GraphQ: Graph Query Processing with Abstraction Refinement

Scalable and Programmable Analytics over Very Large Graphs on a Single PC

Kai Wang Guoqing Xu
University of California, Irvine

Zhendong Su
University of California, Davis

Yu David Liu
SUNY at Binghamton

Abstract

This paper introduces *GraphQ*, a scalable querying framework for very large graphs. GraphQ is built on a key insight that many interesting graph properties — such as finding cliques of a certain size, or finding vertices with a certain page rank — can be effectively computed by exploring only a small fraction of the graph, and traversing the complete graph is an overkill. The centerpiece of our framework is the novel idea of *abstraction refinement*, where the very large graph is represented as multiple levels of abstractions, and a query is processed through iterative refinement across graph abstraction levels. As a result, GraphQ enjoys several distinctive traits unseen in existing graph processing systems: query processing is naturally *budget-aware*, friendly for *out-of-core processing* when "Big Graphs" cannot entirely fit into memory, and endowed with strong correctness properties on query answers. With GraphQ, a wide range of complex *analytical queries* over very large graphs can be answered with resources affordable to *a single PC*, which complies with the recent trend advocating single-machine-based Big Data processing.

Experiments show GraphQ can answer queries in graphs 4-6 times bigger than the memory capacity, only in several seconds to minutes. In contrast, GraphChi, a state-of-the-art graph processing system, takes hours to days to compute a whole-graph solution. An additional comparison with a modified version of GraphChi that terminates immediately when a query is answered shows that GraphQ is on average 1.6–13.4× faster due to its ability to process partial graphs.

1 Introduction

Developing scalable systems for efficient processing of very large graphs is a key challenge faced by Big Data developers and researchers. Given a graph analytical task expressed as a set of user-defined functions (UDF), existing processing systems compute a *complete solution* over the input graph. Despite much progress, computing a complete solution is still time-consuming. For example, using a 32-node cluster, it takes Preglix [5], a state-of-the-art graph processing system, more than 2,500 seconds to compute a complete solution (*i.e.*, all communities in the input graph) over a 70GB webgraph for a simple community detection algorithm.

While necessary in many cases, the computation of complete solutions — and the overhead of maintaining them — seems an overkill for many real-world applications. For example, queries such as "find one path between LA and NYC whose length is $\leq$ 3,000 miles" or "find 10 programmer communities in Southern California whose sizes are $\geq$ 1000" have many real-world usage scenarios *e.g.*, any path whose length is smaller than a threshold between two cities is acceptable for a navigation system. Unlike database queries that can be answered by filtering records, these queries need (iterative) computations over graph vertices and edges. In this paper, we refer to such queries as *analytical queries*. Furthermore, it appears that many of them can be answered by exploring only a *small fraction* of the input graph — if a solution can be found in a subgraph of the input graph, why do we have to exhaustively traverse the entire graph?

This paper is a quest driven by two simple questions: given the great number of real-world applications that need analytical queries, can we have a ground-up redesign of graph processing systems — from the programming model to the runtime engine — that can facilitate query answering over *partial graphs*, so that a client application can quickly obtain satisfactory results? If partial graphs are sufficient, can we answer analytical queries on one single PC so that the client can be satisfied without resorting to clusters?

GraphQ This paper presents *GraphQ*, a novel graph processing framework for analytical queries. In GraphQ, an analytical query has the form *"find n entities from the graph with a given quantitative property"*, which is general enough to express a large class of queries, such as page rank, single source shortest path, community detection, connected components, *etc.* A detailed discussion of GraphQ's answerability can be found in §3. At its core, GraphQ features two interconnected innovations:

- A simple yet expressive *partition-check-refine* programming model that naturally supports programmable analytical queries processed through *incremental* accesses to graph data
- A novel *abstraction refinement* algorithm to support efficient query processing, fundamentally decoupling the resource usage for graph processing from the (potentially massive) size of the graph

From the perspective of a GraphQ user, the very large input graph can be divided into *partitions*. How partitions are defined is programmable, and each partition on the high level can be viewed as a subgraph that GraphQ queries operate on. Query answering in GraphQ follows a repeated lock-step *check-refine* procedure, until either the query is answered or the budget is exhausted.

In particular, (1) the *check* phase aims to answer the query over each individual partition without considering inter-partition edges connecting these partitions. A query is successfully answered if a *check* predicate returns true; (2) if not, a *refine* process is triggered to identify a set of inter-partition edges to add back to the graph. These recovered edges will lead to a broader scope of partitions to assist query answering, and the execution loops back to step (1). Both the *check* procedure (determining whether the query is answered) and the *refine* procedure (determining what new inter-partition edges to include) are programmable, leading to a programming model suitable for defining complex analytical queries with significant in-graph computations.

Key to finding the most profitable inter-partition edges to add in each step is a novel *abstraction refinement* algorithm at the core of its query processing engine. Conceptually, the "Big Graph" under GraphQ is summarized into an *abstraction graph*, which can be intuitively viewed as a "summarization overlay" on top of the complete concrete graph (CG). The abstraction graph serves as a compact "navigation map" to guide the query processing algorithm to find profitable partitions for refinement.

Usage Scenarios We envision that GraphQ can be used in a variety of real-world data analytical applications. Example applications include:

- *Target marketing:* GraphQ can help a business quickly find a target group of customers with given properties;
- *Navigation:* GraphQ can help navigation systems quickly find paths with acceptable lengths
- *Memory-constrained data analytics:* GraphQ can provide good-enough answers for analytical applications with memory constraints

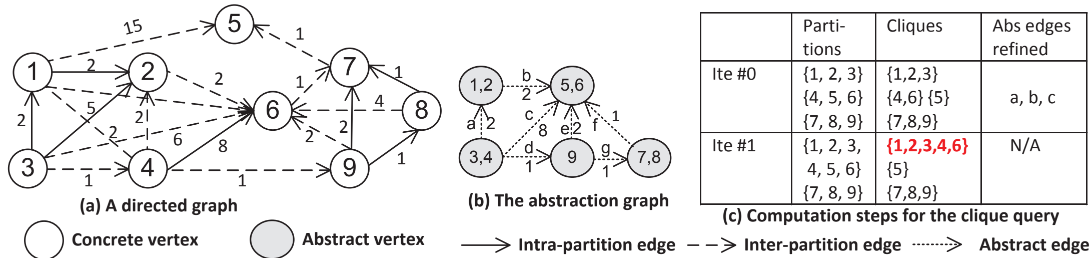
Contributions To the best of our knowledge, our technique is the first to borrow the idea of abstraction refinement from program analysis and verification [8] to process graphs, resulting in a query system that can quickly find correct answers in partial graphs. While there exists a body of work on graph query systems and graph databases (such as GraphChi-DB [17], Neo4j[1], and Titan[2]), the refinement-based query answering in GraphQ provides several unique features unseen in existing systems.

First, GraphQ reflects a ground-up redesign of graph processing systems in the era of "Big Data": unlike the predominant approach of graph querying where only simple graph analytics—those often involving SQL-like semantics where graph vertices/edges are filtered by meeting certain conditions or patterns [13, 14, 7], GraphQ has a strong and general notion of "answerability" which allows for a much wider range of analytical queries to be performed with flexible in-graph computation (*cf.* §3). Furthermore, the abstraction-guided search process makes it possible to answer a query by exploring the most relevant parts of the graph, while a graph database treats all vertices and edges uniformly and thus can be much less efficient.

Second, the idea of abstraction refinement in GraphQ provides a natural data organization and data movement strategy for designing efficient *out-of-core* Big Data systems. In particular, ignoring inter-partition edges (that are abstracted) allows GraphQ to load one partition at a time and perform vertex-centric computation on it independently of other partitions. The ability of exploring only a small fraction of the graph at a time enables GraphQ to answer queries over very large graphs *on one single PC*, thus in compliance with the recent trend that advocates single-machine-based Big Data processing [16, 23, 29, 17]. While our partitions are conceptually similar to GraphChi's shards (*cf.* §4), GraphQ does not need data from multiple partitions simultaneously, leading to significantly reduced random disk accesses compared to GraphChi's parallel sliding window (PSW) algorithm.

Third, GraphQ enjoys a strong notion of *budget awareness*: its query answering capability grows proportionally with the budget used to answer queries. As the refinement progresses, small partitions are merged into larger ones and it is getting increasingly difficult to load a partition into memory. Allowing a big partition to span between memory and disk is a natural choice (which is similar to GraphChi's PSW algorithm). However, continuing the search after the physical memory is exhausted will involve frequent disk I/O and significantly slow down query processing, rendering GraphQ's benefit less obvious compared to whole-graph computation. Hence, we treat the capacity of the main memory as a *budget* and terminate GraphQ with an out-of-budget failure when a merged partition is too big to fit into memory. There are various trade-offs that can be explored by the user to tune GraphQ; a more detailed discussion can be found at the end of §2.

It is important to note that GraphQ is fundamentally different from approximate computing [3, 30, 6], which terminates the computation early to produce *approximate* answers that may contain errors. GraphQ always produces correct answers for the user-specified query goals,



	Partitions	Cliques	Abs edges refined
Ite #0	{1, 2, 3} {4, 5, 6} {7, 8, 9}	{1,2,3} {4,6} {5} {7,8,9}	a, b, c
Ite #1	{1, 2, 3, 4, 5, 6} {7, 8, 9}	**{1,2,3,4,6}** {5} {7,8,9}	N/A

Figure 1: An example graph, its abstraction graph, and the computation steps for finding a clique whose size is ≥5. The answer of the query is highlighted.

but improves the computation scalability and efficiency by finding a scope on the input graph that is sufficient to answer a query.

Summary of Experimental Results Our experimental results show that GraphQ can answer queries in the magnitude of seconds to hundreds of seconds for graphs with several billions of edges. For example, over the twitter-2010 graph, GraphQ quickly identified 64 large communities in *49 seconds* while it took a state-of-the-art system, GraphChi [16], 6.4 hours to compute a whole-graph solution over the same graph. We have also compared GraphQ with a modified version of GraphChi that attempts to answer the same queries by terminating immediately when a satisfiable solution is found. The results demonstrate that GraphQ is, on average, 1.6–13.4× faster than this modified version due to the reduced I/O and computation from the processing of partial graphs. GraphQ is publicly available at https://bitbucket.org/wangk7/graphq.

2 Overview and Programming Model

Background Common to graph processing systems, the graph operated by GraphQ can be mathematically viewed as a directed (sparse) graph, $G = (V, E)$. A value is associated with each vertex $v \in V$, indicating an application-specific property of the vertex. For simplicity, we assume vertex values are labeled from 1 to $|V|$. Given an edge e of the form $u \rightarrow v$ in the graph, e is referred to as vertex v's *in-edge* and as vertex u's *out-edge*. The developer specifies an `update(v)` function, which can access the values of a vertex and its neighboring vertices. These values are fed into a function f that computes a new value for the vertex. The goal of the computation is to "iterate around" vertices to update their values until a global "fixed-point" is reached. This vertex-centric model is widely used in graph processing systems, such as Pregel [20], Pregelix [5], and GraphLab [18].

Figure 1 shows a simple directed graph that we will use as a running example throughout the paper. For each GraphQ query, the user first needs to find a related *base application* that performs whole-graph vertex-centric computation. This is not difficult, since many of these algorithms are readily available. In our example, the base application is Maximal Clique, and the query aims to find a clique whose size is no less than 5 (*i.e.* goal) over the input graph.

GraphQ first divides the concrete graph in Figure 1 (a) into three *partitions* — {1, 2, 3}, {4, 5, 6}, and {7, 8, 9} — a "pre-processing" step that only needs to be performed once for each graph. When the query is submitted, the goal of GraphQ is to use an *abstraction graph* to guide the selection of partitions to be merged, hoping that the query can be answered by merging only a very small number of partitions. Initially, inter-partition edges (shown as arrows with dashed lines) are disabled; they will be gradually recovered.

Programming Model A sample program for answering the clique query can be found in Figure 2. Overall, GraphQ is endowed with an expressive 2-tier programming model to balance simplicity and programmability:

- First, GraphQ *end users* only need to write 2-3 lines of code to submit a query. For example, the end user writes lines 2-5, submitting a `CliqueQuery` to look for `Clique` instances whose size is no fewer than 5 over the `ExampleGraph`.

- Second, GraphQ *library programmers* define how a query can be answered through a flexible programming model that fully supports in-graph computation. In the example, the clique query is defined between lines 7-40, by extending the `Query` and `QueryResult` classes in our library.

We expect regular GraphQ users — those who only care about *what* to query but not *how* to query it — to program only the first tier (between lines 2-5). The appeal of the GraphQ programming model lies in its flexibility. On one hand, the simplicity of the GraphQ first-tier interface is on par with query languages for similar purposes (such as SQL). On the other hand, for programmers concerned with graph processing efficiency, GraphQ provides opportunities for full-fledged programming "under the hood" at the second tier.

```
1
2  // end-user
3  Graph g = new ExampleGraph(); // A partitioned graph
4  CliqueQuery cq = new CliqueQuery(g, 5);
5  List<Clique> qr = cq.submit();
6
7  // library programmer
8  class CliqueQuery extends Query {
9    final Graph G; // graph
10   final int N; // goal
11   final int M; // max # of results to refine with
12   final int K; // max # of partitions to merge
13   final int delta; // the inc over K at each refinement
14
15   List<Partition> initPartitions()
16     { return g.partitions; }
17
18   boolean check(Clique c) {
19     if (c.size()>=N) { report(c); return true; }
20   }
21
22   List<AbstractEdge> refine(Clique c1, Clique c2) {
23       List<AbstractEdge> list;
24       foreach(Vertex v in c1.vertices())
25         foreach(Vertex u in c2.vertices())
26           AbstractEdge ae = g.abstractEdge(u, v);
27           if (ae != null) { list.add(ae); }
28       return list;
29   }
30
31   int resultThreshold() { return M; }
32   int partitionThreshold() { return K; }
33
34   CliqueQuery(Graph g, int n) {
35      this.G = g; this.N = n;
36   }
37 }
38 class Clique extends QueryResult {
39   int refinePriority()  { return size(); }
40   int size() {...}
41 }
```

Figure 2: Programming for answering clique Queries.

Partitions Given a very large graph, one can specify how it is partitioned using GraphQ parameters. A partition is both a logical and a physical concept. Logically, a partition is a subgraph (connected component) of the concrete graph. Physically, it is often aligned with the physical storage unit of data, such as a disk file. In our formulation where the graph vertices are labeled with numbers from 1 to $|V|$, we select partitions as containing vertices with continuous label numbers, and edges connecting those vertices in the concrete graph. Beyond this mathematical formulation is an intuitive goal: if we use labels 1 to $|V|$ to mimic the physical sequence of vertex storage, the partitions should be created to be as aligned with physical storage as possible. Thanks to this design, loading a partition is very efficient due to sequential disk accesses with strong locality.

When a query is defined — such as `CliqueQuery` — the programmer first decides what partitions should be *initially* considered to compute local solutions (*e.g.* cliques). This is supported by overriding the `initPartitions` method of the `Query` class, as in line 16. In our example, this method selects all partitions because we have no knowledge of whether and what cliques exist in each partition initially. GraphQ loads one partition into memory at a time and performs vertex-centric computation on the partition to compute local cliques independently of other partitions.

Observe that this does not contradict with our early discussion of incremental graph data processing: at the local computation phase, all partition-based computations are independent of each other. Therefore, when the data for one partition is loaded, the data for previously loaded partitions can be written back to disk, and at this phase GraphQ does not need to hold data in memory for more than one partition. Overall, this phase is very efficient because all inter-partition edges are ignored and there are only a very small number of random disk accesses.

Abstraction Graph The abstraction graph (AG) summarizes the concrete graph. Each *abstract vertex* in the AG abstracts a set of concrete vertices and each *abstract edge* connects two abstract vertices. An abstract edge can have an *abstract weight* that abstracts the weights of the actual edges it represents, which we have omitted in this short example.

To see the motivation behind the design of AG, observe that inter-partition edges can scatter across the partitions (*i.e.*, disk files) they connect, and knowing whether a concrete edge exists between two partitions requires loading both partitions into memory and a linear scan of them, a potentially costly step with a large number of disk accesses. As a "summarization" of the concrete graph, the AG is much smaller in size and can be *always* held in memory.

GraphQ first checks the existence of an abstract edge on the AG: the non-existence of an abstract edge between two abstract vertices $\bar{u}$ and $\bar{v}$ guarantees the non-existence of a concrete edge between any pair of concrete vertices (u, v) abstracted by $\bar{u}$ and $\bar{v}$; hence, we can safely skip the check of concrete edges. On the other hand, the existence of an abstract edge does not necessarily imply the existence of a concrete edge, and hence, the abstract edge needs to be *refined* to recover the concrete edges it represents.

The granularity of the AG is a design issue to be determined by the user. At one extreme, each partition can be an abstract vertex in the AG. This very coarse-grained abstraction may not be precise enough for GraphQ to quickly eliminate infeasible concrete edges. At the other extreme, a very fine-grained AG may take much space and the computation over the AG (such as a lookup) may take time. Since the AG is always in memory to provide quick guidance, a rule of thumb is to allow the abstraction granularity (*i.e.*, the number of concrete vertices represented by one abstract vertex) to be proportional to the memory capacity.

Using parameters, the user can specify the ratio between the size of the AG and the main memory — the

more memory a system has, the larger AG will be constructed by GraphQ to provide more precise guidance. Figure 1 (b) shows the AG for the concrete graph in Figure 1 (a). The GraphQ runtime uses the simple *interval domain* [9] to abstract concrete vertices — each abstract vertex represents two concrete vertices that have consecutive labels. This simple design turns out to be friendly for performance as well: each abstract edge represents a set of concrete edges stored together in the partition file; since refining an abstract edge needs to load its corresponding concrete edges, storing/loading these edges together maximizes sequential disk accesses and data locality. A detailed explanation of the storage structure can be found in §4.

An alternative route we decide not to pursue is to provide the user full programmability to construct their own AGs. The issue at concern is *correctness*. Our design of the abstraction graph is built upon the principled idea of abstraction refinement, with correctness guarantees (§3). The correctness is hinged upon that the AG is indeed a "sound" abstraction of the concrete graph. We rely on the GraphQ runtime to maintain this notion of sound abstraction.

Abstraction Refinement At the end of each local computation (*i.e.*, over a partition), GraphQ invokes the `check` method of the `Query` object. The method returns **true** if the query can be answered, and the result is reported through the `report` method (see line 19). Query processing terminates. If all local computations are complete and all `check` invocations return **false**, GraphQ tries to merge partitions to provide a larger scope for query answering. Recall that in our initial partition definition, all inter-partition edges have been ignored. The crucial challenge of partition merging thus becomes recovering the inter-partition edges, a process we call *abstraction refinement*.

In GraphQ, the refinement process is guided by the `QueryResult` — `Clique` in our example — from local computations. The key insight is that the results so far should offer clues on which partitions should be merged at a higher priority. The "priority" here can be customized by programmers through overriding the `refinePriority` method of class `QueryResult`. In the clique query example here, the programmer uses the size of the clique as the metric for priority (see line 39). Intuitively, merging partitions where larger cliques have been discovered is more likely to reach the goal of finding a clique of a certain size.

GraphQ next selects M (returned by `resultThreshold` in line 31) results with the highest priorities (*i.e.* largest cliques) for pairwise inspection. For each pair of cliques resulting from different partitions, the `refine` method (line 22) is invoked to verify if there is any potential for the two input cliques to combine into a larger clique. `refine` returns a list of abstract edges that should be refined. The implementation of `refine` is provided by programmers, typically involving the consultation of the AG. In our example, the method returns a list of candidate abstract edges whose corresponding concrete edges may potentially connect vertices from the two existing cliques (in two partitions) in order to form a larger clique.

Based on the returned abstract edges, GraphQ consults the AG to find the concrete edges these abstract edges represent. GraphQ then merges the partitions in which these concrete edges are located. To avoid a large number of partitions to be merged at a time — that would require the data associated all partitions to be loaded into memory at the same time — programmers can set a threshold specified by `partitionThresold`, in line 32. GraphQ adopts an iterative merging process: in each pass, merging only happens when the refinement leads to the merging of no more than K (returned by `partitionThresold`) partitions. If the merged partitions cannot answer the queries, GraphQ increases K by δ (line 13) at each subsequent pass to explore more partitions. This design enables GraphQ to gradually use more memory as the query processing progresses.

GraphQ terminates query processing in one of the 3 scenarios: (1) the `check` method returns **true**, in which case the query is answered; (2) all partitions are merged in one, and the `check` method still returns **false** — a situation in which this query is impossible to answer; and (3) a (memory) budget runs out, in which case GraphQ returns the best `QueryResults` that have been found *so far*. We will rigorously define this notion in §3.

Example Figure 1 (c) shows the GraphQ computational steps for answering the clique query. The three columns in the table show the partitions considered in the beginning of each iteration, the local maximal cliques identified, and the abstract edges selected by GraphQ to refine at the end of the iteration, respectively. Before iteration #0, the user selects all the three partitions via `initPartitions`. The vertex-centric computation of these partitions identifies four local cliques {1, 2, 3}, {4, 6}, {5}, and {7, 8, 9}.

Since the `check` function cannot find a clique whose size is ≥ 5, GraphQ ranks the four local cliques based on their sizes (by calling `refinePriority`) and invokes `refine` five times with the following clique pairs: ({1, 2, 3}, {7, 8, 9}), ({1, 2, 3}, {4, 6}), ({4, 6}, and {7, 8, 9}), ({5},{1, 2, 3}), ({5},{7, 8, 9}). For instance, for input ({1, 2, 3}, {7, 8, 9}), no abstract edge exists on the AG that connects any vertex in the first clique with any vertex in the second. Hence, `refine` returns an empty list.

For input ({1, 2, 3}, {4, 6}), however, GraphQ detects that there is an abstract edge between every abstract

vertex that represents $\{1, 2, 3\}$ and every abstract vertex that represents $\{4, 6\}$. The abstract edges connecting these two cliques (*i.e.*, *a*, *b*, and *c*) are then added into list `list` and returned.

After checking all pairs of cliques, GraphQ obtains 6 lists of abstract edges, among which five span two partitions and one spans three. Suppose K is 2 at this moment. The one spanning three partitions is discarded. For the remaining five lists, (a, b, c) is the first list returned by `refine` (on input ($\{1, 2, 3\}$, $\{4, 6\}$)). These three abstract edges are selected and their refinement adds the following four concrete (inter-partition) edges back to the graph: 4→2, 3→4, 1→5, and 2→6. The second iteration repeats vertex-centric computation by considering a merged partition $\{1, 2, 4, 5, 6\}$. When the partition is processed, a new clique $\{1, 2, 3, 4, 6\}$ is found. Function `check` finds that the clique answers the query; so it reports the clique and terminates the process.

Programmability Discussions In addition to answering queries with user-specified goals, our programming model can also support aggregation queries (min, max, average, *etc*.). For example, to find the largest clique under a memory budget, only minor changes are needed to the `CliqueQuery` example. First, we can define a private field called `max` to the class. Second, we need to update the `check` method as follows:

```
if(c.size()>max)
  {max=c.size(); return false;}
```

The observation here is that `check` should always return **false**. GraphQ will continue the refinement until the (memory) budget runs out, and the result `c` aligns with our intuition of being "the largest `Clique` under the budget based on the user-specified refinement heuristics", a flavor of the budget-aware query processing.

GraphQ can also support *multiplicity* of results, such as the top 30 largest cliques. This is just a variation of the example above. Instead of reporting a clique `c`, the `CliqueQuery` should maintain a "top 30" list, and use it as the argument for `report`.

Trade-off Discussions It is clear that GraphQ provides several trade-offs that the user can explore to tune its performance. First, the memory size determines GraphQ's answerability. A higher budget (*i.e.* more memory) will lead to (1) finding more entities with higher goals, or (2) finding the same number of entities with the same goals more quickly. Since GraphQ can be embedded in a data analytical application running on a PC, imposing a memory budget allows the application to perform intelligent resource management between GraphQ an other parts of the system, obtaining satisfiable query answers while preventing GraphQ from draining the memory.

Another tradeoff is defined by abstraction granularity, that is, the ratio between the size of the AG and the memory size. The larger this ratio is, the more precise guidance the AG provides. On the other hand, holding a very large AG in memory could hurt performance by eclipsing the memory that could have been allocated for data loading and processing. Hence, achieving good performance dictates finding the sweetspot.

3 Abstraction-Guided Query Answering

This section formally presents our core idea of applying abstracting refinement to graph processing. In particular, we rigorously define GraphQ's answerability.

Definition 3.1 (Graph Query). *A user query is a 5-tuple (Δ, ϕ, π, $\diamond$, g) that requests to find, in a directed graph $G = (V_G, E_G)$, Δ entities satisfying a pair of predicates $\langle \phi, \pi \diamond g \rangle$. Definition predicate $\phi \in \Phi$ is a logical formula ($\mathbb{P}(G) \rightarrow \mathbb{B}$) over the set of all G's subgraphs that defines an entity, $\pi \in \Pi$ is a quantitative function ($\mathbb{P}(G) \rightarrow \mathbb{R}$) over the set of subgraphs satisfying ϕ, measuring the entity's* size, *and $\diamond$ is a numerical comparison operator* (e.g.*, $\geq$ or $=$) that compares the output of π with a user-specified* goal of the query $g \in \mathbb{R}$.

This definition is applicable to a wide variety of user queries. For example, for the clique query discussed in §2, ϕ is the following predicate on the vertices and edges of a subgraph $S \subseteq G$, defining a clique:
$\forall\, v_1, v_2 \in V_S$: $\exists e \in E_S$: $e = (v_1, v_2) \vee e = (v_2, v_1)$,
while π is a simple function that returns the number of vertices $|V_S|$ in the subgraph. $\diamond$ and g are $\geq$ and 5, respectively. From this point on, we will refer to ϕ and π as the *definition predicate* and the *size function*, respectively.

Definition 3.2 (Monotonicity of the Size Function). *A query (Δ, ϕ, π, $\diamond$, g) is GraphQ-answerable if $\pi \in \Pi$ is a* monotone *function with respect to operator $\diamond$: $\forall S_1 \in \mathbb{P}(G), S_2 \in \mathbb{P}(G) : S_2 \subseteq S_1 \wedge \phi(S_1) \wedge \phi(S_2) \implies \pi(S_1) \diamond \pi(S_2)$.*

While the user can specify an arbitrary size function π or goal g, π has to be *monotone* in order for GraphQ to answer the query. More precisely, for any subgraphs S_1 and S_2 of the input graph G, if S_2 is a subgraph of S_1 and they both satisfy the definition predicate ϕ, the relationship between their sizes $\pi(S_1)$ and $\pi(S_2)$ is $\pi(S_1) \diamond \pi(S_2)$. For example, if S_2 is a clique with N vertices, and S_1 is a supergraph of S_2 and also a clique, S_1's size must be $\geq N$. Monotonicity of the size function implies that once GraphQ finds a solution that satisfies a query at a certain point, the solution will *always* satisfy the query because GraphQ will only find better solutions in the forward execution. It also matches well with the underlying vertex-centric computation model that gradually propagates the information of a vertex to distant vertices (*i.e.*, which has the same effect as considering increasingly large subgraphs).

Definition 3.3 (Partition). *A partition P of graph G is a subgraph (V_P, E_P) of G such that vertices in V_P have contiguous labels [i, i + $|V_P|$], where $i \in I$ is the minimum integer label a vertex in V_P has and $|V_P|$ is the number of vertices of P. A* partitioning *of G produces a set of partitions $P_1, P_2, \ldots P_k$ such that $\forall j \in [1, k-1]$: $max_{v \in V_{P_j}} label(v) + 1 = min_{v \in V_{P_{j+1}}} label(v)$. An edge $e = (v_1, v_2)$ is an* intra-partition *edge if v_1 and v_2 are in the same partition; otherwise, e is an* inter-partition *edge.*

Logically, each partition is defined by a label range, and physically, it is a disk file containing the edges whose targets fall into the range. The physical structure of a partition will be discussed in §4.

Definition 3.4 (Abstraction Graph). *An abstraction graph ($\bar{V}, \bar{E}, \alpha, \gamma$) summarizes a concrete graph (V, E) using abstraction relation α: $V \to \bar{V}$. The AG is a* sound *abstraction of the concrete graph if $\forall e = (v_1, v_2) \in E$: $\exists \bar{e} = (\bar{v_1}, \bar{v_2}) \in \bar{E} : \bar{v_1}, \bar{v_2} \in \bar{V} \wedge (v_1, \bar{v_1}) \in \alpha \wedge (v_2, \bar{v_2}) \in \alpha$. γ: $\bar{V} \to V$ is a* concretization relation *such that $(\bar{v}, v) \in \gamma$ iff $(v, \bar{v}) \in \alpha$.*

α and γ form a *monotone Galois connection* [9] between G and AG (which are both posets). There are multiple ways to define the abstraction function α. In GraphQ, α is defined based on an interval domain [9]. Specifically, each abstract vertex $\bar{v}$ has an associated interval [*i*, *j*]; $(v, \bar{v}) \in \alpha$ iff *label*(*v*) $\in$ [*i*, *j*]. The primary goal is to make concrete edges whose target vertices have contiguous labels stay together in a partition file. To concretize an abstract edge, GraphQ will only need *sequential accesses* to a partition file, thereby maximizing locality and refinement performance. Different abstract vertices have disjoint intervals. The length of the interval is determined by a user-specified percentage *r* and the maximum heap size *M*—the size of the AG cannot be greater than $r \times M$. The implementation details of the partitioning and the AG construction can be found in §4. Clearly, the AG constructed by the interval domain is a sound abstraction of the input graph.

Lemma 3.5 (Edge Feasibility). *If no abstract edge exists from $\bar{v_1}$ to $\bar{v_2}$ on the AG, there must not exist a concrete edge from v_1 to v_2 on the concrete graph such that $(v_1, \bar{v_1}) \in \alpha$ and $(v_2, \bar{v_2}) \in \alpha$.*

The lemma can be easily proved by contradiction. It enables GraphQ to inspect the AG first to quickly skip over infeasible solutions.

Definition 3.6 (Abstraction Refinement). *Given a subgraph S = (V_s, E_s) of a concrete graph G = (V, E) and its AG = ($\bar{V}$, $\bar{E}$) of G, an abstraction refinement $\sqsubseteq$ on S selects a set of abstract edges $\bar{e} \in \bar{E}$ and adds into E_s all such concrete edges e that $e \in E \setminus E_s : (\bar{e}, e) \in \alpha$. An abstraction refinement of the form $S \sqsubseteq S'$ produces a new subgraph S' = (V_s', E_s'), such that $V_s = V_s'$ and $E_s \subseteq E_s'$. A refinement is an* effective *refinement if $E_s \subset E_s'$.*

The concretization function is used to obtain concrete edges for a selected abstract edge. After an effective refinement, the resulting graph S' becomes a (strict) supergraph of *S*, providing a larger scope for query answering.

Lemma 3.7 (Refinement Soundness). *An entity satisfying the predicates (ϕ, $\pi \diamond g$) found in a subgraph S is preserved by an abstraction refinement on S.*

The lemma shows an important property of our analysis. Since our goal is to find Δ entities, this property guarantees that the entities we find in previous iterations will stay as we enlarge the scope. The lemma can be easily proved by considering Definition 3.2: since the size function π is monotone, if the predicate $\pi(S) \diamond g$ holds in subgraph *S*, the predicate $\pi(S') \diamond g$ must also hold in subgraph S' that is a strict supergraph of *S*. Because S' contains all vertices and edges of *S*, the fact the definition predicate ϕ holds on *S* implies that ϕ also holds on S' (*i.e.*, $\phi(S) \implies \phi(S')$).

Definition 3.8 (Essence of Query Answering). *Given an initial subgraph S = (V, E_s) composed of a set $\mathbb{P}$ of disjoint partitions ((V_1, E_1), ..., (V_j, E_j)) such that $V = V_1 \cup \ldots \cup V_j$ and $E_s = E_1 \cup \ldots \cup E_j$, as well as an AG = ($\bar{V}$, $\bar{E}$), answering a query (Δ, ϕ, π, $\diamond$, g) aims to find a refinement chain $S \sqsubseteq^* S''$ such that there exist at least Δ distinct entities in S'', each of which satisfies both ϕ and $\pi \diamond g$.*

In the worst case, S'' becomes *G* and graph answering has (at least) the same cost as computing a whole-graph solution. Each refinement step bridges multiple partitions. Suppose we have a partition graph (PG) for *G* where each partition is a vertex. The refinement chain starts with a *PG* without edges (*i.e.*, each partition is a connected component), and gradually adds edges and reduces the number of connected components. Suppose PG_S is the PG for a subgraph *S*, ρ is a function that takes a *PG* as input and returns the maximum number of partitions in a connected component of the *PG*, and each initial partition has the (same) size η. We have the following definition:

Definition 3.9 (Budget-Aware Query Answering). *Answering a query under a memory budget M aims to find a refinement chain $S \sqsubseteq^* S''$ such that $\forall$ ($S_1 \sqsubseteq S_2$) $\in \sqsubseteq^*$: $\eta \times \rho(PG_{S_2}) \leq M$.*

In other words, the number of (initial) partitions connected by each refinement step must not exceed a threshold *t* such that $t \times \eta \geq M$. Otherwise, the next iteration

would not have enough memory to load and process these t partitions.

Theorem 3.10 (Soundness of Query Answering). *GraphQ either returns correct solutions or does not return any solution if the vertex-centric computation is correctly implemented.*

Limitations Despite its practical usefulness, GraphQ can only answer queries whose vertex update functions are monotonic, while many real-world problems may not conform to this property. For example, for machine learning algorithms that perform probability propagation on edges (*e.g.*, belief propagation and the coupled EM (CoEM)), the probability in a vertex may fluctuate during the computation, preventing the user from formulating a probability problem as GraphQ queries.

4 Design and Implementation

We have implemented GraphQ based on GraphChi [16], a high-performance single-machine graph processing system. GraphChi has both C++ and Java versions; GraphQ is implemented on top of its Java version to provide an easy way for the user to write UDFs. Our implementation has an approximate of 5K lines of code and is available for download on BitBucket. The pre-processing step splits the graph file into a set of small files with the same format, each representing a partition (*i.e.*, a vertex interval). We modify the *shard* construction algorithm in GraphChi to partition the graph. Similarly to a shard in [16], each partition file contains all in-edges of the vertices that logically belong to the partition; hence, edges stored in a partition file whose sources do not belong to the partition are inter-partition edges.

The AG is constructed when the graph is partitioned. To allow concrete edges (*i.e.*, lines in each text file) represented by the same abstract edge to be physically located together, we first sort all edges in a partition based on the labels of their source vertices — it moves together edges from contiguous vertices. Next, for each abstract vertex (*i.e.*, an interval), we sort edges that come from this interval based on the labels of their target vertices — now the concrete edges represented by the same abstract edge are restructured to stay in a contiguous block of the file. This is a very important handling and will allow efficient disk accesses, provided that large graph processing is often I/O dominated.

For example, for an abstract edge [40, 80] → [1024, 1268], its concrete edges are located physically in the partition file containing the vertex range [1024, 1268]. The first sort moves all edges coming from [40, 80] together. However, among these edges, those going to [1024, 1268] and those not are mixed. The second sort moves them around based on their target vertices, and thus, edges going to contiguous vertices are stored contiguously. Although the interval length used in the abstraction is statically fixed (*i.e.*, defined as a user parameter), we do not allow an abstract vertex to represent concrete vertices spanning two partitions — we adjust the abstraction interval if the number of the last set of vertices in a partition is smaller than the fixed interval size.

Each abstract edge consists of the starting and ending positions of the concrete edges it represents (including the partition ID and the line offsets), as well as various summaries of these edges, such as the number of edges, and the minimum and maximum of their weights. The AG is saved as a disk file after the construction. It will be loaded into memory upon query answering. When an (initial or merged) partition is processed, we modify the parallel sliding window algorithm in GraphChi to load the entire partition into memory. In GraphChi, a *memory shard* is a partition being processed while *sliding shards* are partitions containing out-edges for the vertices in the memory shard. Since inter-partition edges are ignored, GraphQ eliminates sliding shards and treats each partition p as a memory shard. The number of random disk accesses at each step thus equals the number of initial partitions contained in p.

The loaded data may include both enabled and disabled edges; the disabled edges are ignored during processing. Initially, all inter-partition edges are disabled. Refining an abstract edge loads the partitions to be merged and enables the inter-partition edges it represents before starting the computation. We treat the refinement process as an *evolving graph*, and modify the incremental algorithm in GraphChi to only compute and propagate values from the newly added edges.

A user-specified ratio r is used to control the size of the AG. Ideally, we do not want the size of the AG to exceed $r\times$ the memory size. However, this makes it very difficult to select the interval size (*i.e.* abstraction granularity) before doing partitioning, because the size of the AG is related to its number of edges and it is unclear how this number is related to the interval size before scanning the whole graph. To solve the problem, we use the following formula to calculate the interval size i: $i = \frac{size(G)}{M \times r}$, under a rough estimation that if the number of vertices is reduced by i times, the number of edges (and thus the size of the graph) is also reduced by i times. In practice, the size of the AG built using i is always close to $M \times r$, although it often exceeds the threshold.

5 Queries and Methodology

We have implemented UDFs for five common graph algorithms shown in Table 1. The pre-processing time is a one-time cost, which does not contribute to the actual query answering time. For PageRank and Path, GraphQ does not need to compute local results; what par-

Name	*Query GraphQ to Find*	*Init*	*RefinePriority*	*GraphChi Time*	*GraphQ Pre-proc. Time*
PageRank	Δ vertices whose pageranks are $\geq N$	none	X-percentages ($\uparrow$)	1754, 2880 secs.	120+0, 200+0 secs.
Clique	Δ cliques whose sizes are $\geq N$	all	clique sizes ($\uparrow$)	5.5, 50.2 hrs.	400+500, 800+1060 secs.
Community	Δ communities whose sizes are $\geq N$	all	community sizes ($\uparrow$)	3.4, 6.4 hrs.	150+200, 300+400 sec.
Path	Δ paths whose lengths are $\leq N$	none	path lengths ($\downarrow$)	?, ?	200+0, 400+0 secs.
Triangle	Δ vertices whose edge triangles are $\geq N$	all	triangle counts ($\uparrow$)	1990, 3194 secs.	200+300, 400+600 secs.

Table 1: A summary of queries performed in the evaluation: reported are the names and forms of the queries, initial partition selection, priority of partition merging, whole-graph computation times in GraphChi for the uk-2005 [4] and the twitter-2010 [15] graphs, and the time for pre-processing them; $\uparrow$ ($\downarrow$) means the higher (lower) the better; each pre-processing time has two components $a+b$, where a represents the time for partitioning and AG construction, and b represents the time for initial (local) computation; "?" means the whole-graph computation cannot finish in 48 hours.

Name	*Type*	$\lvert V\rvert$	$\lvert E\rvert$	*#IP*	*#MP*	δ
uk-2005 [4]	webgraph	39M	0.75B	50	30	10
twitter-2010 [15]	social network	42M	1.5B	100	50	10

Table 2: Our graph inputs: reported in each section are their names, types, numbers of vertices, numbers of edges, numbers of initial partitions (*IP*), numbers of maximum partitions allowed to be merged before out of budget (*MP*), and numbers of partitions increased at each step (δ, *cf.* line 13 in Figure 2).

titions to be merged can be determined simply based on the structure of each partition. We experimented GraphQ with a variety of graphs. Due to space limitations, this section reports our results with the two largest graphs, shown in Table 2. Since the focus of this work is *not* to improve the whole-graph computation, we have not run other distributed platforms.

PageRank Answering PageRank queries is based on the whole-graph PageRank algorithm used widely to rank pages in a webgraph. The algorithm is not strictly monotone, because vertices with few incoming edges would give out more than they gain in the beginning and thus their pageranks values would drop in the first few iterations. However, after a short "warm-up" phase, popular pages would soon get their values back and their pageranks would continue to grow until the convergence is reached. To get meaningful pagerank values to query upon, we focus on the top 100 vertices reported by GraphChi (among many million vertices in a graph). Their pageranks are very high and these vertices represent the pages that a user is interested in and wants to find from the graph.

Focusing on the most popular vertices also allows us to bypass the non-monotonic computation problem—since the goals are very high, it is only possible to answer a query during monotonic phase (after the non-monotonic warm-up finishes). The refinement logic we implemented favors the merging of partitions that can lead to a larger *X-percentage*. The X-percentage of a partition is defined as the percentage of the outgoing edges of the vertex with the largest degree that stay in the partition. It is a metric that measures the completeness of the edges for the most popular vertex in the partition. The higher the X-percentage is, the quicker it is for the pagerank computation to reach a high value and thus the easier for GraphQ to find popular vertices. PageRank does not need a local phase—from the AG, we directly identify a list of partitions whose merging may yield a large X-percentage.

Clique is based on the Maximal Clique algorithm that computes a maximal clique for each vertex in the graph. Since the input is a directed graph, a set of vertices forms a clique if for each pair of vertices u and v, there are two edges between them going both directions. GraphChi does not support variable-size edge and vertex data, and hence, we used 10 as the upper-bound for the size of a clique we can find. In other words, we associated with each edge and vertex a 44-byte buffer (*i.e.*, 10 vertices take 40 bytes and used an additional 4-byte space in the beginning to save the actual length). Due to the large amount of data swapped between memory and disk, the whole-graph computation over twitter-2010 took more than 2 days.

Path is based on the SSSP algorithm and aims to find paths with acceptable length between a given source and destination. Similarly to Clique, we associated a (fixed-length) buffer with each edge/vertex to store the shortest path for the edge/vertex. Since none of our input graphs have edge weights, we assigned each edge a random weight between 1 and 5. However, the whole-graph computation could not finish processing these graphs in 2 days. To generate reasonable queries for GraphQ, we sampled each graph to get a smaller graph (that is 1/5 of the original graph) and ran the whole-graph SSSP algorithm to obtain the shortest paths between a specified vertex S (randomly chosen) and all other vertices in the sample graph. If there exists a path between S and another vertex v in the small graph, a path must also exist in the original graph. The SSSP computation over even the small graphs took a few hours.

Community is based on a community detection algorithm in which a vertex chooses the most frequent label

of its neighbors as its own label. Triangle uses a triangle counting algorithm that counts the number of edge triangles incident to each vertex. This problem is used in social network analysis for analyzing the graph connectivity properties [27]. For both applications, we obtained their whole-graph solutions and focus on the 100 largest entities (*i.e.*, communities and vertices with most triangles). Community and Triangle favor the merging of partitions that can yield large communities and triangle counts, respectively.

6 Evaluation

Test Setup All experiments were performed on a normal PC with one Intel Core i5-3470 CPU (3.2GHz) and 10GB memory, running Ubuntu 12.04. The JVM used was the HotSpot Client VM (build 24.65-b04, mixed mode, sharing). Some of our results for GraphChi may look different from those reported in [16] due to different versions of GraphChi used as well as different hardware configurations. We have conducted three sets of experiments. First, we performed queries with various goals and Δ to understand the query processing capability of GraphQ. Second, we compared the query answering performance between GraphQ and *GraphChi-ET* (i.e., acronym for "GraphChi with early termination") — a modified version of GraphChi that terminates immediately when a query is answered. Third, we varied the abstraction granularity to understand the impact of abstraction refinement. The first and third sets of experiments ran GraphQ on the PC's embedded 500GB HDD to understand the query performance on a normal PC while a Samsung 850 250GB SSD was used for the second set to minimize the I/O costs, enabling a fair comparison with GraphChi-ET.

6.1 Query Efficiency

In this experiment, the numbers of initial partitions for the two graphs are shown in Table 2. The maximum heap size is 8GB, and the ratio between the AG size and the heap size is 25%. For the two graphs, the maximum number of partitions that can be merged before out of budget is 30 and 50. For each algorithm, GraphQ first performed local computation on initial partitions (as specified by the UDF `initPartitions`). Next, we generated queries whose goals were randomly chosen from different value intervals. Queries with easy goals/small Δ were asked earlier than those with more difficult goals/larger Δ, so that the computation results for earlier queries could serve a basis for later queries (*i.e.*, incremental computation). This explains why answering a difficult query is sometimes faster than answering an easy query (as shown later in this section).

PageRank To better understand the performance, we divided the top 100 vertices (with the highest pagerank values from the whole-graph computation) into several intervals based on their pagerank values. Each interval is thus defined by a pair of lower- and upper-bound pageranks. We generated 20 queries for each interval, each requesting to find Δ vertices with the goal being a randomly generated value that falls into the interval. For each interval reported in Table 3, all 20 queries were successfully answered. The average running time for answering these queries over uk-2005 is shown in the *Time* sections.

Δ	(a) Top20		(b) 20-40		(c) 40-60		(d) 60-100	
	Time	*Par*	*Time*	*Par*	*Time*	*Par*	*Time*	*Par*
1	56.1	20	5.6	10	3.0	10	4.3	10
2	32.2	20	5.0	10	5.1	10	6.6	10
4	120.0	20	27.0	10	19.2	10	21.6	10
8	350.1	30	182.9	30	54.3	20	41.9	20

Table 3: GraphQ performance for answering PageRank queries over uk-2005; each section shows the performance of answering queries on pagerank values that belong to an interval in the top 100 vertex list; reported in each section are the number of entities requested to find (Δ), the average query answering time in seconds (*Time*), and the number of partitions merged when a query is answered (*Par*).

The largest Δ we have tried is 8—GraphQ ran out of budget for most of the queries when a larger Δ was used. When $\Delta \leq 4$, GraphQ could successfully answer all queries even including those from the top 10 category. For twitter-2010, GraphQ always failed on queries whose goals were selected from the top 10 category. Otherwise, it successfully answered all queries. For example, the average time for answering 8 queries whose goals are from the top 10-20 category is 754.7 seconds.

Clique The biggest clique found in twitter-2010 (by the 52-hour whole-graph computation) has 6 vertices and there are totally 66 of them. The (relatively small) size of the maximum clique is expected because a clique in a directed graph has a stronger requirement: bi-directional edges must exist between each pair of vertices. The largest Δ we have tried is 64. Table 4 shows GraphQ's performance as we changed Δ; the running time reported is the average time across answering 20 queries in each interval. GraphQ could easily find 8 of the 66 6-clique (in 823 seconds), but the time increased significantly when we asked for 16 of them. GraphQ could find no more than 26 6-cliques before running out of budget. If a user is willing to sacrifice her goal and look for smaller cliques (say 5-cliques), GraphQ can find 64 of them in 460 seconds (by merging only 10 partitions).

Community The whole-graph computation of community detection took 1.5 hours on uk-2005 and 6.4 hours on twitter-2010. Similarly to PageRank, we

Δ	(a) *Size = 6*		(b) *Size = 5*		(c) *Size = 4*		(d) *Size = 3*	
	Time	*Par*	*Time*	*Par*	*Time*	Par	*Time*	*Par*
1	98.3	10	2.0	10	2.0	10	2.0	10
2	248.1	10	2.0	10	2.3	10	2.0	10
4	489.5	20	2.1	10	2.0	10	8.3	10
8	823.9	20	51	10	2.1	10	8.2	10
16	5960.3	30	49.1	10	2.1	10	9.6	10
32	-	50	144.1	10	2.8	10	16.4	10
64	-	50	460.0	10	128.3	10	20.0	10

Table 4: GraphQ's performance for answering Clique queries over twitter-2010; a "-" sign means some queries in the group could not be answered.

focused on the top 100 largest communities and asked GraphQ for communities of different sizes (that belong to different intervals on the top 100 list). For each interval, we picked 20 random sizes to run GraphQ and the average running time over uk-2005 is reported in Table 5. The whole-graph result shows that there are a few (less than 10) communities that are much larger than the other communities on the list. These communities have many millions of vertices and none of them could be found by GraphQ before the budget ran out. Hence, Table 5 does not include any measurement for queries with a size that belongs to the top 10 interval.

Δ	(a) Top10-20		(b) 20-40		(c) 40-60		(d) 60-100	
	Time	*Par*	*Time*	*Par*	*Time*	*Par*	*Time*	*Par*
1	8.2	10	4.9	10	4.3	10	4.5	10
2	51.8	10	34.5	10	20.1	10	14.2	10
4	142.1	20	63.3	10	27.1	10	25.4	10
8	292.3	20	160.6	20	56.9	10	35.5	10
16	563.4	30	236.7	30	196.7	20	97.7	20
32	-	30	-	30	-	30	332.8	30

Table 5: GraphQ's performance for answering Community queries over uk-2005; each section reports the average time for finding communities whose sizes belong to different intervals in the top 100 community list.

Interestingly, we found that GraphQ performed much better over twitter-2010 than uk-2005: for twitter-2010, GraphQ could easily find (in 162.1 seconds) 256 communities from the top 10-20 range by merging only 20 partitions as well as 1024 communities (in 188.2 seconds) from the top 20-40 range by merging 20 partitions. This is primarily because twitter-2010 is a social network graph in which communities are much "better defined" than a webgraph such as uk-2005.

Path We inspected the whole-graph solution for each sample graph (*cf.* §5) and found a set t of vertices v such that the shortest path on the small graph between S (the source) and each v is between 10 and 25 and contains at least 5 vertices. We randomly selected 20 vertices u from t and queried GraphQ for paths between S and u over the original graph. Based on the length of their shortest paths on the small graph, we used 10, 15, 20, and 25 as the goals to perform queries (recall that each edge has an artificial length between 1 and 5). The average time to answer these queries on twitter-2010 is reported in Figure 6.

Δ	(a) 10		(b) 15		(c) 20		(d) 25	
	Time	*Par*	*Time*	*Par*	*Time*	*Par*	*Time*	*Par*
1	59.5	10	57.6	10	58.1	10	45.2	10
2	55.5	20	53.2	20	49.1	20	65.0	10
4	230.1	50	111.8	20	110.7	20	115.6	20

Table 6: GraphQ's performance for answering Path queries over twitter-2010.

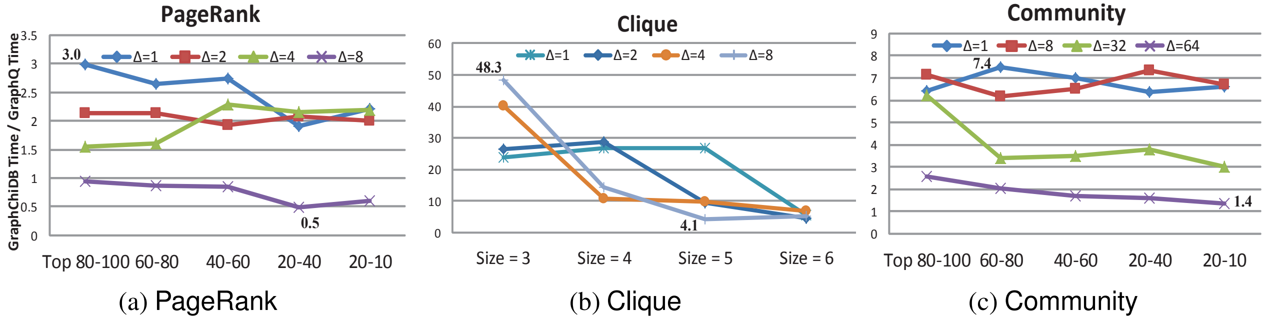
Our results for Path clearly demonstrate the benefit of GraphQ: it took the whole-graph computation 6.2 hours to process a graph only 1/5 as big as twitter-2010, while GraphQ can quickly find many paths of reasonable length in the original twitter graph.

Triangle A similar experiment was performed for Triangle (as shown in Figure 7): we focused on the top 100 vertices with the largest numbers of edge triangles. GraphQ could find only two vertices when a value from the top 10 triangle count list was used as a query goal. However, if the goal is chosen from the top 10-20 interval, GraphQ can easily find 16 vertices (which obviously include some top 10 vertices). It is worth noting that GraphQ found these vertices by combining only 10 partitions. This is easy to understand—edge triangles are local to vertices; computing them does not need to propagate any value on the graph. Hence, vertices with large triangle counts can be easily found as long as (most of) their own edges are recovered.

Δ	(a) Top10-20		(b) 20-40		(c) 40-60		(d) 60-100	
	Time	*Par*	*Time*	*Par*	*Time*	Par	*Time*	*Par*
1	3.3	10	3.0	10	2.9	10	4.5	10
2	3.2	10	3.6	10	3.9	10	7.6	10
4	3.4	10	3.2	10	3.1	10	8.7	10
8	2.8	10	3.3	10	3.0	10	19.6	10
16	2.9	10	2.9	10	3.2	10	313.3	10

Table 7: GraphQ's performance for answering Triangle queries over uk-2005.

The measurements in Table 3–7 also demonstrate the impact of the budget. For twitter-2010, merging 50, 30, 20, and 10 partitions requires, roughly, 6GB, 3.6GB, 2.4GB, and 1.2GB of memory, while, for uk-2005, the amounts of memory needed to merge 30, 20, and 10 partitions are 5.5GB, 4GB, and 2GB, respectively. From these measurements, it is easy to see what queries can and cannot be answered given a memory budget.



(a) PageRank (b) Clique (c) Community

Figure 3: Ratios between the running times of GraphChi-ET and GraphQ over twitter-2010: (a) PageRank: Max = 3.0, Min = 0.5, GeoMean = 1.6; (b) Clique: Max = 48.3, Min = 4.1, GeoMean = 13.4; and (c) Community: Max = 7.5, Min = 1.4, GeoMean = 4.2.

System	Time(s)	Comp.	Comp.Perc.	I/O	IO.Perc.
Q:PR	520.0	147.6	28.4%	372.4	71.6%
ET:PR	301.0	69.0	22.9%	232.0	77.1%
Q:Clique	637.0	548.5	86.1%	88.5	13.9%
ET:Clique	3208.0	2857.1	89.1%	351.0	10.9%
Q:Comm	81.5	25.6	31.4%	55.9	68.6%
ET:Comm	112.0	45.0	40.2%	68.0	60.7%

Table 8: A breakdown of time on computation and I/O for GraphQ and GraphChi-ET for PageRank, Clique, and Comm; measurements were obtained by running the most difficult queries from Figure 3.

6.2 Comparison to GraphChi-ET

GraphChi-ET is a modified version of GraphChi in which we developed a simple interface that allows the user to specify the Δ and goal for a query and then run GraphChi's whole-graph computation to answer the query – the computation is terminated immediately when the query is answered. Figures 3 shows performance comparisons between GraphQ and GraphChi-ET over twitter-2010 on three algorithms using SSD. A similar trend can also be observed on the other two algorithms; their results are omitted due to space limitations.

Note that for PageRank, GraphQ outperforms GraphChi-ET in all cases except when $\Delta = 8$. In this case, GraphQ is about $2\times$ slower than GraphChi-ET because GraphQ needs to merge 50 partitions and is always close to running out of budget. The memory pressure is constantly high, making in-memory computation less efficient than GraphChi-ET's PSW algorithm. For all the other benchmarks, GraphQ runs much faster than GraphChi-ET. An extreme case is when $\Delta = 1$ for Clique, as shown in Figure 3 (b), GraphChi-ET found a 3-clique in 159.5 seconds, while GraphQ successfully answered the query only in 3.3 seconds. This improvement stems primarily from GraphQ's ability of prioritizing partitions and intelligently enlarging the processing scope.

Table 8 shows a detailed breakdown of running time on I/O and computation for answering the most difficult queries from Figure 3 (i.e., those represented by points at the bottom right corner of each plot). These queries have the longest running time, which enables an easier comparison. Clearly, GraphQ reduces both computation and I/O because it loads and processes fewer partitions. However, the percentages of I/O and computation in the total time of each query are roughly the same for GraphQ and GraphChi-ET.

6.3 Impact of Abstraction Refinement

To understand the impact of abstraction refinement, we varied the abstraction granularity by using 0.5GB, 1GB, and 2GB of the heap to store the AG. The numbers of abstract vertices for each partition corresponding to these sizes are $a = 25$, 50, and 100, respectively, for twitter-2010. We fixed the budget at 50 partitions (which consume 6GB memory), so that we could focus on how performance changes with the abstraction granularity. We have also tested GraphQ without abstraction refinement: partitions are randomly selected to merge. These experiments were conducted for all the algorithms; due to space limitations, we show our results only for PageRank.

Figure 4 compares performance under different abstraction granularity for $\Delta = 1$, 4, and 8. While configuration $a = 100$ always yields the best performance, its running time is very close to that of $a = 50$. It is interesting to see that, in many cases (especially when $\Delta = 4$), $a = 25$ yields worse performance than random selection. We carefully inspected this AG and found that the abstraction level is so high that different abstract vertices have similar degrees. The X-percentages for different partitions computed based on the AG are also very similar, and hence, partitions are merged almost in a sequential manner (*e.g.*, partitions 1–10 are first merged, followed by 10–20, *etc.*). In this case, the random selection has a higher probability of finding the appropriate partitions to merge.

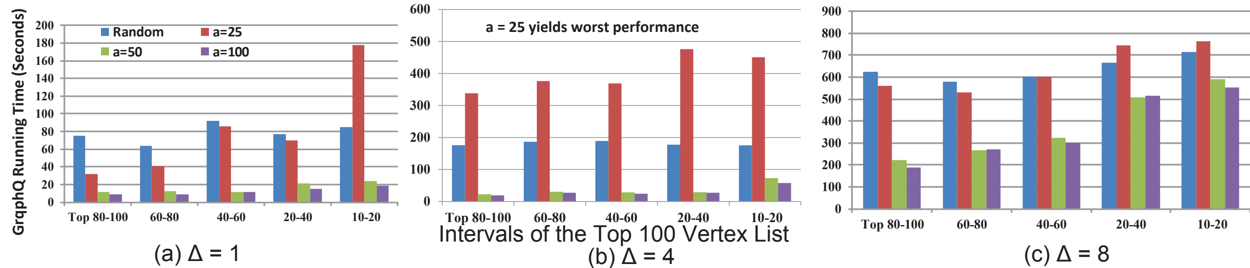


Figure 4: GraphQ's running time (in seconds) for answering PageRank queries over twitter-2010 using different abstraction graphs: *Random* means no refinement is used and partitions are merged randomly; $a = i$ means a partition is represented by i abstract vertices.

Despite its slow running time, random selection found all vertices requested by the queries. This is because, in the twitter graph, the edges of high-degree vertices are reasonably evenly distributed in different partitions of the graph. A similar observation was made for Triangle. But for the other three algorithms, their dependence on the AG is much stronger. For example, GraphQ could not answer any path query without the AG. As another example, no cliques larger than 3 could be found by using random selection.

7 Related Work

Graph Analytics A large body of work [16, 24, 19, 18, 5, 20, 26, 12] exists on efficient processing of large graphs. Existing work focuses on developing either new programming models [5, 18, 20] or algorithms that can reduce systems cost of graph processing such as communication overhead, random disk accesses, or GC effort. GraphChi [16], X-Stream [23], and GridGraph [29] are systems that perform out-of-core graph processing on a single machine. GraphQ differs from all these systems in its way to analyze partial graphs.

Work from [22] proposes *Galois*, a lightweight infrastructure that uses a rich programming model with coordinated and autonomous scheduling to support more efficient whole-graph computation. Unlike the existing systems that compute whole-graph solutions, GraphQ employs abstraction refinement to answer various kinds of analytical queries, facilitating applications that only concern small portions of the graph. There also exists work on graph databases such as Neo4j [1] and GraphChiDB [17]. They focus primiarily on enabling quick lookups on edge and vertex properties, while GraphQ focuses on quickly answering analytical queries.

Approximate Queries There is a vast body of work [13, 14, 7] on providing approximate answers to relational queries. These techniques use synopses like samples [13], histograms [14], and wavelets [7] to efficiently answer database queries. However, they have limited applicability to graph queries. Graph compression/-clustering/summarization [21, 28, 10, 25, 11] has been extensively studied in the database community. These techniques focus on (lossy and lossless) algorithms to summarize the input graph so that graph queries can be answered efficiently on the summary graph. Unlike the graph compression techniques that trade off graph accuracy for efficiency, GraphQ never answers queries over a summary graph, but instead, it only uses the summary graph to rule out infeasible solutions and always resorts to the concrete graph to find a solution. In addition, the graphs used to evaluate the aforementioned systems are relatively small—they only have a few hundreds of vertices and edges, which can be easily loaded into memory. In comparison, the graphs GraphQ analyzes are at the scale of several billions of edges and cannot be entirely loaded into memory.

8 Conclusion

This paper presents GraphQ, a graph query system based on abstraction refinement. GraphQ divides a graph into partitions and merges them with the guidance from a flexible programming model. An abstraction graph is used to quickly rule out infeasible solutions and identify mergeable partitions. GraphQ uses the memory capacity as a budget and tries its best to find solutions before exhausting the memory. GraphQ is the *first* graph processing system that can answer analytical queries over partial graphs, opening up new possibilities to scale up Big Graph processing with small amounts of resources.

Acknowledgements

We would like to thank our shepherd Emil Sit and the anonymous reviewers for their valuable and thorough comments. We also thank Rajiv Gupta, Xiaoning Ding, and Feng Qin for their helpful comments on an early draft of the paper. This material is based upon work supported by the National Science Foundation under the grants CCF-1054515, CCF-1117603, CNS-1321179, CNS-1319187, CCF-1349528, and CCF-1409829, and by the Office of Naval Research under grant N00014-14-1-0549.

References

[1] The neo4j graph database. http://neo4j.com/, 2014.

[2] The Titan Distributed Graph Database. http://thinkaurelius.github.io/titan/, 2014.

[3] W. Baek and T. M. Chilimbi. Green: A framework for supporting energy-conscious programming using controlled approximation. In *PLDI*, pages 198–209, 2010.

[4] P. Boldi, M. Santini, and S. Vigna. A large time-aware web graph. *SIGIR Forum*, 42(2):33–38, 2008.

[5] Y. Bu, V. Borkar, M. J. Carey, and T. Condie. Pregelix: Big(ger) graph analytics on a dataflow engine. *Proc. VLDB Endow.*, 7, 2015.

[6] M. Carbin, D. Kim, S. Misailovic, and M. C. Rinard. Proving acceptability properties of relaxed nondeterministic approximate programs. In *PLDI*, pages 169–180, 2012.

[7] K. Chakrabarti, M. Garofalakis, R. Rastogi, and K. Shim. Approximate query processing using wavelets. *The VLDB Journal*, 10(2-3):199–223, 2001.

[8] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *CAV*, pages 154–169, 2000.

[9] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, pages 238–252, 1977.

[10] W. Fan, J. Li, X. Wang, and Y. Wu. Query preserving graph compression. In *SIGMOD*, pages 157–168, 2012.

[11] W. Fan, X. Wang, and Y. Wu. Querying big graphs within bounded resources. In *SIGMOD*, pages 301–312, 2014.

[12] 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.

[13] J. M. Hellerstein, P. J. Haas, and H. J. Wang. Online aggregation. In *SIGMOD*, pages 171–182, 1997.

[14] Y. E. Ioannidis and V. Poosala. Histogram-based approximation of set-valued query-answers. In *VLDB*, pages 174–185, 1999.

[15] H. Kwak, C. Lee, H. Park, and S. Moon. What is Twitter, a social network or a news media? In *WWW*, pages 591–600, 2010.

[16] A. Kyrola, G. Blelloch, and C. Guestrin. GraphChi: Large-scale graph computation on just a PC. In *OSDI*, pages 31–46, 2012.

[17] A. Kyrola and C. Guestrin. GraphChi-DB: Simple design for a scalable graph database system – on just a PC. http://arxiv.org/pdf/1403.0701v1.pdf.

[18] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. GraphLab: A new parallel framework for machine learning. In *Conference on Uncertainty in Artificial Intelligence (UAI)*, pages 340–349, 2010.

[19] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. Distributed GraphLab: A framework for machine learning in the cloud. *PVLDB*, 5(8):716–727, 2012.

[20] 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.

[21] S. Navlakha, R. Rastogi, and N. Shrivastava. Graph summarization with bounded error. In *SIGMOD*, pages 419–432, 2008.

[22] D. Nguyen, A. Lenharth, and K. Pingali. A lightweight infrastructure for graph analytics. In *SOSP*, pages 456–471, 2013.

[23] A. Roy, I. Mihailovic, and W. Zwaenepoel. X-Stream: Edge-centric graph processing using streaming partitions. In *SOSP*, pages 472–488, 2013.

[24] S. Salihoglu and J. Widom. GPS: A graph processing system. In *Scientific and Statistical Database Management*, July 2013.

[25] H. Toivonen, F. Zhou, A. Hartikainen, and A. Hinkka. Compression of weighted graphs. In *KDD*, pages 965–973, 2011.

[26] K. Vora, S. C. Koduru, and R. Gupta. ASPIRE: Exploiting asynchronous parallelism in iterative algorithms using a relaxed consistency based DSM. In *OOPSLA*, pages 861–878, 2014.

[27] D. J. Watts and S. H. Strogatz. Collective dynamics of 'small-world' networks. *Nature*, 393(6684):440–442, 1998.

[28] Y. Zhou, H. Cheng, and J. X. Yu. Graph clustering based on structural/attribute similarities. *Proc. VLDB Endow.*, 2(1):718–729, 2009.

[29] X. Zhu, W. Han, and W. Chen. GridGraph: Large scale graph processing on a single machine using 2-level hierarchical partitioning. In *USENIX ATC*, 2015.

[30] Z. A. Zhu, S. Misailovic, J. A. Kelner, and M. Rinard. Randomized accuracy-aware program transformations for efficient approximate computations. In *POPL*, pages 441–454, 2012.