Achieving Sub-second Pairwise Query over Evolving Graphs

Hongtao Chen*
Tsinghua University
Beijing, China
cht22@mails.tsinghua.edu.cn

Mingxing Zhang*†
Tsinghua University
Beijing, China
zhang_mingxing@mail.tsinghua.edu.cn

Ke Yang*
Tsinghua University & Beijing HaiZhi XingTu Technology Co., Ltd.
Beijing, China
yangke@stargraph.cn

Kang Chen
Tsinghua University
Beijing, China
chenkang@tsinghua.edu.cn

Albert Zomaya
University of Sydney
Sydney, Australia
albert.zomaya@sydney.edu.au

Yongwei Wu
Tsinghua University
Beijing, China
wuyw@tsinghua.edu.cn

Xuehai Qian
Purdue University
West Lafayette, USA
qian214@purdue.edu

ABSTRACT

Many real-time OLAP systems have been proposed to query evolving data with sub-second latency. Although this feature is highly attractive, it is very hard to be achieved on analytic graph queries that can only be answered after accessing every connected vertex. Fortunately, researchers recently observed that answering pairwise queries is enough for many real-world scenarios. These pairwise queries avoid the exhaustive nature and hence may only need to access a small portion of the graph. Obviously, the crux of achieving low latency is to what extent the system can eliminate unnecessary computations. This pruning process, according to our investigation, is usually achieved by estimating certain upper bounds of the query result in existing systems.

However, our evaluation results demonstrate that these existing upper-bound-only pruning techniques can only prune about half of the vertex activations, which is still far away from achieving the sub-second latency goal on large graphs. In contrast, we found that it is possible to substantially accelerate the processing if we are able to not only estimate the upper bounds, but also foresee a tighter **lower bound** for certain pairs of vertices in the graph. Our experiments show that only less than 1% of the vertices are activated via using this novel lower bound based pruning technique. Based on this observation, we build SGraph, a system that is able to answer dynamic pairwise queries over evolving graphs with sub-second latency. It can ingest millions of updates per second and simultaneously answer pairwise queries with a latency that is several orders of magnitude smaller than state-of-the-art systems.

*The first three authors contributed equally to this research.
†Corresponding Author: Mingxing Zhang (zhang_mingxing@mail.tsinghua.edu.cn)

ASPLOS '23, March 25–29, 2023, Vancouver, BC, Canada

ACM ISBN 978-1-4503-9916-6/23/03.
https://doi.org/10.1145/3575693.3576173

CCS CONCEPTS

• **Mathematics of computing** → **Paths and connectivity problems**; • **Computing methodologies** → *Distributed algorithms.*

KEYWORDS

pairwise query, triangle inequality, graph processing

ACM Reference Format:
Hongtao Chen, Mingxing Zhang, Ke Yang, Kang Chen, Albert Zomaya, Yongwei Wu, and Xuehai Qian. 2023. Achieving Sub-second Pairwise Query over Evolving Graphs. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '23), March 25–29, 2023, Vancouver, BC, Canada.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3575693.3576173

1 INTRODUCTION

1.1 Motivation

In recent years, the topic of real-time OLAP has gained a great deal of attention in the area of relational data analysis. Many real-time OLAP systems [17, 29, 61, 64] have been proposed to query **evolving data** with **sub-second latency**. Although this feature is highly attractive, it seems impossible to achieve the same capability on analytic graph queries. Typical graph applications terminate only after accessing every connected vertex (multiple times) and hence lead to large overhead in both computation and communication. This exhaustive nature (named by Xu et al.[59]) of graph applications precludes the possibility of achieving sub-second latency over large graphs. Fortunately, recent researchers [65] observed that instead of computing an exhaustive "one-to-all-the-others" single-source query, answering a "point-to-point" **pairwise query** is enough for many real-world scenarios. This observation opens the possibility of building real-time OLAP systems for large and evolving graph data.

As an illustration, Single Source Shortest Path (SSSP) is a typical single-source query application that computes the shortest paths from the source to *all* other destination vertices. In contrast, the dynamic "pairwise query" version of SSSP is called Point-to-Point

Shortest Path (PPSP), where the users are only interested in the shortest path between a pair of two arbitrary vertices. PPSP is not only the most important problem in a navigation system [57] over a dynamic road graph but also the basic building block of many high-level graph analysis applications in social/financial networks. For example, recommendation systems need to frequently examine the shortest path between two arbitrary users in a large network extracted from the shopping logs, and the risk detection systems use PPSP to measure the distance between a newly-arrived transaction and certain risk users [22, 28, 35].

Since both the source and destination vertices are given in pairwise queries, the corresponding evaluation algorithm only needs to achieve convergence for this specific pair of vertices. It avoids the exhaustive nature and hence enables the query system to achieve a much smaller latency.

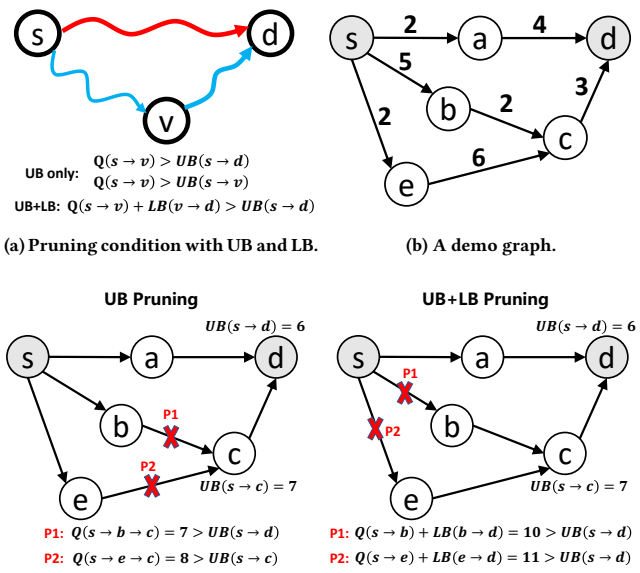
1.2 Challenges and Observations

The key to building efficient evolving graph and pairwise query processing systems is to what extent the system can eliminate wasteful computations. One prominent direction is to make use of the widespread monotonicity of graph applications. According to our investigation, the pruning techniques proposed by most of the existing evolving graph or pairwise query processing systems [18, 26, 56, 59] can be viewed as a leveraging of this simple property. Simply speaking, given source vertex s and destination vertex d, a monotonic graph application attaches a property $Q(s \mapsto v)$ to every vertex v. Then, the algorithm starts from initializing $Q(s \mapsto s) := 0$ and $Q(s \mapsto v) := inf$ for every v except s. These properties are updated in a monotonically decreasing manner through the iterations of the algorithm before $Q(s \mapsto d)$ stabilizing to its final value[1].

Leveraging this property, existing works prune unnecessary updates and computations by establishing an **upper bound** of $Q(s \mapsto v)$ for some or all of the vertices via prior knowledge obtained before [26], or during the query [59], or both. As shown in Figure 1 (a), an update to vertex v can be ignored if it is already larger than $UB(s \mapsto v)$, the current estimated upper bound from the source to vertex v, or $UB(s \mapsto d)$, the current estimated upper bound from the source to the final destination. In PnP [59], these upper bounds are established using the current intermediate result of $Q(s \mapsto v)$ or $Q(s \mapsto d)$. More detailed and sophisticated approaches to estimating upper bounds will be discussed in Section 2.2.

However, upper-bound-only pruning is still not enough to achieve the sub-second latency goal on large graphs. To demonstrate the problem, we present both analysis and evaluation results with the above PPSP example. Theoretically, suppose the queried vertices are randomly selected from the graph, then in expectation, half of the vertices will have a smaller distance from the source than the queried destination. Those vertices can never be pruned thoroughly via an upper-bound-only technique, hence have to be activated at least once and possibly multiple times before convergence. We also validate this analysis by running our implementation on several real-world datasets and counting the number of activated vertices in each iteration. Results show that pruning only half of vertex activations is still several orders of magnitude away from the desired performance. More details about the evaluation will be given in Section 5.3.

[1]Without loss of generality, we concentrate only on monotonically decreasing problems. All the discussion results still hold for monotonically increasing problems by simply interchanging decreasing with increasing, larger with smaller, and positive with negative.



(a) Pruning condition with UB and LB. (b) A demo graph.

(c) PPSP pruning demo for UB-only (left) and UB+LB (right).

Figure 1: An illustration of using Upper Bound (UB) and Lower Bound (LB) to prune unnecessary computations, assuming optimal bounds can be derived.

To resolve this problem, we study the characteristics of wasted activations, i.e., those vertex activations that are not pruned but actually do not contribute to the final result of the query. Results show that these upper bound estimations only make use of the current history of computation, but a capability to foresee the future is required to reach the desired performance. As an illustration, Figure 1(a) visualizes the pruning condition using a triangle of source s, destination d, and an intermediate vertex v. The monotonicity of the graph applications tells us that this activation on v can be omitted if $Q(s \mapsto v)$ plus the lower bound of the distance between v and d, denoted as $LB(v \mapsto d)$, is larger than the upper bound condition stated by $UB(s \mapsto d)$. There are three variables in this condition, where both $Q(s \mapsto v)$ and $UB(s \mapsto d)$ can be estimated by the current processing history. In contrast, the value of $LB(v \mapsto d)$ is related to the future path from v to d that has not been accessed. As a result, since existing works have no hints on the future path, the current upper-bound-only method can be viewed as a conserved form of the generalized formulation, assuming that $LB(v \mapsto d)$ can be as small as zero, which is the **key reason for wasted activations**.

In contrast, it is possible to substantially accelerate the processing if we are able to **foresee a tighter lower bound of** $LB(v \mapsto d)$. Theoretically, if both the lower bound and upper bound estimation are accurate, all the vertices that are not on the best paths between

source and destination, i.e. do not contribute to the final results, can be skipped. We demonstrate this advantage via the hypothetical graph described in Figure 1 (b-c). As we can see, using the upper bound can only prune the activation to vertex c because *1)* the update through $s \mapsto e \mapsto c$ is larger than the upper bound from s to c; and *2)* the update through $s \mapsto b \mapsto c$, though no larger than $UB(s \mapsto c)$, is larger than the upper bound from s to the final destination d. In contrast, if a tight lower bound is given, theoretically, only the activation through $s \mapsto a \mapsto d$ will be allowed because only vertex a is on the critical/shortest path to the final destination. Both activations from s to b and e are pruned due to the tighter pruning condition. Our evaluation results also demonstrate that it is possible to forecast lower bounds that are tight enough in practice. Only less than 1% of the vertices are activated via using our novel lower bound based pruning technique.

1.3 Our Contributions

To bridge the gap between large graph size and strict latency requirement, in this paper, we present the design and evaluation results of SGraph, which is a lower bound based system that can answer dynamic pairwise queries over evolving graphs with sub-second latency. It can ingest millions of updates per second and simultaneously answer pairwise queries with a latency that is several orders of magnitude smaller than state-of-the-art systems.

The crux of our optimization is a novel technique for predicting the lower bounds and a corresponding pruning technique that can prune more than 99% of vertex activations with both upper and lower bounds. A certain amount of prior knowledge is needed to get a good estimation of lower and upper bounds. We design a hub-based solution that consumes only $O(|V|)$ additional space and can be maintained efficiently upon graph updates. Similar to existing works [18, 26, 56, 59], the correctness and generality of our technique are based on the observation that the algorithm of most important pairwise graph queries will iteratively update the property of vertices in a monotonic way that obeys the triangle inequality. A detailed discussion about this assumption is given in Section 2.2.

In order to implement the above optimization for graph query and simultaneously support concurrent graph mutations, SGraph proposes a decoupled architecture and corresponding storage format that directly supports snapshot isolation over adjacency lists. It enables low query latency and high ingesting throughput simultaneously and stores only two additional meta bits for every edge. Our approach takes advantage of the unique property of SGraph that only two recent snapshots are needed during the processing. SGraph also implements the ad-hoc calculation of only needed upper/lower bounds, which is very important for the query performance. Otherwise, the query latency may be dominated by the pre-computation stage since most vertices will not be visited with our powerful pruning method. More details can be found in Section 3.2.

Moreover, existing graph processing frameworks, either vertex-centric [20, 36, 47, 67], edge-centric [43, 68], or graph-centric [48, 54, 60], only hide underlying system implementation from users. However, since there are many different kinds of pairwise queries (e.g., reachability, BFS, shortest path, widest path, label propagation), and the specific business logic of each application may also differ in many details (e.g., different edge/vertex label/property filters), it would be very complicated and cumbersome if users need to design and implement a specific upper/lower bound maintaining and pruning logic for every different kind of these queries. As a result, SGraph proposes a higher-level abstraction that can hide both system implementation and algorithm logic of the pruning technique from users. Users only need to specify the properties of the application, typically in only a few lines of code. Then SGraph will automatically deduce all the logic for prior knowledge generation and maintenance, as well as pruning the query.

Evaluation results on different kinds of pairwise queries and real-world datasets show that SGraph can ingest millions of graph updates per second and simultaneously answer dynamic pairwise queries with sub-second-level latency, which is several orders of magnitude smaller than our baseline implementations of the state-of-the-art pruning techniques (e.g., PnP [59], Tripoline [26]) and industry graph databases (e.g., Neo4j [3], TuGraph [4]). Experiment results show that SGraph is able to answer pairwise queries via only activating less than 1% of the vertices, which demonstrates the effectiveness of our lower bound based pruning technique.

2 CASE STUDY: POINT TO POINT SHORTEST PATH

In this section, we present the intuition of our system via a thorough case study on PPSP, a typical example of pairwise queries. We start with a standard SSSP algorithm that is exhaustive and works only on static graphs, and then demonstrate the procedure of adapting this simple algorithm step by step. The resultant solution is able to dynamically maintain a bunch of upper and lower bounds upon graph updates and use them to largely prune unnecessary computations. We will also discuss the principle underneath this adaption procedure, which is then generalized and formalized into the high-level programming model described in Section 3.1.

2.1 PPSP Problem

Given a pair of source vertex s and destination vertex d, the goal of *Point-to-Point Shortest Path (PPSP)* is to compute the distance of the shortest path from s to d. As a pairwise version of the SSSP problem, although it is exhaustive and expensive, PPSP can also be calculated by the same program of SSSP. Specifically, we attach a property **ub** to every vertex v representing the current observed shortest distance from s to v. We use *ub* because $v.ub$ can also be viewed as the current upper bound of $Q(s \rightarrow v)$. Then, these vertex properties can be initialized and updated by the simple vertex-centric program described in Listing 1.

This is a simple iterating algorithm that starts from activating the source vertex and terminates when there are no more activations. The only difference in the programming model is that we purposely separate the pruning logic from the main vertex program (i.e., the *TryActivate* function). Currently, an update can only be pruned if it violates the **monotonicity** of the application, i.e., this update is already larger than the current upper bound (line 10 of Listing 1).

```
1  func Init(v Vertex, q Query) {
2    // Init ub with infinity
3    v.ub = inf;
4    // Unless it is the source vertex
5    if (v.id == q.source.id)
6      v.ub = 0;
7  }
8  func TryActivate(v Vertex, dis Value) {
9    // Prune non-monotonic updates
10   if (dis > v.ub) return;
11   OnActivate(v, dis);
12 }
13 func OnActivate(v Vertex, dis Value) {
14   v.ub = dis;
15   for e := v.outgoingEdges() {
16     newdis = v.ub + e.weight;
17     TryActivate(e.target, newdis);
18   }
19 }
```

Listing 1: Vertex Program for PPSP.

2.2 Triangle Inequality

Similar to many existing works [18, 26, 56, 59], we observed that most of the monotonic graph applications also follow another important property that is usually described as "Triangle Inequality". This inequality is originally from Euclidean geometry. It states the fact that, for any given triangle, the sum of the lengths of any two sides must be greater than or equal to the length of the third side. Equivalently, the minus of the lengths of any two sides must be smaller than or equal to the length of the third side. This principle can be extended to not only Euclidean geometry but also many important pairwise graph queries by generalizing the above sum/minus/greater_or_equal operators.

Definition 1. *Given a pairwise query $Q(s \mapsto d)$, it follows the triangle inequality if we can define a triplet of binary operators $(\oplus, \ominus, \geq)$ such that the final converged results of $Q(* \mapsto *)$ satisfy*

- $Q(s \mapsto v) \ \oplus \ Q(v \mapsto d) \ \geq \ Q(s \mapsto d)$
- $Q(s \mapsto d) \ \geq \ Q(v \mapsto d) \ \ominus \ Q(v \mapsto s)$

Obviously, PPSP follows the triangle inequality. If the final result of $Q(s \mapsto v)$ plus $Q(v \mapsto d)$ is not greater or equal to $Q(s \mapsto d)$, we can simply choose this path $s \mapsto v \mapsto d$ to obtain a shorter path. Similarly, the second $\ominus$ clause of the inequality is just an equivalent transformation of $Q(v \mapsto s) \oplus Q(s \mapsto d) \ \geq \ Q(v \mapsto d)$. It must hold because otherwise, the result of $Q(v \mapsto d)$ can be updated and hence is not converged. It is demonstrated by existing works [18, 26, 56, 59] that many important pairwise queries, such as reachability, BFS, widest/narrowest path, Viterbi, Radii, etc., also obey this inequality. For these queries, one just needs to override the definition of these abstract sum/minus/greater operators $\oplus/\ominus/\geq$, other than simply using arithmetic $+/-/\geq$ in PPSP.

The first $\oplus$ clause in this definition has been widely used in existing works to obtain a **tighter upper bound** of the query. For example, Tripoline [26], VRGQ [25], and Quegel [65] can optimize pair-wise queries by using hub-based methods to estimate the upper bounds. In VRGQ (for static graphs) and Tripoline (for insertion-only dynamic graphs), these upper bounds are maintained by choosing one or more hub vertices h and calculating $Q(h \mapsto *)$ and $Q(* \mapsto h)$. $Q(h \mapsto *)$ and $Q(* \mapsto h)$ mean that the result of $Q(h \mapsto v)$ and $Q(v \mapsto h)$ for every vertex v is calculated beforehand and stored alongside vertex v. Then, in the initializing period of answering $Q(s \mapsto d)$, these pre-calculated results will be used to initialize the *ub* property with a tighter estimated upper bound. Specifically, for every vertex v except the source s, *v.ub* is initialized to $Q(s \mapsto h) \oplus Q(h \mapsto v)$ instead of infinity. This procedure is formalized in lines 4-6 of the *PostInit* function described in Listing 2, which is executed right after the execution of the aforementioned *Init* function.

Similar ideas have also been applied to incremental graph computing systems. These systems try to dynamically maintain the results of $Q(s \mapsto *)$, e.g., the shortest paths from vertex s to all other vertices. Each time graph updates invalidate (part of) the results, systems like KickStarter [56] and RisGraph [18] re-calculate $Q(s \mapsto *)$ from a tight upper bound to drastically speed up the convergence of the computation.

```
1  // The hub vertex
2  Global h Vertex;
3  func PostInit(v Vertex, q Query) {
4    // Tighter upper bound via h
5    ub = Q(q.source, h) + Q(h, v);
6    if ub < v.ub: v.ub = ub;
7    // Tighter lower bound via h
8    v.lb = 0;
9    lb = Q(h, q.dest) - Q(h, v);
10   if v.lb < lb: v.lb = lb;
11   lb = Q(v, h) - Q(q.dest, h);
12   if v.lb < lb: v.lb = lb;
13 }
14 func TryActivate(v V, dis Value, q Query) {
15   if (dis > v.ub) return;
16   // Prune with lower bound
17   if (dis + v.lb > q.dest.ub) return;
18   OnActivate(v, dis);
19 }
```

Listing 2: Pruning Logic for PPSP.

2.3 Pruning with Lower Bounds

In contrast to the widely-used $\oplus$ clause, according to our investigation, the second $\ominus$ clause of the definition is usually ignored in existing works because it seems to be equivalent to the first clause. However, we found that this ignored clause is, in fact, much more effective with respect to pruning.

As mentioned in Section 1.2, the crux of avoiding wasted activations to vertex v is to foresee a **tighter lower bound** of $Q(v \mapsto d)$ rather than simply using 0. Although the current execution has not explored the path from v to d, this is actually achievable without maintaining any more pre-calculation results other than $Q(h \mapsto *)$ and $Q(* \mapsto h)$. Firstly we can directly infer from the $\ominus$ clause that $Q(v \mapsto d) \ \geq \ Q(h \mapsto d) \ \ominus \ Q(h \mapsto v)$. Secondly, we can also infer another lower bound condition $Q(v \mapsto d) \ \geq \ Q(v \mapsto h) \ \ominus \ Q(d \mapsto h)$ because if this condition does not hold, $v \mapsto d \mapsto h$ will be a path shorter than $v \mapsto h$. With these lower bounds, we can prune more activations by avoiding "$(update\ to\ v) \ \oplus \ LowerBound(v \mapsto d) \ \geq \ UpperBound(s \mapsto d)$".

This logic is formalized in lines 7-12 and 16-17 of Listing 2. As we can see from the algorithm, we attach an additional lower bound property *lb* to vertex v, which is calculated by the above inequalities and the pre-calculated results. Moreover, we add an additional prune condition in the *TryActivate* function to ensure that a vertex is not activated unless it is possible to contribute to the final result of $Q(s \mapsto d)$.

More importantly, we found that Listing 2 outlines a general template for adapting a simple exhaustive single-source query algorithm to an efficient pairwise query algorithm. For applications other than PPSP, one just needs to override the definition of $+/-/\geq$ for type *Value* to the real logic of the abstract operator $\oplus/\ominus/\succeq$. According to our experiments, this overriding can be implemented in only a few lines of code.

2.4 Micro Benchmarks

In order to demonstrate the effectiveness of our novel lower bound based pruning technique, we measure the number of active vertices in PPSP for each iteration over several real-world graphs. As we can see from Figure 2, after using lower bounds, the number of active vertices becomes much smaller than using upper bounds only after about five iterations. In fact, as we will demonstrate later in Section 5.3, the total number of active vertices is about 50% of the total number of vertices in the graph if only upper bounds are used. In comparison, less than 1% of vertices are activated in our novel approach, leading to a significant speedup.

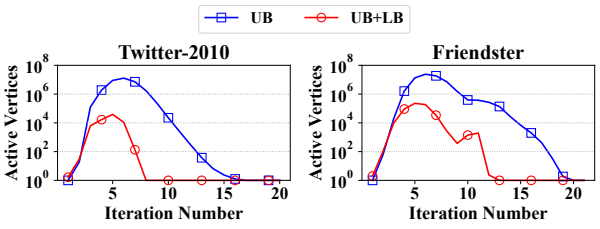


Figure 2: The number of active vertices on each iteration of PPSP query with and without lower bounds.

3 SYSTEM DESIGN

Following the intuition described above, we design and implement an evolving graph processing system called SGraph, which can accelerate the processing of dynamic pairwise queries via both upper and lower bounds. With a higher-level program abstraction, SGraph can support all the graph applications that iteratively update the property of vertex in a monotonic way that obeys the triangle inequality. In this section, we will first introduce SGraph's programming model, and then the main workflow and the corresponding decoupled architecture of SGraph. Finally, we will discuss the necessary system optimizations that are used in SGraph to achieve sub-second query latency.

3.1 Programming Model

Similar to the versatility of OLAP queries, there are many different kinds of pairwise queries, and the specific business logic of each application may also differ in many details. The originally designed upper and lower bounds may become completely useless if the business logic changes a bit by adding an edge (or label or property) filter to prohibit the use of a certain portion of the edges. It would be very complicated and cumbersome if users needed to design and implement a specific upper/lower bound maintaining and pruning logic for every different kind of these queries. As a result, the users desire a generalized abstraction that hides not only the system implementation but also the algorithm logic of the pruning technique from users. The system should be able to automatically generate the building and maintenance logic of upper/lower bounds for each different kind of query.

To this end, we provide a simple vertex-centric programming model to users, which is almost identical to the classical vertex program abstraction. As we can see from Listing 3, the vertex program *OnActivate* has two input variables. Variable v with type *Vertex* provides access to the vertex property and in-going/out-going edges of this vertex. The other *update* variable with type *Value*, which is the same type of vertex property, is the updating information sent from another vertex. This notification is achieved using the provided *TryActivate* function, as shown by Line 7 of Listing 3.

Users of SGraph can express their algorithm and business logic in this vertex program straightforwardly without considering how to avoid the exhaustive nature that prohibits it from finishing in sub-second latency. The initializing (executed in the *Init* function) and pruning logic (executed in *TryActivate* before actually invoking *OnActivate*) are all generated automatically by SGraph. This generation is possible if the user **specifies the triangle inequality** of vertex property for SGraph. Specifically, the user needs to provide five simple functions related to type *Value*, representing different aspects of the triangle inequality. As demonstrated by Listing 3, these five functions express the logic of the aforementioned $0/inf/\oplus/\ominus/\succeq$, respectively. From our experience, they can all be implemented in one or two lines of code in practice.

```
1  func OnActivate(v Vertex, update Value) {
2    for e := v.outgoingEdges() {
3      // Send update to e's target vertex
4      ...; TryActivate(e.target, ....); ...;
5    }
6  }
7  // Expressing the triangle inequility
8  func Zero(): Value { ... }
9  func Inf(): Value { ... }
10 func Add(l Value, r Value): Value { ... }
11 func Minus(l Value, r Value): Value { ... }
12 func LargerOrEqual(l Value, r Value): Bool {
13   ...
14 }
```

Listing 3: Programming Model of SGraph.

With these definitions, the initializing, dependency tracking, incremental maintaining, and pruning procedures of this application are all automatically generated by SGraph. As mentioned before, both the upper and lower bounds can be maintained by hub-based approaches. With a selected hub vertex h, it is enough to calculate all the needed upper and lower bounds by pre-calculating $Q(h \mapsto *)$ and $Q(* \mapsto h)$ (named hub indexes for brevity). For static graphs,

these hub indexes can be calculated directly by the provided *OnActivate* function with an empty pruning logic in *TryActivate*. Thus, we can use this user-provided function to initialize the hub indexes without any further user inputs. Moreover, the indexes can be maintained very efficiently over evolving graphs. It can be proved that once a graph application follows the triangle inequality, it also follows the constraints specified in KickStarter [56]. As a result, the indexes can be maintained by the same mechanism as KickStarter.

Specifically, since all the graph applications we use are monotonic graph algorithms, if only edge/vertex adding are given, the hub indexes $Q(h \mapsto *)$ and $Q(* \mapsto h)$ can be maintained by simply incrementally processing over the old results. The old values right before the updates will serve as a good approximation of the actual results and, hence, it is quicker to reach convergence. In contrast, for edge/vertex deletions, our system will automatically track a tree-structure dependency relationship of these indexes, just like KickStarter. This tree-structure dependency will be used to identify old values that are (directly or transitively) impacted by vertex/edge deletions and adjust those values before they are fed to the subsequent computation. After the adjustment, the trimmed approximation results can be used as a good approximation for accelerating the convergence of maintaining the hub indexes. With the above procedure, SGraph can work only on a subset of vertices impacted by added/deleted edges and restore the correct results efficiently by re-executing *OnActivate* from approximate intermediate results. More optimizations are discussed in Section 3.3.

3.2 Decoupled System Architecture

Similar to the graph databases, SGraph needs to process dynamic graph queries and concurrent graph mutations simultaneously. It would unnecessarily enlarge the query latency and limit the throughput if SGraph processes these queries and mutations in a series mode. Fortunately, we found that even though the arrivals of queries and mutations interleave with each other, it is enough to provide a consistent snapshot of dynamic graphs for each query in many real-world scenarios. This property naturally enables us to decouple the procedure of query processing from graph mutation and upper/lower bounds maintenance. To make use of this advantage, this section presents the main workflow and the corresponding decoupled architecture of SGraph, which is also the key point of achieving low latency and high throughput simultaneously.

As shown in Figure 3, SGraph incrementally maintains a series of snapshots of the graph. The last **closed** Snapshot x is a static version of the graph that contains both a consistent view of the graph and the corresponding indexes. In contrast, the **unclosed** Snapshot x+1 contains only a consistent view of the graph without ready indexes, and the later Snapshot x+2 is still **open** for graph updating. All the dynamic queries are answered upon the most recent closed Snapshot at their arrival time, so that they are not blocked by graph mutations and indexes maintenance. An independent procedure is used to compute the indexes for the unclosed Snapshot x+1 upon the recent graph mutations. As discussed in the above section, such index maintaining procedure is processed incrementally by using the remained indexes in Snapshot x that are not trimmed by the dependency relationship. Since Snapshot x+1 can be transferred to a closed snapshot for serving the queries as long as the index

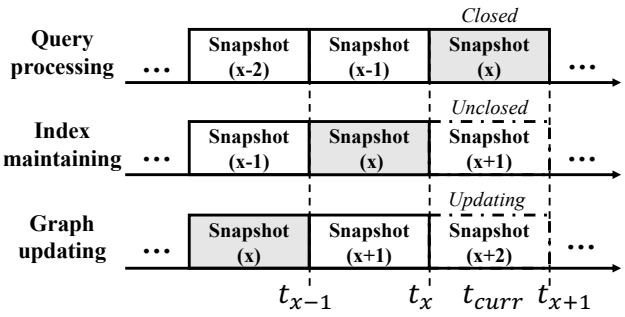


Figure 3: The overview of SGraph's snapshot based decoupled system architecture.

maintaining procedure is finished, the freshness of the query data is only related to the latency of this maintaining algorithm. As we will show in Section 5.4, such freshness can be limited to second-level upon a throughput of millions of mutations per second, which is acceptable in most real-world scenarios. Once an unclosed snapshot finishes its index maintenance, it becomes a closed snapshot, and the current snapshot open for graph mutation is transferred to a new unclosed snapshot.

Storage Design. In order to implement the above decoupled architecture, SGraph needs a multi-version graph storage that supports snapshot isolation. As an illustration, GraphOne [31] provides a hybrid store that utilizes both an edge log and an adjacency list. Graph mutations are first preserved in a circular buffer based edge log to achieve high ingestion throughput. These mutations will be periodically archived into an adjacency list that keeps all the neighbors of a vertex together and indexed by their source vertex, which provides efficient data access for graph processing. Since the cost of this archiving is relatively high, only coarse-grained snapshots can be provided if only the data in the adjacency list are used in the processing. To support fine-grained snapshots, GraphOne combines the per-vertex adjacency list with the shared edge log to obtain a complete edge set during the processing. However, as we will show with more experiment results later in Section 5.4, a larger number of graph mutations buffered in the edge log leads to both a higher ingestion throughput and a lower query performance, because of the read amplification problem caused by scanning the edge log.

To mitigate this problem, SGraph designs a mechanism that directly supports snapshot isolation over the adjacency list. In SGraph, each vertex has its local adjacent edges stored continuously in a dynamic array. When the array size surpasses a certain threshold (set as 512 in our evaluation), an additional dense map [1] is attached to it for fast edge indexing during updating. This hash map is only used by the single graph mutation thread of the corresponding graph partition and hence does not need to be a concurrent hash map (different partitions can use different threads). In order to provide fine-grained consistent snapshots for the processing, *meta bits* are used to indicate whether an edge is visible in certain snapshots. Each edge uses 2 bits since only two recent snapshots (closed and unclosed) are needed in SGraph.

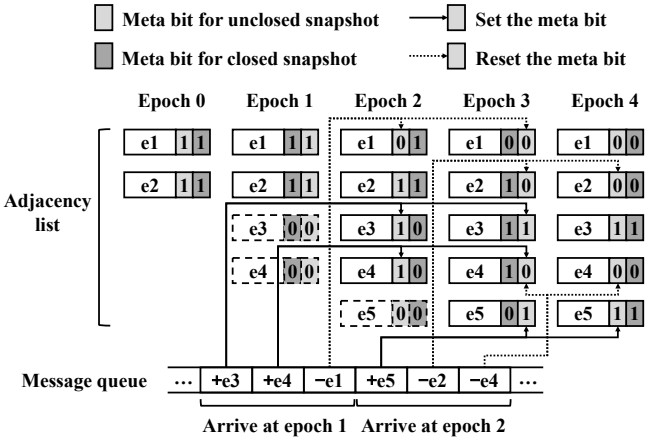


Figure 4: An example of updating the adjacency list of a single vertex in SGraph.

As discussed above, our goal is to *1)* provide consistent snapshots for both query processing and index maintaining threads; *2)* ensure that the query processing threads are never stalled by index maintaining or graph mutations; and *3)* maximize both the ingesting throughput and query performance.

In order to demonstrate our design, we use the example in Figure 4 to describe the update process of the adjacency list for a single vertex in SGraph. First, we assume that there are initially two edges (e1, e2) in the adjacency list before updating, and the system ingests graph mutations related to that vertex from a message queue. As shown in the figure, the meta bits of all these edges are initialized as *11*. Without loss of generality, we assume that the first meta bit is used to indicate its visibility in the closed snapshot at epoch 1. In this case, the query processing thread checks the visibility of an edge (i.e., whether it is in the current closed snapshot) by checking whether the first meta bit of this edge is set to *1*.

During epoch 1, the vertex receives three graph mutation messages to add e3 and e4, and to delete e1, denoted as +e3, +e4 and -e1. The graph mutation thread directly appends e3 and e4 to the adjacency list with meta bits set to *00* (represented by dashed boxes). Note that, during the mutation, index maintaining and query processing threads can simultaneously work on this adjacency list without any blocking, each seeing different snapshots. The edges with meta bits *00* are not visible to both the closed and unclosed snapshots.

After a certain fine-grained period of time, SGraph terminates epoch 1 and starts epoch 2 by **switching the meta bit indicating the closed snapshot from the first bit to the second bit**. After switching, the system can maintain the indexes for the current unclosed snapshot produced by epoch 1 through two steps. As demonstrated in the figure, SGraph first modifies the meta bits of the edges mutated in the last epoch, which changes the meta bits of e3 and e4 to *10* (visible in the unclosed snapshot but not in the closed snapshot) and e1 to *01* (visible in the closed snapshot but not in the unclosed snapshot). Then, SGraph can start the index maintaining algorithm because the maintaining threads can now identify the visible edges in the corresponding unclosed snapshot by checking whether the first meta bit is set. This whole maintaining procedure does not need to block the concurrent querying and ingesting, because the querying threads can check the second bit for visibility and the newly ingested edges (e.g., e5) are labeled with *00*.

After finishing the index maintaining, SGraph starts epoch 3 by switching the indicating bit once again from the second bit to the first bit, and hence the querying threads can make use of the newly maintained indexes. However, at this time, the first step of indexes maintenance (for the unclosed snapshot produced by epoch 2) needs to modify the meta bits of the edges mutated in the last **two** epochs. As a result, the meta bits of e1 are modified to *00* and its space can be recycled later because it will not be used any further; the meta bits of e3 are modified to *11*; the meta bits of e4 are first modified to *11* and then to *10* because e4 is added in epoch 1 and deleted in epoch 2 and hence is only visible in the current closed snapshot; the meta bits of e5 are modified to *01* because it is not visible in the current closed snapshot but visible in the current unclosed snapshot; and the meta bits of e2 are modified to *10* as opposite to e5. Again, these modifications of meta bits do not stall the query processing because they can use the first bit to identify the right closed snapshot. The maintenance procedure of the following epochs will be the same as epoch 3.

In conclusion, the meta bits of every edge will be constantly modified four times during its whole life cycle. From *00* to *01*/*10* to *11* for insertion and from *11* to *01*/*10* to *00* for deletion (*01* or *10* depends on which bit is currently used for indicating the closed snapshot). The whole process does not need to be protected by locks that would stall the query processing. With the carefully storing and manipulating of the meta bits, the producing and accessing of fine-grained snapshots brings very few space and time overhead.

3.3 System Implementation

SGraph is implemented in C++ with around 2000 lines of code, using OpenMP for multi-thread processing and OpenMPI for message passing. The underlying graph engine shares many system designs with state-of-the-art graph systems like Gemini [67], KickStarter [56], and RisGraph [18], such as 2-D graph partitioning, computation-communication overlapping, and multi-thread work stealing. Here we focus on introducing the system designs and tradeoffs specific to pairwise query processing on dynamic graphs.

Hub selection. To further accelerate the speed of indexes maintaining, SGraph maintains K hubs and tracks the dependency of their pre-processing results simultaneously. The advantages of using more than one hub are twofold: *1)* K hubs can derive K upper bounds and $2K$ lower bounds for each vertex. Thus tighter bounds can be selected to deliver more accurate pruning; *2)* although updates between two snapshots may invalidate many of the pre-processing results, the remained ones can still be used to provide upper/lower bounds that can accelerate the maintenance procedure. In SGraph, the K pre-processing results for each vertex are stored together for quick accessing and the updating of these indexes are scheduled in a co-located and coalesced way to reduce the maintaining overhead. Evaluation results show that, on average, maintaining indexes of 16 hubs takes only 3.23x more time than maintaining only a single hub.

If a vertex is not connected to a specific hub, then its corresponding upper bound and lower bound have to be set as inf and 0, respectively. This is a rare case in real-world graphs, since there usually exists a single large WCC/SCC that covers most of the vertices and many small WCC/SCC containing the rest vertices [50]. A recent study [30] shows that the WCC containing the vertex with the maximum degree covers 94.5% or more vertices for all the 15 graphs evaluated in their paper. So, similar to existing works [26], SGraph also intuitively selects K vertices with the highest degree as hubs.

Ad-hoc index calculation. Thanks to SGraph's efficient index-based pruning, during a query, only a tiny portion of vertices need to be activated in each iteration. However, the cost of calculating the indexes themselves can be $O(K|V|)$, if the bounds for all vertices are calculated for every query. This is obviously a huge waste when only a small portion of vertices are accessed during a query. To resolve this problem, SGraph proposes ad-hoc index calculation. Specifically, before starting the process, the lower and upper bounds of $Q(s \mapsto d)$ are first calculated through iterating all the hubs, which only takes $O(K)$ time. As we will show later in Section 5.3, SGraph can directly answer certain queries without accessing the graph if this lower bound equals the upper bound, which leads to extremely low latency. Then, during the processing, SGraph co-locate the hub-based index information with the vertex data. The results of both $Q(h \mapsto v)$ and $Q(v \mapsto h)$ for every hub h are stored with the vertex data of v. In this case, the upper and lower bounds of v can be calculated only when this vertex is activated, and hence omit the unnecessary calculation.

Triangle inequality based bi-directional search. Besides pruning, SGraph also proposes a novel triangle inequality based bi-directional search for pairwise query processing. For a query $Q(s \mapsto d)$, SGraph alternately computes $Q(s \mapsto *)$ and $Q(* \mapsto d)$ from forward and backward directions, respectively. SGraph limits the search space by pruning any vertex v that satisfies $UB(s \mapsto v) \oplus UB(s \mapsto v) \succ UB(s \mapsto d)$ in forward search and pruning any vertex v that satisfies $UB(v \mapsto d) \oplus UB(v \mapsto d) \succ UB(s \mapsto d)$ in backward search.

The bi-directional search does not only supplement the pruning conditions introduced in Section 2, e.g., $UB(s \mapsto v) \oplus LB(v \mapsto d) \succ UB(s \mapsto d)$ in forward search, but also explicitly constraints the search radius in both directions. Intuitively, in an N-dimensional search space, this reduces the search radius by half and the search space by 2^{N-1} times. As we will show later in Section 5.5, the synergy between bi-directional search and our novel lower bound based pruning technique dramatically boosts SGraph's query processing speed.

4 EXAMPLE APPLICATIONS

Besides the PPSP problem described in Section 2, we will present example applications in this section to demonstrate the generalizability and simplicity of SGraph's programming model. All the applications evaluated in this paper and their overridden operators are summarized in Table 1.

Table 1: Overridden operators for each application.

Application	$a \oplus b$	$a \ominus b$	$a \succeq b$
PPSP BFS	$a + b$	$a - b$	$a \geq b$
Reachability Connectivity	$a \wedge b$	$a \vee \neg b$	$a \rightarrow b$
PPWP	$min(a, b)$	$a < b \,?\, a : \infty$	$a \leq b$
PPNP	$max(a, b)$	$a > b \,?\, a : 0$	$a \geq b$
Viterbi	$a \times b$	$a \div b$	$a \leq b$

4.1 Unweighted Graph Queries

Connectivity, *Reachability* and *Breadth First Search (BFS)* are three of the most fundamental pairwise queries on unweighted graphs and are used by more complex applications like bi-connectivity [52], higher-order connectivity [6] and graph clustering [44]. Specifically, *1)* given an undirected graph, $Connectivity(s \mapsto d)$ checks whether there exists a path connecting vertex s and d; *2)* $Reachability(s \mapsto d)$ checks whether there exists a path from vertex s to d on a directed graph; and *3)* $BFS(s \mapsto d)$ searches the shortest path from vertex s to d, assuming that the weight of every edge equals to one.

Connectivity query can also be considered as a pairwise version of the exhaustive *Weakly Connected Component (WCC)* problem. Due to its importance, many works [16, 23, 24, 53] are dedicated to answering connectivity queries over dynamic graphs. However, most of these works need to dynamically maintain complicated dynamic data structures, such as spanning forests [5, 19, 49], that are not easy to implement in a distributed environment. Even worse, all these works take advantage of the unique commutative property of connectivity query. As a result, even the reachability query, a directed graph version of connectivity, cannot be processed efficiently with the above works. Instead, many separate works [15, 42, 66] are dedicated to the reachability query.

In contrast, with SGraph, both connectivity and reachability queries can be answered very efficiently via the same program. Here we take the reachability query as an example. $Q(u \mapsto v)$ could be either *True* or *False*, indicating whether v is reachable from u. Initially, the source vertex is initialized as *True*, while all other vertices are *False*. During processing, the *True* value is propagated to all reachable vertices from the source. This process can be dramatically accelerated by triangle inequality based pruning:

(1) **Upper bound pruning.** If h is reachable from u, and v is reachable from h, then v must be reachable from u, as shown in Figure 5(a).
(2) **Lower bound pruning.** If h can reach u but cannot reach v, then u cannot reach v, either. Otherwise, there would be a path from h to v via u (Figure 5(b)). Similarly, if h is reachable from v but not reachable from u, then u cannot reach v (Figure 5(c)).

The above pruning logic can be automatically derived from the user-defined operators $a \oplus b$, $a \ominus b$ and $a \succeq b$, overridden as $a \wedge b$, $a \vee \neg b$, and $a \rightarrow b$, respectively. In these equations, both a and b are boolean vertex properties, and $a \rightarrow b$ denotes the logical implication operator that "if a is True, then b must also be True". Based on them, SGraph generates the upper bound and lower bound

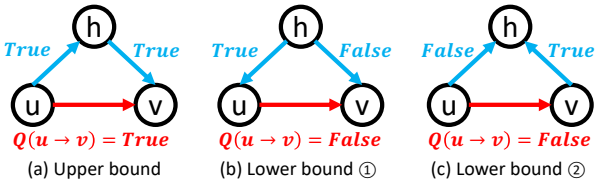


Figure 5: The triangle example of reachability query.

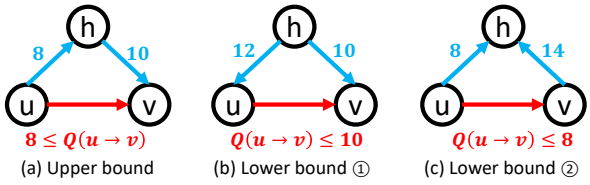


Figure 6: The triangle example of PPWP query.

pruning as:

$$Q(u \mapsto h) \wedge Q(h \mapsto v) \rightarrow Q(u \mapsto v) \quad \text{(a)}$$

$$Q(u \mapsto v) \rightarrow Q(h \mapsto v) \vee \neg Q(h \mapsto u) \quad \text{(b)}$$

$$Q(u \mapsto v) \rightarrow Q(u \mapsto h) \vee \neg Q(v \mapsto h) \quad \text{(c)}$$

The operators for the connectivity query are exactly the same as the reachability query. And the operators for the BFS query are similar to the PPSP query, with the edge weight fixed as one for all edges. We omit the details due to space limitations.

4.2 Weighted Graph Queries

Besides PPSP, *Point-to-Point Widest Path (PPWP)* and *Point-to-Point Narrowest Path (PPNP)* are two important pairwise queries that are widely used in many real-world analysis scenarios. For example, one may want to find the widest or the narrowest route between two cities in traffic planning [7, 41] or a path between two Internet nodes with maximum bandwidth [45]. To detect money laundering, one can model the transaction history as a huge graph and study the PPWP over two suspicious accounts [33, 51]. Similar to connectivity and reachability queries, PPWP and PPNP are also the fundamental kernels and subroutines of many important high-level graph analysis applications, such as the network flow algorithm [58].

Both PPWP and PPNP are defined on weighted graphs, where each edge is associated with a number as its weight. In PPWP, a pairwise query $Q(s \mapsto d)$ finds the path from s to d that maximizes the minimum-weight edge on the path, i.e., the widest path. As a dual problem of PPWP, in PPNP, $Q(s \mapsto d)$ finds the path from s to d with the maximum-weight edge on the path to be the minimum among all legal paths, i.e., the narrowest path.

With SGraph, the acceleration of PPWP and PPNP is rather straightforward by overriding the binary operators in triangle inequality. Here we take the PPWP query as an example. $Q(u \mapsto v)$ denotes the maximum path width from u to v. To answer query $Q(s \mapsto d)$, SGraph initializes the maximum known width from s to s as ∞ and from s to other vertices as 0. During processing, these values are gradually increased until convergence. Again, our triangle inequality based pruning can significantly speed up the convergence, as listed below:

(1) **Upper bound pruning.** For every path $u \mapsto h \mapsto v$, its width equals the narrower one of $u \mapsto h$ and $h \mapsto v$ (Figure 6(a)). Thus the widest path from u to v must be no narrower than this, which serves as the upper bound pruning in SGraph since the operator $\geq$ is overridden to $\leq$.

(2) **Lower bound pruning.** If the width of $h \mapsto u$ is larger than $h \mapsto v$, then $u \mapsto v$ must be no wider than $h \mapsto v$. Otherwise, $h \mapsto u \mapsto v$ would be wider than $h \mapsto v$, contradicting its widest path definition (Figure 6(b)). Similarly, if the width of $v \mapsto h$ is larger than $u \mapsto h$, then $u \mapsto v$ cannot be wider than $u \mapsto h$ (Figure 6(c)).

The above pruning logic can be automatically derived by overriding the operators $a \oplus b$, $a \ominus b$ and $a \geq b$ to as $min(a, b)$, $a < b\,?\,a : \infty$ and $a \leq b$, respectively. This implies the following pruning semantics:

$$min(Q(u \mapsto h), Q(h \mapsto v)) \leq Q(u \mapsto v) \quad \text{(a)}$$

$$Q(u \mapsto v) \leq (Q(h \mapsto v) < Q(h \mapsto u)\,?\,Q(h \mapsto v) : \infty) \quad \text{(b)}$$

$$Q(u \mapsto v) \leq (Q(u \mapsto h) < Q(v \mapsto h)\,?\,Q(u \mapsto h) : \infty) \quad \text{(c)}$$

The overridden operators of PPNP are dual to PPWP, as shown in Table 1.

4.3 Machine Learning Queries

In addition to traditional graph analysis applications, recently there are also many data mining and machine learning algorithms modeled and calculated as graph processing problems. One prominent example is the Viterbi algorithm (Viterbi) [34], which is widely used in convolutional code decoding [55], speech recognition [40], bioinformatics [63], etc [9, 14]. Viterbi searches for the most likely state sequence that has the maximum posterior probability, called the Viterbi path. Viterbi is defined on weighted graphs, where the weight of an edge denotes the conditional probability of a state, and the product of edge weight along a path denotes the conditional probability of a sequence of states. A Viterbi query $Q(s \mapsto d)$ finds the path from s to d with the maximum edge weight product. The pruning of the Viterbi query is similar to the PPSP query. With the operators $a \oplus b$, $a \ominus b$ and $a \geq b$ overridden as $a \times b$, $a \div b$ and $a \leq b$, respectively, SGraph automates the triangle inequality based pruning as:

$$Q(u \mapsto h) \times Q(h \mapsto v) \leq Q(u \mapsto v) \quad \text{(a)}$$

$$Q(u \mapsto v) \leq Q(h \mapsto v) \div Q(h \mapsto u) \quad \text{(b)}$$

$$Q(u \mapsto v) \leq Q(u \mapsto h) \div Q(v \mapsto h) \quad \text{(c)}$$

5 EVALUATION

Our experiments are set up on a 10-node cluster with 24 physical cores and 375GB of memory per machine. The machines are connected through a 200Gbps Infiniband Network.

Table 2 shows the real-world graph datasets used in the evaluation. Twitter-2010 and Friendster are large social network graphs. UK-2007-05 and Gsh-2015-host are large web crawl graphs. The average degree ranges from 52 to 71. Our evaluation is based on a versatile set of seven different applications, namely Shortest Path (PPSP), Breadth First Search (BFS), Reachability, Connectivity,

Widest Path (PPWP), Narrowest Path (PPNP), and Viterbi Algorithm (Viterbi). All these applications are important pairwise graph queries that *1)* not only themselves are widely used; *2)* but also the combination of these basic kernels can form many important high-level graph analysis applications such as clustering, classification, and prediction. In the following experiments, SGraph selects K=16 vertices with the highest degree as hubs. We run 1000 different queries for all the experiments and present the average results.

5.1 Query Performance

First, we compare the query performance of SGraph with PnP [59], Tripoline [26], Neo4j [3], and TuGraph [4]. PnP is currently, as far as we know, the fastest single-machine pairwise query processing system. It elaborates on the potential and importance of point-to-point queries and inspires our optimizations on SGraph. A comparison with our implementation of distributed PnP can be used to measure the effectiveness of our optimizations. In contrast, Tripoline is designed to use the triangle inequality property to accelerate the process of exhaustive "one-to-all-the-others" single-source queries. For a fair comparison, we extend Tripoline's upper bound based pruning with the pruning approach of PnP to efficiently answer pairwise queries, which is equivalent to the *UB pruning condition* introduced in Figure 1. We denote the extended version as *Tripoline+*. Since both PnP and Tripoline are not open source, we re-implement their mechanism based on Gemini [67] distributed graph processing framework, the same execution engine of SGraph.

Table 3 gives the average query processing latency for seven applications. As shown in the table, for all the applications other than BFS, SGraph can be much faster than PnP because of the indexes. Different from SGraph, PnP does not require any pre-computation and hence can always compute on the freshest data. However, this benefit also limits its performance when SGraph can take advantage of the maintained pre-computed triangle inequality based indexes. PnP's two-phase algorithm is also particularly effective on BFS, because the second phase can be fully skipped. In contrast, *1)* in PPSP/Viterbi/PPWP/PPNP queries, the second phase for queries whose destination vertex is reachable from the source vertex cannot be skipped; and *2)* in connectivity/reachability queries, though the second phase can still be skipped in PnP, SGraph is much more efficient since the queries can usually be answered by accessing only the indexes. More details can be found later in Section 5.3 where we compare the number of activated vertices between PnP and SGraph. The overhead of graph updating and index maintaining are evaluated in Section 5.4. Although SGraph is not able to provide the same freshness guarantee as PnP, thanks to our system and architectural optimizations, SGraph can provide sub-second-level freshness while ingesting millions of updates per second, which is sufficient in most real-world scenarios.

Table 2: Real-world graph datasets.

Graph	$\|V\|$	$\|E\|$	Type
Twitter-2010(TW) [32]	41.7M	1.47B	directed
Friendster(FS) [62]	65.6M	1.81B	undirected
UK-2007-05(UK) [12]	106M	3.74B	directed
Gsh-2015-host(GS) [10]	68.7M	1.80B	directed

Table 3: Query performance (time in milliseconds).

Algorithm	Graph	PnP	Tripoline+	SGraph
PPSP	TW	365 (10.9×)	312 (9.29×)	33.6
	FS	752 (18.0×)	757 (18.1×)	41.8
	UK	1471 (18.8×)	1656 (21.2×)	78.2
	GS	513 (16.6×)	518 (16.8×)	30.9
Viterbi	TW	384 (10.8×)	364 (10.3×)	35.4
	FS	725 (18.2×)	794 (19.9×)	39.9
	UK	984 (9.28×)	1214 (11.5×)	106
	GS	453 (13.0×)	471 (13.5×)	34.9
PPWP	TW	235 (79.1×)	10.1 (3.40×)	2.97
	FS	533 (1697×)	8.85 (28.2×)	0.314
	UK	1632 (85.4×)	225 (11.8×)	19.1
	GS	479 (63.3×)	42.5 (5.61×)	7.57
PPNP	TW	234 (75.7×)	11.4 (3.69×)	3.09
	FS	533 (1605×)	8.32 (25.1×)	0.332
	UK	1645 (87.5×)	244 (13.0×)	18.8
	GS	475 (65.4×)	40.7 (5.61×)	7.26
BFS	TW	22.1 (1.33×)	155 (9.34×)	16.6
	FS	41.7 (1.63×)	380 (14.8×)	25.6
	UK	114 (1.22×)	743 (7.93×)	93.7
	GS	30.3 (1.16×)	191 (7.32×)	26.1
Reachability	TW	16.0 (24.9×)	7.72 (12.0×)	0.642
	FS	29.0 (92.1×)	8.88 (28.2×)	0.315
	UK	109 (33.5×)	22.5 (6.92×)	3.25
	GS	22.2 (17.2×)	11.3 (8.76×)	1.29
Connectivity	TW	35.8 (117×)	6.16 (20.2×)	0.305
	FS	29.0 (92.1×)	8.88 (28.2×)	0.315
	UK	94.3 (289×)	12.3 (37.7×)	0.326
	GS	36.4 (111×)	8.28 (25.3×)	0.327

However, as shown by the comparison with Tripoline+, without the lower bound estimation, simply using indexes to estimate tighter upper bounds is still not enough. Tripoline+ can achieve a significant speedup on PPWP and PPNP, where the final result is a selection instead of an accumulation over the path. In such cases, the error of bound estimation is also not accumulated, and hence the bounds are tighter than in other applications. However, SGraph is still $3-28\times$ times faster than Tripoline+ in such cases. According to the evaluation of these existing systems, without lower bounds, neither *1)* altering the search direction; nor *2)* adding the upper-bound-only pruning methods, can lead to satisfactory performance. Furthermore, as we will demonstrate in the breakdown analysis later in Section 5.5, without lower bounds, a combination of the above two optimizations still performs several times slower than our final approach.

We also compare SGraph with Neo4j and TuGraph. The latter is reported to be the fastest industry graph database in LDBC SNB [2]. The results show that SGraph is $262-659\times$ faster than TuGraph for PPSP query, $249-378\times$ faster for Viterbi query, and more than

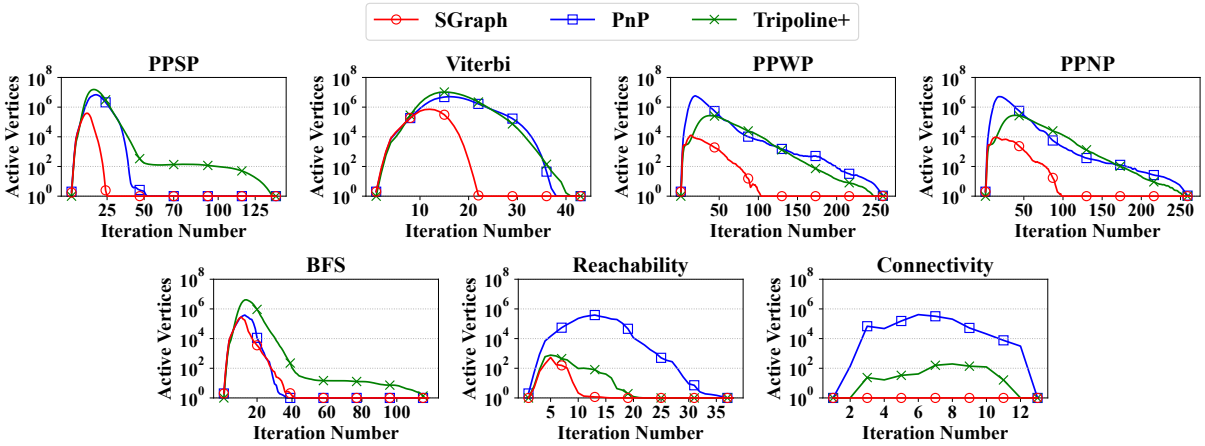


Figure 7: The average number of active vertices on each iteration during the query on UK-2007-05.

three orders of magnitude faster for the other five applications. The speedup over Neo4j is even larger than TuGraph. We omit their results in Table 3 as they are significantly slower than the other three systems.

5.2 Larger Graph Dataset

To evaluate the query performance of SGraph on larger graph datasets, we use UK-2014 [10, 11, 13] as an extension of UK-2007-05. It has similar properties to UK-2007-05, and has 788 million vertices and 47.6 billion edges, which is much larger than UK-2007-05. We use all seven applications to compare the query speedup of SGraph on UK-2007-05 and UK-2014, taking the faster one in PnP and Tripoline+ as a baseline for each application. As shown in figure 8, with the expansion of graph size, the speedup of PPSP/Viterbi/BFS query increases by $2-3\times$, because of SGraph 's capability of fast convergence; the speedup of Connectivity query increases by 5+ times, because SGraph can still answer most queries by accessing only the indexes; and the speedup of the other three applications also increases slightly. The results show that the pruning technique of SGraph works better with larger graph datasets, and thus can deal with massive data in complex real-world scenes.

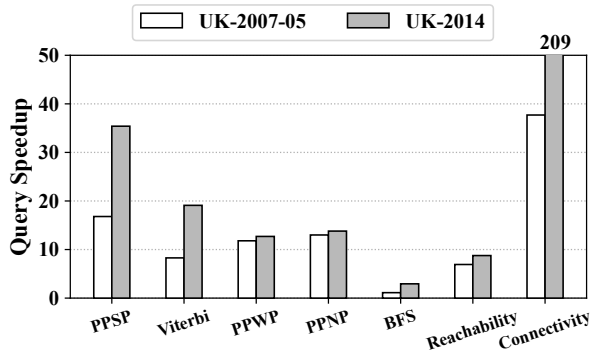


Figure 8: The query speedup of SGraph on UK-2007-05 and UK-2014.

5.3 Vertex Activation

To further verify the pruning effects of SGraph, We measure the activation ratio of each system and it is less than 1% for SGraph in most applications, which is 20+ times lower than other systems. Figure 7 presents the average number of active vertices on each iteration during the query on UK-2007-05, the largest graph in our datasets. As we can see, for all three systems, with the increase of iteration number, the number of active vertices first increases to the peak and then decreases, which also depicts the process of exponentially expanding the search space from the source vertex (or destination vertex, or both) and gradually converging. Compared with the other two systems, SGraph typically uses fewer iteration rounds to both reach the peak and converge, and the peak value of active vertices is several orders of magnitude smaller. This demonstrates that our lower bound based pruning technique significantly reduces search space.

The lower bound based pruning is particularly useful for PPWP, PPNP, reachability, and connectivity, resulting in an activation ratio very close to zero. For these queries, the upper bound of $Q(s \mapsto d)$ is only slightly larger or even equal to the lower bound of $Q(s \mapsto d)$, which is the key reason for the effectiveness of the lower bound. Even further, it is possible that $UB(s \mapsto d)$ is equal to $LB(s \mapsto d)$ in many cases, where the answer can be determined without visiting the graph (i.e., zero vertex activation). As for BFS, the pruning effects of upper/lower bounds are not that significant compared to the usage of bi-directional search. This is because that UK-2007-05 is a power graph that has a small diameter and many high-degree power vertices.

5.4 Update Performance

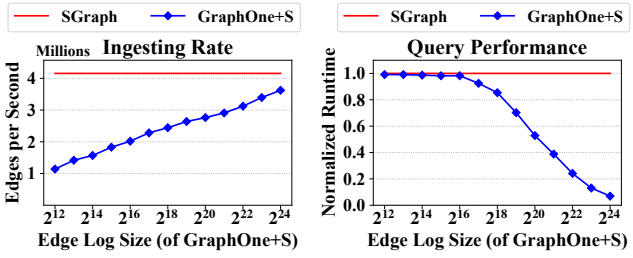
As described in Section 3.1, in order to maintain the lower and upper bounds over evolving graphs, we adopt a hub-based incremental processing technique. In this section, we evaluate the index maintain performance, which also shows the maximum throughput of graph updates SGraph can ingest. In our experiments, 70% of the edges are loaded as the initial graph, then randomly selected updates are streamed in batches. The batch size is set as 0.01M,

Table 4: Time (s) to maintain the indexes for each batch.

Algorithm	Batch Size	TW	FS	UK	GS
PPSP	0.01M	0.321	0.227	0.783	0.468
	0.1M	0.453	0.315	1.22	0.636
	1M	1.39	0.846	2.66	2.06
Viterbi	0.01M	0.325	0.231	0.807	0.475
	0.1M	0.462	0.309	0.940	0.636
	1M	1.42	0.822	2.24	1.82
PPWP	0.01M	0.320	0.246	0.808	0.565
	0.1M	0.471	0.427	1.35	0.964
	1M	1.89	1.56	3.82	4.10
PPNP	0.01M	0.306	0.260	0.782	0.537
	0.1M	0.497	0.440	1.33	0.918
	1M	1.96	1.60	3.70	3.96
BFS	0.01M	0.368	0.219	0.780	0.459
	0.1M	0.419	0.280	0.985	0.562
	1M	1.22	0.689	2.22	1.43
Reachability	0.01M	0.342	0.209	0.652	0.434
	0.1M	0.368	0.229	0.776	0.470
	1M	0.665	0.418	1.38	0.816
Connectivity	0.01M	0.163	0.209	0.332	0.230
	0.1M	0.174	0.229	0.356	0.232
	1M	0.309	0.418	0.541	0.450

0.1M, and 1M, respectively. Each batch has 50% additions and 50% deletions. Table 4 shows the average time cost of maintaining the indexes for each batch in SGraph. With the increase of the batch size, the per-batch processing time grows sub-linearly. i.e., the average time of each update decreases. This is because the indexes invalidated by different updates always overlap greatly.

Thanks to SGraph's decoupled architecture, the updates will not block queries, but only affect the freshness of query results. As we can see from Table 4, SGraph is able to provide sub-second level freshness while ingesting millions of updates per second for simple queries, and constrain the freshness within several seconds for the other complicated queries.



(a) Ingesting rate. (b) Query performance.

Figure 9: A comparison of the storage format of SGraph and GraphOne+S.

We also compare our storage format that enables SGraph's snapshot based decoupled architecture with GraphOne [31]. In our experiments, we use the same lower and upper bound based algorithm to process PPSP queries on Twitter-2010, but replace the storage format with GraphOne's hybrid design. The combined version is denoted as *GraphOne+S*. As we can see from Figure 9(a), with the increase of the number of buffered mutations in the shared edge logs, GraphOne+S can also ingest millions of updates per second. However, as shown in Figure 9(b), the query performance of GraphOne+S continuously decreases when the size of the edge log increases. This is because, in order to enable fine-grained snapshots in GraphOne+S, when answer a query, the system needs to scan the whole edge log once for each iteration to combine the graph mutations in the logs with the adjacency list. Since the vertex activation ratio is very low in SGraph, such repetitive scan leads to non-negligible unnecessary costs. Note that this does not mean that our method can lead to a better general-purpose dynamic graph storage format. It is specially designed for our decoupled architecture that the maximum ingesting rate is bounded by the speed of maintaining the indexes and only two rolling snapshots are needed.

5.5 Breakdown

We analyze the performance impact of the three main pruning methods in SGraph, i.e., UB for upper bound based pruning, LB for lower bound based pruning, and BS for bi-directional search. Table 5 reports the result of the breakdown test, taking PPSP as an example. Each entry is the average speedup over the baseline, which does not apply the above pruning methods and hence degenerates into a straightforward algorithm. As expected, the speedup of applying UB only is relatively low, since upper bound based pruning techniques can only prune the vertices with an upper bound larger than the known upper bound of the queried destination vertex. For BS and LB, the effect of BS differs a lot on different graphs. It accelerates the query significantly on Friendster, which is the only undirected graph, but has a smaller benefit on directed graphs. On the other hand, LB provides more stable accelerations. More importantly, it is really interesting to see that the synergy among all these three optimizations is the key reason for our significant speedup. Applying UB only with LB or only with BS will be several times slower than the final results, as the lower bounds can take effect in both directions and hence largely reduce the search space, leading to a combinatorial effect.

Table 5: PPSP speedups with different pruning methods.

Graph	UB	UB+BS	UB+LB	UB+BS+LB
TW	1.43×	3.28×	5.89×	20.7×
FS	1.66×	18.1×	4.12×	30.2×
UK	1.92×	2.11×	3.97×	34.8×
GS	2.04×	2.79×	4.79×	35.7×

5.6 Scalability

Finally, we evaluate the inter-node scalability of SGraph using PPSP, since it is one of the most time-consuming applications and thus

will benefit significantly from scalability. We process PPSP queries with 1, 2, 4, and 8 nodes on different graphs, respectively. Results show that SGraph with an 8-node cluster is about $4-6\times$ times faster than with a single machine, which demonstrates the good scalability of SGraph. For benchmarks where only a tiny portion of vertices will be activated (e.g., reachability), adding more nodes can also help hold larger graphs and speed up the ingestion.

6 RELATED WORKS

6.1 Dynamic Graphs

Evolving graphs have attracted a lot of attention in recent years [8]. Many systems have been proposed to provide timely responses to online analytic graph queries via incremental algorithms. For example, Tornado [46] observes that many analytic graph problems are solved by starting with an initial guess and repeatedly refining the solution. Loops starting from good initial guesses usually converge faster. Differential Dataflow [39] records the whole trajectory of past iterative processing procedures instead of only the former results. It is a generalized approach that extends incremental processing to iterative applications, not only iterative graph processing.

A common assumption of the former works is that the intermediate value of a previous version is indeed closer to the actual result than the initial value, even when the graph mutates. However, Vora et al. [56] found that this implicit assumption does not hold if the graph mutations include edge deletions. In such cases, the graph mutations may break the monotonicity of graph applications and invalidate the intermediate values being maintained. To resolve this problem, systems like KickStarter [56] and GraphBolt [37] carefully track dependencies between vertex values (that are being computed) and edge modifications. By recording these dependencies, they can efficiently identify values that are (directly or transitively) impacted by edge deletions and incrementally adjust those values before they are fed to the subsequent computation. A recent work RisGraph [18] also applies KickStarter's approach for incremental computation to its design based on concurrent ingestion of fine-grained updates and queries.

Comparison. While the details for achieving incremental changes vary across systems, according to our investigation, they all record a certain kind of intermediate results that help them converge faster or exclude part of the graph from re-computation. Moreover, the space complexity of such intermediate results is **at least $O(|V|)$ for each specific query**. Since these intermediate results cannot be reused if the query is changed, it is unacceptable to extend these methods naively to support dynamic queries.

A recent work Tripoline [26] tries to avoid this problem by using a hub-based approach and triangle inequality, which is similar to our work. However, as compared in Section 5.1, the lack of the lower bound estimation makes Tripoline several orders of magnitude slower than our system. Moreover, Tripoline only supports insertion-only graph mutations (i.e., no edge deletion) and is not optimized for pairwise query (cannot avoid the exhaustive nature inherited in the graph application).

6.2 Pairwise Queries

Recently, the unique property of pairwise queries has attracted interest from many researchers. For example, Hub^2 [27] proposes a specialized accelerator for PPSP queries that uses high degree vertices as hubs to accelerate the PPSP queries. To implement this algorithm in a distributed environment, Quegel [65] allows users to construct distributed graph indexes at graph loading. However, Quegel simply allows users to write arbitrary code for constructing static indexes at the beginning. All the cumbersome indexes building logic needs to be implemented by the users themselves. And it does not support dynamic graphs, once the graph is updated, the stored index may no longer be correct. There are also many theoretical efforts [21, 38] on accelerating dynamic pairwise graph queries via constructing and maintaining indexes structure like graph synopses and sketches. To the best of our knowledge, these algorithms cannot be implemented straightforwardly in existing distributed graph processing systems. PnP [59] is a recent single-machine system that specializes in processing pairwise queries. It proposes a generalized pruning technique that is applicable for many different kinds of workloads but also not based on lower bounds.

Comparision. As discussed above, many important kinds of dynamic graph queries have been supported in a static graph scenario, but the indexes needed by these algorithms are too complicated to be efficiently maintained when the graph mutates. Since all these algorithms are designed case by case for a certain kind of application, it is also hard to summarize a generalized form of building such indexes for a wide range of graph applications. More importantly, although some efforts have been made to propose a generalized framework for processing pairwise queries, they are based only on upper bounds.

7 CONCLUSION

In this paper, we present the design, implementation, and evaluation results of SGraph. Based on our novel lower bound based pruning technique, SGraph can ingest millions of updates per second and simultaneously answer pairwise queries with a latency that is several orders of magnitude smaller than state-of-the-art systems. Moreover, SGraph proposes a higher-level abstraction that can hide both system implementation and algorithm logic of pruning technique from users. Users only need to specify the properties of the application, typically in only one line of code. Then, SGraph will automatically deduce all the logic for prior knowledge generation and maintenance, as well as pruning for the query.

ACKNOWLEDGMENTS

We thank our shepherd and all the reviewers for their valuable comments and suggestions. This Work is supported by National Key Research & Development Program of China (2020YFC1522702), Natural Science Foundation of China (62141216, 61877035), Young Elite Scientists Sponsorship Program by CAST (2022-2024), Tsinghua University Initiative Scientific Research Program, Tsinghua University - Meituan Joint Institute for Digital Life, and Beijing HaiZhi XingTu Technology Co., Ltd.

REFERENCES

[1] [n.d.]. Google Dense Hashmap. Retrieved January 6, 2022 from https://github.com/sparsehash/sparsehash

[2] [n.d.]. LDBC-SNB. Retrieved October 7, 2022 from https://ldbcouncil.org/benchmarks/snb/

[3] [n.d.]. Neo4j. Retrieved November 15, 2021 from https://neo4j.com/

[4] [n.d.]. TuGraph. Retrieved October 7, 2022 from https://www.tugraph.com/

[5] Stephen Alstrup, Jacob Holm, Kristian de Lichtenberg, and Mikkel Thorup. 1997. Minimizing diameters of dynamic trees. In *Automata, Languages and Programming*. Springer Berlin Heidelberg, Berlin, Heidelberg, 270–280. https://doi.org/10.1007/3-540-63165-8_184

[6] Michael O. Ball, Charles J. Colbourn, and J. Scott Provan. 1995. Chapter 11 Network reliability. In *Network Models*. Handbooks in Operations Research and Management Science, Vol. 7. Elsevier, 673–762. https://doi.org/10.1016/S0927-0507(05)80128-8

[7] Oded Berman and Gabriel Y Handler. 1987. Optimal minimax path of a single service unit on a network to nonservice destinations. *Transportation Science* 21, 2 (1987), 115–122. https://doi.org/10.1287/trsc.21.2.115

[8] Maciej Besta, Marc Fischer, Vasiliki Kalavri, Michael Kapralov, and Torsten Hoefler. 2021. Practice of Streaming Processing of Dynamic Graphs: Concepts, Models, and Systems. *IEEE Transactions on Parallel and Distributed Systems* (2021), 1–1. https://doi.org/10.1109/TPDS.2021.3131677

[9] Ramaprasad Bhar and Shigeyuki Hamori. 2004. *Hidden Markov models: applications to financial economics*. Vol. 40. Springer Science & Business Media. https://doi.org/10.1007/b109046

[10] Paolo Boldi, Andrea Marino, Massimo Santini, and Sebastiano Vigna. 2018. BUbiNG: Massive Crawling for the Masses. *ACM Trans. Web* 12, 2, Article 12 (jun 2018), 26 pages. https://doi.org/10.1145/3160017

[11] Paolo Boldi, Marco Rosa, Massimo Santini, and Sebastiano Vigna. 2011. Layered Label Propagation: A Multiresolution Coordinate-Free Ordering for Compressing Social Networks. In *Proceedings of the 20th International Conference on World Wide Web* (Hyderabad, India) *(WWW '11)*. Association for Computing Machinery, New York, NY, USA, 587–596. https://doi.org/10.1145/1963405.1963488

[12] Paolo Boldi, Massimo Santini, and Sebastiano Vigna. 2008. A Large Time-Aware Web Graph. *SIGIR Forum* 42, 2 (nov 2008), 33–38. https://doi.org/10.1145/1480506.1480511

[13] P. Boldi and S. Vigna. 2004. The Webgraph Framework I: Compression Techniques. In *Proceedings of the 13th International Conference on World Wide Web* (New York, NY, USA) *(WWW '04)*. Association for Computing Machinery, New York, NY, USA, 595–602. https://doi.org/10.1145/988672.988752

[14] Horst Bunke and Terry Michael Caelli. 2001. *Hidden Markov models: applications in computer vision*. Vol. 45. World Scientific.

[15] Edith Cohen, Eran Halperin, Haim Kaplan, and Uri Zwick. 2003. Reachability and Distance Queries via 2-Hop Labels. *SIAM J. Comput.* 32, 5 (2003), 1338–1355. https://doi.org/10.1137/S0097539702403098

[16] Laxman Dhulipala, Changwan Hong, and Julian Shun. 2020. ConnectIt: A Framework for Static and Incremental Parallel Graph Connectivity Algorithms. arXiv:2008.03909 Retrieved August 14, 2020 from https://arxiv.org/abs/2008.03909

[17] Harish Doraiswamy, Huy T. Vo, Cláudio T. Silva, and Juliana Freire. 2016. A GPU-based index to support interactive spatio-temporal queries over historical data. In *2016 IEEE 32nd International Conference on Data Engineering (ICDE)*. 1086–1097. https://doi.org/10.1109/ICDE.2016.7498315

[18] Guanyu Feng, Zixuan Ma, Daixuan Li, Shengqi Chen, Xiaowei Zhu, Wentao Han, and Wenguang Chen. 2021. RisGraph: A Real-Time Streaming System for Evolving Graphs to Support Sub-Millisecond Per-Update Analysis at Millions Ops/s. In *Proceedings of the 2021 International Conference on Management of Data* (Virtual Event, China) *(SIGMOD '21)*. Association for Computing Machinery, New York, NY, USA, 513–527. https://doi.org/10.1145/3448016.3457263

[19] Greg N. Frederickson. 1983. Data Structures for On-Line Updating of Minimum Spanning Trees. In *Proceedings of the Fifteenth Annual ACM Symposium on Theory of Computing (STOC '83)*. Association for Computing Machinery, New York, NY, USA, 252–257. https://doi.org/10.1145/800061.808754

[20] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. 2012. PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation* (Hollywood, CA, USA) *(OSDI'12)*. USENIX Association, USA, 17–30.

[21] Sudipto Guha and Andrew McGregor. 2012. Graph Synopses, Sketches, and Streams: A Survey. *Proc. VLDB Endow.* 5, 12 (Aug. 2012), 2030–2031. https://doi.org/10.14778/2367502.2367570

[22] Q. Guo, F. Zhuang, C. Qin, H. Zhu, X. Xie, H. Xiong, and Q. He. 5555. A Survey on Knowledge Graph-Based Recommender Systems. *IEEE Transactions on Knowledge and Data Engineering* 01 (oct 5555), 1–1. https://doi.org/10.1109/TKDE.2020.3028705

[23] Monika R. Henzinger and Valerie King. 1999. Randomized Fully Dynamic Graph Algorithms with Polylogarithmic Time per Operation. *J. ACM* 46, 4 (jul 1999), 502–516. https://doi.org/10.1145/320211.320215

[24] Jacob Holm, Kristian de Lichtenberg, and Mikkel Thorup. 2001. Poly-Logarithmic Deterministic Fully-Dynamic Algorithms for Connectivity, Minimum Spanning Tree, 2-Edge, and Biconnectivity. *J. ACM* 48, 4 (jul 2001), 723–760. https://doi.org/10.1145/502090.502095

[25] Xiaolin Jiang, Chengshuo Xu, and Rajiv Gupta. 2021. VRGQ: Evaluating a Stream of Iterative Graph Queries via Value Reuse. *SIGOPS Oper. Syst. Rev.* 55, 1 (jun 2021), 11–20. https://doi.org/10.1145/3469379.3469382

[26] Xiaolin Jiang, Chengshuo Xu, Xizhe Yin, Zhijia Zhao, and Rajiv Gupta. 2021. Tripoline: Generalized Incremental Graph Processing via Graph Triangle Inequality. In *Proceedings of the Sixteenth European Conference on Computer Systems* (Online Event, United Kingdom) *(EuroSys '21)*. Association for Computing Machinery, New York, NY, USA, 17–32. https://doi.org/10.1145/3447786.3456226

[27] Ruoming Jin, Ning Ruan, Bo You, and Haixun Wang. 2013. Hub-Accelerator: Fast and Exact Shortest Path Computation in Large Social Networks. arXiv:1305.0507 Retrieved August 13, 2018 from http://arxiv.org/abs/1305.0507

[28] Kevin Joseph and Hui Jiang. 2019. Content Based News Recommendation via Shortest Entity Distance over Knowledge Graphs. In *Companion Proceedings of The 2019 World Wide Web Conference* (San Francisco, USA) *(WWW '19)*. Association for Computing Machinery, New York, NY, USA, 690–699. https://doi.org/10.1145/3308560.3317703

[29] Niranjan Kamat, Prasanth Jayachandran, Karthik Tunga, and Arnab Nandi. 2014. Distributed and interactive cube exploration. In *2014 IEEE 30th International Conference on Data Engineering*. 472–483. https://doi.org/10.1109/ICDE.2014.6816674

[30] Mohsen Koohi Esfahani, Peter Kilpatrick, and Hans Vandierendonck. 2021. Thrifty Label Propagation: Fast Connected Components for Skewed-Degree Graphs. In *2021 IEEE International Conference on Cluster Computing (CLUSTER)*. 226–237. https://doi.org/10.1109/Cluster48925.2021.00042

[31] Pradeep Kumar and H. Howie Huang. 2020. GraphOne: A Data Store for Real-Time Analytics on Evolving Graphs. *ACM Trans. Storage* 15, 4, Article 29 (jan 2020), 40 pages. https://doi.org/10.1145/3364180

[32] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. 2010. What is Twitter, a Social Network or a News Media?. In *Proceedings of the 19th International Conference on World Wide Web* (Raleigh, North Carolina, USA) *(WWW '10)*. Association for Computing Machinery, New York, NY, USA, 591–600. https://doi.org/10.1145/1772690.1772751

[33] Meng-Chieh Lee, Yue Zhao, Aluna Wang, Pierre Jinghong Liang, Leman Akoglu, Vincent S. Tseng, and Christos Faloutsos. 2020. AutoAudit: Mining Accounting and Time-Evolving Graphs. In *2020 IEEE International Conference on Big Data (Big Data)*. 950–956. https://doi.org/10.1109/BigData50022.2020.9378346

[34] Jüri Lember, Dario Gasbarra, Alexey Koloydenko, and Kristi Kuljus. 2019. Estimation of Viterbi path in Bayesian hidden Markov models. *Metron* 77, 2 (2019), 137–169. https://doi.org/10.1007/s40300-019-00152-7

[35] Jun Ma, Danqing Zhang, Yun Wang, Yan Zhang, and Alexey Pozdnoukhov. 2018. GraphRAD: a graph-based risky account detection system. In *Proceedings of ACM SIGKDD conference, London, UK*, Vol. 9.

[36] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. 2010. Pregel: A System for Large-Scale Graph Processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data* (Indianapolis, Indiana, USA) *(SIGMOD '10)*. Association for Computing Machinery, New York, NY, USA, 135–146. https://doi.org/10.1145/1807167.1807184

[37] Mugilan Mariappan and Keval Vora. 2019. GraphBolt: Dependency-Driven Synchronous Processing of Streaming Graphs. In *Proceedings of the Fourteenth EuroSys Conference 2019* (Dresden, Germany) *(EuroSys '19)*. Association for Computing Machinery, New York, NY, USA, Article 25, 16 pages. https://doi.org/10.1145/3302424.3303974

[38] Andrew McGregor. 2014. Graph Stream Algorithms: A Survey. *SIGMOD Rec.* 43, 1 (May 2014), 9–20. https://doi.org/10.1145/2627692.2627694

[39] Frank McSherry, Derek Gordon Murray, Rebecca Isaacs, and Michael Isard. 2013. Differential Dataflow. In *Sixth Biennial Conference on Innovative Data Systems Research, CIDR 2013, Asilomar, CA, USA, January 6-9, 2013, Online Proceedings*. www.cidrdb.org.

[40] J. Picone. 1990. Continuous speech recognition using hidden Markov models. *IEEE ASSP Magazine* 7, 3 (1990), 26–41. https://doi.org/10.1109/53.54527

[41] Maurice Pollack. 1960. Letter to the Editor—The Maximum Capacity Through a Network. *Operations Research* 8, 5 (1960), 733–736. https://doi.org/10.1287/opre.8.5.733

[42] Liam Roditty and Uri Zwick. 2008. Improved Dynamic Reachability Algorithms for Directed Graphs. *SIAM J. Comput.* 37, 5 (2008), 1455–1471. https://doi.org/10.1137/060650271

[43] Amitabha Roy, Ivo Mihailovic, and Willy Zwaenepoel. 2013. X-Stream: Edge-Centric Graph Processing Using Streaming Partitions. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (Farminton, Pennsylvania) *(SOSP '13)*. Association for Computing Machinery, New York, NY, USA, 472–488. https://doi.org/10.1145/2517349.2522740

[44] Satu Elisa Schaeffer. 2007. Graph clustering. *Computer Science Review* 1, 1 (2007), 27–64. https://doi.org/10.1016/j.cosrev.2007.05.001

[45] N. Shacham. 1992. Multicast routing of hierarchical data. In *[Conference Record] SUPERCOMM/ICC '92 Discovering a New World of Communications*. 1217–1221 vol.3. https://doi.org/10.1109/ICC.1992.268047

[46] Xiaogang Shi, Bin Cui, Yingxia Shao, and Yunhai Tong. 2016. Tornado: A System For Real-Time Iterative Analysis Over Evolving Data. In *Proceedings of the 2016 International Conference on Management of Data* (San Francisco, California, USA) *(SIGMOD '16)*. Association for Computing Machinery, New York, NY, USA, 417–430. https://doi.org/10.1145/2882903.2882950

[47] Julian Shun and Guy E. Blelloch. 2013. Ligra: A Lightweight Graph Processing Framework for Shared Memory. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Shenzhen, China) *(PPoPP '13)*. Association for Computing Machinery, New York, NY, USA, 135–146. https://doi.org/10.1145/2442516.2442530

[48] Yogesh Simmhan, Alok Kumbhare, Charith Wickramaarachchi, Soonil Nagarkar, Santosh Ravi, Cauligi Raghavendra, and Viktor Prasanna. 2014. Goffish: A sub-graph centric framework for large-scale graph analytics. In *European Conference on Parallel Processing*. Springer, 451–462. https://doi.org/10.1007/978-3-319-09873-9_38

[49] Daniel D. Sleator and Robert Endre Tarjan. 1981. A Data Structure for Dynamic Trees. In *Proceedings of the Thirteenth Annual ACM Symposium on Theory of Computing* (Milwaukee, Wisconsin, USA) *(STOC '81)*. Association for Computing Machinery, New York, NY, USA, 114–122. https://doi.org/10.1145/800076.802464

[50] George M. Slota, Sivasankaran Rajamanickam, and Kamesh Madduri. 2014. BFS and Coloring-Based Parallel Algorithms for Strongly Connected Components and Related Problems. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium*. 550–559. https://doi.org/10.1109/IPDPS.2014.64

[51] Xiaobing Sun, Wenjie Feng, Shenghua Liu, Yuyang Xie, Siddharth Bhatia, Bryan Hooi, Wenhan Wang, and Xueqi Cheng. 2022. MonLAD: Money Laundering Agents Detection in Transaction Streams. In *Proceedings of the Fifteenth ACM International Conference on Web Search and Data Mining* (Virtual Event, AZ, USA) *(WSDM '22)*. Association for Computing Machinery, New York, NY, USA, 976–986. https://doi.org/10.1145/3488560.3498418

[52] Robert Tarjan. 1972. Depth-First Search and Linear Graph Algorithms. *SIAM J. Comput.* 1, 2 (1972), 146–160. https://doi.org/10.1137/0201010

[53] Mikkel Thorup. 2000. Near-Optimal Fully-Dynamic Graph Connectivity. In *Proceedings of the Thirty-Second Annual ACM Symposium on Theory of Computing* (Portland, Oregon, USA) *(STOC '00)*. Association for Computing Machinery, New York, NY, USA, 343–350. https://doi.org/10.1145/335305.335345

[54] Yuanyuan Tian, Andrey Balmin, Severin Andreas Corsten, Shirish Tatikonda, and John McPherson. 2013. From "Think like a Vertex" to "Think like a Graph". *Proc. VLDB Endow.* 7, 3 (nov 2013), 193–204. https://doi.org/10.14778/2732232.2732238

[55] A. Viterbi. 1967. Error bounds for convolutional codes and an asymptotically optimum decoding algorithm. *IEEE Transactions on Information Theory* 13, 2 (1967), 260–269. https://doi.org/10.1109/TIT.1967.1054010

[56] Keval Vora, Rajiv Gupta, and Guoqing Xu. 2017. KickStarter: Fast and Accurate Computations on Streaming Graphs via Trimmed Approximations. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems* (Xi'an, China) *(ASPLOS '17)*. Association for Computing Machinery, New York, NY, USA, 237–251. https://doi.org/10.1145/3037697.3037748

[57] Henan Wang, Guoliang Li, Huiqi Hu, Shuo Chen, Bingwen Shen, Hao Wu, Wen-Syan Li, and Kian-Lee Tan. 2014. R3: A Real-Time Route Recommendation System. *Proc. VLDB Endow.* 7, 13 (aug 2014), 1549–1552. https://doi.org/10.14778/2733004.2733027

[58] David P Williamson. 2019. *Network flow algorithms*. Cambridge University Press.

[59] Chengshuo Xu, Keval Vora, and Rajiv Gupta. 2019. PnP: Pruning and Prediction for Point-To-Point Iterative Graph Analytics. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems* (Providence, RI, USA) *(ASPLOS '19)*. Association for Computing Machinery, New York, NY, USA, 587–600. https://doi.org/10.1145/3297858.3304012

[60] Da Yan, James Cheng, Yi Lu, and Wilfred Ng. 2014. Blogel: A Block-Centric Framework for Distributed Computation on Real-World Graphs. *Proc. VLDB Endow.* 7, 14 (oct 2014), 1981–1992. https://doi.org/10.14778/2733085.2733103

[61] Fangjin Yang, Eric Tschetter, Xavier Léauté, Nelson Ray, Gian Merlino, and Deep Ganguli. 2014. Druid: A Real-Time Analytical Data Store. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data* (Snowbird, Utah, USA) *(SIGMOD '14)*. Association for Computing Machinery, New York, NY, USA, 157–168. https://doi.org/10.1145/2588555.2595631

[62] Jaewon Yang and Jure Leskovec. 2012. Defining and Evaluating Network Communities Based on Ground-Truth. In *Proceedings of the ACM SIGKDD Workshop on Mining Data Semantics* (Beijing, China) *(MDS '12)*. Association for Computing Machinery, New York, NY, USA, Article 3, 8 pages. https://doi.org/10.1145/2350190.2350193

[63] Byung-Jun Yoon. 2009. Hidden Markov Models and their Applications in Biological Sequence Analysis. *Current genomics* 10, 6 (September 2009), 402—415. https://doi.org/10.2174/138920209789177575

[64] Chaoqun Zhan, Maomeng Su, Chuangxian Wei, Xiaoqiang Peng, Liang Lin, Sheng Wang, Zhe Chen, Feifei Li, Yue Pan, Fang Zheng, and Chengliang Chai. 2019. AnalyticDB: Real-Time OLAP Database System at Alibaba Cloud. *Proc. VLDB Endow.* 12, 12 (aug 2019), 2059–2070. https://doi.org/10.14778/3352063.3352124

[65] Qizhen Zhang, Da Yan, and James Cheng. 2016. Quegel: A General-Purpose System for Querying Big Graphs. In *Proceedings of the 2016 International Conference on Management of Data* (San Francisco, California, USA) *(SIGMOD '16)*. Association for Computing Machinery, New York, NY, USA, 2189–2192. https://doi.org/10.1145/2882903.2899398

[66] Andy Diwen Zhu, Wenqing Lin, Sibo Wang, and Xiaokui Xiao. 2014. Reachability Queries on Large Dynamic Graphs: A Total Order Approach. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data* (Snowbird, Utah, USA) *(SIGMOD '14)*. Association for Computing Machinery, New York, NY, USA, 1323–1334. https://doi.org/10.1145/2588555.2612181

[67] Xiaowei Zhu, Wenguang Chen, Weimin Zheng, and Xiaosong Ma. 2016. Gemini: A Computation-Centric Distributed Graph Processing System. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation* (Savannah, GA, USA) *(OSDI'16)*. USENIX Association, USA, 301–316.

[68] Xiaowei Zhu, Wentao Han, and Wenguang Chen. 2015. GridGraph: Large-Scale Graph Processing on a Single Machine Using 2-Level Hierarchical Partitioning. In *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference* (Santa Clara, CA) *(USENIX ATC '15)*. USENIX Association, USA, 375–386.

Received 2022-07-07; accepted 2022-09-22