Quegel: A General-Purpose System for Querying Big Graphs

Qizhen Zhang Da Yan James Cheng
Department of Computer Science and Engineering
The Chinese University of Hong Kong
{qzzhang, yanda, jcheng}@cse.cuhk.edu.hk

ABSTRACT

Inspired by Google's Pregel, many distributed graph processing systems have been developed recently to process big graphs. These systems expose a vertex-centric programming interface to users, where a programmer thinks like a vertex when designing parallel graph algorithms. However, existing systems are designed for tasks where most vertices in a graph participate in the computation, and they are not suitable for processing light-workload graph queries which only access a small portion of vertices. This is because their programming model can seriously under-utilize the resources in a cluster for processing graph queries.

In this demonstration, we introduce a general-purpose system for querying big graphs, called **Quegel**, which treats queries as first-class citizens in the design of its computing model. Quegel adopts a novel *superstep-sharing* execution model to overcome the weaknesses of existing systems. We demonstrate it is user-friendly to write parallel graph-querying programs with Quegel's interface; and we also show that Quegel is able to achieve real-time response time in various applications, including the two applications that we plan to demonstrate: point-to-point shortest-path queries and XML keyword search.

1. INTRODUCTION

Graph data are very common in real-life applications nowadays, such as online social networks, mobile communication networks and the Semantic Web. These graphs often contain billions to trillions of vertices and edges, and require dedicated infrastructure for efficient processing. Pregel [7] and Pregel-like systems [1, 3, 11, 10, 6] have become popular in recently years due to their user-friendly programming paradigm and good horizontal scalability. In a Pregel-like system, a programmer only needs to specify the behavior of one generic vertex, and the execution of the specified computation logic on all vertices is automatically scheduled by the system, which also handles other issues such as fault tolerance and horizontal scalability.

However, existing Pregel-like systems are designed for graph-analytic tasks with *heavy-weight* workload, where most part of a graph or the entire graph is accessed. For example, the PageRank algorithm of [7] accesses all vertices in each iteration. However, many real-world applications involve various types of graph querying, whose computation is *light-weight* as they only need to access a small portion of the input graph. For instance, in our collaboration with researchers in a large online shopping platform, we have seen huge demands for querying different aspects of big graphs for all sorts of analysis to boost sales and improve customer experience. A particular example is that they need to frequently examine the *shortest-path distance* between some users in a large network extracted from their online shopping data. In this case, *point-to-point shortest-path* (PPSP) queries studied in this work is much more efficient than *single-source shortest-path* (SSSP) [7] because only the paths between the queried users are of interest.

The importance of querying big graph has also been recognized in some recent work [5]. The work identifies two kinds of systems: (1) those for offline graph analytics (e.g, Pregel and GraphLab) and (2) those for online graph querying, including Horton [9] and G-SPARQL [8]. However, the online systems are only tailored for specific queries, and so far, there lacks a *general-purpose framework* that allows users to easily design distributed algorithms for efficiently answering various types of queries on big graphs.

While we may also answer graph queries using an existing Pregel-like system, this approach suffers from the following weaknesses. There are two solutions to utilizing an existing Pregel-like system for processing queries on demand:

- to process queries one by one, which results in a low throughput since the communication workload of one query is usually too light to fully utilize the network bandwidth and there can be many synchronization barriers; or
- to hardcode a program to process a batch of queries together and to take care of the stop conditions manually, which is not user-friendly and may suffer from the straggler problem towards the end of the processing, since most queries may have finished their processing.

There is also no obvious solution to using graph indexing to process queries in existing Pregel-like systems.

To address the above limitations, we developed **Quegel**, a distributed graph-querying engine, which treats queries as first-class citizens: users only need to write a Pregel-like algorithm for processing a generic query, and the system automatically schedules the processing of multiple incoming queries on demand. As a result, Quegel is able to answer a query as long as it can be processed by a Pregel-style algorithm (but with much better performance). Quegel adopts a novel *superstep-sharing execution model* to effectively utilize the cluster resources, and uses many other optimization techniques to improve system performance and reduce memory consumption. Quegel also provides a convenient inter-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

SIGMOD'16, June 26-July 01, 2016, San Francisco, CA, USA
© 2016 ACM. ISBN 978-1-4503-3531-7/16/06...$15.00
DOI: http://dx.doi.org/10.1145/2882903.2899398

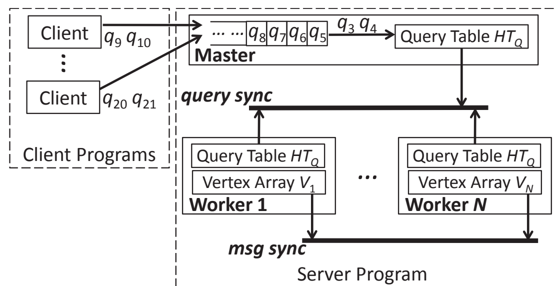

Figure 1: System architecture of Quegel

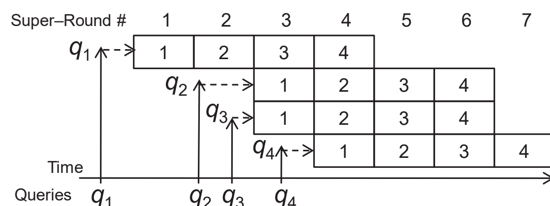

Figure 2: Illustration of superstep-sharing

face for constructing graph indexes, which can significantly improve query performance. To our knowledge, Quegel is the first general-purpose programming framework for querying big graphs at interactive speeds on a distributed cluster.

The rest of this paper is organized as follows. Section 2 introduces the system design of Quegel, and illustrates how the *superstep-sharing execution model* efficiently processes multiple queries on demand, and manages memory space efficiently. Section 3 presents the programming interface of Quegel, and demonstrates it is easy to program graph-querying algorithms in Quegel. Finally, in Section 4, we compare the performance of Quegel with the state-of-the-art vertex-centric systems, and introduce our demonstration plan.

2. THE QUEGEL SYSTEM

Due to space limit, we refer readers to the full paper of Quegel [12] for more details of Quegel.

System Architecture. Quegel follows a Client/Server architecture. There is a server program that is responsible for loading the input graph and processing incoming graph queries. Users submit their queries to the client programs, which then send them to the server. Quegel supports an arbitrary number of clients. The architecture of Quegel is shown in Figure 1. The server program consists of a master and a cluster of workers. The master keeps receiving incoming queries and appending them to a query queue. Queries are periodically fetched from the queue to be processed, whose information is maintained in a query table HT_Q. The HT_Q of master is synchronized to all workers at each communication barrier for processing, as indicated by “query sync” in Figure 1. Meanwhile, vertices on different workers exchange messages with each other, as indicated by “msg sync”.

The server program is deployed on top of Hadoop Distributed File System (HDFS). Initially, the server program loads an input graph G from HDFS, i.e., it distributes vertices into main memories of the workers. After the graph is loaded, if users have enabled graph indexing, each worker will build an index from its local vertices. Then, the server program receives incoming queries and processes them with the user-defined computing logic in a similar way as in Pregel. To interact with the server program, users may either type their queries in a client console, or submit a file containing a batch of queries.

Application Scenarios. Quegel adopts a novel *superstep-sharing* execution model to address the problems of existing systems mentioned in Section 1. Before introducing the execution model of Quegel, we first present the design goals of Quegel. Since it is difficult to achieve both high querying throughput and short response time in querying a big graph, we target at two big graph querying scenarios as follows, both of which are common in real world applications.

Scenarios (i): Interactive Querying, in which a user submits a query, checks the result, and then submits another refined query again, until he/she obtains the desired result. In this scenario, a query is expected to be answered at interactive speed. For example, consider a data scientist in a social media company, who interacts with its social network to study user behaviors. However, no existing Pregel-like system can meet this requirement.

Scenarios (ii): Batch Querying, in which a user submits a batch of queries to the system, and these queries need to be answered in a reasonable amount of time. For example, consider the vertex-pair sampling approaches for estimating graph metrics (e.g., diameter, betweenness centrality), where a large number of PPSP queries need to be answered. Quegel is up to two orders of magnitude faster than existing systems for batch querying, and can thus provide more accurate estimates of the various graph metrics.

Superstep-Sharing. To meet the requirements of both scenarios mentioned above, Quegel adopts a novel *superstep-sharing execution model*. In this model, each iteration is called a **super-round**. In a super-round, Quegel evaluates all queries in HT_Q by one superstep (hence the name *superstep-sharing*); while from the perspective of a query, it is processed one superstep after another like in Pregel. Quegel numbers the superstep of each query separately, and thus if two queries are submitted at two different super-rounds, their superstep numbers are different in the same super-round.

Figure 2 illustrates the execution process of superstep-sharing, in which four queries q_1, q_2, q_3 and q_4 are submitted to Quegel at different time. Assume that all the four queries need 4 supersteps to be processed. At the first two super-rounds, there is only one query q_1, which executes two supersteps. Since q_2 and q_3 are received when Quegel is processing super-round 2, at the beginning of super-round 3, they are fetched from the query queue of the master for processing together with q_1 (note that in this super-round, the superstep number of q_1 is 3, while the superstep number of q_2 and q_3 are 1). When Quegel is processing this super-round, q_4 is appended to query queue. When super-round 4 begins, q_4 is fetched for processing along with the other three queries. In this super-round, q_1 executes its last superstep, q_2 and q_3 execute their second superstep, and q_4 executes its first superstep.

When processing a super-round, different workers run in parallel, where each worker evaluates every query $q \in HT_Q$. If q has not been answered yet, the worker performs the user-defined vertex-centric computation on each of its vertices that are activated by q; otherwise, the worker reports the results of q and and releases the resources occupied by q. Messages (resp. aggregator and control information) of all queries are only synchronized at the end of a super-round, as illustrated by “msg sync” (resp. “query sync” for query-specific aggregators and control information) in Figure 1.

Benefits of Superstep-Sharing. For interactive querying where queries are processed one at a time, the superstep-sharing model

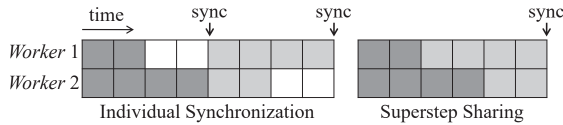

Figure 3: Load balancing of superstep-sharing

uses all the resources of the cluster to process each query as in Pregel. However, Quegel has a shorter query latency since the graph is already memory-resident, and graph indexing is supported.

For batch querying, superstep-sharing combines the workloads of multiple queries together in each super-round to better utilize the cluster resources. This model is more efficient than answering queries one after another (as in existing Pregel-like systems), since only one synchronization barrier is required in each super-round. In other words, in a super-round, the messages of multiple queries can be sent in one batch to better utilize the network bandwidth. The superstep-sharing model also significantly reduces the synchronization cost. Note that since the workload of processing each query is light, performing a synchronization barrier for each query is relatively expensive.

Superstep-sharing also results in a more balanced workload distribution. We explain the reason with the example given in Figure 3, which shows the execution of two queries in one superstep in a cluster of two workers. The first query (darker shading) takes 2 time units on Worker 1 and 4 time units on Worker 2, while the second query (lighter shading) takes 4 time units on Worker 1 and 2 time units on Worker 2. The figure on the left of Figure 3 shows the case where the second query needs to wait for the synchronization of the first query before starting its processing. Thus, 8 time units are required in total. In contrast, by using superstep-sharing as illustrated in the figure on the right of Figure 3, only 6 time units are required.

System Design. Three kinds of data are managed in Quegel: (i) *V-data*, whose value only depends on a vertex, for example, the adjacency list of the vertex. (ii) *VQ-data*, whose value depends on both a vertex and a query, e.g., the distance of a vertex from the source in a PPSP query. (iii) *Q-data*, whose value depends on a query, such as the superstep number of each query.

As shown in Figure 1, in Quegel, each worker keeps a hash table HT_Q that holds the Q-data of every query in evaluation. When a new query q is fetched from the master for processing, its Q-data is inserted into HT_Q of every worker; correspondingly, after q is finished with evaluation, its Q-data is removed from the tables. Similarly, each vertex maintains its VQ-data by a lookup table LUT_v, where $LUT_v[q]$ refers to the VQ-data for query q. To reduce memory consumption, Quegel allocates a VQ-data to a vertex v for query q only if q accesses v in its processing.

3. PROGRAMMING INTERFACE

To write a program in Quegel, a programmer only needs to (1) subclass the base classes of Quegel with proper template arguments, and (2) implement the user-defined functions (UDFs) to specify the application logic. There are two important base classes in Quegel, *Vertex* and *Worker*, which we introduce next.

Vertex Class. The *Vertex* class has an UDF *compute*(.) for programmers to specify the computation behavior of a vertex. In *compute*(.), function *get_query()* may be called to get the content of q, the query currently being processed (e.g., source and destination vertices in a PPSP query). There are also interfaces for users to

```
class BFSVertex:public Vertex<VertexID, int, BFSValue, char, intpair>{
    virtual void compute(MessageContainer& messages){
        if(superstep()==1) {
            broadcast message to neighbors
        }
        else if(qvalue()==INT_MAX){
            //step i marks all vertices (i-1) hops away from src
            qvalue()=superstep()-1;
            if(id == get_query().v2) force_terminate();  //dst reached
            else {
              broadcast message to neighbors
            }
        }
        vote_to_halt();
    }
};

class BFSWorker:public Worker<BFSVertex>{
    virtual BFSVertex* toVertex(char* line){
        parse line to vertex object and return
    }
    virtual intpair toQuery(char* line){
         parse line to (src, dst) and return
    }
}
```

Figure 4: Code snippet of implementation of BFS for PPSP

access the VQ-data of the current vertex (e.g., the vertex value for query q) or update it (e.g., vote the vertex to halt). One may also obtain other Q-data of the current query q, such as q's superstep number, and may call *force_terminate()* to terminate the processing of q. The vertex class takes the form *Vertex*<I, V^Q, V^V, M, Q>, where the five template arguments are given as follows: (1) <I> specifies the type (e.g., int) of vertex ID. (2) <V^Q> specifies the type of the VQ-data of a vertex. (3) <V^V> specifies the type of V-data of a vertex. (4) <M> specifies the type of messages exchanged between vertices. (5) <Q> specifies the query content type (e.g., for a PPSP query, <Q> is a pair of source and destination vertices).

Worker Class. Worker class provides UDFs to specify the format of data input and output, including how to parse a query string into a query content, how to parse an input line from HDFS into a vertex object, how to save the VQ-data of a query (i.e., query results) to HDFS, how to construct a local index from the vertices of a worker, etc. The worker class takes the form of *Worker*<T_{vtx}, T_{idx}>, where <T_{vtx}> specifies the user-defined subclass of *Vertex*, and <T_{idx}> specifies the optional index class.

The interface of Quegel is easy to use. For example, to implement a breadth-first search (BFS) based algorithm for answering PPSP queries, users only need to define the subclasses of *Vertex* and *Worker* as illustrated in Figure 4. The code for the Quegel system and various applications on top of it can be found in [2].

4. THE DEMONSTRATION

In this section, we present our demonstration plan, including the settings of demonstration, and the demonstration scenarios.

4.1 Settings of Demonstration

Environment. We will run the Quegel applications in a cluster of 21 machines (with 1 serving as the master), each with two 2.0GHz Intel Xeon E5-2620 CPUs and 48GB DDR3 RAM. The machines are connected by Gigabite Ethernet, running 64-bit CentOS 6.5 with Linux kernel 2.6.32. The HDFS is from Apache Hadoop 2.7.1.

Applications. To demonstrate the efficiency of Quegel for querying big graphs, we have implemented many graph querying algorithms in Quegel, two of which are given as follows (others include point-to-point reachability queries, terrain shortest path queries, graph keyword queries, etc.). Table 1 shows the real graph datasets we will use in our demonstration.

Table 1: Graph Datasets (M=million)

Dataset	$\lvert V \rvert$	$\lvert E \rvert$	Max Deg	Avg Deg
Twitter	52.58 M	1963 M	0.78 M	37.34
BTC	164.7 M	772.8 M	1.64 M	4.69
LiveJ	10.69 M	224.6 M	1.05 M	21.01

(a) Datasets for PPSP Queries

Dataset	$\lvert V \rvert$	Doc Size	Graph Size
DBLP	81.85M	1.4 GB	4.9 GB
XMark	170.53 M	5.5 GB	14 GB

(b) Datasets for XML Keyword Search

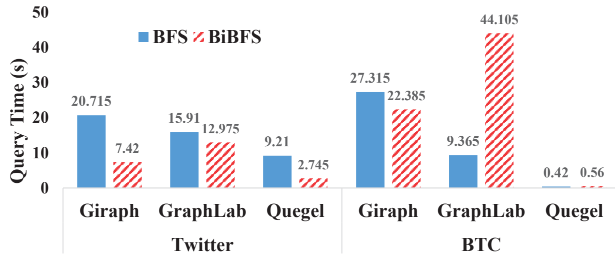

Figure 5: Performance comparison on *Twitter* and *BTC*

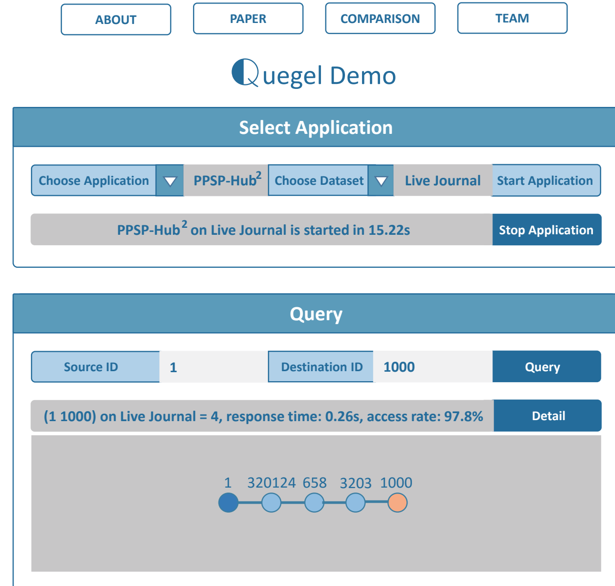

Figure 6: A screenshot of Quegel Demo

(i) PPSP Queries. We implemented three algorithms to answer PPSP queries: BFS, bidirectional BFS (abbr. *BiBFS*) and *Hub*^{2}-*Labeling* (abbr. *Hub*^{2}) [4]. *Hub*^{2} uses a graph index and achieves much better performance than *BFS* and *BiBFS*, and while it is easy to implement *Hub*^{2} using Quegel, it is not clear how to implement it in existing vertex-centric systems.

(ii) XML Keyword Search. This application requires each worker to construct a local inverted index on its vertices, to map each keyword to those vertices that contain it. The indexing can be easily implemented in Quegel, and then queries can be evaluated by starting message propagation from the matched vertices.

System Comparison. To give the audience an idea about the performance of Quegel, we compare Quegel with two state-of-the-art systems, Giraph [1] and GraphLab [3], by running *BFS* and *BiBFS* on the two datasets *Twitter* and *BTC* for solving 20 randomly-generated PPSP queries. Figure 5 shows the average query performance of the systems, which shows that Quegel is much faster than Giraph and GraphLab. In fact, when *Hub*^{2} is used in Quegel, we can answer multiple PPSP queries in a second. More performance results can be found in the full Quegel paper [12].

4.2 Demonstration Scenarios

For the demo, we built a web interface (of brower/server architecture) on top of the Quegel system, and we call the website *Quegel Demo*. A user inputs his/her graph queries into a browser, which transmits them to the web server of *Quegel Demo*. *Quegel Demo* then submits those queries to our Quegel backend, and whenever a query gets answered, *Quegel Demo* sends the results back to the browser for display.

The web interface of *Quegel Demo* allows users to select the desired querying application and dataset to start the Quegel server program. After the program is started, a user can submit queries through a webpage. *Quegel Demo* allows users to customize the visualization module of each specific application. For example, for PPSP queries, we implemented a module for displaying the shortest path (see Figure 6), while for XML keyword search, the visualization module displays the results as tree fragments. We provide over 10 scenarios for demonstration in *Quegel Demo*.

The target audience of this demo includes anyone who is interested in querying big graphs, and *Quegel Demo* supports many attendees to participate at the same time. The participants can simply select their interested applications on a webpage of *Quegel Demo*, and type their queries into the browser. *Quegel Demo* will process these queries in parallel and respond to users at interactive speeds. Users can see the visualized query results and the reported performance statistics on the webpage.

Acknowledgments. We thank the reviewers for their valuable comments. The authors are supported by the Hong Kong GRF 2150851.

5. REFERENCES

[1] Apache Giraph: http://giraph.apache.org.
[2] Quegel: http://www.cse.cuhk.edu.hk/quegel/.
[3] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *OSDI*, pages 17–30, 2012.
[4] R. Jin, N. Ruan, B. You, and H. Wang. Hub-accelerator: Fast and exact shortest path computation in large social networks. *CoRR*, abs/1305.0507, 2013.
[5] A. Khan and S. Elnikety. Systems for big-graphs. *PVLDB*, 7(13):1709–1710, 2014.
[6] Y. Lu, J. Cheng, D. Yan, and H. Wu. Large-scale distributed graph computing systems: An experimental evaluation. *PVLDB*, 8(3):281–292, 2014.
[7] G. Malewicz, M. H. Austern, A. J. C. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *SIGMOD*, pages 135–146, 2010.
[8] S. Sakr, S. Elnikety, and Y. He. G-SPARQL: a hybrid engine for querying large attributed graphs. In *CIKM*, pages 335–344, 2012.
[9] M. Sarwat, S. Elnikety, Y. He, and M. F. Mokbel. Horton+: A distributed system for processing declarative reachability queries over partitioned graphs. *PVLDB*, 6(14):1918–1929, 2013.
[10] D. Yan, J. Cheng, Y. Lu, and W. Ng. Blogel: A block-centric framework for distributed computation on real-world graphs. *PVLDB*, 7(14):1981–1992, 2014.
[11] D. Yan, J. Cheng, Y. Lu, and W. Ng. Effective techniques for message reduction and load balancing in distributed graph computation. In *WWW*, pages 1307–1317, 2015.
[12] D. Yan, J. Cheng, M. T. Özsu, F. Yang, Y. Lu, J. C. S. Lui, Q. Zhang, and W. Ng. A general-purpose query-centric framework for querying big graphs. *PVLDB*, 9(7):564–575, 2016.