GeaBase: A High-Performance Distributed Graph Database for Industry-Scale Applications

Zhisong Fu, Zhengwei Wu, Houyi Li, Yize Li, Min Wu, Xiaojie Chen, Xiaomeng Ye, Benquan Yu, Xi Hu
Ant Finacial, Inc.

***Abstract*—Graph analytics have been gaining tractions rapidly in the past few years. It has a wide array of application areas in the industry, ranging from e-commerce, social network and recommendation systems to fraud detection and virtually any problem that requires insights into data connections, not just data itself. In this paper, we present *GeaBase*, a new distributed graph database that provides the capability to store and analyze graph-structured data in real-time at massive scale. We describe the details of the system and the implementation, including a novel update architecture, called *Update Center* (UC), and a new language that is suitable for both graph traversal and analytics. We also compare the performance of GeaBase to a widely used open-source graph database *Titan*. Experiments show that GeaBase is up to 182x faster than Titan in our testing scenarios. We also achieves 22x higher throughput on social network workloads in the comparison.**

I. Introduction

We are in the age of big data. Connections between data are of same importance as data itself. Together, they record information reflecting the real world. A Graph, defined as $<V, E>$, is a natural way to represent data and their connections. Here V represents data, or namely nodes, and E represents connections between data, namely edges.

Graph databases are introduced to efficiently store and query the graph. Graphs stored in the graph database are usually property graph models (nodes, edges and properties) (see Fig. 1).

A key feature of the graph database is that edges (or connections) are treated as the core component of the model, along with vertexes. Hence, complex topological structures can be retrieved efficiently. In contrast, with conventional relational databases, connections between data are stored in separate tables, and queries searching for connections require join operations, which is usually very expensive.

However, it is challenging to design a high-performance graph database for industry-scale applications. First, irregular data structure of a graph usually leads to random access pattern to the storage system, and hence results in poor data locality; Second, in order to store a large scale of graph, the data is usually partitioned, which leads to high communication cost and imbalanced workloads; Finally, data consistency in a fast changing and distributed graph database is also very challenging.

In this paper, we introduce GeaBase (Graph Exploration and Analytics Database) that provides real-time graph traversal and analytics capabilities for industry-scale applications. We will describe the full detail of GeaBase architecture and implementation. GeaBase employs techniques, such as moving computation to where data is, double-queue update pipeline and user-stickiness *et al.* , to achieve high-performance and data consistency.

The rest of this paper is structured as follows. In Section II, we describe the related work in the literature. In Section III, we discuss implementation details and data structures of GeaBase. In Section IV we discuss the performance of GeaBase and compare the results with *Titan*, an open-source distributed graph database. In Section V we summarize the results and discuss future research directions related to this work.

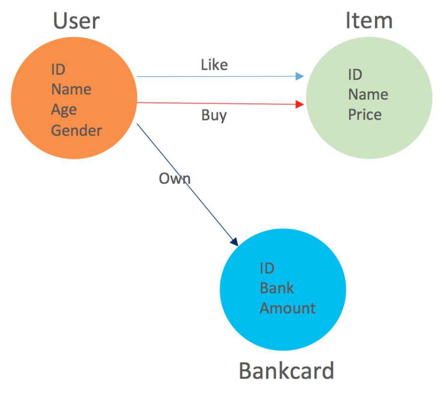

Fig. 1. An example of property graph. The user, item and bankcard are three nodes, and their connections (like, buy, own) are modeled as directed edges.

II. Related Work

Several graph databases and graph analytics systems have been introduced in the literature, for unlocking the value of data connections. Neo4j [1] is the best-known graph database according to db-engines.com [2]. Its initial release was more than ten years ago, and Neo4j has built a large develper community. However, Neo4j offers limited support for scalability (scale-up only), and hence cannot handle very large dataset for big companies like Alibaba, Inc. The state-of-the-art distributed databases that have scale-out capability [1], [3], [4] typically employ standard graph query language like *Gremlins* [5] or *SparQL* [6]. These query languages have limited support of graph analytics. Offline graph analytics systems, such as those proposed in [7]–[10], are not able to update while processing queries or respond in real-time.

978-1-5386-1072-5/17 $31.00 © 2017 IEEE
DOI 10.1109/CBD.2017.37

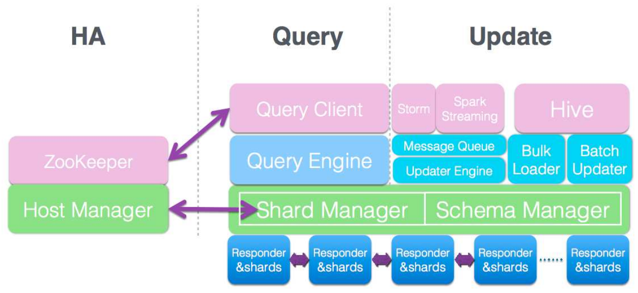

Fig. 2. System Architecture

TABLE I
STRUCTURE OF NODE KEY-VAULE RECORD

key		value
id (int64)	type (int32)	encoded_properties

TABLE II
STRUCTURE OF EDGE KEY-VAULE RECORD

	key				value
	int64	int32	int64	int64	
OutEdge	srcid	type	timestamp	dstid	encoded_properties
InEdge	dstid	-type	timestamp	srcid	encoded_properties

III. System Overview

In this section, we present the GeaBase system architecture in detail. As seen in Fig 2, GeaBase is composed of four modules: high availability module, query engine, update module and graph storage. We will describe these four modules respectively in the following subsections.

A. Graph Storage

In GeaBase, nodes and edges are all stored as key-value pairs. Properties are serialized and stored in the value part according a the user-defined schema. The records are stored in different shards according to a hash-based sharding strategy. Usually, a GeaBase instance contains tens or hundreds of shards that are maintained by the shard manager. We next describe the data model and storage model respectively in more detail.

1) Data Model: The basic data model of graph databases is to describe object-link-property (OLP). GeaBase data model designing aims to support a massive scale, multi-dimensional graph which contain typed-nodes, directed-typed-edges with properties on them. Cause multi-dimensional graph is composed by various types of nodes and edges, which represent various class of objects and complicated relationships between objects.

The data model of a GeaBase instance is described by a *Graph Schema* as mentioned above, which contain a node type list and a edge type list. A type of the nodes in *Graph Schema* contains one filed declaring its type, and a list key-value pairs reprenting the property fileds of the node. Source node type, destination node type, edge type, property list and orientation(directed or undirected) are all needed for a valid node in the schema. Besides the basic reqirements mentioned above, users can add optional fields to nodes and edges(e.g. time to live field). In a word, relenvent topology structure and serialization format can be known from this *Graph Schema* when a specific edge or node type is given.

2) Storage Model: TableIII-A3 gives the structure of the key values in the GeaBase storage. Specifically, a node is stored as a single key-value pair, which is composed of node id, node type, and node properties. A directed edge is as two key-value pairs which are called as *OutEdge* and *InEdge*. Both of the two are composed by source id, edge type, timestamp, destination id and edge properties. The *OutEdge* can be considered as forward index which is used for searching the trace to destination when source node is given (called as out-edge-navigation). The *InEdge* can be considered as reverse index which is used for searching the trace to source node when destination is given (called as in-edge-navigation). All of the edge types, node types and properties format of a given edge or node type are defined in *GraphSchema* mentiened above.

To save storage, the edge properties can be stored in one direction. For example, a certain type of edges stored properties on *OutEdge*, that means storing the forward index with real properties and reverse index with empty properties. Assuming that the capacity rate of key and vaule is $1 : n$, storing properties in both side need $2n + 2$ times capacity, storing properties in single side need $n + 1$ times capacity.

3) Sharding Strategy: Based on the data model and the storage model above, sharding is performed as follows. All records are sharded by the first 8-bytes (int64) of the key. For example, *Node* record would be scatterred by the node ID, an *OutEdge* record would be scatterred by the source ID, whereas *InEdge* is by destination ID.

The motivation of this sharding strategy is to avoid transferring massive data over network. The source node and all *OutEdges* originating from it are located in the same shard under the strategy. The same applys to the destination node and the *InEdges*. Hence, the properties of edges and destination node could be fetched in the same shard in case of forward traversal.

B. High Availability Module

The availability of GeaBase service is implemented using a ZooKeeper service, by keeping a heartbeat between the zookeeper service and all GeaBase clusters. Each GeaBase cluster has a full copy of graph data, and hence is named as a GeaBase *replica*. If one GeaBase server, or one replica crashes, the heartbeat between zookeeper and this server times out, and then this server is removed from ZooKeeper. When clients and other GeaBase servers detect this change, they redirect the computations to other servers, who have the corresponding data.

C. Update Module

This section describes details of the update module. GeaBase provides both real-time and batch approaches for data update, which are described in detail respectively in this section.

1) Real-time Update: Real-time data becomes increasingly important in business decision, financial risk management and other industrial analysis fields. For example, in e-commerce, real-time user action data, such as click, collect, purchase, are crucial to build user profile and optimize recommendation effect. In financial field, real-time user purchase and transaction data are the fundamental resource for risk management.

As seen from Fig. 3, two real-time update modes, synchronous mode and asynchronous mode, are provided for different situtaions. In synchronous mode, the upstream system translates an update event to a query string and sends it to one GeaBase server directly and waits for server to send back the response with its update status. If an update event succeeds, the subsequent read queries would get the latest data. When this mode is used together with the user-stickiness strategy mentioned in III-A, the users would always get consistent data.

In some cases, user concerns more about the date update throughput more than update timeliness. Upstream system could choose to translate the update event to a message and then put it to a distributed queue to be consumed by all GeaBase replicas asynchronously. In practice, the message queue chosen is the closest one to the upstream system when multiple distributed message queue service in different region are available. The job left for every GeaBase replica is continously consuming the update message queued by clients from all the message queues in all the regions.

Some applications do not require strict update order, and they need high update throughput. For these cases, GeaBase

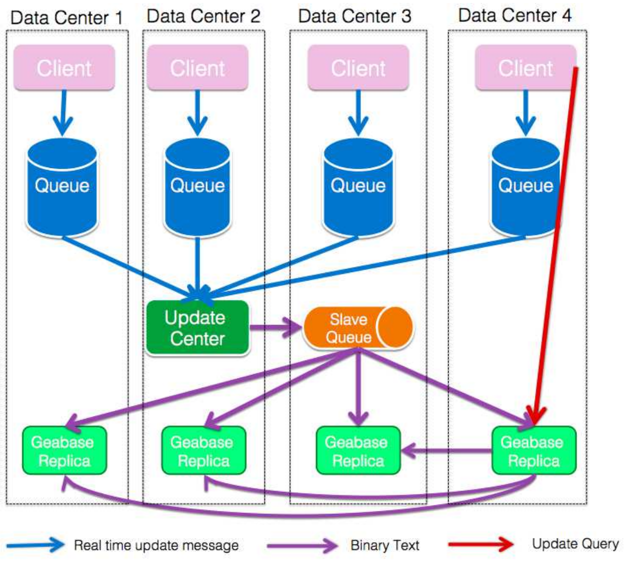

Fig. 3. Real time update architecture: synchronous mode and asynchronous mode

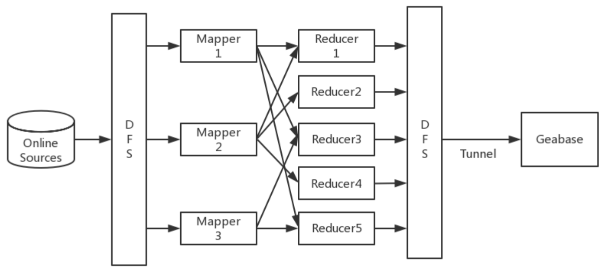

Fig. 4. Batch center architecture

provides an asynchronous mode for real-time update. The upstream system translates the update event to a message and then put it to a distributed queue, which is then consumed by all GeaBase replicas asynchronously.

Specifically, the data in the message queue is partitioned corresponding to the GeaBase sharding strategy that is described in III-A, and each GeaBase server subscribes to consume the partitions of the queue corresponding to the shards that it holds. This architecture guarantees *weak* message sequence. In each partition, the message sequence is determined by the order the message is received. However, the sequence is not guaranteed across partitions. The node or edge messages that share the same key ID will be sent to the same queue partition, therefore, the add/update/delete operations on the same node or edge are processed according to their sending order.

However, this architecture has three drawbacks. First, it does not ensure data consistency between replicas. Second, the message encoding procedure is done repeatedly for each replicas, which wastes valuable CPU resource. Third, there are too many queue consumers, and this would increase the pressure on the queue servers.

To solve the problems mentioned above, an update center architecture is proposed. As seen from the figure Fig. 3, one of the replicas is chosen as the update center, which is responsible to consume the message queue that stores user generated messages. We call this queue the *master queue*. Then, the update center writes the encoded messages into its storage and also sends it to another queue that we call the *slave queue*. All other replicas (we call *slave replicas*) consume the slave queue and update their storage accordingly. In this way, all replicas always consume identical data and hence ensure data consistency. Besides, the encoding process is performed only in the update center. It is also worth mentioning that the update center can be switched to any other replica, because the only difference between them is that they consume different message queue.

2) Batch Update: Despite the real-time update approach, there are cases when users need to import the batch computing result from data warehouses Fig. 4. The data in data warehouse are typically formatted as tables. We require that all tables contain the following elements: key ID, values and tag. The tag indicates a specific update method, such add, delete, update. GeaBase shards the table, and each server reads a subset of table and updates the database accordingly.

D. Query Engine

1) Engine Structure: One of the basic query engine design principles is to perform computation at where data is. A query is processed as follows. When a GeaBase server receives a query request, it checks the query statement's syntax correctness, and then generates the traversal and computation plan corresponding to the semantic of the query. When the query needs data that is not local, the query engine generates a subquery that retrieves remote data and send the subquery to the server where the destination node resides via internal request. If no internal requests needed, the traversal plan ends and result data is sent back to clients.

2) Query Language: GeaBase provides a new set of query language for graph query and analysis. GeaBase query language supports three kinds of operators:

- graph CRUD operations: GetNodeProp (get nodes properties), GetEdgeProp (get edges properties), Nav(igation) (graph navigation, see below), GetDistance, Limit, Add/Delete Node/Edge
- traversal, computation: Sort, Union, Subtract, Combine (set operations), Agg(group by), For (iterations), variables
- analysis: FindLoop, ShortestDistance, KCore, *et al*.
- build-in and user-defined functions for flexibility.

The GeaBase query language has a LISP lik grammar that looks like

```
(operator [:ATTR=value])*
```

where [:ATTR=value] can be a sub-query.

In the following, we illustrates some most common used operators. Please refer to our web site for more details

3) Nav: Like select clause in SQL, operator Nav(igation) is the most used operator in GeaBase query language.

```
(Nav  [:START= | :DATA=]
      [:EDGE_TYPE= |
        :REG_EDGE_TYPE=]
      [:DEPTH=][:FILTER=][:RETURN=]
      [...]
)
```

where

- start: the starting node id(s)
- edge_type | reg_edge_type: the specified edge type (edge_type) or regular expression (reg_edge_type) during the navigation.
- depth: the maximum depth of navigation (default=1, maximun=5).
- filter: logical conditions (like where clause in SQL)

Here is an example:

```
(Nav  :DATA=(Nav  :START=123
                  :EDGE_TYPE="friend"
                  :RETURN=@name, @age
                  :FILTER=@age>40)
      :EDGE_TYPE="family"
      :RETURN=$0, $1, @location, @city
)
```

The inner Nav-subquery

```
(Nav  :START=123  :EDGE_TYPE="friend"
      :RETURN=@name, @age  :FILTER=@name>40)
```

returns friends' name and age of node 123 while friends' age should be larger than 40. And then the outer Nav-subquery returns those friends' name and age ($0,$1), with those friends' family's location and city (@location, @city)

IV. Experiments

In this section, we present experiments to demonstrate the performance of the proposed system. We use the social dataset from Twitter [11] to illustrate the latency, throughput and scalability of GeaBase with a collection of queries, and compare the results with a popular open-source graph database, Titan. The performance data, as well as implementation related details, are provided in the following order: 1) latency analysis, 2) throughput analysis, followed by 3) scalability analysis.

The twitter dataset we use in our our experiments has around 1.47 billion social relations (edges in the graph) of follower-following topology that obeys a non-power-law follower distribution. In order to test the performance, we add four properties in each edge: a short string property of ten bytes, a long string property of one hundred bytes, one int64 property and one double property in each edge. The following experiments are conducted with three types of queries:

- One-hop: a NAV query executes a traversal operation that starts at a given a node (a user in our case), and traverses the edges to read the nodes that represent the followers of the starting user and return one property on the edge. In this query, one-hop traversal is executed, and ten thousand of nodes and all their properties are retrieved.
- Two-hop: in this query, two recursive NAV operations are executed to retrieve the two-hop followers of a user, with each *NAV* retrieve one hundred of nodes for each starting node. Hence, Ten thousand of nodes are retrieved in total, and one propert (the short string property) is also returned with each node.
- Four-hop: similar to the two-hop case, in this query, four recursive NAV queries are executed to retrieve the four-hop followers of a user, with each NAV operation retrieving one hundred of nodes for each starting node. One property is also retrieved for each node.

A. Experimental Setup

- Performance evaluation is conducted on machines with Linux kernel v2.6.32, 256GB RAM, dual Intel Xeon 16-Core CPU E5-2682 v4 at 2.50GHz and SSD for storage. The machines are connected with 10Gb ethernet.
- Titan version is community v1.1.0 in our experiments, with HBase as its backend storage and Elasticsearch as its

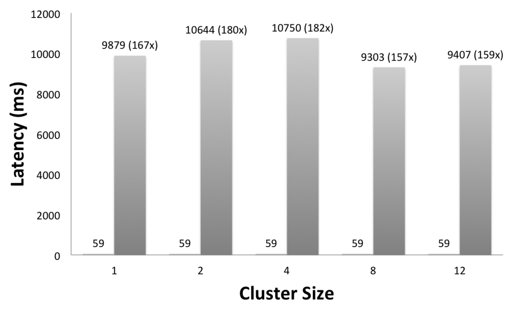

Fig. 5. Lantency comparison between GeaBase and Titan on one-hop query. Left column is GeaBase data, right column is Titan data.

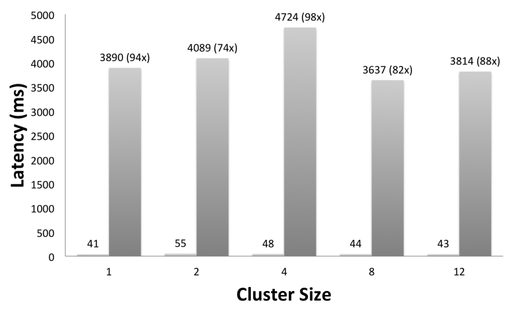

Fig. 6. Lantency comparison between GeaBase and Titan on two-hop query. Left column is GeaBase data, right column is Titan data.

mixed index service. Hadoop/HDFS version is 2.7.2 with three replicas, HBase version is 1.2.3, and Elasticsearch version is 1.5.2.

- In Titan we define a schema that follows the GeaBase's test data structrue to build a standard Titan graph. User is considered as a string proprty in the node. Titan generates a globally unique ID for each node, it will be stored in HBase as a Row Key. We also add a composite index with the property of user as key for all nodes for fast querying, and another composite index with the same user property in edge as key for all edges. In the client end, we use Java API to build a TitanGraph object and it will directly connect to the HBase cluster.
- All performance data are average of ten runs. Each query is sent only once, so the banckend storage cache will never be hit.

B. Latency Results

We first examine the performance of GeaBase by measuring the latency of a single query chosen from the three queries listed above, and compare the result with Titan.

The results shown in Fig. 5, 6 and 7 demonstrate the performance of GeaBase with the one-hop, two-hop and four-hop queries, respectively. As you can see from the figures, GeaBase achieves a speedup of up to 182x, compared to Titan, in the one-hop query case, and for the four-hop query case, we achieve up to 62x speedup.

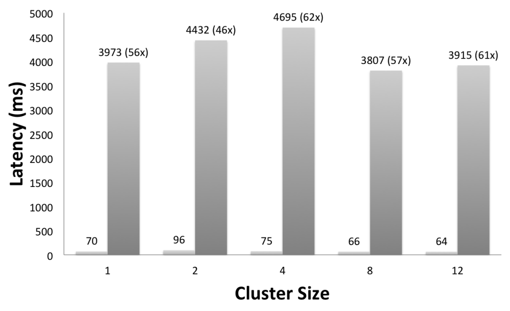

Fig. 7. Lantency comparison between GeaBase and Titan on four-hop query. Left column is GeaBase data, right column is Titan data.

The reasons for the large performance difference are as follows. First, GeaBase can achieve better data locality by data clustering (*i.e.*, all edges for each node stored together) and employing strategy of moving computation to where data is. On the other hand, Titan needs to search the starting node ID in the composite index to find the globally unique ID, and then locate the HBase row of the ID, where propertites are stored. This process leads to at leat two retrieval operations, while GeaBase needs only one operation. On the other hand, all the property keys and node IDs are stored in the same namespace, as there is only one table in Titan, that will reduce the query speed.

Besides, in the second hop traversal, Titan splits each query into a sequence of RPC calls with the same connection. For example, if you want to retrieve a node's one hundred followers, it may generate at leat one hundred remote calls, and all of these calls are sent serially, that is very expensive.

Otherwise, HBase is running on JVM, and garbage collection (GC) is uncontrollable. While GeaBase is totally written in C++ and doesn't have this problem.

The figures also show that the performance gap between GeaBase and Titan is narrower for the four-hop query case. This is because with four-hop query, the communication cost is much higher than in the case of one-hop query, where there is no communication among the workers. While for Titan, the internal communication is not a considerable factor compare with the reasons above. Hence, the performance gap between GeaBase and Titan is narrower, since commucation cost is similar given the same network specification.

C. Throughput Results

We next evaluate GeaBase's throughput performance. We use the same query as in Section IV-B and measure the total throughput of the system.

The results from Fig. 8, 9 and 10 demonstrate the superior performance again, compared to Titan. GeaBase achieves up 5x to 20x better throughput in the experiment.

D. Scalability Results

To investigate how GeaBase scales, we measure GeaBase's throughput on the same data and same query with varying number of servers. The result is shown in Fig. 11. As you can

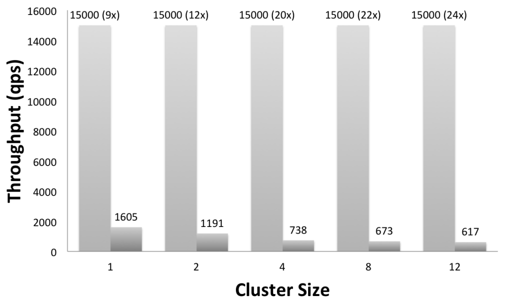

Fig. 8. Throughput comparison between GeaBase and Titan on one-hop query in queries per second. The left column is for GeaBase and the right column is for Titan. The data reported in this figure is throughput per machine.

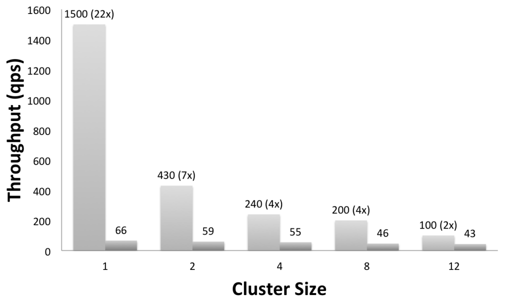

Fig. 9. Throughput comparison between GeaBase and Titan on two-hop query in queries per second. The left column is for GeaBase and the right column is for Titan. The data reported in this figure is throughput per machine.

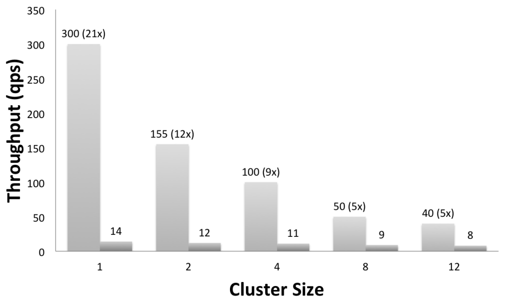

Fig. 10. Throughput comparison between GeaBase and Titan on four-hop query in queries per second. The left column is for GeaBase and the right column is for Titan. The data reported in this figure is throughput per machine.

see, GeaBase achieves almost linear scalability for the one-hop query case. For the two-hop and four-hop cases, the GeaBase engine needs to send remote queries to other servers due to the data sharding, and hence communication cost is much higher. This leads to poorer scalability for graph applications. GeaBase trys to improve this by grouping nodes according to target servers and minimizes the cost.

V. Conclusion

This work presents GeaBase, a high-performance and scalable distributed system for online graph traversal and analysis.

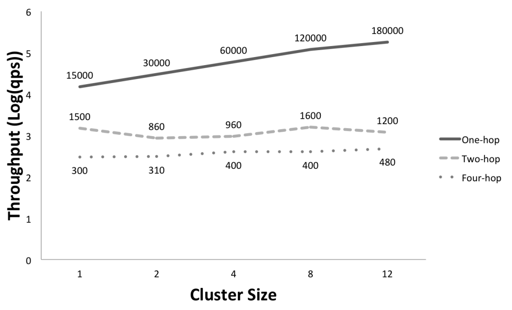

Fig. 11. Scalability results of GeaBase.

The proposed system employs novel strategies such as UC architecture and user-stickiness to achieve high-performance graph queries with consistency support. In this paper, we also compare GeaBase with a popular distributed open-source graph database, Titan, and GeaBase achieves up to 182x speedup. GeaBase provides some consistency function with a lightweight strategy (user-stickiness), however, this may not be enough in many applications. We will explore such problems in our future work. Also, we are planning to support more graph analysis functions.

Acknowledgment

The authors would like to thank Yimin Huang, Fenglin Hou, Bing Huang, Feng Chen, Hao Jiang, Jyh-Herng Chow for their work in developing GeaBase, and we would also like to thank all the users of GeaBase within Alibaba group for providing helpful feedback, suggestions, and bug reports.

References

[1] S. Zhang, C. Zhu, J. K. O. Sin, and P. K. T. Mok, "A novel ultrathin elevated channel low-temperature poly-Si TFT," *IEEE Electron Device Lett.*, vol. 20, pp. 569–571, Nov. 1999.

[2] Db-engines. [Online]. Available: https://db-engines.com/en/ranking/graph+dbms

[3] Titan. [Online]. Available: http://titan.thinkaurelius.com/

[4] Datastax. [Online]. Available: http://www.datastax.com/

[5] Gremlins. [Online]. Available: http://tinkerpop.apache.org/

[6] Sparql. [Online]. Available: https://www.w3.org/TR/rdf-sparql-query/

[7] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: A system for large-scale graph processing," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '10. New York, NY, USA: ACM, 2010, pp. 135–146. [Online]. Available: http://doi.acm.org/10.1145/1807167.1807184

[8] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "Powergraph: Distributed graph-parallel computation on natural graphs," in *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'12. Berkeley, CA, USA: USENIX Association, 2012, pp. 17–30. [Online]. Available: http://dl.acm.org/citation.cfm?id=2387880.2387883

[9] Z. Fu, H. K. Dasari, B. Bebee, M. Berzins, and B. Thompson, "Parallel breadth first search on gpu clusters," in *2014 IEEE International Conference on Big Data (Big Data)*, Oct 2014, pp. 110–118.

[10] Z. Fu, M. Personick, and B. Thompson, "Mapgraph: A high level api for fast development of high performance graph analytics on gpus," in *Proceedings of Workshop on GRAph Data Management Experiences and Systems*, ser. GRADES'14. New York, NY, USA: ACM, 2014, pp. 2:1–2:6. [Online]. Available: http://doi.acm.org/10.1145/2621934.2621936

[11] Twitter. [Online]. Available: http://an.kaist.ac.kr/traces/WWW2010.html