UNIVERSITY OF CALIFORNIA
RIVERSIDE

On the Scalability and Efficiency of Graph Processing Systems

A Dissertation submitted in partial satisfaction
of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

by

Xizhe Yin

June 2024

Dissertation Committee:

Dr. Rajiv Gupta, Co-Chairperson
Dr. Zhijia Zhao, Co-Chairperson
Dr. Nael Abu-Ghazaleh
Dr. Manu Sridharan

The Dissertation of Xizhe Yin is approved:

Committee Co-Chairperson

Committee Co-Chairperson

University of California, Riverside

Acknowledgments

Looking back at my Ph.D. years, I am deeply grateful to my co-advisors, Prof. Rajiv Gupta and Prof. Zhijia Zhao. It has been an incredible honor to work with two exceptional researchers. The idea brainstorming, research discussions, and paper-writing moments were truly memorable. Their sharp minds and passion for research profoundly shaped my Ph.D. studies and my belief in conducting good research. Though their suggestions sometimes initially seemed nonsensical, they were always thought-provoking, both in research and in life.

I sincerely thank my thesis committee members, Prof. Nael Abu-Ghazaleh and Prof. Manu Sridharan, for their constructive feedback, which significantly strengthened my thesis.

To all my lab mates and UCR CSE friends, including Chaitanya Ananda, Mahbod Afarin, Umar Farooq, Chao Gao, Xiaolin Jiang, Lin Jiang, Zexin Li, Abbas Mazloumi, Yuxin Qiu, Junqiao Qiu, Rebin Silva, Xiaofan Sun, Chengshuo Xu, Rui Yang, Jingyao Zhang, and countless others, thank you for the friendship, support, and small talks in the lab.

Special thanks to my friend Zhexing Li for hiking together in the mountains of Southern California during the pandemic.

I thank the UCR graduate division and our CSE department, including Dr. Marek Chrobak, Dr. Emma H. Wilson, and Vanda Yamaguchi, for guiding me through a smooth graduation process.

I am grateful to Prof. Weiming Shen, my mentor during my master's study, for his advice and encouragement to pursue a Ph.D.

Lastly, I thank my parents and parents-in-law for their care and understanding. Most importantly, I thank my wife for her unwavering love and support. Without my family's courage and belief in me, this thesis would not have been possible.

To my parents for all the support.

ABSTRACT OF THE DISSERTATION

On the Scalability and Efficiency of Graph Processing Systems

by

Xizhe Yin

Doctor of Philosophy, Graduate Program in Computer Science
University of California, Riverside, June 2024
Dr. Rajiv Gupta, Co-Chairperson
Dr. Zhijia Zhao, Co-Chairperson

Developing scalable and efficient graph systems to support low-latency streaming analysis and high-throughput concurrent query evaluation poses significant challenges. This thesis proposes solutions to address these challenges.

Existing streaming graph systems require prior knowledge of queries; otherwise, they fall back to expensive full query evaluations. The proposed Tripoline system overcomes this by applying principles similar to triangle inequalities in Euclidean geometry, generalizing them for various vertex-specific graph problems. This allows the reuse of existing query results to accelerate new queries. Tripoline achieves a speedup of up to 41.5× compared to traditional streaming graph engines.

Another limitation of streaming graph analysis is that when changes to the graphs become large, the incremental graph computation starts running slower than the full query evaluation. Moreover, present incremental graph algorithms cannot handle edge weight updates elegantly, resorting to a sub-optimal two-phase process. The proposed IncBoost addresses these scalability issues with algorithmic enhancements and system-level optimiza-

tions. It employs a novel bottom-up dependency tracing technique to identify vertices affected by edge updates without accessing the graph data. It introduces a direct method for handling edge weight updates. IncBoost scales to handle large update batches with sizes of 30% to 50% of the graph and achieves up to a 4.9× speedup over RisGraph.

To enhance the efficiency of a query processing system, concurrent query processing is used for higher throughput. However, efficiency is often hampered by misaligned graph traversals, causing unfavorable last-level cache misses. A runtime system called Glign is developed to address this issue by automatically aligning graph traversals for concurrent queries. Glign features novel optimizations at three levels: (1) the intra-iteration alignment effectively reduces memory footprint for graph traversals; (2) the inter-iteration alignment enlarges the overlapping of vertices for better-shared graph access; (3) a query batch formation strategy predicts query subsets with better affinity for batching. Glign outperforms the state-of-the-art concurrent graph processing systems, achieving speedups of up to 4.7×.

To improve the system throughput of graph-based approximate nearest neighbor search (ANNS), this dissertation explores the design of graph construction and search phases, particularly with temporal information. The proposed graph construction algorithm considers the correlation between input queries and the final answers in the time dimension. A fully parameterized best-first search algorithm is also designed for flexible performance tuning. These techniques improve query throughput by up to 1.9× compared to the state-of-the-art DiskANN while maintaining recall. Additionally, the constructed graph size can be reduced by up to 30% without compromising query quality.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Graph analytics is crucial for extracting insights from large volumes of connected data, such as social networks, web graphs, internet topology, and brain networks. These analyses use iterative algorithms to update the properties of vertices within a graph until reaching a stable solution. Due to the vast size of real-world graphs, which often contain millions of vertices and billions of edges, graph analytics is both data-intensive and compute-intensive, requiring substantial computational resources.

To manage the challenges of scale and computational demand, there has been significant interest in developing efficient graph analytics systems. Systems such as Ligra [106], GraphLab [66], GraphIt [66], PnP [126], PowerGraph [41], GridGraph [142], Gemini [140], and many others have been introduced to handle large graphs effectively. These systems are primarily designed to evaluate a single query on a static graph. The focus is on optimizing the computation of individual graph queries by leveraging memory abstraction and designing highly parallelized iterative algorithms.

In contrast to the above work, the focus of this thesis is on two more complex and demanding scenarios in graph processing involving streaming (changing) graphs and multi-query evaluation systems as described below.

(1) Low-latency evaluation of queries over a streaming graph. The graph dynamically changes with frequent edge insertions and deletions. This scenario is demanding because dynamic changes require the system to promptly update its data structures and recalculate affected query results in real-time. Applications such as network optimization, temporal graphs analyses, fraud detection, and real-time recommendation systems necessitate immediate responses to graph changes, making timely processing critical. Additionally, as the graph grows, the system must scale efficiently to handle increasing volumes of updates, which traditional static graph processing methods cannot achieve.

(2) High-throughput evaluation of multiple queries on static graphs. This scenario is challenging because multiple queries access the graph simultaneously, leading to resource contention and degraded performance. Graph processing systems such as GraphM [139] and Krill [18] have been developed to enable concurrent query evaluation on static graphs. The aim is to compute multiple graph queries at the same time on the same graph. However, the shared graph access among concurrent queries is insufficient, causing frequent last-level cache misses and limiting such systems' throughput.

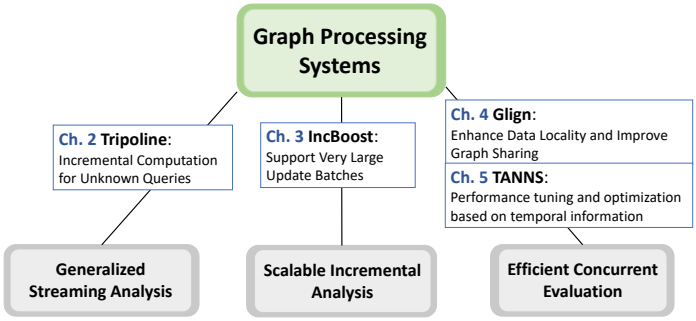


Figure 1.1: Dissertation Overview.

1.1 Dissertation Overview

This dissertation focuses on improving the scalability and efficiency of graph processing systems in these complex scenarios. It proposes system and algorithmic optimizations from three perspectives: (1) Generalizability of streaming graph systems, (2) Scalability of incremental graph computation, and (3) Efficiency of concurrent graph processing.

1.1.1 Generalizability of Streaming Graph Systems

Tripoline: Generalized Streaming Analysis Existing streaming systems are limited by their need for apriori knowledge of the query (i.e., the query has to be installed and evaluated once); otherwise, such systems have to fall back to the expensive full query evaluation that starts from scratch. `Tripoline` sidesteps this limitation by applying principles akin to triangle inequalities in Euclidean geometry. These principles can be generalized for several vertex-specific graph problems and help establish constraints between the evaluation of one graph query and the results of another, thus enabling the reuse of existing query results to accelerate arbitrary new user queries. `Tripoline` has demonstrated a remarkable speedup of up to 41.5× compared to the traditional streaming graph engine `Aspen` [27]. Moreover,

`Tripoline`'s triangle inequality-based optimization has also been adopted to `Differential Dataflow` [84], a state-of-the-art general-purpose streaming framework, and provided extra speedups over its existing design.

1.1.2 Scalability of Incremental Graph Computation

IncBoost: Scalable Incremental Analysis Efficient support of massive graph updates is demanding, as it is commonly seen in evolving graphs and dynamic communication network analyses where a relatively large region of the graph can be updated simultaneously. However, existing incremental graph processing systems do not scale well when changes to the graphs become large – incremental computation starts running slower than the full (non-incremental) query evaluation. Moreover, present incremental graph algorithms cannot handle edge weight updates elegantly, resorting to a sub-optimal two-phase process instead of a direct method.

`IncBoost` addressed the above issues by proposing both algorithmic enhancements and system-level optimizations. `IncBoost` employs a novel *bottom-up dependency tracing* technique that identifies vertices affected by edge updates, obviating the need to access the graph data. Additionally, a new algorithmic approach is proposed for directly handling edge weight updates. This approach leverages a local monotonicity test to discern whether to treat weight increments or decrements differently, thus circumventing the two-phase process and eliminating unnecessary computation. Experimental evaluations demonstrate that `IncBoost` successfully scales to handle very large update batches with sizes of 30% to 60% of the graph and achieves up to 4.9× speedup over `RisGraph` [34], a state-of-the-art streaming graph processing system.

1.1.3 Efficiency of Concurrent Graph Processing

Glign: Concurrent Multi-query Evaluation In the multi-query evaluation scenario, the efficiency of concurrent evaluations is often hampered by the misalignment in graph traversals, resulting in unfavorable last-level cache misses. To address this issue, a runtime system called `Glign` is developed, which automatically aligns graph traversals for concurrent queries. `Glign` is equipped with novel optimizations at three computation levels: (1) Intra-iteration: A *query-oblivious frontier* design is introduced to deliberately ignore the frontier differences across all queries, effectively reducing memory footprint. (2) Inter-iteration: By identifying the "heavy iterations" of graph queries, a *delayed-start* strategy is used by `Glign` to align them, which effectively enlarges the overlapping of vertices for better-shared graph access. (3) Query batch formation: By leveraging heuristics for heavy iteration estimation, `Glign` predicts which query subsets will most likely exhibit better affinity when batched together. Combining these techniques, `Glign` consistently outperforms existing state-of-the-art systems, achieving speedups of up to 4.7×.

TANNS: High-throughput Graph-based Approximate Nearest Neighbor Search Approximate nearest-neighbor search (ANNS) has become a key component in modern deep learning applications, serving efficient similarity search over high-dimensional vector data. Recently, graph-based ANNS solutions have demonstrated high throughput by concurrently evaluating ANNS queries. These systems construct query graphs from vector datasets and evaluate user queries using best-first search algorithms. However, the design space for both the graph construction and search phases has not been fully explored. Specifically, the construction phase often overlooks potential correlations between input queries and final

answers when temporal information exists. Additionally, in the best-first search, the design space has not been fully exploited for potential performance improvements. To address these limitations, we introduce the `TANNS` system. `TANNS` constructs a proximity graph that incorporates temporal information, revealing correlations between queries and results. It also features a fully parameterized search algorithm, allowing extensive performance tuning. `TANNS` can achieve up to a 1.9× speedup in query throughput compared to the state-of-the-art DiskANN implementation while maintaining the same recall levels. Moreover, the size of the constructed graph can be reduced by up to 30% without compromising query quality.

1.2 Dissertation Organization

The rest of the dissertation is organized as follows. Chapter 2 presents Tripoline, the system for generalized streaming graph analysis that is able to incrementally evaluate arbitrary user queries via triangle inequalities. Chapter 3 introduces IncBoost, the scalable incremental graph query processing engine that can support very large update batches. Chapter 4 describes the Glign system for concurrent graph query evaluations and introduces several alignment techniques. Chapter 5 presents techniques for enhancing query throughput of approximate nearest neighbor search (ANNS). It introduces a fully parameterized search algorithm for performance tuning and an adaptive graph construction algorithm that leverages temporal information to improve query evaluation. Chapter 6 provides a detailed discussion of related works in graph processing. Finally, Chapter 7 concludes this dissertation and discusses potential future directions in this field.

Chapter 2

Generalizing Incremental Graph Processing via Triangle Inequality

2.1 Introduction

In many real-world application scenarios, a stream of updates is continuously applied to the graph, often in batches for better efficiency, known as the *streaming graph* scenario. Taking social network graphs as an example, new data that carry rich connection information, such as tweets, are continuously generated, causing updates to the existing graph. Similar scenarios also occur in the mining of online shopping activities, where new purchases may generate new connections between customers (e.g., those who bought the same product) and between products (e.g., those that are bought together). In such scenarios, new edges and vertices are continuously added to the graph.

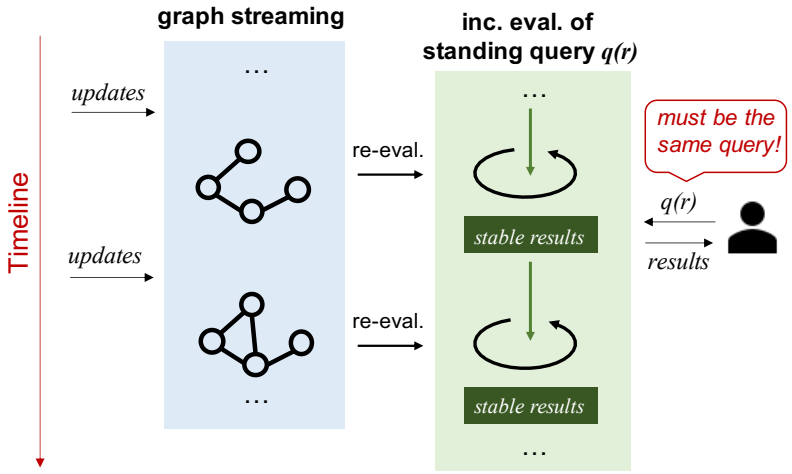


Figure 2.1: Streaming Graph Processing with Incremental Query Evaluation and the Limitation of Existing Solutions.

In the streaming graph scenario, to reduce the latency of query evaluation, it is critical to evaluate the expensive iterative graph queries incrementally upon graph updates. ***State of the Art.*** Several streaming graph systems have been proposed recently with support for incremental evaluation of iterative graph queries. Examples include Kineograph [22], Tornado [105], Naiad [84], KickStarter [117], Graphbolt [75], and so on. The basic idea of these systems is to reevaluate the query each time the graph gets updated, as illustrated in Figure 2.1. Instead of reevaluating the query from scratch (i.e., a full reevaluation), they start the reevaluation directly on the results of the previous evaluation, performing just enough calculations based on the newly inserted edges and vertices until the results stabilize again. As the new edges and vertices in each update batch usually represent just a tiny fraction of the existing graph, the incremental evaluation usually converges much faster than a full reevaluation.

However, the above approach requires a priori knowledge of the query to be incrementally evaluated, referred to as the *standing query*. This may not be an issue for queries without source vertex specification, such as PageRank; but creates a fundamental challenge for vertex-specific queries, like breath-first search (BFS), which target specific vertices of interest (e.g., $BFS(v_5)$). The source vertex of interest may be unknown until the query arrives. Thus, only the pre-selected standing queries (e.g., $BFS(v_5)$) can be incrementally evaluated; queries with other source vertices have to undergo an expensive full evaluation once they are received. This limitation significantly compromises the generality of the existing incremental streaming graph systems.

In this chapter, we propose a principled way to generalize incremental graph processing so that vertex-specific queries without their prior knowledge may also benefit from incremental processing. The key to our solution is a concept called *graph triangle inequality*. Similar to the classic triangle inequality in Euclidean space, triangle inequalities with generalized distance and comparison operators may also be derived for vertex-specific queries in the graph space. Based on them, we can establish rigorous constraints between a user query (whose source vertex can be any vertex in the graph) and the pre-selected standing query, thus enabling reusing the results of the latter to accelerate the evaluation of the former. We refer to this technique as *graph triangle inequality-based incremental processing*. For correctness, the graph query implementation is assumed to be monotonic and safe under asynchrony (more details are given in Section 2.4.3).

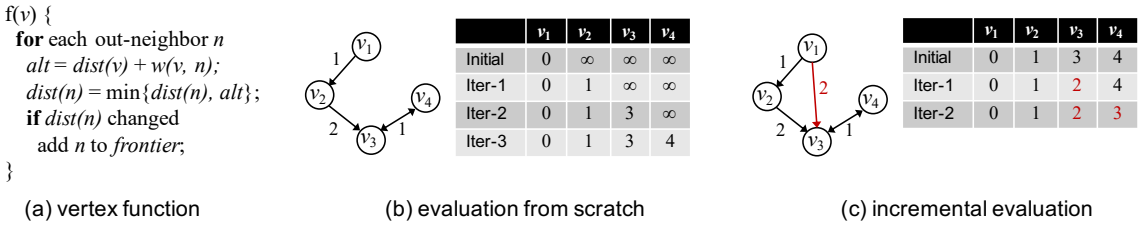
To demonstrate the effectiveness of the above generalized incremental graph processing, we developed a streaming graph system on top of a state-of-the-art streaming

graph engine called Aspen [27], which offers a compact yet efficient data structure for high-throughput graph updates. We name the new system `Tripoline` to encapsulate its essence: use the graph triangle inequality as a "trampoline" to fast-forward the evaluation of queries different from the standing one. By continuously and incrementally evaluating a small set of pre-selected standing queries upon graph updates, Tripoline can incrementally evaluate previously unseen queries based on the results of the standing ones and the triangle inequalities.

We evaluated Tripoline using eight types of vertex-specific graph queries and four real-world large graphs (more details in Section 2.6). The results show that the performance benefits of Tripoline vary depending on the vertex-specific problems (and their graph triangle inequalities). Overall, we observed 8.83-30.52× speedups on four types of the evaluated graph queries, 1.18-1.89× speedups on three types of graph queries, and limited speedup (1.08×) on one type of graph queries.

In summary, this work makes the following contributions:

- It proposes to leverage *graph triangle inequality* in the scheme of incremental graph processing, which, to our knowledge, for the first time enables generalized incremental evaluation of vertex-specific queries;
- It introduces the triangle abstraction based on a pair of generalized distance and comparison operators and establishes the specific graph triangle inequalities for a spectrum of vertex-specific graph queries.



	v_1	v_2	v_3	v_4
Initial	0	∞	∞	∞
Iter-1	0	1	∞	∞
Iter-2	0	1	3	∞
Iter-3	0	1	3	4

	v_1	v_2	v_3	v_4
Initial	0	1	3	4
Iter-1	0	1	2	4
Iter-2	0	1	2	3

Figure 2.2: Vertex-Centric Programming and Incremental Query Evaluation (SSSP(v_1) as the Example).

- Finally, it develops `Tripoline`, a streaming graph system that supports *generalized* incremental evaluation for vertex-specific graph queries. The system has shown substantial performance improvements on multiple vertex-specific iterative graph queries.

2.2 Background

Vertex-Centric Programming. A commonly used model for programming graph applications is the *vertex-centric programming* model. It was first introduced by Pregel [70] based on the bulk synchronous parallel (BSP) model [114]. The model requires defining a *vertex function* that computes some properties of the vertices (a.k.a. vertex values). The graph computations start from some default initial vertex values, then apply the vertex function across all (or a subset of) vertices of the graph, iteration by iteration until the vertex values become stable (or some threshold is reached).

Figure 2.2-(a) illustrates a vertex-centric implementation of the single-source shortest path (SSSP) query, which finds the shortest distances from a source vertex to all other vertices in the graph. The vertex function $f(v)$ computes an alternative distance based on the current value of the vertex and then compares it with the value of each of its outgo-

ing neighbors. If the new value is less than the existing one, the neighbor's value will be updated. This is known as the *push* model [1]. As shown in Figure 2.2-(b), initially, all the vertex values are set to ∞, except the source vertex whose value is set to zero. Then, the vertex function $f(v)$ is evaluated across all the vertices over iterations until all the vertex values stop changing. To improve the efficiency, an active vertex list (a.k.a. *frontier*) can be maintained, which only consists of vertices whose values were changed in the last iteration, so only vertices in the frontier need to be evaluated in each iteration. In the case of SSSP, the frontier is initialized with only the source vertex and will become empty once all the vertex values are converged. Hereinafter, this work assumes a frontier-based implementation of the push model.

Incremental Graph Processing. As mentioned earlier, in the common streaming graph processing scenario [22, 105, 84, 117, 75], the graph is continuously updated with new edges and vertices, usually in batches for better efficiency. A recent work [117] also discussed the scenario with edge deletions, which is orthogonal to the focus of this work. Like many prior works [22, 105], we assume the growing graph scenarios in this work. After each batch of insertions, the standing graph query needs to be reevaluated to reflect the latest results. Instead of re-evaluating the query on the updated graph from scratch (i.e., viewing it as a completely new graph), existing streaming graph systems adopt an incremental graph query evaluation strategy to improve efficiency.

The design of the incremental query evaluation naturally matches the BSP model in the aforementioned vertex-centric programming. Consider the example in Figure 2.2-(c).

[1] The vertex function can also be implemented using a *pull* model, which updates the value of the vertex based on its in-neighbors' values.

After a new edge (v_1, v_3) is inserted, the reevaluation directly starts from the converged vertex values of the prior evaluation rather than initializing the vertex values with ∞. To ensure correctness, the source vertices of the newly inserted edges (i.e., v_1) need to be inserted into the frontier, which resumes the iterations until a new stabilization is reached. As each insertion batch is typically a tiny fraction of the existing graph, the reevaluation tends to terminate much faster than a full reevaluation [22, 117].

Despite the promise of incremental graph processing, there exists a fundamental limitation in the existing design – it assumes prior knowledge of the query. The assumption holds for queries that do not depend on a specific vertex, such as PageRank, but imposes a major obstacle for vertex-specific queries, like BFS and SSSP. For the latter, the incremental evaluation would work only for the pre-selected standing query, like SSSP(v_1); for queries originating at other vertices in the graph, an expensive full evaluation is required.

In fact, vertex-specific graph queries appear more common than "whole-graph queries" in real-world applications. First, vertex-specific queries are concerned with the interests or capture the perspective of a specific vertex, which are common in online shopping and social networks, such as generating recommendations for individual customer [130] and finding the overlap of friends of two specific users [23]. Second, as subproblems, vertex-specific graph queries often require less time and space than their counterpart whole-graph queries (e.g., SSSP vs all-pair shortest path). This is especially critical in the streaming graph scenario, where the query evaluation needs to keep up with the graph updates.

In summary, incremental graph processing is essential to streaming graph systems. However, its existing design suffers from a fundamental applicability challenge for an im-

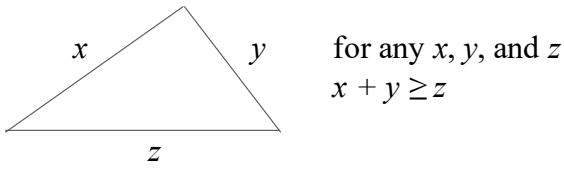


Figure 2.3: Triangle Inequality in Euclidean Geometry.

portant group of graph queries – vertex-specific queries. Before presenting our solution, we first introduce the key principle behind it – graph triangle inequalities.

2.3 Graph Triangle Inequality

In this section, we first provide an intuition of *graph triangle inequality*, then formally define the principle and present several graph triangle inequalities examples.

2.3.1 Intuition

Triangle inequality [46], as illustrated in Figure 2.3, is a basic principle in Euclidean geometry. It states the fact that for any given triangle Δxyz, the sum of the lengths of any two sides must be greater than or equal to the length of the third side. Prior research [30] has shown the possibility of leveraging triangle inequality to accelerate K-means clustering in the Euclidean space. Inspired by this, we wondered if similar principles exist in graph problems and, hence, may be used to optimize streaming graph processing. In fact, for a spectrum of vertex-specific graph problems, similar inequalities can be naturally derived. Next, we first use SSSP as an example to introduce the graph triangle inequality because it calculates distances which are similar to the lengths in the classical triangle inequality, except that the "domain" is a graph rather than the Euclidean space.

SSSP Triangle. It is not hard to find that the vertices in a graph are analogous to the points in the Euclidean space. The *distance* between two points in the Euclidean space is the length of the line segment connecting them. Similarly, the *distance* between two vertices v_1 and v_2 in a weighted graph is the minimum weight of all paths connecting them:

$$dist(v_1, v_2) = min\{w(p) \mid p \text{ is a path from } v_1 \text{ to } v_2\} \tag{2.1}$$

where $w(p)$ is the sum of weights on all the edges in path p. Note that, for undirected graphs, as paths are symmetric, we have $dist(v_1, v_2) = dist(v_2, v_1)$. Based on this analogy, it is not hard to find that a triangle inequality also holds for graph distances, as illustrated in Figure 2.4.

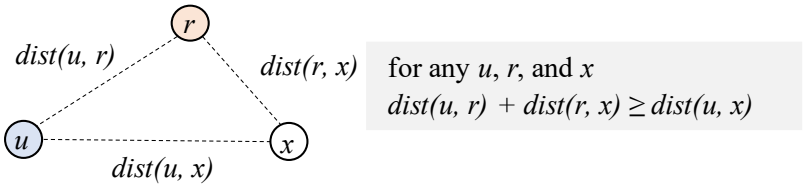


Figure 2.4: Triangle Inequality in SSSP (dashed lines represent the shortest paths between two vertices).

In fact, the above triangle inequality in Figure 2.4 becomes obvious once one realizes that the shortest path from u to r can be concatenated with the shortest path from r to x, and the resulted path is just one of the many paths from u to x, hence must be no shorter than the shortest path from u to x.

Actually, the above graph triangle inequality based on the distances between vertices is well-known in the theoretical graph community [13]. Some prior work [24] has exploited this principle to approximate distances in web-scale large graphs, which shares some of the spirit of this work. However, as we will demonstrate shortly, our work discusses

a broader definition of "distance" that goes beyond the conventional one shown in Equation 2.1. Furthermore, our work exploits the graph triangle inequality in a different context – streaming graph processing, where the accuracy of each query result is always guaranteed – no approximation is allowed.

For brevity, we refer to the above distance-based triangle inequality as *SSSP triangle*. Next, we generalize it by defining a more general definition of "distance" and two abstract operators for addition and comparison, respectively.

2.3.2 Triangle Abstraction

Rather than referring to the distance, we define the graph triangle inequality for a property between two vertices – $property(v_1, v_2)$, where (v_1, v_2) is an ordered pair for directed graphs and an unordered pair for undirected graphs.

Definition 1. Given the property definition between two vertices $property(v_1, v_2)$, the *graph triangle inequality* can be formally defined by the following equation:

$$property(v_1, v_2) \oplus property(v_2, v_3) \succeq property(v_1, v_3) \tag{2.2}$$

where $\oplus$ depicts an abstract addition and $\succeq$ represents an abstract greater than or equal to operator.

To demonstrate the generality of the triangle abstraction, we next present several concrete graph triangle inequalities that are not based on the distance property.

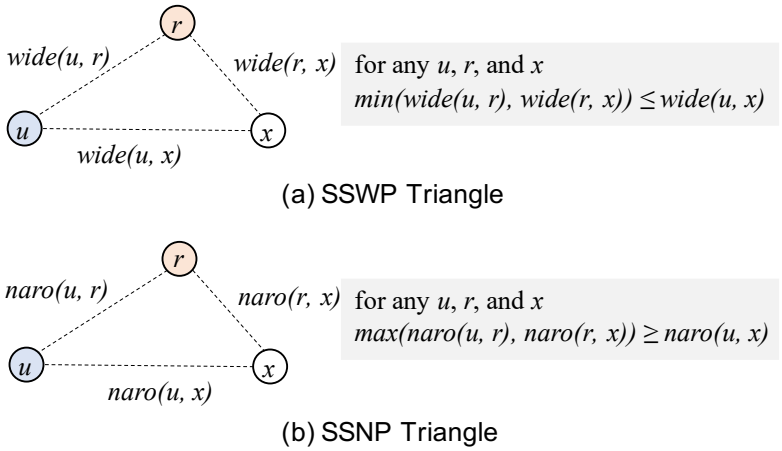


Figure 2.5: Triangle Inequalities in SSWP and SSNP (dashed lines depict the widest/narrowest paths between vertices).

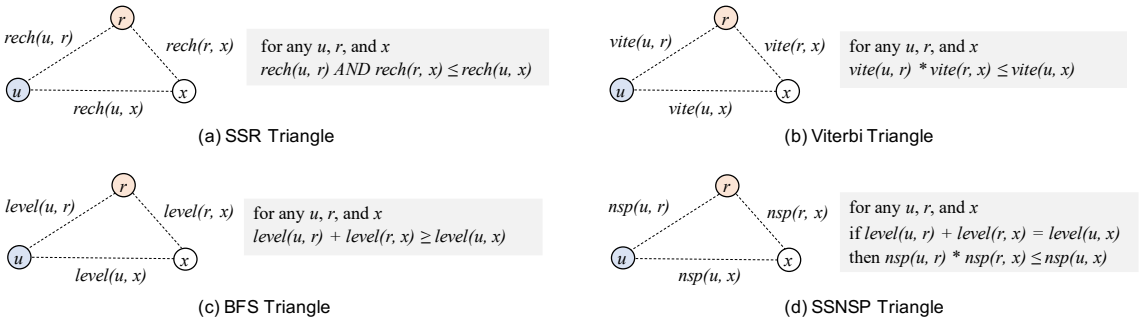


Figure 2.6: Triangle Inequalities in SSR, Viterbi, BFS/Radii, and SSNSP (where a dashed line depicts the connectivity, maximum probability path, BFS-level, and the shortest paths between two vertices, respectively).

SSWP/SSNP Triangle. SSWP and SSNP are abbreviations for single-source widest path and single-source narrowest path, respectively. Both of them play important roles in network routing [124] and transportation planning [25].

Given a source vertex v, SSWP and SSNP compute the widest and narrowest path from v to every other vertex in the graph. The widest path between two vertices is the path whose minimum edge weight is the largest, while the narrowest path between two vertices is the path whose maximum edge weight is the smallest, as defined below:

$$wide(v_1, v_2) = max\{minw(p) \mid \text{ path } p \text{ from } v_1 \text{ to } v_2\} \tag{2.3}$$

$$naro(v_1, v_2) = min\{maxw(p) \mid \text{ path } p \text{ from } v_1 \text{ to } v_2\} \tag{2.4}$$

where $minw(p)$ and $maxw(p)$ represent the minimum and maximum edge weight along path p, respectively.

Based on their definitions, it is not difficult to derive the triangle inequalities for SSWP and SSNP, shown in Figure 2.5. The reasoning behind these inequalities is similar to that of SSSP, except that they are based on different addition $\oplus$ and comparison $\succeq$ operators. For example, the inequality holds for SSWP because the widest paths from u to r and from r to x can be concatenated, and the width of the concatenated path must be no larger than the width of the widest path from u to x as it is just one of the paths from u to x. Similarly, we refer to the triangle inequalities for SSWP and SSNP as *SSWP triangle* and *SSNP triangle*, respectively, for brevity.

Other Triangles. Due to space limitations, we next briefly present the triangle inequalities for the other graph problems that we have considered. These include:

- Single-source reachability (SSR) [47] which finds all the vertices connected to the source vertex. Figure 2.6-(a) shows its triangle inequality based on the reachability property defined in Equation 2.5.

- Viterbi algorithm (Viterbi) [62] which computes the probability along the Viterbi path (a state path that maximizes the conditional probability) from the source vertex. Figure 2.6-(b) shows its triangle inequality based on the *vite* property defined in Equation 2.6, where $w(p)$ depicts the total weights of edges in path p.

- Breath-first search (BFS) [81] which computes the level of each vertex in the BFS tree rooted at the source vertex. Figure 2.6-(c) shows its triangle inequality based on the BFS level property defined in Equation 2.7, where $nEdges(p)$ depicts the number edges in path p.

- Radii estimation (Radii) [106] which estimates the graph radius by running multiple SSSP and selecting the largest distance among their results. As it is based on SSSP, its triangle inequality is just that of SSSP.

- Single-source number of shortest path (SSNSP) [102] which computes not only the BFS levels [2], but also the number of shortest paths from the source vertex to all the other vertices. Figure 2.6-(d) shows its triangle inequality based on both the BFS level property and the number of shortest paths property. The latter is defined in Equation 2.8, where $|\cdot|$ depicts the set size.

$$rech(v_1, v_2) = \begin{cases} 1 & \text{if a path from } v_1 \text{ to } v_2 \text{ exists} \\ 0 & \text{otherwise} \end{cases} \tag{2.5}$$

[2] In this case, SSNSP is for unweighted graphs.

$$vite(v_1, v_2) = max\{1/w(p) \mid \text{path } p \text{ from } v_1 \text{ to } v_2\} \tag{2.6}$$

$$level(v_1, v_2) = min\{nEdges(p) \mid \text{path } p \text{ from } v_1 \text{ to } v_2\} \tag{2.7}$$

$$nsp(v_1, v_2) = |\{\text{the shortest paths from } v_1 \text{ to } v_2\}| \tag{2.8}$$

For brevity, we refer to the above triangle inequalities as *SSR triangle*, *Viterbi triangle*, *BFS triangle*, and *SSNSP triangle*, respectively. Among these triangles, the Viterbi triangle and BFS triangle can be intuitively derived just based on their definitions and the fact that the paths from u to r and from r to x can be concatenated to form one path from u to x. For SSR triangle, the situation is different in that the property of interest (i.e., reachability) is about the existence of any path between two vertices. In this case, a logical AND perfectly fits in the role of the $\oplus$ operator. The last one, the SSNSP triangle is also special in that it requires a predicate (condition) for the triangle inequality to hold. As we will show later, the predicate actually affects the effectiveness of the triangle inequality in the use of incremental query evaluation.

In summary, the graph triangle inequality, as abstracted in Equation 2.2, is generally enough to capture a spectrum of vertex-specific graph problems.

2.4 Generalized Incremental Evaluation

In this section, we show that, based on the graph triangle inequality abstraction, incremental evaluation of queries without a priori knowledge can be achieved in general.

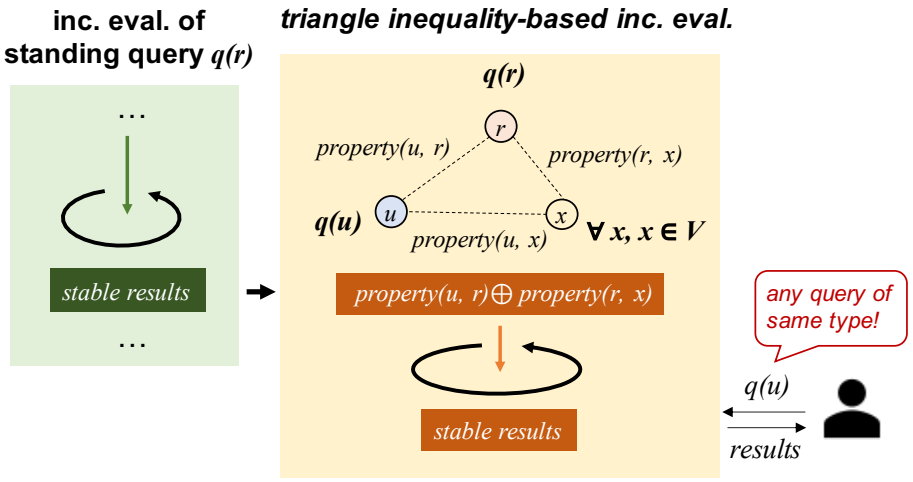


Figure 2.7: Triangle Inequality (Δ)-based Incremental Query Evaluation for an Arbitrary Query of the Same Type (i.e., a query starting from a different source vertex but asking for the same graph property, like SSWP(v_1) and SSWP(v_2)).

2.4.1 Δ-based Incremental Evaluation

The key to our solution is a principled way of "connecting" the evaluation of a vertex-specific query to the results of another query evaluation of the same type (e.g., SSWP) based on their graph triangle inequality.

Execution Model. Figure 2.7 illustrates the basic idea of our solution. Assume $q(r)$ is the pre-selected standing query (the selection will be discussed later), where r is the source vertex. In the programming system, query $q()$ is a user-specified function that implements the (vertex-specific) querying logic while r is the parameter to the function.

First, the standing query $q(r)$ is evaluated continuously and incrementally upon graph updates, like those in the existing incremental graph processing systems [84, 22, 117]. Meanwhile, the system accepts user queries like $q(u)$ which is of the same type as $q(r)$, but its source vertex u could be any vertex in the graph. From the evaluation of $q(r)$, we can obtain the values of $property(r, u)$ and $property(r, x)$. For easier explanation, here we

assume the graph is undirected (directed ones will be discussed later), which means that we can also obtain $property(u, r)$ – the same as $property(r, u)$. In addition to vertices r and u, consider an arbitrary vertex x in the graph different from r and u. Together, r, u, and x form a triangle, just like one of those in Section 2.3. Then, based on the addition operator $\oplus$ in the triangle abstraction (see Equation 2.2), we can compute the following value set:

$$\Delta(u, r) = \{property(u, r) \oplus property(r, x) \mid x \in V\} \tag{2.9}$$

Next, instead of evaluating $q(u)$ from scratch (i.e., using the default initial values) on the current version of the graph, the system starts its evaluation directly from $\Delta(u, r)$, and runs until all the vertex values are converged. Note that, just like full evaluation, the above incremental evaluation also starts from the source vertex u (i.e., the frontier is initialized with u).

In the above process, the system maintains an in-memory state consisting of three parts: (i) the streaming graph, (ii) the evaluation of standing query, and (iii) the evaluation of user query. As detailed later, in our prototype, the streaming graph can be incrementally maintained with a compression tree-based data structure (Aspen [27]), while the results of query evaluation are kept in a *property array* of size $|V|$.

We refer to the above streaming graph execution model as *triangle inequality-based incremental evaluation*, or Δ*-based incremental evaluation* [3] for short.

In the following sections, we will first extend the proposed Δ-based incremental evaluation to directed graphs, then analyze its correctness, benefits and costs, and finally discuss how the standing query can be selected.

[3]Here, Δ reads as triangle inequality, not delta (difference).

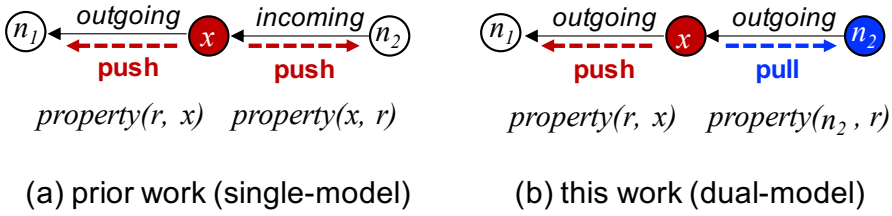


Figure 2.8: Computing $property(r, x)$ and $property(x, r)$ on Directed Graphs: Prior Work [24, 22] vs. Our Solution.

2.4.2 Dual-Model Evaluation for Directed Graphs

In the case of directed graphs, $property(u, r)$ may not be the same as $property(r, u)$, thus not available in the evaluation results of the standing query $q(r)$. In this case, we turn to the reversed graph problem, denoted as $q^{-1}(r)$, which computes the properties from all vertices to r.

$$\text{Results of } q^{-1}(r) = \{property(x, r) \mid x \in V\} \tag{2.10}$$

Taking SSSP as an example, $\text{SSSP}^{-1}(v)$ is to find the shortest distance from every vertex in the graph to v.

A straightforward way to evaluate $q^{-1}(r)$, as elaborated in prior work [24, 22], is to update values of the in-neighbors, rather than the out-neighbors as in the evaluation of $q(r)$. Figure 2.8-(a) illustrates this idea. However, with this solution, the evaluations of queries $q(r)$ and $q^{-1}(r)$ need to access both outgoing and incoming edges efficiently (i.e., indices for both outgoing and incoming neighbors). This not only doubles the memory consumption of the edge data (two-way indices rather than one-way), but also increases the cost of streaming graph maintenance – need to keep both incoming and outgoing edge representations up to date.

To address the above issue, we propose a novel *dual-model query evaluation* solution for directed graphs. The solution enables us to evaluate both queries $q(r)$ and $q^{-1}(r)$ on a graph with only one-way edge representation (outgoing or incoming edge-based). The key to this solution is the fact that both the push/pull model and incoming/outgoing edges are relative. From the global view, a push model along the incoming edges from the perspective of x is equivalent to a pull model along the outgoing edges from the perspective of one of x's neighbors, say n_2 [4], as illustrated in Figure 2.8-(b). By adopting both models for the two queries, respectively, a one-way edge representation is sufficient for calculating both $property(r, x)$ and $property(x, r)$ for any x in V.

We have presented the Δ-based incremental evaluation on both undirected and directed graphs. Next, we discuss its applicability and correctness.

2.4.3 Applicability and Correctness

First, Δ-based incremental evaluation targets vertex-specific queries. For non-vertex-specific queries, such as PageRank and connected components (CC), because they are already well-suited to the existing incremental graph computation models [117, 22], they can be incrementally evaluated without triangle inequalities (also supported by Tripoline).

Second, to apply Δ-based incremental evaluation to a type of vertex-specific query, a graph triangle inequality needs to be established. Note that the triangle inequality is derived based on the property of interest rather than the specific implementation of its queries. Given $property(u, *)$, where $*$ refers to any vertex in the graph, a triangle inequality among $property(u, r)$, $property(r, x)$, and $property(u, x)$ often can be intuitively derived

[4] Or vice versa if the pull model is assumed to be the default model.

based on the fact that a path from u to r, then to x is just one of the possible paths from u to x. Following this intuition, in Section 2.3, we have demonstrated the possibility of establishing triangle inequalities for several commonly seen graph problems.

Though triangle inequality is independent of the query implementation, some properties of the vertex function $f(v)$ are still closely relevant to the correctness of the triangle inequality-based incremental evaluation. Next, we mainly discuss two such properties: *monotonicity* and *safety under asynchrony*, which are formally defined below.

Definition 2. In vertex-centric programming framework, vertex function $f(v)$ is *monotonic* if all vertex values only change monotonically across iterations.

Definition 3. In vertex-centric programming framework, vertex function $f(v)$ is *safe under asynchrony*, or *async-safe* for short, if the vertex values still converge correctly even when $f(v)$ is executed asynchronously based on the new values of its neighbors calculated in the current iteration.

Note that, for vertex-centric graph algorithms, the above two properties are not rare. In fact, they are in the abstraction of many existing graph programming frameworks [66, 33, 117, 102]. For example, GraphLab [66] asynchronously executes graph algorithms for better efficiency, GRAPE [33] relies on the monotonicity of iterative graph algorithms for automatic parallelization, Subway [102] leverages the asynchrony and monotonicity to reduce the data transfer in out-of-memory graph processing, and most relevantly, KickStarter [117] requires monotonicity to support edge deletions in streaming graph processing. In the following discussion, we assume that the vertex function $f(v)$ is monotonic and async-safe.

In the following, for conciseness, we use $t_{init}(x)$ to denote the initial value of vertex x under the Δ-based incremental evaluation of query $q(u)$ (i.e., $t_{init}(x) = property(u, r) \oplus property(r, x)$), and $t_{conv}(x)$ to denote the correct converged value of x (i.e., $t_{conv}(x) = property(u, x)$).

Lemma 4. In Δ-based incremental evaluation, if vertex x's initial value $t_{init}(x) \succ t_{conv}(x)$, then at least one of its in-neighbors, say vertex z, must be initialized with value $t_{init}(z)$, such that $t_{init}(z) \succ t_{conv}(z)$, and z is on the path from source vertex u to x that yields $t_{conv}(x)$.

Proof Sketch. By contradiction, assume that all in-neighbors of x, denoted as z_i, $1 \leq i \leq k$ (where k is the number of neighbors of x), are initialized with their correct converged values $t_{conv}(z_i)$, $1 \leq i \leq k$, then the vertex r in the standing query $q(r)$ is on the paths from u to z_i that yield $t_{conv}(z_i)$. Also, o¡zl¿ne in-neighbor of x must be on the path from source vertex u to x that yields $t_{conv}(x)$. Together, we have that r is on one path from u to x that yields $t_{conv}(x)$. Thus, $t_{init}(x) = t_{conv}(x)$, which contradicts with the assumption in the lemma. □

Based on Lemma 4, we have the following conclusion.

Theorem 5. Given a vertex function $f(\cdot)$ that is monotonic and async-safe, if triangle inequality holds on the property that $f(\cdot)$ computes, the Δ-based incremental evaluation yields the same results as the non-incremental evaluation.

Proof Sketch. Consider an arbitrary vertex x, which is initialized with $t_{init}(x)$ by Δ-based incremental evaluation. First, based on triangle inequality, $t_{init}(x) \succeq t_{conv}(x)$, where $t_{conv}(x)$

is the correct converged value of vertex x. If $t_{init}(x) = t_{conv}(x)$, then based on monotonicity, the evaluation will not change its value, so it will remain correct in the end. Otherwise, if $t_{init}(x) \succ t_{conv}(x)$, by applying Lemma 4 on vertex x, we know there exists one in-neighbor of x, say z, which is on the path that yields $t_{conv}(x)$ and its initial value $t_{init}(z)$ $\succ t_{conv}(z)$. Similarly, we reapply Lemma 4 on vertex z. By repeating these, we can find a reversed path starting from vertex x, along which all the vertices have initial values that are greater than their correct converged values, and they are on the path from u to x that yields $t_{conv}(x)$. If the reversed path can reach the source vertex u, an activation of u will gradually stabilize all the vertex values along the path with their correct converged values, including x's value. Here, monotonicity ensures that the initial values of these vertices will be updated with their corresponding correct converged values (as $t_{init}(x) \succ t_{conv}(x)$), while async-safety ensures that these updates will not alter the converged values even when they are performed asynchronously. On the other hand, if the reversed path cannot reach the source vertex, then it would stay unchanged (the default initial value). □

Besides the theoretical correctness discussion of Δ-based incremental evaluation, our experimental evaluation also confirmed the correctness of results under many different testing cases (Section 2.6). Next, we discuss the benefits and costs of Δ-based incremental evaluation.

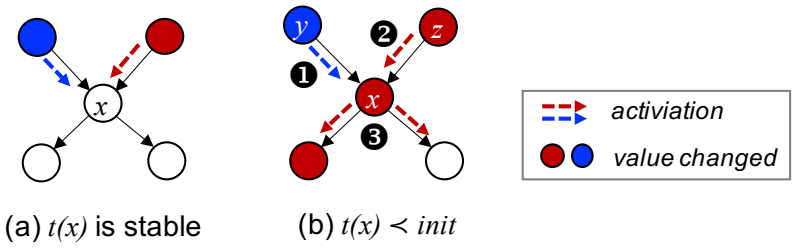


Figure 2.9: Benefits of Δ-based Incremental Evaluation.

2.4.4 Cost-Benefit Analysis

To examine the benefits of Δ-based incremental evaluation, we discuss how the two basic cases in its initialization: $t_{init}(x) = t_{conv}(x)$ and $t_{init}(x) \succ t_{conv}(x)$, affect the computations.

Figure 2.9-(a) illustrates the first case $t_{init}(x) = t_{conv}(x)$, where the vertex value will not be changed during iterations according to monotonicity, thus the vertex will never activate its out-neighbors (bottom two vertices) [5]. This means that all value propagations reaching x stop. In this way, it reduces the amount of computations. In the second case, as shown in Figure 2.9-(b), the initial value of x is not stable, but better than the default initial value (i.e., $t(x) \prec init$). In this case, it may "block" some value propagations (e.g., the blue one which yields a worse value than $t_{init}(x)$), but allow others (e.g., the red one which yields a better value than $t_{init}(x)$).

In both cases, the benefits come from the reduction of the vertex function evaluations. Thus, the benefits can be roughly captured by the *activation ratio*, denoted as R_{act}:

$$R_{act} = \frac{N_{act} \text{ with } \Delta\text{-based inc. eval.}}{N_{act} \text{ without } \Delta\text{-based inc. eval.}} \tag{2.11}$$

[5] They may still be activated due to changes of their other in-neighbors.

where N_{act} denotes the total number of times that the vertex function is evaluated. Note that $R_{act} \leq 1$ because the new initial values $\Delta(u, r)$ are no worse than the default ones. In general, the closer $\Delta(u, r)$ are to the stable values, the lower the R_{act} is. In addition, as a side-effect, the reduction of vertex activations often leads to fewer number of iterations – faster convergence.

On the other hand, Δ-based incremental evaluation also brings overhead. A direct cost comes from calculating the initial values – $\{property(u, r) \oplus property(r, x) \mid x \in V\}$. As $property(u, r)$ is fixed for the given standing query $q(r)$ and user query $q(u)$, the calculation simply traverses the results of $q(r)$ to read $property(r, x)$, $x \in V$, from an array. Due to spatial locality, this overhead is often negligible (e.g., about 0.3% for SSSP). Besides that, there are indirect costs regarding the incremental evaluation of standing queries. For example, for direct graphs, we incrementally evaluate not only $q(r)$, but also $q^{-1}(r)$. Depending on applications, the evaluation of $q^{-1}(r)$ may be counted as an overhead if its results are not needed. As we will discuss shortly, we may also want to incrementally evaluate multiple standing queries, whose costs may also be counted as the overheads depending on the application (more details in Section 2.4.5).

In summary, the performance improvements of Δ-based incremental evaluation mainly depend on the activation ratio R_{act}. Even with low R_{act}, the incremental evaluation only introduces limited direct overhead. Next, we discuss a key factor for R_{act} – the selection of standing query.

2.4.5 Standing Query Selection and Cost Management

The effectiveness of Δ-based incremental evaluation, roughly measured by R_{act}, depends on which query is selected as the standing query $q(r)$. A better selection may yield a lower R_{act}, thus a higher speedup. Moreover, is it worthwhile to select multiple standing queries? We address these questions next. First, we present two basic selection strategies.

Triangle-based Selection. As discussed in Section 2.4.4, the effectiveness of Δ-based incremental evaluation depends on how close the initial values $\Delta(u, r)$ are to the stable values (in terms of $\prec$), hence it is the better to select the standing query that yields lower values of $\Delta(u, r)$. Based on this, given user query $q(u)$, we select $q(r^*)$, such that,

$$r^* = \mathbf{arg\ min}_{r \in V} \sum_{\forall x \in V} property(u, r) \oplus property(r, x) \tag{2.12}$$

$$= \mathbf{arg\ min}_{r \in V}\ property(u, r) \cdot |V| \oplus \sum_{\forall x \in V} property(r, x) \tag{2.13}$$

However, in practice, we do not know the user query $q(u)$ – u can be any vertex in V. To find the best $q(r)$ overall, we need to compute the summation in Equation 2.12 for every u in V and select the one that minimizes the summation of those summations. Essentially, this requires collecting the $property(v_i, v_j)$ between every pair of vertices in the graph. Apparently, this is impractical for large graphs [6] even in non-streaming scenarios due to the high time and space complexities, not to mention the streaming scenarios where the property values change as the graph is updated.

Topology-based Selection. From the perspective of graph topology, it may be attempted to select the standing query $q(r)$ whose source vertex r is closer to the vertex u in the user

[6] For small graphs that are affordable for collecting these properties, the results can be directly cached – no need for incremental evaluation.

query $q(u)$ in terms of the number of hops, because in this way, u and r share more paths or path segments to other vertices. Interestingly, we find that this heuristic only works for some graph problems, such as SSSP and BFS, but not the others, like SSWP and Viterbi. The reason is that the heuristic may contradict the triangle inequalities. Taking SSWP as an example, in fact, the more hops that u and r are away from each other, the larger value $wide(u, r)$ might be, thus the better value $wide(u, r) \oplus wide(r, x)$ may possess.

Instead, for graph topology, we focus on the reachability of r in the standing query $q(r)$ to the other vertices in the graph. In fact, to effectively leverage the triangle inequality, there should be at least one path from r to vertex u in the user query $q(u)$, and to every other vertex x, $x \in V$; otherwise, $property(u, r) \oplus property(r, x)$ would be as "worst" as the default initial value (e.g., ∞ in SSSP). One simple yet reliable way to approximate the reachability is to select a query with a high-degree source vertex [7], which is more likely to reach a larger amount of vertices. Thus, we have the following heuristic for selecting standing query $q(r)$.

$$r^* = \mathbf{arg\ max}_{r \in V}\ degree(r) \tag{2.14}$$

As shown next, in practice, we adopt a solution combining the *triangle-based* and *topology-based* selections to achieve a balance between complexity and effectiveness. The key to exploiting this tradeoff is adopting multiple standing queries.

Selecting Multiple Standing Queries. First, we pre-select a set of K standing queries offline using the topology-base selection, that is, queries with the top-K high-degree vertices:

$$Standing_K = \{q(r_1), q(r_2), \cdots, q(r_K)\}$$

[7] Following the push model, here it refers to the out-degrees.

Then, at runtime, we pick the best one among the K standing queries based on the specific user query $q(u)$, according to a simplified version of Equation 2.13:

$$r^* = \underset{r \in Standing_K}{\textbf{arg min}} \; property(u, r) \tag{2.15}$$

Equation 2.15 is based on our experimental finding that, for the standing queries with top-K high-degree vertices, there is a limited variation for the summation in Equation 2.13.

In this way, the standing query selection not only becomes query-specific, but also incurs negligible runtime overhead.

Managing the Costs. However, incrementally evaluating multiple standing queries may take longer – each time the graph is updated, it has to ensure that the evaluation of every standing query reaches stabilization. Here, we present two ways to alleviate these costs.

First, we evaluate the K standing queries in *batch* mode. That is, we maintain a combined frontier for all the active vertices among the K queries, and for each active vertex v, we apply the vertex function for the K standing queries together (those are inactive on v are masked). In this way, both the graph and vertex value arrays of standing queries can be accessed in a coalesced manner, thus incurring much less cost compared to evaluating each standing query separately.

Second, we can adjust K to exploit the tradeoff between the maintaining cost of standing queries and the effectiveness of Δ-based incremental evaluation. When the user queries are made relatively more frequently than the graph updates (in batches), we may afford a larger K, as the overhead can be amortized by more user queries. In the opposite scenarios, we may reduce K such that the (incremental) standing query evaluation can finish quickly, and the following user query evaluation can start earlier.

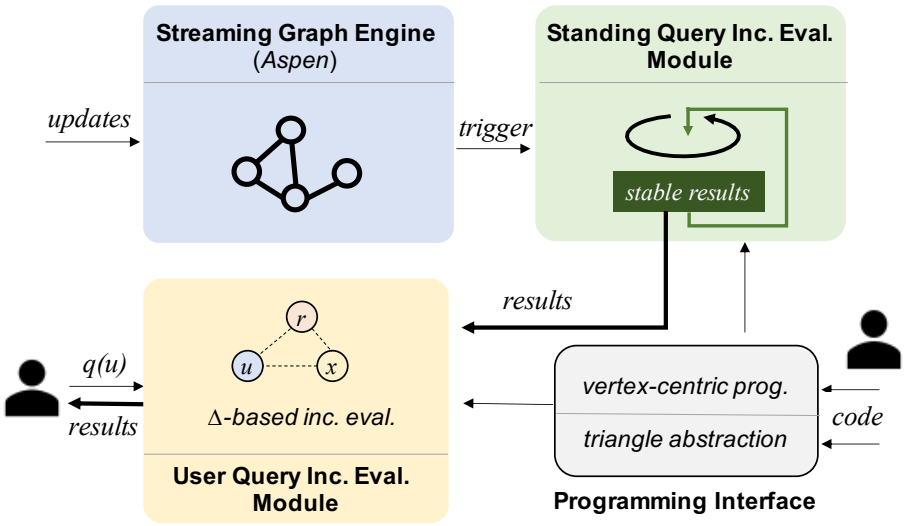


Figure 2.10: System Architecture of Tripoline.

So far, we have discussed the major aspects of the proposed Δ-based incremental evaluation. Next, we present a new streaming graph processing system that supports Δ-based incremental evaluation for vertex-specific queries.

2.5 Implementation of Tripoline

Based on the proposed generalized incremental evaluation, we developed *Tripoline*, a shared-memory streaming graph processing system. To our best knowledge, Tripoline is the first system of this kind that supports incremental processing of vertex-specific queries requiring no a priori knowledge of source vertices. Figure 2.10 illustrates its high-level structure, which consists of four major components:

- A *streaming graph engine* that accepts graph updates and maintains the data structures of the current graph. As the focus of this work is not to build such an engine, Tripoline adopts a state-of-the-art streaming graph engine called *Aspen* [27]. Inter-

nally, Aspen leverages a compressed tree-based graph representation to achieve both high-space efficiency and high-throughput graph updates. Note that the released Aspen does not support edge weights; we extended it to support this feature, so that more applications can be evaluated.

- A *standing query evaluation module* that continuously and incrementally evaluates a set of standing queries upon graph updates. For better efficiency, we implemented the *batch* mode mentioned in Section 2.4.5.

- A *user query evaluation module* that employs Δ-based incremental evaluation to fulfill the user requests.

- Finally, a *programming interface* that not only provides the conventional vertex-centric programming, but also offers a triangle abstraction for specifying the triangle inequality of the specific graph problem. Basically, the developers need to overwrite the generic addition and comparison operators $\oplus$ and $\succeq$, respectively.

Configuration and Parameters. The above three runtime modules (colored boxes in Figure 2.10) are configured to be executed exclusively (i.e., in serial), though each of them runs in parallel individually. This configuration maximizes the resource availability for each task: graph updates, standing query evaluation, and user query evaluation, respectively.

In our current setup, K standing queries are first selected based on their reachability to all possible source vertices in the user queries, for which we choose the top-K high-degree vertices as an approximation (i.e., the "topology-based selection" in Section 2.4.5). With that, the only parameter to be tuned is K, which depends on the query type and the

memory capacity of the machine (a larger K means the results of more queries need to be kept in memory). When the system is set up initially, K can be tuned and selected using a few sample values (as shown later in the evaluation – Table 2.5). To ease its deployment, a basic auto-tuner can be added to make the K selection transparent to the users. Furthermore, the standing query selection might be further improved based on the distribution of user queries when it is available.

Note that non-vertex-specific queries (e.g., PageRank and CC) can also be implemented on Tripoline, in which case, the system simply maintains them incrementally as the standing queries, like the existing incremental query evaluation [84, 22]. Similarly, vertex-specific queries with a priori knowledge can be treated as the standing queries, so that they can be maintained incrementally and answered directly.

In addition, Tripoline includes a set of built-in benchmarks for which the vertex functions are designed to satisfy the desired properties for correctness (see Section 2.4.3). Table 2.1 summarizes their vertex functions.

2.6 Evaluation

2.6.1 Methodology

We compiled the built-in benchmarks of Tripoline using g++ 8.3, and ran the experiments on a 32-core Linux server. The server is equipped with Intel Xeon CPU E5-2683 v4 CPU and 512GB memory, running on CentOS 7.9.

Table 2.1: Benchmarks in Tripoline

Bench.	Pseudo-code of Vertex function

BFS	**for** each out-neighbor n of s
	  level(n) = min { level(n), level(s) +1 };
	  if level(n) changed **then** add n to $frontier$;

SSSP	**for** each out-neighbor n of s
	  dist(n) = min { dist(n), dist(s) + w(s, n) };
	  if dist(n) changed **then** add n to $frontier$;

SSWP	**for** each out-neighbor n of s
	  wide(n) = max { wide(n), min { wide(s), w(s, n) } };
	  if wide(n) changed **then** add n to $frontier$;

SSNP	**for** each out-neighbor n of s
	  naro(n) = min { naro(n), max { naro(s), w(s, n) } };
	  if naro(n) changed **then** add n to $frontier$;

Viterbi	**for** each out-neighbor n of s
	  vite(n) = max { vite(n), vite(s) / w(s, n) };
	  if vite(n) changed **then** add n to $frontier$;

SSR	**for** each out-neighbor n of s
	  rech(n) = true;
	  if rech(n) changed **then** add n to $frontier$;

Radii	**for** each out-neighbor n of s
	  dist1(n) = min { dist1(n), dist1(s) + w(s, n) };
	  $\cdots$
	  dist16(n) = min { dist16(n), dist16(s) + w(s, n) };
	  if any dist$*$(n) changed **then** add n to $frontier$;

SSNSP	**for** each out-neighbor n of s
	  if level(n) == level(s) + 1 **then**
	  delta(n) += delta(s); add n to $frontier$;
	ssnsp(s) += delta(s);

The experiments used a set of four real-world large graphs whose statistics are listed in Table 2.2. Like many existing graph systems, such as PowerGraph [41], PowerLyra [20], and Tigr [86], Tripoline mainly targets power-law graphs, which are more common in real-world applications. Thus, this evaluation focuses on such kinds of graphs. Similar to prior work [105, 117, 75], we assume that a substantial portion of edges – 50%, 60%, and 70%, has been streamed in, then the remaining edges of the graph are streamed in batches of randomly selected edges. By default, we set the update batch size to 10K. Note that, under the design of Tripoline, the impact of update batch size is limited to the standing query evaluation, which has been intensively studied in the evaluation of Aspen [27]. But, for completeness, we have included results for different batch sizes (from 1K to 500K).

As to the number of standing queries K, by default, we set it to 16. To demonstrate the tradeoff between benefits and costs in adopting multiple standing queries (see Section 2.4.5), we also vary the value of K from 1 to 64 and report their impacts to the standing and user query evaluations.

For each benchmark, we randomly selected 256 non-trivial user queries (whose source vertices are of degree more than two). After a batch of graph updates have been applied and the evaluation of standing queries have been re-stabilized, we evaluated each of the 256 user queries three times repetitively and reported the averaged performance. To obtain sufficient samples, we collected the performance results from the first five consecutive batches of updates at each preset starting point (50%, 60%, and 70% portions of edges).

Table 2.2: Statistics of Input Graphs

Graph	Type	$\|V\|$	$\|E\|$	Avg. Degree
Orkut	undirected	3.1M	234M	76.3
Friendster	undirected	68M	2.9B	75.7
LiveJournal	directed	4.8M	69M	28.3
Twitter	directed	41M	1.5B	70.5

Table 2.3: Speedups of Δ-based Incremental Evaluation over Non-Incremental Evaluation.

Each entry is in the format of *average speedup* [*speedup standard deviation*, *average time (seconds) with incremental eval.*] of 256 user queries

Graph	SSSP	SSWP	Viterbi	BFS
OR-50	2.52 [1.88, 0.15]	31.70 [6.76, 0.01]	40.16 [5.17, 0.01]	1.23 [1.04, 0.12]
OR-60	2.42 [1.69, 0.17]	33.91 [6.45, 0.01]	37.94 [3.95, 0.01]	1.25 [1.20, 0.13]
OR-70	2.45 [1.82, 0.18]	34.88 [6.14, 0.01]	39.90 [4.29, 0.01]	1.30 [1.38, 0.15]
FR-50	1.34 [0.13, 10.90]	29.69 [5.32, 0.40]	35.49 [5.38, 0.46]	1.02 [0.20, 6.62]
FR-60	1.34 [0.11, 12.26]	35.23 [5.86, 0.38]	41.48 [5.31, 0.38]	1.02 [0.24, 7.16]
FR-70	1.34 [0.18, 13.79]	36.09 [5.76, 0.42]	39.95 [3.87, 0.45]	1.01 [0.11, 8.63]
LJ-50	1.68 [0.62, 0.14]	9.27 [1.88, 0.02]	22.91 [4.60, 0.02]	1.10 [0.30, 0.08]
LJ-60	1.81 [0.87, 0.13]	11.56 [2.30, 0.01]	26.88 [5.47, 0.02]	1.12 [0.36, 0.07]
LJ-70	1.74 [0.73, 0.15]	10.60 [2.00, 0.02]	23.4 [4.60, 0.02]	1.12 [0.33, 0.08]
TW-50	1.97 [1.25, 1.24]	13.17 [2.28, 0.14]	17.61 [2.41, 0.13]	1.49 [0.80, 0.85]
TW-60	1.95 [1.18, 1.45]	15.97 [2.55, 0.13]	19.14 [2.33, 0.13]	1.56 [0.98, 0.94]
TW-70	2.11 [1.84, 1.56]	17.23 [2.61, 0.13]	21.41 [2.74, 0.13]	1.61 [1.12, 0.98]
avg.	1.89	23.28	30.52	1.24

Table 2.3: Speedups of Δ-based Incremental Evaluation over Non-Incremental Evaluation. (Continued)

Each entry is in the format of *average speedup* [*speedup standard deviation*, *average time (seconds) with incremental eval.*] of 256 user queries

Graph	SSNP	SSR	Radii	SSNSP
OR-50	26.30 [5.42, 0.01]	10.40 [0.31, 0.01]	1.21 [0.05, 2.22]	1.09 [0.18, 0.25]
OR-60	29.06 [5.30, 0.01]	10.86 [0.27, 0.01]	1.22 [0.06, 2.43]	1.09 [0.18, 0.27]
OR-70	30.47 [5.13, 0.01]	11.70 [0.61, 0.01]	1.23 [0.05, 2.75]	1.10 [0.19, 0.29]
FR-50	17.30 [3.06, 0.45]	9.28 [0.27, 0.47]	1.16 [0.05, 50.09]	1.00 [0.03, 8.59]
FR-60	18.77 [2.96, 0.45]	10.44 [0.23, 0.45]	1.18 [0.04, 56.43]	1.00 [0.03, 9.58]
FR-70	20.56 [3.04, 0.45]	11.43 [0.30, 0.45]	1.16 [0.05, 61.52]	1.00 [0.03, 9.99]
LJ-50	10.23 [2.08, 0.02]	4.94 [0.39, 0.02]	1.14 [0.03, 1.28]	1.03 [0.11, 0.18]
LJ-60	11.53 [2.35, 0.02]	5.50 [0.43, 0.02]	1.16 [0.05, 1.31]	1.03 [0.11, 0.20]
LJ-70	12.56 [2.49, 0.02]	6.01 [0.46, 0.02]	1.17 [0.04, 1.49]	1.03 [0.11, 0.20]
TW-50	12.76 [1.77, 0.13]	7.87 [0.13, 0.15]	1.16 [0.07, 11.42]	1.16 [0.32, 2.18]
TW-60	13.42 [1.81, 0.14]	8.32 [0.25, 0.15]	1.15 [0.07, 11.85]	1.18 [0.34, 2.27]
TW-70	15.94 [2.07, 0.13]	9.21 [0.20, 0.15]	1.19 [0.06, 13.51]	1.20 [0.41, 2.51]
avg.	18.24	8.83	1.18	1.08

2.6.2 Speedups

Table 2.3 lists the speedups of Δ-based incremental evaluation of user queries over the non-incremental query evaluation and the average time of the former. Overall, we observe a wide range of speedups across benchmarks. The highest come from the case of Viterbi (17.6-41.5×), while the lowest are observed on SSNSP (1.0-1.2×). In between, the results show significant performance improvements in the cases of SSWP (9.3-36.1×), SSNP (10.2-30.5×), and SSR (5.0-11.7×), and modest speedups in the remaining cases: SSSP (1.3-2.5×), BFS (1.0-1.6×), and Radii (1.1-1.2×). This large variation of speedups clearly indicates that the effectiveness of Δ-based incremental evaluation depends on the graph problems, in particular, their graph triangle inequalities. As mentioned in Section 2.4.4,

Table 2.4: Vertex Activation Ratio of Δ-based Incremental Evaluation over Non-Incremental Evaluation.

Each entry is: average [standard derivation] of 256 user queries

	OR-60	FR-60	LJ-60	TW-60
SSSP	44.4% [13.1%]	61.7% [4.8%]	56% [12.2%]	52.8% [11.1%]
SSWP	1.9E-7 [9.0E-8]	1.3E-8 [3.6E-9]	0.79% (8.8%)	4.0E-8 [3.0E-8]
Viterbi	3.5E-7 [8.9E-7]	6.7E-8 [3.2E-7]	0.95% [9.1%]	1.7E-7 [2.8E-7]
BFS	82.2% [18.2%]	98% [6.9%]	89.4% [16.5%]	65.8% [22.6%]
SSNP	1.9E-7 [1.4E-7]	1.4E-8 [9.4E-9]	0.78% [8.8%]	3.6E-8 [2.3E-8]
SSR	3.3E-7 [0]	1.7E-8 [0]	0.78% [8.8%]	3.2E-8 [2.8E-9]
Radii	98.9% [3.7%]	91.9% [4.5%]	92.21% [4.06%]	93.9% [6.8%]
SSNSP	98.9% [4.3%]	99.97% [0.2%]	98.58% [4.88%]	94.9% [10.1%]

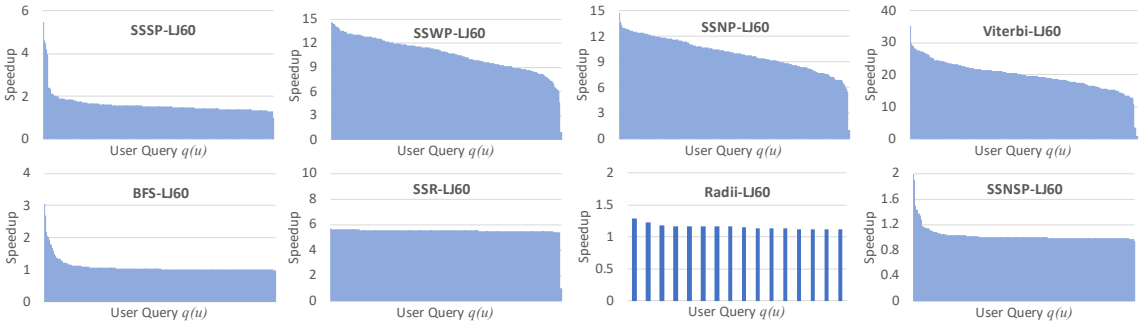


Figure 2.11: Speedup Distributions of 256 User Queries (16 queries in the case of Radii; x-axis is for the user queries while y-axis is for the speedups of Δ-based incremental evaluation; the user queries are sorted by the corresponding speedups.)

the effectiveness can be measured more directly by the activation ratio R_{act}. Table 2.4 reports this ratio for cases where the graph is 60% loaded.

Overall, we find that the results are consistent with the speedups – lower activation ratios usually correspond higher speedups. More specifically, we find R_{act} is extremely low (less than 1%) in the cases of SSWP, SSNP, and Viterbi, which means more than 99% of the vertex activations are avoided by incremental evaluation. Our further investigation reveals an interesting fact: the initial values $\Delta(u, r)$ of incremental evaluation are nearly all stable

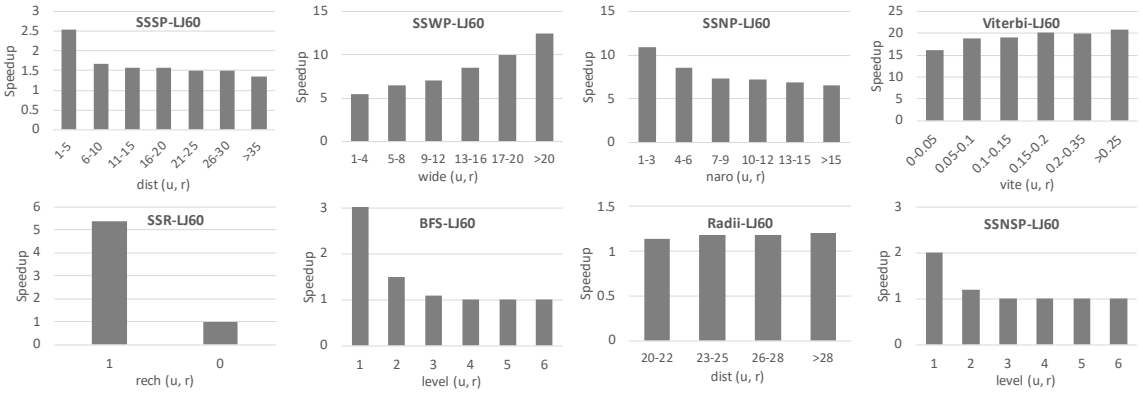


Figure 2.12: Correlations between Speedups and $property(u, r)$ for Verifying the Standing Query Selection Heuristic.

values, that is, the "=" part of the inequality holds – the first case of benefits we discussed in Section 2.4.4. There could be multiple reasons causing this phenomenon. One of them is the min-max nature of the graph problems. In the cases of SSWP and SSNP, the whole vertex function is based on the calculation of `min` and `max`. In these cases, it is not hard to prove that, for undirected graphs or strongly connected components (SCC) of a directed graph, if $property(u, r) \neq property(r, x)$, then the inequality turns to be the equality, thus, the initial values are already stable. For Viterbi, one key reason for the high stable ratio could be related to the max-division operation in the vertex function. The function tries to choose the edge with the lowest weight to propagate the probability – dividing value by the edge weight, and as we know, the lowest edge weight is one, thus the probability is likely to stay the same simply because "$vite(v_1, v_2)$ divided by 1 equals $vite(v_1, v_2)$". Back to the inequality, as long as there exists a path from u to r or a path from r to x where the edge weights are all ones, then the inequality becomes an equality. This effect can be significantly amplified by the power-law nature of the graphs, where u, r, and x are only a few hops apart, thus the conditions are very likely to become true.

At the other end of the stable ratio spectrum, Radii and SSNSP show the highest stable ratios (over 90%), meaning that less than 10% of vertex function activations are actually saved by incremental evaluation. Note that, even though Radii mainly involves a group of SSSP evaluations, its stable ratios are much higher than those of SSSP. This is because the number of vertex activations in Radii is bottlenecked by the slowest SSSP query – the one with the most number of vertex activations. For SSNSP, our evaluation involves two rounds: (i) a BFS round which computes the level of each vertex and (ii) a counting round which counts the number of paths corresponding to the lowest levels. The activation ratios reported in Table 2.4 are for the second round. In this case, the main problem comes from the *conditional* inequality as shown in Figure 2.6-(d). Our profiling shows that the condition is false for 90% of the cases during the initialization, which substantially limits the effectiveness of Δ-based incremental evaluation, resulting in a high activation ratio.

Besides the aggregated speedups shown in Table 2.3, we also report the speedups of the 256 individual queries on graph LiveJournal with 60% edges loaded in Figure 2.11. From the results, we can see three patterns roughly. For SSSP, BFS, and SSNSP, the speedup distributions are mostly biased, followed by SSWP, SSNP, Viterbi, and Radii, and finally, the distribution of SSR is almost uniform. The variations, to a large extent, depend on the $property(u, r)$, which connects the user query $q(u)$ and standing query $q(r)$. We thus discuss it together with the standing query selection next.

2.6.3 Standing Query Selection

Standing query selection is critical to the effectiveness of Δ-based incremental evaluation. To examine its impact, we grouped the speedups by $property(u, r)$ – the heuristic

that we use for selecting the standing query (see Section 2.4.5). The results are reported in Figure 2.12. In the cases of SSSP, SSWP, SSNP, BFS, and SSNSP, there are clear correlations between $property(u, r)$ and the speedup; for Viterbi, the trend is less obvious, but still observable; for Radii, no significant enough correlation is observed; finally, for SSR, as the property is binary, we will discuss it separately. Note that whether the trend is increasing or decreasing depends on the comparison operator $\succeq$. For SSSP, SSNP, BFS, Radii, and SSNSP, $\succeq$ is $\geq$, while for SSWP and Viterbi, $\succeq$ is $\leq$. From this perspective, the trends align well with our standing query selection heuristic – the "lower" the $property(u, r)$ is, the higher the speedup is achieved. Note that, in the case of SSR, the heuristic always chooses the standing query with $property(u, r) = 1$, which is also the one with a higher speedup. For Radii, we used the maximum distance among 16 SSSP queries, in which case the "averaging effect" blurs the correlation.

Also, note that, in the cases of SSSP, BFS, and SSNSP, the speedups are more sensitive to $property(u, r)$ when its value is low, which explains the biased speedup distributions in the corresponding graph problems shown in Figure 2.11.

Besides the selection heuristic, another important factor to the performance is the number of standing queries – K (see Section 2.4.4). A larger K offers more options for selecting the standing query, thus potentially making the incremental evaluation more effective; on the other hand, a larger K can also increase the costs: (i) the time for incremental standing query evaluation; and (ii) the time for selecting one from the K standing queries for applying the triangle inequality. For the latter, as the selection simply accesses K vertex values in the property arrays of K standing queries and chooses one based on

Table 2.5: Benefits and Costs of Incrementally Evaluating K Standing Queries (on graph TW-60).

Each entry is: avg. user query speedup [standing queries eval. time(s)]

#LQ	1	2	4	16	64
SSSP	1.43 [0.30]	1.43 [0.45]	1.70 [0.71]	1.95 [1.42]	2.36 [4.73]
SSWP	16.28 [0.30]	15.98 [0.47]	15.53 [0.71]	15.97 [1.23]	14.97 [3.51]
Viterbi	18.63 [0.30]	17.87 [0.45]	19.09 [0.69]	19.14 [1.20]	17.84 [3.44]
BFS	1.03 [0.32]	1.04 [0.43]	1.23 [0.67]	1.56 [1.29]	1.86 [4.45]
SSNP	13.44 [0.37]	13.09 [0.45]	13.24 [0.75]	13.42 [1.37]	13.84 [3.92]
SSR	8.39 [0.35]	8.60 [0.45]	8.22 [0.66]	8.32 [1.36]	8.12 [4.05]
Radii	1.11 [0.36]	1.13 [0.53]	1.16 [0.97]	1.15 [1.59]	0.84 [4.68]
SSNSP	1.01 [1.74]	1.00 [0.58]	1.09 [0.91]	1.18 [2.10]	1.28 [6.46]

Equation 2.12, the runtime cost is negligible. For the former, we report the standing query evaluation time for K from 1 to 64 in Table 2.5 (numbers in brackets).

Table 2.5 also reports how K affects the speedups. For SSSP, BFS, and SSNSP, larger K tends to yield higher speedups; For the others, there are no similar trends, which means that adding more standing queries does not necessarily increase the effectiveness of Δ-based incremental evaluation; for such cases, a single standing query is sufficient. Note that, when K increases, the evaluation time of standing queries only increases sub-linearly, thanks to the *batch* mode execution. As to the space cost, the value can be computed by $(8 + 2)$ bytes $\times$ $Bsize$ $\times$ $|V|$, where 8 is the size of vertex value (`double`/`long`) and 2 is the two masks (`boolean`) of vertex activeness in the prior and current iterations. In general, users can tune K based on the sensitivity of their graph problem and the resource constraints.

Table 2.6: Standing Query Evaluation Time under Different Update Batch Sizes on LJ-60 and FR-60.

Graph	Bsize	SSSP	SSWP	Viterbi	BFS	SSNP	SSR	Radii	SSNSP
LJ-60	1K	0.09	0.08	0.09	0.09	0.09	0.07	0.13	0.17
	10K	0.13	0.10	0.10	0.09	0.11	0.08	0.17	0.19
	50K	0.15	0.12	0.12	0.11	0.13	0.09	0.20	0.22
	100K	0.16	0.14	0.14	0.11	0.16	0.10	0.21	0.22
	500K	0.23	0.18	0.19	0.17	0.20	0.14	0.29	0.27
FR-60	1K	2.09	1.66	1.78	1.86	1.72	1.73	2.29	3.77
	10K	2.50	1.95	2.00	2.09	1.87	1.78	2.73	4.04
	50K	2.69	2.16	2.36	2.30	2.22	2.03	3.11	3.52
	100K	3.08	2.52	2.60	2.67	2.44	2.38	3.30	3.88
	500K	4.14	3.52	3.55	3.70	3.59	3.33	4.30	4.83

2.6.4 Graph Streaming

Next, we briefly report the impact of the graph update batch size on the standing query evaluation. A detailed evaluation can be found in Aspen [27]. Table 2.6 shows the standing query evaluation time with update batch size varying from 1K to 500K. The results show that the evaluation time increases sub-linearly as the batch size increases. The main reason for the sub-linear increase is that computations for handling different new edges are largely shared. For example, many new edges may appear on the same paths, thus sharing the activations of vertices along the paths. Moreover, the efficient data structure (a purely functional tree) ensures fast graph data access for incremental query evaluation.

2.6.5 Integration into Differential Dataflow

Though Tripoline is implemented based on Aspen [27], the idea of triangle inequality-based optimization may also be adopted in other streaming graph systems. To demonstrate

its generality, we examined the potential of adopting it in a state-of-the-art general-purpose streaming framework, called *Differential Dataflow* (DD) [84, 79].

In fact, the latest version of DD also supports inter-query sharing, called *shared arrangements* [79]. In earlier versions of DD, each query [8] needs to maintain an indexed state over the input stream independently. In the context of streaming graphs, this means that each query needs to maintain its own indexed graph (for outgoing and/or incoming edges) over a stream of edge pairs. This creates unnecessary redundancies when different graph queries want to access the same input stream (edge-pair stream). Shared arrangements address this issue by allowing different queries to share the same indexed state (graph), rather than maintaining its own copy. Note that shared arrangements are an orthogonal improvement to the proposed triangle inequality optimization—the former shares the indexed graph data structure across queries while the latter "shares" the query evaluation state, that is, the vertex values (e.g., distances of all vertices to the source vertex in SSSP) across queries. The latter requires establishing the triangle inequalities to be applicable.

Experiment Setup. We pulled the latest version of DD from its GitHub repository [9]. To integrate the triangle inequality optimization, we added a `filter` to its graph processing dataflow. The `filter` applies a predicate to each element of a collection, and removes those for which the predicate returns `false`. In specific, the predicates are constructed based on triangle inequality $\Delta(u, r) \succeq property(u, x)$ (see Equations 2.2 and 2.9). For other operators used in the dataflow (such as `join_map`, `concat`, and `reduce`), we kept them intact.

[8] Here, a query refers to a type of queries in our context.

[9] `https://github.com/TimelyDataflow/differential-dataflow`, Jan 22, 2021.

Table 2.7: Performance of Differential Dataflow with Triangle Inequality Optimization on LJ and TW at 60% and 100%.

(DD-SA: differential dataflow with shared arrangements;
DD-SA-Tri: DD-SA with triangle inequality optimization)

Graph	Method	BFS	SSSP	SSWP
LJ-60	DD-SA	0.97s	6.90s	3.50s
	DD-SA-Tri	0.93s	2.68s	0.48s
	Speedup	[1.04×]	[2.57×]	[7.29×]
TW-60	DD-SA	6.91s	42.88s	22.97s
	DD-SA-Tri	7.23s	10.74s	5.75s
	Speedup	[0.96×]	[3.99×]	[3.99×]
LJ-100	DD-SA	1.10s	8.41s	4.63s
	DD-SA-Tri	1.11s	3.24s	0.52s
	Speedup	[0.99×]	[2.60×]	[8.90×]
TW-100	DD-SA	10.69s	58.63s	32.68s
	DD-SA-Tri	10.71s	14.72s	7.74s
	Speedup	[1.00×]	[3.98×]	[4.22×]

Note that the above integration may not be the only way to adopt triangle inequality optimization into DD. We choose this design for its simplicity and modularity - it isolates the modifications to one dataflow operator, leaving other parts of the graph processing dataflow intact. A more intrusive integration that yields better performance might be possible, but requires a redesign of the existing DD to some extent.

Due to space limits, we focus our evaluation on three query benchmarks (BFS, SSSP, and SSWP) and two graphs (LJ and TW), at 60% and 100% loaded ratios. For each configuration, we issued 256 queries (the same as the prior experiments) and collected the average time of query evaluation.

Performance Results. Table 2.7 reports the performance with and without the triangle inequality optimization. Note that the baseline (DD-SA) is the DD with shared arrange-

Table 2.8: Reduction of *reduce* Operations for DD-SA with Triangle Inequality Optimization on LJ-100.

Graph	Method	BFS	SSSP	SSWP
LJ-100	DD-SA	9156594	30418846	20622003
	DD-SA-Tri	8956638	17570555	6292821
	Reduction	[1.02×]	[1.73×]	[3.28×]

ments enabled. In general, the results are of similar trends as those reported for Tripoline (see Table 2.3): (i) for SSSP and SSWP, the speedups are more significant, ranging from 2.57× to 3.99× for SSSP and 3.99× to 8.90× for SSWP; (ii) by contrast, the speedups for BFS are limited, actually they are close to one. In the context of DD, the effectiveness of triangle inequality optimization can be reflected by the number of invocations of the downstream `reduce` operator, which are shown in Table 2.8. For SSSP and SSWP, there are significant reductions in the invocations of `reduce` operator, while for BFS, the reduction is very limited. These results align with the speedups of the three types of queries.

2.7 Summary

This work reveals a fundamental limitation in the existing streaming graph systems – lack of incremental evaluation for queries without a priori knowledge. To address the limitation, this work proposes to leverage the graph triangle inequalities that can be naturally derived from vertex-specific graph problems to enable such capabilities. This idea leads to a generalized incremental processing design for vertex-specific queries, in which the correctness is ensured by the triangle inequality and proper design of the vertex functions, and the efficiency is optimized based on the "distance" between the user and standing

queries. Finally, our evaluation of the developed system Tripoline confirms the effectiveness of the proposed techniques for a spectrum of graph problems on real-world graphs.

Chapter 3

Scaling Incremental Graph Query Evaluation for Large Update Batches

3.1 Introduction

While most existing graph system research has focused on static graphs, real-world graphs are usually dynamic. For example, on social networks, users join, connect, and interact with each other over time. New friendships, status updates, and interactions constantly change the graph structure [23]. In online recommendation systems, as users rate, review, or interact with items, the graph that represents the user-item relationships evolves [121]. More obviously, transportation networks, such as road networks or airline

networks, undergo constant changes due to factors like traffic patterns, road closures, and flight schedules [67].

Motivated by the dynamic nature of real-world graphs, a series of systems have been proposed recently for changing graphs, such as `Kineograph` [22], `Chronos` [43], `Tornado` [105], `KickStarter` [117], `Aspen` [27], `GraphBolt` [75], `Ingress` [40], `Tripoline` [56], and more recently `RisGraph` [34]. Instead of re-evaluating the queries from scratch, most of these systems incrementally update query results in response to the changes to the graph.

For path-based algorithms like single-source shortest path (SSSP), state-of-the-art incremental approaches (e.g., `RisGraph` [34]) have shown great scalability—handling large batches of edge insertions up to 30-50% of the graph size (see Figure 3.1). However, it remains a fundamental challenge to scale the incremental evaluation for edge deletions and weight updates. As shown in Figure 3.1, the existing incremental method can only scale the batches of edge deletions and weight updates up to 10-15% of the graph size. In fact, efficient handling of substantial graph updates, especially edge deletions and weight updates, is crucial for real-world analytics. For example, in dynamic communication networks, link weights often signify key attributes like latency, bandwidth, reliability, or cost. These attributes fluctuate based on user demands and network conditions that vary over time, causing substantial graph updates [48, 92, 57]. In evolving graph analysis, when comparing two temporally distant snapshots of the same graph, the extent of changes between them can be considerable, leading to a large update batch (e.g., 30% edges of the Stack Overflow temporal network [63]).

Figure 3.1: Scalability of Incremental Evaluation (The maximum update ratio where inc. eval. is faster than the full eval. on LiveJournal graph).

A closer examination of the existing incremental graph query processing systems highlighted a few challenges in scaling up the handling of edge deletions and weight updates:

- *Expensive dependency tracing.* When an edge is deleted, it is required to find all the affected vertices. It is intuitive to trace down the dependencies (like a tree) originating from the deleted edge [117, 34] (see Section 3.2.1). This *top-down* dependency tracing requires to access both the graph and the dependency data, making it the dominating cost of edge deletion handling (70-80% based on our observation).

- *Two-round handling.* Existing incremental systems [117, 34] handle edge weight changes in two rounds: delete the edges with their old weights, then reinsert them with the new weights. While being intuitive, this solution often involves a large amount of unnecessary computations.

- *Unawareness of workload.* State-of-the-art graph systems employ a single processing strategy for updating batches of varying sizes. However, substantial changes in batch

size result in different computation characteristics, potentially leading to sub-optimal performance.

To address the above challenges, this work introduces three key techniques for scaling the incremental evaluation for edge deletions and weight updates: (i) *bottom-up dependency tracing*; (ii) *direct edge weight change handling*; (iii) *workload-adaptive evaluation*.

- ***Bottom-up dependency tracing*** traverses the dependency data (e.g., a tree) bottom-up to find the vertices affected by edge deletions. Unlike top-down tracing, it solely relies on the dependency data, bypassing the graph access.

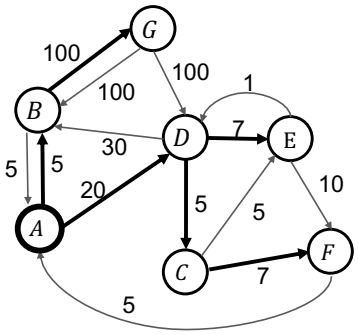
- ***Direct weight change handling*** directly processes edge weight updates in a single round. The key is to separate weight increases and decreases and treat them in a way similar to edge insertions and deletions. To support it, this work designs a method to test if a weight change violates the monotonicity of the graph algorithm. Based on the test result, the system chooses the right treatment.

- ***Workload-adaptive evaluation*** addresses the changing behaviors of computations due to changes in the workload volume. It automatically selects the dependency tracing strategy and the representation of the active vertices. In general, for relatively small batches, top-down tracing with a sparse representation is chosen, while for large batches, bottom-up tracing with a dense representation is used.

To integrate the above techniques, we implemented a new graph system for incremental query evaluation—`IncBoost`. It builds upon the widely used in-memory graph processing framework, `Ligra` [106]. Our evaluation focuses on comparing the performance

```
/* single-source shortest path */
f_sssp(v) {
   for each out-neighbor n of v {
      if (dist[n] > dist[v] + w(v,n)) {
         dist[n] = dist[v] + w(v,n);
         add n to frontier;
      }
}
```

(a) vertex function



(b) a directed graph

Iter#	A	B	C	D	E	F	G	Frontier
0	0	∞	∞	∞	∞	∞	∞	{A}
1	0	5	∞	20	∞	∞	∞	{B, D}
2	0	5	25	20	27	∞	105	{C, E, G}
3	0	5	25	20	27	32	105	{F}
4	0	5	25	20	27	32	105	{}

(c) iterative evaluation of *SSSP(A)* from scratch

Figure 3.2: Full Evaluation of Query $SSSP(A)$ (thick edges are dependent edges for the given query).

of `IncBoost` against the state-of-the-art system, `RisGraph`. Our results show that `IncBoost` can boost the update batch size from 10-15% to 50-60% of the graph size for edge deletions and weight changes (as shown in Figure 3.1), without losing the benefits of incremental evaluation. More specifically, for large update batches, our results indicate up to 1.6× speedup in dependency tracing with the bottom-up approach, while the direct weight update handling delivers 2.1× speedup over the two-round approach. Overall, `IncBoost` achieves up to 3.1× and 203× speedups for edge deletions, 4.9× and 299× speedups for edge weight updates over `RisGraph` and `KickStarter`, respectively.

3.2 Background

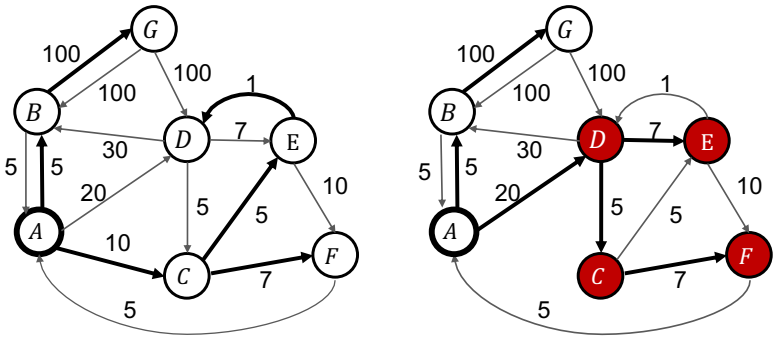
3.2.1 Existing Incremental Methods

To avoid the expensive full evaluation each time after the graph is updated, incremental query evaluation has been proposed [105, 117, 75, 27, 40, 56, 34]. Next, we present the basic ideas of incremental query evaluation with respect to the three types of graph updates[1]: (i) edge insertions, (ii) edge deletions, and (iii) weight updates. We will focus on the widely studied monotonic path-based graph algorithms and use the example in Figure 3.2 to help explain the ideas.

Edge Insertion Handling. Assume a new edge $(A, C, 10)$ is inserted to the graph in Figure 3.2-b. The edge creates a new way to reach C through A which may result in a better value for C. To find it out, we can apply the vertex function on A but limit its scope to only the out-neighbor C (like an edge function). Based on vertex A's prior result, which is 0 (see Figure 3.2-c) and the weight of the new edge "10", a new best value "10" is found for C. Next, we need to propagate this new value of C to the other vertices in the graph. To achieve this, we can put C to the frontier and resume the iterative query evaluation, as illustrated by Figure 3.3-c. Once all values are converged again, the latest shortest distances are found.

Edge Deletion Handling. To handle edge deletions, the system needs to track the dependencies among vertices and memorize how the values were computed. Consider the example in Figure 3.3-a, thick edges reflect the dependencies among the final values of vertices. For

[1] A vertex deletion deletes all the edges of the vertex, while deleting/inserting a vertex without any edges is usually a trivial case to compute.



(a) after inserting edge AC **(b) after removing edge AC**

Iter#	A	B	C	D	E	F	G	Frontier
init	0	5	10	20	27	32	105	{C}
5	0	5	10	20	15	17	105	{E, F}
6	0	5	10	16	15	17	105	{D}
7	0	5	10	16	15	17	105	{}

(c) re-convergence of *SSSP(A)* after inserting edge AC

Iter#	A	B	C	D	E	F	G	Frontier
reset	0	5	∞	∞	∞	∞	105	-
init	0	5	∞	20	∞	∞	105	{D}
8	0	5	25	20	27	∞	105	{C, E}
9	0	5	25	20	27	32	105	{F}
10	0	5	25	20	27	32	105	{}

(d) re-convergence of *SSSP(A)* after removing edge AC

Figure 3.3: Incremental Evaluation of Query $SSSP(A)$ (vertices in red are those affected by the deletion of edge AC).

example, the final value of D, "16", is computed based on the final value of E, "15", so D depends on E. For path-based graph queries, the dependencies form a tree rooted at the query's source vertex (more details in Section 3.3). Figure 3.3-a shows a dependency tree (thick edges) rooted at vertex A.

In general, there are three steps in handling an edge deletion.

① *Dependency Tracing.* If the deleted edge is NOT a dependent edge, then no vertices are affected; otherwise, it requires finding out the affected vertices. Consider the

graph in Figure 3.3-a, deleting edge DE has no effects on the value of any vertex. However, deleting edge AC, a dependent edge, may impact the values of vertices that depend on this edge. First, the directly impacted vertex is C. Without edge AC, C's prior value "10" is no longer valid, so do the values of other vertices that depend on C, including E, F, and D (see Figure 3.3-a). To ensure correctness, they need to be *reset* to ∞ (see Figure 3.3-d).

② *"Jump-Start"*. This step finds a *safe approximation* value (i.e., no better than the best value) for each reset vertex [34]. One way is to "pull" values from their in-neighbors and use them to update the values of these reset vertices. In Figure 3.3-d, the "**init**" row shows the initial values of reset vertices after applying a *pull* operation.

③ *Re-convergence*. The graph system then resumes the iterative evaluation until all values are re-converged. Note that, during this time, the value propagation only happens within the reset vertices because the other vertices do not depend on the deleted edge.

Among the three steps, we found that dependency tracing often dominates the execution time of edge deletion handling for the existing systems (about 70-80% for large deletion batches).

For both edge insertion and deletion handling, the correctness is ensured by the safe approximation of affected vertices' values and the monotonicity of the iterative graph algorithms [117].

Edge Weight Change Handling. Existing graph systems [117, 75, 34] treat an edge weight change as two separate updates: an edge deletion and an edge insertion, and process them in two rounds. While simplifying the design, the two-round method may incur a lot of unnecessary computations.

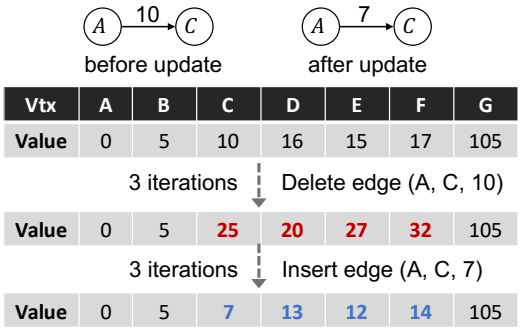


Figure 3.4: Two-Round Handling of Weight Update.

At the high level, the two-round handling always "takes a detour" to reach the final convergence. As illustrated by the example in Figure 3.4, when the weight of edge AC is changed from 10 to 7, the two-round handling first makes the values of affected vertices worse (larger values in SSSP) in the first round, then re-converges them to better values in the second round. This "detour" greatly limits the efficiency of incremental edge weight change handling.

One Strategy for All Cases. Finally, existing graph systems [117, 75, 34] uniformly employ a single processing strategy, keeping the same representation of the frontier and traversal direction regardless of the workload. Based on our observation, the lack of workload adaption often leads to sub-optimal performance.

In the upcoming sections, we address existing limitations. In Section 3.3, we introduce a new dependency tracing strategy for large deletion batches. In Section 3.4, we explore a direct method for handling weight updates. Lastly, in Section 3.5, we present an adaptive processing scheme based on update batch size.

Table 3.1: Average-case Complexities (d_{in} and d_{out} are the average in-degree and out-degree, respectively).

Representation	Children (per vtx)	Parent (per vtx)	Maint. (per vtx)	Space (all)
Sparse children vec.	$O(1)$	$O(d_{in})$	$O(d_{out})$	$O(\|V\|)$
Dense children vec.	$O(d_{out})$	$O(d_{in} * d_{out})$	$O(d_{out})$	$O(\|E\|)$
Children hashtable	$O(1)$	$O(d_{in})$	$O(d_{out})$	$O(\|V\|)$
Children in nbr. vec.	$O(1)$	$O(d_{in} * d_{out})$	$O(d_{out})$	$O(1)$
Parent vec.	$O(d_{out})$	$O(1)$	$O(d_{in})$	$O(\|V\|)$
Depen. discovery	$O(d_{out})$	$O(d_{in})$	-	-

3.3 Dependency Tracing

Dependency tracing is a key step in handling edge deletions for incremental query evaluation. It helps the graph system identify vertices affected by an edge deletion. In this section, we first discuss the data representation options for dependency, then introduce the strategies for dependency tracing, including the existing *top-down tracing* and our new design—*bottom-up tracing*.

First, we define the dependency in our context more formally.

Definition 6 (Dependency)**.** Given a graph algorithm and a graph, if the value of a vertex v_i is determined solely by the value of one of its in-neighbors v_j [2], vertex v_i depends on v_j. We refer to v_i as the dependent child while v_j as the dependent parent.

The dependency relations among vertices form a tree structure for vertex-centric queries like SSSP, while for weakly connected components (WCC), the dependencies form a forest. In the following, we refer to them as *dependency trees*.

[2]It is possible that multiple in-neighbors have the same value, but during the iterative evaluation, only one of them will be actually used to update the value of this vertex.

3.3.1 Dependency Representation

Although the design of data structure for dynamic graphs has been widely discussed, such as adjacency lists [34, 117] and tree-based representation for graph multi-versioning [27], there are limited discussion on the data representation for dependency.

Our discussion covers the lookup costs for dependent children and parent, the maintenance cost, and the space overhead. The children lookup is used for identifying affected vertices during dependency tracing, while the parent lookup could be used when changing the parent of a vertex. The maintenance is to update the dependency tree to reflect new dependencies after the graph is updated and the query is incrementally evaluated.

In total, we examined six design choices for storing dependency (assuming the dynamic graph is stored in an adjacency vector):

- *Sparse children vector* stores the dependent children of a vertex (their indices in the neighbor vector) in a vector.
- *Dense children vector* combines a boolean vector with the neighbor vector to indicate dependent children.
- *Children hashtable* stores the dependent children of a vertex in a separate hashtable (unordered set).
- *Children in neighbor vector* arranges the dependent children at the beginning of the neighbor vector, separating them from the rest neighbors with a pointer.
- *Parent vector* stores dependent parent (index in the vertex array) of each vertex in a vector.

- *Dependency discovery* detects the dependencies by checking the value relations between two neighboring vertices—an in-neighbor that determines a vertex's value is its parent.

Table 3.1 lists the complexities based on the average in-degree d_{in} and out-degree d_{out} of the graph. In general, children-based representations (except the dense children vector) offer constant access time to the dependent children, but take longer to find the parent of a vertex as they have to examine each in-neighbor of the vertex to find out if it has this vertex as a child. In comparison, parent vector offers constant access time to the parent of a vertex, but takes longer to find the children of a vertex as it needs to scan the out-neighbors of the vertex and check which one has this vertex as the parent. Dependency discovery does not explicitly store the dependency, so its maintenance and space costs are zero, but it requires extra computations to find out the dependent children.

Besides complexities, another key factor is the parallelization cost. Since a vertex may have multiple dependent children, updating the children of different vertices for children-based representations (except the dense one) requires use of locks when performed in parallel. By contrast, the parent vector and dependency discovery require no locks for parallel parent changing.

In addition, one may use two representations together to address the disadvantages of each. For example, when using children-based representations along with the parent vector, the parent lookup cost could be reduced to $O(1)$ at the cost of more memory usage.

Offline vs. Online Maintenance. The maintenance costs listed in Table 3.1 assume an offline approach—the dependency tree is updated after the re-convergence. For children-

based representations, this can be done by traversing the out-neighbors of each vertex to identify the new dependent children. Likewise, for parent vector, it needs to traverse the in-neighbors of each vertex to identify the new dependent parent. In both cases, the offline approach needs to access the graph structure which could be costly.

Alternatively, one can update the dependency tree online during the re-convergence of the affected vertices. In this case, each time the value of a vertex is updated, the parent-child dependency is also updated. For children-based representations, it involves removing a child from its old parent's children list and adding it to that of the new parent. Note that changing children for different vertices in parallel requires locks. For parent vector, the dependency update involves updating the parent of the affected vertex.

For some graph algorithms (e.g., SSSP), the value of a vertex may be updated multiple times before reaching the convergence. In this case, the online approach would involve some "unnecessary" dependency updates. On the other hand, by updating the value and dependency together, the online approach avoids the additional graph access that would otherwise occur in the offline approach.

So far, we have examined several dependency representations in terms of the benefits and costs. The best choice also depends on how dependency tracing is conducted.

3.3.2 Top-Down Dependency Tracing

Given the dependency information, the goal of dependency tracing is to find out all the vertices affected by edge deletions. Existing systems, like KickStarter [117] and RisGraph [34], follow a top-down dependency tracing strategy and use the parent vector to represent the dependency tree, as outlined in Algorithm 1.

Algorithm 1 Top-down Dependency Tracing

```
1: function DepTracingTopDown(G, parent, S_0)
2:     S = S_0
3:     Frontier_cur = S_0
4:     Frontier_next = ∅
5:     while Frontier_cur ≠ ∅ do
6:         parfor u in Frontier_cur do
7:             for v in G[u].outNeighbors do
8:                 if parent[v] == u && v ∉ S then
9:                     Frontier_next = Frontier_next ∪ {v}
10:                    S = S ∪ {v}
11:        end parfor
12:        swap(Frontier_cur, Frontier_next)
13:        Frontier_next = ∅
14:    return S
```

When edges are deleted from the graph, there is a set of vertices whose values directly impacted (see Section 3.2), referred to as *DDI vertices* (for "deletion directly impacted"), denoted as S_0. Top-down tracing starts from vertices in S_0 and traverses the dependency tree downwards till reaching leaves. At last, the algorithm outputs the visited vertices as the full set of impacted vertices S. The traversal can be expressed as a frontier-based iterative algorithm.

Figure 3.5-a illustrates the top-down dependency tracing, where $S_0 = \{B, D\}$. After the tracing, the impacted vertex set $S = \{B, C, D, E\}$.

Scalability Issues. Top-down tracing works well when the directly impacted vertex set S_0 is relatively small. When S_0 becomes larger, we observed that the performance of dependency tracing degrades badly—it can take up to 80% of the total handling time.

We found the primary factor limiting the scalability of top-down tracing is the *mismatch between top-down tree traversal and the use of a parent vector*. The former needs

the dependent children of vertices, but given the parent vector, it has to access the graph—scanning the out-neighbors of a vertex to find its dependent children (Line 7-8 in Algorithm 1), which takes $O(d_{out})$ time complexity (see Table 3.1). When there are a large number of directly impacted vertices, the cost of graph accessing becomes significant.

One way to address the above issue is to use the children-based representations outlined in Section 3.3.1, instead of the parent vector. While most of them offer constant time to access the dependent children, as discussed earlier, they have their own issues. For *sparse children vector* and *children in neighbor vector*, it is costly to remove a dependent child, which requires shifting the elements in the vector. More generally, all children-based representations take non-trivial time to maintain the parent-child relations online (see Table 3.1). In fact, most of them require locks during parallel children updates. Overall, the costs outweigh the benefits, hence the prior systems [117, 34] still chose to use the parent vector despite the mismatch.

3.3.3 Bottom-up Dependency Tracing

In this work, we explore a novel way to address the mismatch by proposing a *bottom-up* traversal strategy that aligns well with the parent vector. Moreover, it does not require accessing the graph at all. Together, they make it a promising solution for dependency tracing in handling large-scale edge deletion batches.

It is less intuitive to traverse from the leaves of a dependency tree "backward". We need to address two key questions: ① how to identify the leaf vertices of the dependency tree? and ② how to correctly find impacted vertices starting from the leaves?

① *Identifying leaves.* Leaves are vertices with no children. If we know the count of dependent children for each vertex, it becomes trivial to identify the leaves. However, to collect the dependent children count for a vertex v, we may still need to know which out-neighbors are the dependent children of v—going back to the same situation as in the top-down dependency tracing. In fact, there is a way to work around the above issue. The key is to maintain the dependent children counts of vertices incrementally, instead of computing them from scratch. To do so, we keep a copy of the old parent vector before graph updates are applied and compare it with the new parent vector afterwards. If the parent of a vertex v is changed from p_1 to p_2, we decrement the count of p_1 and increment the count of p_2. For parallel count updates, atomic operations (like `fetch_add` and `fetch_sub`) are required.

Compared to children-based representations, our solution keeps the count of dependent children instead of maintaining the list of dependent children. This difference brings a few critical advantages for an efficient implementation:

- First, the count of dependent children is a single integer that can be easily stored in a vector for all the vertices;
- Second, maintaining the counts of dependent children does NOT need to access the graph;
- Lastly, its parallelization needs atomic operations, instead of the locks required in children-based representations.

With the leaves of the dependency tree, the next question is to find all vertices impacted by the edge deletions.

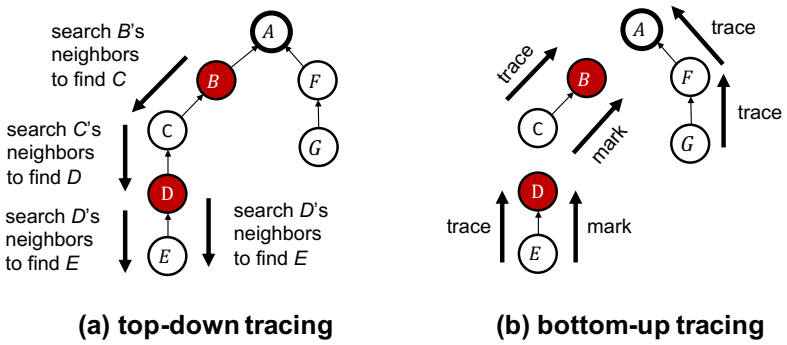


Figure 3.5: Top-Down vs. Bottom-up Dependency Tracing (solid circles are vertices directly impacted by edge deletions, i.e., S_0).

② *Finding impacted vertices.* An intuitive idea is to start from each leaf and traverse the tree bottom up until it reaches a vertex directly impacted by the edge deletions (i.e., a vertex from S_0 in Algorithm 1) or the root of the tree. However, there is a caveat to the above idea—along one bottom-up traversal path, there could be multiple vertices directly impacted by edge deletions, as illustrated in Figure 3.5-a. Stopping the traversal at the place where the first directly impacted vertex is found might miss out other impacted ones that appear even higher up in the path.

To address the above issue, we first introduce a new concept:

Definition 7 (DDI Tree)**.** A DDI tree is a maximum subtree in the dependency tree where only the root is a DDI vertex.

In Figure 3.5-a, there are two DDI trees: one with vertices B and C and the other with vertices D and E, where B and D are the only DDI vertices in each and both are the root of their subtrees.

Given a dependency tree and a set of DDI vertices (i.e., S_0), we can find all DDI trees by detaching each DDI vertex from its parent. Each detachment creates a new DDI

tree, thus, in total, there would be $|S_0|$ DDI trees created. Since only the DDI vertices are detached from the dependency tree, we have the following conclusion:

Theorem 8. After the DDI tree detachments, the remaining part of the dependency tree form a single tree.

We refer to this remaining single tree as the *reminder tree*.

Definition 9 (Reminder Tree)**.** Given a dependency tree and a set of DDI vertices, the reminder tree is the remaining part of the dependency tree after all DDI trees are detached.

Figure 3.5-b shows a reminder tree with three vertices $\{A, F, G\}$.

During the DDI tree detachments, we also update the counts of children of their parents, which may create a set of new leaves. If we start a bottom-up traversal from both the new leaves and the original leaves, denoted as $leaves_{all}$, then we can cover all the vertices in the original dependency tree, more importantly, we can tell if a vertex u is impacted by the edge deletions or not.

- If a vertex u is visited on a path that reaches a DDI vertex (i.e., part of a DDI tree), it is impacted by deletions, $u \in S$.
- Otherwise, u must be on a path to the original root of the dependency tree (i.e., part of the reminder tree), in which case it is NOT impacted by deletions, $u \notin S$.

However, at the beginning of the bottom-up traversal, it would be unknown if a traversal path will finally reach a DDI vertex in S_0 or the original tree root. Thus, it cannot decide if a visited vertex should be marked (i.e., added to S_v) or not. To address this,

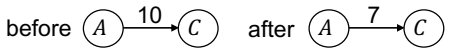
Algorithm 2 Bottom-up Dependency Tracing

```
1: function DepTracingBottomUp(parent, leaf, S_0)
2:     removeDeletedEdges(parent)
3:     updateLeaves(leaf) /* leaf is represented using a boolean array */
4:     S = S_0 /* S_0 and S are represented using boolean arrays */
5:     parfor v in leaf do
6:         p = v /* keep a copy for potential re-traversing */
7:         while hasParent(p) and p ∉ S do /* bottom-up traversal */
8:             p = parent[v]
9:         if p ∈ S then /* stopped at an impacted vertex */
10:            while hasParent(v) and v ∉ S do /* re-traverse */
11:                S = S ∪ {v} /* mark it as an impacted vertex */
12:                v = parent[v]
13:    end parfor
14:    return S
```

we first assume every path eventually reaches the original root, so no vertices are marked during this traversal. Later if the assumption fails—the traversal did encounter a DDI vertex, our algorithm would re-traverse this path from the leaf, and this time it marks the vertices visited along the path as the impacted ones (see Figure 3.5-b). More details of this bottom-up tracing are outlined in Algorithm 2.

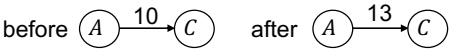
3.4 Weight Updates Handling

As explained earlier (Section 3.2), existing graph systems [117, 34, 40] simulate an edge weight update with an edge deletion followed by an edge insertion, which takes a "detour" to reach the final convergence. In this section, we show that it is possible to directly handle weight changes in a single round. We will start the discussion using SSSP, then generalize the ideas to other graph algorithms.



Iter#	A	B	C	D	E	F	G
Init	0	5	7	16	15	17	105
1	0	5	7	16	12	14	105
2	0	5	7	13	12	14	105
3	0	5	7	13	12	14	105

(a) re-convergence of *SSSP(A)* after weight decrease



Iter#	A	B	C	D	E	F	G
reset	0	5	∞	∞	∞	∞	105
init	0	5	13	20	∞	∞	105
1	0	5	13	20	18	20	105
2	0	5	13	19	18	20	105
3	0	5	13	19	18	20	105

(b) re-convergence of *SSSP(A)* after weight increase

Figure 3.6: Direct Handling of Weight Changes.

3.4.1 Case Study: SSSP

The key to effectively addressing changes in edge weights is to differentiate between "weight increases" and "weight decreases". Consider the example in Figure 3.3-a, if the weight of edge AC is decreased from 10 to 7, because AC is a dependent edge, we can easily tell that the value of vertex C should also be updated from 10 to 7. After this, we can propagate the new value of C to the other vertices in the graph by resuming the iterative evaluation from vertex C. Figure 3.6-a illustrates this process, which is similar to the handling of an edge insertion (see Figure 3.3-c).

Using the same example, but this time assume the weight of edge AC is increased from 10 to 13. Again, because AC is a dependent edge, the old value of vertex C becomes invalid. Since the edge weight was increased, the new value of C may come from another

different in-neighbor. So, we check all the in-neighbors of C to find out the new best value. However, some of its in-neighbors may also be impacted by this weight increase, as a result, their values should not be valid at the moment. In this case, to be safe, we need to first reset the value of C and the values of all the vertices depending on C to ∞. After this, for each reset vertex, we can safely "pull" the values from its in-neighbors to get a new *approximated* value. Finally, we need to resume the iterative evaluation starting from all the affected vertices. Figure 3.6-b illustrates this process, which, in fact, is similar to the 3-step handling (tracing→jump-start→re-convergence) of an edge deletion outlined in Section 3.2 (see Figure 3.3-d).

In summary, in the case of SSSP, edge weight decreases can be handled similarly to edge insertions, whereas edge weight increases can be addressed similarly to edge deletions. Also, similar to how correctness was ensured in handling edge insertions and edge deletions [117], the correctness of handling weight increases and decreases is ensured by the *safe approximation* of affected vertices' values and the *monotonicity* of the iterative SSSP algorithm.

Next, we generalize this insight to some other graph algorithms.

3.4.2 Generalization

Though the insight from SSSP may look intuitive, it is non-trivial to generalize them to other graph algorithms. To address this, we turn to a fundamental property of SSSP—*monotonicity*, a property that is well-known by the graph research community.

Theorem 10. During an iterative evaluation of query $SSSP(u)$, the value of every vertex—the shortest path distance from the source u to this vertex—never increases.

The monotonicity can be "violated" when (i) a dependent edge is deleted or (ii) the weight of a dependent edge is increased. In both cases, the old values of impacted vertices become "better" than the actual best values (under the changed graph), which is incorrect. To recover from this wrong state, we need to reset the values of all affected vertices, approximate their new initial values, and re-propagate these values until all values are converged again.

By contrast, when a new edge is inserted or the weight of an edge decreases, the monotonicity still holds—the old values remain no better than the actual best values. Thus, we can simply resume the iterative evaluation from these old values.

Besides SSSP, some other common weight-based iterative graph algorithms also exhibit monotonicity, such as Single-Source Widest Path (SSWP), Single-Source Narrowest Path (SSNP), and Viterbi.

Definition 11 (Monotonicity)**.** A weight-based iterative graph algorithm is monotonic if the value of every vertex varies in such a way that it either never decreases or never increases.

As a result, it might be possible to extend this optimization for SSSP to these algorithms as well. The pivotal question is: *for a weight-based monotonic graph algorithm, which of the two scenarios, weight increase or weight decrease, violates the monotonicity*?

To answer this question, we propose *local monotonicity test*.

Definition 12 (Local Monotonicity Test)**.** Given a weight change (increase/decrease) on edge (u, v), if (u, v) is a dependent edge, the test resets the value of the directly impacted

Algorithm 3 Local Monotonicity Test

```
1: function LocalMonoTest(u, v, w, val, parent, f)
2:     tmp_val ← val
3:     if parent[v] == u then  /* (u, v) is a dependent edge */
4:         tmp_val[v] = INIT
5:     new = f(u, v, w, tmp_val)
6:     if |INIT − new| ≥ |INIT − val[v]| then
7:         return true
8:     return false
```

Algorithm 4 Direct Handling for Weight Updates

```
1: function DirectHandling(G, src, val_old, parent_old, updates, f)
2:     (val, parent) ← (val_old, parent_old)
3:     (S_0, Frontier) ← (∅, ∅)
4:     for (u, v, w_new) in updates do
5:         Pass ← LocalMonoTest(u, v, w_new, val, parent, f)
6:         if Pass then
7:             if f(u, v, w_new, val) improves v then  /* insertion-like */
8:                 val[v] ← f(u, v, w_new, val)
9:                 Frontier ← Frontier ∪ {v}
10:        else /* deletion-like */
11:            S_0 ← S_0 ∪ {v}
12:    S ← DepTracing(G, S_0, val, parent)
13:    Pull(G, S, val, parent)  /* assign an approx. val for each vtx in S*/
14:    Frontier ← Frontier ∪ S
15:    Compute(G, val, parent, Frontier)
16:    return (val, parent)
```

vertex v to the initial value $INIT$. Then, it applies the edge function $f(\cdot)$ to u. If the new value of vertex v is **closer** to $INIT$ than the old value, the monotonicity is violated; otherwise, the change passes the test.

The ideas of this test are also summarized in Algorithm 3, which returns `true` only when the weight change conforms with the monotonicity of the given graph algorithm.

If a weight change passes the local monotonicity test, it can be handled like edge insertions; otherwise, it has to be treated like edge deletions (requiring dependency tracing).

Table 3.2: Local Monotonicity Test Results

Bench.	Edge Function $f(\cdot)$	$INIT$	Monot.	Passed
SSSP	$min(val(v), val(u) + w)$	∞	$\downarrow$	$w \downarrow$
SSWP	$max(val(v), min(val(u), w))$	0	$\uparrow$	$w \uparrow$
SSNP	$min(val(v), max(val(u), w))$	∞	$\downarrow$	$w \downarrow$
Viterbi	$max(val(v), val(u)/w)$	0	$\uparrow$	$w \downarrow$

Table 3.2 lists the results of this test to the aforementioned weight-based monotonic graph algorithms. The last column shows the passed cases.

With the help of local monotonicity testing, the idea of direct weight change handling can be generalized to all weight-based iterative graph algorithms that exhibit monotonicity. Algorithm 4 outlines the direct edge weight handling, which generalizes the handling process discussed in Section 3.4.1.

3.5 Workload-Adaptive Evaluation

Based on our observations, no single processing design works the best in all scenarios. Therefore, we found it is better to select the dependency tracing direction and the data representation based on the workload, in particular, the volume of graph updates.

3.5.1 Selection of Tracing Strategy

For small update batches, top-down tracing begins with a small set of DDI vertices (S_0). By traversing the dependency tree downwards, it tends to visit only a small portion of the dependency tree. In comparison, bottom-up tracing always starts from all the leaves

of DDI trees and the remainder tree and its traversal covers all the vertices in the original dependency tree ($O(|V|)$).

With larger update batches, top-down tracing would visit a larger portion of the dependency tree. Since its traversal depends on the graph (see Figure 3.5-a), it would also traverse a larger portion of the graph ($O(|E|)$), escalating the cost. In contrast, bottom-up tracing still performs similar traversals, except for more re-traversals due to a larger S_0 (see Lines 9-12 in Algorithm 2).

As a result, we found bottom-up tracing is more efficient when dealing with relatively large update batches, while top-down tracing is better suited for smaller batches. Based on the above insight, we design a workload-adaptive evaluation that selects the dependency tracing strategy based on the given workload.

Specifically, we define a threshold H_r for the following ratio:

$$r_{tracing} = |S_0|/|V| \tag{3.1}$$

where $|S_0|$ is the number of DDI vertices and $|V|$ is the graph size. If $r_{tracing}$ ¿ H_r, bottom-up tracing is used; otherwise, top-down tracing is employed. Based on our experimental results, we found that 0.015 is a practical value for the threshold.

We chose $|S_0|$ rather than the update batch size because the latter may not indicate workload accurately for dependency tracing. As pointed out by prior work [117, 34], most edges in the deletion batch are not dependent edges, thus carrying no computations.

Note that other factors, such as the locations of deleted edges and the degrees of impacted vertices, may also affect the performance of dependency tracing. However, it is

impractical to derive guidelines based on such fine-grained factors. In this work, we choose a simple and practically working policy (see Section 3.7 for results).

3.5.2 Selection of Data Representation

Another key design factor is the data representation. For bottom-up dependency tracing (see Algorithm 2), there are two ways to store the set of (deletions or weight increases) impacted vertices S.

- *Dense representation* which uses a boolean array whose size equals to the number of vertices in the graph $|V|$ to indicate which vertices are impacted.
- *Sparse representation* which directly stores the impacted vertices in a set (or a vector), whose size equals to the number of impacted vertices $|S|$.

Similar classifications have been used for storing the frontier (i.e., active vertices) in conventional static graph processing [106, 137, 140, 58, 141, 85]. The switching threshold is usually set empirically based on the frontier size and the number of outgoing edges [106]. In general, the sparse representation works better when the frontier size is relatively small as it is more space-efficient and requires no membership checking while the dense representation works better for relatively large frontiers as it requires no locks.

However, we are unaware of discussions on data representations for affected vertices in dynamic graph processing. For example, RisGraph chose a fixed scheme—sparse representation. However, we observe that the sparse representation does not scale well as the batch size increases. When the batch size is roughly greater than 1.5% of the total

edges, the dense representation offers better performance, thanks to its better support for concurrent updates.

Based on the aforementioned rationale and complexity of the system design, we opt to use a dense representation for bottom-up tracing and a sparse one for top-down tracing. With this fixed coupling, the graph system only needs to determine the tracing strategy, simplifying the decision making.

3.6 `IncBoost` Implementation

We implemented the above proposed ideas in a new graph system, called `IncBoost`. `IncBoost` extends `Ligra` [106]—an in-memory static graph processing framework. To support dynamic graphs, we replaced the Compressed Sparse Row (CSR) [113] format used in `Ligra` with *indexed adjacency lists* (from `RisGraph` [34]). Basically, for vertices whose degrees are greater than a threshold (set to 512), their edges are indexed by a hashtable where the key is the vertex ID of the destination and the value is the position of the vertex in the edge adjacency list. `IncBoost` provides a set of APIs for common graph updates, which include `EdgeDeletions`, `EdgeInsertions`, `WeightUpdates`, `VertexDeletions`, and `VertexInsertions`.

When inserting an edge, `IncBoost` appends the new edge to the end of the edge list and updates the index. For edge deletions, `IncBoost` first swaps the edge to delete and the last edge in the edge list, then updates the degree and index of the relevant vertex accordingly. Additionally, `IncBoost` also conducts batch insertions and deletions using *batch reordering* [11] to ensure lock-free edge mutations: edges are clustered by the edge source

upon the batch arrival, then edge mutations from the same source are applied sequentially. `RisGraph` applies all edge deletions in parallel as it does not require the swap operation but keeps tomb (deleted) edges.

`IncBoost` uses either *bottom-up* or *top-down* dependency tracing to handle edge deletions based on the workload ($r_{tracing}$ is set to 0.015). For edge weight updates, `IncBoost` handles them with the direct approach. To match `RisGraph`'s performance for very small batches, we provide a highly tuned sparse representation of the frontier. Additionally, we set a parallelism threshold (8K), below which a sequential implementation is adopted to avoid the overhead associated with parallel primitives. Regarding space usage, `IncBoost` requires an additional array of size $|V|$ for storing the dependent children counts.

3.7 Evaluation

We compare `IncBoost` with two state-of-the-art graph systems, `KickStarter` [117] and `RisGraph` [34]. For `KickStarter`, we chose its latest version with graph mutation optimizations (`DZig` [74]). Both systems were configured according to the instructions from their repositories. We also report the scalability of `IncBoost` and the detailed performance of different dependency tracing methods.

We ran experiments on a 32-core machine (CentOS 7.9) with Intel Xeon E5-2683 v4 CPU and 512GB memory. All source code were compiled with g++ 7.3. To avoid impacts of non-uniform memory access (NUMA), we only used a single socket with 16 physical cores and 32 hyper-threads.

Table 3.3: Graph Statistics ("D" for directed and "T" for temporal)

Graph	Abbr.	D	T	$\|V\|$	$\|E\|$	Avg. deg.
LiveJournal [9]	LJ	✓	✗	4.8M	69M	14.2
Orkut [63]	OR	✗	✗	3.1M	234M	76.3
Wikipedia [3]	WP	✓	✗	13.5M	437M	32.2
StackOverflow [63]	SO	✓	✓	2.6M	63.5M	24.4
Wiki-Dynamic [59]	WD	✓	✓	2.2M	43.3M	19.7
Twitter [60]	TW	✓	✗	41.7M	1.5B	35.3
UK-2007 [59]	UK	✗	✗	105.2M	3.3B	31.4
roadNet-USA [98]	RD	✗	✗	24M	58M	2.4

Table 3.4: Performance of Incremental Processing (Deletion Batch)

col.: query exec. time (in seconds); S.: small batch (1K); M.: medium batch (6%); L.: large batch (30%).

		SSSP			SSWP			SSNP			Viterbi			BFS			WCC		
		S.	M.	L.	S.	M.	L.	S.	M.	L.	S.	M.	L.	S.	M.	L.	S.	M.	L.
LJ	KickStarter	1.3E-2	0.54	1.34	1.6E-2	0.85	1.63	1.8E-2	0.75	1.63	1.4E-2	0.80	1.64	1.1E-2	0.27	0.70	1.3E-2	0.29	0.85
	RisGraph	3.0E-4	0.19	0.74	3.8E-4	0.13	0.77	3.2E-4	0.14	0.75	1.6E-4	0.13	0.80	2.5E-4	0.05	0.27	2.0E-4	0.03	0.21
	IncBoost	2.1E-4	0.09	0.21	3.6E-4	0.10	0.28	2.6E-4	0.09	0.29	2.2E-4	0.09	0.27	1.7E-4	0.04	0.11	1.8E-4	0.03	0.05
OR	KickStarter	1.1E-2	1.08	2.25	9.7E-3	0.67	1.91	1.0E-2	0.71	1.79	1.1E-2	1.74	2.70	9.5E-3	0.80	1.61	9.3E-3	0.49	1.29
	RisGraph	1.4E-4	0.24	1.12	1.5E-4	0.06	0.50	1.5E-4	0.05	0.39	1.5E-4	0.08	0.90	1.7E-4	0.08	0.36	1.2E-4	0.03	0.15
	IncBoost	1.7E-4	0.15	0.38	1.4E-4	0.03	0.36	1.5E-4	0.03	0.29	2.7E-4	0.05	0.20	1.5E-4	0.05	0.16	9.6E-5	0.02	0.05
WP	KickStarter	3.2E-2	1.59	4.02	4.2E-2	2.57	4.77	3.7E-2	1.84	4.42	3.2E-2	1.86	4.59	2.8E-2	0.98	2.81	2.7E-2	0.96	2.45
	RisGraph	2.2E-4	0.36	1.81	1.2E-4	0.24	1.65	2.3E-4	0.24	1.51	2.3E-4	0.22	1.44	1.3E-4	0.20	1.15	1.6E-4	0.08	0.41
	IncBoost	2.0E-4	0.24	0.74	1.3E-4	0.16	0.98	2.5E-4	0.17	0.60	1.7E-4	0.15	0.43	1.2E-4	0.10	0.37	1.4E-4	0.09	0.20
TW	KickStarter	5.2E-2	2.84	11.80	7.4E-2	7.54	12.29	6.9E-2	5.20	12.40	6.6E-2	3.94	11.49	7.0E-02	4.63	12.11	6.9E-2	2.51	7.86
	RisGraph	2.7E-4	0.94	9.01	1.8E-4	0.40	11.43	1.6E-4	0.37	11.41	1.7E-4	0.76	9.25	1.4E-4	1.06	7.86	1.9E-4	0.29	2.47
	IncBoost	3.5E-4	0.74	3.27	2.0E-4	0.27	4.13	1.5E-4	0.28	4.20	1.7E-4	0.51	3.41	1.5E-4	0.78	3.41	1.7E-4	0.31	1.93
UK	KickStarter	2.2E-1	16.66	34.04	4.3E-1	14.01	33.70	1.8E-1	12.56	29.74	2.8E-1	20.48	41.74	1.5E-1	7.68	13.94	1.6E-1	5.56	15.72
	RisGraph	4.5E-4	5.28	15.86	2.7E-4	0.93	9.04	3.1E-4	1.28	12.27	2.1E-4	3.45	14.60	2.6E-4	1.82	7.47	1.6E-4	0.74	4.52
	IncBoost	2.1E-4	1.87	5.81	1.2E-4	0.56	6.19	1.2E-4	0.85	4.72	1.6E-4	2.12	5.56	1.8E-4	0.87	3.24	1.4E-4	0.55	2.38
Geo	vs. KS	**158.41**×	**6.29**×	**5.34**×	**265.49**×	**18.17**×	**4.75**×	**216.10**×	**14.67**×	**5.44**×	**201.43**×	**12.49**×	**7.44**×	**206.10**×	**8.95**×	**5.99**×	**228.72**×	**12.67**×	**10.58**×
	vs. Ris	**1.16**×	**1.79**×	**2.86**×	**1.16**×	**1.55**×	**1.92**×	**1.26**×	**1.52**×	**2.29**×	**0.94**×	**1.53**×	**3.19**×	**1.19**×	**1.64**×	**2.48**×	**1.15**×	**1.20**×	**2.25**×

We used six path-based graph algorithms, including SSSP, SSWP, SSNP (single-source narrowest path), BFS, WCC (weakly connected components), and Viterbi[3]. Except for BFS and WCC, all the other algorithms operate on weighted graphs. Table 3.2 list some of the relevant properties of the first four algorithms.

We chose eight real-world graphs listed in Table 3.3. All edge weights are integers between 1 and $\log_2 |V|$. The size of the update batch varies from small (1K edges) to medium (6% of total edges) and large (15% to 40% of total edges). The updates in the batches are sampled randomly. To perform weight updates experiments, the new weights are selected

[3]The Viterbi algorithm [62] finds the most likely sequence of hidden states (i.e., the Viterbi path) in a Hidden Markov Model (HMM), which is widely used in speech recognition [91], code decoding [116], etc.

Table 3.5: Performance of Incremental Processing (Weight Update Batch)

col.: query exec. time (in seconds); S.: small batch (1K); M.: medium batch (6%); L.: large batch (30%).

		SSSP			SSWP			SSNP			Viterbi		
		S.	M.	L.	S.	M.	L.	S.	M.	L.	S.	M.	L.
LJ	KickStarter	1.70E-02	0.65	1.97	2.1E-2	1.01	2.31	2.3E-2	0.82	2.08	1.9E-2	0.97	2.21
	RisGraph	3.6E-4	0.26	1.10	4.5E-4	0.17	1.05	3.7E-4	0.17	0.94	3.9E-4	0.18	1.03
	IncBoost	2.0E-4	0.08	0.22	2.3E-4	0.08	0.27	2.1E-4	0.08	0.20	2.7E-4	0.10	0.24
OR	KickStarter	1.4E-2	1.20	2.82	1.2E-2	0.74	2.21	1.3E-2	0.76	2.06	1.4E-2	1.92	3.39
	RisGraph	1.7E-4	0.31	1.63	1.8E-4	0.09	0.65	1.8E-4	0.08	0.52	1.8E-4	0.11	1.09
	IncBoost	1.5E-4	0.09	0.31	7.1E-5	0.05	0.15	1.2E-4	0.04	0.13	1.5E-4	0.09	0.33
WP	KickStarter	4.5E-2	1.88	5.07	5.3E-2	3.16	6.93	4.9E-2	2.07	5.31	4.6E-2	2.14	5.81
	RisGraph	2.5E-4	0.52	2.67	1.6E-4	0.44	3.20	2.6E-4	0.34	2.02	2.6E-4	0.32	1.97
	IncBoost	2.0E-4	0.28	0.85	2.9E-4	0.29	1.10	1.9E-4	0.16	0.52	2.7E-4	0.17	0.68
TW	KickStarter	1.1E-1	3.96	22.54	1.1E-1	8.39	20.60	9.8E-2	5.48	20.80	1.0E-1	4.48	23.95
	RisGraph	3.1E-4	1.40	18.61	2.3E-4	0.62	15.67	1.9E-4	0.59	15.67	3.4E-4	1.18	19.00
	IncBoost	1.9E-4	0.67	4.06	7.9E-5	0.14	5.84	1.2E-4	0.25	1.25	2.0E-4	0.40	3.98
UK	KickStarter	3.2E-1	21.04	48.23	5.3E-1	16.77	44.92	3.9E-1	14.52	36.86	4.2E-1	24.27	54.65
	RisGraph	1.1E-3	4.42	25.48	8.7E-4	5.66	24.51	6.6E-4	2.29	15.61	6.6E-4	2.16	18.53
	IncBoost	1.4E-3	1.99	6.07	8.0E-4	1.48	4.38	9.8E-4	0.89	3.88	2.4E-4	1.01	5.65
Geo	vs. KS	**197.52**×	**7.36**×	**5.51**×	**299.06**×	**13.21**×	**5.50**×	**271.55**×	**13.02**×	**9.15**×	**204.31**×	**12.66**×	**6.81**×
	vs. Ris	**1.27**×	**2.51**×	**4.37**×	**1.54**×	**2.55**×	**3.73**×	**1.31**×	**2.23**×	**5.13**×	**1.51**×	**1.91**×	**3.66**×

randomly within ±50% range of the old weights. For temporal (timestamped) graphs (SO and WD), the sampled batches may contain mixed updates (insertions, deletions, and weight updates), thus we report their results for mixed batches only. We chose non-trivial sources for vertex-specific queries. The reported times are the average over three runs.

3.7.1 Performance

This section compares `IncBoost` against two existing systems in terms of processing time for update batches of three representative sizes: small (1K edges), medium (6% edges), and large (30% edges), with a focus on edge deletions and weight change updates.

Edge Insertions. As discussed in Section 3.2.1, edge insertion batch is an easier case to process and `IncBoost` achieves comparable performance as `RisGraph` for small batches but scales better by switching to the dense representation (for both dependency tracing and iterative evaluation). On average, the speedups `IncBoost` over `RisGraph` are 0.98×,

Table 3.6: Performance of Incremental Processing (Mixed Updates Batch)

col.: query exec. time (in seconds); S.: small batch (1K); M.: medium batch (6%); L.: large batch (30%).

		SSSP			SSWP			SSNP			Viterbi		
		S.	M.	L.	S.	M.	L.	S.	M.	L.	S.	M.	L.
LJ	KickStarter	2.3E-02	0.41	1.04	9.5E-03	0.49	1.11	7.7E-03	0.15	0.77	8.6E-03	0.16	0.98
	RisGraph	3.6E-04	0.15	0.60	3.5E-04	0.12	0.46	3.6E-04	0.11	0.43	3.9E-04	0.12	0.45
	IncBoost	2.9E-04	0.08	0.27	2.7E-04	0.07	0.22	2.1E-04	0.07	0.23	2.5E-04	0.07	0.21
TW	KickStarter	9.8E-02	2.74	10.39	5.1E-02	4.19	10.30	5.1E-02	2.51	9.43	9.8E-02	2.85	4.49
	RisGraph	3.1E-04	1.00	5.31	2.3E-04	0.38	1.64	1.9E-04	0.38	1.51	2.4E-04	0.84	3.02
	IncBoost	2.3E-04	0.66	2.60	1.3E-04	0.30	1.02	1.1E-04	0.32	0.99	1.5E-04	0.52	1.72
SO	KickStarter	2.0E-01	0.73	1.04	2.0E-01	0.91	1.18	1.9E-01	0.84	1.11	3.3E-02	0.66	1.14
	RisGraph	2.7E-04	0.18	1.56	2.9E-04	0.06	0.59	2.8E-04	0.06	0.28	3.6E-04	0.08	0.27
	IncBoost	2.9E-04	0.10	0.31	2.6E-04	0.04	0.18	2.6E-04	0.04	0.17	3.3E-04	0.04	0.14
WD	KickStarter	2.2E-02	0.49	1.01	3.5E-02	0.55	1.27	2.6E-02	0.69	1.16	2.2E-02	0.73	1.39
	RisGraph	8.3E-04	0.19	1.80	4.3E-04	0.10	0.39	5.0E-04	0.07	0.39	4.8E-04	0.10	0.38
	IncBoost	9.0E-04	0.09	0.33	4.7E-04	0.05	0.16	3.7E-04	0.04	0.17	3.4E-04	0.03	0.14
Geo	vs. KS	**153.98**×	**5.31**×	**3.56**×	**168.34**×	**12.08**×	**7.21**×	**171.45**×	**8.86**×	**6.16**×	**108.87**×	**8.35**×	**5.67**×
	vs. Ris	**1.10**×	**1.75**×	**3.35**×	**1.23**×	**1.57**×	**2.30**×	**1.43**×	**1.46**×	**1.81**×	**1.39**×	**2.00**×	**2.13**×

1.30×, and 2.0× for small, medium, and large batches, respectively. More results on the performance of edge insertions can be found in Section 3.7.2.

Edge Deletions. Table 3.4 shows the incremental computation time for handling deletion batches. Thanks to its workload-adaptive evaluation, `IncBoost` evaluated all small batches using *top-down* tracing with a *sparse representation*, and medium and large batches using the *bottom-up* tracing with the *dense representation*.

For small batches, `IncBoost` exhibits similar performance to `RisGraph`, with speedups from 0.94× to 1.26×. For medium and large batches, the speedups of `IncBoost` over `RisGraph` become more significant, ranging from 1.20× to 1.79× and from 1.92× to 3.19×, respectively, thanks to its use of the bottom-up dependency tracing. `KickStarter` is the least competitive among the three systems for all three batch sizes. Note that, for smaller batches, `KickStarter` shows even worse performance, mainly because of its use of the dense data representation for dependency tracing and iterative evaluation. As discussed earlier, for smaller update batches, the sparse data representation offers better efficiency.

Weight Updates. The handling time for edge weight updates is presented in Table 3.5,

which covers four algorithms for weighted graphs (SSSP, SSWP, SSNP, and Viterbi). On average, we found `IncBoost` is 2.6× (1.3×-5.1×) faster than `RisGraph` and 87.0× (6.8×-299×) faster than `KickStarter`. In general, the speedups follow the same trends as those in the edge deletion case.

The primary reason for the speedups of `IncBoost` is its adoption of a direct approach to handling weight changes, rather than the two-round approach used by the existing graph systems. To show a more direct comparison, we also implemented the two-round approach in `IncBoost`, which we denote as `IB-2R`. Table 3.7 reports the detailed profiling results on batches of medium size (6% batch), including the ratio of directly impacted vertices ($|S_0|/|V|$), the ratio of all impacted vertices ($|S|/|V|$), the number of vertex activations during iterative computation (Tot. Act.), the tracing time (Tr. Time), and the iterative computation time (Iter. Time). For `IB-2R`, the "Iter. Time" is the sum of iterative computation times for both deletion and insertion rounds.

The results show that with direct weight update handling, the number of vertices requiring dependency tracing ($|S_0|$) is reduced by half compared to the two-round handling. This is because the update batch consists of an equal distribution of weight increases and decreases (50%-50%), and the direct approach treats the two cases separately, one for each round. For the same reasons, the total number of vertex activations and iterative computation time are also significantly reduced accordingly.

Mixed Batches. `IncBoost` is capable of handling heterogeneous batches that contain mixed types of updates: edge insertions, edge deletions, and edge weight updates. `RisGraph` only supports batches containing a single type of updates (homogeneous update batches).

Table 3.7: Profiling of Direct and Two-round on SSSP.

		$\|S_0\|/\|V\|$	$\|S\|/\|V\|$	Tot. Act.	Tr. Time	Iter. Time
LJ	IB-2R	5.33%	29.60%	3.6E+06	0.054	0.086
	IncBoost	2.73%	13.20%	1.6E+06	0.037	0.043
WP	IB-2R	3.37%	15.09%	4.6E+06	0.144	0.305
	IncBoost	1.78%	7.77%	2.4E+06	0.104	0.167
TW	IB-2R	4.85%	15.05%	1.1E+07	0.456	0.646
	IncBoost	2.42%	7.10%	5.2E+06	0.317	0.352

KickStarter supports batches of mixed insertions and deletions, but could not handle batches which have insertions and deletions of the same edge. Fortunately both systems can preprocess mixed batches into homogeneous sub-batches.

Table 3.6 presents the performance of evaluating mixed batches where the batch is configured as containing 50% edge insertions and 50% edge deletions when the graph is non-temporal. For temporal graphs that have timestamps associated with each edge, we delete edges with older timestamps and insert newer ones. If an edge is inserted more than once in a batch, it is considered as a weight update. Under this setting, a temporal graph update batch contains mixed updates. For example, a 30% WD batch may contain 33% insertions, 51% deletions, and 16% weight updates. Generally, we observed that the speedups fall between those achieved in pure weight updates and pure deletion batches.

Table 3.8: Graph Mutation Throughput (edges/sec, TW Graph)

	Edge updates per second		
	Insertions	Deletions	Weight Updates
RisGraph	1.1E+07	1.2E+07	5.8E+06
Aspen	3.9E+07	3.7E+07	Not Suported
IncBoost	1.4E+07	1.3E+07	1.5E+07

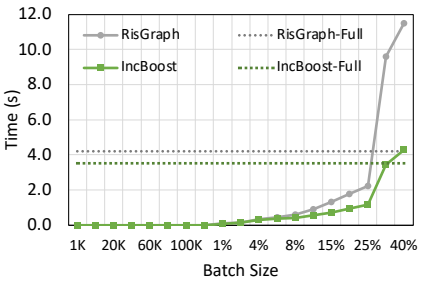
Graph Mutations. For completeness, we briefly compare the cost of graph mutations with `RisGraph` and `Aspen` [27]. Table 3.8 reports the throughput of the edge-related updates. Note that the released version of Aspen only supports unweighted and undirected graphs. While `IncBoost` and `RisGraph` exhibit comparable throughput for edge insertions and deletions, both show significantly lower throughput compared to `Aspen`, mainly due to `Aspen`'s utilization of a compressed tree data structure for the graph. For weight changes, `IncBoost` achieves a throughput roughly 2.6× higher than that of `RisGraph`, thanks to its avoidance of the two-around handling.

Performance on Road Networks. We evaluated `IncBoost` on a non-power-law graph: roadNet-USA [98]. Unlike power-law graphs, road networks often have a higher vertex-to-edge ratio and their edges are distributed more evenly across vertices. These properties lead to a high ratio of dependent edges. As a result, graph updates tend to affect a larger portion of the vertices.

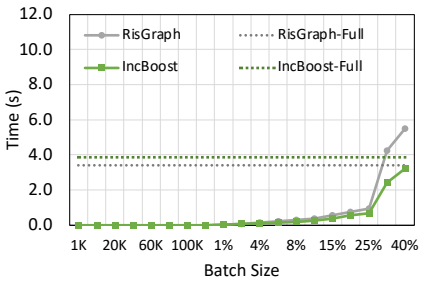
Our results show that `IncBoost` and `RisGraph` are capable of handling up to 80K edge insertions and 20K edge deletions for SSSP before becoming slower than the full evaluation. To put it in the context, 20K edge deletions (0.03% of the total edges) affect 73% vertices in roadNet-USA.

3.7.2 Workload Scalability

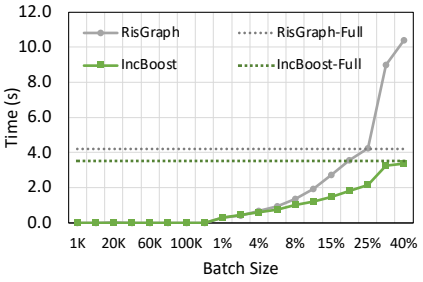
Figure 3.7 reports processing time of `IncBoost` and `RisGraph` for different batch sizes. The horizontal dotted lines indicate the full query evaluation times. For all three



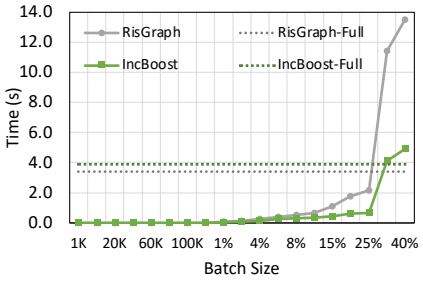
(a) SSSP Edge Insertions



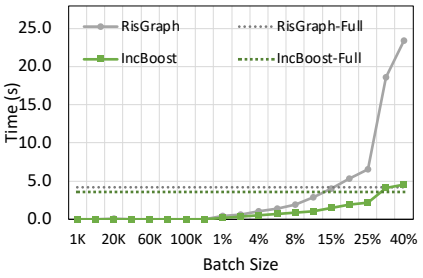
(b) SSWP Edge Insertions



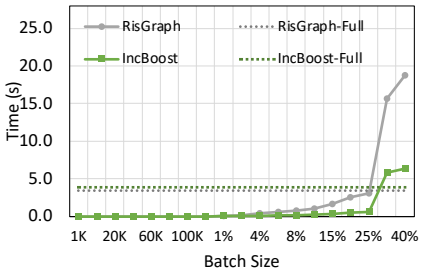
(c) SSSP Edge Deletions



(d) SSWP Edge Deletions



(e) SSSP Weight Updates



(f) SSWP Weight Updates

Figure 3.7: Scalability with Varying Batch Sizes on TW Graph.

batch types (edge insertions, deletions, and weight updates), both systems deliver notably fast incremental evaluation for relatively small batches (below 500K).

As update batch size increases, the two systems scale well on edge insertions. However, for edge deletions and weight updates, `IncBoost` scales much better than `RisGraph` and its incremental computation remains faster than the full query evaluation even for large

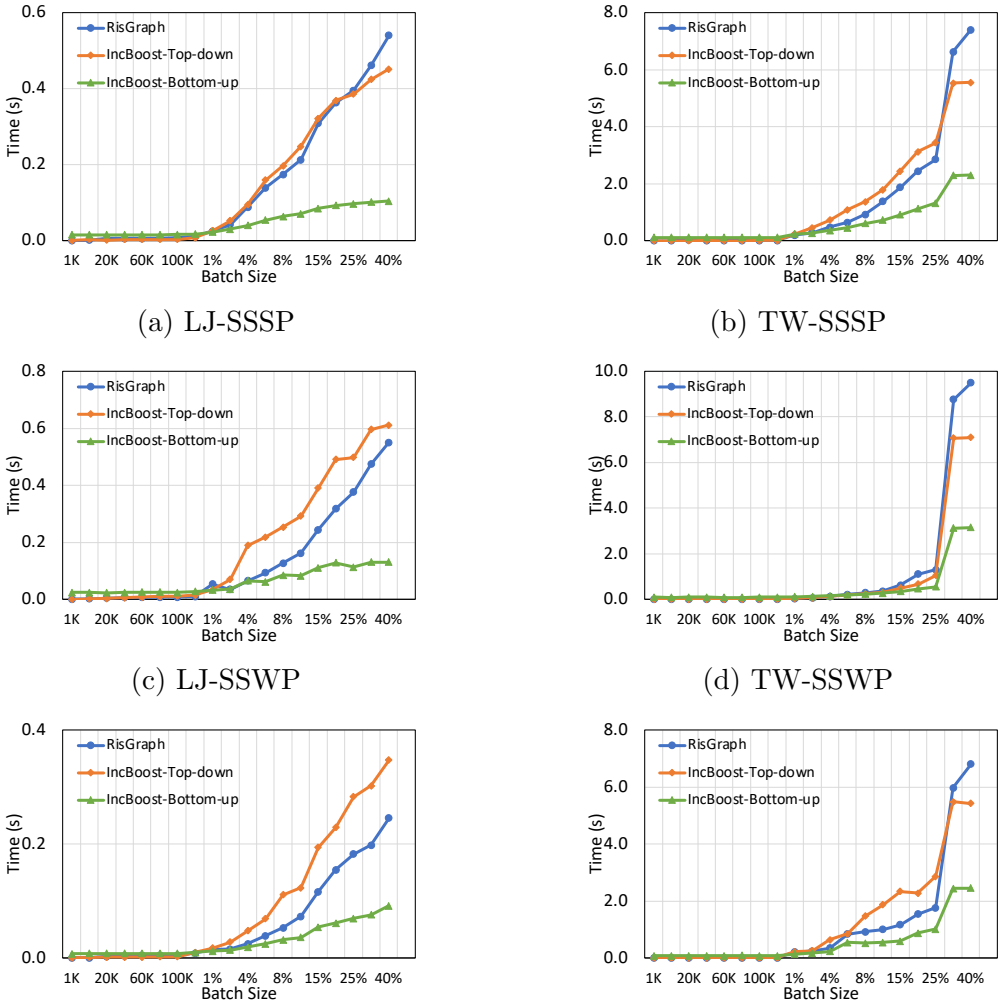
batch sizes (30% - 40%), while `RisGraph` struggles to yield performance benefits when the batch size gets close to 20% for SSSP edge deletions and 15% for SSSP weight updates.

Although `RisGraph` exhibits super linearity when evaluating batches with sizes from 6% to 30%, it does not imply the system can be more efficient if breaking the larger batch down into several small batches. Evaluating multiple (different) sub-batches is more costly than evaluating one big batch since the former can cause a lot of affected vertices to converge to unnecessary states, incurring even more vertex activations. In addition, a large batch can potentially cause a large portion of vertices to be affected by edge deletions, which are non-linear to the batch size.

3.7.3 Dependency Tracing

Figure 3.8 presents the costs of dependency tracing under different configurations of `IncBoost` and `RisGraph`. In general, we found the performance trend of `IncBoost-Top-down` closely aligns with that of `RisGraph`, since both use the top-down tracing.

For update batches with sizes less than 1% of the graph, we noticed `IncBoost-Bottom-up` performs the poorest. However, as the batch size approaches approximately 1.5% of the total graph size, a significant shift occurs—`IncBoost-Bottom-up` transforms into the fastest method. These results validate the necessity of workload-adaptive evaluation (see Section 3.5).



(a) LJ-SSSP (b) TW-SSSP

(c) LJ-SSWP (d) TW-SSWP

(e) LJ-BFS (f) TW-BFS

Figure 3.8: Dependency Tracing Performance.

3.7.4 Bottom-up Tracing in a Distributed System

`IncBoost` is implemented as a shared memory graph processing system. However, adapting the ideas of `IncBoost` to a distributed environment does not require algorithmic changes. In fact, thanks to its avoidance of graph access, the bottom-up dependency tracing can be efficiently performed on a single node.

With a moderate level of effort, we have adopted the idea of bottom-up dependency tracing into a state-of-the-art distributed graph processing system `Gemini` [140], which clearly demonstrates the applicability of our techniques in the distributed setting.

`Gemini` uses the master-mirror notion to partition and distribute vertices across nodes. Every active vertex broadcasts its vertex value as well as the parent information from the master to its mirrors. This introduces extra communication overhead for the top-down dependency tracing as it may traverse several partitions of the graph on different nodes. The bottom-up tracing saves the graph traversal and communication overhead by performing the dependency tracing on a single node and then broadcasting the parent array to other nodes in the end.

Table 3.9 reports the costs of dependency tracing in `Gemini` for SSSP on TW graph. The results cover three representative batch sizes (1k, 6%, and 30%) to test the top-down and bottom-up tracing performance. For larger batches, the bottom-up tracing in `Gemini` delivers more performance improvements (4.7× to 7.1× speedup over the top-down one) than it does in a shared memory environment thanks to the additional savings in communication cost. On the other hand, the top-down tracing remains outperforming the bottom-up one when the batch size is small.

Table 3.9: Dependency Tracing Time in Gemini (seconds)

	1K	6%	30%
Top-down	0.03	0.47	1.27
Bottom-up	0.07	0.10	0.18

3.8 Summary

This work targets the scalability limitations in handling edge deletions and weight updates for incremental graph query evaluation. For edge deletions, it introduces a bottom-up dependency tracing strategy, along with a workload-adaptive evaluation scheme that changes the tracing strategy based on the update volume. For weight changes, it presents a direct approach to handle the weight changes, instead of simulating them with edge insertions and deletions. It is general enough to cover a group of weight-based monotonic graph algorithms. Finally, it demonstrates the effectiveness of the proposed ideas with a new graph system `IncBoost`. Our results show that `IncBoost` is able to scale to very large update batches with sizes of 30% to 60% of the graph size.

Chapter 4

Taming Misaligned Graph Traversals in Concurrent Graph Processing

4.1 Introduction

Although the last decade witnessed significant advances in developing efficient graph processing systems, support for concurrent query evaluation remains underexplored. Most existing graph processing systems are designed to process one analytical query each time, such as a single-source shortest path (SSSP) query. On the other hand, as the demands of graph analytics grow, so do the needs for concurrent evaluation of graph queries [129, 134, 139]. A prior study on social network applications shows that most graph query jobs are executed concurrently [129]. To fill this gap, several concurrent graph processing sys-

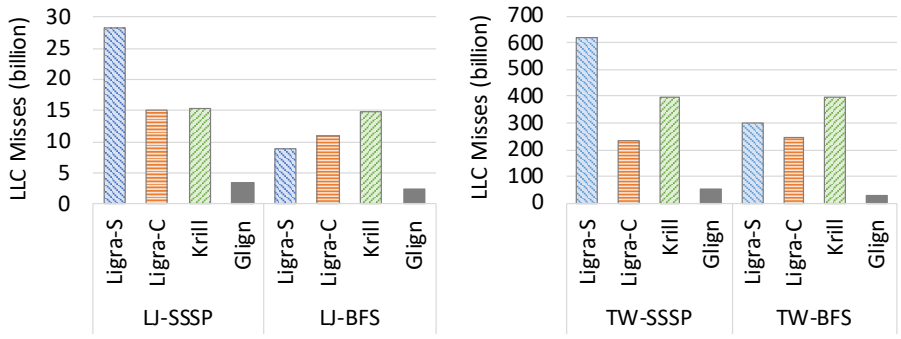


Figure 4.1: Last-Level Cache Misses (64 concurrent queries on LiveJournal [9] and Twitter [60], measured by `perf` profiler).

tems [88, 129, 134, 139, 18] have been proposed in recent years, including `Seraph` [129] for distributed platforms, `CGraph` [134] and `GraphM` [139] with supports for out-of-core processing, and `Congra` [88] and `Krill` [18] which focus on in-memory evaluation of a batch of concurrent graph queries.

Opportunities and Challenges. By evaluating multiple queries simultaneously on a graph, concurrent graph processing enables graph access sharing across queries via the memory hierarchy, that is, the graph data fetched to the cache(s) by one query may be used directly by other queries. Intuitively, such sharing may reduce the total number of cache misses, benefiting overall performance. However, this work finds that the actual cache miss reduction brought by concurrent graph query evaluation could be quite limited.

Figure 4.1 reports the last-level cache (LLC) misses of evaluating 64 concurrent queries on two graphs using some representative graph systems. As a baseline, `Ligra-S` evaluates the queries one by one using `Ligra` [106], a well-known in-memory graph processing framework that evaluates each query in parallel. In comparison, `Ligra-C` evaluates all 64 queries simultaneously using an extended `Ligra` with basic concurrency supports (see

Section 4.4); `Krill` is a state-of-the-art concurrent graph processing system just released recently [18]. As the results show, even with a concurrency degree of 64 queries, the cache misses of `Ligra-C` and `Krill` are reduced by a limited fraction compared to `Ligra-S`, and sometimes, their cache misses may even exceed that of the baseline (LJ-BFS and TW-BFS).

A primary reason causing the above unfavorable results lies in *the potential "misalignment" of underlying graph traversals among concurrent queries*. Though the above 64 queries are of the same type, they may traverse the graph very differently due to their vertex-specific nature (i.e., starting from different source vertices). For queries of different types, their underlying graph traversals can be even more diverse. When the traversals are misaligned—visiting different parts of the graph for most time of the processing, the concurrent evaluation of queries will not benefit much from the shared memory accesses. Even worse, they may even "hurt" each other by competing for the caches.

Solution of This Work. To address the above issue, this work proposes a runtime system for in-memory graph processing on multi-core platforms [1], namely, `Glign` [2]. `Glign` can automatically align different graph traversals of concurrent queries to maximize graph access sharing. As a result, it can significantly reduce cache misses compared to other systems (see Figure 4.1). `Glign` primarily targets vertex-specific queries that employ iterative graph algorithms for evaluation, such as SSSP and BFS. In addition, to benefit the most from `Glign`, the vertex function of the iterative algorithms $f(v)$ needs to be monotonic, a common property shared by many vertex-centric graph query algorithms [117, 56, 102]. Next, we briefly introduce the key techniques behind `Glign`.

[1] Similar ideas could be applied to out-of-core and distributed processing scenarios.
[2] Pronounced as /gline/.

First, like most existing concurrent graph processing systems [134, 139, 18], `Glign` synchronizes the iterations of different queries during evaluation—the barriers used for iterative evaluation are shared across queries. This design allows `Glign` to treat the iterations as a logical timeline for aligning graph traversals. To distinguish them from the iterations in single-query graph processing, we refer to the iterations shared by queries as *global iterations*.

Based on the global iterations, `Glign` addresses the problem of graph traversal misalignment at three levels:

- *Intra-iteration alignment.* In each iteration of the evaluation, a query needs to access an active part of the graph (a.k.a. *frontier*). Intuitively, the active parts of different queries may overlap. If the overlapped parts are accessed around the same time, the evaluation will benefit from temporal locality.

- *Inter-iteration alignment.* For a given batch of queries, `Glign` allows their evaluation to start at *different* global iterations, thus making it possible to align the iterations across queries based on their graph access sharing.

- *Alignment-aware batching.* At the high level, considering all the concurrent queries available, which queries should be put into the same evaluation batch? Different batching strategies may yield different amounts of graph access sharing.

For intra-iteration alignment, existing designs [18, 125] require two levels of frontiers to achieve synchronized frontier traversal. Instead, `Glign` proposes *query-oblivious frontier*, a single-level frontier that deliberately ignores the frontier differences across queries. This is possible if the vertex function of the query is monotonic. On the other hand, this

may evaluate extra vertices due to its inability to distinguish some inactive ones for certain queries. Overall, we found the benefits of reduced memory (from the use of a single-level frontier) easily outweigh the side effects of extra computations.

For inter-iteration alignment and alignment-aware batching, `Glign` leverages an important insight revealed in this work:

the "heavy iterations" of concurrent queries

should be well aligned during the evaluation.

Here, "heavy iterations" refer to iterations that access a relatively larger portion of the graph (i.e., a large frontier). The insight is backed by two facts. First, heavy iterations often dominate the total processing cost of a query; Second, larger frontiers often expose more opportunities for intra-iteration alignments—a potentially larger overlapping among the frontiers of different queries.

The above insight reduces the two higher-level alignments into the *alignment of heavy iterations*. To solve the latter, this work uses a simple yet effective heuristic to estimate the arrival time of heavy iterations. Based on the estimation, two scheduling techniques are proposed to improve the alignments:

- *Delayed start.* For a given batch of concurrent queries, this technique postpones the start of the evaluation of certain queries to later global iterations, based on the arrival time differences of their heavy iterations;

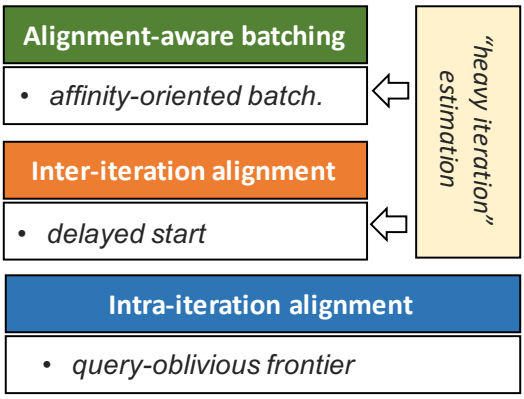


Figure 4.2: Three Levels of Alignments from `Glign`.

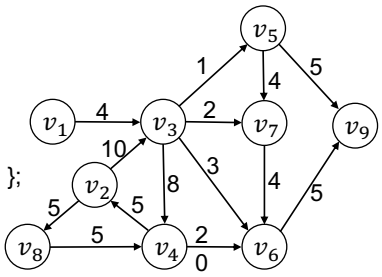
- *Affinity-oriented batching.* Considering all the concurrent queries received, it groups queries with closer arrival times of heavy iterations (affinity) to the same evaluation batch.

Figure 4.2 lists the above techniques. To confirm their effectiveness, this work evaluated `Glign` with commonly used graphs and query benchmarks, and compared it with two state-of-the-art concurrent graph systems: `GraphM` [139] and `Krill` [18]. The results show that the proposed alignment techniques can reduce the LLC misses by a significant ratio. They also show that `Glign` achieves on average 3.6× speedup over `Krill` and 4.7× speedup over `GraphM`.

In summary, this work makes a three-fold contribution:

- First, it reveals a key performance issue in concurrent graph processing—graph traversal misalignments, and categorizes it at three levels of the graph processing system.
- Second, it proposes a series of techniques to address the misalignments at each level: a new design of synchronized frontier traversal and two scheduling techniques based on the insight of heavy iterations.

```
/* single-source shortest path */
f_sssp(v) {
  for each out-neighbor n of v {
    dist(d) = min { dist(n), dist(v) + w(v, n) };
    if dist(n) changed
      then add n to frontier;
}
```



(a) (b)

Figure 4.3: Example Vertex Function and Graph.

- Finally, it systematically evaluates the above techniques and compares `Glign` with the state-of-the-art systems.

4.2 Background

Table 4.1: Iterative Evaluation of sssp(v_1)

Iter#	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8	v_9	Frontier
0	0	∞	∞	∞	∞	∞	∞	∞	∞	$\{v_1\}$
1	0	∞	4	∞	∞	∞	∞	∞	∞	$\{v_3\}$
2	0	∞	4	12	5	7	6	∞	∞	$\{v_4, v_5, v_6, v_7\}$
3	0	17	4	12	5	7	6	∞	10	$\{v_2, v_9\}$
4	0	17	4	12	5	7	6	22	10	$\{v_8\}$
5	0	17	4	12	5	7	6	22	10	$\{\}$

4.2.1 Concurrent Evaluation of Graph Queries

Recently, several graph processing systems have been proposed to support concurrent graph query evaluation, such as `Seraph` [129], `CGraph` [134], `GraphM` [139], `Congra` [88], and `Krill` [18]. Take the more recent system `Krill` as an example. `Krill` is built on top of

`Ligra` [106], a state-of-the-art in-memory graph processing system. Under the hood, `Ligra` exploits the vertex-level parallelism where the vertex function is executed on the active vertices (in the frontier) in parallel, guided by a work stealing scheduler (from Cilk [14]). So even for a single query, the system can evaluate it in parallel with relatively balanced workload across CPU cores.

Though the designs and targeted platforms may vary, the above mentioned systems all support simultaneous evaluation of multiple graph queries on a given graph. The basic idea is to put each K queries into an evaluation batch $\mathcal{B}$, meanwhile maintain a separate vertex value array and frontier for each query in $\mathcal{B}$. The evaluation stops when all frontiers in $\mathcal{B}$ become empty. For in-memory graph processing, K is bounded by the memory capacity for the space costs of vertex value arrays (and others).

Table 4.2: Graph Access Sharing between Two Queries.

Iter#	Frontier(sssp(v_2))	Frontier(sssp(v_8))
0	$\{v_2\}$	$\{v_8\}$
1	$\{v_3, v_8\}$	$\{v_4\}$
2	$\{v_4, v_5, v_6, v_7\}$	$\{v_2, v_6\}$
3	$\{v_9\}$	$\{v_3, v_9\}$
4	$\{\}$	$\{v_5, v_6, v_7\}$
5	$\{\}$	$\{v_9\}$
6	$\{\}$	$\{\}$

A key potential benefit of concurrent graph query evaluation is that the sharing of graph accesses via the memory hierarchy, which improves the overall data locality. Consider two queries, sssp(v_2) and sssp(v_8), to the graph in Figure 4.3-(b). In fact, both queries need to access the out-neighbors of vertices v_2 - v_9 during the evaluation, as dictated by their frontiers in Figure 4.2. If the graph data fetched by one query still resides in the shared

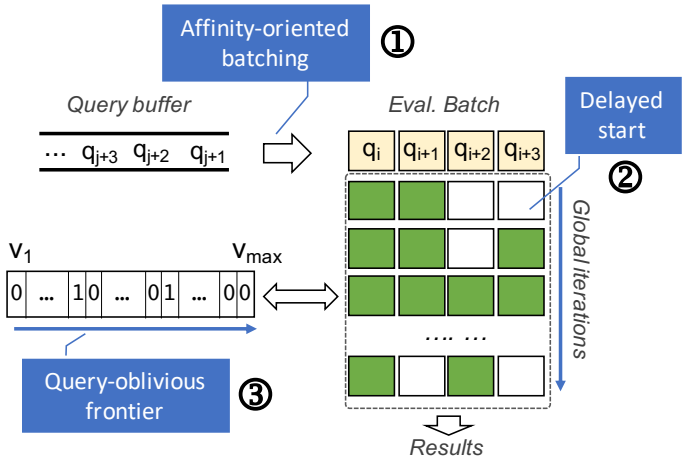


Figure 4.4: Overview of `Glign`.

cache when the other query tries to access it (i.e., temporal locality), the overall cache misses could be dramatically reduced. However, for many real-world graphs, their sizes are well beyond the cache capacity. In order to benefit from this temporal locality, the graph traversals should be roughly *aligned*—visiting the same vertices (and their out-neighbors) around the same time.

In fact, as reported earlier in Figure 4.1, the underlying graph traversals on real-world graphs could be largely misaligned in the existing concurrent graph processing systems, limiting the benefits of shared graph accesses. In the following, we will present a solution to addressing the graph traversal misalignment issue—`Glign`.

4.3 Glign Design

Figure 4.4 illustrates the high-level ideas of `Glign`. First, considering all the concurrent queries available in the buffer, `Glign` uses an *affinity-oriented batching* strategy to group queries with higher potential of large amount of graph access sharing to the same

evaluation batch (see ①). After a batch is formed, `Glign` estimates the "heavy iterations" for each query in the batch. Based on their differences in arriving times (global iterations), `Glign` delays the start of certain queries in order to align the "heavy iterations" of different queries (see ②). Finally, within each global iteration, `Glign` traverses a single unified frontier, called *query-oblivious frontier*, to make sure that all the shared graph accesses (dictated by the overlapping of frontiers) are accessed in a fully coalesced manner (see ③). Next, we will present each of these key techniques in detail. Due to their dependences, we will introduce them in reverse order with respect to the number labels in Figure 4.4.

4.3.1 Global Iterations

First, we introduce the concept of *global iterations*, which serve as the basis for some of the proposed alignments. Given a batch of iterative graph queries, there are two ways to evaluate them:

- *Synchronous evaluation* evaluates queries in the batch in the same pace with respect to iterations (see Figure 4.4). This is ensured by a series of global barriers that are shared across queries in the batch. Most existing concurrent graph systems (`CGraph` [134], `GraphM` [139] and `Krill` [18]) follow this scheme.

- *Asynchronous evaluation* evaluates each query in the batch independently, regardless of the evaluation pace of other queries, that is, the iterations of evaluating different queries may be interleaved arbitrarily. `Congra` [88] uses this scheme.

Clearly, the asynchronous design has no control over the graph traversals, so the traversals may or may not align well depending on their interleaving in a specific evaluation.

For this reason, `Glign` follows the synchronous evaluation. To distinguish the iterations in the synchronous batch evaluation from those in single-query evaluation, we refer to the former as *global iterations*.

4.3.2 Intra-Iteration Alignment

A commonly used design for evaluating concurrent graph queries is to keep a frontier for each query q_i in the evaluation batch $\mathcal{B}$. In a global iteration, the frontiers of different queries are traversed independently, as shown in Figure 4.5-(a). The frontier is designed as a boolean array `frontier[]`, where the i-th element shows the activeness of vertex v_i. If `frontier[i]=1`, the vertex function $f(v)$ needs to be evaluated on v_i, including accessing the out-neighbors of v_i. In another word, the frontier traversal defines how graph data is accessed in a global iteration. When these frontiers of different queries are traversed independently, there is no guarantee that the commonly used graph data are accessed around the same time. As a result, the data locality could become sub-optimal.

To ensure that different frontiers are traversed in a synchronized manner, some recent works (`Krill` [18] and `SimGQ` [125]) propose to add an extra frontier, called *unified frontier*, defined as follows:

$$Frontier_{union} = \bigvee_{q_i \in \mathcal{B}} Frontier_{q_i} \tag{4.1}$$

where $Frontier_{q_i}$ (a boolean array) is the frontier for evaluating query q_i and $\bigvee$ is the logical OR operator. This means that as long as vertex v_i is active for one query in the batch, $Frontier_{union}(i) = 1$. To synchronize the frontier traversals, we can simply traverse the unified frontier: if its value for vertex v_i is "1", we further check each individual frontier

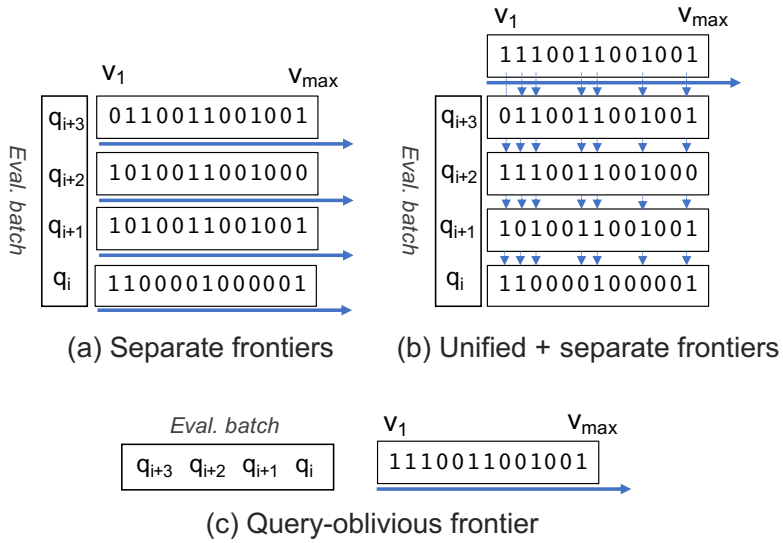


Figure 4.5: Different Designs of Frontier Traversal.

$Frontier_{q_i}$ to find out the specific queries for which v_i needs to be evaluated (see Figure 4.5-(b)).

The above design ensures that the shared accesses to an active vertex and its out-neighbors are perfectly aligned across queries. However, there are some caveats associated with this design. First, it increases the memory cost with an extra labeling array $Frontier_{union}$; Second, it needs to check the frontiers at two levels. Overall, our evaluation reports limited performance benefits (see Section 4.4).

To avoid the above caveats, this work proposes an alternative design to the synchronized frontier traversal, called *query-oblivious frontier*. This new design explores an interesting tradeoff between computations and memory accesses, which to our best knowledge, has not yet been discussed before by any prior work.

Query-Oblivious Frontier. Our key insight is *to deliberately ignore the differences among the frontiers of queries in the evaluation batch*, that is, when a vertex function $f(v)$ is invoked, it is applied for all queries in $\mathcal{B}$. This eliminates the need of second-level frontiers ($Frontier_{q_i}$, q_i in $\mathcal{B}$) used in the prior design. Figure 4.5-(c) illustrates this idea with a single frontier $Frontier_{union}$.

However, the above design immediately raises two concerns:

- *Correctness.* Does the evaluation based on a single unified frontier ($Frontier_{union}$) always produce the same results as the one using two levels of (or seperate) frontiers?
- *Efficiency.* A vertex v that is not in the frontier of query q_i would be evaluated anyway, if v is in the frontier of some other query in the batch. This introduces extra computations.

First, for correctness, we have established a theorem for safely adopting query-oblivious frontier for a range of iterative queries based on the *monotonicity* property of their vertex functions.

Definition 13. In vertex-centric programming, a vertex function $f(\cdot)$ is **monotonic** iff. it always changes the values of vertices monotonically (always increasing or decreasing) over iterations.

In fact, the monotonicity property has been widely exploited by multiple existing graph systems for better efficiency [66, 102] and it serves as the basis for incremental query evaluation [117, 56].

Theorem 14. Evaluating a query batch using query-oblivious frontier yields the same vertex values as the evaluation using separate frontiers iff. the vertex function is monotonic.

Proof Sketch. Without loss of generality, assume the evaluation batch consists of two queries q_1 and q_2, and the evaluation is in global iteration i. Consider an arbitrary vertex v_j. Assume v_j is inactive according to q_1's frontier, but active based on q_2's frontier. Thus, v_j is marked as an active one in the unified frontier, as illustrated in Figure 4.6. Next, we discuss the impacts of an extra evaluation of v_j with q_1's vertex function, that is, $f_{q_1}(v_j)$. There are two basic cases. In the first case, at the time of evaluating $f_{q_1}(v_j)$, the value of v_j has not been updated by any of its in-neighbors. As its value remains the same as it was in the prior iteration $i-1$, this evaluation will not change the value of any of its out-neighbors (like v_t). In the second case, at the time of evaluating $f_{q_1}(v_j)$, the value of v_j has been updated by at least one of its in-neighbors in the current iteration i. In this case, the evaluation may update the value(s) of some out-neighbor(s) of v_j, causing some side-effects. However, note that even if separate frontiers are used, v_j would be marked as an active vertex and evaluated in the next iteration $i+1$. That is, using query-oblivious frontier might lead to some earlier evaluation of certain vertices that are supposed to be evaluated in the next iteration— a form of asynchronous evaluation. One sufficient for the correctness of asynchronous query evaluation is that the query evaluation should be monotonic. □

Second, as to the efficiency concern, will the extra evaluation of inactive vertices slow down the overall processing? Interestingly, our evaluation (see Section 4.4) shows that using query-oblivious frontier can substantially improve the overall performance despite the extra evaluation of inactive vertices. This is due to fact that query-oblivious frontier skips

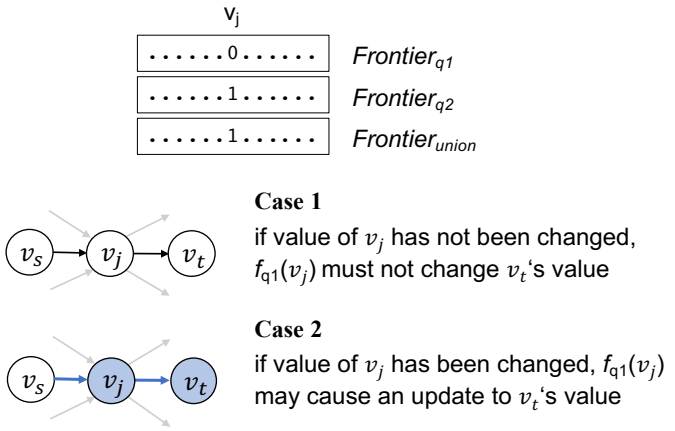


Figure 4.6: Correctness of Using Query-Oblivious Frontier.

the maintenance and accesses to the separate frontiers all together, dramatically reducing the memory footprint of concurrent query evaluation (see Section 4.4).

So far, we have introduced the intra-iteration alignment which addresses the potential misalignments among different frontier traversals in a global iteration. Next, we will shift the focus to more coarse-grained misalignments, along the iterations of different queries and during the formation of query batches.

4.3.3 Inter-Iteration Alignment

We first use a simple example to motivate the alignment problem, then formalize it and present a heuristic-based solution.

Motivation. In general, the evaluation of a graph query may access different parts of the graph in different iterations, thus the amount of graph sharing may vary depending on the interleaving of the (local) iterations of different queries. Revisit the examples in Table 4.2 (Section 4.2) and assume the two `sssp` queries are evaluated in the same batch, then compare their frontier overlapping per iteration with those in Table 4.3 where a different alignment

between the (local) iterations of the two queries is used: $\texttt{sssp}(v_2)$ starts two iterations later than $\texttt{sssp}(v_8)$. From the comparison, we can find that the latter alignment exposes more overlapped active vertices than the former (6 v.s. 2). As a result, when these active vertices are evaluated and their (out-)neighbors are accessed, the latter alignment will yield more shared graph accesses.

Table 4.3: A Better Alignment of Iterations.

Iter#	Frontier(sssp(v_2))	Frontier(sssp(v_8))
0	–	$\{v_8\}$
1	–	$\{v_4\}$
2	$\{\underline{v_2}\}$	$\{\underline{v_2}, v_6\}$
3	$\{\underline{v_3}, v_8\}$	$\{\underline{v_3}, v_9\}$
4	$\{v_4, \underline{v_5}, \underline{v_6}, \underline{v_7}\}$	$\{\underline{v_5}, \underline{v_6}, \underline{v_7}\}$
5	$\{\underline{v_9}\}$	$\{\underline{v_9}\}$
6	$\{\}$	$\{\}$

Next, we formalize the above inter-iteration alignment problem.

Problem Formalization. First, we define the alignment vector I for a given batch of queries $\mathcal{B}$ as follows:

Definition 15. Given a batch $\mathcal{B}$ of n queries $[q_1, q_2, ..., q_n]$, its ***alignment vector*** $I = [a_1, a_2, ..., a_n]$, where a_i is the global iteration number from which the evaluation of q_i is started.

Considering the batch $\mathcal{B}$=[$\texttt{sssp}(v_2), \texttt{sssp}(v_8)$], Table 4.3 shows an alignment where the alignment vector $I = [2, 0]$.

Next, we introduce the concept of *affinity* to quantify the amount of graph access sharing, which is defined as follows:

Definition 16. Given a query batch $\mathcal{B}$ and an alignment vector I, the **affinity** of this evaluation is defined by the following equation:

$$\mathit{Affinity}(\mathcal{B}, I) = 1 - \frac{\sum_{j=1}^{K} |\mathit{Frontier}^{j}_{union}|}{\sum_{j=1}^{K} \sum_{q_i \in \mathcal{B}} |\mathit{Frontier}^{j}_{q_i}|} \tag{4.2}$$

where $\mathit{Frontier}^{j}_{union}$ and $\mathit{Frontier}^{j}_{q_i}$ are the unified frontier and the separate frontier for query q_i, respectively, at iteration j, and K is the total number of global iterations.

Again, consider the examples in Table 4.3, where we have

$$\mathit{Frontier}^{2}_{union} = \{v_2, v_6\},\ \mathit{Frontier}^{3}_{union} = \{v_3, v_8, v_9\},$$
$$\mathit{Frontier}^{4}_{union} = \{v_4, v_5, v_6, v_7\},\ \mathit{Frontier}^{5}_{union} = \{v_9\}.$$

Hence, $\mathit{Affinity}(\mathcal{B}, I) = 1 - (2 + 3 + 4 + 1)/(8 + 10) = 1/2$. In comparison, we can also calculate the affinity for the prior alignment in Table 4.2: $\mathit{Affinity}(\mathcal{B}, I') = 1 - (2 + 3 + 5 + 2 + 3 + 1)/(8 + 10) = 1/9$. Obviously, the former achieves a significantly higher affinity.

The best affinity occurs when all separate frontiers are perfectly overlapped, while the worst affinity happens when no separate frontiers overlap at all through all the iterations. Note that the affinity may become negative in certain cases, due to the potential asynchronous evaluation as detailed in the proof of Theorem 14.

The above definition of affinity is based on the ratio of active vertices. Alternatively, one can also define affinity based on the ratio of active edges (the outgoing edges of active vertices), which is supposed to be more precise. However, our evaluation shows minimal differences between the two definitions in practice.

Now we can formalize the inter-iteration alignment problem:

$$\max_{\forall I}\ \mathit{Affinity}(\mathcal{B}, I) \tag{4.3}$$

That is, given an evaluation batch $\mathcal{B}$, the problem is to find out the best alignment vector I that maximizes the affinity.

Unfortunately, as the frontier of each query will NOT be known until the query execution is finished, we cannot precompute the affinity for a batch of queries without evaluating them. Hence, we cannot solve the above optimization problem precisely in advance. To address this challenge, we need a more proactive approach. Next, we present a heuristic to approximate the best alignment.

Heuristic-based Solution. First, we observe that *the distribution of frontier sizes tends to be highly biased across iterations*, thanks to the power-law nature of many real-world graphs. Figure 4.7 reports the frontier sizes across iterations during the evaluation of a few vertex-specific queries on two real-world graphs.

From the results, we can easily find some patterns in evaluating vertex-specific queries on power-law graphs: in the early iterations, the frontier grows exponentially, which we call the *expansion phase*. After reaching the "peak", the frontier starts to shrink quickly and steadily until it becomes empty, referred to as *stabilization phase*. The several iterations around the "peak" often dominate overall size of frontiers for the whole query evaluation.

With the above observations, we decide to focus the alignment on these dominating iterations, referred to as "*heavy iterations*". The rationale behind this decision is two-fold:

- First, heavy iterations expose more opportunities for shared graph accesses. The larger the frontiers are, the more likely that they overlap. In the extreme case, when the (separate) frontiers include all vertices, they are perfectly overlapped.

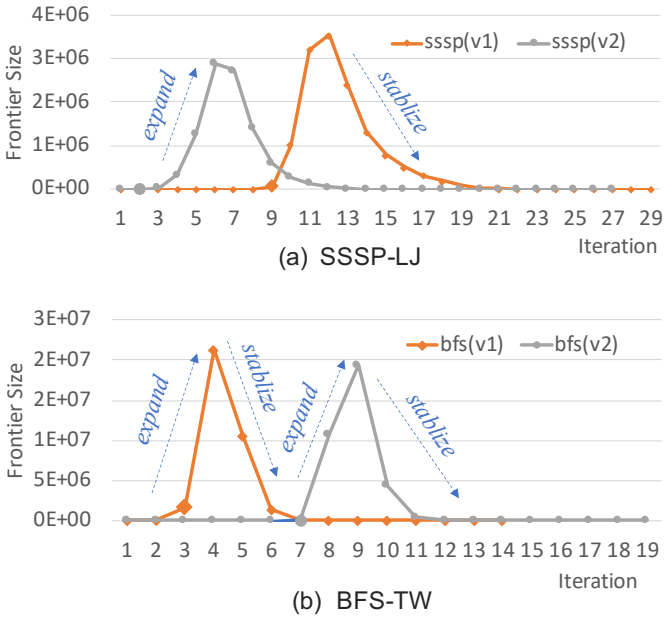


Figure 4.7: Frontier Size Distribution across Iterations.

- Second, as the vertex activations in the heavy iterations often dominate the total amount of vertex activations for the whole evaluation, their alignments could make a significant impact on the overall alignment.

To demonstrate the effectiveness of heavy iteration alignment in improving the overall affinity, we manually delayed the "faster" queries in Figure 4.7 such that their "peaks" align with those in the "slower" queries. As a result, we observed that the affinity value gets improved from -0.11 to 0.34 and from 0 to 0.47, respectively.

One can quantify the heavy iterations based on different metrics, such as the ranking of frontier sizes across iterations. However, regardless the metric being used, just like affinity, heavy iterations are *unknown* before the query evaluation. In fact, one may choose to detect the heavy iterations dynamically during the query evaluation and use that information to guide the alignments at runtime, for example, pausing the evaluation

Table 4.4: Arrival Time of Heavy Iterations

Query	Arrival Time	Query	Arrival Time
sssp(v1)	iter. 9	bfs(v1)	iter. 3
sssp(v2)	iter. 2	bfs(v2)	iter. 7

of queries whose heavy iterations have arrived, then resuming them after all the "slowest" query has reached its heavy iterations. Though the idea sounds promising, it has a caveat—to "pause and resume" the iterative evaluation of some queries, the system needs to keep "their contexts"—their individual frontiers. This requires to a design similar to the two-level frontiers (see Figure 4.5-(b)). As discussed earlier, this design is inferior to the query-oblivious frontier in terms of performance.

To work around the above dilemma, we propose to proactively approximate the *"arrival time" of heavy iterations*—the iteration that marks the beginning of heavy iterations. The key insight behind our arrival time approximation is the correlation between frontier size and the activation of high-degree vertices:

When evaluating a vertex-specific query on a power-law graph,

the frontier size often grows sharply once a high-degree vertex is activated.

To demonstrate the above phenomenon, Table 4.4 lists the first iteration where at least one of the top-4 high degree vertices is activated for the four queries used in Figure 4.7. These iterations are also highlighted with larger marks in Figure 4.7, which clearly indicate the beginning of (relatively) heavy iterations.

Based on the above discussion, we only need to identify the first iteration where a high-degree vertex is activated. In fact, given a high-degree vertex v_h (based on a threshold),

it takes i iterations for it to be activated, where i is the least number of hops from the source vertex v in query $q(v)$ to the high-degree vertex v_h. Such information can be pre-computed simply by running a BFS query on the high-degree vertex v_h, that is, `bfs(`v_h`)`. Note that, for directed graphs, the BFS query should run on the edge-reversed graphs or use a pull-based model.

Figure 4.8 summarizes our reasoning—reducing the inter-iteration alignment to the problem of BFS queries on high-degree vertices.

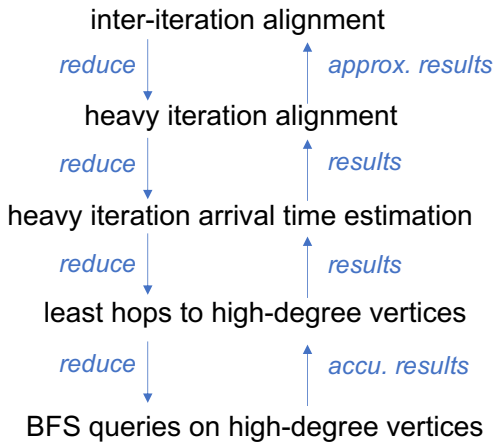


Figure 4.8: Flow of Solving Inter-Iteration Alignment.

Next, we present the algorithm of inter-iteration alignment (see Figure 4.9). First, some one-time preparation is needed: (i) identifying the top-K high-degree vertices in terms of the out-degree (due to the use of push model) at Line 2; (ii) reversing the edges' directions if the graph is directed (Line 3); (iii) running a BFS query on each selected high-degree vertex to find the least number of hops from an arbitrary vertex v_i to each high-degree vertex v_h (Line 4-5).

```
1  /* preparation for graph G */
2  HV = getTopHighOutDegreeVertices(G, K)
3  G_r = getEdgeReversedGraph(G)
4  for each vertex v_h in HV
5     leastHops[v_1 - v_max][v_h] = bfs (G_r, v_h)
6
7  /* find alignment vector I for query batch B */
8  getAlignment(B) {
9     for each query q(v_i) in B
10       closestHV[v_i] = min { leastHops[v_i][v_h] | ∀ v_h ∈ HV }
11    latestInBatch = max { closestHV[v_i] | ∀ q(v_i) ∈ B }
12    for each query q(v_i) in B
13       I [v_i] = latestInBatch − closestHV[v_i]
14 }
```

Figure 4.9: Pseudocode for Inter-Iteration Alignment.

After the preparation, the algorithm is ready to compute the alignment vector I for a given query batch $\mathcal{B}$. First, it computes the least hop number to the closest high-degree vertex for each query $q(v_i)$, stored in `closestHV[`v_i`]`. Then, it finds the largest value among `cloestHV[`v_i`]` for queries in the batch, which essentially is the latest time of reaching a high-degree vertex (i.e., arrival time) for a query in the batch, stored in `latestInBatch`. Finally, based on the difference of arrival time relative to the latest (`latestInBatch`), it calculates the alignment vector I.

From another perspective, the alignment delays the start time of certain queries in the evaluation batch, thus we also refer to the above technique as *delayed start*. As shown later in the evaluation, compared to the ground truth—the optimal alignment, the accuracy of the above heuristic-based alignment is quite high (usually off by at most 2 iterations) and also its performance is close to that with the optimal alignment. See more details in Section 4.4.

Next, we will move our discussion of alignment to the most coarse-grained level—query batching.

4.3.4 Alignment-Aware Batching

In the prior section, we align (local) iterations of different queries to improve the sharing of graph accesses (measured by affinity). In fact, we may achieve similar or even better effects by grouping queries whose iterations are better aligned into the same batch. For example, it is better to put `sssp(`v_1`)` (in Table 4.1) and `sssp(`v_2`)` into the same batch, rather than `sssp(`v_2`)` and `sssp(`v_8`)` (in Table 4.2), as the former exposes a better alignment at the iteration level.

Based on the above intuition, we propose *affinity-oriented query batching*. The goal is to maximize the affinity for queries in a batch through the management of batching policy.

Affinity-Oriented Query Batching. By default, all queries in the evaluation buffer are processed in the order they are received. Though intuitive, this first-come, first-serve policy may produce query batches with low affinity. To avoid this, affinity-oriented batching creates batches based on the affinity among queries, which can be approximated as detailed in the prior section. However, in theory, a simple affinity-oriented batching may postpone the processing of some queries—those exhibit poor affinity with most queries, with an unbounded delay. To avoid this caveat, we limit the number of queries considered each time for affinity-oriented batching using a threshold B_w. That is, every B_w queries in the buffer are scheduled together, referred to as a *batching window*.

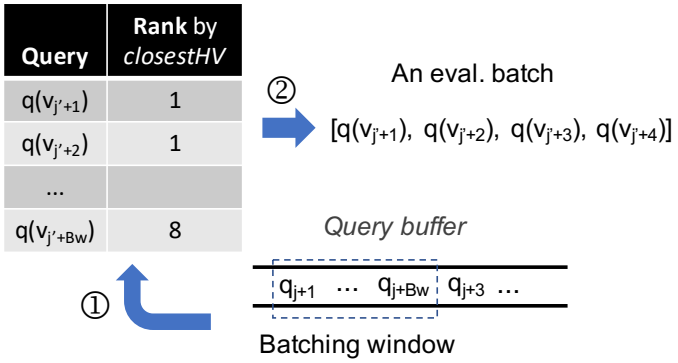


Figure 4.10: Affinity-Aware Query Batching.

Figure 4.10 illustrates the idea of affinity-oriented query batching. First, the earliest received B_w queries in the buffer are selected and ranked by their least number of hops to the closest high-degree vertex (i.e., `closestHV[]` in Figure 4.9). Then, every consecutive $|\mathcal{B}|$ ranked queries in the batch window are selected to form an evaluation batch. Note that array `closestHV[]` is pre-computed just like that used in inter-iteration alignment.

Connection with Inter-Iteration Alignment. Note that unlike the intra-iteration alignment which is orthogonal to the later two alignment techniques, the relation between the two inter-iteration alignments and affinity-oriented batching are not fully orthogonal. This is due to the fact that both aim at improving the affinity defined at the iteration level. In another word, if affinity-oriented batching has been employed, then the extra benefits from inter-iteration alignment might be limited, even though in theory, each alignment technique may provide its unique benefits.

So far, we have introduced the alignments at all three levels. Next, we briefly explain their implementations.

4.3.5 Implementation

We implemented `Glign` on top of the popular in-memory graph processing engine `Ligra` [106]. In fact, thanks to its high-efficiency, `Ligra` also serves as the base for two recently proposed concurrent graph systems: `Congra` [88] and `Krill` [18]. One of our goals is to implement `Glign` as a transparent runtime system, thus keeping the original programming interface of `Ligra` mostly untouched. For example, developers can still call functions `EdgeMap()` and `VertexMap()` for traversing edges and vertices, and `VertexSubset` remains to be the frontier representation.

Under the hood, `Glign` maintains the query-oblivious frontier and leverages the original parallelization supports from `Ligra` to process each query in parallel. To support batching, we extended `Ligra` to let it consume a query buffer based on the batch size B and the batching policy (e.g., affinity-oriented batching). At the beginning of an iteration, `Glign` first checks the alignment vector I (in Section 4.3.3) to decide if some queries need to be started at the current iteration. For better locality, the memory layout of vertex values is implemented as a single array and the value of vertex v_j for query q_i can be accessed via `ValArray[`v_j`*B+i]`, where `B` is the batch size. When the graph is loaded into the memory for the first time, Glign will automatically compute the least hops from each vertex to the closest high degree vertex (`closestHV[]`), which will later be used to guide the alignments.

4.4 Evaluation

This section evaluates the effectiveness of the proposed alignment techniques and the efficiency of `Glign`.

Table 4.5: Methods in Evaluation

Method	Brief Description
`Ligra-S`	Eval. queries in batch one by one w/ `Ligra` [106]
`Ligra-C`	Eval. queries in batch simultaneously w/ `Ligra` [106]
`GraphM` [139]	A high-throughput concurrent graph system
`Krill` [18]	A compiler & runtime for concurrent graph processing
`Glign-Intra`	`Glign` with only intra-iteration alignment
`Glign-Inter`	`Glign-Intra` + inter-iteration alignment
`Glign-Batch`	`Glign-Intra` + affinity-oriented batching
`Glign`	`Glign` with all proposed alignment techniques

4.4.1 Methodology

First, we set up two baselines for comparing with `Glign`: (i) `Ligra-S` and (ii) `Ligra-C`. The former evaluates the queries in a batch one after another, using `Ligra` [106]—a state-of-the-art in-memory graph processing engine. Note that `Ligra` itself processes each single query in parallel. The latter extends `Ligra` to support concurrent query evaluation using both unified and separated frontiers (see Section 4.3.2), representing a design that has been adopted by the existing concurrent graph processing systems [139, 18, 65].

Also, we compare `Glign` in general with two state-of-the-art concurrent graph processing systems that are publicly accessible: `GraphM` [139] and `Krill` [18]. To show the contributions of different techniques, `Glign` is configured differently as listed in Table 4.5. In addition to the above systems, we also tested a system design that exploits query-level parallelism—each concurrent query is evaluated using the serial implementation from BGL [107], while different queries are processed on different threads. However, we found it ran slower than our baseline `Ligra-S` in most cases tested.

Queries. We evaluated five types of graph queries, including BFS (*breadth-first search*), SSSP (*single source shortest path*), SSWP (*single source widest path*), SSNP (*single source narrowest path*), and Viterbi. Table 4.6 lists their vertex functions in pseudo-code.

All queries are vertex-specific in that they start from one source vertex and compute property values for all vertices in the graph. To generate the query set for each type of query, we followed a sampling strategy similar to the one used by Qi and others [93]. First, the graph vertices are divided into disjoint bins based on their distances (hops) to the (top-4) high-degree vertices. Then, these bins are scanned in rounds, and in each round a vertex is randomly picked from each bin, until 512 vertices are selected, which serve as the source vertices of our queries. In this way, the selected queries provide a better coverage of the entire graph structure. We assume all the 512 queries are already in the buffer when the systems start to process them. This allows us to focus on evaluating the throughput of the concurrent systems. One can also add the query arrival time information, with which the latency of processing each query could also be inferred. We leave such latency study for future work.

Besides grouping queries of the same types into the same query buffer (i.e., homogeneous query buffer), we also generated a query buffer of mixed types of queries, randomly selected with types of BFS, SSSP, SSWP, and SSNP. We refer to this scenario as "Heter".

By default, we set the query batch size to 64. For evaluating the impacts of batch size, we changed the batch size from 2 to 128.

Graph Data Sets. We primarily evaluate `Glign` on power-law graphs, which include five real-world graphs. For completeness, we also evaluate `Glign` on a couple of road networks,

Table 4.6: Vertex Functions of Graph Queries

Bench.	Pseudo-code of Vertex function $f(s)$
BFS	**for** each out-neighbor d of s level(d) = min { level(d), level(s) +1 }; **if** level(d) changed **then** add d to $frontier$;
SSSP	**for** each out-neighbor d of s dist(d) = min { dist(d), dist(s) + w(s, d) }; **if** dist(d) changed **then** add d to $frontier$;
SSWP	**for** each out-neighbor d of s wide(d) = max { wide(d), min { wide(s), w(s, d) } }; **if** wide(d) changed **then** add d to $frontier$;
Viterbi	**for** each out-neighbor d of s viterbi(d) = max { viterbi(d), viterbi(s) / w(s, d) }; **if** viterbi(d) changed **then** add d to $frontier$;
SSNP	**for** each out-neighbor d of s narrow(d) = min { narrow(d), max { narrow(s), w(s, d) } }; **if** narrow(d) changed **then** add d to $frontier$;

which are more like planar graphs. Their basic properties are summarized in Table 4.7. The number of edges ranges from 69M to 3.6B, and the diameter ranges from 10 to 9100.

Table 4.7: Graph Statistics

Graph	Abbr.	Directed	$\|V\|$	$\|E\|$	Avg. deg.	Dia.
LiveJournal [9]	LJ	Yes	4.8M	69M	14.2	13
Wikipedia [3]	WP	Yes	14M	437M	32.2	10
UK-2002 [15]	UK2	No	19M	524M	28.3	45
Twitter [60]	TW	Yes	42M	1.5B	35.3	15
Friendster [2]	FR	No	125M	3.6B	28.9	38
roadNet-CA [98]	RD-CA	No	2.0M	5.5M	2.81	849
roadNet-USA [98]	RD-US	No	24M	58M	2.41	9100

The experiments are conducted on a 32-core Linux server that equips with Intel Xeon E5-2683 v4 CPU and 512GB memory. The application is compiled with g++ 6.3 and runs on CentOS 7.9.

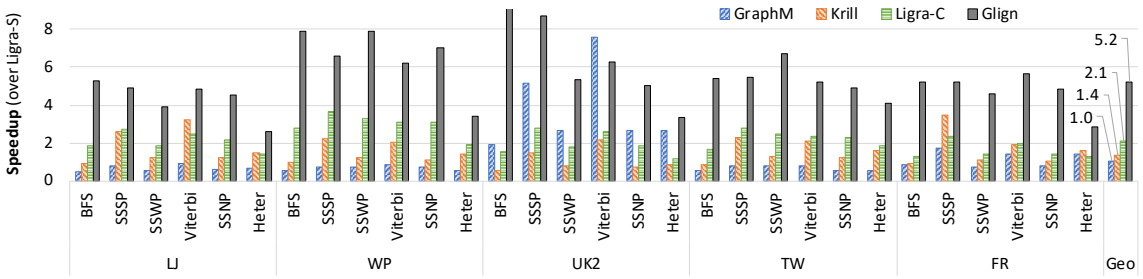


Figure 4.11: Overall Performance.

Next, we will first report the overall performance, followed by more detailed evaluation of each alignment technique.

4.4.2 Overall Performance

Table 4.8 shows the total execution time of evaluating a buffer of 512 queries using `Ligra-S`, while Figure 4.11 reports the speedups of the other systems over `Ligra-S`. From the results, we find that `Glign` clearly outperforms the other systems in almost all the cases (except for the case UK2-Viterbi). The highest speedup it reaches is 9.4× (in the case of UK2-BFS). On average, `Glign` achieves 5.2× speedup over the baseline `Ligra-S`.

Among the other systems, `GraphM` exhibits similar performance as `Ligra-S`. Note that, unlike the other systems in our evaluation, `GraphM` is not built on top of `Ligra`, instead, it is built on top of `GridGraph` [142], a system mainly designed for out-of-core graph processing. This difference in the base system selection could be one of the reasons that cause `GraphM` to perform worse than the other concurrent graph systems used in our evaluation.

`Krill` and `Ligra-C` both achieve a substantial average speedup over `Ligra-S` (1.4× and 2.1×), confirming the general benefits of concurrent query evaluation. However,

Table 4.8: Time of Evaluating 512 Queries using `Ligra-S`

	LJ	WP	UK2	TW	FR
BFS	66s	300s	334s	1104s	3646s
SSSP	176s	579s	986s	2332s	10076s
SSWP	97s	392s	507s	1734s	4580s
Viterbi	224s	499s	1388s	2129s	7926s
SSNP	99s	372s	493s	1630s	4440s
Heter	107s	398s	567s	1819s	5858s

both perform worse than `Glign`, though all the three systems share the same base system (`Ligra`). We believe this is mainly due to the locality improvement brought by the alignment techniques that `Glign` employed.

To confirm the above speculation, we measured the last-level cache (LLC) misses using the `perf` tool. The results are listed in Table 4.9. To save space, we report the results mainly for two graphs. From the results, we find that `Glign` incurs significantly less LLC misses than the other methods. For example, on LJ graph, `Glign`'s LLC misses is only 12%, 21%, 5%, and 23% of those (on average) incurred by `Ligra-S`, `Ligra-C`, `GraphM`, and `Krill`, respectively. These significant LLC miss reduction echos the substantial speedups brought by `Glign` as shown earlier in Figure 4.11.

In the following, we will break down the performance gains of `Glign` by evaluating each alignment technique.

4.4.3 Intra-Iteration Alignment

In this section, we evaluate the proposed query-oblivious frontier (Section 4.3.2) and compare it with the two-level frontier design (i.e., unified and separate frontiers) that

Table 4.9: LLC Misses (in billions)

		Ligra-S	Ligra-C	GraphM	Krill	Glign
LJ	BFS	9	11	44	15	**1**
	SSSP	28	15	77	15	**4**
	SSWP	21	12	52	15	**2**
	Viterbi	53	20	91	8	**4**
	SSNP	21	11	52	15	**2**
	Heter	23	15	45	11	**4**
Mean		26	14	60	13	**3**
TW	BFS	302	245	1083	394	**26**
	SSSP	621	237	1698	399	**53**
	SSWP	545	217	963	397	**29**
	Viterbi	757	242	1622	171	**48**
	SSNP	540	214	1202	396	**36**
	Heter	563	252	1201	241	**52**
Mean		555	235	1295	333	**41**

is employed by some of the existing concurrent query processing systems. Both designs ensure synchronized frontier traversal—the key to intra-iteration alignment. In our setting, `Ligra-C` employs the two-level frontier design, while `Glign-Intra` uses the query-oblivious frontier (other alignment techniques are disabled).

First, we have verified the correctness of all the query results produced by `Glign-Intra`, thus experimentally demonstrated the correctness of this new frontier design with real-world large data, complimenting the theoretical proof in Section 4.3.2.

Second, in terms of performance, Figure 4.12 reports speedups of `Glign-Intra` over `Ligra-C`. The results show that `Glign-Intra` yields consistent speedups across different queries and graphs, which range from 1.13× to 1.96×.

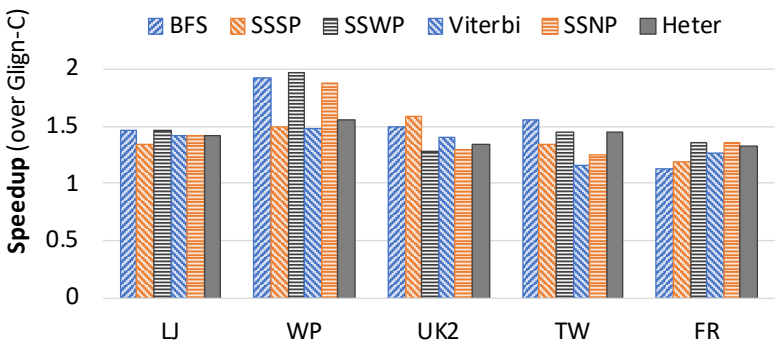


Figure 4.12: Speedups of `Glign-Intra` over `Ligra-C`

Table 4.10: LLC Misses Reduction by `Glign-Intra`

(Numbers are the ratios between the LLC misses of `Glign-Intra` and the LLC misses of `Ligra-C`)

	LJ	OR	WP	UK2	TW	FR
BFS	22%	29%	24%	13%	23%	32%
SSSP	33%	42%	37%	12%	34%	36%
SSWP	32%	44%	27%	17%	28%	24%
Viterbi	34%	44%	39%	19%	36%	31%
SSNP	32%	41%	28%	17%	31%	21%
Heter	35%	48%	42%	22%	31%	31%
Geomean	31%	41%	32%	16%	30%	29%

In addition, we also collected the LLC misses of `Glign-Intra`, which may offer more direct evidence on the effectiveness of query-oblivious frontier. The results are reported in Table 4.10. From the results, we find that `Glign-Intra` can consistently reduce the LLC misses under all the evaluated cases, and the average reduction is quite substantial—its LLC misses are only around 30% on average of those incurred by the baseline `Ligra-C`. The reduction mainly comes from the elimination of separate frontiers and the two-level frontier checking used by the baseline and other existing concurrent graph systems. These results, to a large extent, explain `Glign-Intra`'s speedups shown in Figure 4.12.

Table 4.11: Memory Footprint Breakdown (64 queries)

	LJ		TW	
	Ligra-C	Glign-Intra	Ligra-C	Glign-Intra
Graph	545MB	545MB	11,400MB	11,400MB
Vertex Value	1,180MB	1,180MB	10,200MB	10,200MB
Frontier	296MB	4.6MB	2,540MB	39.7MB

Finally, to get a sense of the memory reduction brought by the query-oblivious frontier in comparison to a two-level frontier, we profiled the memory footprints. Table 4.11 reports the sizes of major data structures in `Ligra-C` and `Glign`: (i) the graph topology data, the values of all vertices, and the frontier (as a labeling array). Note that even though the frontier is only a relatively small portion of the total memory footprint, it is accessed entirely in every iteration. In comparison, only some parts of the graph and some vertex values are accessed in each iteration. The results show that the frontier size is reduced dramatically with query-oblivious frontier, which leads to the LLC misses reduction as shown in Table 4.10.

4.4.4 Inter-Iteration Alignment

To show the benefits of inter-iteration alignment, we compare `Glign-Inter` with `Glign-Intra`.

Performance. Figure 4.13 reports the speedups of `Glign-Inter` over `Glign-Intra`. Overall, `Glign-Inter` achieves better performance in all evaluated cases, except for WP-SSWP and WP-SSNP. The speedups range from 0.89× to 2.95×. This demonstrates the benefits of our proposed inter-iteration alignment technique—*delayed start*. In general, we found

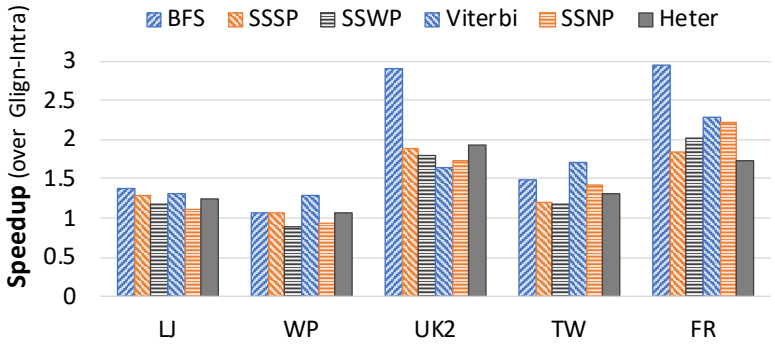


Figure 4.13: Speedups of `Glign-Inter` over `Glign-Intra`

`Glign-Inter` does not perform significantly better on WP graph. The reason might be related to the fact that WP has a relatively smaller diameter (see Table 4.7). Smaller diameters imply that the vertices selected as the source vertices of the queries tend to be closer to each other in terms of hops. Based on the discussion in Section 4.3.3, queries with closer source vertices tend to align better (i.e., yielding better affinity) during their evaluation. As a result, there is less room for inter-iteration alignment to improve.

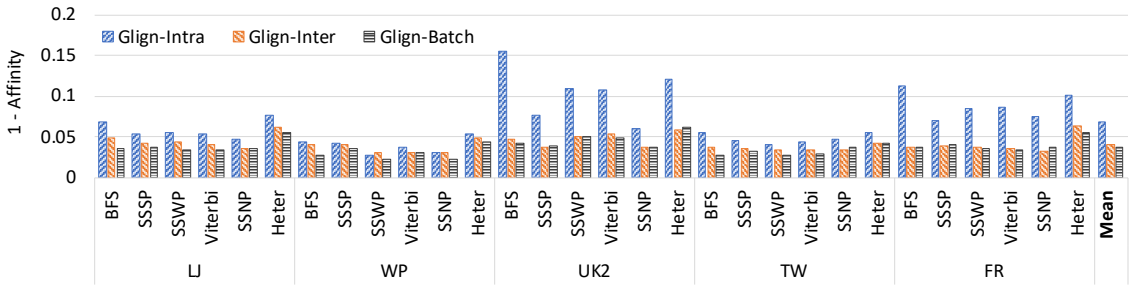


Figure 4.14: Affinity Comparison: `Glign-Intra` vs `Glign-Inter` vs `Glign-Batch`

Affinity. To get a deeper understanding of the improvements, we also collected the affinity values (see Definition 16). The results are shown in Figure 4.14. Note that we set the Y-axis to $1 - affinity$ because, for a batch of 64 queries, the affinity value tends to be close

to 1, using $1 - affinity$ can better reflect its significants. In fact, $1 - affinity$ reflects *how much different frontiers are misaligned*, thus the lower the value is, the better alignment we get. The results show that `Glign-Inter` substantially reduces the divergence, from 0.068 to 0.041 on average. The highest reductions happen to BFS on UK2 and FR graphs, which match well with the top two best speedups (2.91× and 2.95×) reported in Figure 4.13. The results also show that divergence for `Glign-Intra` is already very low in the case of WP graph, leaving little room for further improvements. This explains the limited speedups achieved by `Glign-Inter` on this graph (see Figure 4.13).

In addition, we report the LLC miss reduction by `Glign-Inter` over `Glign-Intra` in Table 4.12. In general, the results echo well the above findings. For example, the highest cache miss reductions also happen to BFS on UK2 and FR graphs (37% and 32%) and the reduction ratio on WP graph is the least.

Table 4.12: LLC Misses Reduction by `Glign-Inter`

(Numbers are the ratios between the LLC misses of `Glign-Inter` and the LLC misses of `Glign-Intra`)

	LJ	WP	UK2	TW	FR
BFS	67%	96%	37%	68%	32%
SSSP	73%	91%	53%	77%	51%
SSWP	73%	101%	39%	76%	48%
Viterbi	64%	69%	38%	60%	37%
SSNP	74%	97%	41%	67%	42%
Heter	64%	81%	31%	65%	51%
Geomean	69%	88%	39%	69%	43%

Heuristic v.s. Ground Truth. To examine the effectiveness of the proposed heuristic, we profiled the ground-truth best alignments for 512 sampled batches, where each batch consists of only two queries. The best alignment of each batch is found by exhaustively

Table 4.13: Ground Truth Study of `Glign-Inter`

			Speedup		
Diff	Cnt	Ratio	Glign-Intra	Glign-Inter	Best-Align
0	172	33.6%	1.36×	1.51×	1.51×
1	217	42.4%	1.32×	1.42×	1.51×
2	102	19.9%	1.20×	1.27×	1.46×
3	20	3.9%	1.13×	1.20×	1.47×
4	0	0.0%	N/A	N/A	N/A
5	1	0.2%	1.62×	1.52×	1.56×
Sum./Avg.	512	100%	1.30×	1.41×	1.50×

trying all possible alignments and calculating the corresponding affinity value. Table 4.13 summarizes our findings. Among the 512 batches, our heuristic finds the best alignments in 33.6% of them, and the difference with the optimal alignment is within 2 iterations for over 95% cases. As to the speedups, the best alignments outperform ours (1.50× vs 1.45×), indicating that extra room exists for further improving the performance via alignments.

Profiling Costs. As discussed in Section 4.3.3, our heuristic requires profiling—running BFS queries on (four) high-degree vertices. Note that the profiling happens at the beginning when the system and graph is set up. It is a one-time effort for each graph, with benefits applying to different types of queries that run on the graph. Table 4.14 lists the profiling costs on two graphs (LJ and TW), compared to the query evaluation time on the same graphs. During the concurrent query evaluation, accessing the profiling result (a table lookup) is quick and the cost is negligible.

Table 4.14: Profiling Costs

		LJ	TW
Profiling Cost		0.20s	3.84s
Query Eval. Cost (batch size:64, `Glign`)	SSSP	4.47s	53.40s
	BFS	1.56s	25.51s

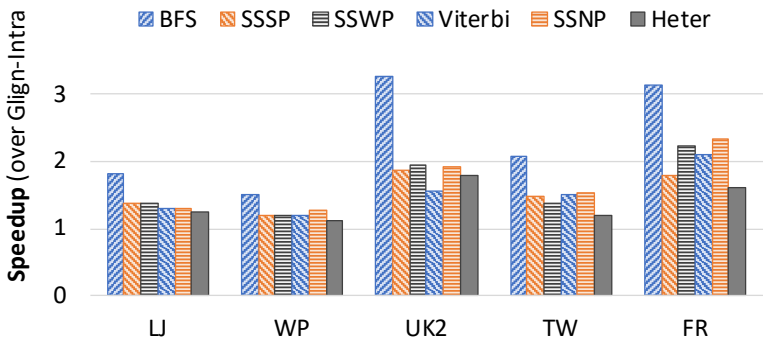


Figure 4.15: Speedups of `Glign-Batch` over `Glign-Intra`

4.4.5 Alignment-Oriented Batching

To demonstrate the benefits of alignment-oriented batching, we compare `Glign-Batch` with `Glign-Intra`. Figure 4.15 reports the speedups of `Glign-Batch` over `Glign-Intra`. These speedups are slightly higher than those achieved by `Glign-Inter` (Figure 4.13). This is expected as both alignments essentially explore the same affinity opportunities. The additional improvements indicate that the alignment opportunities across queries (in the query buffer) are slightly higher than those within a single evaluation batch, which is also confirmed by the affinity differences between `Glign-Inter` and `Glign-Batch` as shown in Figure 4.14.

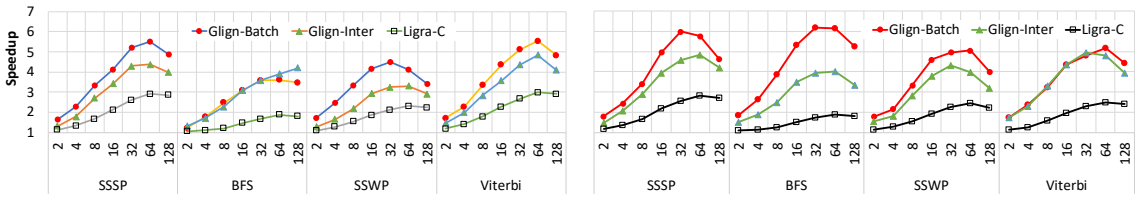


Figure 4.16: Impacts of Query Batch Size (Left: LJ graph, Right: TW graph)

4.4.6 Impacts of Batch Size

To understand the sensitivity of `Glign` to a basic parameter—the batch size. We changed the batch size from 2 to 128. Note that 128 is the largest value `Glign` can achieve based on the memory capacity of our machine and the size of the evaluated graphs.

Figure 4.16 reports the results using four types of queries and two input graphs. Most curves in the figure follow a similar trend, that is, the speedup first grows as the batch size increases, until the batch size reaches around 64, then the speedup starts to drop. The upward trend indicates that increasing the degree of concurrency tends to be beneficial, while the downward trend indicates there is a limit for the benefit—the memory pressure also increases as the batch size increases, which eventually would curb the gain.

4.4.7 Performance on Road Networks

Though `Glign` is primarily designed for processing power-law graphs, for completeness, we also report its performance on some road networks (RD-CA and RD-US). Table 4.15 reports the speedups of `Glign` and their variants over our baseline `Ligra-S`. First, the results show that `Glign-Intra` still achieves good or even higher speedups (e.g., 9.3× and 14.7× speedups for BFS). This is due the fact that, typically, only a small portion of the road network needs to be accessed in each iteration, which makes the costs to

Table 4.15: Performance on Road Networks

	RD-CA			RD-US		
	SSSP	BFS	SSWP	SSSP	BFS	SSWP
Ligra-S	369.8s	239.5s	219.6s	15224s	7916 s	1298 s
Ligra-C	2.91×	5.30×	1.45×	1.25×	2.66×	0.30×
Glign-Intra	6.08×	9.31×	2.99×	2.04×	14.67×	1.86×
Glign-Inter	6.85×	10.05×	3.15×	1.75×	13.51×	1.25×
Glign-Batch	8.36×	11.15×	3.66×	2.52×	15.37×	1.91×
Glign	8.90×	12.41×	3.64×	2.77×	16.91×	1.29×

access frontier(s) relatively higher. On the other hand, the extra benefits brought by the inter-iteration alignment and alignment-oriented batching are more limited. This is because evaluation on such graphs often fails to yield sufficiently "heavy" iterations, making the affinity issue less of a concern.

4.4.8 Comparison with iBFS

Finally, we compare `Glign` with `iBFS` [65]—a specialized graph system dedicated to concurrent BFS queries. It is an early work that groups BFS query instances and leverages shared frontier traversal, which resembles affinity-oriented batching. However, there are a few key differences between the two. First, `iBFS` maintains both the unified and separate frontiers to achieve synchronized frontier traversal, just like `Ligra-C` and `Krill`. In comparison, `Glign` uses unified frontier only (i.e., query-oblivious frontier); Second, to group BFS queries, `iBFS` uses a different heuristic based on the out-degrees of source vertices. In specific, it requires two conditions for grouping queries: (i) out-degrees of source vertices should be less than p; and (ii) the source vertices must connect to at least one common vertex whose out degree is greater than q. In comparison, `Glign` groups queries

Table 4.16: Comparison with iBFS

	iBFS	Glign-Intra	Glign-Batch
LJ	16.6s	0.98×	1.78×
OR	31.6s	0.94×	1.49×
WP	41.1s	0.92×	1.45×
UK2	130.8s	0.95×	3.17×
TW	276.3s	1.08×	2.10×
FR	2465.1s	1.02×	3.04×

only based on the affinity (or number of hops to a high-degree vertex). Last but not least, iBFS does not support inter-iteration alignment. To experimentally compare the two, we have implemented the heuristic of iBFS in Ligra-C. Note that the original iBFS work is implemented for the GPU platform.

Table 4.16 reports the performance of using iBFS for evaluating 512 BFS queries, and the speedups of Glign-Intra and Glign-Batch over iBFS. Overall, we find that the performance of iBFS is similar to Glign-Intra, but substantially slower than Glign-Batch. On average the gap between iBFS and Glign-Batch is between 1.45× and 3.17×. A further examination reveals that the heuristic of iBFS is too strict—it works better when there are an extremely large number of queries involved (e.g., querying all vertices in the graph).

4.5 Summary

This chapter reveals a major performance issue in concurrent graph processing—alignment of graph traversals. It addresses this issue at three levels. First, it proposes the query-oblivious frontier to achieve synchronized frontier traversal within each global iteration. Second, it introduces a heuristic-based solution based a series of insights and

observations to intelligently align the iterations of different queries and to group queries with different affinities. It integrates the proposed techniques into a runtime system called `Glign`. A full evaluation of `Glign` has confirmed the effectiveness of the proposed alignments and demonstrated superior performance over state-of-the-art concurrent graph processing systems.

Chapter 5

Improving Throughput of Graph-based ANNS with Temporal Information

5.1 Introduction

Similarity search in high-dimensional datasets has become an important part of modern deep learning applications, including recommendation systems, retrieval augmented generation, content filtering, large language models (LLMs), and others. These datasets usually contain millions or billions of high-dimensional vector representations (a.k.a. embeddings) of documents, images, videos, and user content generated by pre-trained deep neural networks. The algorithm used to find vectors similar to user input from datasets is known as the k-nearest neighbor search, where the most similar k embeddings for a query are returned.

Calculating the exact k-nearest vectors can be very expensive in high-dimensional space, and most real-world applications can tolerate small errors. Therefore, the *approximate nearest neighbor search (ANNS)* has been widely deployed in production environments, which provides better query latency and throughput and retains good search quality.

There are several approaches to solving the ANNS problem. The Inverted File Indexing (IVF) method [7, 108, 52] partitions vectors into buckets. Thus, queries need only search a portion of datasets rather than the entire space. More recently, graph-based solutions for ANNS [90, 51, 72, 97] have been shown to perform superiorly while achieving high recall. Graph-based ANNS algorithms first construct a proximity graph from the vector dataset. Queries are evaluated on the constructed graph through a *best-first search*. The *best-first search* terminates when all vertices in a fixed-size buffer (the beam) have been visited. And the top-k vectors are returned as the query results.

By carefully profiling and characterizing the ANNS workload, we have identified some limitations in existing ANNS solutions:

Unexploited parameter space Despite the high throughput (queries per second, or QPS) achieved by graph-based ANNS, existing systems control the QPS-recall tradeoff only at a coarse-grained level: there is one parameter called the beam size, which is the maximum queue size of the best-first search. The larger the beam size, the higher the recall and the worse the QPS is.

Unexploited nearest neighbors correlation The query-dataset correlation was first described by Baranchuk et al. [10] as "content drift", where the recency relationship is

observed among user queries and the returned results. In several real-world datasets (images or ad videos), user queries usually exhibit consistent temporal distributions of their nearest neighbors over periods of time. For example, more recent data are more likely to be close to the query (the recency pattern), or nearest neighbors are more likely to appear in some specific period of time (the seasonal pattern). Unfortunately, such temporal information is not embedded in the vector and is not exploited for query optimization.

To this end, we propose TANNS, an adaptive and efficient system for graph-based approximate nearest neighbor search with learned query-data correlation. TANNS improves both the graph construction and search by novel techniques. First, we show that **the best-first search can be better parameterized to create more room for higher performance**. We observed that the search can be divided into two phases according to the hardness of achieving a certain recall score: **the first phase** is very fast but can easily find a good number of accurate top-k nearest neighbors to the query vector (80% to 90% according to our profiling); **the second phase** runs much slower than the first one. Since the best-first search algorithm stops only when there are no more unvisited vertices in the buffer queue and it is oblivious to the recall score, there is no explicit early termination condition that satisfies a target recall. This results in many vertex visits and distance computations for the second phase but only slightly improves the recall score, which we refer to as "finding needles in a haystack" in the ANNS problem.

Based on the above interesting observation, our first goal is to improve the efficiency of the search further. Specifically, we introduce new parameters for the two phases, including step sizes that control the greediness of exploration at each iteration of the search

(i.e., how many vertices are expanded per iteration) and a cut-off factor for discarding unpromising candidates. Because these two phases are artificially divided, each exhibiting different patterns (fast or slow), we find that each phase requires a different set of parameters. Generally speaking, the first phase requires a smaller step size (usually 1) to converge without harming the search quality. The search in the second phase can be more aggressive with a larger step size. Further, the cut-off factor should be different in each phase, unlike previous solutions [50, 72] that have applied a cut for the whole search.

To exploit the correlation between queries and their results, we show that it is possible to construct and refine the graph to favor these distributions. We have developed a new algorithm for constructing the graph, which connects vertices in an adaptive manner. By incorporating such correlation information, we are able to create a smaller and less dense graph, while still providing faster navigation to nearest neighbors. As far as we know, these techniques represent the first set of methods for optimizing graph queries under such a biased distribution.

Overall, in this work, we make the following contributions:

- We conducted detailed profiling and gave an in-depth analysis of the graph-based ANNS workload and depicted its optimization space. Our observations led to a full parameterization of the search algorithm. We also provided a framework for parameter auto-selection.

- To incorporate data that are not embedded into vectors (e.g., temporal information), we proposed a new graph construction algorithm to favor the query-results correlation.

- We conducted comprehensive experiments and demonstrated that, to achieve the same level of recall, TANNS can improve the query throughput by up to 1.9× compared to DiskANN when query-data correlation exists while reducing the constructed graph size by around 30%. Our techniques introduce no slowdown for queries that do not exhibit any correlation with their results.

5.2 Background

5.2.1 Approximate Nearest Neighbor Search (ANNS)

We first introduce the problem definition of k-NNS. Given a dataset P of n points (vectors) in d-dimension space and a query point q, the k nearest neighbor search returns a set $\mathcal{K}$ that contains k points, such that $max_{p\in\mathcal{K}}||p,q|| \leq min_{p\in\mathcal{P}\setminus\mathcal{K}}||p,q||$. Note that the distance between two points $p, q \in \mathbb{R}^d$, denoted as $||p,q||$, can be either Euclidean distance (L_2 norm) or cosine distance. The Euclidean distance is used in most real-world datasets.

k-ANNS is defined as k-approximate NNS, in which only approximate results are returned. Without any ambiguity, we use ANNS for k-ANNS throughout this paper. The most commonly used metric for measuring the accuracy of ANNS is the *recall* score. Specifically, the $k@k'$ recall of query q is defined as $\frac{|\mathcal{K}\cap\mathcal{K}'|}{|\mathcal{K}|}$, where $\mathcal{K}$ is the ground-truth set of k-nearest neighbors of q in the dataset and $\mathcal{K}'$ is the output of an k'-ANNS algorithm. In this paper, we use k' that is equal to k when calculating recall scores.

5.2.2 Graph-based ANNS

ANNS can be efficiently solved using graph-based approaches. Graph-based ANNS consists of two parts: graph construction and search on the graph.

ANNS graph construction There are various graph construction algorithms, including NSG [36], HNSW [71], and DiskANN [51], and many others [83, 78, 19, 28, 35, 45], most of which build the *Proximity Graphs* that enable fast navigation of a query point to its closest neighbors in the dataset. For example, the graph construction algorithm, Vamana from DiskANN [51], builds the proximity graph incrementally: it inserts points into the already built graph by using Algorithm 6. The high-level idea is that for a given dataset $\mathcal{P}$ and a point $p \in \mathcal{P}$, the out-neighbors of p (denoted as $N_{out}(p)$) are decided through the beam search and a pruning procedure (Algorithm 5 and Algorithm 7).

The pruning procedure is used for selecting both long and short edges as p's out-neighbors. This helps avoid results converging to local optima. The rationale behind this pruning is that long edges connect neighbors that are far away, and short edges connect neighbors that are close to point p. Creating long edges is necessary as they provide fast navigation during graph traversals towards the region of points close to the query point, while short edges ensure that once such a region is reached, the search can converge quickly.

Beam search Beam search is a variant of *best-first search* used for answering ANNS queries. It works in a greedy manner. A queue or buffer called *beam* $\mathcal{L}$ is maintained, which has the maximum capacity of L (the beam size). The beam stores points of the dataset along with their distances to the query point. In each iteration, the search algorithm expands

Algorithm 5 Beam Search

```
1: function BEAMSEARCH(G, s, q, k, L)
2:     L ← {s}
3:     V ← ∅
4:     while L \ V ≠ ∅ do
5:         p* ← arg min_{p∈L\V} d(p, q)
6:         L ← L ∪ N_out(p*)
7:         V ← V ∪ p*
8:         if |L| > L then
9:             update L to retain closest L points to q
10:    return [closest k points from L; V]
```

Algorithm 6 Insert

```
1: function INSERT(p, G, s, L, R)
2:     L, V ← beamSearch(p, s, L, 1)
3:     N_out(p) ← prune(p, V, R)
4:     for v ∈ N_out(p) do
5:         N_out(v) ← N_out(v) ∪ {p}
6:         if |N_out(x)| > R then
7:             N_out(x) ← prune(x, N_out(x), R)
```

the *unvisited point in the beam* that is currently the closest to the query vector. The out-neighbors of the selected point are examined and their distances to the query point are computed. Neighboring points are inserted into the beam, and points in the beam will be sorted according to the distances in ascending order. This iterative process continues until all points in the beam are visited. The top-k points are returned as the ANNS results for the query. The beam search algorithm is depicted in Algorithm 5.

Algorithm 7 Prune from DiskANN

```
1: function PRUNE(p, V, R)
2:     V ← V ∪ N_out(p)
3:     L ← ∅
4:     while V ≠ ∅ do
5:         p* ← arg min_{p'∈V} d(p, p')
6:         L ← L ∪ {p*}
7:         if |L| = R then
8:             break
9:         for p' ∈ V do
10:            if α · d(p*, p') ≤ d(p, p') then
11:                V ← V \ p'
12:    return L
```

5.3 ANNS Workload Characterization

In this section, we report several interesting observations while profiling the ANNS workload. Based on our findings, we then describe potential optimization space for ANNS and introduce a fully parameterized beam search algorithm.

5.3.1 Parallelism of ANNS queries

The computation pattern of graph-based ANNS queries is quite different from other graph queries (e.g., BFS, SSSP, PageRank, etc.). One key difference is the computation load of a single query evaluation. Vertex-specific queries such as SSSP require visiting all vertices in the graph to calculate the shortest path values from the query source vertex to all other vertices, which leads to a large memory footprint and many vertex value computations within one iteration. To improve the performance of such queries, efficient parallel algorithms have been proposed, most of which exploit the intra-query and intra-iteration parallelism. On the contrary, an ANNS query only visits a small portion of the graph and a few vertices are computed for their distances to the query vector. For example, on

BIGANN-50M dataset, on average, each query only visits around 360 points and calculates 7000 distances to find the top-10 nearest neighbors accurately. In addition, most ANNS systems aim at improving the query throughput (QPS) [51, 72, 90] by serving as many concurrent queries as possible.

Based on these observations, we propose to optimize ANNS throughput by improving the beam search algorithm rather than the parallelism of a single query evaluation.

5.3.2 ANNS Phases Based on Hardness

Almost all graph-based ANNS solutions use the greedy beam search depicted in Algorithm 5 with different implementation details. The beam search algorithm works iteratively until there are no more unvisited vertices in the beam $\mathcal{L}$, and it has been shown to achieve good performance and quality for solving ANNS queries; however, the throughput-recall tradeoff can be tricky. For ANNS queries, the quality (recall) of returned top-k results is determined by the *number of distance computations* during the graph traversal of search. The more vertices in the graph have their distances to the input query vector computed, the higher the chance that vertices close to the query can be found. The *beam size* L for the beam search decides how many distances can be computed. The larger beam size allows for exploring more of the graph, thus enhancing the recall score. However, increasing the beam size for better recall significantly deteriorates the system's query throughput, which we refer to as the dilemma of recall and throughput. To improve the query throughput while achieving a recall target, recent works, such as DiskANN [51] and ParlayANN [72], use a parameter sweeping approach to find the best beam size for a given dataset and report the highest throughput. Further, ParlayANN also adopts a method called $(1 + \epsilon)$-cutting

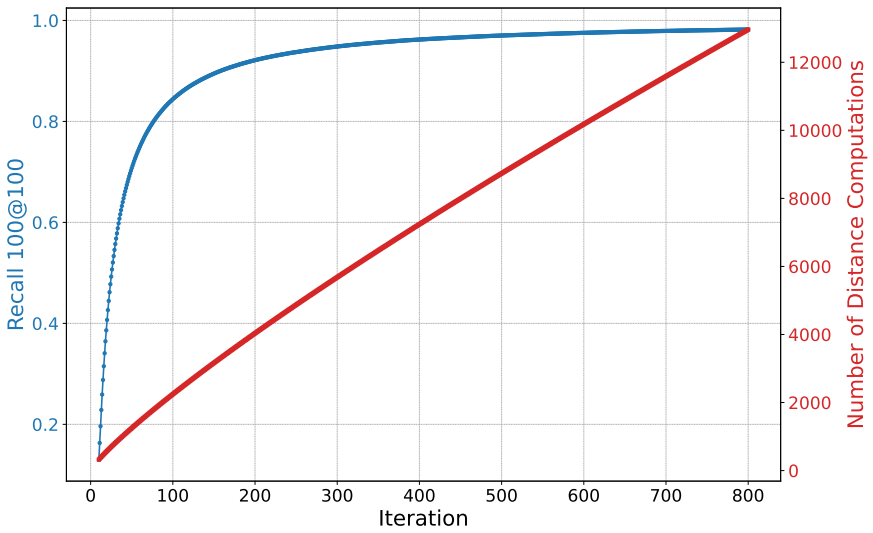


Figure 5.1: Iteration vs. Recall 100@100 and Number of Distance Computations.

from Iwasaki et al. [50] for the search. In each iteration, only vertices with distances to the query point that are less than $(1 + \epsilon)$ times the current k-th nearest neighbor will be kept in the beam.

However, sweeping the above parameters helps obtain a better QPS-recall tradeoff curve. We show the potential to increase query throughput by exploring the beam search design space. The intuition is that *not every iteration contributes equally to the final results*, which we refer to as the hardness of ANNS computation. Figure 5.1 shows the recall score (left y-axis) for the top-100 nearest neighbors in each iteration on dataset BIGANN-50M (50 million points) averaged over 10K queries. In this example, the progress of ANNS query evaluation accelerates rapidly within the first 100 iterations; the algorithm quickly identifies over 80% of the final answers. Figure 5.1 also shows the accumulated total number of distance computations (right y-axis). Combining these two lines, it is easy to see that the

number of distance computations required to identify the top-k nearest neighbor increases as the iterative query evaluation progresses. In the later iterations, numerous distance calculations are performed, yielding only a few closest neighbors.

Based on the above phenomenon, it is natural to divide the search into two phases according to the hardness of discovering closer neighbors to the query point. The first phase easily finds the majority of nearest neighbors in a short time. The second phase runs much longer and calculates many distances to ensure that the query quality reaches the high recall regime (from 0.9 to 0.999). However, the benefit-cost ratio is low in the second phase, and terminating the search early usually leads to lower recalls.

5.3.3 Parameterized Beam Search

To further optimize the search, we propose a fully parameterized beam search algorithm by introducing separate cut-off factors (ϵ) and expanding sizes (e) for the two phases. The expanding size is the number of vertices to be expanded in one iteration (the original beam search always expands one vertex).

We now explain the rationales for introducing new parameters. First of all, the two phases exhibit completely different graph traversal patterns. The first phase easily finds many accurate vertices due to the fast navigability of the constructed graph. In this case, long edges help locate the graph region close to the query vector. It is intuitive to think that Phase 1 should use a smaller expanding size than Phase 2 to avoid prematurely populating the beam with unpromising vertices. Similarly, the cut-off factor for Phase 1 should be as large as possible, meaning little to no vertices should be discarded. In Phase 2, the search explores more vertices to improve the query quality. However, the distance

differences among vertices explored in Phase 2 are subtle. This indicates that a more aggressive exploration scheme should be employed, i.e., expanding multiple vertices in one iteration. The vertices to be expanded in the same iteration are quite similar in terms of their distances to the query vector, so expanding them in the same iteration gives their neighbors an equal opportunity to be considered.

To distinguish the two phases, one can use the number of iterations. However, the iteration count is not a good indicator as it is affected by the datasets and how the graph was constructed (the maximum degree). Instead, we propose to use a simple criterion, which is the iteration when the first k vertices in the sorted beam are all visited, to decide the boundary of the two phases. This criterion can be easily examined at runtime during the query evaluation with little overhead. Algorithm 8 depicts the fully parameterized beam search algorithm. We also remark on another benefit of identifying two phases – it enables early termination of ANNS computation when only good enough results are required. We found that the recall achieved by Phase 1 is only the dataset's intrinsic property and is irrelevant to query vectors. Thus, the proposed criterion that the first k vertices have been visited can be used to stop a beam search with arguably good recall scores.

5.4 Temporal Data-Assisted Graph Construction

When constructing the graph from a vector dataset, existing works do not incorporate additional information, such as the timestamp associated with each data point. Such information is lost as it is not embedded into the data vector. In this section, we introduce our technique for further enhancing graph construction when additional information

Algorithm 8 Parameterized Beam Search

```
1: function parameterizedBeamSearch(G, s, q, k, L, e_1, e_2, ϵ_1, ϵ_2)
2:     L ← {s}
3:     V ← ∅
4:     [ϵ, e] ← [ϵ_1, e_1]
5:     check ← true
6:     while L \ V ≠ ∅ do
7:         len ← min(e, |L \ V|)
8:         P ← ∅
9:         for i = 0...len − 1 do
10:            p* ← arg min_{p∈L\V} d(p, q)
11:            P ← P ∪ {p*}
12:            V ← V ∪ {p*}
13:        for p ∈ P do
14:            ngh_len ← Deg(p) − 1
15:            for i = 0...ngh_len do
16:                v ← N_out[i]
17:                if d(v, q) ≤ ϵ * d(L^kth, q) then
18:                    L ← L ∪ {v}
19:        if |L| > L then
20:            update L to retain closest L points to q
21:        if check then
22:            L_k ← TopK(L)                    ▷ k=min(|L \ V|, max(k,10))
23:            if L_k \ V = ∅ then
24:                check ← false
25:                [ϵ, e] ← [ϵ_2, e_2]
26:    return [closest k points from L; V]
```

is available. The enhanced graph has a smaller size and can boost the ANNS query evaluation without harming the query quality. Specifically, we illustrate our technique using the temporal information of a dataset (timestamps).

5.4.1 Exploiting the Query-Results Correlation

It has been observed that in real-world datasets, there is usually a recency relation between an input query and ANNS results [10]. One common pattern is that more recent content is more likely to appear in the top-k results of ANNS queries. Another

commonly seen pattern is the seasonal trend, in which contents from the same season as the query time are more likely to be returned. Figure 5.2 shows an example of such query-results correlation in the time dimension[1]. The vectors in the dataset are evenly distributed over a 36-month span, while the queries' nearest neighbors exhibit a temporal correlation. However, neither index-based nor graph-based ANNS systems have effectively utilized this temporal correlation to enhance the query performance.

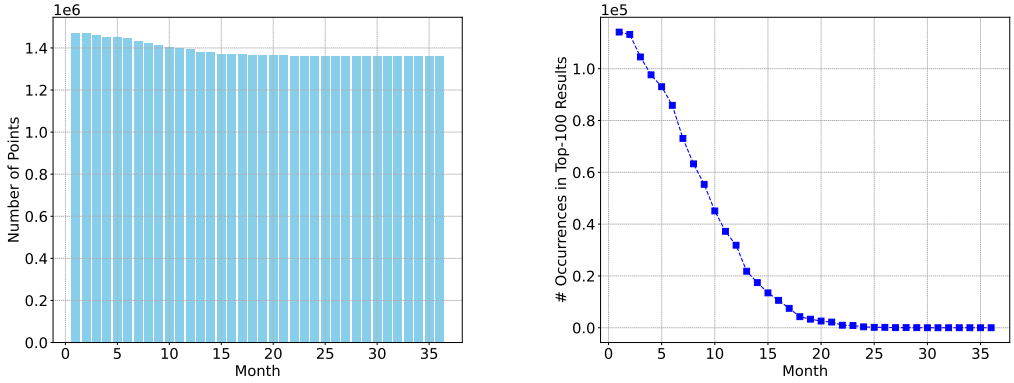


Figure 5.2: Vectors Distribution (left) and Query-Results Correlation (right) on the BIGANN-50M Dataset.

In this work, we propose to construct the ANNS graph *adaptively*. The intuition is that rather than pruning out points based on a fixed scaling factor α (Line 10 in Algorithm 7) and the vectors distance, we define α as a function of time to consider the temporal correlation of query-results when determining whether to create an edge between two nodes. We begin by modeling the query-results correlation, denoted by $\alpha(t)$, and then illustrate how $\alpha(t)$ can be used to guide the construction of the ANNS graph.

[1]Synthetic timestamps are used to illustrate this pattern due to the lack of available public datasets.

To incorporate the temporal correlation, we use a fitting function to convert the pruning factor α to $\alpha(t)$:

$$\alpha(t) = b - \frac{b-a}{1+e^{kt-T/2}} \tag{5.1}$$

$\alpha(t)$ is a scaled and shifted logistic function. Parameters a, b, k, and T are the scaling and shifting factors. The range of $\alpha(t)$ is $[a, b)$, T is the time span of interested data points in the dataset, k controls the steepness of $\alpha(t)$, and t is the absolute time difference between two data points ($t = |t_{p^*} - t_{p'}|$).

We now explain the rationales of using $\alpha(t)$. The original pruning procedure (Algorithm 7) removes vertex p' from p's out-neighbors candidates set if p' is too close to p^*. This ensures that both short and long edges can be added to p's outgoing edge list, resulting in better navigability. Our design (Algorithm 9) considers the extra time dimension; it prunes more aggressively for vertices that are close in time. The pruning plays a less significant role for vertices that are far away in time — edges are inserted mainly based on distance. The overall effect of our $\alpha(t)$-based pruning for graph construction is that it provides better navigability for vertices that are close in time, i.e., the search reaches the region near the query point more quickly. In addition, the constructed graph has a smaller average degree, reducing the memory footprint.

Algorithm 9 Adaptive Pruning

function AdaptivePrune(p, $\mathcal{V}$, R)
  $\mathcal{V} \leftarrow \mathcal{V} \cup N_{out}(p)$
  $\mathcal{L} \leftarrow \emptyset$
  while $\mathcal{V} \neq \emptyset$ **do**
    $p^* \leftarrow \arg\min_{p' \in \mathcal{V}} d(p, p')$
    $\mathcal{L} \leftarrow \mathcal{L} \cup \{p^*\}$
    if $|L| = R$ **then**
      break
    for $p' \in \mathcal{V}$ **do**
      if $\alpha(|t_{p^*} - t_{p'}|) \cdot d(p^*, p') \leq d(p, p')$ **then**
        $\mathcal{V} \leftarrow \mathcal{V} \setminus p'$
  return $\mathcal{L}$

5.5 Evaluation

5.5.1 Experimental Setup

We evaluate the TANNS and compare it with DiskANN [51]. ParlayANN [72] provides implementations of the above algorithms within their framework. To perform a fair comparison, we also implemented our graph construction algorithm and parameterized beam search on top of `ParlayANN`. All experiments were run on a Google Cloud n2-standard-128 instance with two Intel Xeon 3rd-generation CPUs with 128 vCPUs in total and 512 GB memory.

Datasets We use five datasets in our experiments: BIGANN-10M, BIGANN-50M, DEEP-10M, GIST, and TEXT2IMAGE-10M. Among them, BIGANN-10M and BIGANN-50M are sampled from the BIGANN dataset [53] that consists of 1 billion SIFT images embedded as 128-dimensional vectors. DEEP-50M is randomly sampled from the DEEP1B dataset [8] that consists of 1 billion image vector embeddings of 96 dimensions. The GIST dataset was first introduced by Jegou et al. [52]. It contains 1M Holiday images, and the descriptor

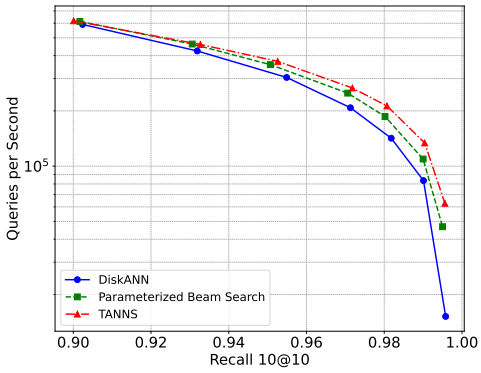
dimensionality is 960. TEXT2IMAGE-10M is randomly sampled from TEXT2IMAGE [1], which contains image embeddings produced by the SeResNet-101 model and textual queries encoded by the DSSM model, both have a dimension of 200.

Timestamps generation Since the datasets studied by Baranchuk et al. [10] are proprietary and not publicly available, we mimic the recency pattern observed in their VideoAds dataset in the following way. We first randomly generate a timestamp for each point in the dataset, and the time range spans 36 months. We then evaluate a set of sampled queries and use their top-100 results to mimic the query-results correlation, which follows a half-normal distribution. During the graph construction, we use the following adaptive pruning function $\alpha(t)$ in Equation 5.1 for all datasets:

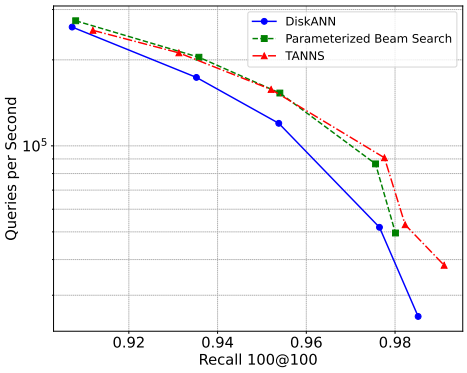
$$\alpha(t) = 1.8 - \frac{0.8}{1 + e^{0.8t-16}} \tag{5.2}$$

Here $a = 1.0$, $b = 1.8$, $k = 0.8$, and $T = 36$.

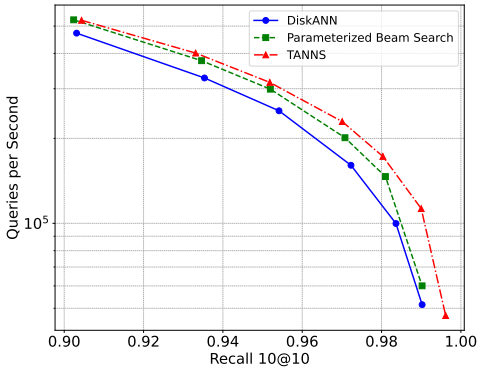
Algorithm parameters All graphs are constructed with $R = 32$ (the degree bound), $L = 64$ (the beam size when building the graph), and $\alpha = 1.2$ (the pruning parameter). For the beam search, we perform a parameter sweep over the parameter space for both the baseline algorithm and ours and choose the best performance for each specific recall target. As we are interested in the high recall regime (from 0.8 to 0.999), we omit the results for low recalls (from 0.0 to 0.7).



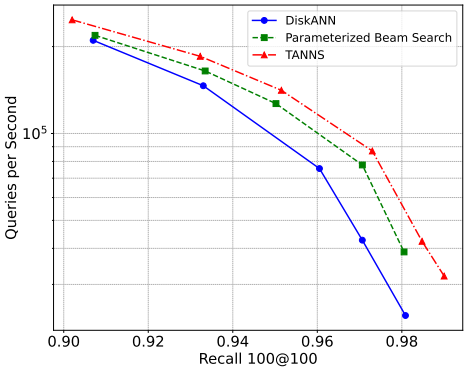
(a) BIGANN-10M, k=10



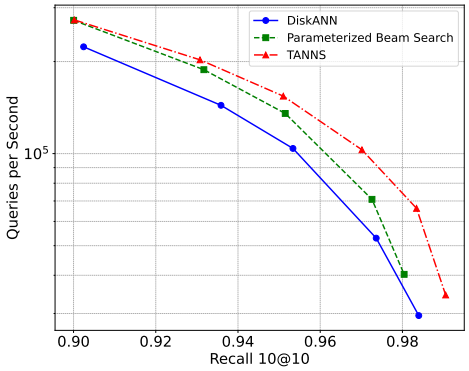
(b) BIGANN-10M, k=100



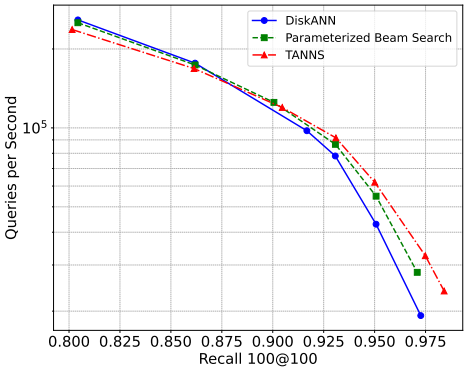
(c) BIGANN-50M, k=10



(d) BIGANN-50M, k=100

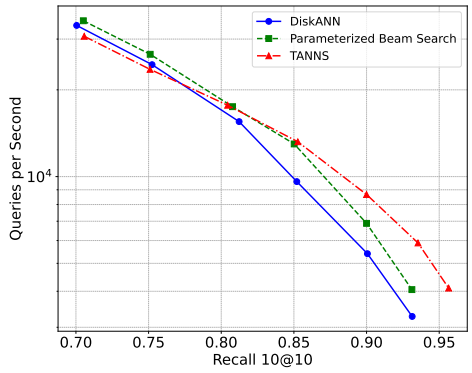


(e) DEEP-10M, k=10

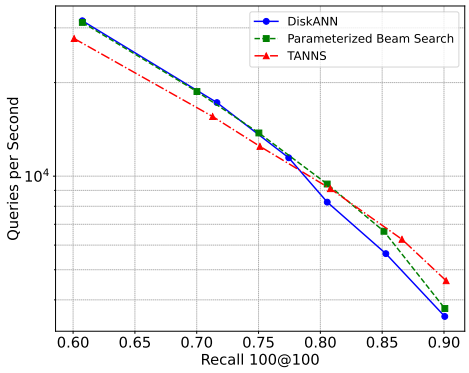


(f) DEEP-10M, k=100

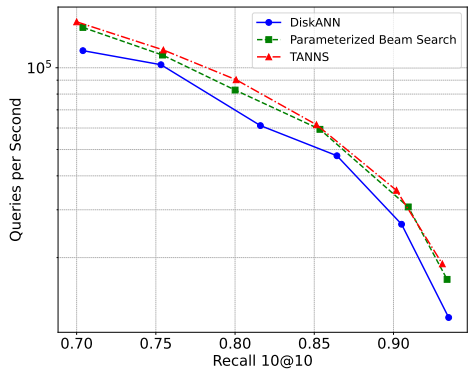
Figure 5.3: QPS-recall curves.



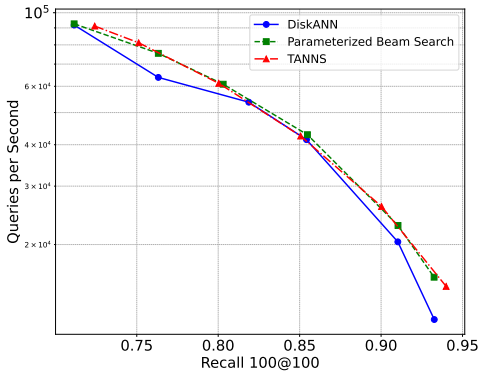
(a) GIST, k=10



(b) GIST, k=100



(c) TEXT2IMAGE-10M, k=10



(d) TEXT2IMAGE-10M, k=100

Figure 5.4: QPS-recall curves. (Continued)

5.5.2 Comparison with DiskANN

We use graphs constructed by DiskANN's Vamana algorithm (Algorithms 6 and 7) and evaluate queries using the default beam search in Algorithm 5 as the baseline setup. We report the performance of the proposed parameterized beam search using baseline graphs as inputs. The TANNS is also evaluated by using the graphs constructed by Algorithm 9 with the generated timestamp data and the parameterized beam search. We report the QPS-recall curves for both $k = 10$ and $k = 100$ on all datasets. All queries are issued

simultaneously and evaluated concurrently since throughput is a more relevant metric when serving ANNS queries on large shared memory machines.

Figures 5.3 and 5.4 show QPS-recall curves for a variety of datasets. In general, our parameterized beam search can achieve better throughput for the same recall target than the original beam search used in DiskANN *with the same input graph*. This throughput improvement is mainly due to the two-phase separation of beam search and newly introduced parameters for each phase. Our experiments indicate that, for all datasets, the parameter sweep consistently selects a large cut-off factor ϵ_1 (resulting in no dropping) and a unit step size ($e_1 = 1$) for Phase 1. For Phase 2, the algorithm selects the step size e_2 larger than e_1 and a smaller cut-off factor ϵ_2 (resulting in a more significant dropping effect). This observation confirms the necessity of dividing the search into two phases: the first phase is critical in finding most of the final nearest neighbors, so any cut-off or larger step sizes can harm the quality of Phase 1. The second phase only finds a few final nearest neighbors, but it requires many distance computations. Therefore, expanding the candidates and dropping unpromising results more aggressively can improve the efficiency of finding the nearest neighbors.

TANNS further improves the query throughput by up to 4.1× (BIGANN-10M, $k = 10$) compared to the baseline DiskANN, when using both parameterized beam search and the temporal information constructed graph. On average, TANNS achieves 1.9× higher throughput than DiskANN for the highest recall.

Table 5.1: Graph Statistics

Graph		Avg. Degree
BIGANN-10M	DiskANN	29.1
	Ours	24.2
BIGANN-50M	DiskANN	10.9
	Ours	9.5
DEEP-10M	DiskANN	27.2
	Ours	21.6
GIST	DiskANN	16.7
	Ours	10.8
TEXT2IMAGE-10M	DiskANN	27.7
	Ours	21.0

5.5.3 Graph Size Reduction

Table 5.1 shows the graph sizes when constructed by DiskANN's Vamana algorithm and our adaptive construction algorithm. When the query-results correlation exists, the graphs constructed by our adaptive pruning algorithm have smaller average degrees and reduced graph size.

5.6 Summary

We introduce the TANNS system, which constructs a proximity graph incorporating temporal information and optimizes the evaluation of queries that exhibit correlations between queries and results. A fully parameterized search algorithm is also designed, allowing for extensive performance tuning. TANNS achieves up to a 1.9× speedup in query throughput compared to the state-of-the-art DiskANN implementation while maintaining the same recall levels. Additionally, TANNS can reduce the size of the constructed graph by up to 30% without compromising query quality.

Chapter 6

Related Work

6.1 Static Graph Processing Systems

Many graph systems [41, 20, 66, 70, 120, 100, 42, 99, 104, 106, 118, 126, 54, 87, 103] are capable of solving queries on static graphs in various scenarios. PnP [126] offers efficient algorithms for computing point-to-point graph queries. CoreGraph [54] exploits edge centralities and creates a proxy graph to support fast iterative graph query evaluation. PowerGraph [41], PowerLyra [20], GraphX [42], and Pregel [70] solve queries in a distributed environment. Tigr [87] and Subway [103] are GPU graph systems.

To improve the performance of a single query evaluation, GraphIt [137, 136] designs a DSL that provides custom scheduling functions for exploring various optimization opportunities. Julienne [26] is specialized for bucketing-based algorithms like k-core and approximate set-cover, which cannot be supported efficiently in Ligra [106]. There are also works aimed at improving the memory locality for a single query evaluation [82, 133, 135].

6.2 Streaming Graph Analyses

Streaming Graph Systems. Many querying systems for dynamic graphs employ incremental evaluation. Earlier systems either only support incremental query evaluation in the presence of edge insertions (e.g., Chronous [43]) or compute approximate query results (e.g., Kineograph [22]). To handle edge deletions for incremental query evaluation, systems like KickStarter [117], RisGraph [34], and Tornado [105] record the value dependency for monotonic path-based algorithms (KickStarter also tracks each vertex's level in the dependency tree). Tornado utilizes Lamport Clocks [61] to guarantee consistency and correctness in a distributed environment. GraphBolt [75] and DZig [74] support both edge insertions and deletions for accumulative graph algorithms. Ingress [40] is a system that automatically incrementalizes graph algorithms. GIM-V [109] employs incremental graph processing based on matrix-vector operations. Differential Dataflow [80, 79] and Naiad [84] are generalized incremental computation models which are also capable of processing graph workloads. CommonGraph [5, 6] incrementally evaluates a stream of graph snapshots by finding a common graph and converting all edge deletions to edge insertions.

Incremental Algorithms for Streaming Graph Analyses. To the best of our knowledge, the first incremental algorithm for handling SSSP edge deletions was described by Ramalingam et al. [96], which briefly points out the connection between the handling of weight changes and the handling of edge insertions and deletions. Some recent works have focused on finding theoretical bounds for various incremental graph algorithms [31, 32], particularly when link weights undergo slow changes [48], and when incorporating temporal information [12].

Data Structures for Changing Graphs. There have been works for enhancing graph mutation performance. Aspen [27] supports low-latency graph mutation by using a compressed tree-based graph representation. VCSR [49] leverages packed memory array (PMA) to enable graph mutation on CSR. In addition, the indexing method has been employed to improve graph mutation performance [34, 115]. Recently, Terrace [89] proposed to use a hierarchical data structure to store edges based on the vertex degree.

6.3 Concurrent Graph Evaluation Systems

Seraph [128, 129] is an early work studying system-level supports for concurrent graph query processing. Its key idea is to decouple the graph structure from query-specific data to allow concurrent query evaluations to *share the common graph* structure data. Other early works include iBFS [65] and multi-source BFS [110], which are systems dedicated to the evaluation of concurrent BFS queries. iBFS groups BFS instances to leverage *shared frontier traversal*. While the above works introduced the ideas of graph sharing and frontier sharing among concurrent queries, Seraph is aimed at the distributed environment, and iBFS targets GPU platforms. Krill [18] is a recent work that not only exploits graph sharing but also gains benefits from efficient management of *property data* being computed.

A few other systems also target distributed and out-of-core concurrent graph query processing. For example, GraphM [139] supports concurrent query evaluation with graph sharing. Queries can be submitted at any time and executed concurrently. MultiLyra [76] and BEAD [77] support efficient batched query evaluation with graph sharing and frontier sharing in order to amortize the communication costs among computation nodes in the

cluster. Finally, CGraph [134] optimizes the multi-query processing with a focus on *out-of-core* systems and also supports evolving graphs. It also benefits from the sharing of the underlying graph.

There are recent works for multi-query processing on a single multicore shared memory machine. Congra [88] is built on top of Ligra [106] and tries to maximize the memory bandwidth by forking processes for new queries through the guidance of a scheduler. However, each graph query requires heavy offline profiling to obtain the memory bandwidth and scalability characteristics. Moreover, since each query is evaluated by a separate process, Congra exploits no graph sharing or frontier sharing. SimGQ [125] exploits the shared sub-computations of queries in the batch and adopts the reuse technique proposed in VRGQ [55].

6.4 Graph Applications in Emerging Domains

Graph processing systems are becoming increasingly important in various fields, including graph neural networks (GNNs), vector search, and healthcare. In the context of graph neural networks, these systems are essential for handling large-scale graph data effectively, which is crucial for training models that can capture complex relationships and node features [112, 111, 127, 143, 145, 132, 123, 144, 21]. In vector search, which is fundamental to many modern AI applications, such as recommendation systems and large language models [38, 44, 73, 101, 119, 122], graph processing systems optimize the retrieval of nearest neighbors in high-dimensional spaces [51, 39, 72, 36, 90]. Furthermore, in healthcare, these systems allow for the analysis of intricate networks of patient data and disease patterns, leading to better predictive models and personalized treatment plans [64, 4, 69, 68, 29].

6.5 Hardware Accelerators for Graph Processing

GraphPulse [94] is an asynchronous graph processing accelerator for static graphs. JetStream [95] supports streaming graphs and incremental computations. It also exploits the monotonicity property of iterative graph queries. GraphPulse and JetStream address the single query evaluation scenario. LCCG [138] is a graph accelerator that supports concurrent graph jobs by utilizing a topology-aware approach with new hardware units. HyperGRAF [17] and HDReason [16] are hardware accelerators for knowledge graph mining and reasoning. Basak et al. [11] improves streaming graph analyses through software and hardware co-design driven by input knowledge and batch-reordering technique. MEGA [37] is the hardware accelerator for graph analytics over evolving graphs. It supports evaluating queries over multiple graph snapshots simultaneously. GraphABCD [131] supports asynchronous graph analytics with a faster convergence rate via block coordinate descent.

Chapter 7

Conclusions and Future Directions

7.1 Conclusions

This dissertation presents several innovative strategies that aim to improve the scalability and efficiency of graph processing systems. These strategies are designed to support low-latency streaming analysis and high-throughput concurrent query evaluation. The contributions can be summarized across four main enhancements: the Tripoline system for generalized streaming graph analysis, IncBoost for scalable incremental graph computation, the Glign runtime system for concurrent query evaluation, and enhancements to graph-based approximate nearest neighbor search (ANNS).

The Tripoline system is a major improvement in streaming graph systems. In the past, graph systems were limited because they required prior knowledge of queries, which often resulted in computationally expensive full query evaluations. However, Tripoline overcomes this limitation by using principles similar to triangle inequalities. This allows

existing query results to be reused, which speeds up the evaluation of new user queries and expands the applicability of the system to various vertex-specific graph problems.

Addressing the scalability challenges inherent in incremental graph computations, especially under conditions of significant graph changes, IncBoost introduces algorithmic enhancements and system-level optimizations. By adopting a novel bottom-up dependency tracing technique, IncBoost efficiently identifies vertices impacted by edge updates, thereby bypassing the need for extensive data access and handling edge weight updates more directly and efficiently than conventional methods.

Glign enhances the shared graph access of concurrent query processing, a critical aspect for achieving higher throughput in graph analysis. Glign's innovative alignment strategies—spanning intra-iteration and inter-iteration phases—optimize memory access patterns, substantially reducing last-level cache misses and improving the performance of concurrent graph traversals.

The dissertation also explores the design space of graph construction and searching phases for ANNS, particularly in scenarios involving temporal information. The proposed graph construction algorithm considers potential correlations between input queries and final answers over time, enhancing the relevance and accuracy of search results. Moreover, a fully parameterized best-first search algorithm has been devised, offering enhanced flexibility for performance tuning.

7.2 Future Directions

Designing efficient systems for non-monotonic graph algorithm. This dissertation primarily focuses on vertex-centric, monotonic path-based graph algorithms. However, there is also a significant demand for systems optimized for non-monotonic graph algorithms. Non-monotonic algorithms, such as PageRank, Personalized Rank, and Graph Neural Networks (GNNs), allow for approximated computation and can benefit greatly from algorithmic and system-level optimizations.

Graph-based approximate nearest neighbor search (ANNS) represents just one of many non-monotonic algorithms. Extending the principles and techniques developed in this dissertation to other non-monotonic algorithms holds great potential. These algorithms can benefit from batching for better cache localities, incremental computation for promptly reflecting fast-changing data, and insights from datasets, such as data sparsity, which can also shed light on designing more efficient graph systems in those domains.

System support for graph applications on heterogeneous computing platforms. This dissertation focuses on shared memory machine environment and extends some of the proposed techniques into distributed systems. Emerging platforms such as CPU-GPU heterogeneous computation and Compute Express Link (CXL)–enabled interconnected computation systems provide significant potential for data-intensive workloads with optimized resource utilization and energy efficiency.

System-level and algorithmic optimizations of graph algorithms targeting these new architectures are in high demand. Graph computation tasks can be dispatched to

different processor types (such as GPUs, CPUs, and FPGAs) based on their workload characteristics to achieve optimized overall performance.

Our proposed two-phase ANNS algorithm is a promising example for CPU-GPU heterogeneous computing. In this approach, the fast and accurate first phase can be executed on the multicore CPU to meet low-latency requirements, while the more computationally expensive second phase can benefit from GPU's massively parallel architecture.

Bibliography

[1] Benchmarks for Billion-Scale Similarity Search. Date accessed May 11, 2024 from `https://research.yandex.com/blog/benchmarks-for-billion-scale-similarity-search`.

[2] Friendster network dataset. `http://konect.cc/networks/friendster/`, 2013. Accessed: 2022-01-02.

[3] Wikipedia links, english network dataset. `http://konect.cc/networks/wikipedia_link_en/`, 2013. Accessed: 2022-01-02.

[4] Bilal Abu-Salih, Muhammad Al-Qurishi, Mohammed Alweshah, Mohammad Al-Smadi, Reem Alfayez, and Heba Saadeh. Healthcare knowledge graph construction: A systematic review of the state-of-the-art, open issues, and opportunities. *Journal of Big Data*, 10(1):81, 2023.

[5] Mahbod Afarin, Chao Gao, Shafiur Rahman, Nael Abu-Ghazaleh, and Rajiv Gupta. Commongraph: Graph analytics on evolving data. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS 2023, page 133–145, New York, NY, USA, 2023. Association for Computing Machinery.

[6] Mahbod Afarin, Chao Gao, Shafiur Rahman, Nael Abu-Ghazaleh, and Rajiv Gupta. Commongraph: Graph analytics on evolving data (abstract). In *Proceedings of the 2023 ACM Workshop on Highlights of Parallel Computing*, HOPC '23, page 1–2, New York, NY, USA, 2023. Association for Computing Machinery.

[7] Alexandr Andoni, Piotr Indyk, Thijs Laarhoven, Ilya Razenshteyn, and Ludwig Schmidt. Practical and optimal lsh for angular distance. *Advances in neural information processing systems*, 28, 2015.

[8] Artem Babenko and Victor Lempitsky. Efficient indexing of billion-scale datasets of deep descriptors. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2055–2063, 2016.

[9] Lars Backstrom, Dan Huttenlocher, Jon Kleinberg, and Xiangyang Lan. Group formation in large social networks: membership, growth, and evolution. In *Proceedings*

of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining, pages 44–54, 2006.

[10] Dmitry Baranchuk, Matthijs Douze, Yash Upadhyay, and I Zeki Yalniz. Dedrift: Robust similarity search under content drift. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 11026–11035, 2023.

[11] Abanti Basak, Zheng Qu, Jilan Lin, Alaa R Alameldeen, Zeshan Chishti, Yufei Ding, and Yuan Xie. Improving streaming graph processing performance using input knowledge. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 1036–1050, 2021.

[12] Matthew Baxter, Tarek Elgindy, Andreas T Ernst, Thomas Kalinowski, and Martin WP Savelsbergh. Incremental network design with shortest paths. *European Journal of Operational Research*, 238(3):675–684, 2014.

[13] Markus Bläser. A new approximation algorithm for the asymmetric tsp with triangle inequality. *ACM Transactions on Algorithms (TALG)*, 4(4):1–15, 2008.

[14] Robert D Blumofe, Christopher F Joerg, Bradley C Kuszmaul, Charles E Leiserson, Keith H Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. *ACM SigPlan Notices*, 30(8):207–216, 1995.

[15] Paolo Boldi and Sebastiano Vigna. The webgraph framework i: compression techniques. In *Proceedings of the 13th international conference on World Wide Web*, pages 595–602, 2004.

[16] Hanning Chen, Yang Ni, Ali Zakeri, Zhuowen Zou, Sanggeon Yun, Fei Wen, Behnam Khaleghi, Narayan Srinivasa, Hugo Latapie, and Mohsen Imani. Hdreason: Algorithm-hardware codesign for hyperdimensional knowledge graph reasoning, 2024.

[17] Hanning Chen, Ali Zakeri, Fei Wen, Hamza Errahmouni Barkam, and Mohsen Imani. Hypergraf: Hyperdimensional graph-based reasoning acceleration on fpga. In *2023 33rd International Conference on Field-Programmable Logic and Applications (FPL)*, pages 34–41. IEEE, 2023.

[18] Hongzheng Chen, Minghua Shen, Nong Xiao, and Yutong Lu. Krill: a compiler and runtime system for concurrent graph processing. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–16, 2021.

[19] Qi Chen, Haidong Wang, Mingqin Li, Gang Ren, Scarlett Li, Jeffery Zhu, Jason Li, Chuanjie Liu, Lintao Zhang, and Jingdong Wang. Sptag: A library for fast approximate nearest neighbor search, 2018.

[20] Rong Chen, Jiaxin Shi, Yanzhe Chen, Binyu Zang, Haibing Guan, and Haibo Chen. Powerlyra: Differentiated graph computation and partitioning on skewed graphs. *ACM Transactions on Parallel Computing (TOPC)*, 5(3):1–39, 2019.

[21] Ting Chen, Song Bian, and Yizhou Sun. Are powerful graph neural nets necessary? a dissection on graph classification. *arXiv preprint arXiv:1905.04579*, 2019.

[22] Raymond Cheng, Ji Hong, Aapo Kyrola, Youshan Miao, Xuetian Weng, Ming Wu, Fan Yang, Lidong Zhou, Feng Zhao, and Enhong Chen. Kineograph: taking the pulse of a fast-changing and connected world. In *Proceedings of the 7th ACM european conference on Computer Systems*, pages 85–98, 2012.

[23] Avery Ching, Sergey Edunov, Maja Kabiljo, Dionysios Logothetis, and Sambavi Muthukrishnan. One trillion edges: Graph processing at facebook-scale. *Proceedings of the VLDB Endowment*, 8(12):1804–1815, 2015.

[24] Atish Das Sarma, Sreenivas Gollapudi, Marc Najork, and Rina Panigrahy. A sketch-based distance oracle for web-scale graphs. In *Proceedings of the third ACM international conference on Web search and data mining*, pages 401–410, 2010.

[25] Fethi Demim, Kahina Louadj, and Abdelkrim Nemra. Path planning for unmanned ground vehicle. In *2018 5th International Conference on Control, Decision and Information Technologies (CoDIT)*, pages 748–750. IEEE, 2018.

[26] Laxman Dhulipala, Guy Blelloch, and Julian Shun. Julienne: A framework for parallel graph algorithms using work-efficient bucketing. In *Proceedings of the 29th ACM Symposium on Parallelism in Algorithms and Architectures*, pages 293–304, 2017.

[27] Laxman Dhulipala, Guy E Blelloch, and Julian Shun. Low-latency graph streaming using compressed purely-functional trees. In *Proceedings of the 40th ACM SIGPLAN conference on programming language design and implementation*, pages 918–934, 2019.

[28] Wei Dong, Charikar Moses, and Kai Li. Efficient k-nearest neighbor graph construction for generic similarity measures. In *Proceedings of the 20th international conference on World wide web*, pages 577–586, 2011.

[29] Xinyu Dong, Rachel Wong, Weimin Lyu, Kayley Abell-Hart, Jianyuan Deng, Yinan Liu, Janos G Hajagos, Richard N Rosenthal, Chao Chen, and Fusheng Wang. An integrated lstm-heterorgnn model for interpretable opioid overdose risk prediction. *Artificial intelligence in medicine*, 135:102439, 2023.

[30] Charles Elkan. Using the triangle inequality to accelerate k-means. In *Proceedings of the 20th international conference on Machine Learning (ICML-03)*, pages 147–153, 2003.

[31] Wenfei Fan and Chao Tian. Incremental graph computations: Doable and undoable. *ACM Transactions on Database Systems (TODS)*, 47(2):1–44, 2022.

[32] Wenfei Fan, Chao Tian, Ruiqi Xu, Qiang Yin, Wenyuan Yu, and Jingren Zhou. Incrementalizing graph algorithms. In *Proceedings of the 2021 International Conference on Management of Data*, pages 459–471, 2021.

[33] Wenfei Fan, Jingbo Xu, Yinghui Wu, Wenyuan Yu, and Jiaxin Jiang. Grape: Parallelizing sequential graph computations. *Proceedings of the VLDB Endowment*, 10(12):1889–1892, 2017.

[34] Guanyu Feng, Zixuan Ma, Daixuan Li, Shengqi Chen, Xiaowei Zhu, Wentao Han, and Wenguang Chen. 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*, pages 513–527, 2021.

[35] Cong Fu, Changxu Wang, and Deng Cai. High dimensional similarity search with satellite system graph: Efficiency, scalability, and unindexed query compatibility. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 44(8):4139–4150, 2021.

[36] Cong Fu, Chao Xiang, Changxu Wang, and Deng Cai. Fast approximate nearest neighbor search with the navigating spreading-out graph. *arXiv preprint arXiv:1707.00143*, 2017.

[37] Chao Gao, Mahbod Afarin, Shafiur Rahman, Nael Abu-Ghazaleh, and Rajiv Gupta. Mega evolving graph accelerator. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '23, page 310–323, New York, NY, USA, 2023. Association for Computing Machinery.

[38] Chao Gao and Sai Qian Zhang. Dlora: Distributed parameter-efficient fine-tuning solution for large language model. *arXiv preprint arXiv:2404.05182*, 2024.

[39] Siddharth Gollapudi, Neel Karia, Varun Sivashankar, Ravishankar Krishnaswamy, Nikit Begwani, Swapnil Raz, Yiyong Lin, Yin Zhang, Neelam Mahapatro, Premkumar Srinivasan, et al. Filtered-diskann: Graph algorithms for approximate nearest neighbor search with filters. In *Proceedings of the ACM Web Conference 2023*, pages 3406–3416, 2023.

[40] Shufeng Gong, Chao Tian, Qiang Yin, Wenyuan Yu, Yanfeng Zhang, Liang Geng, Song Yu, Ge Yu, and Jingren Zhou. Automating incremental graph processing with flexible memoization. *Proceedings of the VLDB Endowment*, 14(9):1613–1625, 2021.

[41] Joseph E Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. PowerGraph: Distributed graph-parallel computation on natural graphs. In *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 17–30, 2012.

[42] Joseph E Gonzalez, Reynold S Xin, Ankur Dave, Daniel Crankshaw, Michael J Franklin, and Ion Stoica. Graphx: Graph processing in a distributed dataflow framework. In *11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14)*, pages 599–613, 2014.

[43] Wentao Han, Youshan Miao, Kaiwei Li, Ming Wu, Fan Yang, Lidong Zhou, Vijayan Prabhakaran, Wenguang Chen, and Enhong Chen. Chronos: a graph engine for temporal graph analysis. In *Proceedings of the Ninth European Conference on Computer Systems*, pages 1–14, 2014.

[44] Zeyu Han, Chao Gao, Jinyang Liu, Sai Qian Zhang, et al. Parameter-efficient fine-tuning for large models: A comprehensive survey. *arXiv preprint arXiv:2403.14608*, 2024.

[45] Ben Harwood and Tom Drummond. Fanng: Fast approximate nearest neighbour graphs. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 5713–5722, 2016.

[46] Thomas Little Heath et al. *The thirteen books of Euclid's Elements*. Courier Corporation, 1956.

[47] Monika Henzinger, Sebastian Krinninger, and Danupon Nanongkai. Sublinear-time decremental algorithms for single-source reachability and shortest paths on directed graphs. In *Proceedings of the forty-sixth annual ACM symposium on Theory of computing*, pages 674–683, 2014.

[48] Monika Henzinger, Ami Paz, and Stefan Schmid. On the complexity of weight-dynamic network algorithms. In *2021 IFIP Networking Conference (IFIP Networking)*, pages 1–9. IEEE, 2021.

[49] Abdullah Al Raqibul Islam, Dong Dai, and Dazhao Cheng. Vcsr: Mutable csr graph format using vertex-centric packed memory array. In *2022 22nd IEEE International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, pages 71–80. IEEE, 2022.

[50] Masajiro Iwasaki and Daisuke Miyazaki. Optimization of indexing based on k-nearest neighbor graph for proximity search in high-dimensional data. *arXiv preprint arXiv:1810.07355*, 2018.

[51] Suhas Jayaram Subramanya, Fnu Devvrit, Harsha Vardhan Simhadri, Ravishankar Krishnawamy, and Rohan Kadekodi. Diskann: Fast accurate billion-point nearest neighbor search on a single node. *Advances in Neural Information Processing Systems*, 32, 2019.

[52] Herve Jegou, Matthijs Douze, and Cordelia Schmid. Product quantization for nearest neighbor search. *IEEE transactions on pattern analysis and machine intelligence*, 33(1):117–128, 2010.

[53] Hervé Jégou, Romain Tavenard, Matthijs Douze, and Laurent Amsaleg. Searching in one billion vectors: re-rank with source coding. In *2011 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 861–864. IEEE, 2011.

[54] Xiaolin Jiang, Mahbod Afarin, Zhijia Zhao, Nael Abu-Ghazaleh, and Rajiv Gupta. Core graph: Exploiting edge centrality to speedup the evaluation of iterative graph queries. In *Proceedings of the Nineteenth European Conference on Computer Systems*, pages 18–32, 2024.

[55] Xiaolin Jiang, Chengshuo Xu, and Rajiv Gupta. VRGQ: Evaluating a stream of iterative graph queries via value reuse. *ACM SIGOPS Operating Systems Review*, 55(1):11–20, 2021.

[56] Xiaolin Jiang, Chengshuo Xu, Xizhe Yin, Zhijia Zhao, and Rajiv Gupta. Tripoline: generalized incremental graph processing via graph triangle inequality. In *Proceedings of the Sixteenth European Conference on Computer Systems*, pages 17–32, 2021.

[57] Wolfgang Kellerer, Patrick Kalmbach, Andreas Blenk, Arsany Basta, Martin Reisslein, and Stefan Schmid. Adaptable and data-driven softwarized networks: Review, opportunities, and challenges. *Proceedings of the IEEE*, 107(4):711–731, 2019.

[58] Pradeep Kumar and H Howie Huang. Graphone: A data store for real-time analytics on evolving graphs. *ACM Transactions on Storage (TOS)*, 15(4):1–40, 2020.

[59] Jérôme Kunegis. Konect: the koblenz network collection. In *Proceedings of the 22nd international conference on World Wide Web*, pages 1343–1350, 2013.

[60] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. What is twitter, a social network or a news media? In *Proceedings of the 19th international conference on World wide web*, pages 591–600, 2010.

[61] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. In *Concurrency: the Works of Leslie Lamport*, pages 179–196. 2019.

[62] Jüri Lember, Dario Gasbarra, Alexey Koloydenko, and Kristi Kuljus. Estimation of viterbi path in bayesian hidden markov models. *Metron*, 77:137–169, 2019.

[63] Jure Leskovec and Andrej Krevl. Snap datasets: Stanford large network dataset collection, 2014.

[64] Michelle M Li, Kexin Huang, and Marinka Zitnik. Graph representation learning in biomedicine and healthcare. *Nature Biomedical Engineering*, 6(12):1353–1369, 2022.

[65] Hang Liu, H Howie Huang, and Yang Hu. iBFS: Concurrent breadth-first search on GPUs. In *Proceedings of the 2016 International Conference on Management of Data*, pages 403–416, 2016.

[66] Yucheng Low, Joseph E Gonzalez, Aapo Kyrola, Danny Bickson, Carlos E Guestrin, and Joseph Hellerstein. GraphLab: A new framework for parallel machine learning. *arXiv preprint arXiv:1408.2041*, 2014.

[67] Dennis Luxen and Christian Vetter. Real-time routing with openstreetmap data. In *Proceedings of the 19th ACM SIGSPATIAL international conference on advances in geographic information systems*, pages 513–516, 2011.

[68] Weimin Lyu, Xinyu Dong, Rachel Wong, Songzhu Zheng, Kayley Abell-Hart, Fusheng Wang, and Chao Chen. A multimodal transformer: Fusing clinical notes with structured ehr data for interpretable in-hospital mortality prediction. In *AMIA Annual Symposium Proceedings*, volume 2022, page 719. American Medical Informatics Association, 2022.

[69] Haixu Ma, Donglin Zeng, and Yufeng Liu. Learning individualized treatment rules with many treatments: A supervised clustering approach using adaptive fusion. *Advances in Neural Information Processing Systems*, 35:15956–15969, 2022.

[70] Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 135–146, 2010.

[71] Yu A Malkov and Dmitry A Yashunin. Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs. *IEEE transactions on pattern analysis and machine intelligence*, 42(4):824–836, 2018.

[72] Magdalen Dobson Manohar, Zheqi Shen, Guy Blelloch, Laxman Dhulipala, Yan Gu, Harsha Vardhan Simhadri, and Yihan Sun. Parlayann: Scalable and deterministic parallel graph-based approximate nearest neighbor search algorithms. In *Proceedings of the 29th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*, pages 270–285, 2024.

[73] Yu Mao, Weilan Wang, Hongchao Du, Nan Guan, and Chun Jason Xue. On the compressibility of quantized large language models. *arXiv preprint arXiv:2403.01384*, 2024.

[74] Mugilan Mariappan, Joanna Che, and Keval Vora. Dzig: sparsity-aware incremental processing of streaming graphs. In *Proceedings of the Sixteenth European Conference on Computer Systems*, pages 83–98, 2021.

[75] Mugilan Mariappan and Keval Vora. Graphbolt: Dependency-driven synchronous processing of streaming graphs. In *Proceedings of the Fourteenth EuroSys Conference 2019*, pages 1–16, 2019.

[76] Abbas Mazloumi, Xiaolin Jiang, and Rajiv Gupta. Multilyra: Scalable distributed evaluation of batches of iterative graph queries. In *2019 IEEE International Conference on Big Data (Big Data)*, pages 349–358. IEEE, 2019.

[77] Abbas Mazloumi, Chengshuo Xu, Zhijia Zhao, and Rajiv Gupta. BEAD: Batched evaluation of iterative graph queries with evolving analytics demands. In *2020 IEEE International Conference on Big Data (Big Data)*, pages 461–468. IEEE, 2020.

[78] Leland McInnes. Pynndescent for fast approximate nearest neighbors. *Webpage. Retrieved December*, 15:2022, 2020.

[79] Frank McSherry, Andrea Lattuada, Malte Schwarzkopf, and Timothy Roscoe. Shared arrangements: practical inter-query sharing for streaming dataflows. *Proceedings of the VLDB Endowment*, 13(10).

[80] Frank McSherry, Derek Gordon Murray, Rebecca Isaacs, and Michael Isard. Differential dataflow. In *CIDR*, 2013.

[81] Duane Merrill, Michael Garland, and Andrew Grimshaw. Scalable gpu graph traversal. *ACM Sigplan Notices*, 47(8):117–128, 2012.

[82] Anurag Mukkara, Nathan Beckmann, Maleen Abeydeera, Xiaosong Ma, and Daniel Sanchez. Exploiting locality in graph analytics through hardware-accelerated traversal scheduling. In *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-51, page 1–14. IEEE Press, 2018.

[83] Javier Vargas Munoz, Marcos A Gonçalves, Zanoni Dias, and Ricardo da S Torres. Hierarchical clustering-based graphs for large scale approximate nearest neighbor search. *Pattern Recognition*, 96:106970, 2019.

[84] Derek G Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. Naiad: a timely dataflow system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 439–455, 2013.

[85] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. A lightweight infrastructure for graph analytics. In *Proceedings of the twenty-fourth ACM symposium on operating systems principles*, pages 456–471, 2013.

[86] Amir Hossein Nodehi Sabet, Junqiao Qiu, and Zhijia Zhao. Tigr: Transforming irregular graphs for gpu-friendly graph processing. *ACM SIGPLAN Notices*, 53(2):622–636, 2018.

[87] Amir Hossein Nodehi Sabet, Junqiao Qiu, and Zhijia Zhao. Tigr: Transforming irregular graphs for gpu-friendly graph processing. *ACM SIGPLAN Notices*, 53(2):622–636, 2018.

[88] Peitian Pan and Chao Li. Congra: Towards efficient processing of concurrent graph queries on shared-memory machines. In *2017 IEEE International Conference on Computer Design (ICCD)*, pages 217–224. IEEE, 2017.

[89] Prashant Pandey, Brian Wheatman, Helen Xu, and Aydin Buluc. Terrace: A hierarchical graph container for skewed dynamic graphs. In *Proceedings of the 2021 International Conference on Management of Data*, pages 1372–1385, 2021.

[90] Zhen Peng, Minjia Zhang, Kai Li, Ruoming Jin, and Bin Ren. iqan: Fast and accurate vector search with efficient intra-query parallelism on multi-core architectures. In *Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*, pages 313–328, 2023.

[91] Joseph Picone. Continuous speech recognition using hidden markov models. *IEEE Assp magazine*, 7(3):26–41, 1990.

[92] Enric Pujol, Ingmar Poese, Johannes Zerwas, Georgios Smaragdakis, and Anja Feldmann. Steering hyper-giants' traffic at scale. In *Proceedings of the 15th International Conference on Emerging Networking Experiments And Technologies*, pages 82–95, 2019.

[93] Zichao Qi, Yanghua Xiao, Bin Shao, and Haixun Wang. Toward a distance oracle for billion-node graphs. *Proceedings of the VLDB Endowment*, 7(1):61–72, 2013.

[94] Shafiur Rahman, Nael Abu-Ghazaleh, and Rajiv Gupta. Graphpulse: An event-driven hardware accelerator for asynchronous graph processing. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 908–921. IEEE, 2020.

[95] Shafiur Rahman, Mahbod Afarin, Nael Abu-Ghazaleh, and Rajiv Gupta. Jetstream: Graph analytics on streaming data with event-driven hardware accelerator. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 1091–1105, 2021.

[96] Ganesan Ramalingam and Thomas Reps. An incremental algorithm for a generalization of the shortest-path problem. *Journal of Algorithms*, 21(2):267–305, 1996.

[97] Amir Raoofy, Roman Karlstetter, Martin Schreiber, Carsten Trinitis, and Martin Schulz. Overcoming weak scaling challenges in tree-based nearest neighbor time series mining. In *International Conference on High Performance Computing*, pages 317–338. Springer, 2023.

[98] Ryan Rossi and Nesreen Ahmed. The network data repository with interactive graph analytics and visualization. In *Twenty-Ninth AAAI Conference on Artificial Intelligence*, 2015.

[99] Amitabha Roy, Laurent Bindschaedler, Jasmina Malicevic, and Willy Zwaenepoel. Chaos: Scale-out graph processing from secondary storage. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 410–424, 2015.

[100] Amitabha Roy, Ivo Mihailovic, and Willy Zwaenepoel. X-stream: Edge-centric graph processing using streaming partitions. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 472–488, 2013.

[101] Kangrui Ruan, Xin He, Jiyang Wang, Xiaozhou Zhou, Helian Feng, and Ali Kebarighotbi. S2e: Towards an end-to-end entity resolution solution from acoustic signal. In *ICASSP 2024-2024 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 10441–10445. IEEE, 2024.

[102] Amir Hossein Nodehi Sabet, Zhijia Zhao, and Rajiv Gupta. Subway: minimizing data transfer during out-of-gpu-memory graph processing. In *Proceedings of the Fifteenth European Conference on Computer Systems*, pages 1–16, 2020.

[103] Amir Hossein Nodehi Sabet, Zhijia Zhao, and Rajiv Gupta. Subway: Minimizing data transfer during out-of-gpu-memory graph processing. In *Proceedings of the Fifteenth European Conference on Computer Systems*, pages 1–16, 2020.

[104] Semih Salihoglu and Jennifer Widom. Gps: A graph processing system. In *Proceedings of the 25th International Conference on Scientific and Statistical Database Management*, pages 1–12, 2013.

[105] Xiaogang Shi, Bin Cui, Yingxia Shao, and Yunhai Tong. Tornado: A system for real-time iterative analysis over evolving data. In *Proceedings of the 2016 International Conference on Management of Data*, pages 417–430, 2016.

[106] Julian Shun and Guy E Blelloch. Ligra: a lightweight graph processing framework for shared memory. In *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 135–146, 2013.

[107] Jeremy Siek, Lie-Quan Lee, Andrew Lumsdaine, et al. *The boost graph library*, volume 243. Pearson India, 2002.

[108] Narayanan Sundaram, Aizana Turmukhametova, Nadathur Satish, Todd Mostak, Piotr Indyk, Samuel Madden, and Pradeep Dubey. Streaming similarity search over one billion tweets using parallel locality-sensitive hashing. *Proceedings of the VLDB Endowment*, 6(14):1930–1941, 2013.

[109] Toyotaro Suzumura, Shunsuke Nishii, and Masaru Ganse. Towards large-scale graph stream processing platform. In *Proceedings of the 23rd International Conference on World Wide Web*, pages 1321–1326, 2014.

[110] Manuel Then, Moritz Kaufmann, Fernando Chirigati, Tuan-Anh Hoang-Vu, Kien Pham, Alfons Kemper, Thomas Neumann, and Huy T. Vo. The more the merrier: Efficient multi-source graph traversal. *Proc. VLDB Endow.*, 8(4):449–460, 2014.

[111] Yijun Tian, Kaiwen Dong, Chunhui Zhang, Chuxu Zhang, and Nitesh V Chawla. Heterogeneous graph masked autoencoders. In *AAAI*, 2023.

[112] Yijun Tian, Huan Song, Zichen Wang, Haozhu Wang, Ziqing Hu, Fang Wang, Nitesh V Chawla, and Panpan Xu. Graph neural prompting with large language models. In *AAAI*, 2024.

[113] William F Tinney and John W Walker. Direct solutions of sparse network equations by optimally ordered triangular factorization. *Proceedings of the IEEE*, 55(11):1801–1809, 1967.

[114] Leslie G Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990.

[115] Alexander van der Grinten, Maria Predari, and Florian Willich. A fast data structure for dynamic graphs based on hash-indexed adjacency blocks. In *20th International Symposium on Experimental Algorithms (SEA 2022)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2022.

[116] Andrew Viterbi. Error bounds for convolutional codes and an asymptotically optimum decoding algorithm. *IEEE transactions on Information Theory*, 13(2):260–269, 1967.

[117] Keval Vora, Rajiv Gupta, and Guoqing Xu. 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*, pages 237–251, 2017.

[118] Keval Vora, Guoqing Xu, and Rajiv Gupta. Load the edges you need: A generic I/O optimization for disk-based graph processing. In *2016 USENIX Annual Technical Conference (USENIX ATC'16)*, pages 507–522, 2016.

[119] Jun Wang, Yufei Cui, Yu Mao, Nan Guan, and Chun Jason Xue. Pre-processing matters: A segment search method for wsi classification. *arXiv preprint arXiv:2404.11161*, 2024.

[120] Kai Wang, Guoqing Xu, Zhendong Su, and Yu David Liu. Graphq: Graph query processing with abstraction refinement—scalable and programmable analytics over very large graphs on a single {PC}. In *2015 USENIX Annual Technical Conference (USENIX ATC'15)*, pages 387–401, 2015.

[121] Shoujin Wang, Liang Hu, Yan Wang, Xiangnan He, Quan Z Sheng, Mehmet A Orgun, Longbing Cao, Francesco Ricci, and Philip S Yu. Graph learning based recommender systems: A review. *arXiv preprint arXiv:2105.06339*, 2021.

[122] Yifan Wang, Haodi Ma, and Daisy Zhe Wang. Lider: an efficient high-dimensional learned index for large-scale dense passage retrieval. *Proceedings of the VLDB Endowment*, 16(2):154–166, 2022.

[123] Yuke Wang, Boyuan Feng, Gushu Li, Shuangchen Li, Lei Deng, Yuan Xie, and Yufei Ding. {GNNAdvisor}: An adaptive and efficient runtime system for {GNN} acceleration on {GPUs}. In *15th USENIX symposium on operating systems design and implementation (OSDI 21)*, pages 515–531, 2021.

[124] Zheng Wang and Jon Crowcroft. Quality-of-service routing for supporting multimedia applications. *IEEE Journal on selected areas in communications*, 14(7):1228–1234, 1996.

[125] Chengshuo Xu, Abbas Mazloumi, Xiaolin Jiang, and Rajiv Gupta. SimGQ: Simultaneously evaluating iterative graph queries. In *2020 IEEE 27th International Conference on High Performance Computing, Data, and Analytics (HiPC)*, pages 1–10. IEEE, 2020.

[126] Chengshuo Xu, Keval Vora, and Rajiv Gupta. 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*, pages 587–600, 2019.

[127] Zexing Xu, Linjun Zhang, Sitan Yang, and Nan Jiang. Peak period demand forecasting with proxy data: Gnn-enhanced meta-learning. In *Proceedings of the IEEE/CVF Winter Conference on Applications of Computer Vision*, pages 726–735, 2024.

[128] Jilong Xue, Zhi Yang, Shian Hou, and Yafei Dai. Processing concurrent graph analytics with decoupled computation model. *IEEE Transactions on Computers*, 66(5):876–890, 2016.

[129] Jilong Xue, Zhi Yang, Zhi Qu, Shian Hou, and Yafei Dai. Seraph: an efficient, low-cost system for concurrent graph processing. In *Proceedings of the 23rd international symposium on High-performance parallel and distributed computing*, pages 227–238, 2014.

[130] Kaige Yang and Laura Toni. Graph-based recommendation system. In *2018 IEEE Global Conference on Signal and Information Processing (GlobalSIP)*, pages 798–802. IEEE, 2018.

[131] Yifan Yang, Zhaoshi Li, Yangdong Deng, Zhiwei Liu, Shouyi Yin, Shaojun Wei, and Leibo Liu. Graphabcd: Scaling out graph analytics with asynchronous block coordinate descent. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 419–432. IEEE, 2020.

[132] Zhaoning Yu and Hongyang Gao. Molecular representation learning via heterogeneous motif graph neural networks. In *International Conference on Machine Learning*, pages 25581–25594. PMLR, 2022.

[133] Pingpeng Yuan, Wenya Zhang, Changfeng Xie, Hai Jin, Ling Liu, and Kisung Lee. Fast iterative graph computation: A path centric approach. SC '14, page 401–412. IEEE Press, 2014.

[134] Yu Zhang, Xiaofei Liao, Hai Jin, Lin Gu, Ligang He, Bingsheng He, and Haikun Liu. Cgraph: A correlations-aware approach for efficient concurrent iterative graph processing. In *2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18)*, pages 441–452, 2018.

[135] Yu Zhang, Xiaofei Liao, Hai Jin, Bingsheng He, Haikun Liu, and Lin Gu. Digraph: An efficient path-based iterative directed graph processing system on multiple gpus. ASPLOS '19, page 601–614, New York, NY, USA, 2019. Association for Computing Machinery.

[136] Yunming Zhang, Ajay Brahmakshatriya, Xinyi Chen, Laxman Dhulipala, Shoaib Kamil, Saman Amarasinghe, and Julian Shun. Optimizing ordered graph algorithms with graphit. *arXiv preprint arXiv:1911.07260*, 2019.

[137] Yunming Zhang, Mengjiao Yang, Riyadh Baghdadi, Shoaib Kamil, Julian Shun, and Saman Amarasinghe. Graphit: A high-performance graph dsl. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA):1–30, 2018.

[138] Jin Zhao, Yu Zhang, Xiaofei Liao, Ligang He, Bingsheng He, Hai Jin, and Haikun Liu. Lccg: a locality-centric hardware accelerator for high throughput of concurrent graph processing. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–14, 2021.

[139] Jin Zhao, Yu Zhang, Xiaofei Liao, Ligang He, Bingsheng He, Hai Jin, Haikun Liu, and Yicheng Chen. GraphM: an efficient storage system for high throughput of concurrent graph processing. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–14, 2019.

[140] Xiaowei Zhu, Wenguang Chen, Weimin Zheng, and Xiaosong Ma. Gemini: A computation-centric distributed graph processing system. In *12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16)*, pages 301–316, 2016.

[141] Xiaowei Zhu, Guanyu Feng, Marco Serafini, Xiaosong Ma, Jiping Yu, Lei Xie, Ashraf Aboulnaga, and Wenguang Chen. Livegraph: A transactional graph storage system with purely sequential adjacency list scans. *Proceedings of the VLDB Endowment*, 13(7).

[142] Xiaowei Zhu, Wentao Han, and Wenguang Chen. {GridGraph}:{Large-Scale} graph processing on a single machine using 2-level hierarchical partitioning. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pages 375–386, 2015.

[143] Jun Zhuang and Mohammad Al Hasan. Defending graph convolutional networks against dynamic graph perturbations via bayesian self-supervision. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 36, pages 4405–4413, 2022.

[144] Jun Zhuang and Mohammad Al Hasan. Robust node classification on graphs: Jointly from bayesian label transition and topology-based label propagation. In *Proceedings of the 31st ACM International Conference on Information & Knowledge Management*, pages 2795–2805, 2022.

[145] Jun Zhuang and Mohammad Al Hasan. How does bayesian noisy self-supervision defend graph convolutional networks? *Neural Processing Letters*, 54(4):2997–3018, 2022.