2012 IEEE Fifth International Conference on Cloud Computing

Distributed Graph Database for Large-Scale Social Computing

Li-Yung Ho
Institute of Information Science
Academia Sinica,
Department of Computer Science and Information Engineering
National Taiwan University
Taipei, Taiwan
Email: lyho@iis.sinica.edu.tw

Jan-Jan Wu
Institute of Information Science,
Research Center for Information Technology Innovation
Academia Sinica,
Taipei, Taiwan
Email: wuj@iis.sinica.edu.tw

Pangfeng Liu
Department of Computer Science and Information Engineering,
Graduate Institute of Networking and Multimedia,
National Taiwan University
Taipei, Taiwan
Email: pangfeng@csie.ntu.edu.tw

***Abstract*—We present an efficient distributed graph database architecture for large scale social computing. The architecture consists of a distributed graph data processing system and a distributed graph data storage system. We leverage the advantages of both systems to achieve efficient social computing.**

We conduct extensive experiments to demonstrate the performance of our system. We employ four real-world, large scale social networks – YouTube, Flicker, LiveJournal and Orkut as test data. We also implement several representative social applications and graph algorithms to examine the performance of our system. We employ two main optimization techniques in our system – indexing and graph partitioning. Experimental results indicate that our system outperforms GoldenOrb, an implementation of Pregel model from Google.

***Index Terms*—graph database, social network, social computing, cloud computing,**

I. Introduction

Social network applications provided by Facebook [1], Twitter [2] and YouTube [3] have become the most popular and important Web 2.0 applications recently. People share their daily life experiences by uploading movies and pictures to these social network websites with mobile devices and desktop computers. As a result we can easily know what our friends do, what they eat, and where they go. People are connected more closer to each other by sharing these data that describe their personal lives.

The amount of data in social network websites is tremendous. Because of the popularity of smart phone now it is easy for everyone to generate personal data and transmit them through the Internet. As a result those companies that provide social network services face a serious challenge in managing and retrieving useful information from such a large amount of data. Data volume in the scale of xeta-byte has been mentioned in [4], [5], [6], and the data growth rate is still rapidly increasing. As a result we need new approaches to efficiently store, process, and secure these data with scalability.

Recently NoSQL database has become the focus of large scale data processing. NoSQL databases is classified into two major categories – *key-value* and *column-family*. Examples of NoSQL database include Dynamo [7] from Amazon and BigTable [8] from Google. NoSQL databases are *highly scalable* because of their simple data schema and weaker consistency model when compared with relation database, and they are *fault-tolerant* because they mostly run in large scale cluster environments where failure is the norm.

We cannot apply NoSQL database to social computing for the following three reasons. First, data records stored in NoSQL database are assumed to be *independent* from each other. However most data in social computing are closely related. For example, a tagged picture can relate to multiple users, and possibly their other data as well.

Second, structure queries are very common in social computing, but it is very difficult to express structure queries concisely in NoSQL database. Social network application developers often have to use complex SQL queries to explore data relationship. These queries usually involve joining multiple tables, which may degrade performance severely. For example, most open source implementations of column-family databases only support indexing on the primary key, so to join multiple tables on the secondary key will cause extensive scan on data and is therefore inefficient.

Figure 1 compares the complexity of query expression between SQL and a graph query language of Neo4j [9] graph database. The queries in Figure 1 are to explore the structure among employees of a company. The expression of SQL query is be very complex and unreadable, as shown in the left column of Figure 1. In contrast the right column of figure 1 shows the query expression in Neo4j [9] graph database, which is more elegant and readable. The Neo4j query first traverses the network with a depth first search from the root of the tree, then retrieves the nodes and prints the properties of each node.

Finally, it is very common for a social application to traverse a graph consisting of persons based on their relationship. However, the current query interface of NoSQL database is not adequate to support graph traversal, since NoSQL only supports very primitive operations like get and set, and it is not intuitive to express graph traversal with these primitive NoSQL operations.

Graph database provides better support for social comput-

978-0-7695-4755-8/12 $26.00 © 2012 IEEE
DOI 10.1109/CLOUD.2012.33

SQL	Neo4j

SELECT LEVEL seq2, seq1,
element id, REPORT TO,
PARENT ID,MFLAG,
element_name, element _type
FROM (select LEVEL seq1,
element id, REPORT TO,
PARENT ID, MFLAG,
element name, element type,
FROM M WAY TREE
WHERE hierarchy id = 1
START WITH PARENT ID is NULL
CONNECT BY PRIOR
element id = PARENT ID)
START WITH REPORT TO IS NULL
CONNECT BY PRIOR
element id = REPORT TO
	Set nodes =
	trav -o depth -r
	PARENT_OF:outgoing
	for(Node v: nodes){
	print(v.getProperty(id))
	print(v.getProperty(name))
	print(v.getProperty(type))
	}

Fig. 1. Graph traversal query of SQL and Neo4j

ing. Data relation is the first class citizen in a graph database because graph database is optimized for structured query. For example, graph database provides both node and edge index, so graph traversal is very efficient. This has been confirmed by Vicknair et al. [10] that graph database has better performance than relational database in graph traversal. Application developers can benefit from graph database because it provide intuitive query interface to process a graph, so the query expression becomes more readable, and easy to maintain.

In order to support a large-scale social computing application we need a large scale graph database. To provide such a database it requires a *distributed graph data store* and a *distributed data processing system*. To the best of our knowledge, there has not been any open source and cloud-ready distributed graph database available to social computing. This lack of support for large scale social computing motives us to develop our distributed graph database, which is the goal of this paper.

We proposed and developed a two-layer distributed graph database – the upper layer is a distributed data processing system and the lower layer is a distributed graph data store. Current graph database systems do not provide these two functionalities simultaneously. For example, GoldenOrb [11] uses HDFS (Hadoop File System [12]) as the backend data store, which is not suitable for graph data management. InfiniteGraph [13] is a distributed graph database system but it does not have a large scale data processing system to process the data. Trinity [14] is also a distributed graph database system and has a similar architecture to our proposed system. However, Trinity focuses on real-time transactional processing and uses a distributed memory storage system as its backend store. The storage is limited by the amount of memory on each node, so it cannot process large scale data.

We summarize the contributions of this paper as follow.

- We develop an efficient distributed graph database for large scale social computing. The system consists of a distributed graph data store and a graph processing system. The graph data store provides indexing on nodes and edges for processing efficiency, and a user friendly data manipulation interface for facilitating graph data processing.
- We employ METIS [15] to partition the graph data. Many existing systems, like GoldenOrb [11], Pregel [16] and Giraph [17], use hashing to partition the data. However, hashing does not preserve data locality therefore it is not suitable to partition graph data. METIS provides better data locality than hashing and reduces the communication traffics among nodes in our distributed graph database.
- We run applications on four real-world, well-known social graphs – YouTube [3], Flicker [18], LiveJournal [19] and Orkut [20] to demonstrate the performance of our system. The four social networks consist of up to millions of nodes and ten millions of edges.

We organize the rest of this paper as follows. Section II introduces several distributed graph databases and graph processing systems and compares the differences with our system. We introduce the system architecture and components in Section III. Section IV describes the dataset of social networks and graph applications we used in the experiments. Section V demonstrates the evaluation of our system. Finally, we conclude this paper in section VI.

II. RELATED WORK

There are several distributed graph data management systems in the literature. InfiniteGraph [13] is a commercial product of distributed graph database. InfiniteGraph is based on the object-oriented database developed by Objectivity Inc. InfiniteGraph supports different levels of consistency control, from ACID to eventually consistency, and claims to be scalable. However, the free version of InfiniteGraph only supports up to one million nodes and edges, which are insufficient for real social network applications. For example, Orkut [20] contains more than 3 millions of nodes and billions of edges. In addition InfiniteGraph does not have a distributed graph data processing system for users to develop their applications to process a distributed graph.

HyperGraphDB [21] is a distributed graph database development framework. Users can customize the backend storage and indexing methods to build their own distributed graph databases. HyperGraphDB also provides a P2P layer to distribute the data. The data model in HyperGraphDB allows a hyper graph where an edge is generalized to connect more than two nodes [21]. Despite that HyperGraphDB is a framework for distributed graph data management, it does not support distributed graph data processing.

There are also parallel graph processing systems in the literature, e.g., Pregel [16] from Google, Trinity [14] from Microsoft, and GoldenOrb [11], an open source implementation of Pregel. Pregel is a computation model proposed by Google. The design goal of Pregel is to process large-scale graphs. Pregel splits computation into multiple super-steps. In each super-step every node of the graph performs a user-defined computation function independently, i.e., the computation of a node in a super-step is independent from the computation of other nodes. After the computation completes nodes send messages to other nodes in the graph and vote to halt. The messages sent by a node at the end of a super-step will be

received by the destination node at the beginning of the next super step, and the messages may affect the computation at the destination node. When all nodes agree to halt the computation stops.

Trinity [14] is a distributed in-memory graph database developed by Microsoft. The data model in Trinity is hypergraph, just like the hypergraph model in HyperGraphDB [21], i.e., an edge in a hypergraph can connect an arbitrary number of nodes, instead of two in traditional graphs. Also Trinity allows heterogeneous nodes and edges so that the nodes and edges can contain different types of data.

Trinity is optimized for concurrent on-line transactional query and supports various types of indexing. [14]. Trinity has a graph data processing system like Pregel, but unlike Pregel, Trinity supports both synchronous and asynchronous processing modes. As a result, Trinity does not have super-step in Pregel that synchronizes the computations on nodes. Operations like finding shortest path can work in asynchronous mode for better performance.

III. ARCHITECTURE

We describe the components and architecture of our distributed graph database system. Our system consists of two main components – a *distributed graph data processing system* and a *distributed graph data store*.

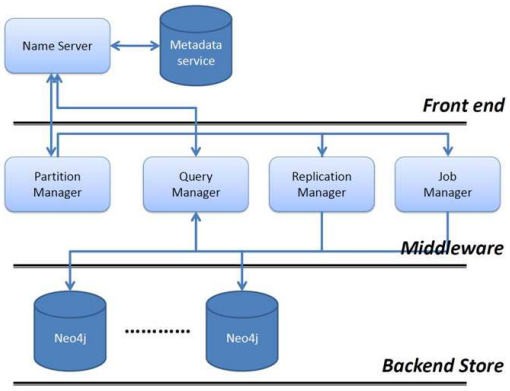


Fig. 2. The architecture of our distributed graph data storage system

A. Distributed Data Processing System

The data processing system reads data from the distributed data store, processes the data, and writes the computation results back to the distributed data store. Users develop and run their applications on the distributed graph processing system. The applications use the API provided by the backend distributed data store to facilitate data manipulation.

We use GoldenOrb [11], an open source implementation of Pregel, to be our distributed graph data processing system. However, the default backend store of GoldenOrb is the Hadoop file system, which is not optimized for graph processing. Therefore, we implement an *InputFormat* interface and connect GoldenOrb to a distributed graph data storage system we developed. Moreover, GoldenOrb uses hashing to partition graph data, which does not preserve data locality. Instead we use METIS [15] to partition the graph because METIS minimizes the number of cut edges while maintaining a balanced workload among partitions.

B. Distributed Graph Data Storage System

There are three layers in a distributed graph data storage system – the *front end layer*, the *middleware layer* and the *backend storage layer*. Please refer to Figure 2 for an illustration.

1) Front End Layer: The front end layer receives user queries, returns the results to users, and stores the metadata of the partitioned graph. The front end layer includes a *name server* and a *metadata server*. The name server receives queries and looks up necessary metadata from the metadata server. The metadata and the query are then passed to the query manager for processing. Note that the metadata server only stores the topology of the entire graph and the locations of all nodes and edges, not the data of nodes and edges.

2) Middleware Layer: The middleware layer consists of four components – a *partition manager* that partitions the input graph, a *replication manager* that replicates graph data, a *query manager* that processes query about the graph, and a *job manager* that provides access to the backend data store.

a) Partition Manager: The partition manager partitions an input graph according to a learning process. When a new node is added into the graph database, it is *not* assigned to any partition, but cached at the metadata server. The name server then starts gathering information about this node, e.g., which partition this node communicates most often with. The name server will collect necessary information required by the partitioning algorithm so that the algorithm can make a good decision. After the name server has collected sufficient information it calls the partition manager to assign the node to a partition, based on the recommendation from the partitioning algorithm. The partition manager then writes an insertion message into the queue of the job manager, and the job manager performs the insertion operation to the backend data store of the corresponding partition.

b) Query Manager: The query manager receives queries from users, dispatches query to the backend data store, and then looks up metadata from the name server for nodes and edges involved in the query, e.g., the location of the target nodes and edges. Then the query manager queries each backend store to retrieve the related data about the nodes and edges. After each backend store returns the related data, the query manager collects the results from the backend store and rebuilds the graph being queried, and sends it back to the name server.

c) Replication Manager: The replication manager is responsible for data replication. We classify the replicas in a distributed graph data storage system into *static replica* and *dynamic replica*. The purpose of static replica is fault-tolerance so each node and each edge has its own static replica. The purpose of dynamic replica is to balance workload, so only frequently accessed data, i.e., hot data, need to be replicated

to distribute the workload. The number of static replica is determined when the system initializes, but the number of dynamic replica and those to be replicated are determined at runtime.

The replication manager duplicates data dynamically to adapt to the current data access status. The replication manager analyzes the log of query manager to determine the hot data and the appropriate locations to replicate them according to a replication algorithm. Then replication manager issues request for creating duplicated node/edge messages into the queue of the job manager to duplicate nodes and edges. When a data becomes less frequently accessed, i.e., the data becomes cold, the replication manager removes its dynamic replicas in order to eliminate the extra update and consistency maintenance costs. Similarly to the deletion of regular data, the replication manager analyzes the query log and then adds node/edge deletion messages into the queue of the job manager to delete dynamic replicas.

d) Job Manager: The job manager provides an interface to access the backend data store. The job manager queues all I/O operations to the backend data store, and executes these operations from the queue. Note that each backend store has its own queue so that the operations to the same backend store are serialized.

3) Backend Storage Layer: The backend storage layer consists of a set of standalone, share-nothing graph databases. We choose Neo4j [9] as our backend data store. Neo4j has been proven to be a stable and high performance graph database [9], and is also widely used in graph database community.

IV. Social Networks and Applications

We evaluate our distributed graph database system by running representative graph applications on real-world social networks. The social networks we tested are from the relation among users from Orkut [20], YouTube [3], LiveJournal [19] and Flicker [18]. The graph applications we tested include page rank [22], maximum value propagation [16], bipartite matching, influence spreading [23] and random walk [24]. Bipartite matching are standard graph application building blocks [15]. Page rank is the process to evaluate the importance of web pages [22]. Influence spreading and random walk are two applications in social network analysis.

A. Social Networks

We evaluate our distributed graph database system by running graph applications on four popular real-world social networks consisting of Orkut, Youtube, LiveJournal and Flicker users respectively. The social network data sets are crawled by Mislove et al. [25] and published in an anonymous form. Table I summarizes the numbers of nodes and edges from the four social networks.

We partition these graphs with METIS and store each part of a graph in a node of the distributed graph data store. As a result we can classify the graph edges into two categories – *cut-edges* that connect graph nodes assigned to two different partitions, and *intra-edges* that connect two graph nodes that are assigned to the same partition. Consequently we define the *quality* of a partitioning as the *cut-edges ratio*, i.e., the ratio of the number of cut-edges to the total number of edges in the graph. The lower the cut-edges ratio, the better the quality of a partitioning, because only cut-edges will incur communication overheads.

Social Networks	Nodes (millions)	Edges (millions)
Orkut	1.2	42.4
Flicker	1.86	15.7
LiveJournal	5.28	48.8
YouTube	1.16	3.01

TABLE I
THE NUMBERS OF NODES AND EDGES OF FOUR SOCIAL NETWORKS

Two factors affect the quality of partitions made by METIS – the *imbalance ratio* and the *number* of partitions requested. We define imbalance ratio $\mathcal{I}$ as the *ratio* between the number of nodes of the largest partition and the average number of nodes in a partition. METIS uses the imbalance ratio as a threshold parameter to guide the partition so that the number of nodes in the largest partition will be no more than $\mathcal{I}$ times the average partition size.

The imbalance ratio affects the quality of partition as follows. In a social network a group of tightly connected nodes form a *community*, and the sizes of communities are very different in a social network [25]. Typically, there are one or two very large communities in a social network. However, METIS partitions a graph by minimizing the number of cut-edges while limiting the size of the largest partition. As a result a community cannot grow to its actual size because of the size limit imposed by METIS, so the number cut-edges increases and the quality of the partition suffers.

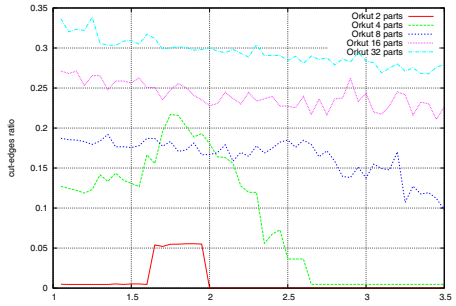
The number of partitions affects the quality of partitions similarly as the imbalance ratio does. If we instruct METIS to divide a social network into more partitions, the number of nodes allowed in each parturition decreases, which prevents the communities of a social graph to grow to their proper size.

Figure 3 depicts the cut-edges ratio of the four social networks partitioned by METIS, under different numbers of partitions and imbalance ratio. Figure 3 shows a general trend that for all social networks the cut-edges ratio decreases when the imbalance ratio increases. However, the difference is not significant. As a result we use a imbalance ratio 1.05 to control the size of the largest partition in our experiments.

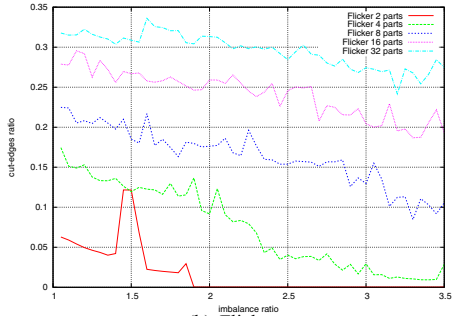
Figure 3 also indicates that the fewer the partition, the lower the cut-edges ratio. However, fewer partitions will not be able to distribute work to all available machines because we map one partition to a machine. In our experiment, we partition the social networks into *sixteen* partitions in order to map each partition to a machine in our cluster.

B. Graph Applications

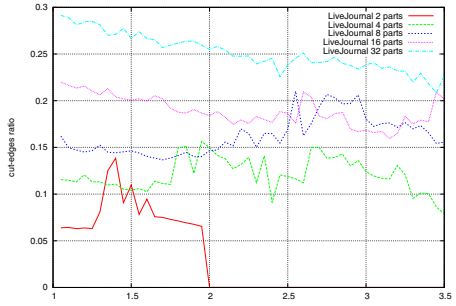
We use several representative applications as benchmarks. Note that most of these applications are suggested by Malewicz et al [16].



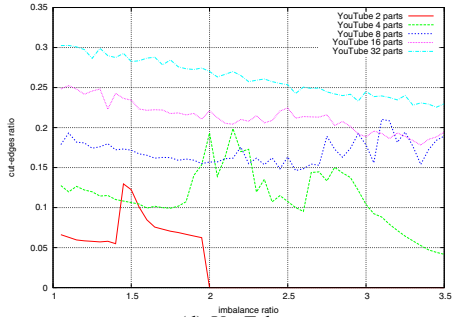
(a) Orkut



(b) Flicker



(c) LiveJournal



(d) YouTube

Fig. 3. Cut-edges ratio with varying imbalance ratio

1) Max Value Propagation: Max Value Propagation determines the largest value in a graph. Each node has an initial value, which was propagated to neighbors in the first super step. In the following super steps, as long as a node receives a value that is larger than the one it has, this node sets its value to the larger one and propagates the new value to its neighbors. On the other hand, if a node receives a value that is less than or equal to the one this node has, it ignores the value and halts. The process terminates when all nodes halt.

2) Single Source Shortest Path: Single Source Shortest Path finds the shortest path from a specified node (*source*) to all the other nodes (*destinations*) in the graph. Every node has a path length and initially all nodes, except the source node, set their path lengths to infinity. In the first step the source node sends a path length 1 to each of its neighbors and halts. In the following steps, if a node receives a path length that is shorter than its current path length, it sets its path length to the received path length, adds 1 to it, and sends the new path length to all of its neighbors. If a node receives a path length that is equal to or larger than its current path length, it ignores the path length.

3) Bipartite Matching: Bipartite matching matches every node to one of its neighbors in a bipartite graph. We implemented the randomized bipartite matching algorithm described by Malewicz et al. [16]. The nodes are partitioned into two sets – left and right, and each node is given a tag to indicate which group it belongs to.

The matching proceeds in cycles of four phases. In phase one, each left node not yet matched sends a matching request to its neighbors and halts. In phase two, if a right node not yet matched receives requests for matching, it randomly selects one and sends a message granting the request, otherwise, it halts. In phase three, if a left node not yet matched received granting messages, it chooses one and sends back an acceptance message to the chosen right node, and sends reject messages to those that were not chosen. If a left node not yet matched does not receive any granting message, it sends a matching request to its neighbors and halts. Finally in phase four the right node not yet matched chooses one acceptance message and notes the matched left node.

4) Influence Spreading: Influence spreading is an important research problem in social network analysis. Influence spreading is a technique to model how an *entity* is transmitted in a network. The entity could be information, power, or disease.

We implement the linear threshold model [23] for influence spreading. A node in the linear threshold model has two states, *active* and *inactive*. The model assumes that a node can switch from inactive to active, but it cannot switch from active to inactive. Each node v independently selects a loading threshold θ_v uniformly at random in range [0,1]. An active node u influences its neighbor v according to an edge weight $W_{u,v}$. A node v becomes active if $\sum_{u \in A} W_{u,v} \geq \theta_v$, where A is the set of active neighbors of v.

Let S be an initial set of active nodes, and $\sigma(S)$ be the expected number of active nodes that will be influenced by S, when all θ_v are from a uniform distribution between 0 and 1 [26]. To compute the exact number of $\sigma(S)$ for a given S in linear threshold model is a #P-hard problem in general graph [26].

Instead of computing the expected active nodes, we compute the final number of active nodes when all θ_v are given. Our implementation of the linear threshold model then efficiently determines which nodes will become active, given the initial

active node set S and the given threshold θ_v for every node v.

5) PageRank: We implement a PageRank algorithm described by Malewicz et al. [16]. Initially every node has a value $\frac{1}{N}$, where N is the number of nodes in the graph. In the first step every node sends its initial value divided by its out-degree to all its neighbors. In the next steps every node sums up values it received as s and set its value to $\frac{\alpha}{N} + (1-\alpha) \times s$, where α is a user-defined constant between 0 and 1.

6) Random Walk: Random walk is a technique used in various optimization problems [27], [28], [29]. For example, social network analysis uses random walk to detect community. The intuition is that it is easy for a random walker to be trapped in a group of nodes that are tightly connected. That is, the nodes in the same community tend to *see* the similar set of nodes.

This *similarity* is defined on the *transition probability* of a node. We define $P^l_{i,j}$ to be the probability for a random walker to move from node i to node j with l steps, and $P^l_{i,J} = \{P^l_{i,j}, \forall j \in V\}$ to be the probability vector of node i, where V is the set of nodes in the graph.

We compute the probability vector for each node and detect the community of a social network by comparing the *distance* of probability vectors between two nodes. The distance can be defined by Euclidean distance of the two probability vectors [24]. If two nodes have short distance between their probability vectors, they are likely to be in the same community.

V. Experiments

We conduct experiments to demonstrate the efficiency of graph computation from the indexing capability of the backend storage system. In addition we compare two partitioning methodologies – *hashing* and *METIS*, in terms of performance and the number of messages sent.

A. Setup

The experimental environment is a 16-node cluster. The cluster consists of two racks – rack A and rack B. Both rack A and rack B have eight machines. Rack A has eight Intel Xeon E5504 machines and each has 13 gigabyte RAM. Rack B has eight Intel Xeon E5520 machines and each has 23 gigabyte RAM. The machines in the same rack are connected by a gigabits switch. We run each application 10 times and use the average execution time as the performance metric.

B. Result

1) Indexing: Pregel [16] and GoldenOrb [11] do not use tailored backend storage systems for graph data. Pregel uses Google File System [30] and GoldenOrb uses Hadoop File System [12]. The two file system are not designed for graph data; instead the design principle is to process large amount of raw data with high throughput. However, relation among data should be the first citizen for graph data, and the storage system of graph should enable fast graph traversal. On the contrary Hadoop file system does not provide indexing for searching data, so it could not support efficient graph traversing on a social network.

We compare the performance of Hadoop file system and our distributed graph data store. The metric is the execution time of maximum value computation on a sub-graph of Orkut social network. With the help of indexing and fast graph traversal capability from backend Neo4j, our distributed storage system can identify and retrieve the sub-graph very quickly. In contrast, a Hadoop file system has to scan through the entire graph and filter out the targeted sub-graph, which is not efficient for a large social network. Other applications also exhibit the similar performance gain of Neo4j on Orkut graph.

Figure 4 shows the execution time of computing the maximum value of a sub-graph for different sizes of Orkut graphs. The size of Orkut graphs varies from 10k nodes to one million nodes. Each experiment select a sub-graph with 1% of the nodes to compute the maximum value. The execution time includes data loading time (lower part in deep blue) and computation time (the upper part in light blue).

Figure 4 indicates that Hadoop file system spends lots of time on loading data and filtering out the targeted sub-graph. The overhead becomes more obvious when the graph size increases. In contrast our system retrieves the targeted sub-graph quickly with the help of indexing, without loading unnecessary data or a filtering process.

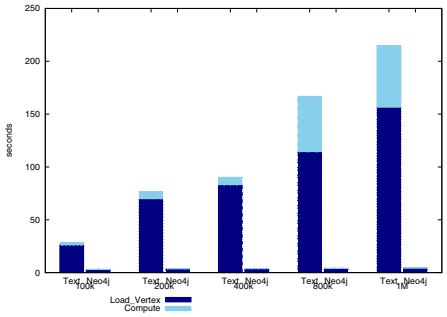
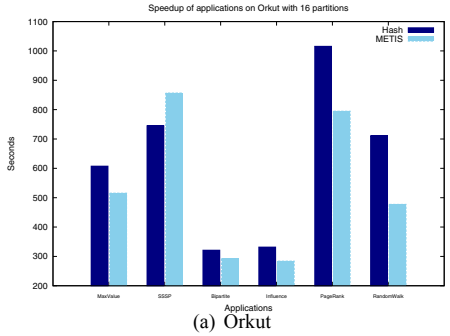


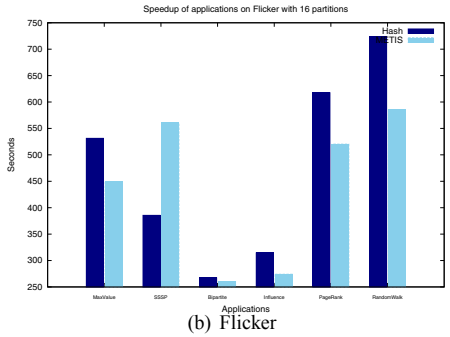
Fig. 4. Execution time breakdown for computing the maximum value of a sub-graph with Hadoop and Neo4j

2) Partitioning: We now compare two partition methods – hashing and METIS. We use them to partition a social network and store each part of the network in a distributed, sharing-nothing Neo4j storage system. We then run graph applications on the two sets of partitioned data from hashing and METIS respectively, and compare the performance of the applications.

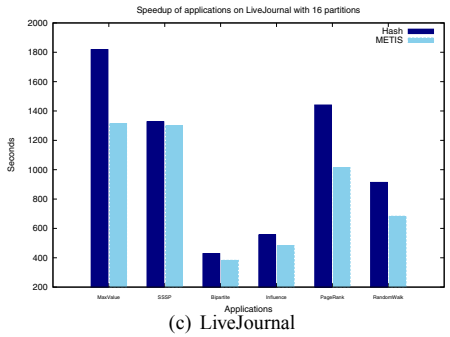
Figure 5 (a) through (d) indicate the execution time of seven applications on four social networks. We observe that all applications, except single source shortest path computation, perform better with METIS partitioning. For example, MaxValue, PageRank and Random Walk perform much better with METIS partitioning because they generate heavy message traffics among partitions in large social networks, and METIS does reduce the cross-partition traffics by minimizing the number of cut-edges among partitions. As a result we reduce the communication overhead and improve the performance.



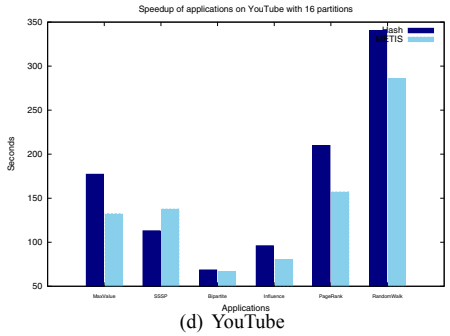
(a) Orkut



(b) Flicker



(c) LiveJournal



(d) YouTube

Fig. 5. Execution time comparison between Hash and METIS partitioning

Figure 6 and Figure 7 show the breakdown of messages sent among partitions in Random Walk application on Orkut graphs. Random Walk sends about equal number of messages from every partition to all other partitions under hash partitioning. In contrast Random Walk tends to send messages to itself under METIS partitioning. This is a clear evidence that METIS reduces the cross-partition traffics significantly and makes the computation more localized. For example, Random Walk runs 32% faster on METIS than hashing on Orkut graphs.

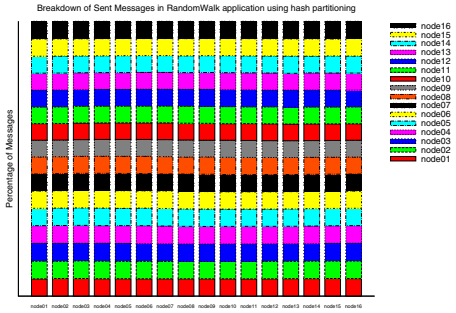


Fig. 6. Message breakdowns of hash partition in Random Walk

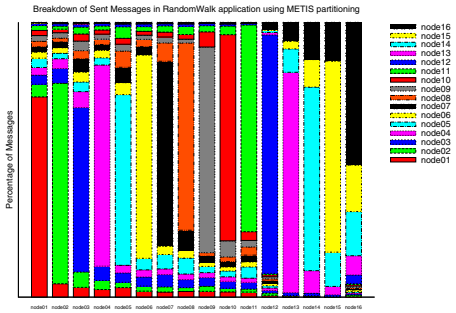


Fig. 7. Message breakdowns of METIS partition in Random Walk

Bipartite matching and influence spread generate less messages among partitions. In these two applications a vertex performs computation only when it receives messages, so they incur light computation workload. Nevertheless these applications still run better on METIS than on hashing; for example the improvement is 15% on running influence spreading on YouTube graph.

Single source shortest path computation performs worse in METIS than in hashing, due to a heavy load imbalance at the first few steps. Single source shortest path computation propagates path information to neighbors, so a tightly connected partition will generate many messages to itself at the first few steps. For example, Figure 8 indicates that when running single source shortest path application on Flicker graphs, node06 and node11 send over 1.5 millions messages to themselves at step 4, while other partitions only process ten to hundred thousands of messages. The reason is the partitions in node06 and node11 are tightly connected, and the number of intra-edges of the two partitions are about 5 times the average number of intra-edges in other partitions. Since machines must synchronize between super-step, other machines will wait for node06 and node11 to finish. This load imbalance inflates the execution time of single source shortest path application.

VI. Conclusion

We develop an efficient distributed graph database system for large scale social computing. The system consists of a

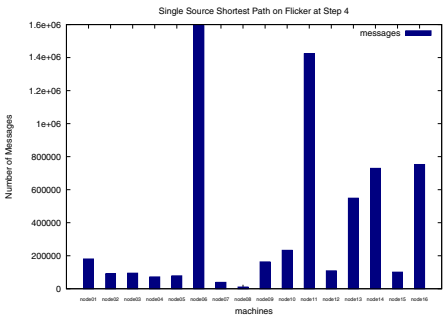


Fig. 8. Number of message at step 4

data processing system and a storage system. We leverage the advantages of both systems to achieve efficient social computing. The data processing system provides capability to process extremely large social networks, and the backend storage system provides efficient data manipulation.

We conduct experiments to demonstrate the efficiency of our system by running representative applications on real-world large scale social networks including Youtube [3], Flicker [18], LiveJournal [19] and Orkut [20]. Experimental results indicate that our system outperforms Hadoop file system [12] in sub-graph computation on a social network. Our backend storage system provides indexing and fast traversal capability so that the applications can retrieve the targeted sub-graph efficiently without loading or filtering data.

We employ METIS [15] to partition social networks and store each part of them in the sharing-nothing Neo4j databases of our distributed storage system. The experimental results demonstrate that METIS partitioning does reduce the amount of cross-partition messages and make the computation more localized for social network computing.

REFERENCES

[1] "Facebook," http://www.facebook.com/.

[2] "Twitter," http://twitter.com/.

[3] "Youtube," http://www.youtube.com/.

[4] R. Baeza-Yates and R. Ramakrishnan, "Data challenges at yahoo!" in *Proceedings of the 11th international conference on Extending database technology: Advances in database technology*, ser. EDBT '08. New York, NY, USA: ACM, 2008, pp. 652–655. [Online]. Available: http://doi.acm.org/10.1145/1353343.1353421

[5] A. Silberstein, A. Machanavajjhala, and R. Ramakrishnan, "Feed following: the big data challenge in social applications," in *Databases and Social Networks*, ser. DBSocial '11. New York, NY, USA: ACM, 2011, pp. 1–6. [Online]. Available: http://doi.acm.org/10.1145/1996413.1996414

[6] A. Thusoo, Z. Shao, S. Anthony, D. Borthakur, N. Jain, J. Sen Sarma, R. Murthy, and H. Liu, "Data warehousing and analytics infrastructure at facebook," in *Proceedings of the 2010 international conference on Management of data*, ser. SIGMOD '10. New York, NY, USA: ACM, 2010, pp. 1013–1020. [Online]. Available: http://doi.acm.org/10.1145/1807167.1807278

[7] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: amazon's highly available key-value store," in *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, ser. SOSP '07. New York, NY, USA: ACM, 2007, pp. 205–220. [Online]. Available: http://doi.acm.org/10.1145/1294261.1294281

[8] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "Bigtable: A distributed storage system for structured data," *ACM Trans. Comput. Syst.*, vol. 26, pp. 4:1–4:26, June 2008. [Online]. Available: http://doi.acm.org/10.1145/1365815.1365816

[9] "Neo4j," http://neo4j.org/.

[10] C. Vicknair, M. Macias, Z. Zhao, X. Nan, Y. Chen, and D. Wilkins, "A comparison of a graph database and a relational database: a data provenance perspective," in *Proceedings of the 48th Annual Southeast Regional Conference*, ser. ACM SE '10. New York, NY, USA: ACM, 2010, pp. 42:1–42:6. [Online]. Available: http://doi.acm.org/10.1145/1900008.1900067

[11] "Goldenorb," http://goldenorbos.org/.

[12] "Hadoop file system," http://hadoop.apache.org/hdfs/.

[13] "Infinitegraph," http://www.infinitegraph.com/.

[14] "Trinity," http://research.microsoft.com/en -us/projects/trinity/default.aspx.

[15] G. Karypis and V. Kumar, "Multilevel algorithms for multi-constraint graph partitioning," in *Supercomputing, 1998. SC98. IEEE/ACM Conference on*, nov. 1998, p. 28.

[16] 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 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

[17] "Giraph," http://incubator.apache.org/giraph/.

[18] http://www.flickr.com/.

[19] http://www.livejournal.com/.

[20] "Orkut," http://www.orkut.com.

[21] "Hypergraphdb," http://www.kobrix.com/hgdb.jsp.

[22] S. Brin and L. Page, "The anatomy of a large-scale hypertextual web search engine," in *Proceedings of the seventh international conference on World Wide Web 7*, ser. WWW7. Amsterdam, The Netherlands, The Netherlands: Elsevier Science Publishers B. V., 1998, pp. 107–117. [Online]. Available: http://dl.acm.org/citation.cfm?id=297805.297827

[23] D. Kempe, J. Kleinberg, and E. Tardos, "Maximizing the spread of influence through a social network," in *Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining*, ser. KDD '03. New York, NY, USA: ACM, 2003, pp. 137–146. [Online]. Available: http://doi.acm.org/10.1145/956750.956769

[24] P. Pons and M. Latapy, "Computing communities in large networks using random walks," in *Computer and Information Sciences - ISCIS 2005*, ser. Lecture Notes in Computer Science, p. Yolum, T. Gungor, F. Gurgen, and C. Ozturan, Eds. Springer Berlin / Heidelberg, 2005, pp. 284–293.

[25] A. Mislove, M. Marcon, K. P. Gummadi, P. Druschel, and B. Bhattacharjee, "Measurement and Analysis of Online Social Networks," in *Proceedings of the 5th ACM/Usenix Internet Measurement Conference (IMC'07)*, San Diego, CA, October 2007.

[26] W. Chen, Y. Yuan, and L. Zhang, "Scalable influence maximization in social networks under the linear threshold model," in *Proceedings of the 2010 IEEE International Conference on Data Mining*, ser. ICDM '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 88–97. [Online]. Available: http://dx.doi.org/10.1109/ICDM.2010.118

[27] M. Doi and S. Edwards, *The theory of polymer dynamics*, ser. International series of monographs on physics. Clarendon Press, 1988. [Online]. Available: http://books.google.com.tw/books?id=dMzGyWs3GKcC

[28] N. Goel and N. Dyn, *Stochastic models in biology*. Academic Press, 1974. [Online]. Available: http://books.google.com.tw/books?id=xK4fC6UYQ68C

[29] S. Redner, *A Guide to First-Passage Processes*. Cambridge University Press, 2001. [Online]. Available: http://books.google.com.tw/books?id=xtsqMh3VC98C

[30] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The google file system," in *Proceedings of the nineteenth ACM symposium on Operating systems principles*, ser. SOSP '03. New York, NY, USA: ACM, 2003, pp. 29–43. [Online]. Available: http://doi.acm.org/10.1145/945445.945450