A General-Purpose Query-Centric Framework for Querying Big Graphs

Da Yan¹, James Cheng², M. Tamer Özsu³, Fan Yang⁴, Yi Liu⁵, John C. S. Lui⁶, Qizhen Zhang⁷, Wilfred Ng⁸

¹Department of Computer Science and Engineering, The Chinese University of Hong Kong
²jcheng, ³fyang, ⁴ylu, ⁵cslui, ⁷qzzhang}@cse.cuhk.edu.hk
³David R. Cheriton School of Computer Science, University of Waterloo
⁴tozs@cwaterloo.ca
⁺Department of Computer Science and Engineering, The Hong Kong University of Science and Technology
⁸wilfred@cse.ust.hk

ABSTRACT

Pioneered by Google’s Pregel, many distributed systems have been developed for large-scale graph analytics. These systems employ a user-friendly “think like a vertex” programming model, and exhibit good scalability for tasks where the majority of graph vertices participate in computation. However, the design of these systems can seriously under-utilize the resources in a cluster for processing light-workload graph queries, where only a small fraction of vertices need to be accessed. In this work, we develop a new open-source system, called Quegel, for querying big graphs. Quegel treats queries as first-class citizens in its design: users only need to specify the Pregel-like algorithm for a generic query, and Quegel processes light-workload graph queries on demand, using a novel superstep-sharing execution model to effectively utilize the cluster resources. Quegel further provides a convenient interface for constructing graph indexes, which significantly improve query performance but are not supported by existing graph-parallel systems. Our experiments verified that Quegel is highly efficient in answering various types of graph queries and is up to orders of magnitude faster than existing systems.

1. INTRODUCTION

Big graphs are common in real-life applications today, for example, online social networks and mobile communication networks have billions of users, and web graphs and Semantic webs can be even bigger. Processing such big graphs typically require a special infrastructure, and the most popular ones are Pregel [17] and Pregel-like systems [1, 7, 8, 15, 21, 26]. In a Pregel-like system, a programmer thinks like a vertex and only needs to specify the behavior of one vertex, and the system automatically schedules the execution of the specified computing logic on all vertices. The system also handles fault tolerance and scales out without extra effort from programmers.

Existing Pregel-like systems, however, are designed for heavy-weight graph computation (i.e., analytic workloads), where the majority part of a graph or the entire graph is accessed [28]. For example, Pregel’s PageRank algorithm [17] accesses the whole graph in each iteration. However, many real-world applications involve various types of graph querying, whose computation is light-weight in the sense that only a small portion of the input graph needs to be accessed. For example, in our collaboration with researchers from one of the world’s largest online shopping platforms, we have seen huge demands for querying different aspects of big graphs for all sorts of analysis to boost sales and improve customer experience. In particular, they need to frequently examine the shortest-path distance between some users in a large network extracted from their online shopping data. While Pregel’s single-source shortest-path (SSSP) algorithm [17] can be applied here, much of the computation will be wasted because only those paths between the queried users are of interest. Instead, it is much more efficient to apply point-to-point shortest-path (PPSP) queries, which only traverse a small part of the input graph. We also worked with a large telecom operator, and our experience is that graph queries (with light-weight workloads) are integral parts of analyzing massive mobile phone and SMS networks.

The importance of querying big graphs has also been recognized in some recent work [12], where two kinds of systems are identified: (1) systems for offline graph analytics (such as Pregel and GraphLab) and (2) systems for online graph querying, including Horton [22], G-SPARQL [20] and Trinity [23]. However, Horton and G-SPARQL are tailor-made only for specific types of queries. Trinity supports graph query processing, but compared with Pregel, its main advantage is that it keeps the input graph in main memories so that the graph does not have to be re-loaded for each query. The Trinity paper [23] also argues that indexing is too expensive for big graphs and thus Trinity does not support indexing. In the VLDB 2015 conference, there is also a workshop “Big-O(Q): Big Graphs Online Querying”, but the works presented there only study algorithms for specific types of queries. So far, there lacks a general-purpose framework that allows users to easily design distributed algorithms for answering various types of queries on big graphs.

One may, of course, use existing vertex-centric systems to process queries on big graphs, but these systems are not suitable for processing light-weight graph queries. To illustrate, consider processing PPSP queries on a 1.96-billion-edge Twitter graph used in our experiments. To answer one query \((s, t)\) by bidirectional breadth-first search (BiBFS) in our cluster, Giraph takes over 100 seconds, which is intolerable for a data analyst who wants to ex-
amine the distance between users in an online social network with short response time. To process queries on demand using an existing vertex-centric system, a user has the following two options: (1) to process queries one after another, which leads to a low throughput since the communication workload of each query is usually too light to fully utilize the network bandwidth and many synchronization barriers are incurred; or (2) to write a program to explicitly process a batch of queries in parallel, which is not easy for users and may not fully utilize the network bandwidth towards the end of the processing, since most queries may have finished their processing and only a small number of queries are still being processed. It is also not clear how to use graph indexing for query processing in existing vertex-centric systems.

To address the limitations of existing systems in querying big graphs, we developed a distributed system, called Quegel, for large-scale graph querying. We implemented the Hub-Labeling approach [11] in Quegel, and it can achieve interactive speeds for PPSp querying on the same Twitter graph mentioned above. Quegel 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 has a wide application scope, since any query that can be processed by a Pregel-style vertex-centric algorithm can be answered by Quegel, and much more efficiently. Under this query-centric design, Quegel adopts a novel superstep-sharing execution model to effectively utilize the cluster resources, and an efficient mechanism for managing vertex states that significantly reduces memory consumption. Quegel further provides a convenient interface for constructing indexes to 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. We have successfully applied Quegel to process five important types of graph queries (to be presented in Section 5), and Quegel achieves performance up to orders of magnitude faster than existing systems.

The rest of this paper is organized as follows. We review related work in Section 2. In Section 3, we highlight important concepts in the design of Quegel, and key implementation issues. We introduce the programming model of Quegel in Section 4, and describe some graph querying problems as well as their Quegel algorithms in Section 5. Finally, we evaluate the performance of Quegel in Section 6 and conclude the paper in Section 7.

2. RELATED WORK

We first review existing vertex-centric graph-parallel systems. We consider an input graph $G = (V, E)$ stored on Hadoop distributed file system (HDFS), where each vertex $v \in V$ is associated with its adjacency list (i.e., $v$’s neighbors). If $G$ is undirected, we denote $v$’s neighbors by $\Gamma(v)$. If $G$ is directed, we denote $v$’s in-neighbors and out-neighbors by $\Gamma_{in}(v)$ and $\Gamma_{out}(v)$, respectively. Each vertex $v$ also has a value $a(v)$ storing $v$’s vertex value. Graph computation is run on a cluster of workers, where each worker is a computing thread/process, and a machine may run multiple workers.

Pregel [17]. Pregel adopts the bulk synchronous parallel (BSP) model. It distributes vertices to workers in a cluster, where each vertex is associated with its adjacency list. A Pregel program computes in iterations, where each iteration is called a superstep. Pregel requires users to specify a user-defined function (UDF) $\text{compute}()$. In each superstep, each active vertex $v$ calls $\text{compute}(msgs)$, where $msgs$ is the set of incoming messages sent from other vertices in the previous superstep. In $v.\text{compute}(msgs)$, $v$ may process $msgs$ and update $a(v)$, send new messages to other vertices, and vote to halt (i.e., deactivate itself). A halted vertex is reactivated if it receives a message in a subsequent superstep. The program terminates when all vertices are deactivated and no new message is generated. Finally, the results (e.g., $a(v)$) are dumped to HDFS.

Pregel also allows users to implement an aggregator for global communication. Each vertex can provide a value to an aggregator in $\text{compute}()$ in a superstep. The system aggregates those values and makes the aggregated result available to all vertices in the next superstep.

Distributed Vertex-Centric Systems. Many Pregel-like systems have been developed, including Giraph [1], GPS [21], GraphX [8], and Pregel+ [26]. New features are introduced by these systems. For example, GPS proposed to mirror high-degree vertices on other machines, and Pregel+ proposed the integration mirroring and message combining as well as a request-respond mechanism, to reduce communication workload. While these systems strictly follow the synchronous data-pushing model of Pregel, GraphLab [15] adopts an asynchronous data-pulling model, where each vertex actively pulls data from its neighbors rather than passively receives messages. A subsequent version of GraphLab, called PowerGraph [7], partitions the graph by edges rather than by vertices to achieve more balanced workload. While the asynchronous model leads to faster convergence for some tasks like random walk, [16] and [9] reported that GraphLab’s asynchronous mode is generally slower than synchronous execution mainly due to the expensive cost of locking/unlocking.

Single-PC Vertex-Centric Systems. There are also other vertex-centric systems, such as GraphChi [13] and X-Stream [19], designed to run on a single PC by manipulating a big graph on disk. However, these systems need to scan the whole graph on disk once for each iteration of computation even if only a small fraction of vertices need to perform computation, which is inefficient for lightweight querying workloads.

Weaknesses of Existing Systems for Graph Querying. In our experience of working with researchers in e-commerce companies and telecom operators, we found that existing vertex-centric systems cannot support query processing efficiently nor do they provide a user-friendly programming interface to do so. If we write a vertex-centric algorithm for a generic query, we have to run a job for every incoming query. As a result, each superstep transmits only the few messages of one light-weight query which cannot fully utilize the network bandwidth. Moreover, there are a lot of synchronization barriers, one for each superstep of each query, which is costly. Furthermore, some systems such as Giraph bind graph loading with graph computation (i.e., processing a query in our context) for each job, and the loading time can significantly degrade the performance.

An alternative to the one-query-at-a-time approach is to hard code a vertex-centric algorithm to process a batch of $k$ queries, where $k$ can be an input argument. However, in the $\text{compute}()$ function, one has to differentiate the incoming messages and/or aggregators of different queries and update $k$ vertex values accordingly. In addition, existing vertex-centric framework checks the stop condition for the whole job, and users need to take care of additional details such as when a vertex can be deactivated (e.g., when it should be halted for all the $k$ queries), which should originally be handled by the system itself. More critically, the one-batch-at-a-time approach does not solve the problem of low utilization of network bandwidth, since in later stage when most queries finish their processing, only a small number of queries (or stragglers) are still being processed and hence the number of messages generated is too small to sufficiently utilize the network bandwidth.
3. THE QUEGEL SYSTEM

The system architecture of Quegel is shown in Figure 1. Quegel runs two types of programs: a server program that loads an input graph and processes incoming graph queries, and an arbitrary number of client programs for users to submit their queries to the server program. The server program consists of a master and a cluster of workers. The master receives incoming queries and appends them to a query queue. The master then fetches queries from the query queue to be processed, and maintains the information of these queries in a query table $HT_Q$. The queries in the query queue are fetched into $HT_Q$ at each communication barrier to start their evaluation, and the new content of $HT_Q$ is synchronized to all workers, as indicated by "query sync" in Figure 1. Meanwhile, vertices on different workers send messages to each other to process the queries, as indicated by "msg sync".

The server program of Quegel is deployed with Hadoop Distributed File System (HDFS). Before query processing starts, the server program first loads an input graph $G$ from HDFS, i.e., distributing vertices to main memories of different workers in the cluster. If indexing is enabled, a local index will be built from the vertices of each worker. After $G$ is loaded (and index is constructed), the server program receives and processes incoming queries using the computing logic specified by a vertex UDF $\text{compute}(\cdot)$ as in Pregel. Users may type their queries from a client console, or submit a batch of queries with a file. After a query is evaluated, users may specify Quegel to print the answer to the console, or to dump the answer to HDFS if its size is large (e.g., when the answer contains many subgraphs).

3.1 Execution Model: Superstep-Sharing

Quegel uses a new execution model called superstep-sharing, which overcomes the weaknesses of existing systems presented in Section 2. We first present the hardness of querying a big graph in general, which influences the design of our model.

**Hardness of Big Graph Querying and Our Design Objective.** We consider the processing of a large graph that is stored across a cluster of machines. Due to the poor locality of graph data, each query usually accesses vertices spreading through the whole big graph in distributed sites, and thus the processing of each query requires network communication. Since the message transmission of each superstep incurs round-trip delay, it is difficult (if not unrealistic) for distributed vertex-centric computation to achieve response time comparable to that of single-machine algorithms on a small graph (e.g., in milliseconds). Therefore, our goal is to answer a query in interactive speed (e.g., in a second or several seconds). Moreover, due to the sheer size of a big graph, the total workload of a batch of queries can be huge even if each query accesses just a fraction of the graph. Therefore, it is difficult (if not unrealistic) to achieve both high throughput and low latency.

Due to the latency-throughput tradeoff, one can only expect either interactive speed or high throughput but not both, and thus, our design objective focuses on developing a model for the following two scenarios of querying big graphs, both of which are common in real life applications.

**Scenario (i): Interactive Querying**, where a user interacts with Quegel by submitting a query, checking the query results, refining the query based on the results and re-submitting the refined query, until the desired results are obtained. In such a scenario, there are usually only one user (e.g., a data scientist) or several users analyzing a big graph by posing interactive queries, but each query should be answered in a second or several seconds. No existing vertex-centric system can achieve such query latency on a big graph.

**Scenario (ii): Batch Querying**, where batches of queries are submitted to Quegel, and they need to be answered within a reasonable amount of time. An example of batch querying is given by the vertex-pair sampling application mentioned in Section 1 for estimating graph metrics, where a large number of PPSP queries need to be answered. Quegel achieves throughput 186 and 38.6 times higher than Giraph and GraphLab for processing PPSP queries, and thus allows the graph metrics to be estimated more accurately.

**Superstep-Sharing.** We propose a superstep-sharing execution model to meet the requirements of both interactive querying and batch querying. Specifically, Quegel processes graph queries in iterations called super-rounds. In a super-round, all queries that are being processed proceed their computation by one superstep (thus the name superstep-sharing), while from the perspective of an individual query, Quegel processes it superstep by superstep as in Pregel. For the processing of each query, supersteps are numbered. Different queries may have different superstep number in the same super-round, if they are submitted to Quegel at different time.

We illustrate the execution process of superstep-sharing by the example in Figure 2, where we have four queries $q_1$, $q_2$, $q_3$ and $q_4$ submitted to Quegel at different time. For simplicity, assume that the processing of each query takes 4 supersteps. As shown in Figure 2, at super-round 1, there is only $q_1$, and $q_1$ executes 1 superstep. At the beginning of super-round 2, there is still only $q_1$, and $q_1$ executes another superstep. At super-round 3, Quegel also received $q_2$ and $q_3$, and $q_1$, $q_2$, and $q_3$ all execute 1 superstep (note that $q_1$ executes the third superstep in its processing, while $q_2$ and $q_3$ execute the first superstep in their processing). At super-round 4,
Figure 2: Illustration of superstep-sharing

and 2 time units on Worker 2. When the queries are processed individually, the first query needs to be synchronized before the second query starts to be processed. Thus, 8 time units are required in total. Using superstep-sharing, only one synchronization is needed at the end of the super-round, thus requiring only 6 time units.

One issue that remains is how to set the capacity parameter $C$. Obviously, the larger the number of queries being concurrently processed, the more fully is the network bandwidth utilized. But the value of $C$ should be limited by the available RAM space. The input graph consumes $O(|V| + |E|)$ RAM space, while each query $q$ consumes $O(|V_q|)$ space, where $V_q$ denotes the set of vertices accessed by $q$. Thus, $O(|V| + |E| + C)$ should not exceed the available RAM space, though in most cases this is not a concern as $|V_q| \ll |V|$. While setting $C$ larger tends to improve the throughput, the throughput converges when the network bandwidth is saturated. In a cluster such as ours which is connected by Gigabit Ethernet, we found that the throughput usually converges when $C$ is increased to 8 (for the graph queries we tested), which indicates that Quegel has already fully utilized the network bandwidth.

3.2 System Design

Quegel manages three kinds of data: (i) V-data, whose value only depends on a vertex $v$, such as $v$'s adjacency list. (ii) VQ-data, whose value depends on both a vertex $v$ and a query $q$. For example, the vertex value $a(v)$ is query-dependent: in a PPSP query $q = (s,t)$, $a(v)$ keeps the estimated value of the shortest distance from $s$ to $v$, denoted by $d(s,v)$, whose value depends on the source vertex $s$. As $a(v)$ is w.r.t. a query $q$, we use $a(v,q)$ to denote $a(v)$ w.r.t. $q$. Other examples of VQ-data include the active/halted state $(s,t)$ for a PPSP query, the outgoing message buffer of $v$ (i.e., input to $v$.compute($s,t$)). (iii) Q-data, whose value only depends on a query $q$. For example, at any moment, each query $q$ has a unique superstep number. Other examples of Q-data include the query content (e.g., $(s,t)$ for a PPSP query), the outgoing message buffers, aggregated values, and control information that decides whether the computation should terminate.

Let $Q = \{q_1, \ldots, q_n\}$ be the set of queries currently being processed, and let $id(q_i)$ be the query ID of each $q_i \in Q$. Each worker of Quegel maintains a hash table $HT_Q$ to keep the Q-data of each query in $Q$ (see Figure 1). The Q-data of a query $q_i$ can be obtained from $HT_Q$ by providing the query ID $id(q_i)$, and we denote it by $HT_Q[q_i]$. When a new query $q$ is fetched from master's query queue to start its processing, the Q-data of $q$ is inserted into $HT_Q$; while after $q$ reports or dumps its results, the Q-data of $q$ is removed from $HT_Q$ of every worker.

Each worker $W$ also maintains an array of vertices, varray, each element of which maintains the V-data and VQ-data of a vertex $v$ that is distributed to $W$. The VQ-data of a vertex $v$ is organized by a look-up table $LU_T$, where the VQ-data related to a query $q_i$ can be obtained by providing the query ID $id(q_i)$, and we denote it by $LU_T[q_i]$. Since every vertex $v$ needs to maintain a table $LU_T$, we implement it using a space-efficient balanced binary search tree rather than a hash table. The data kept by each table entry $LU_T[v]$ include the vertex value $a_i(v)$, the active/halted state of $v$ (in $q$), and the incoming message buffer of $v$ (for $q$).
Unlike the one-shot-at-a-time approach with existing vertex-centric systems, where each vertex needs to maintain $k$ vertex values no matter whether it is accessed by a query, we design Quegel to be more space efficient. We require that a vertex $v$ is allocated a state for a query $q$ only if $q$ accesses $v$ during its processing, which is achieved by the following design. When vertex $v$ is activated for the first time during the processing of $q$, the VQ-data of $v$ is initialized and inserted into $LT_v$. After a query $q$ reports or dumps its results, the VQ-data of $q$ (i.e., $LT_v[q]$) is removed from $LT_v$ of every vertex $v$ in $G$ accessed by $q$.

Each worker also maintains a hash table $HT_v$, such that the position of a vertex element $v$ in $varray$ can be obtained by providing the vertex ID of $v$. We denote the obtained vertex element by $HT_v[v]$. The table $HT_v$ is useful in two places: (1) when a message targeted at vertex $v$ is received, the system will obtain the incoming message buffer of $v$ from $varray[pos]$ where $pos$ is computed as $HT_v[v]$, and then append the message to the buffer; (2) when an initial vertex $v$ is activated using its vertex ID at the beginning of a query $q$, the system will initialize the VQ-data of $v$ for $q$, and insert it into $LT_v$, which is obtained from $varray[pos]$ where $pos$ is computed as $HT_v[v]$. We shall see how users can activate the (usually small) initial set of vertices in Quegel for processing $q$ without scanning all vertices in Section 4.

An important feature of Quegel is that, it only requires a user to specify the computing logic for a generic vertex and a generic query; the processing of concrete queries is handled by Quegel and is totally transparent to users. For this purpose, each worker $W$ maintains two global context objects: (i) query context $C_{query}$, which keeps the Q-data of the query that $W$ is processing; and (ii) vertex context $C_{vertex}$, which keeps the VQ-data of the current vertex that $W$ is processing for the current query. In a superround, when a worker starts to process each query $q_i$, it first obtains $HT_v[q_i]$ and assigns it to $C_{query}$, so that when a user accesses the Q-data of the current query in UDF $compute()$ (e.g., to get the superstep number or to append messages to outgoing message buffers), the system will access $C_{query}$ directly without looking up from $HT_v$. Moreover, during the processing of $q_i$, and before the worker calls $compute()$ on each vertex $v$, it first obtains $LT_v[q_i]$ and assigns it to $C_{vertex}$, so that any access or update to the VQ-data of $v$ in $compute()$ (e.g., obtaining $a_0(v)$ or letting $v$ vote to halt) directly operates on $C_{vertex}$ without looking up from $LT_v$.

As an illustration, consider the example shown in Figure 4, where there are 3 queries being evaluated and the computation proceeds for 3 supersteps. Moreover, we assume that 4 vertices call $compute()$ in each superstep of each query. As an example, when processing a superstep $(i+2)$, $C_{query}$ is set to $HT_v[q_i]$ before evaluating $v_1$ for $q_3$; and when the evaluation arrives at $v_3$, $C_{vertex}$ is set to $LT_v[q_i]$ before $v_3.compute()$ is called. Figure 4 also shows a simplified code of $compute()$ for shortest path computation, and inside $v_3.compute()$ for $q_3$, $a(v)$ is accessed once in Line 1 and twice in Line 3, all of which use the value $a_0(v)$ stored in $C_{vertex} = LT_v[q_i]$ directly; while Line 1 accesses the superstep number which is obtained from $C_{query} = HT_v[q_i]$ directly.

One benefit of using the context objects $C_{vertex}$ and $C_{query}$ is that, due to the access pattern locality of superstep-sharing, repetitive lookups of tables $HT_v$ and $LT_v$ are avoided. Another benefit is that, users can write their program exactly like in Pregel (e.g., to access $a(v)$ and superstep number) and the processing of concrete queries is transparent to users.

### 4. PROGRAMMING INTERFACE

The programming interface of Quegel incorporates many unique features designed for querying workload. For example, the interface allows users to construct distributed graph indexes at graph loading. The interface also allows users to activate only an initial (usually small) set of vertices, denoted by $V_0$, for processing a query $q$ without checking all vertices. Note that we cannot activate $V_0$ during graph loading because $V_0$ depends on each incoming query $q$.

Quegel defines a set of base classes, each of which is associated with some template arguments. To write an application program, a user only needs to (1) subclass the base classes with the template arguments properly specified, and to (2) implement the UDFs according to the application logic. We now describe these base classes.

**Vertex Class.** Figure 5 summarizes the key API of the Vertex class. As in Pregel, the Vertex class has a UDF $compute()$ for users to specify the computing logic. In $compute()$, a user may call $get_query()$ to obtain the content of the current query $q_{cur}$. A user may also access other Q-data in $compute()$, such as getting $q_{cur}$’s superstep number, sending messages (which appends messages to $q_{cur}$’s outgoing message buffers), and getting $q_{cur}$’s aggregated value from the previous superstep. Quegel also allows a vertex to call $force_terminate()$ to terminate the computation of $q_{cur}$ at the end of the current superstep. All these operations access the Q-data fields from $C_{query}$ directly.

The vertex class of Quegel is defined as $Vertex<\langle I, \sum V^i, V^V \cdot M, Q \rangle>$, which has five template arguments: (1) $\langle I \rangle$ specifies the type (e.g., int) of the ID of a vertex (which is V-data). (2) $\langle V^0 \rangle$ specifies the type of the query-dependent attribute of a vertex $v$, i.e., $a_0(v)$ (which is VQ-data). (3) $\langle V^V \rangle$ specifies the type of the query-independent attribute of a vertex $v$, denoted by $a^V(v)$ (which is V-data). We do not hard-code the adjacency list structure in order to provide more flexibility. For example, a user may define $a^V(v)$ to include two adjacency lists, one for in-neighbors and the other for out-neighbors, which is useful for algorithms such as bidirectional BFS. Other V-data can also be included in $a^V(v)$, such as vertex labels used for search space pruning in some query processing algorithms. (4) $\langle M \rangle$ specifies the type of the messages that are exchanged between vertices. (5) $\langle Q \rangle$ specifies the type of the content of a query. For example, for a PPSP query, $\langle Q \rangle$ is a pair of vertex IDs indicating the source and target vertices. In $compute()$, a user may access $a^V(v)$ by calling $value()$, and access $a_0(v)$ by calling $queryvalue()$. 

<table>
<thead>
<tr>
<th>$C_{query}$</th>
<th>Superstep $i$</th>
<th>Superstep $(i + 1)$</th>
<th>Superstep $(i + 2)$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>$C_{vertex}$</td>
<td>2 3 4 1 2 3 4</td>
<td>1 2 3 4 1 2 3 4</td>
<td>1 2 3 4 1 2 3 4</td>
</tr>
</tbody>
</table>

Figure 4: Illustration of context objects

One benefit of using the context objects $C_{vertex}$ and $C_{query}$ is that, due to the access pattern locality of superstep-sharing, repetitive lookups of tables $HT_v$ and $LT_v$ are avoided. Another benefit is that, users can write their program exactly like in Pregel (e.g., to access $a(v)$ and superstep number) and the processing of concrete queries is transparent to users.
Suppose that a set of $k$ queries, $Q$, is being processed, then each vertex conceptually has $k$ query-dependent attributes $a_q(v)$, one for each query $q \in Q$. Since a query normally only accesses a small fraction of the vertices, to be space-efficient, Quegel allocates space to $a_q(v)$ as well as other VQ-data only at the time when the vertex is first accessed during the processing of $q$. Accordingly, Quegel provides a UDF $init_{value}(q)$ for users to specify how to initialize $a_q(v)$ when a vertex is first accessed by $q$. For example, for a PPSP query $q = (s,t)$, where $a_q(v)$ keeps the estimated value of $d(s,v)$, one may implement $init_{value}(s,t)$ as follows: if $v = s$, $a_q(v) \leftarrow 0$; else, $a_q(v) \leftarrow \infty$. The state of $v$ is always initialized to be active by the system, since when the space of the state is allocated, $v$ is activated for the first time and should participate in the processing of $q$ in the current superstep. Function $init_{value}(q)$ is the only UDF of the Vertex class in addition to $compute()$.

**Worker Class.** The Vertex class presented above is mainly for users to specify the graph computation logic. Quegel provides another base class, Worker<$T_{es}$, $T_{idx}$>, for specifying the input/output format and for executing the computation of each worker. The template argument $<T_{es}>$ specifies the user-defined subclass of Vertex. The template argument $<T_{idx}>$ is optional, and if distributed indexing (to be introduced shortly) is enabled, $<T_{idx}>$ specifies the user-defined index class.

Figure 6 summarizes the key API of the Worker class. The Worker class has a function run(param), which implements the execution procedure of Quegel as described at the beginning of Section 3. After users define their subclasses to implement the computing logic, they can call run() to start a Quegel job. Here, param specifies job parameters such as the HDFS path of the input graph $G$. During the execution, we allow each query to change $\delta v(v)$ of a vertex $v$, and when a user closes the Quegel program from the console, he/she may specify Quegel to save the changed graph (V-data only) to HDFS, before freeing the memory space consumed by $G$.

The Worker class has four formatting UDFs, which are used (1) to specify how to parse a line of the input file into a vertex of $G$ in main memory, (2) to specify how to parse a query string (input by a user from the console or a file) into the query content of type $<Q>$, (3) to specify how to write the information of a vertex $v$ (e.g., $a_q(v)$) to HDFS after a query is answered, and (4) to specify how to write the changed V-data of a vertex $v$ to HDFS when a Quegel job terminates. The last UDF is optional, and is only useful if users enable the end-of-job graph dumping.

Quegel allows each worker to construct a local index from its loaded vertices before query processing begins. We illustrate this process by considering a vertex-labeled graph $G$ where each vertex $v$ contains text $\Psi(v)$, and show how to construct an inverted index on each worker $W$, so that given a keyword $k$, it returns a list of vertices on $W$ whose text contains $k$. This kind of index is useful in XML keyword search, subgraph pattern matching, and graph keyword search. Specifically, recall that each worker in Quegel maintains its vertices in an array varray. If indexing is enabled, a UDF $load2Idx(v, pos)$ will be called to process each vertex $v$ in varray immediately after graph loading, where $pos$ is $v$’s position in varray. To construct inverted indexes in Quegel, a user may specify $<T_{idx}>$ as a user-defined inverted index class, and implement $load2Idx(v, pos)$ to add $pos$ to the inverted list of each keyword $k$ in $\Psi(v)$. There are also indices that cannot be constructed simply from local vertices, and we shall see how to handle such an application in Quegel in Section 5.1.

When a query is first scheduled for processing, each worker calls a UDF $init_{activate}()$ to activate only the relevant vertices specified by users. For example, in a PPSP query $(s,t)$, only $s$ and $t$ are activated initially; while for querying a vertex-labeled graph, only those vertices whose text contains at least one keyword in the query are activated. Inside $init_{activate}()$, one may call $getVpos(vertexID)$ to get the position $pos$ of a vertex in varray (which actually looks up the hash table $HT_v$ of each worker), and then call $activate(pos)$ to activate the vertex. For example, to activate $s$ in aPPSP query $(s,t)$, a user may specify $init_{activate}()$ to first call $getVpos(s)$ to return $s$’s position $pos$. If $s$ is on the current worker, $pos$ will be returned and one may then call $activate(pos)$ to activate $s$ in $init_{activate}()$. Otherwise, $getVpos(s)$ returns -1 and no action needs to be performed in $init_{activate}()$. For querying a vertex-labeled graph, in $init_{activate}()$ one may first get the positions of the keyword-matched vertices from the inverted index, and then activate them with $activate(pos)$.

**Other Base Classes.** Quegel also provides other base classes such as Combiner and Aggregator, for which users can subclass them to specify the logic of message combiner [17] and aggregator [17].
although our algorithms can be easily modified to output the actual shortest path(s) by recording the previous in-neighbor of each vertex on its shortest path.

5.1.1 Algorithms without Indexing

Breadth-First Search (BFS). The simplest way of answering a PPSP query \( q = (s,t) \) is to perform BFS from \( s \), until the search reaches \( t \). In this algorithm, \( a_q(v) \) is specified as the current estimation of \( d(s,v) \). UDF Worker: \( \text{init}\_activate() \) should activate \( s \) before starting to process \( q \), and when a vertex \( v \) is first activated by \( q \), UDF \( v\text{-init}\_value(s,t) \) should set \( d(s,v) \) to 0 if \( v = s \), and to \( \infty \) otherwise. UDF \( v\text{-compute}() \) is implemented as follows. Let \( step_p \) be the superstep number of \( q \). If \( step_p = 1 \), then \( v \) is \( s \), and \( v \) broadcasts messages to its out-neighbors to activate them before voting to halt. If \( step_p > 1 \), Case 1: if \( d(s,v) = \infty \), then \( v \) is visited for the first time, and it sets \( d(s,v) \leftarrow step_p - 1 \), activates its out-neighbors by sending messages and votes to halt; moreover, if \( v = t \), \( v \) calls force_terminate() since \( d(s,t) \) has been computed; Case 2: if \( d(s,v) \neq \infty \), then \( v \) has been activated by \( q \) before, and hence \( v \) votes to halt directly. Finally, only \( t \) reports \( a_q(t) = d(s,t) \) on the console and nothing is dumped to HDPS.

Bidirectional BFS (BiBFS). A more efficient algorithm is to perform forward (resp., backward) BFS from \( s \) (resp., \( t \)) until a vertex \( v \) is visited in both directions (or, bi-reached). Let \( C \) be the set of bi-reached vertices when BiBFS stops, then \( d(s,v) \) is given by \( \min_{v \in C} \{ d(s,v) + d(v,t) \} \). We take the minimum since when BiBFS stops at iteration \( i \), \( d(s,v) + d(v,t) \) for a vertex \( v \in C \) may be either \((2i - 1) \) or \( 2i \).

The Queuel algorithm for BiBFS is similar to that for BFS, with the following differences. Now, \( a_q(v) \) keeps a pair \( (d(s,v),d(v,t)) \). UDF \( \text{init}\_value(s,t) \) sets \( d(s,v) \) (resp., \( d(v,t) \)) to 0 if \( v = s \) (resp., \( v = t \)), and to \( \infty \) otherwise. Both \( s \) and \( t \) are activated by \( \text{init}\_activate() \) initially, and two types of messages are used in order to perform forward BFS and backward BFS in parallel without interfering with each other. In \( v\text{-compute}() \), if both \( d(s,v) \neq \infty \) and \( d(v,t) \neq \infty \), \( v \) should call force_terminate() since \( v \) is bi-reached. Then, an aggregator is used to collect the distance \( d(s,v) + d(v,t) \) of each \( v \in C \), and to obtain the smallest one as \( d(s,t) \) for reporting. To terminate earlier when \( s \) cannot reach \( t \), we also use aggregator to compute the frequencies of messages sent in both directions in each superstep, respectively. If the number of messages sent in either direction is 0, the aggregator calls force_terminate() and reports \( d(s,t) = \infty \).

5.1.2 Hub²: An Algorithm with Indexing

Many real graphs exhibit skewed degree distribution, where some vertices (e.g., celebrities in a social network) connect to a large number of other vertices. We call such vertices as hubs. During BFS, visiting a hub results in visiting a large number of vertices. We now consider how to pre-compute \( L(v) \) (see Appendix G) for each vertex \( v \). The above algorithm requires that each vertex \( v \) stores \( L(v) \) in \( a_q(v) \). We now consider how to pre-compute \( L(v) \) in Queuel. This indexing procedure can be accomplished by performing \( [H] \) BFS operations, each starting from a hub \( h \in H \). Interestingly, if we regard each BFS operation from a hub \( h \) as a BFS query \( (h) \) in Queuel, then the entire procedure can be formulated as an independent Queuel job with the query set \{ \( (h) \mid h \in H \) \}. We process a BFS query \((h)\) in Queuel as follows. The query-dependent attribute of a vertex \( v \) is defined as \( d_\text{query}(v) = (d(h,v), \text{prev}(v)) \), where \( \text{prev}(v) \) is a flag indicating whether any shortest path from \( h \) to \( v \) passes through another hub \( h'/h \neq h \) or \( h' \neq v \). Queuel starts processing \((h)\) by calling \( \text{init}\_activate() \) to activate \( h \). The UDF \( v\text{-init}\_value(h) \) is specified to set \( \text{prev}(v) \leftarrow \text{FALSE} \), and to set \( d(h,v) \leftarrow 0 \) if \( v = h \) or set \( d(h,v) \leftarrow \infty \) otherwise. The UDF \( v\text{-compute}() \) is implemented as follows. In this algorithm, a message sent by \( v \) indicates whether there exists a shortest path from \( h \) to \( v \) that contains another hub \( h'/h \neq h \) (here, \( h' \) can be \( v \)); if so, for any vertex \( u \in H \) newly activated by that message, it holds that \( h \neq H \). Based on this idea, the algorithm is given as follows. In superstep \( i \), if \( a_q(v) = (d(h,v), \text{prev}(v)) \), where \( \text{prev}(v) \) is a flag indicating whether any shortest path from \( h \) to \( v \) that contains another hub \( h'/h \neq h \) (here, \( h' \) can be \( v \)) if so, for any vertex \( u \neq H \) newly activated by that message, it holds that \( h \neq H \). Based on this idea, the algorithm is given as follows. In superstep \( i \), if \( a_q(v) = (d(h,v), \text{prev}(v)) \), where \( \text{prev}(v) \) is a flag indicating whether any shortest path from \( h \) to \( v \) that contains another hub \( h'/h \neq h \) (here, \( h' \) can be \( v \)) if so, for any vertex \( u \neq H \) newly activated by that message, it holds that \( h \neq H \). Based on this idea, the algorithm is given as follows. In superstep \( i \), if \( a_q(v) = (d(h,v), \text{prev}(v)) \), where \( \text{prev}(v) \) is a flag indicating whether any shortest path from \( h \) to \( v \) that contains another hub \( h'/h \neq h \) (here, \( h' \) can be \( v \)) if so, for any vertex \( u \neq H \) newly activated by that message, it holds that \( h \neq H \). Based on this idea, the algorithm is given as follows.
its neighbors. Finally, \(v\) votes to halt.

To compute \(L(v)\) using the above algorithm, we specify the query-independent attribute of a vertex \(v\) as \(a^1(v) = (G(v), L(v))\), where \(L(v)\) is initially empty. After a query \(h\) is processed, we perform the following operation in the query dumping UDF: (i) if \(v \notin H\), \(v\) adds \((h, d(h, v))\) to \(L(v)\) only if \(\text{pre}_H(v) = \text{FALSE}\); (ii) if \(v \in H\), \(v\) always adds \((h, d(h, v))\) to \(L(v)\).

After all the \(H\) queries are processed, \(L(v)\) is fully computed for each \(v \in V\). Then, each vertex \(v\) saves \(L(v)\) along with other V-data to HDFs, which is to be loaded later by the Quegel program for processing PPSP queries described previously.

Extension to Directed Graphs. If \(G\) is directed, we make the following changes. First, each vertex \(v\) now has in-degree \(\Gamma_{\text{in}}(v)\) and out-degree \(\Gamma_{\text{out}}(v)\), and thus we consider three different ways of picking hubs, i.e., picking those vertices with the highest (i) in-degree, or (ii) out-degree, or (iii) sum of in-degree and out-degree.

Second, each vertex \(v\) now maintains two core-hub sets: an entry-hub set \(H_v^\text{in}\) and an exit-hub set \(H_v^\text{out}\). A hub \(h \in H\) is an entry-hub (resp., exit-hub) of \(v\) if \(\text{in}(h, v)\) exists (resp., \(\text{out}(h, v)\)) in any shortest path from \(v\) to \(h\) (resp., from \(h\) to \(v\)). Accordingly, we obtain two lists of hub-distance labels, \(L_v^\text{in}(v)\) and \(L_v^\text{out}(v)\). During indexing, we construct \(L_v^\text{in}(v)\) (resp., \(L_v^\text{out}(v)\)) by backward (resp., forward) BFS, i.e., sending messages to in-neighbors (out-neighbors). When answering PPSP queries, we compute \(d_{\text{hub}}\) similarly but \(h_i \in H_i\) (resp., \(h_i \in H_i^\text{out}\)) is now replaced by \(h_i \in H_i^\text{in}\) (resp., \(h_i \in H_i^\text{out}\)).

5.2 XML Keyword Search

Section 5.1 illustrated how graph indexing itself can be formulated as an individual Quegel program. We now present another application of Quegel, i.e., keyword search on XML documents, which makes use of the distributed indexing interface of Quegel described in Section 4 directly. Compared with traditional algorithms that rely on disk-based indexes [14, 32], our Quegel algorithms are much easier to program, and they avoid the expensive cost of constructing any disk-based index. Although simple MapReduce solution has also been developed, it takes around 15 seconds to process each keyword query on an XML document whose size is merely 200MB [30]. The low efficiency is because MapReduce is not designed for querying workload. In contrast, our Quegel program answers the same kind of keyword queries on much larger XML documents in less than a second. Let us first review the query semantics of XML keyword search, and then discuss XML keyword query processing in Quegel, followed by a real application of the query in an online shopping platform.

5.2.1 Query Semantics

An XML document can be regarded as a rooted tree, where internal vertices are XML tags and leaf vertices are texts. To illustrate, Figure 7 shows the tree structure of an XML document describing the information of a research lab. We denote the set of words contained in the tag or text of a vertex \(v\) by \(\psi(v)\), and if a keyword \(k \in \psi(v)\), we call \(v\) as a matching vertex of \(k\) (or, \(v\) matches \(k\)). Given an XML document modeled by a tree \(T\), an XML keyword query \(q = \{k_1, k_2, \ldots, k_n\}\) finds a set of trees, each of which is a fragment of \(T\), denoted by \(R\), such that for each keyword \(k_i \in q\), there exists a vertex \(v\) in \(R\) matching \(k_i\). We call each result tree \(R\) as a matching tree of \(q\).

Different semantics have been proposed to define what a meaningful matching tree \(R\) could be. Most semantics require that the root of \(R\) be the Lowest Common Ancestor (LCA) of \(q\). However, if \(q\) contains any keyword in its subtree, the subtree rooted at vertex \(v\) is not since it is an ancestor of another LCA, i.e. vertex 9. Let us denote the subtree of \(T\) rooted at vertex \(v\) by \(T_v\), then a vertex \(v\) is an LCA of \(q\) if \(T_v\) contains at least one occurrence of all keywords in \(q\), after pruning any subtree \(T_{u}\) (where \(u\) is a child of \(v\)) which already contains all keywords in \(q\). Referring to Figure 7 again, both vertices 1 and 9 are ELCA of \(q\) since after pruning any subtree rooted at vertex 6, there still exist vertices 3 and 5 and matching the keywords in \(q\). In contrast, if \(q = \{\text{Tom}, \text{Graph}\}\), then vertex 9 is an ELCA of \(q\), while vertex 1 is not an ELCA of \(q\) since after pruning the subtree rooted at vertex 6, there is no vertex matching \(\text{Tom}\).

Once the root, \(r\), of a matching tree is determined, we may return the whole subtree \(T_r\) as the result tree \(R\). However, if \(r\) is at a top level of the input XML tree, \(T_r\) can be large (e.g., the subtree rooted at vertex 1) and may contain much irrelevant information. For an SLCA, \(\text{MaxMatch}\) [14] was proposed to prune irrelevant parts from \(T_r\) to form \(R\). Let \(K(v)\) be the set of keywords matched by the vertices in \(T_v\). If a vertex \(v_1\) has a sibling \(v_2\), where \(K(v_1) \subseteq K(v_2)\), then \(T_v\) is pruned. For example, let \(q = \{\text{Tom}, \text{Graph}\}\) and consider the subtree rooted at vertex 1 in Figure 7. Since vertex 9 contains \{Tom, Graph\} in its subtree while its sibling vertex 14 does not contain any keyword in its subtree, the subtree rooted at vertex 14 is pruned.

5.2.2 Query Algorithms

Due to space limit, we only present the Quegel algorithms for computing SLCA, and the algorithms for computing ELCA and MaxMatch can be found in the full version of this paper [27]. The Quegel program first loads the graph that represents the XML document (the graph is obtained by parsing the XML document with a SAX parser), where each vertex \(v\) is associated with its parent \(\text{pa}(v)\) and its children \(\text{children}(v)\) (V-data). Then, each worker constructs an inverted index from the loaded vertices using the indexing interface described in Section 4.

To process a query \(q\), the UDF \text{init}_{activate}() activates only those vertices \(v\) with \(\psi(v) \cap q \neq \emptyset\). The query-dependent attribute of each vertex \(v, a^1(v)\), maintains \(\psi(v)\), \(\Gamma^1(v)\), and \(\psi(v)\), and the query-dependent attribute \(a^2(v)\) maintains a bitmap \(bm(v)\), where bit \(i\) (denoted by \(bm(v)[i]\)) equals 1 if keyword \(k_i\) exists in subtree \(T_v\) and 0 otherwise. The UDF \text{init}_{value}() sets each bit \(bm(v)[i]\) to 1 if \(k_i \in \psi(v)\) and 0 otherwise. For simplicity, if all the bits of \(bm(v)\) are 1, we call \(bm(v)\) as all-one. We now describe the query processing logic of \text{v.compute}() for computing SLCA as follows.
Computing SLCA in Quegel. In superstep 1, all matching vertices have been activated by \(\text{init} \_\text{activate}()\), and each matching vertex \(v\) sends \(bm(v)\) to its parent \(pa(v)\) and votes to halt. In superstep \(i\) \((i > 1)\), there are two cases in processing a vertex \(v\). Case (a): if some bit of \(bm(v)\) is 0, \(v\) computes the bitwise-OR of \(bm(v)\) and those bitmaps received from its children, which is denoted by \(bm_{\text{OR}}\). If \(bm_{\text{OR}} \neq bm(v)\), then some new bit of \(bm(v)\) should be set due to a newly matched keyword; thus, \(v\) sets \(bm(v) = bm_{\text{OR}}\), and sends the updated \(bm\), to its parent \(pa(v)\). In addition, if \(bm_{\text{OR}}\) is all-one, then (1) if \(v\) receives an all-one bitmap from a child, \(v\) is labeled as a non-SLCA (the label is also maintained in \(aq(v)\)); (2) otherwise, \(v\) is labeled as an SLCA. Case (b): if \(bm(v)\) is all-one, then \(v\) has been labeled either as an SLCA or as a non-SLCA (because a descendant is an SLCA) in an earlier superstep. (1) If \(v\) is labeled as a non-SLCA, \(v\) votes to halt directly; while (2) if \(v\) is labeled as an SLCA, and \(v\) receives an all-one bitmap from a child, then \(v\) labels itself as a non-SLCA. Finally, \(v\) votes to halt.

In the above algorithm, a vertex may send messages to its parent multiple times. To make sure that each vertex sends at most one message to its parent, we design another level-aligned algorithm as follows. Specifically, we pre-compute the level of each vertex \(v\) in the XML tree, denoted by \(\ell(v)\), by performing BFS from the tree root (with a traditional Pregel job). Then, our Quegel program loads the preprocessed data, where each vertex \(v\) also maintains \(\ell(v)\) in \(aq(v)\). The UDF \(\text{v.compute()}\) is designed as follows. Initially, we use an aggregator to collect the maximum level of all the matching vertices, denoted by \(\ell_{\text{max}}\). The aggregator maintains \(\ell_{\text{max}}\) and decrements it by one after each superstep. In a superstep, a vertex \(v\) at level \(\ell_{\text{max}}\) computes the bitwise-OR of \(bm(v)\) and all the bitmaps received from its children at level \((\ell_{\text{max}} + 1)\); the bitwise-OR is then assigned to \(bm(v)\) and sent to \(v\)'s parent \(pa(v)\). Moreover, if an all-one bitmap is received, \(v\) labels itself as a non-SLCA directly; otherwise, and if \(bm(v)\) becomes all-one, then \(v\) sends \(aq(v)\) to its parent \(pa(v)\).

Applications of XML Keyword Search. Though originally proposed for querying a single XML document [14, 32], our algorithms can also be used to query a large corpus of many XML documents. We illustrate this by one application in e-commerce. During online shopping, a customer may pose a keyword query (in the form of an AJAX request) from a web browser to search for interested products. The web server will obtain the matched products from the database, organize them as an XML document, and send it back to the client side. The browser of the client will then parse the XML document by a Javascript script to display the results. The server may log the various AJAX responses to disk, so that data scientists and sellers may pose XML keyword queries on the logged XML corpus to study customers’ search behaviors of specific products, to help them make better business decisions.

6. EXPERIMENTAL EVALUATION

We now evaluate the performance of Quegel. We only report the experimental results for answering PPSP and XML keyword queries, and the results of processing other types of queries can be found in the full version of this paper [27]. The source code of the Quegel system and algorithms of its five applications can be found in: http://www.cse.cuhk.edu.hk/quegel.

The experiments were conducted on a cluster of 15 machines, each with 24 cores (two Intel Xeon E5-2620 CPU) and 48GB RAM. The machines are connected by Gigabit Ethernet. In Quegel, each worker corresponds to one process that uses a core. We ran 8 workers per machine (i.e., 120 workers in total) for all the experiments of Quegel, since running more workers per machine does not significantly improve query performance due to the limited network bandwidth. We used HDFS of Hadoop 1.2.1 on the cluster.

Table 1(a) shows the datasets used in our experiments on PPSP queries: (i) Twitter [5]: Twitter who-follows-who network based on a snapshot taken in 2009; (ii) BTC [3]: a semantic graph converted from the Billion Triple Challenge 2009 RDF dataset; and (iii) a small dataset LiveJ [4] that refers to a bipartite network of LiveJournal users and their group memberships, which is used to demonstrate the poor scalability of some existing systems. Twitter is directed while BTC and LiveJ are undirected. Table 1(a) also shows the maximum and average vertex degree of each graph, and we can see that the degree distribution is highly skewed. We randomly generate vertex pairs \((s, t)\) on each dataset for running PPSP query processing, and the percentage of queries where \(s\) can reach \(t\) is shown in column “Reach Rate” of Table 1(a).

Comparison with Neo4j, GraphChi, and GraphX. We first compare Quegel with (1) Neo4j [18]: a well-known graph database; (2) GraphChi [13]: a single-machine graph processing system; and (3) GraphX [8]: a graph-parallel framework built on Spark (one of the most popular big data systems now). Neo4j and GraphChi were run on one of the machines in the cluster, while GraphX (shipped in Spark 1.4.1) was run on all the machines, using all cores and RAMs available.

These three systems have poor scalability for processing PPSP queries, as they either ran out of memory or are too time-consuming to process the two larger graphs, Twitter and BTC. We were only able to record the results for them on the smallest dataset, LiveJ. Table 2 reports the performance of these systems for answering 20 randomly-generated PPSP queries \((s, t)\) on LiveJ, where \(s\) cannot reach \(t\) in only three queries Q3, Q12, and Q15.

We adopt the BFS, BiBFS and \(H^2\) algorithms described in Section 5.1, and 1000 hubs were selected for \(H^2\). The results of Quegel are used for comparison with the other systems, to demonstrate why these systems are inefficient for graph querying. Quegel took 2912 seconds (end-to-end indexing time including graph loading/dumping) to compute the label set \(L(v)\) for every vertex \(v\). As Table 2 shows, Quegel answers every query in less than 3 seconds even by simply running BFS, while the time is consistently much less than one second when \(H^2\) is used. Quegel’s graph loading time for \(H^2\) is longer than for BFS and BiBFS since the program for \(H^2\) also needs to load \(L(v)\) of every vertex \(v\).

Since Neo4j provides a built-in “shortestPath” function, we directly use this function to answer the 20 PPSP queries. Neo4j spent over 17 hours just to import LiveJ, and the imported graph consumes 64GB disk space while LiveJ itself is only 1.1GB. In fact, Twitter could not be imported even after running for several days, and all disk space was used up when importing BTC. As Table 2 shows, the querying time is very unstable and can vary from a second to many hours, especially when \(s\) cannot reach \(t\). This is mainly because the built-in “shortestPath” function often needs to
we compared Quegel with Giraph 1.0.0 and GraphLab 2.2 by running-machine; and GraphX could not process small graphs sometimes even slower than the single-machine GraphChi system, Table 2 also shows that the performance of GraphX is similar and reach 7.9 which is too slow for interactive querying. Moreover, BFS is even faster than BiBFS and smallest graph list. Since GraphLab adopts an edge-centric model which keeps the graph in RAM, while GraphChi does not load the graph into RAM but keeps it on disk (thought it took 193 seconds to compute shards over 1 day) GraphX only finished 239 out of the 1000 BFS jobs, while GraphChi does not utilize RAM space efficiently and ran out of all RAM in our cluster when main-memory space was close to that for BFS and BiBFS, which is not as user-friendly as with Quegel’s API. It took GraphChi 129,508 seconds (1.5 days) to finish the 1000 BFS jobs, while after 90,390 seconds (over 1 day) GraphX only finished 239 out of the 1000 jobs. We can see that, even without considering the postprocessing time to obtain \( L(v) \), their indexing time is already orders of magnitude longer than that of Quegel.

Despite the difficulty in computing \( L(v) \) in GraphChi and GraphX, we tested the effect of \( Hub^2 \) for query processing in GraphChi and GraphX with the aid of Quegel. We first use Quegel to build \( Hub^2 \) and obtain the upperbound \( d_{ub} \) of each of the 20 queries, and then we directly hard-code \( d_{ub} \) as an additional input argument to each GraphChi and GraphX job in addition to the query \( (s,t) \). In this way, we can process each query in GraphChi and GraphX as if we use \( Hub^2 \) to obtain \( d_{ub} \) for the query.

GraphX took 6–9 seconds to load the graph into RAM, while GraphChi does not load the graph into RAM but keeps it on disk (thought it took 193 seconds to compute shards over LiveJ). GraphX’s loading time for \( Hub^2 \) is close to that for BFS and BiBFS, since \( d_{ub} \) is given as a query input and there is no need to load the list \( L(v) \) of any vertex \( v \). As shown in Table 2, both GraphChi and GraphX took tens to hundreds of seconds to answer a query on the smallest graph \( LiveJ \), which is too slow for interactive querying. Moreover, BFS is even faster than BiBFS and \( Hub^2 \) when it cannot reach \( t \), since BiBFS and \( Hub^2 \) additionally need to perform backward BFS from \( t \) to process vertices in \( t \)’s connected component. Table 2 also shows that the performance of GraphX is similar and sometimes even slower than the single-machine GraphChi system, although GraphX was run on 15 machines. GraphX also does not utilize RAM space efficiently: during the query processing on the small \( LiveJ \), GraphX already used more than half the RAM in each machine; and GraphX could not process BTC and Twitter as it ran out of all RAM in our cluster.

Comparison with Distributed Vertex-Centric Systems. Next, we compared Quegel with Giraph 1.0.0 and GraphLab 2.2 by running the PPSP algorithms of Section 5.1 on the two large graphs, Twitter and BTC. For each dataset, we randomly generated 1000 queries. Since Giraph and GraphLab both use multi-threading, they have access to all the 24 cores on each of the 15 machines.

Giraph needs to load the input graph \( G \) from HDFS for the evaluation of each query, and the high and dominating start-up overhead cancels any performance benefit of \( Hub^2 \) in query processing. Thus, we only ran BFS and BiBFS on Giraph, and only for the first 20 queries since it takes too long to finish all 1000 queries. GraphLab can keep \( G \) in main memory for repeated querying after \( G \) is loaded. However, its design is not suitable for \( Hub^2 \); keeping \( L(v) \) as an attribute of each vertex \( v \) takes excessive memory space, since GraphLab adopts an edge-centric model which makes duplicate copies of vertices. Moreover, during \( Hub^2 \) indexing, synchronizing \( L(v) \) among replicas of each vertex \( v \) is expensive. Thus, we simulated \( Hub^2 \) indexing in GraphLab by performing each hub-sourced BFS query and dumping its result in serial. In this way, a vertex \( v \) does not need to maintain \( L(v) \) during indexing, but postprocessing is needed to obtain \( L(v) \), which is not as user-friendly as with Quegel’s API.

Due to the above reasons, we also hard-code the upperbound \( d_{ub} \) (obtained from Quegel) as an additional input for each query in GraphLab (as what we did for GraphChi and GraphX), in order to test of the effect of \( Hub^2 \) in GraphLab. We will show that even with this direct hard-coding of \( d_{ub} \) in each query, GraphLab is still much slower than Quegel, which indicates that only our Quegel system is able to fully explore the performance benefit brought by \( Hub^2 \).

Table 3(a) and 3(b) report the cumulative time taken by Giraph, GraphLab and Quegel on Twitter and BTC for the first 20 queries. Here, Giraph’s loading time is contributed by all 20 queries, while that of GraphLab and Quegel is one-off before query processing begins. Also, we specify Quegel to report query results on the console and hence there is no dumping time. For all experiments of Quegel, we set the capacity parameter \( C = 8 \) unless otherwise stated. We
also show the average access rate of a query (i.e., the percentage of vertices accessed) in the last row “Access”.

Table 3(a) shows that Quegel is significantly faster than Giraph and GraphLab when bidirectional BFS is used to process the queries. When BFS is used, Quegel is still considerably faster than Giraph and GraphLab, but is 3 times longer than Quegel’s bidirectional BFS. This can be explained by the access rate of the queries, which actually demonstrates the effectiveness of Quegel’s specialized design for querying workload that accesses only a small portion of the graph. The result also reveals that the other systems are not suitable for graph querying, as Giraph’s loading time is already much longer than its actual querying time, while GraphLab’s querying time is unsatisfactory when the access rate is small. Also note that the loading time for BiBFS is usually longer than that for BFS, since each vertex \( v \) needs to load \( I_u(v) \) in addition to \( I_{out}(v) \).

Table 3(b) shows that the performance gap between Quegel and the other systems is significantly larger than on Twitter, since the access rate on BTC is much smaller than that on Twitter. This demonstrates that the design of Quegel is highly suitable for processing queries with small access rate. Another interesting observation is that, BFS has a smaller access rate than bidirectional BFS on BTC. This is because BTC consists of many connected components, and thus most queries \((s, t)\) are not reachable. For such a query, BFS terminates once all vertices in the connected component of \( s \) are visited, while bidirectional BFS also visits the vertices in \( t \)’s connected component, leading to a larger access rate.

**Effect of Indexing.** We next show that Hub\(^2\) indexing can significantly improve Quegel’s query performance. For Twitter, we chose hubs as the top-\( k \) vertices with the highest in-degree, out-degree, and their sum, and found that the results are similar. Thus, we only report the results for hubs with highest in-degree. Table 4(a) shows the Hub\(^2\) indexing time of GraphLab and Quegel when \( k = 1000 \). We can see that Quegel is faster than GraphLab, even though we do not consider the postprocessing cost of GraphLab to obtain \( L(v) \). We also show the results of Quegel when \( k = 100 \) to demonstrate the effect of \( k \). We can see from Table 4(a) that each hub-sourced BFS takes about 30 seconds on average in Quegel, and thus, \( k \) cannot be too large in order to keep the preprocessing time reasonable.

Table 4(b) reports the total time of processing the 1000 queries by Quegel and GraphLab. Clearly, when index is applied, Quegel’s performance is significantly improved, which can be explained by the reduction in the access rate. The result demonstrates the effectiveness of graph indexing. Quegel’s loading time with Hub\(^2\) is longer than BFS and BiBFS since it needs to load \( L(v) \) of every vertex \( v \), but graph loading is a one-off preprocessing operation and has no influence on subsequent query performance.

When \( k = 1000 \), Quegel processes 1000 queries in 339.6 seconds with Hub\(^2\), which is about 10 times faster than with BiBFS. In contrast, Hub\(^2\) in GraphLab is only slightly faster than BiBFS even though the access rate is much smaller, and it is 35 times slower than Hub\(^2\) in Quegel.

We also built Hub\(^2\) on BTC, using the top-1000 vertices with the highest degree. Table 5(a) shows the indexing time, where Quegel is 10 times faster than GraphLab. This is because BTC consists of many connected components and each BFS from a hub accesses only one component, and Quegel is able to benefit significantly from the small access rate while GraphLab cannot. Table 5(b) shows the time of processing the 1000 queries by Quegel and GraphLab. The result shows that Hub\(^2\) significantly improves the query performance, and Quegel can process over 7 PPSP queries per second on BTC (82 times faster than GraphLab).

**Effect of Capacity Parameter.** We next examine how the capacity parameter \( C \) influences the throughput of query processing in Quegel, by processing the first 512 queries with Hub\(^2\) \((k = 1000)\) on Twitter with different values of \( C \). Table 6(a) reports the total time of processing the 512 queries, where we can see that processing queries one by one (i.e., \( C = 1 \)) is significantly slower than when a larger capacity is used. For example, when \( C = 8 \), the total query processing time is only \( 1/3 \) of that when \( C = 1 \), which verifies the effectiveness of superstep-sharing. However, the query throughput does not increase when we further increase \( C \), since the cluster resources are already fully utilized. Similar results have also been observed on the other datasets. Note that Quegel took 538.5 seconds to process 512 queries on Twitter when \( C = 1 \), and thus the average time of processing a query is still around 1 second, which is similar to the response time on the small Livej dataset (see Table 2). This shows that the interactive querying performance of Quegel scales well to graph size.

**Performance on XML Keyword Search.** We evaluate the performance of processing XML keyword queries in Quegel on the DBLP dataset [2], as well as a larger XMark benchmark dataset [6]. Table 1(b) shows the data statistics, where \( |V| \) refers to the number of vertices in the XML tree, “Doc Size” refers to the file size of the raw XML document, and “Graph Size” refers to the file size of the parsed graph representation. Note that it is not clear how to implement our algorithms in other graph-parallel systems; for example, they do not provide a convenient way to construct and utilize inverted index. We generated 1000 keyword queries for each dataset, by randomly picking a query from a query pool each time, and repeating it for 1000 times. The query pools were obtained from existing work [24, 31, 32, 33], each contains tens of well-selected keyword queries.

---

**Table 4:** Cumulative time on Twitter \((1k\) PPSP queries\)

<table>
<thead>
<tr>
<th>GraphLab</th>
<th>Quegel</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Load</strong></td>
<td><strong>Save</strong></td>
</tr>
<tr>
<td>117.2</td>
<td>79.7</td>
</tr>
<tr>
<td>49.7</td>
<td>49.8</td>
</tr>
</tbody>
</table>

**Table 5:** Cumulative time on BTC \((1k\) PPSP queries\)

<table>
<thead>
<tr>
<th>GraphLab</th>
<th>Quegel</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Load</strong></td>
<td><strong>Save</strong></td>
</tr>
<tr>
<td>105.9</td>
<td>199.8</td>
</tr>
<tr>
<td>104.3</td>
<td>42.2</td>
</tr>
</tbody>
</table>

**Table 6:** Effect of \( C \) and machine # (second)

<table>
<thead>
<tr>
<th>( C )</th>
<th>( # C )</th>
<th># Mac</th>
<th># Index</th>
<th># Query</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>538.5</td>
<td>16</td>
<td>189.1</td>
<td>61466</td>
</tr>
<tr>
<td>2</td>
<td>210.7</td>
<td>32</td>
<td>190.1</td>
<td>45360</td>
</tr>
<tr>
<td>4</td>
<td>192.1</td>
<td>64</td>
<td>188.4</td>
<td>43007</td>
</tr>
<tr>
<td>8</td>
<td>187.3</td>
<td>128</td>
<td>187.1</td>
<td>35412</td>
</tr>
</tbody>
</table>

(a) Effect of \( C \) \((|C| = 512)\)

(b) Effect of Mac # \((|C| = 14)\)
Table 7: Results on XML keyword search

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Time</th>
<th>SLCA</th>
<th>Naïve</th>
<th>ELCA</th>
<th>MaxMatch</th>
</tr>
</thead>
<tbody>
<tr>
<td>DBLP</td>
<td>Load</td>
<td>17.2</td>
<td>33.1</td>
<td>33.7</td>
<td>26.3</td>
</tr>
<tr>
<td></td>
<td>Index</td>
<td>18.1</td>
<td>34.7</td>
<td>35.3</td>
<td>27.0</td>
</tr>
<tr>
<td></td>
<td>Access</td>
<td>0.35%</td>
<td>0.38%</td>
<td>0.36%</td>
<td>0.5%</td>
</tr>
<tr>
<td>XMark</td>
<td>Load</td>
<td>44.3</td>
<td>58.3</td>
<td>68.6</td>
<td>89.6</td>
</tr>
<tr>
<td></td>
<td>Index</td>
<td>47.7</td>
<td>61.9</td>
<td>72.2</td>
<td>93.1</td>
</tr>
<tr>
<td></td>
<td>Query</td>
<td>1550</td>
<td>1930</td>
<td>1986</td>
<td>5994</td>
</tr>
<tr>
<td></td>
<td>Access</td>
<td>5.9%</td>
<td>5.9%</td>
<td>5.9%</td>
<td>10.1%</td>
</tr>
</tbody>
</table>

Table 7 reports the performance of computing SLCA, ELCA and MaxMatch on the datasets using the 1000 queries. The reported metrics include the one-off graph loading and inverted index construction time, the total time of processing the 1000 queries (including result dumping), and the average access rate of a keyword query. The results verify that Quegel obtains good performance. For computing SCLas, the average processing time of each query is only 0.5–0.6 second on DBLP. The time is longer on XMark, which is because XMark queries are less selective, and thus their access rates are much higher than those of DBLP queries. Finally, we can observe that the level-aligned version of SLCA algorithm is more efficient than the simple one on DBLP, but it is slower on XMark. This is because, vertices at the upper levels of DBLP’s XML tree have high fan-outs, and thus the level-alignment technique significantly reduces the number of messages; while the vertex-fan-out in XMark is generally small, and the cost incurred by the aggregator out-weighs the benefit of message reduction.

7. CONCLUSIONS

We developed a distributed system, Quegel, for general-purpose querying of big graphs. To our knowledge, this is the first work that studies how to apply Pregel’s user-friendly vertex-centric programming interface to efficiently process queries in big graphs. This is also the first general-purpose system that applies graph indexing to speed up query processing in a distributed platform. In addition to the good performance obtained for answering PPS and XML keyword queries presented in this paper, Quegel has also been shown to achieve good performance for processing terrain shortest path queries, point-to-point reachability queries, and graph keyword queries. The details of using Quegel to process these queries and the experimental evaluation results can be found in the full version of this paper [27].

For future work, we will continue to improve the performance of Quegel by designing various optimization techniques, proposing efficient graph indexes to be adopted in Quegel, and developing more applications of Quegel.

Acknowledgments. We thank the reviewers for their valuable comments. The authors are partially supported by the Hong Kong GRF 2150851, the Natural Sciences and Engineering Research Council (NSERC) of Canada, and Grant no. HKUST-T12-403/11-2.

8. REFERENCES