA, 2011. ACM.

[6] F. Belleau, M.-A. Nolin, N. Tourigny, P. Rigault, and J. Morissette. Bio2rdf: Towards a mashup to build bioinformatics knowledge systems. *Journal of Biomedical Informatics*, 41(5):706 – 716, 2008.

[7] 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, New York, NY, USA, 2005. ACM.

[8] R. Chen, M. Yang, X. Weng, B. Choi, B. He, and X. Li. Improving large graph processing on partitioned graphs in the cloud. In *Proceedings of the Third ACM Symposium on Cloud Computing*, SoCC '12, pages 3:1–3:13, New York, NY, USA, 2012. ACM.

[9] CloudGraph. Cloudgraph .net graph database. URL: http://www.cloudgraph.com/, 2014.

[10] D. Cunningham, D. Grove, B. Herta, A. Iyengar, K. Kawachiya, H. Murata, V. Saraswat, M. Takeuchi, and O. Tardieu. Resilient x10: Efficient failure-aware programming. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '14, pages 67–80, New York, NY, USA, 2014. ACM.

[11] M. Dayarathna, C. Houngkaew, H. Ogata, and T. Suzumura. Scalable performance of scalegraph for large scale graph analysis. In *High Performance Computing (HiPC), 2012 19th International Conference on*, pages 1–9, Dec 2012.

[12] M. Dayarathna, C. Houngkaew, and T. Suzumura. Introducing scalegraph: An x10 library for billion scale graph analytics. In *Proceedings of the 2012 ACM SIGPLAN X10 Workshop*, X10 '12, pages 6:1–6:9, New York, NY, USA, 2012. ACM.

[13] 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, Dec 2012.

[14] M. Dayarathna and T. Suzumura. Graph database benchmarking on cloud environments with xgdbench. *Automated Software Engineering*, 21(4):509–533, 2014.

[15] J. Dongarra and et al. The international exascale software project roadmap. *International Journal of High Performance Computing Applications*, 25(1):3–60, 2011.

[16] Dydra. Dydra: Networks made friendly. URL: http://dydra.com/, 2014.

[17] A. Eberhart, P. Haase, D. Oberle, and V. Zacharias. Semantic technologies and cloud computing. In D. Fensel, editor, *Foundations for the Web of Information and Services*, pages 239–251. Springer Berlin Heidelberg, 2011.

[18] D. Gregor and A. Lumsdaine. Lifting sequential graph algorithms for distributed-memory parallel computation. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '05, pages 423–437, New York, NY, USA, 2005. ACM.

[19] V. Gulisano, R. Jimenez-Peris, M. Patio-Martinez, C. Soriente, and P. Valduriez. Streamcloud: An elastic and scalable data streaming system. *Parallel and Distributed Systems, IEEE Transactions on*, 23(12):2351 –2365, dec. 2012.

[20] 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.

[21] IBM. X10: Performance and productivity at scale. URL: http://x10-lang.org/, 2014.

[22] A. Ishii and T. Suzumura. Elastic stream computing with clouds. In *Cloud Computing (CLOUD), 2011 IEEE International Conference on*, pages 195 –202, july 2011.

[23] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J. Sci. Comput.*, 20(1):359–392, Dec. 1998.

[24] W. Kleiminger, E. Kalyvianaki, and P. Pietzuch. Balancing load in stream processing with the cloud. In *Data Engineering Workshops (ICDEW), 2011 IEEE 27th International Conference on*, pages 16 –21, april 2011.

[25] A. Labouseur, J. Birnbaum, J. Olsen, PaulW., S. Spillane, J. Vijayan, J.-H. Hwang, and W.-S. Han. The g* graph database: efficiently managing large distributed dynamic graphs. *Distributed and Parallel Databases*, pages 1–36, 2014.

[26] J. Leskovec, K. J. Lang, A. Dasgupta, and M. W. Mahoney. Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters. *CoRR*, abs/0810.1355, 2008.

[27] 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, april 2011.

[28] U. Minhas, S. Rajagopalan, B. Cully, A. Aboulnaga, K. Salem, and A. Warfield. Remusdb: transparent high availability for database systems. *The VLDB Journal*, 22:29–45, 2013.

[29] J. Mondal and A. Deshpande. Managing large dynamic graphs efficiently. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, pages 145–156, New York, NY, USA, 2012. ACM.

[30] Neo4j. Neo4j - the world's leading graph database. URL: http://www.neo4j.org/, 2014.

[31] NuvolaBase. Nuvolabase: cloudize your data - commercial support, training and services about orientdb. URL: http://www.nuvolabase.com/site/, 2014.

[32] NuvolaBase. Orientdb document-graph nosql database. URL: http://www.orientdb.org/, 2014.

[33] B. Shao, H. Wang, and Y. Li. Trinity: A distributed graph engine on a memory cloud. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 505–516, New York, NY, USA, 2013. ACM.

[34] B. Shao, H. Wang, and Y. Xiao. Managing and mining large graphs: systems and implementations. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, pages 589–592, New York, NY, USA, 2012. ACM.

[35] S. S. Skiena. *The Algorithm Design Manual*. Springer, 2 edition, 2008.

[36] The HSQL Development Group. Hsqldb. URL: http://hsqldb.org/, 2014.

[37] T. White. *Hadoop: The definitive guide*. " O'Reilly Media, Inc.", 2012.

[38] E. Willighagen, J. Alvarsson, A. Andersson, M. Eklund, S. Lampa, M. Lapins, O. Spjuth, and J. Wikberg. Linking the resource description framework to cheminformatics and proteochemometrics. *Journal of Biomedical Semantics*, 2:1–24, 2011.

[39] Y. Wu and L. Raschid. Approxrank: Estimating rank for a subgraph. In *Data Engineering, 2009. ICDE '09. IEEE 25th International Conference on*, pages 54–65, March 2009.

[40] Z. Yan-hua, F. Lei, and Y. Zhi. Optimization of cloud database route scheduling based on combination of genetic algorithm and ant colony algorithm. *Procedia Engineering*, 15(0):3341 – 3345, 2011.

[41] R. Zafarani and H. Liu. Social computing data repository at ASU, 2009.

[42] L. Zhou, R. Wang, C. Cui, and C. Xie. Gis application model based on cloud computing. In J. Lei, F. Wang, M. Li, and Y. Luo, editors, *Network Computing and Information Security*, volume 345 of *Communications in Computer and Information Science*, pages 130–136. Springer Berlin Heidelberg, 2012.