UVVs: Identifying Unchanged Vertex Values in Evolving Graphs via Intersection-Union Analysis

Mahbod Afarin* *CSE Department* *UC Riverside* Riverside, USA mafar001@ucr.edu
Chao Gao* *CSE Department* *UC Riverside* Riverside, USA cgao037@ucr.edu
Xizhe Yin *CSE Department* *UC Riverside* Riverside, USA xyin014@ucr.edu
Zhijia Zhao *CSE Department* *UC Riverside* Riverside, USA zhijia@cs.ucr.edu
Nael Abu-Ghazaleh *CSE Department* *UC Riverside* Riverside, USA nael@cs.ucr.edu
Rajiv Gupta *CSE Department* *UC Riverside* Riverside, USA rajivg@ucr.edu

***Abstract*—Evaluating a query over a large, irregular graph is inherently challenging. This challenge intensifies when solving a query over a sequence of snapshots of an evolving graph, where changes occur through the addition and deletion of edges. We carried out a study that shows that due to the gradually changing nature of evolving graphs, when a vertex-specific query (e.g., SSSP) is evaluated over a sequence of 25 to 100 snapshots, for 67.8% to 99.8% of vertices, the query results remain unchanged across all snapshots. Therefore, the *Unchanged Vertex Values* (UVVs) can be computed once and then minimal analysis can be performed for each snapshot to obtain the results for the remaining vertices in that snapshot. We develop a novel *intersection-union analysis* that accurately computes lower and upper bounds of vertex values across all snapshots. When the lower and upper bounds for a vertex are found to be equal, we can safely conclude that the value found for the vertex remains the same across all snapshots. Therefore, the rest of our query evaluation is limited to computing values across snapshots for vertices whose bounds do not match. We optimize this latter step evaluation by concurrently performing incremental computations on all snapshots over a significantly smaller subgraph. Our experiments with several benchmarks and graphs show that we need to carry out per snapshot incremental analysis for under 42% of vertices on a graph with under 32% of edges. Our approach delivers speedups of 2.01-12.23× compared to the state-of-the-art RisGraph implementation of the KickStarter-based incremental algorithm for 64 snapshots.**

I. Introduction

Graph analytics are employed in many domains to uncover insights from connected data. There has been much work resulting in scalable graph analytics systems for GPUs, multicore servers, and clusters [39], [38], [19], [48], [55], [58], [68], [35], [28], [27], [50], [60], [11], [22], [45], [4], [16], [41], [25], [2]. Most real-world graphs change dynamically over time [53]. Therefore, there has recently been a great deal of interest in analytics over changing graphs [57], [40], [15], [3], [24], [56], [20]. Efficient dynamic graph processing has diverse applications, including social network analysis for community detection and influence propagation [5], [18], personalized recommendation systems [8], [30], and telecommunication networks for traffic management and fault detection [49], [36]. It is also crucial in financial networks for fraud detection and risk assessment [7], [32], biological networks like gene regulatory networks [37], [64], and transportation networks for traffic flow optimization [59], [66]. In e-commerce, it aids in customer interaction analysis and supply chain management [61], [47], while in cybersecurity, it enhances intrusion detection and network defense [6], [17]. Additionally, smart cities benefit from urban planning and resource management applications [10], [46], and healthcare uses include epidemic tracking and patient monitoring [21], [52]. These applications underscore the importance of dynamic graph processing.

As a graph continues to evolve, a sequence of snapshots is captured which grows in length over time. An *evolving graph query* is aimed at analyzing the evolution of a graph property (e.g., shortest paths) over a time window. That is, an evolving graph query typically requires a graph query to be solved over a sequence of snapshots $G_0, G_1, \ldots, G_n$. By selecting the time window, the user requests query evaluation for all snapshots within a time window to observe trends in query results (e.g., changes in shortest paths). As the duration of the time window for query evaluation increases, the number of snapshots that must be analyzed also increases, and hence the cost of query evaluation rises. Thus, efficiently evaluating a query over many snapshots is an important open problem.

Existing Approaches. To reduce the high cost of query evaluation over a large number of snapshots, existing approaches such as Tegra [24] and CommonGraph [3] leverage incremental algorithms. A general incremental algorithm that supports both edge additions and deletions was first proposed in KickStarter [57] and then further extended and optimized by Graphbolt [40] and RisGraph [15] respectively. While both Tegra and CommonGraph employ incremental algorithms, there is a major difference. Tegra explicitly processes both additions and deletions using incremental algorithms developed in KickStarter [57] and Graphbolt [40]. In contrast, CommonGraph [3] converts edge deletions between snapshots into edge additions between a common graph and each snapshot, thus trading expensive deletion processing for relatively inexpensive addition processing. The common graph is first used to evaluate a query. Starting from the common graph and query results computed for it, edge additions are processed to incrementally compute query results for each snapshot.

*Both authors contributed equally to this research.

1530-2075/26/$31.00 ©2026 IEEE
DOI 10.1109/IPDPS65963.2026.00021

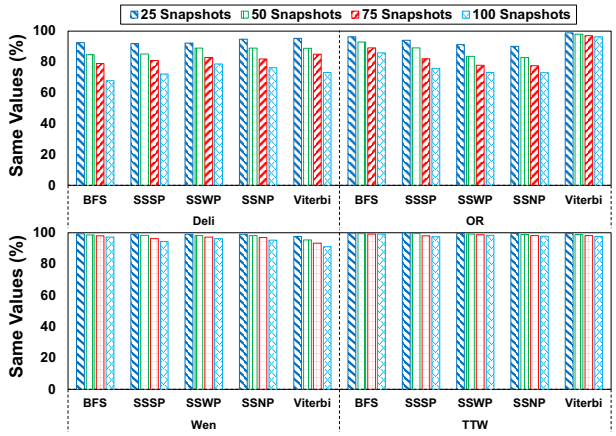


Fig. 1: Given series of 100 snapshots obtained by performing 100K edge updates (50% deletions and 50% additions) from one snapshot to the next, the above plot gives the percentage of vertex property values that remain unchanged across 25, 50, 75, and 100 snapshots for 4 input graphs and 5 benchmarks. 100K edges represent between 0.1% of edges in Deli to 0.025% for Wen. The plots for the LJ and Fr graphs exhibit similar trends to those shown here.

Our Insight – UVVs. We develop a new insight and a novel algorithm to take advantage of the insight to substantially optimize query evaluation across multiple snapshots.

Our key insight originates from the gradually changing nature of an evolving graph. From the motivating data in Figure 1, we find that even when considering a fairly large number of snapshots, for most input graphs and algorithms, a large percentage of vertex property values remain the same across snapshots in a typical time window of interest. For example, consider that the number of LinkedIn users has been increasing at around 1% per month between July 2020 and July 2022 [23]. A query that analyzes snapshots every day corresponds roughly to a change of 0.03% of the graph between snapshots, similar to the percentage of changes in Figure 1. From the results of a study presented in Figure 1, we observe that, **given a vertex-specific query (e.g., SSSP), the query results for 67.8% to 99.8% of vertices remain unchanged across 25 to 100 consecutive snapshots.** Therefore, the *unchanged vertex values* (UVVs) can be computed once, and then minimal analysis can be performed for each snapshot to obtain the results for the remaining vertices in that snapshot. Moreover, the incremental computation can be performed over a much *smaller graph* obtained by eliminating the incoming edges of all vertices with unchanged values – since no updates need to be applied to their values, the incoming edges play no role in the incremental computation for any snapshot.

Our Solution. To take advantage of the above insight, and resulting opportunities for optimizing evaluation of an evolving graph query, we need to address the following key challenge. Given a vertex-specific query Q and a sequence of snapshots, we must **identify vertices with unchanged vertex values** (UVVs). To address this challenge, we develop a novel **intersection-union analysis** to compute lower and upper bounds of a vertex value across all snapshots. When the bounds are found to be equal, we can **safely** conclude that the vertex value found remains the same across all the provided snapshots. Therefore, the rest of our query evaluation is limited to computing values across snapshots for vertices whose bounds were not equal. Our approach is applicable to path-based monotonic algorithms. A path-based monotonic algorithm incrementally explores or constructs paths within a graph while ensuring that the computed metric along each path (e.g., distance, cost) adheres to a monotonic property, such as non-increasing or non-decreasing values. As the algorithm progresses, it extends partial paths in such a way that the solution's quality improves or remains constant, ensuring convergence to an optimal or sub-optimal solution without regressing to a worse state.

We also optimize evaluation of query Q by *concurrently* performing incremental computations for all snapshots over a much smaller graph that we refer to as the Q-Relevant Sub-Graph (QRS). The smaller graph is obtained by eliminating the incoming edges of all vertices with unchanged values.

Our experiments on several benchmarks and graphs over 64 snapshots show that per-snapshot incremental analysis is needed for under 42% of vertices and 32% of edges. When we incrementally evaluate the query on each snapshot using QRS, the cost of evaluation is lowered. Our approach delivers speedups of up to 12.23× over the state-of-the-art RisGraph implementation of the KickStarter-based incremental algorithm for 64 snapshots.

II. Background

An evolving graph consists of a series of snapshots G_0, $G_1 \cdots G_n$ of a graph captured over time. In evolving graph analytics, we solve a query over a time window during which graph evolves. Multiple snapshots of the graph at different points in time in the time window are given, and query results must be computed for all these snapshots. Evolving graph analytics is motivated by a user's need to observe trends in a graph property. A naive approach for evaluating a query over all snapshots, shown in Figure 2(a), evaluates a query from scratch on each snapshot. To overcome the inefficiency of this approach, *incremental approaches* were proposed: the KickStarter-based streaming approach; and the Common Graph deletion-free approach.

A. KickStarter-based Incremental Approach

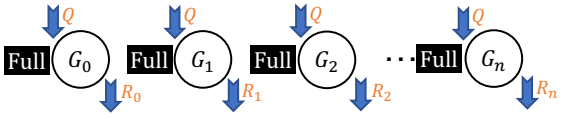
Without loss in generality, we assume that all vertices are present in all snapshots and the changes from one snapshot to the next are represented in the form of additions and deletions of the edges applied to an earlier snapshot that produces the next snapshot. The batches of edges, including additions and deletions, are denoted as $\delta_1, \delta_2 \cdots \delta_n$ in Figure 2(b). The KickStarter-based [57] incremental approach evaluates the query of the first snapshot G_0 from scratch and then

incrementally processes δ_1 through δ_n in turn to obtain query results for G_1 through G_n as shown in Figure 2(b).

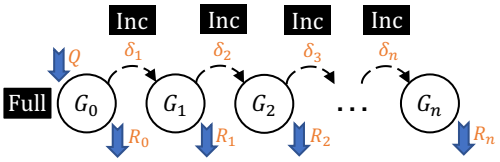
B. Common Graph Deletion-Free Approach

Past work on JetStream [51] has shown that incremental processing for an edge deletion operation is significantly more expensive than the edge addition operation for monotonic graph queries. Therefore, the Common Graph approach was proposed to avoid both redundant computation and expensive handling of deletions (see Figure 2(c)). Common Graph is the subgraph that is shared by all the snapshots under consideration. Therefore, solving the query on it, and then streaming different batches of edge additions enables incrementally computing the query on each snapshot without having to explicitly deal with edge deletions.

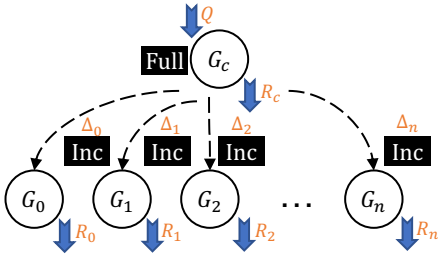
Common Graph is the subgraph that is common to all snapshots of the evolving graph. Therefore, each snapshot can be obtained by simply adding an appropriate subset of edges to the Common Graph, that is, no edge deletions are required to convert the Common Graph to any snapshot. After computing the query on this Common Graph, by adding the missing edges



(a) Naive Technique: Full Computation (**Full**) on each snapshot of the graph (G_0, G_1, ..., G_n) and independently calculate the results for each snapshot from scratch.



(b) KickStarter-based Incremental: Full Computation (**Full**) on the first snapshot of the graph (G_0), and Incrementally (**Inc**) apply the delta batches (δ_1, δ_2, ..., δ_n) to find the results for each snapshot in ($G_0 \rightarrow G_1 \rightarrow \cdots G_n$).



(c) Deletion-Free Common Graph: Full Computation (**Full**) on the Common Graph and Incrementally (**Inc**) add the delta batches (Δ_0, Δ_1, ..., Δ_n) to the Common Graph to find the results for each snapshot (G_0, G_1, ..., G_n).

Fig. 2: Strategies for Evolving Graph Query Evaluation.

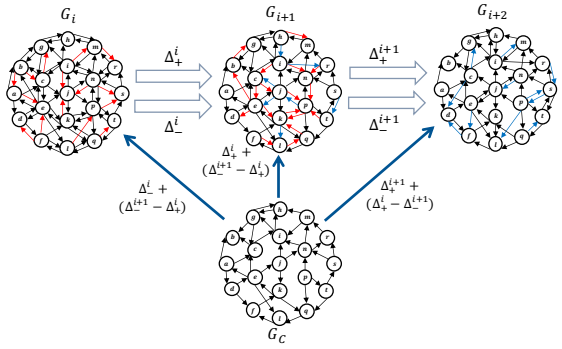


Fig. 3: Common Graph G_C of snapshots G_i, G_{i+1}, G_{i+2}.

for a snapshot and incrementally updating the query results in response to the additions, the query results for the snapshot are obtained. This is called the *direct hop* approach. If multiple snapshots require many common edges to be added to obtain them from the Common Graph, then we can also carry out the additions in stages to share incremental subcomputations among subsets of snapshots. This is called the *work sharing*.

Figure 3 shows the Common Graph G_C for three snapshots G_i, G_{i+1}, and G_{i+2}. We add Δ_+^i and remove Δ_-^i edges to incrementally derive G_{i+1} from G_i. Similarly, we can derive G_{i+2} from G_{i+1} by respectively adding and deleting the delta batches of edges. The *Common Graph* for the three snapshots, G_C, is also shown. The *direct hop* approach adds different subsets of edges to the Common Graph to derive the three snapshots as shown in the figure. For example, to derive G_i we should combine three batches of edges (Δ_-^i, Δ_-^{i+1}, and Δ_-^{i+2}) and apply them once to G_C. The main strength of the *direct hop* workflow is that we can add all the addition delta batches independently and find all the snapshots in parallel. The main limitation of the *direct hop* is the redundant addition operations. For example, to derive G_i and G_{i+1} we must add Δ_-^{i+1} and Δ_-^{i+2} twice to G_C. Therefore, *work sharing* was proposed to further reduce redundant additions.

Though Common Graph provides significant speedups over KickStarter-based method [3], its scalability is limited. Therefore, we argue that to further optimize performance, it is essential to **eliminate wasteful work on analyzing UVV vertices and traversing incoming edges of UVV vertices in the Common Graph**. In subsequent sections, we demonstrate how to identify UVVs and exploit them to improve scalability.

III. Our Approach Based Upon Identifying Unchanged Vertex Values (UVVs)

From our motivating study, we observed that the query results computed for different snapshots are substantially the same, i.e., the addition and deletion of edges frequently cause changes to property values of a small subset of vertices. In our discussion, we use UVVs to refer to vertices with Unchanged Vertex Values. The example in Figure 4 illustrates the presence of UVVs – 6 of the 10 total shortest path values computed from source vertex s are the same for the two snapshots; these are the ones marked green.

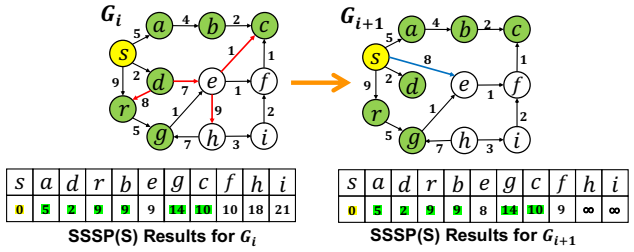


Fig. 4: Results of query SSSP(s) for two consecutive snapshots. The second snapshot is obtained by deleting red edges from the first snapshot and also adding the blue edges to the second snapshot. Note that the shortest path values for vertices marked in green are identical for both snapshots.

Put differently, we observe that, for a specific query Q, there are many vertices whose property values remain unchanged across all snapshots.

This observation motivates us to identify UVVs and then use them to eliminate wasteful work performed during incremental computations, including computations that attempt to update UVV vertices and edge traversals that lead to UVV vertices. By identifying UVVs and shrinking the size of the graph over which incremental computations for a given query Q are performed, we will affect this optimization. The reduced graph is named Q-Relevant Subgraph (QRS) for query Q. Next we describe our algorithm details and illustrate via an example.

Let us consider the shortest path query in this discussion, though our approach is applicable to other vertex specific path-based monotonic algorithms. Furthermore, without loss of generality, assume that all vertices are present in all snapshots. Only the set of edges present differs across the snapshots due to batches of updates in the form of edge additions and deletions that are performed as the graph evolves.

Our approach for identifying UVVs and generating the query specific Q-Relevant Subgraph is as follows.

a) **Step 1***: Bounding Vertex Values for a Query:* Let E_0, E_1, $\cdots$ E_n denote the sets of edges corresponding to the evolving graph's snapshots G_0, G_1, $\cdots$ G_n. We consider two graphs that are derived from the above snapshots as follows:

- **Intersection Graph** $G_\cap$: This is the graph that contains edges that are common to all the snapshots, i.e., $E_\cap = E_0 \cap \cdots \cap E_n$.
- **Union Graph** $G_\cup$: This is the graph that contains all edges present across all the snapshots, i.e., $E_\cup = E_0 \cup \cdots \cup E_n$.

We evaluate the shortest path query for source vertex s on both $G_\cap$ and $G_\cup$. Let us denote the shortest path value computed for some vertex v corresponding to $G_\cap$ and $G_\cup$ by $Val_\cap(s,v)$ and $Val_\cup(s,v)$. The following theorem captures the relationship between the shortest path value of v for any snapshot G_i.

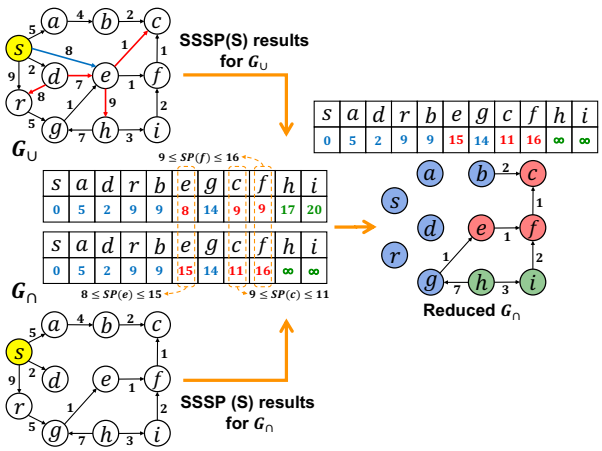


Fig. 5: The Union Graph $G_\cup$ provides upperbounds on path lengths across all snapshots while the Intersection Graph $G_\cap$ provides the lowerbounds. Query Relevant Graph obtained by reducing $G_\cap$ and the query results used to bootstrap incremental computations.

Theorem 1. Given source vertex s, the shortest path values from s to vertex v for $G_\cap$ and $G_\cup$, that is, $Val_\cap(s,v)$ and $Val_\cup(s,v)$, represent the *upperbound* and *lowerbound* over the shortest path value of vertex v across all snapshots G_0, G_1 $\cdots$ G_n.

Proof. We observe that the Intersection Graph $G_\cap$ contains only a subset of paths from any snapshot G_i because $E_\cap = E_0 \cap \cdots \cap E_n$. Therefore, the shortest path value of vertex v corresponding to snapshot G_i, denoted by $Val_i(s,v)$, is bounded by $Val_\cap(s,v)$ as follows:

$$Val_i(s,v) \le Val_\cap(s,v)$$

Similarly, we observe that the Union Graph $G_\cup$ contains a superset of paths from any snapshot G_i because $E_\cup = E_0 \cup \cdots \cup E_n$. Therefore, the shortest path value of vertex v corresponding to snapshot G_i, denoted by $Val_i(s,v)$, is bounded $Val_\cup(s,v)$ as follows:

$$Val_\cup(s,v) \le Val_i(s,v)$$

Therefore, we conclude the following:

$$Val_\cup(s,v) \le Val_i(s,v) \le Val_\cap(s,v)$$

Note that the above result holds even when vertex v is not reachable from s in G_i or $G_\cap$ (i.e., shortest path value is ∞).

Finally, in an evolving graph, an edge between a pair of nodes may be added and deleted a number of times. This type of edge will not be present in $G_\cap$ since it is not present in all the snapshots. However, it will be present in $G_\cup$. The weight of this edge can be set to the minimum of all weights encountered for this edge to obtain a safe lowerbound.

The results of computing the lower and upper bounds for our SSSP example using the intersection and union graphs are shown in Figure 5. While in our discussion we presented the upper and lower bounds for $Val_i(s,v)$ for the SSSP algorithm,

TABLE I: Upper and Lower bounds for different algorithms.

Alg.	Upperbound and Lowerbound for $Val_i(s,v)$
BFS	$Val_{\cup}(s,v) \leq Val_i(s,v) \leq Val_{\cap}(s,v)$
SSWP	$Val_{\cap}(s,v) \leq Val_i(s,v) \leq Val_{\cup}(s,v)$
SSNP	$Val_{\cup}(s,v) \leq Val_i(s,v) \leq Val_{\cap}(s,v)$
SSSP	$Val_{\cup}(s,v) \leq Val_i(s,v) \leq Val_{\cap}(s,v)$
Viterbi	$Val_{\cap}(s,v) \leq Val_i(s,v) \leq Val_{\cup}(s,v)$

in Table I we present the bounds for various benchmarks that we use in our evaluation (see Table II). The upper and lower bounds for the SSSP, SSNP, and BFS algorithms are similar because we take the minimum of all possible results for a node – the intersection graph gives us the upperbound and the union graph the lowerbound. On the other hand, for the SSWP and Viterbi algorithms, since we take the maximum of all possible results, the union graph gives us the upperbound, and the intersection graph gives us the lowerbound.

b) **Step 2***: Identifying UVVs – Vertices Whose Values Remain the Same Across All Snapshots:* So far we have observed that by solving a shortest path query on both $G_{\cup}$ and $G_{\cap}$ we can bound the path lengths for any vertex across all snapshots. This is illustrated in the example shown in Figure 5. It is interesting to note that for many vertices, the lowerbound and upperbound match precisely. This implies that the **shortest path lengths for these vertices remain unchanged across all snapshots**. Consequently, we already have their results, and now we need to perform incremental computations to update the results of only a subset of vertices in each snapshot.

Theorem 2. Given a shortest path query with source vertex s and some vertex v:

$$if \;\; Val_{\cup}(s,v) = Val_{\cap}(s,v) \;\; then$$

$$\forall i \; [0 \ldots n] \;\; Val_i(s,v) = Val_{\cup}(s,v) = Val_{\cap}(s,v),$$

i.e., the query result value for vertex v is the same for all snapshots and equal to $Val_{\cup}(s,v)$ (or $Val_{\cap}(s,v)$).

Proof. Since $Val_{\cup}(s,v) = Val_{\cap}(s,v)$, the shortest path from s to v of the same length is present both in $G_{\cap}$ and $G_{\cup}$. Moreover, the presence of shortest path in $G_{\cap}$ implies that it is also present in all the snapshots because $G_{\cap}$ is the subgraph of each snapshot. Furthermore, there cannot be any path from s to v in any snapshot G_i that is of a shorter path length than $Val_{\cap}(s,v)$ even though G_i contains edges that are not present in $G_{\cap}$. This is because if such a path existed, it would also be present in $G_{\cup}$ and that would contradict the fact that $Val_{\cup}(s,v) = Val_{\cap}(s,v)$.

Note that when $Val_{\cup}(s,v) = Val_{\cap}(s,v)$ we have already found the shortest path value from s to v for all snapshots. However, when $Val_{\cup}(s,v) \neq Val_{\cap}(s,v)$, it does not imply that $Val_i(s,v)$ cannot be the same for all snapshots.

Our algorithm is safe but not complete, that is, for a given query it does not identify all vertices for which the shortest path value remains the same across all snapshots. In Figure 5 we could not identify that its value remains unchanged. This is because neither $G_{\cap}$ nor $G_{\cup}$ provide the value 10, rather they provided values 11 and 9. Yet, in Figure 6, note that the shortest path value for vertex c is 10 in both snapshots.

c) **Step 3***: Deriving Q-Relevant Subgraph:* Before performing incremental computations, we can substantially reduce the size of the Intersection Graph as follows.

> For each vertex whose property value has already been determined, that is, its lowerbound and upperbound are found to be equal, the set of its incoming edges can be removed from the graph.

This is because for the vertices whose results are already known, no vertex updates are needed, and the incoming edges that are responsible for those updates can be safely eliminated. In Figure 5, we show the resulting Q-Relevant Subgraph obtained after reducing the Intersection Graph $G_{\cap}$ by eliminating incoming edges of vertices with precise vertex values. The shortest path values of vertices that bootstrap the next incremental computation phase correspond to the shortest path values obtained by solving the query on $G_{\cap}$.

d) **Step 4***: Incremental Computations for Snapshots:* Starting from the Q-Relevant Subgraph, and initial results computed from the Intersection Graph, we perform incremental computations to obtain precise results for both snapshots (see Figure 6). Note that when a batch of additions is used for incremental computation, the batch can also be further reduced by eliminating the edges whose sink is a node with already known precise solution. In Figure 6, the edge from d to r need not be streamed for snapshot G_i because the value for vertex r is already precise and the same in both snapshots.

In conclusion, while our algorithm is safe, it does not identify all vertices whose value remains the same across all snapshots. However, as the experimental results presented later in the paper will demonstrate, our algorithm is highly effective as it identifies nearly all the vertices whose property value remains unchanged across all snapshots.

e) Algorithm Summary: Algorithm 1 shows how the QRS is found. The inputs to the algorithm are the Intersection Graph ($G_{\cap}$) and the Union Graph ($G_{\cup}$) of all the snapshots (line 1), the delta batches, which must be added to the Intersection Graph ($G_{\cap}$) to obtain the results for each snapshot of the graph. The number of delta batches equals the number of snapshots (Δ_0, Δ_1, ..., Δ_n). Additionally, we should specify the query (Q) as an input to the algorithm. The output of the algorithm is the results of the query evaluation for each snapshot of the graph (line 3). Therefore, we have n arrays to represent results of each snapshot, R_0, R_1, ..., R_n.

The first step of the algorithm is to compute the results of solving the query on the Intersection Graph ($G_{\cap}$) and the Union Graph ($G_{\cup}$). Therefore, we should define two result arrays, $R_{\cap}$ and $R_{\cup}$, to store the outcomes of the query evaluation on each graph. The size of these two arrays is proportional to the number of vertices in the graph, which is m (lines 4-6). The COMPUTE (Graph G, Query Q) function will evaluate the query Q on the graph G (lines 7-8). In the second step of the algorithm, we need to compare the results

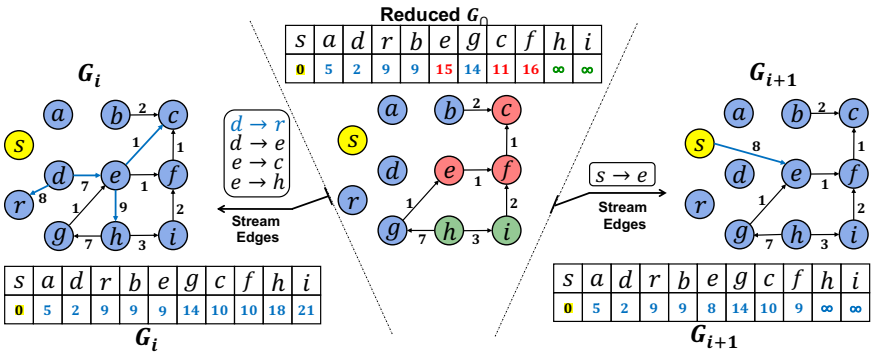


Fig. 6: Carrying out incremental computations via edge additions to obtain final query results for each snapshot. Note that for vertex c, the final query result is the same (10) for both snapshots while its lower and upper bounds were 9 and 11.

Algorithm 1 Finding Q-Relevant Subgraph

```
1: Inputs: Intersection Graph (G∩); Union Graph (G∪);
2: Addition Edges Batches (Δ0, Δ1, ..., Δn); Query (Q).
3: Output: Results for Solving Query (Q) on Each Snapshot of the
   Graphs (R0, R1, ..., Rn).
4: Compute Q on G∩ and G∪ (m is number of vertices)
5: R∩[1, m]: array results for G∩
6: R∪[1, m]: array results for G∪
7: R∩[1, m] ← COMPUTE (G∩, Q)
8: R∪[1, m] ← COMPUTE (G∪, Q)
9: Find Values That Are Same On Both G∩ and G∪
10: found: set for storing vertices with precise values
11: for each i ∈ [1, m] do
12:     if R∩[i] == R∪[i] then
13:         found ← insert(i)
14:     end if
15: end for
16: Reduce the size of G∩ and delta batches and Find GQRS
17: for each v ∈ found do
18:     REMOVEINCOMINGEDGES(G∩, v)
19:     REMOVEDELTAADDITIONBATCHES(v)
20: end for
21: GQRS ← G∩
22: Add the Batches to GQRS and Find the Results
23: for each i ∈ [0, n] do
24:     Ri ← INCREMENT (GQRS, Δi)
25: end for
26: Function for removing the incoming edges
27: function REMOVEINCOMINGEDGES(Graph G, Vertex v)
28:     for each x ∈ G[v].inNeighbors do
29:         remove edge (x, v)
30:     end for
31: end function
32: Function for removing edges from delta batches
33: function REMOVEDELTAADDITIONBATCHES(Vertex v)
34:     for each i ∈ [0, n] do
35:         for each edge (u, x) ∈ Δi do
36:             if x == v then
37:                 remove edge (u, v)
38:             end if
39:         end for
40:     end for
41: end function
```

from the two value arrays $R_\cap$ and $R_\cup$. Therefore, we should use a for loop to iterate over each element of these arrays. If an element is the same in both $R_\cap$ and $R_\cup$, we should mark the vertex and add it to a set named $found$ (lines 9-15).

Next, we should reduce the size of the Intersection Graph ($G_\cap$) using the $found$ set. Currently, the $found$ set consists of all the vertices with the same value across all the snapshots. Therefore, we should remove the incoming edges of those vertices that are in the $found$ set using the REMOVEINCOMINGEDGES function. We should also remove the edges in the delta batches that have the same destination as those in the $found$ set using the REMOVEDELTAADDITION function (lines 16-21). Then, we can rename the reduced $G_\cap$ graph to G_{QRS}. Finally, we should incrementally add the reduced-size delta batches to the Q-Relevant Subgraph (G_{QRS}) to determine results of query Q on each snapshot (lines 22-25). The function REMOVEINCOMINGEDGES(Graph G, Vertex v) has two inputs: Graph G and Vertex v. It iterates over each vertex in G, and if there is an edge leading to vertex v, it removes that edge (lines 26-31). The function REMOVEDELTAADDITION(Vertex v) takes a vertex as its input and iterates over all the edges in the delta batches. If it finds edge to v, it removes that edge (lines 32-41).

IV. CONCURRENT INCREMENTAL COMPUTATIONS

Starting from the Q-Relevant Subgraph G_{QRS}, the query results for each snapshot can be found by incremental computation of the addition batch (Δ_i). One way to calculate query results for all snapshots is through a sequence of snapshot evaluations, i.e., n rounds of incremental computation ($INCREMENT(G_{QRS}, \Delta'_i)$, where $i = 0, ..., n$ and Δ'_i is the reduced batch corresponding to Δ_i). However, this approach has two issues: 1) resource underutilization and 2) data locality is not fully exploited. First, incremental computation (especially for edge additions) is lightweight in comparison to full query evaluation, potentially leading to underutilization of machine resources. Second, identical edges, whether in G_{QRS} or Δ'_i, may be traversed multiple times across snapshots, degrading cache locality.

Instead of evaluating snapshots one by one, we propose concurrent evaluation. An augmented graph with versioning and *snapshot-oblivious* frontier are used for efficiency.

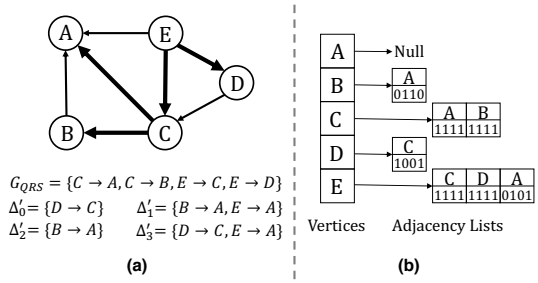


Fig. 7: (a) Versioned Graph; and (b) the augmented adjacency list (only lists for out-going edges are shown; bold edges are G_{QRS} edges; and there are 4 snapshots)

A. Versioned Graph Representation

We augment the Q-Relevant Subgraph G_{QRS} with extra edge versioning information to show which snapshots an edge belongs to. A 64-bit variable is used for storing such version information of an edge (more bits can be added for supporting greater than 64 snapshots). The bit at each location indicates if the edge is present in the corresponding snapshot. For edges that are common to all snapshots, their version labels are all 1s (i.e., `1111....1111`).

The augmented versioned graph has more edges than the Q-Relevant Subgraph – edges in G_{QRS} plus reduced addition batches ($G_{QRS} \cup \Delta'_0 \cup \Delta'_1 \cup ...\Delta'_n$). Since G_{QRS} is reduced from the Intersection Graph ($G_\cap$), it contains a subset of edges in $G_\cap$ that are common to all snapshots. Those common edges are stored at the beginning of the adjacency lists, followed by snapshot-specific edges.

An augmented graph is shown in Figure 7. G_{QRS} has four edges (edges in bold) that have `1111`s as their version value in the adjacency lists. Four graph snapshots are embedded into the augmented graph, which can be obtained by adding Δ'_0 through Δ'_3 to G_{QRS}, respectively. Edge $\langle D \to C \rangle$ is present in both snapshots 0 and 3, so its version number is `1001`. Edges $\langle E \to C \rangle$ and $\langle E \to D \rangle$ are common to all snapshots.

B. Concurrent Snapshot Evaluation

Now we describe how multiple snapshots are evaluated concurrently. In traditional graph query evaluation, the out-going edges of active vertices are evaluated by the edge function and vertices that have their values changed will be put into the frontier. In the concurrent snapshot evaluation, there are two aspects to consider: 1) the ownership of edges that must be checked when traversing the versioned graph; and 2) the ownership of active vertices that help distinguish which vertex is active for which snapshot. The edge ownership cannot be neglected as it affects the correctness of concurrent snapshot evaluation. We show that the ownership of active vertices can be further relaxed for better performance. In a basic concurrent evaluation design, it is intuitive to maintain a separate frontier for each snapshot since an active vertex may not be active for all snapshots; however, this introduces extra overhead due to the maintenance and access of multiple

Algorithm 2 Concurrent Evaluation of Snapshots

```
1:  function BATCHEVALUATION(G, n, Δ[...], f)
2:      (F_so, F_next) = (∅, ∅)  Snapshot-oblivious frontier
3:      R_{i∈[0:n−1]} = R_QRS  Initialize results for each snapshot
4:      parfor Δ_i in Δ[...] do  Processing addition batches
5:          parfor ⟨u → v, w⟩ in Δ_i do
6:              if f(⟨u → v, w⟩) improves R_i[v] then
7:                  R_i[v]=f(⟨u → v, w⟩)
8:                  F_so=F_so ∪ {v}
        Concurrent snapshot evaluation
9:      while F_so ≠ ∅ do
10:         parfor v in F_so do
11:             for x in v's out-going neighbors do
12:                 for i from 0 to n − 1 do
                        Check if the edge belongs to snapshot i
13:                     if snapshotHasEdge(i,⟨v → x⟩) then
14:                         if f(⟨v → x, w⟩) improves R_i[x] then
15:                             R_i[x]=f(⟨v → x, w⟩)
16:                             F_next=F_next ∪ {x}
17:         swap(F_next, F_so)
18:         F_next = ∅
19:     end while
20:     return R_{i∈[0:n−1]}
21: end function
```

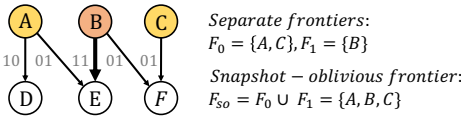


Fig. 8: *Snapshot-oblivious frontier* and concurrent snapshot traversal.

frontiers. Instead, `UVVs` employs a design called *snapshot-oblivious* frontier, inspired by recent works on concurrent graph query processing [62], [42]. The *snapshot-oblivious* frontier does not distinguish which vertex is active for which snapshot; given a batch of snapshots, it simply treats the vertex active for all snapshots by using a single frontier, which is the union of all separate frontiers. The correctness of *snapshot-oblivious* frontier is guaranteed by the monotonic property of graph algorithms.

Figure 8 shows an example of *snapshot-oblivious* frontier and concurrent snapshot traversal. Two snapshots are considered in this case with their frontiers F_0 and F_1. The out-going edges of active vertices in F_0 and F_1 have their ownership checked before the edge function can be applied. For example, active vertex A's out-going edge $\langle A \to E \rangle$ that is owned by snapshot S_0 (the version label is `01`) will be evaluated for updating the results of S_0, while $\langle A \to D \rangle$ will not be evaluated for S_0 as it only belongs to S_1. Regarding the ownership of active vertices, because of the use of *snapshot-oblivious* frontier F_{so}, vertex B's out-going edges are blindly evaluated for both snapshots even if it is only active for S_1. The benefits of using *snapshot-oblivious* frontier (no maintaining and checking separate frontiers) outweigh the extra computation overhead it introduces. Similar observation has been made in `Glign` [63] about the *query-oblivious* frontier.

The concurrent snapshot evaluation in Algorithm 2 takes the Q-Relevant Subgraph as input and applies the addition batch Δ_i to incrementally compute query results on snapshot i. The incremental computation for edge additions adds vertex v to

the frontier if a new edge $\langle u, v\rangle$ in addition batch improves the vertex value of v (Lines 4 – 8), e.g., a shorter shortest path is found after adding the edge. Iterative computation continues till the frontier becomes empty. At Line 10, the *snapshot-oblivious* frontier does not distinguish between snapshots; it evaluates the vertex for all snapshots which may introduce redundant computation but still outweighs the overhead of tracking separate frontiers for each snapshot. Out-going edges of active vertices are evaluated for snapshots that contain the edges (Lines 12 – 16).

V. System

Based on the proposed UVV finding algorithm and concurrent snapshots query evaluation engine, we developed a system for shared-memory. To our knowledge, this is the first system of its kind that supports concurrent evolving-graph query evaluation while using a very small portion of a large graph. *Our system is built upon Risgraph as it provides the most optimized implementation of KickStarter-based incremental approach that serves as the baseline in our evaluation.* Our UVV system consists of several components described below:

- An *evolving graph engine* maintains data structures of the multiple snapshots. It is built on the RisGraph [15] streaming graph engine and adopts version management of Fig 7(b). RisGraph leverages a fast addition query function and uses a pull-push hybrid mechanism and we extend it to support concurrent snapshot traversal.
- A graph reduction phase is added to the system that reduces the graph size to create a *query-relevant* graph.
- We employ a snapshot scheduler, which takes user queries as input and maximizes the reuse of snapshots. A snapshot-oblivious frontier mechanism is added to RisGraph. Users can pick the snapshots of interest.
- Finally, a simple programming interface is provided where the user can program by following the vertex-centric paradigm and providing the system with the query source and snapshots to get query results.

System Execution Engine. Two execution models are supported: concurrent and non-concurrent. Both perform graph reduction for a given user query. As depicted in Algorithm 1, we query the intersection graph and the union graph to find UVVs. The system then deletes all incoming edges of UVV vertices to create the query-relevant subgraph. The overhead of this step is included in query evaluation time.

In the non-concurrent execution mode, the system executes a query targeting multiple snapshots in two steps: the scheduling phase and the computation phase. During the scheduling phase, the query execution plan follows Fig 3, and the engine follows the scheduling order to perform addition-only incremental computation for all the queried snapshots.

VI. Performance Evaluation

We have successfully implemented our idea on top of the RisGraph [15] system which is the fastest streaming system that supports incremental algorithms for handling both edge additions and deletions. We made it suitable for analyzing an evolving graph by performing incremental computations incrementally over a versioned graph representation. We conducted our experiments on a shared memory machine on Google Cloud that has two Intel Xeon 2.60 GHz processors with 48 cores and 768GB memory. All code was compiled with g++ version 9.4.0 and run on Ubuntu 20.04.

Benchmarks: We evaluated five graph benchmarks: BFS (Breadth First Search), SSSP (Single Source Shortest Path), SSWP (Single Source Widest Path), SSNP (Single Source Narrowest Path), and Viterbi. Table II lists the benchmarks along with their edge functions.

TABLE II: Benchmarks and their Edge Functions.

Alg.	EdgeFunction $(e(u,v))$
BFS	$CASMIN(Val(v), min(Val(u)+1, val(v)))$
SSWP	$CASMAX(Val(v), min(Val(u), wt(u,v)))$
SSNP	$CASMIN(Val(v), max(Val(u), wt(u,v)))$
SSSP	$CASMIN(Val(v), Val(u)+wt(u,v))$
Viterbi	$CASMAX(Val(v), Val(u)/wt(u,v))$

TABLE III: Edges and Vertices of the Input Graphs.

Input Graph	#Edges	#Vertices	Avgdegree
LJ [9]	68M	4.8M	28.26
Deli [1]	101M	34M	17.16
OR [43]	117M	3.1M	76.28
Wen [33]	437M	13.5M	64.32
TW [34]	1.46B	41.6M	70.51
Fr [29]	2.58B	68.3M	55.00

Graph Data Sets: We used 5 real-world input graphs (LJ, OR, Wen, TW, and Fr) and one real-world temporal graph (Deli) to demonstrate that this approach works for temporal graphs as well, as shown in Table III. The Deli graph is the largest temporal graph with real-world batch updates that we could find. We created the temporal versions of the five real-world input graphs by using the original graph as the first snapshot and generating consecutive snapshots with 150k edge updates per round. The range for the number of vertices in our input graphs is from 68M to 2.6B, and the range for the number of edges is from 4.8M to 68.3M.

A. Speedups

In our experiments, we compare the execution times for query evaluation for six input graphs and five benchmarks across a sequence of 64 snapshots. Between two consecutive snapshots there are 150,000 edge updates, half of them are additions and half are deletions. As mentioned earlier in the introduction, when considering a fairly large number of snapshots, a large percentage of vertex property values remain unchanged across snapshots for most input graphs and algorithms within a typical time window of interest. Therefore, 150,000 edge updates is a realistic scenario in real-world settings. We present our results in Table IV.

TABLE IV: Average Execution Time for KickStarter-based (KS) method in milliseconds, and speedup for CommonGraph (CG), Q-Relevant Subgraph (QRS), Concurrent QRS (CQRS) given 64 Snapshots and 150,000 batch size.

Alg.	BFS	SSSP	SSWP	SSNP	Viterbi
LJ (LiveJournal)					
KS	1562.1	2292.2	2249.6	2183.0	2677.5
CG	1.17×	1.24×	1.32×	1.39×	1.07×
QRS	1.30×	1.53×	2.08×	2.43×	1.67×
CQRS	3.60×	2.01×	8.09×	7.62×	8.87×
Deli (Delicious)					
KS	891.45	1542.11	1417.51	1281.89	1508.07
CG	1.31×	1.42×	1.51×	1.85×	1.15×
QRS	1.83×	2.12×	2.91×	2.1×	1.61×
CQRS	3.79×	5.41×	7.25×	3.58×	3.45×
OR (Orkut)					
KS	905.9	1361.7	1499.1	1330.5	1310.2
CG	1.11×	1.06×	1.44×	1.17×	1.08×
QRS	1.56×	1.14×	1.60×	1.53×	1.25×
CQRS	3.69×	3.72×	8.37×	3.76×	3.03×
Wen (WikipediaLinks)					
KS	919.3	1850.1	1391.9	994.2	1007.3
CG	1.23×	1.42×	1.37×	1.46×	1.10×
QRS	1.55×	1.77×	1.85×	2.25×	1.52×
CQRS	3.07×	6.19×	11.7×	5.90×	3.56×
TW (Twitter)					
KS	971.8	1997.6	1993.2	1304.9	1774.7
CG	1.06×	1.86×	1.61×	1.20×	1.03×
QRS	1.44×	2.94×	3.44×	1.51×	2.61×
CQRS	3.77×	8.88×	10.23×	6.01×	8.25×
Fr (Friendster)					
KS	1349.8	2680.9	1951.6	1824.5	2968.5
CG	1.32×	1.13×	1.25×	1.2×	1.16×
QRS	2.08×	1.4×	3.57×	4.3×	1.82×
CQRS	6.5×	8.19×	9.57×	12.23×	11.95×

KickStarter-based (KS): baseline incremental approach does full computation on the first snapshot followed by repeated incremental processing of addition and deletion batches to get results for subsequent snapshots (see Figure 2(a)). The first row of Table IV for each benchmark displays the KS execution times in milliseconds. Our implementation is based on RisGraph as it represents the most optimized implementation of KickStarter.

CommonGraph (CG): results are presented for the best performing implementation [3], which is the work-sharing approach and present its speedups over KS in the second row CG for each benchmark in Table IV. We did not observe a substantial speedup when implementing CommonGraph because the RisGraph system is highly optimized, and the deletion operation is not as expensive compared to prior systems.

Q-Relevant Subgraph work-sharing (QRS): The next row QRS of Table IV gives the speedup achieved by the Q-Relevant Subgraph work-sharing method over KS. The QRS method significantly reduces the size of the graph over which incremental computations are performed and hence leads to a considerable speedup in obtaining results for individual snapshots. *However, there is overhead associated with generating the QRS graph (overhead cost) which we take into account by including it in the total query evaluation time.* QRS yields speedups ranging from 1.25× to 4.3× over KS.

Concurrent Q-Relevant Subgraph (CQRS): The fourth row CQRS of Table IV gives the speed-up achieved by Concurrent Q-Relevant Subgraph (CQRS) over KS. In this technique, we concurrently add delta batches to the QRS. *We have included the time for QRS generation (overhead cost) in the query evaluation time.* CQRS yields speedups ranging from 2.01× to 12.23× over KS.

B. Effectiveness and Accuracy of UVV Detection

After detecting UVVs, we determine the fraction of vertices whose results require incremental updates. By removing the incoming edges of UVV vertices, the remaining edge fraction in QRS is identified. The proportions of vertices and edges involved in incremental computation range from 0.3%–42% and 0.16%–32%, respectively (see Figure 9).

This large reduction is due to high accuracy of UVV detection. Figure 10 shows that percentage of values that are UVVs (purple bar) is extremely high. Moreover, our approach, despite being based on lower and upper bounds, is highly effective in detecting most of the UVVs (green bar). Therefore, the prevalence of UVVs, and our highly effective approach for identifying them, leads to the benefits observed.

C. QRS Generation Overhead

Next we present the portion of QRS and CQRS execution times spent on QRS generation. Figure 11 shows the execution times for all methods normalized to the KickStarter-based approach (KS). The red segments in the QRS and CQRS bars represent the QRS generation times. These times are included in query evaluation times because QRS generation must be performed for each query. The QRS generation consists of four steps. First, we solve the query on the intersection graph. Then, we incrementally add the missing edges to the intersection graph to obtain the results on the union graph. We compare the two value arrays, and if we find the same value for a node in both the $G_{\cup}$ and $G_{\cap}$ arrays, we should remove the incoming edges for that vertex. However, due to the much larger number of matches than mismatches, we implement this step in reverse. Instead of removing the incoming edges of the

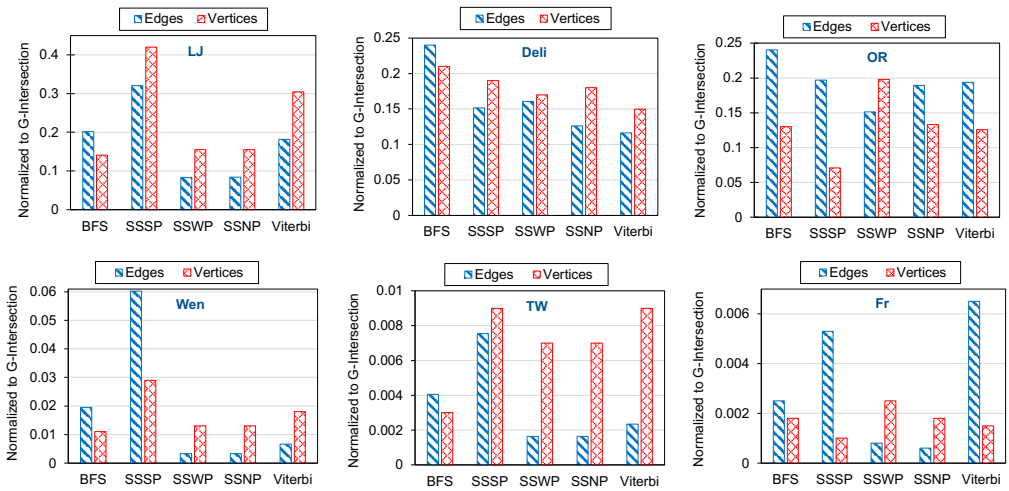


Fig. 9: Remaining fraction of edges in QRS (blue bars) and fraction of vertices whose values are computed via incremental computation (red bars), both normalized with respect to edges and vertices in $G_{\cap}$. This experiment uses 64 snapshots and a batch size of 150,000 edge updates (half additions and half deletions).

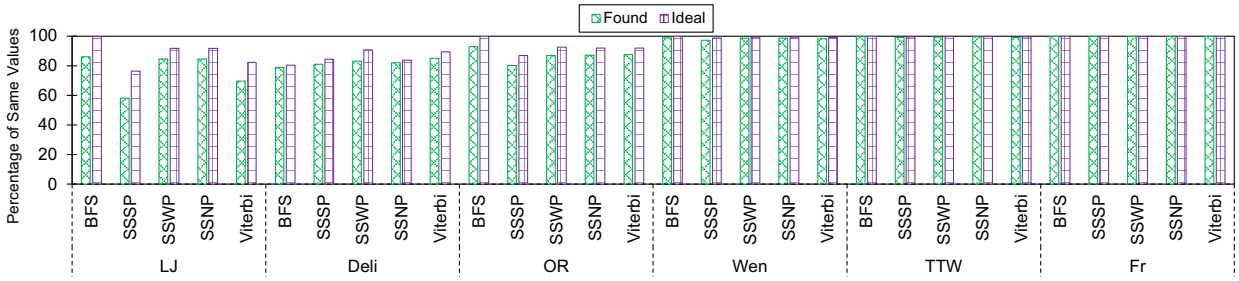


Fig. 10: Percentage of vertices that are UVVs (purple bars) vs. percentage of UVVs successfully detected by our algorithm (green bars). This experiment uses 64 snapshots and a batch size of 150,000 edge updates (half additions and half deletions).

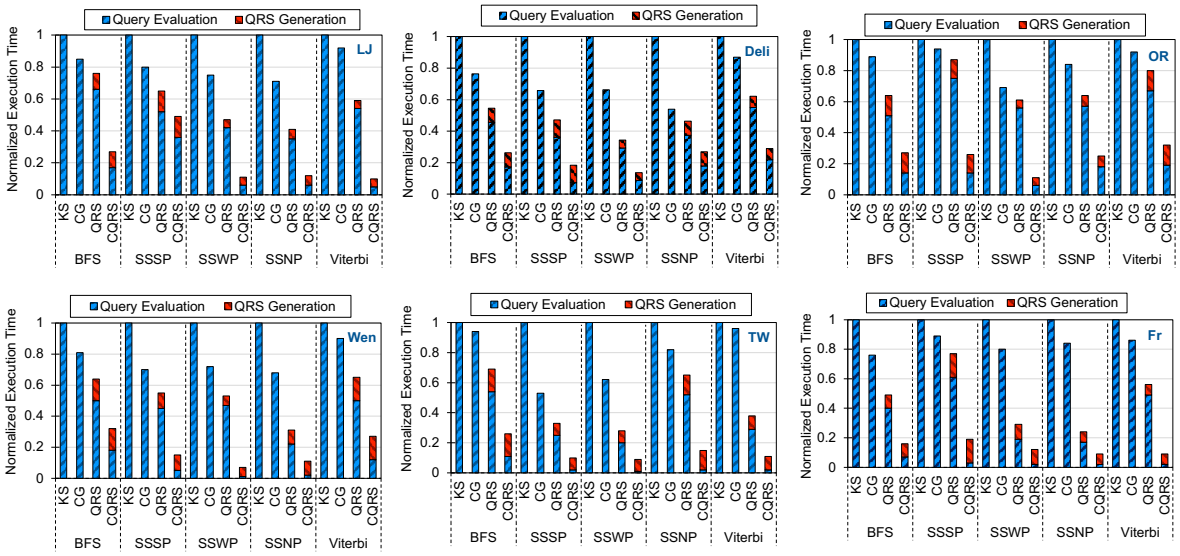


Fig. 11: The normalized execution times for KickStarter-based (KS), CommonGraph Work-Sharing (CG), Q-Relevant Subgraph (QRS), and Concurrent Q-Relevant Subgraph (CQRS) are presented with a breakdown of QRS generation time (overhead cost) and query evaluation times for 64 snapshots and 150,000 edge updates (half additions and half deletions).

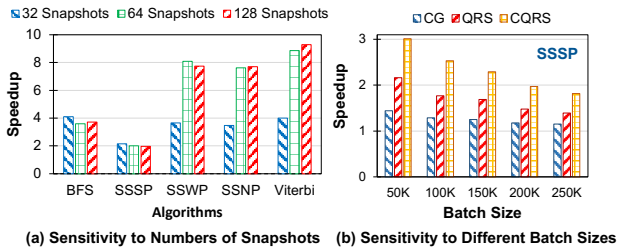


Fig. 12: (a) Sensitivity to the number of snapshots (32, 64, and 128) with batches of 150K edge updates for the LiveJournal graph; (b) Sensitivity to batch sizes of 50K, 100K, 150K, 200K, and 250K for the LiveJournal and the SSSP algorithm.

matching values, we add incoming edges for the mismatching values. On average, the QRS generation time accounts for 18.45%/56.01% of the total execution time for QRS/CQRS.

D. Sensitivity to Number of Snapshots

We studied the performance of UVV for 32, 64, and 128 snapshots. Figure 12(a) shows high speedups across varying number of snapshots, benchmarks, and graphs. Speedups for 64 and 128 snapshots are close, whereas the speedups for 32 snapshots are lower. This is because the resources available on our server are not fully utilized by 32 snapshots but are by 64 and 128 snapshots. We studied the sensitivity to different batch sizes for the LiveJournal graph. We chose LiveJournal for this experiment because it is our smallest graph, allowing us to observe the effects of changes in batch sizes more clearly. Figure 12(b) shows that larger batch sizes give lower speedups because they lead to more updates and fewer UVVs.

VII. Related Work

Evolving graph analytics: Recent works on dynamic graphs are Common Graph [3], RisGraph [15], and Tegra [24]. Common Graph transforms all deletions into additions. Query is executed on this graph and then missing edges are added incrementally for each snapshot. We exploited UVVs while preserving all the benefits of the Common Graph. In addition, unlike Common Graph, incremental computations are done concurrently over a versioned graph representation. RisGraph and Tegra *explicitly* process deletions and additions. For deletions they use the KickStarter [57] algorithm. RisGraph uses a new data structure for quick edge additions and removals, but this leads to memory size increase of 3.25x to 3.38x. Tegra offers an API for efficient querying over time windows, using a compact in-memory graph format. Both RisGraph and Tegra use algorithms from streaming systems to facilitate incremental computations. GraphOne [31] and Aspen [14] are other systems supporting dynamic and streaming graphs, while Chronos [20] and FA+PA [56] optimize memory and computation costs. But they lack edge deletion support when the goal is to perform query evaluation. LiveGraph [67] handles dynamic graph updates using transactionally consistent adjacency lists, improving performance for evolving graphs. Other systems use graph sharing when simultaneously evaluating multiple queries on one graph version [12], [63], [65].

Streaming graph analytics: These algorithms maintain a single graph version that is continuously updated, along with query results progressively refreshed as new batches of changes arrive. They focus on two key aspects: fast graph mutation and incremental query evaluation. In contrast, all versions of an evolving graph (i.e., multiple snapshots) are available from the outset, so graph mutation is not a concern once a multi-versioned representation is created. However, the incremental query evaluations developed for streaming graph systems are leveraged by evolving graph systems (e.g., Tegra [24], Common Graph [3]). The primary focus of these systems is on incremental computation, specifically on how to effectively update query outcomes. Early streaming platforms like Kineograph [13], Naiad [44], Tornado [54], and Tripoline [26] only support edge additions. However, KickStarter [57] and GraphBolt [40] support edge deletions.

VIII. Conclusion

We identified a new opportunity for optimizing the evaluation of an evolving graph query over a large number of snapshots. We showed that a large fraction of vertices whose query results do not change can be identified and computed once for all snapshots and exploited to minimize incremental computations performed concurrently for all snapshots over a much smaller graph. Our approach substantially outperforms state of the art incremental approaches.

Acknowledgment

This work is supported by NSF grants CCF-2512416, CCF-2226448, CCF-2106383, and CCF-2002554 to UC Riverside.

References

[1] Delicious. https://delicious.com/
[2] Afarin, M.: Redundancy Removal for Accelerating Graph Processing Workloads. Phd thesis, University of California, Riverside (2025), https://escholarship.org/content/qt7sp930rh/qt7sp930rh.pdf, eScholarship Repository
[3] Afarin, M., Gao, C., Rahman, S., Abu-Ghazaleh, N., Gupta, R.: Commongraph: Graph analytics on evolving data. In: Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'23), Vancouver, BC, Canada, March 25–29, 2023. pp. 133–145. ACM (2023). https://doi.org/10.1145/3575693.3575713, https://doi.org/10.1145/3575693.3575713
[4] Afarin, M., Gao, C., Rahman, S., Abu-Ghazaleh, N., Gupta, R.: Commongraph: Graph analytics on evolving data (abstract). In: Proceedings of the 2023 ACM Workshop on Highlights of Parallel Computing. p. 1–2. HOPC '23, Association for Computing Machinery, New York, NY, USA (2023). https://doi.org/10.1145/3597635.3598022, https://doi.org/10.1145/3597635.3598022
[5] Aggarwal, C.C., Subbian, K.: Evolutionary network analysis: A survey. ACM Computing Surveys (CSUR) **47**(1), 10 (2014)
[6] Ahmed, M.M., Mahmood, A., Hu, J.: A survey of network anomaly detection techniques. Journal of Network and Computer Applications **60**, 19–31 (2016)
[7] Akoglu, L., Tong, H., Koutra, D.: Graph-based anomaly detection and description: A survey. Data Mining and Knowledge Discovery **29**(3), 626–688 (2015)
[8] Amatriain, X., Basilico, J.: Recommender systems in industry: A netflix case study. In: Recommender systems handbook, pp. 385–419. Springer (2015)

[9] Backstrom, L., Huttenlocher, D., Kleinberg, J., Lan, X.: 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. pp. 44–54. KDD '06, ACM, New York, NY, USA (2006). https://doi.org/10.1145/1150402.1150412

[10] Batty, M., Axhausen, K.W., Giannotti, F., Pozdnoukhov, A., Bazzani, A., Wachowicz, M., Ouzounis, G., Portugali, Y.: Smart cities of the future. The European Physical Journal Special Topics **214**(1), 481–518 (2012)

[11] Ben-Nun, T., Sutton, M., Pai, S., Pingali, K.: Groute: An asynchronous multi-gpu programming model for irregular computations. In: Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. pp. 235–248. PPoPP '17 (2017)

[12] Chen, H., Shen, M., Xiao, N., Lu, Y.: Krill: A compiler and runtime system for concurrent graph processing. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. SC '21, Association for Computing Machinery, New York, NY, USA (2021). https://doi.org/10.1145/3458817.3476159, https://doi.org/10.1145/3458817.3476159

[13] Cheng, R., Hong, J., Kyrola, A., Miao, Y., Weng, X., Wu, M., Yang, F., Zhou, L., Zhao, F., Chen, E.: Kineograph: taking the pulse of a fast-changing and connected world. In: Proceedings of the 7th ACM european conference on Computer Systems. pp. 85–98 (2012)

[14] Dhulipala, L., Blelloch, G.E., Shun, J.: Low-latency graph streaming using compressed purely-functional trees. In: Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation. p. 918–934. PLDI 2019, Association for Computing Machinery, New York, NY, USA (2019). https://doi.org/10.1145/3314221.3314598, https://doi.org/10.1145/3314221.3314598

[15] Feng, G., Ma, Z., Li, D., Chen, S., Zhu, X., Han, W., Chen, W.: 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. p. 513–527. SIGMOD '21, Association for Computing Machinery, New York, NY, USA (2021). https://doi.org/10.1145/3448016.3457263, https://doi.org/10.1145/3448016.3457263

[16] Gao, C., Afarin, M., Rahman, S., Abu-Ghazaleh, N., Gupta, R.: Mega evolving graph accelerator. In: Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture. p. 310–323. MICRO '23, Association for Computing Machinery, New York, NY, USA (2023). https://doi.org/10.1145/3613424.3614260, https://doi.org/10.1145/3613424.3614260

[17] Garcia, S., Grill, M., Stiborek, M., Zunino, A.: An empirical comparison of botnet detection methods. Computers & Security **45**, 100–123 (2014)

[18] Giatsidis, C., Thilikos, D.M., Vazirgiannis, M.: D-cores: Measuring collaboration of directed graphs based on degeneracy. Knowledge and Information Systems **28**(3), 761–795 (2011)

[19] Gonzalez, J.E., Low, Y., Gu, H., Bickson, D., Guestrin, C.: Powergraph: Distributed graph-parallel computation on natural graphs. In: Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI). pp. 17–30 (2012)

[20] Han, W., Miao, Y., Li, K., Wu, M., Yang, F., Zhou, L., Prabhakaran, V., Chen, W., Chen, E.: Chronos: a graph engine for temporal graph analysis. In: Proceedings of the Ninth European Conference on Computer Systems. pp. 1–14 (2014)

[21] He, D., Liu, L., Chen, J.: Dynamic epidemic control strategy in complex networks. Physics Letters A **380**(36), 2960–2964 (2016)

[22] Hong, C., Sukumaran-Rajam, A., Kim, J., Sadayappan, P.: Multigraph: Efficient graph processing on gpus. In: Proceedings of the 26th International Conference on Parallel Architectures and Compilation Techniques. pp. 27–40. PACT '17 (2017). https://doi.org/10.1109/PACT.2017.48

[23] Digital 2022: Global overview report (2022), dataReportal–Global Digital Insights, available from https://hootsuite.widen.net/s/fzbtsvmcnl/digital-2022-october-statshot-report

[24] Iyer, A.P., Pu, Q., Patel, K., Gonzalez, J.E., Stoica, I.: TEGRA: Efficient Ad-Hoc analytics on evolving graphs. In: 18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21). pp. 337–355. USENIX Association (Apr 2021), https://www.usenix.org/conference/nsdi21/presentation/iyer

[25] Jiang, X., Afarin, M., Zhao, Z., Abu-Ghazaleh, N., Gupta, R.: Core graph: Exploiting edge centrality to speedup the evaluation of iterative graph queries. In: Proceedings of the Nineteenth European Conference on Computer Systems. p. 18–32. EuroSys '24, Association for Computing Machinery, New York, NY, USA (2024). https://doi.org/10.1145/3627703.3629571, https://doi.org/10.1145/3627703.3629571

[26] Jiang, X., Xu, C., Yin, X., Zhao, Z., Gupta, R.: Tripoline: generalized incremental graph processing via graph triangle inequality. In: EuroSys '21: Sixteenth European Conference on Computer Systems, Online Event, United Kingdom, April 26-28, 2021. pp. 17–32. ACM (2021). https://doi.org/10.1145/3447786.3456226, https://doi.org/10.1145/3447786.3456226

[27] Khorasani, F., Gupta, R., Bhuyan, L.N.: Scalable simd-efficient graph processing on gpus. In: Proceedings of the International Conference on Parallel Architectures and Compilation. pp. 39–50. PACT '15 (2015). https://doi.org/10.1109/PACT.2015.15

[28] Khorasani, F., Vora, K., Gupta, R., Bhuyan, L.N.: Cusha: vertex-centric graph processing on gpus. In: Proceedings of the 23rd International Symposium on High-Performance Parallel and Distributed Computing. pp. 239–252. HPDC '14, ACM (2014). https://doi.org/10.1145/2600212.2600227

[29] KONECT Project: Friendster network dataset. http://konect.cc/networks/friendster/ (2017), accessed: 2024-08-06

[30] Koren, Y., Bell, R., Volinsky, C.: Matrix factorization techniques for recommender systems. Computer **42**(8), 30–37 (2009)

[31] Kumar, P., Huang, H.H.: Graphone: A data store for real-time analytics on evolving graphs. ACM Trans. Storage **15**(4) (Jan 2020). https://doi.org/10.1145/3364180, https://doi.org/10.1145/3364180

[32] Kumar, S., Spezzano, F., Subrahmanian, V., Faloutsos, C.: Edge weight prediction in weighted signed networks. In: IEEE 16th International Conference on Data Mining (ICDM). pp. 221–230. IEEE (2016)

[33] Kunegis, J.: Konect – the koblenz network collection. In: Proceedings of International Conference on World Wide Web Companion, May 13–17, 2013, Rio de Janeiro, Brazil. pp. 1343–1350. ACM (2013). https://doi.org/10.1145/3575693.3575713, https://doi.org/10.1145/3575693.3575713

[34] Kwak, H., Lee, C., Park, H., Moon, S.: What is Twitter, a social network or a news media? In: Proc. 19th International Conference on World Wide Web. pp. 591–600. WWW '10, ACM, New York, NY, USA (2010). https://doi.org/10.1145/1772690.1772751

[35] Kyrola, A., Blelloch, G.E., Guestrin, C.: Graphchi: Large-scale graph computation on just a PC. In: USENIX Symposium on Operating Systems Design and Implementation (OSDI). pp. 31–46. USENIX Association (2012), https://www.usenix.org/conference/osdi12/technical-sessions/presentation/kyrola

[36] Li, L., Deng, D., Wei, J.: Survey on traffic anomaly detection. Journal of Network and Computer Applications **70**, 151–163 (2017)

[37] Liu, X., Wang, Y., Ji, X., Wen, W., Wang, G.: Dynamic network biomarkers for identifying critical transitions and their driving networks of multistage acute lung injury. PLoS One **13**(8), e0201985 (2018)

[38] Low, Y., Gonzalez, J.E., Kyrola, A., Bickson, D., Guestrin, C.E., Hellerstein, J.: Graphlab: A new framework for parallel machine learning. arXiv preprint arXiv:1408.2041 (2014)

[39] Malewicz, G., Austern, M.H., Bik, A.J., Dehnert, J.C., Horn, I., Leiser, N., Czajkowski, G.: Pregel: a system for large-scale graph processing. In: Proceedings of the 2010 ACM SIGMOD International Conference on Management of data. pp. 135–146 (2010)

[40] Mariappan, M., Vora, K.: Graphbolt: Dependency-driven synchronous processing of streaming graphs. In: Proceedings of the Fourteenth EuroSys Conference 2019. pp. 1–16 (2019)

[41] Mazloumi, A., Afarin, M., Gupta, R.: Expressway: Prioritizing edges for distributed evaluation of graph queries. In: 2023 IEEE International Conference on Big Data (BigData). pp. 4362–4371 (2023). https://doi.org/10.1109/BigData59044.2023.10386860

[42] Mazloumi, A., Jiang, X., Gupta, R.: Multilyra: Scalable distributed evaluation of batches of iterative graph queries. In: 2019 IEEE International Conference on Big Data (Big Data). pp. 349–358. IEEE (2019)

[43] Mislove, A., Marcon, M., Gummadi, K.P., Druschel, P., Bhattacharjee, B.: Measurement and analysis of online social networks. In: Proceedings of the 7th ACM SIGCOMM conference on Internet measurement. pp. 29–42 (2007)

[44] Murray, D.G., McSherry, F., Isaacs, R., Isard, M., Barham, P., Abadi, M.: Naiad: a timely dataflow system. In: Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles. pp. 439–455 (2013)

[45] Nazaraliyev, N., Sadredini, E., Abu-Ghazaleh, N.: Gpuvm: Gpu-driven unified virtual memory. arXiv preprint arXiv:2411.05309 (2024)

[46] Neirotti, P., De Marco, A., Cagliano, R., Mangano, G., Scorrano, F.: Current trends in smart city initiatives: Some stylised facts. Cities **38**, 25–36 (2014)

[47] Ngai, E.W.T., Chau, D.C.K., Chan, T.L.A.: Information technology, operational, and management competencies for supply chain agility: Findings from case studies. The Journal of Strategic Information Systems **21**(3), 232–249 (2012)

[48] Nguyen, D., Lenharth, A., Pingali, K.: A lightweight infrastructure for graph analytics. In: Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles. p. 456–471. SOSP '13 (2013). https://doi.org/10.1145/2517349.2522739

[49] Nguyen, H.A., Aiello, M., van der Mei, R.D.: A survey on performance management for 5g mobile networks. IEEE Communications Surveys & Tutorials **20**(4), 3242–3276 (2018)

[50] Nodehi Sabet, A.H., Qiu, J., Zhao, Z.: Tigr: Transforming irregular graphs for gpu-friendly graph processing. ACM SIGPLAN Notices **53**(2), 622–636 (2018)

[51] Rahman, S., Afarin, M., Abu-Ghazaleh, N., Gupta, R.: Jetstream: Graph analytics on streaming data with event-driven hardware accelerator. In: MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture. p. 1091–1105. MICRO '21, Association for Computing Machinery, New York, NY, USA (2021). https://doi.org/10.1145/3466752.3480126, https://doi.org/10.1145/3466752.3480126

[52] Rumsfeld, J.S., Joynt, K.E., Maddox, T.M.: Big data analytics to improve cardiovascular care: Promise and challenges. Nature Reviews Cardiology **13**(6), 350–359 (2016)

[53] Sakr, S., Bonifati, A., Voigt, H., Iosup, A., Ammar, K., Angles, R., Aref, W., Arenas, M., Besta, M., Boncz, P.A., et al.: The future is big graphs: a community view on graph processing systems. Communications of the ACM **64**(9), 62–71 (2021)

[54] Shi, X., Cui, B., Shao, Y., Tong, Y.: Tornado: A system for real-time iterative analysis over evolving data. In: Proceedings of the 2016 International Conference on Management of Data. pp. 417–430 (2016)

[55] Shun, J., Blelloch, G.E.: Ligra: a lightweight graph processing framework for shared memory. In: Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming. pp. 135–146 (2013)

[56] Vora, K., Gupta, R., Xu, G.: Synergistic analysis of evolving graphs. ACM Transactions on Architecture and Code Optimization (TACO) **13**(4), 1–27 (2016)

[57] Vora, K., Gupta, R., Xu, G.: 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. pp. 237–251 (2017)

[58] Vora, K., Tian, C., Gupta, R., Hu, Z.: Coral: Confined recovery in distributed asynchronous graph processing. In: Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems. pp. 223–236. ASPLOS '17 (2017). https://doi.org/10.1145/3037697.3037747

[59] Wang, J., Deng, W., Han, B., Yuan, Y.: Dynamic traffic flow prediction using transfer learning. IEEE Transactions on Intelligent Transportation Systems **17**(10), 2949–2959 (2016)

[60] Wang, Y., Pan, Y., Davidson, A.A., Wu, Y., Yang, C., Wang, L., Osama, M., Yuan, C., Liu, W., Riffel, A.T., Owens, J.D.: Gunrock: GPU graph analytics. ACM Transactions on Parallel Computing **4**(1), 3:1–3:49 (2017). https://doi.org/10.1145/3108140

[61] Yan, X., Sang, G., Xu, X., Jiang, W.: Survey on e-commerce recommender systems. IEEE Access **5**, 10911–10922 (2017)

[62] Yin, X., Zhao, Z., Gupta, R.: Glign: Taming misaligned graph traversals in concurrent graph processing. In: Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1. p. 78–92. ASPLOS 2023, Association for Computing Machinery, New York, NY, USA (2022). https://doi.org/10.1145/3567955.3567963, https://doi.org/10.1145/3567955.3567963

[63] Yin, X., Zhao, Z., Gupta, R.: Glign: Taming misaligned graph traversals in concurrent graph processing. In: Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1. p. 78–92. ASPLOS 2023, Association for Computing Machinery, New York, NY, USA (2022). https://doi.org/10.1145/3567955.3567963, https://doi.org/10.1145/3567955.3567963

[64] Zhang, X., Shasha, D.: Inferring the dynamic gene regulatory network underlying cell cycle regulation from microarray time series data. Journal of Bioinformatics and Computational Biology **10**(01), 1240002 (2012)

[65] Zhao, J., Zhang, Y., Liao, X., He, L., He, B., Jin, H., Liu, H., Chen, Y.: 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. SC '19, Association for Computing Machinery, New York, NY, USA (2019). https://doi.org/10.1145/3295500.3356143, https://doi.org/10.1145/3295500.3356143

[66] Zheng, Y., Liu, F., Hsieh, H.P.: U-air: When urban air quality inference meets big data. In: Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. pp. 1436–1444. ACM (2013)

[67] Zhu, X., Feng, G., Serafini, M., Ma, X., Yu, J., Xie, L., Aboulnaga, A., Chen, W.: Livegraph: a transactional graph storage system with purely sequential adjacency list scans. Proc. VLDB Endow. **13**(7), 1020–1034 (Mar 2020). https://doi.org/10.14778/3384345.3384351, https://doi.org/10.14778/3384345.3384351

[68] Zhu, X., Han, W., Chen, W.: Gridgraph: Large-scale graph processing on a single machine using 2-level hierarchical partitioning. In: USENIX Annual Technical Conference (USENIX ATC), July 8-10, Santa Clara, CA, USA. pp. 375–386. USENIX Association (2015), https://www.usenix.org/conference/atc15/technical-session/presentation/zhu

Appendix: Artifact Description

Artifact Description (AD)

IX. Overview of Contributions and Artifacts

A. Paper's Main Contributions

C_1 **Observation: many vertex values do not change across a time window.** In evolving graphs with gradual changes, evaluating a vertex-specific query (e.g., SSSP) over 25–100 snapshots shows that a large fraction of vertices keep exactly the same result across the entire window. This motivates computing these *Unchanged Vertex Values (UVVs)* once and reusing them across snapshots.

C_2 **Intersection–Union analysis to safely detect UVVs.** We introduce a safe bounding analysis based on two derived graphs: the *intersection graph* $G_\cap$ (edges common to all snapshots) and the *union graph* $G_\cup$ (edges appearing in any snapshot). By evaluating the query on $G_\cap$ and $G_\cup$, we obtain per-vertex lower/upper bounds across *all* snapshots; if the bounds are equal for a vertex, that vertex is *provably* a UVV.

C_3 **Query-Relevant Subgraph (QRS) to shrink incremental work.** After UVVs are identified, we build a smaller graph tailored to the query by removing incoming edges of UVV vertices (since their values never need updating) and pruning update batches accordingly. This reduced graph, the *QRS*, substantially lowers the amount of incremental processing needed per snapshot.

C_4 **Concurrent incremental evaluation over many snapshots (CQRS).** We design a concurrent execution mode that evaluates many snapshots together on top of a *versioned* graph representation and a *snapshot-oblivious frontier*. Implemented on top of RisGraph, this concurrent mode achieves the largest speedups over state-of-the-art incremental baselines.

B. Computational Artifacts

A_1 **UVVs system implementation + experiment runner (code).**
DOI (archived artifact):
`https://doi.org/10.5281/zenodo.18283774`
Contents: implementation of (i) intersection–union UVV detection, (ii) QRS construction and delta pruning, (iii) non-concurrent QRS execution, (iv) concurrent CQRS execution, and (v) baselines (RisGraph/KickStarter-style incremental; CommonGraph work-sharing). Includes scripts to reproduce the main figures/tables.

A_2 **Datasets + preprocessing + snapshot/delta generation (data + scripts).**
DOI (archived artifact):
`https://doi.org/10.5281/zenodo.18279487`
Contents: scripts to download/convert input graphs, generate synthetic snapshot sequences via edge updates (add/delete), and emit per-snapshot delta batches (and metadata such as seeds and parameters).

Artifact ID	Contributions Supported	Related Paper Elements
A_1	C_2, C_3, C_4	Algorithms (UVV/QRS and CQRS); Figures 9–12; Table IV (speedups)
A_2	C_1, C_2, C_3	Figure 1 (snapshot study); Table III (datasets); Inputs for Figures 9–12 and Table IV (snapshots/deltas)

X. Artifact Identification

A. Computational Artifact A_1

Artifact A_1: UVVs system implementation and experiment runner

Relation To Contributions

Artifact A_1 contains the paper's full pipeline for fast query evaluation over a snapshot window: (1) **UVV detection** using intersection–union bounds ($G_\cap$ and $G_\cup$); (2) **QRS construction** by removing incoming edges of proven-UVV vertices and pruning delta batches; and (3) **query execution** using either (a) per-snapshot incremental evaluation on QRS, or (b) **concurrent** evaluation (CQRS) using a versioned graph and snapshot-oblivious frontier. It also includes baseline implementations used in the paper for comparison (RisGraph/KickStarter incremental and CommonGraph work-sharing).

Expected Results

Running A_1 should reproduce the paper's main trends over a window of snapshots:

- **UVVs are common:** a high fraction of vertices are detected as UVVs (close to the "ideal" UVV rate reported in the paper).
- **QRS is much smaller:** only a minority of vertices require incremental updates, and the remaining edge set is a small fraction of $G_\cap$.
- **Speedups:** QRS improves over the RisGraph/KickStarter baseline, and CQRS yields the largest speedups when evaluating many snapshots together.

Artifact Setup (incl. Inputs)

Hardware:

- Recommended (paper-scale): a shared-memory server with many CPU cores and large RAM (the paper used 48 cores and 768 GB RAM).
- Disk: enough space for large graphs and snapshot inputs (often hundreds of GB to TB, depending on graphs and snapshot count).

Software:

- OS: Ubuntu 20.04.
- Compiler: `g++` 9.4.0 with C++17 support.
- Build: Make/CMake (as packaged).
- Optional: Python 3 for orchestration and plotting scripts.
- Baseline: RisGraph-based code path (UVVs extends RisGraph with versioned edges and concurrent traversal).

Datasets / Inputs:

- Graphs used in the paper: LJ (LiveJournal), OR (Orkut), Wen (WikipediaLinks), TW (Twitter), Fr (Friendster), and Deli (Delicious temporal graph).
- Main evaluation setting: 64 snapshots; 150,000 edge updates between consecutive snapshots.
- Motivating study setting: 25–100 snapshots with 100K edge updates per round (50/50 add/delete).
- Each run requires: a base snapshot (e.g., G_0 or $G_\cap$ in the system format), per-snapshot delta batches, and a query configuration.

Installation and Deployment:

- Build the system with `g++-9`; enable parallel runtime support (e.g., OpenMP/pthreads).
- For large graphs, system limits may need adjustment.

Artifact Execution

Workflow overview.

T_1 **Load inputs.** Load the base graph and snapshot deltas; prepare $G_\cap$ and $G_\cup$.

T_2 **Detect UVVs.** Run the query on $G_\cap$ and $G_\cup$ to obtain bound arrays; mark vertices whose bounds match.

T_3 **Build QRS.** Remove incoming edges of UVVs from $G_\cap$ and prune delta edges whose destination is a UVV.

T_4 **Evaluate snapshots.**

- *QRS (non-concurrent):* apply each reduced delta batch and run incremental propagation.
- *CQRS (concurrent):* build a versioned augmented graph and evaluate snapshots concurrently using a snapshot-oblivious frontier.

T_5 **Collect and plot.** Produce runtimes, speedups, UVV detection rates, and QRS reduction metrics; generate tables/plots matching the paper.

Dependencies: $T_1 \to T_2 \to T_3 \to T_4 \to T_5$.

Paper-default parameters (examples).

- Snapshots: $n = 64$ (sensitivity: $n \in \{32, 64, 128\}$).
- Update batch size: 150,000.
- Benchmarks: BFS, SSSP, SSWP, SSNP, Viterbi.

Artifact Analysis (incl. Outputs)

Outputs.

- Logs (CSV/JSON/text) with: UVV counts, QRS size (edge/vertex fractions), runtime breakdown (including QRS generation), and speedups vs. baselines.
- Reproduced paper elements: speedup table(s) and plots analogous to Figures 9–12 (UVV detection, QRS reduction, sensitivity).
- (Optional) per-snapshot result arrays (can be large).

B. Computational Artifact A_2

Artifact A_2: Datasets, preprocessing, and snapshot generation inputs/scripts

Relation To Contributions

Artifact A_2 provides the data inputs required to reproduce the motivating UVV study and the main evaluation. It supports C_1 by generating snapshot windows with controlled update rates, and supports C_2/C_3 by producing the derived inputs needed for UVV detection and QRS construction.

Expected Results

Running A_2 should produce:

- a snapshot window (base snapshot + delta batches);
- intersection/union representations ($G_\cap$ and $G_\cup$); and
- metadata consumed by A_1.

Artifact Setup (incl. Inputs)

Hardware:

- Sufficient disk/RAM for chosen graphs and snapshots.
- Multi-core CPU recommended for preprocessing.

Software:

- Scripting environment (e.g., shell utilities) as packaged.
- Standard compression tools (e.g., `gzip`/`zstd`).

Datasets / Inputs:

- Real-world graphs: LJ, OR, Wen, TW, Fr, and Deli.
- Synthetic temporalization for static graphs: treat the original as snapshot 0, then generate subsequent snapshots via controlled edge updates (paper default: 150K updates/round; 50% additions and 50% deletions).

Installation and Deployment:

- Download raw graph datasets referenced by the paper.
- Convert edge lists into the input format expected by A_1.
- Generate delta batches, and compute $G_\cap$ and $G_\cup$.

Artifact Execution

Workflow overview.

T_1 Download/ingest raw dataset.

T_2 Preprocess/normalize (e.g., relabel vertices, add weights if needed, convert formats).

T_3 Generate evolving-graph inputs: produce n delta batches with a chosen batch size and add/delete ratio.

T_4 Compute derived graphs/edge sets: $G_\cap$ and $G_\cup$.

Dependencies: $T_1 \to T_2 \to T_3 \to T_4$.

Key parameters.

- Number of snapshots n (default: 64).
- Batch size (default: 150K; sensitivity: 50K–250K).
- Add/delete ratio (default: 50/50).

Artifact Analysis (incl. Outputs)

Outputs.

- Preprocessed base graph in the required format.
- Delta batches $\{\Delta_i\}$ (and deletions if needed), or materialized snapshots $\{G_i\}$.
- Derived graphs: $G_\cap$ (intersection) and $G_\cup$ (union).
- Metadata files: n, batch sizes, add/delete ratios, seeds, and optional source-vertex selections.