CommonGraph: Graph Analytics on Evolving Data

Mahbod Afarin*
mafar001@ucr.edu
CSE Department, UC Riverside
USA

Chao Gao*
cgao037@ucr.edu
CSE Department, UC Riverside
USA

Shafiur Rahman
mrahm008@ucr.edu
CSE Department, UC Riverside
USA

Nael Abu-Ghazaleh
nael@cs.ucr.edu
CSE Department, UC Riverside
USA

Rajiv Gupta
rajivg@ucr.edu
CSE Department, UC Riverside
USA

ABSTRACT

We consider the problem of graph analytics on evolving graphs (i.e., graphs that change over time). In this scenario, a query typically needs to be applied to different snapshots of the graph over an extended time window, for example to track the evolution of a property over time. Solving a query independently on multiple snapshots is inefficient due to repeated execution of subcomputation common to multiple snapshots. At the same time, we show that using streaming, where we start from the earliest snapshot and stream the changes to the graph incrementally updating the query results one snapshot at a time is also inefficient. We propose *CommonGraph*, an approach for efficient processing of queries on evolving graphs. We first observe that deletion operations are significantly more expensive than addition operations for many graph queries (those that are monotonic). *CommonGraph* converts all deletions to additions by finding a common graph that exists across all snapshots. After computing the query on this graph, to reach any snapshot, we simply need to add the missing edges and incrementally update the query results. *CommonGraph* also allows sharing of common additions among snapshots that require them, and breaks the sequential dependency inherent in the traditional streaming approach where snapshots are processed in sequence, enabling additional opportunities for parallelism. We incorporate the *CommonGraph* approach by extending the KickStarter streaming framework. We implement optimizations that enable efficient handling of edge additions without resorting to expensive in place graph mutations, significantly reducing the streaming overhead, and enabling direct reuse of shared edges among different snapshots. *CommonGraph* achieves 1.38×-8.17× improvement in performance over Kickstarter across multiple benchmarks.

CCS CONCEPTS

• **Computing methodologies** → **Parallel computing methodologies**; • **Information systems** → **Computing platforms**.

*Both authors contributed equally to this research.

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

ACM ISBN 978-1-4503-9916-6/23/03.
https://doi.org/10.1145/3575693.3575713

KEYWORDS

evolving graphs, iterative graph algorithms, work sharing

ACM Reference Format:
Mahbod Afarin, Chao Gao, Shafiur Rahman, Nael Abu-Ghazaleh, and Rajiv Gupta. 2023. 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 '23), March 25–29, 2023, Vancouver, BC, Canada.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3575693.3575713

1 INTRODUCTION

Analyses on large graphs are an increasingly important computational workload as graph analytics is employed in many domains – social networks [7, 10, 13, 21, 42], web graphs [30], brain networks [6] and others – to uncover insights by mining high volumes of connected data. Due to the iterative nature of many graph analytics workloads, repeated passes over the graph are required until the algorithm converges to a stable solution. Since real-world graphs can be very large (e.g., YahooWeb has 1.4 billion vertices and 6.6 billion edges), iterative graph analytics workloads are highly memory-intensive. There has been significant interest in developing scalable and efficient graph analytics systems. Some examples of such systems are Ligra [41], Galois [35], GraphChi [29], GridGraph [48], GraphLab [31], GraphX [18], and PowerGraph [17].

Graphs are often dynamic, with edges and vertices being added or removed over time [5]. There are two broad classes of analyses on dynamic graphs: (1) *Streaming graph analytics*: where results of a query are continuously updated as the graph continues to change because updates to it stream in over time; for example, we may want to maintain the shortest path to a destination as the traffic conditions change. Typically incremental algorithms are employed to update query results in response to graph changes and thus avoid recomputing the query from scratch [8, 14, 23, 32, 38, 40, 44]; and (2) *Evolving graph analytics*: in this scenario, queries seek to answer questions about a dynamic graph as it evolves over longer time scales [19, 22, 43]. Multiple snapshots of the graph are available in this scenario. Typically an evolving graph (EG) query seeks to *track a graph property over a long time scale* by computing its value at different snapshots within the time window identified by the query. This makes the problem potentially significantly more computationally expensive than both traditional analytics on a static graph as well as streaming analytics on a changing graph.

A straightforward approach to this problem is to apply the query to the individual snapshots independently; however, this approach

has significant overheads since we end up solving the query independently on many different snapshots. Alternatively, another approach that has been used [15, 22] starts from the earliest snapshot and uses streaming to move from one snapshot to the next in sequence. While this approach results in significantly less work, provided that the number of changes between snapshots is not too large, it has a number of drawbacks. First, we show that for many algorithms the cost of edge deletions is very high; thus, performance benefits of the incremental algorithm are limited. Second, the solution moves between snapshots in sequence which limits opportunities for sharing query evaluation work among them.

In this paper, we propose *CommonGraph*, a new algorithm and system for efficiently evaluating an evolving graph query. *CommonGraph* reduces the overhead of multiple evaluations by identifying the common subgraph that is shared by the multiple snapshots of interest. Specifically, we first process this common graph, sharing this overhead among all the snapshots, and then stream in additional edges that convert this common graph into each of the other snapshots. *CommonGraph* is motivated by two key new insights:

(1) *Converting expensive deletions to additions.* We show that the incremental cost of processing deletions is significantly greater than additions for an important class of algorithms that is monotonic (these are the algorithms supported by Kickstarter).Therefore, the *CommonGraph* is designed to naturally convert all deletion operations into addition operations– we start with the common set of edges across a group of snapshots, and only have to add edges to reach any snapshot, eliminating the high overhead deletions. The overhead of graph mutation is also reduced with only additions; and

(2) *Breaking the dependencies between incremental computations.* The addition of edges to reach any snapshot is an independent operation and thus, we can benefit from the efficient incremental additions algorithm for all snapshots. This makes possible a new structure, the *Triangular Grid*, that enables further *work sharing* among subsets of snapshots via reuse of the edge additions shared by them.

We implement *CommonGraph* by extending the Kickstarter [44] system, a state of the art streaming graph framework; *CommonGraph* is a general algorithmic idea that can be implemented within other frameworks as well. We observe that the dynamic graph update process within kickstarter is extremely expensive, and, given the structure of *CommonGraph* come up with a graph representation that adds the new edges without having to change the primary graph representation. This strategy also enables us to share the common changes naturally among multiple snapshots in a space efficient way. This implementation enables us to compare *CommonGraph* against the two baselines: computing the query on each snapshot; and streaming all the changes from the initial snapshot.

The key contributions of our work are as follows:

- A new approach to evolving graphs analysis that avoids processing of expensive deletions by converting all graph updates to additions over the *CommonGraph*, replaces expensive deletions by additions, and removes dependencies that enable work sharing among the snapshots.

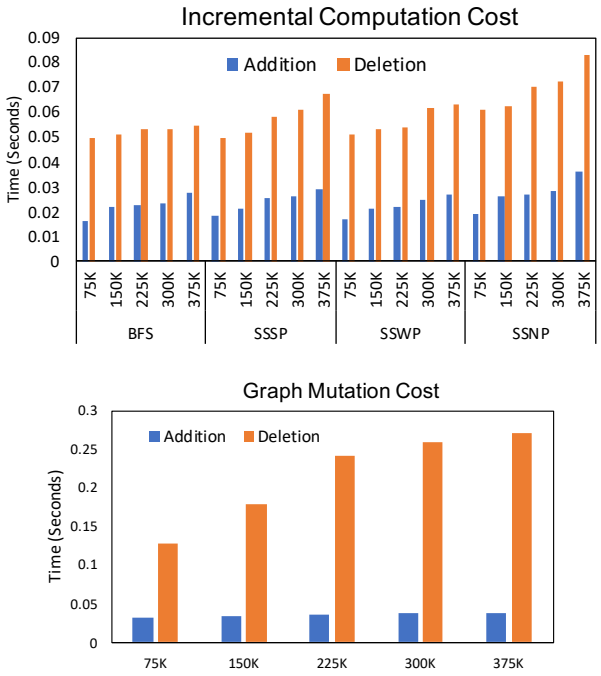

Figure 1: Kickstarter [44] Costs: (Top) Computation Cost of Deletions vs. Additions; and (Bottom) Graph Mutation Cost.

- A new structure called *Triangular Grid* (TG) that exposes work sharing possibilities among the snapshots. We demonstrate that maximizing work sharing corresponds to solving the Steiner tree problem on the TG, enabling further optimization of evolving graph queries.
- A graph representation that avoids the need to mutate graphs and enables reuse of edges by snapshots that share them.
- We build the *CommonGraph* that exploits the three ideas above to achieve speedups ranging between 1.38× and 8.17× over Kickstarter both by avoiding deletions and performing work sharing. Further speedups are also possible since the evaluations of the snapshots are highly parallelizable.

2 LIMITATIONS OF EXISTING SYSTEMS AND OVERVIEW OF COMMONGRAPH

In this section, we discuss the factors that limit the performance of existing evolving graph processing systems and then provide an overview of *CommonGraph* and how it overcomes these limitations.

2.1 Dynamic Graph Workloads and Systems

Dynamic graph systems can be categorized into streaming graph and evolving graph systems. A streaming graph system maintains a single version of the changing graph and, using incremental algorithms, continuously updates the results of a query as the graph is modified to maintain the query solution relative to the latest graph. To amortize the cost of streaming, typically updates are batched. An evolving system evaluates a query on multiple snapshots of a graph, that can stretch over a large time window, extracting historical/trend data. It may carry out query evaluation on multiple

snapshots in parallel or start with one snapshot at a time and leverage incremental algorithms developed for streaming systems to reach others.

Let G_t represent the snapshot of a dynamic graph at snapshot t. At snapshot $t+1$, the graph changes to G_{t+1} via application of two *batches of updates*, a batch of edge additions Δ_+^t and a batch of edge deletions Δ_-^t. In a streaming graph system a single version of graph is maintained, i.e. G_t changes to G_{t+1} after application of batches of updates (i.e., Δ_+^t and Δ_-^t). In an evolving graph scenario multiple snapshots of the graph $[G_{t_0}, G_{t_1}, G_{t_2}, ... G_{t_n}]$ are considered (e.g., for a transportation network, the snapshots may correspond to different amounts of traffic at different times of the day, or across different days of the week). If the snapshots are very far away from each other in time, streaming a large batch of updates may be more expensive than evaluating the query on the two snapshots independently. However, if snapshots are closer in time (more accurately, the number of updates is not excessive), then the incremental algorithms used by streaming systems can be leveraged to avoid redundant recomputation.

When incremental algorithms are being leveraged, after evaluating the query on one snapshot, the graph for the snapshot is first mutated (i.e., the data structures are changed in place) to obtain the one for the other snapshot and then the incremental algorithm is used to update query results. A primary observation that motivates *CommonGraph* is that when the system has to handle both deletions and additions, the cost of incremental computation that handles deletions is significantly higher than that for additions. Moreover, the cost of graph mutations is also significant. In Figure 1 these effects are shown for the Kickstarter [44] system. The incremental cost of processing a batch of deletions is nearly 3× the cost of processing an equal number of additions, and this observation holds across batches of different sizes. Handling deletions is more expensive because the algorithm is more complex for monotonic algorithms and impacts on query results are more widespread across the graph, necessitating significantly more processing. Finally, Kickstarter's cost of graph mutation is also several times greater for deletions than additions.

2.2 CommonGraph: Converting Deletions to Additions

To address the above problems (i.e., the high incremental cost of deletions and the significant cost of mutation), our system shown in Figure 2 based on *CommonGraph* introduces these three complementary techniques. The graph is represented in form of the shared *CommonGraph* and additional batches of edges (Δ batches) that can be used in conjunction with the *CommonGraph* to realize different snapshots. This representation of the graph and its updates allows different query evaluation schedules (shown as red arrows) to be realized that do not require deletions, incorporate work sharing, and do not require explicit graph mutation. Next, we provide an overview of these three features.

Converting Deletions to Additions. To overcome the high computational cost of processing edge deletions, we make a key observation: all deletions can be converted to additions by computing a *CommonGraph* that includes only the edges that are common to all the snapshots under consideration. Once query results have been computed on this graph, then results for any snapshot can be computed by *adding the batch of missing edges* to the *CommonGraph* and employing the incremental algorithm to update query results. That is, we can avoid the use of the more expensive incremental algorithm for deletions since deletions become additions if we reverse the order in which the snapshots are processed.

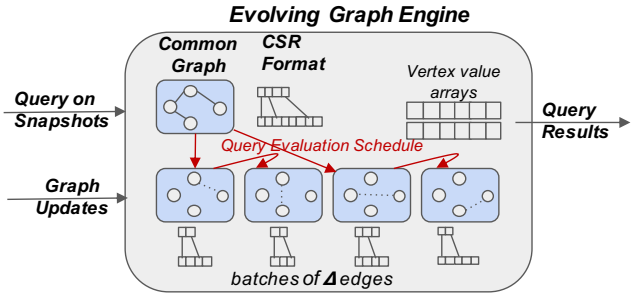

Figure 2: Our Approach.

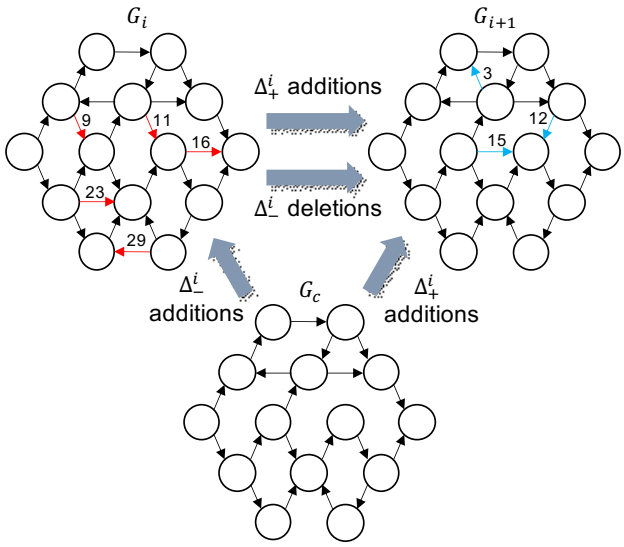

Figure 3: *CommonGraph* G_c for two snapshots G_i and G_{i+1}; the latter is created via a Δ_+^i additions (blue edges) and Δ_-^i deletions (red edges); going from G_c to G_i and G_{i+1} requires only addition of Δ_-^i and Δ_+^i respectively.

Figure 3 illustrates how *CommonGraph* converts deletions into additions. The top two graphs represent two snapshots of the evolving graph. The standard streaming approach will apply Δ_+^i edge additions and Δ_-^i deletions to update query results for the first snapshot G_i and obtain the query results for second snapshot G_{i+1}. On the other hand, by creating the *CommonGraph* G_c, we can incrementally update query results for G_c via independently performing edge additions Δ_+^i and Δ_-^i to obtain results for G_i and G_{i+1}.

Work Sharing for a Large Number of Snapshots. Although the *CommonGraph* achieves work sharing among all the snapshots to process the common graph itself, as the time window grows and the

number of snapshots increases, additional opportunities for work sharing of the graph updates among subsets of snapshots arise. Specifically, any subset of the snapshots may share additional edges in common (for example, if we constructed the common graph just for that subset), and can share work if we stream the additional edges to reach this larger common graph together instead of each query adding them separately to each snapshot. Let us assume that we have n snapshots to consider. Our second contribution, the *Triangular Grid* (TG) representation, allows systematic exploration and discovery of a n incremental computations that result in query results for all n snapshots while at the same time maximizing the overall *work sharing* beyond the base level of sharing that the *CommonGraph* naturally achieves.

The TG representation incorporates original snapshots, the *CommonGraph* representation, and many intermediate *CommonGraphs* that correspond to *CommonGraphs* for subsequences of original snapshots. Edges connecting snapshots are labeled with batches of edge additions that convert one *CommonGraph* into another representing a smaller sequence of snapshots. We show that solving the Steiner Tree problem on the *Triangular Grid* representation identifies the sequence of transitions through the grid that maximizes reuse. Algorithms for constructing the TG representation and finding the set of incremental computations that maximize work sharing are discussed in the next section.

Evolving Graph Representation for Evaluating Query. Besides the two algorithmic contributions above, we also identify a significant source of overhead in streaming graph systems. Specifically, we observe that there is a high cost of mutation –updating the graph representation – for converting one snapshot into another. To address this overhead, we introduce a new representation that significantly reduces the mutation overhead. In addition to the Compressed Sparse Row (CSR) representation of the *CommonGraph*, we also create CSR representations of the batches of edges that correspond to edges that transition from one node to another in the TG. This way, the *CommonGraph* is never changed but rather batches of additional edges that need to be added to the *CommonGraph* to obtain an intermediate or final snapshot are simply loaded to augment the *CommonGraph*. Different snapshots are thus represented by the set of additional edges they include.

Next we present the *Triangular Grid* representation and algorithms for finding query evaluation schedule to exploit *work sharing*.

3 COMMONGRAPH ALGORITHMS

We present two *CommonGraph* based algorithms for evaluating a query on a sequence of snapshots corresponding to a time window. First, we present the *direct hop* algorithm for query evaluation and show the work sharing enabled by the *CommonGraph* reduces the cost of query evaluation. Second, we present the *Triangular Grid* (TG) representation, which provides *CommonGraph* representations for subsequences of snapshots, and show how it is used to find an even better query evaluation schedule that maximizes work sharing.

3.1 Direct Hop Query Evaluation

Since the *CommonGraph* represents edges common to all snapshots under consideration, a significant degree of *work sharing* across

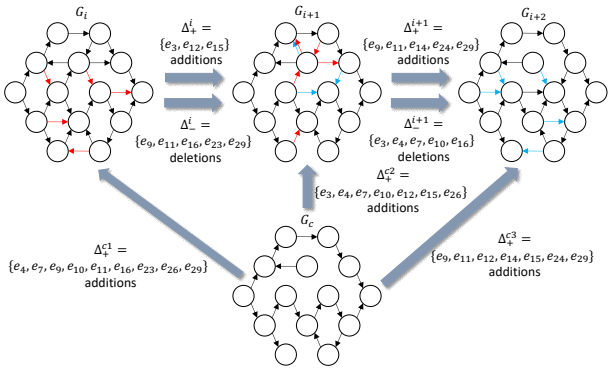

Figure 4: Showing an example for direct-hop algorithm.

snapshots is achieved by evaluating the query on the *CommonGraph* and then updating the results incrementally by applying the impact of edge additions corresponding to each snapshot. This approach avoids the need to use the expensive incremental algorithm for deletions while achieving significant work sharing via the *CommonGraph*. We name this method the *direct-hop* algorithm because the query evaluation schedule directly computes the query results for each snapshot from the results of the *CommonGraph*.

The example in Figure 4 illustrates the benefits of this approach. Let us consider we have three snapshots namely G_i, G_{i+1}, and G_{i+2}. Let us assume that the batches of edge additions and deletions that derive G_{i+1} from G_i (i.e., Δ_+^i and Δ_-^i) and then derive G_{i+2} from G_{i+1} (i.e., Δ_+^{i+1} and Δ_-^{i+1}) are as shown in Figure 4.

Consider that the *CommonGraph* G_c that has all the common edges from the three snapshots as shown in Figure 4. We can reach G_i, G_{i+1}, and G_{i+2} by adding Δ_+^{c1}, Δ_+^{c2}, and Δ_+^{c3} to G_c respectively. Next we show that the direct-hop algorithm with only additions is more efficient than the incremental approach used in Kickstarter that evolves the graph from G_i to G_{i+1} to G_{i+2} with both additions and deletions. Finding query results for G_{i+1} from G_i and G_{i+2} from G_{i+1} will require processing 8 additions ($|\Delta_+^i| + |\Delta_+^{i+1}|$) and 10 deletions ($|\Delta_-^i| + |\Delta_-^{i+1}|$). On the other hand, direct-hop approach will require processing of 22 additions ($|\Delta_+^{c1}| + |\Delta_+^{c2}| + |\Delta_+^{c3}|$). Since deletions are more expensive than additions (3× in Kickstarter), direct-hop evaluation is expected to be more efficient. This observation is confirmed by our experimental results presented later in the paper. Note that while Kickstarter will compute the query from scratch on G_i, direct-hop wil do so on the *CommonGraph*. Since, the *CommonGraph* is a subgraph of G_i, we conservatively (in favor of Kickstarter) assume that these costs are similar.

Finally, we observe that if we simply want to incrementally evaluate a query on snapshot G_{i+1}, then evaluating it from G_c is less expensive than evaluating it from G_i. The former requires 7 additions and the latter requires 3 additions and 5 deletions.

3.2 Triangular Grid Based Algorithm for Query Evaluation with Maximal Work Sharing

In this section we consider *work sharing* when evaluating a query over a long sequence of snapshots $G_0, G_1, \cdots, G_n$. Once again G_c includes all the edges that are common to all the snapshots $G_0, G_1, \cdots, G_n$

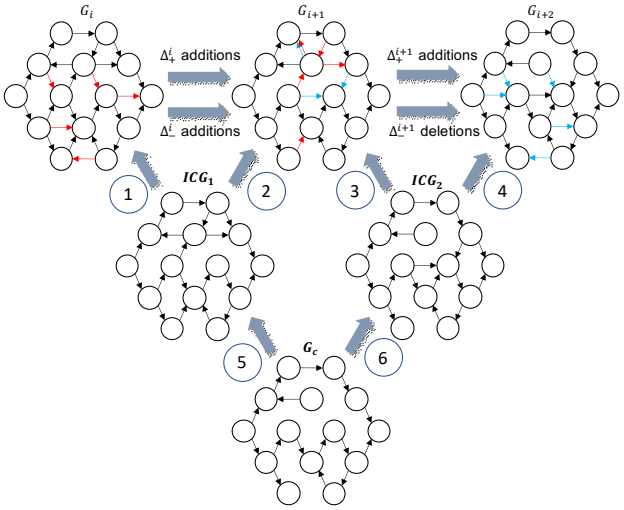

Figure 5: Triagular Grid (TG) corresponding to three original snapshots G_i, G_{i+1}, and G_{i+2}.

and, using the additions-only incremental algorithm, from the results of G_c the results for $G_0, G_1, \cdots, G_n$ can be independently computed via direct-hop approach. However, we observe that when there are large number of snapshots, *it becomes possible to create intermediate common graphs for subsequences of snapshots and take advantage of additional work sharing opportunities.*

Triangular Grid: To take advantage of additional work sharing we introduce the *Triangular Grid* (TG) representation which is illustrated, without loss of generality, for three snapshots G_i, G_{i+1}, and G_{i+2} in Figure 5. The TG includes two *Intermediate Common-Graphs* (ICGs) that are common-graphs for pairs of original snapshots. In Figure 5, $G_{c(i\rightarrow i+1)}$ is the ICG for G_i and G_{i+1} while $G_{c(i+1\rightarrow i+2)}$ is the ICG for G_{i+1} and G_{i+2}. Finally, the *CommonGraph* for *all* three snapshots is denoted as $G_{c(i\rightarrow i+2)}$ or simply G_c.

Note that although we have shown a TG for three snapshots, the symmetric nature of the representation shows that it applies to any arbitrary number of snapshots. If there are n original snapshots to be analyzed, the TG will contain exactly $n-2$ intermediate levels as the number of ICGs reduces by one for each level. More importantly we note that edges from the root *CommonGraph* G_c to the two ICGs and then from the two ICGs to the original snapshots are all labelled *exclusively with additions*. That is, to move from the root G_c to any original snapshot via intermediate ICGs we only need to perform batches of edge additions. Finally, by starting from the root G_c, and then potentially moving through exactly one ICG at each intermediate level, a query evaluation schedule can be chosen that exploits additional sharing among subsequence of snapshots represented by the chosen ICG. TG guarantees the presence of an ICG to represent any consecutive sub-sequence of snapshots. It is important to note that the ICGs are never stored and are generated on demand only by streaming the common edge additions if they are needed (that is, when we need to compute multiple snapshots reachable from the ICG).

Next we show how, from the batches of additions and deletions between original snapshots, we can compute the additions labelling all other edges in the TG leading to or originating from the ICGs. Let us assume that the batches of edge additions and deletions that derive G_{i+1} from G_i (i.e., Δ_+^i and Δ_-^i) and then derive G_{i+2} from G_{i+1} (i.e., Δ_+^{i+1} and Δ_-^{i+1}) are as follows:

$$\Delta_+^i = \{e3, e12, e15\}$$
$$\Delta_-^i = \{e9, e11, e16, e23, e29\}$$
$$\Delta_+^{i+1} = \{e9, e11, e14, e24, e29\}$$
$$\Delta_-^{i+1} = \{e3, e4, e7, e10, e26\}$$

Around the intermediate level of the TG (i.e., just below the original snapshots and just above G_c), we can easily compute the values of the following six batches of additions corresponding to the six edges (four involving original snapshots and two involving G_c) as follows:

1. $\Delta_+^{ICG_1\rightarrow G_i} = \Delta_-^i = \{e9, e11, e16, e23, e29\}$
2. $\Delta_+^{ICG_1\rightarrow G_{i+1}} = \Delta_+^i = \{e3, e12, e15\}$
3. $\Delta_+^{ICG_2\rightarrow G_{i+1}} = \Delta_-^{i+1} = \{e3, e4, e7, e10, e26\}$
4. $\Delta_+^{ICG_2\rightarrow G_{i+2}} = \Delta_+^{i+1} = \{e9, e11, e14, e24, e29\}$
5. $\Delta_+^{G_c\rightarrow ICG_1} = \Delta_-^{i+1} - \Delta_+^i = \{e4, e7, e10, e26\}$
6. $\Delta_+^{G_c\rightarrow ICG_2} = \Delta_+^i - \Delta_-^{i+1} = \{e12, e15\}$

Query Evaluation Schedules: Every tree rooted at G_c and including all leaves (snapshots) represents a query evaluation schedule. For our example, the TG representation and two query evaluation schedules are shown in Figure 6. Recall that for this example, in the preceding section, we had shown that the cost of direct-hop schedule is 22 additions. However, when we consider the two trees shown here, their costs are 19 additions (for $Tree_1$) and 21 additions (for $Tree_2$). This is because both the schedules shown take advantage of additional *work sharing*: in $Tree_1$, by including ICG_1, we can share additions that are common to G_i and G_{i+1} but not present in G_{i+2}; and in $Tree_2$, by including ICG_2, we can share additions that are common to G_{i+1} and G_{i+2} but not present in G_i. In other words,

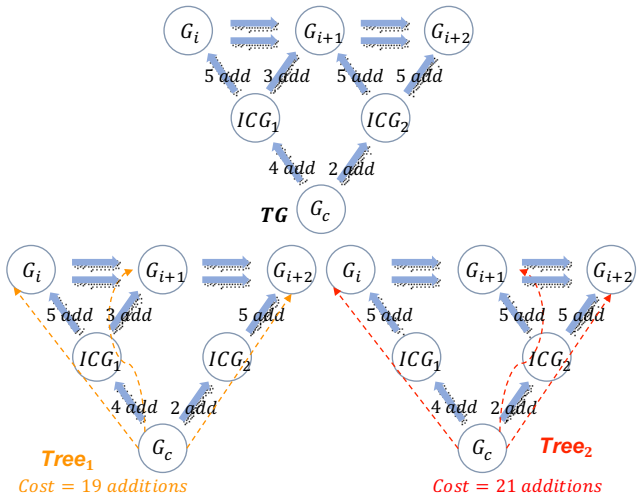

Figure 6: Query Evaluation Schedules: (TG) Triangular Grid; ($Tree_1$, $Tree_2$) Two Trees Corresponding to Query Evaluation Schedules with Different Costs.

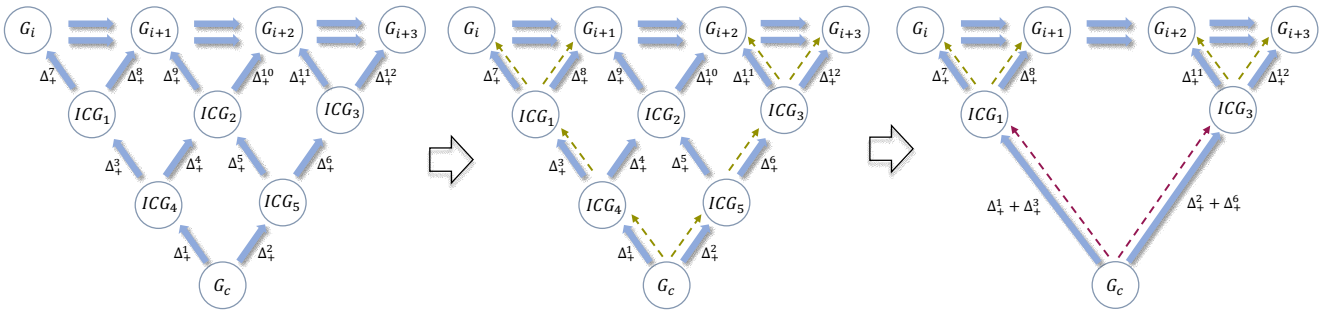

Figure 7: Algorithm Steps: (a) Create TG; (b) Identify Steiner Tree; and (c) Bypass nodes with indegree and outdegree of 1.

Algorithm 1 Algorithm for Identifying Minimum Cost Query Evaluation Schedule using the Steiner Tree Algorithm.

Inputs: Snapshots; Additions; and Deletions.
Snapshots– $G_1(V_1, E_1)$, $G_2(V_2, E_2)$, ... , $G_N(V_N, E_N)$;
Batches of Additions – $\Delta^+_{G_1}$, $\Delta^+_{G_2}$, ... , $\Delta^+_{G_N}$;
Batches of Deletions – $\Delta^-_{G_1}$, $\Delta^-_{G_2}$, ... , $\Delta^-_{G_N}$;
Output: TG & $Tree$ (Minimum Cost Schedule).

Build-Triangular-Grid;
Identify-Steiner-Tree;
Compress-Steiner-Tree;

function Build-Triangular-Grid
    $S \leftarrow$ Snapshots; $TG \leftarrow \phi$; $i \leftarrow 0$;
    for each $G_i(V_i, E_i)$ and $G_{i+1}(V_{i+1}, E_{i+1}) \in S$ **do**
        if G_i and G_{i+1} are leaf nodes **then**
            $w_1 = \Delta^-_{G_i}$; $w_2 = \Delta^+_{G_i}$
        else
            $w_1 = \Delta^-_{G_{i+1}} - \Delta^+_{G_i}$; $w_2 = \Delta^+_{G_i} - \Delta^-_{G_{i+1}}$;
            $\Delta^+_{ICG_i} = w_2$; $\Delta^-_{ICG_i} = w_1$
        end if
        $TG \leftarrow TG \cup \{(ICG_i, G_i, w_1)$
        $TG \leftarrow TG \cup \{(ICG_i, G_{i+1}, w_2)$
        $S \leftarrow S \cup ICG_i$; $i \leftarrow i + 1$
    end for
end function

function Identify-Steiner-Tree
    Let $Terminals$ be the leaves and root in TG;
    $Tree \leftarrow$ subtree of TG with at least one terminal;
    while $Tree$ does not span all terminals **do**
        Select a terminal x not in $Tree$ such that
            x is closest to a vertex in $Tree$
        Add to $Tree$, shortest path connecting x to $Tree$
    end while
end function

function Compress-Steiner-Tree
    for each vertex $v \in Tree$ **do**
        if v has one incoming and one outgoing edge **then**
            Bypass v
        end if
    end for
end function

the edges that are traversed multiple times are only computed once stopping at the ICG where the paths diverge, reducing the number of additions through work sharing.

While the reduction in number of total additions in the above example is small, it should be noted that this reduction will grow as the number of snapshots grows because, especially in higher levels of the TG, each edge can lead to many snapshots and would have been added independently and redundantly for each snapshot in direct-hop. Moreover, a secondary factor is that the size of G_c becomes smaller as the number of edge additions and deletions increases with more snapshots, leading to more unexploited work sharing opportunities among the subsequences of snapshots grow. Finally, note that when there is a path from G_c to a leaf snapshot which passes through an ICG which has exactly one incoming and outgoing edge in the $Tree$ identified, we simply bypass the node and combine the addition batches for incoming and outgoing edges into one larger batch to maximize parallelism. In our example, in $Tree_1$ we bypass ICG_2 and in $Tree_2$ we bypass ICG_1.

Complete Algorithm for Finding Minimum Cost Query Evaluation Schedule. The algorithm consists of three steps as shown in Figure 7. In the first step, we will create the *Triangular Grid* TG for a given sequence of snapshots on which a user query needs to be evaluated. The optimal query evaluation schedule consists of the tree that reaches all the snapshots with minimum total cost; this minimum cost represents the maximal degree of reuse, allowing the cost to drop to this minimal from the direct-hop cost which has no reuse.

This problem is the Steiner tree problem [20]. Thus, the second step of our algorithm finds the best paths from G_c to all the snapshots using the Steiner tree algorithm that finds a minimum cost tree *Tree*, that is, the sum of additions labelling the edges included in *Tree* is the minimum. Finally, in the third step, we bypass *ICG* nodes that have exactly one incoming and one outgoing edge in *Tree* and merge the edge addition batches for individual edges into one batch corresponding to the union of the edge batches on the incoming and outgoing edges of each bypassed ICG. In Figure 7 we can bypass nodes and combine batches Δ_+^1 and Δ_+^3 into one batch and also combine Δ_+^2 and Δ_+^6 into one batch.

The pseudocode of the above three step process is given in Algorithm 1. It should be noted that this algorithm is general in two respects. First, it handles an *arbitrary number of snapshots*, i.e. it builds the TG with appropriate number of levels. Second, since it supports bypassing, it *subsumes the direct-hop solution*, i.e. it is perfectly capable of generating the direct-hop query schedule if all the available sharing can be achieved by the G_c graph and no additional sharing opportunities exist among subsequences of snapshots.

4 COMMONGRAPH SYSTEM

In this section we discuss our system design and implementation for processing evolving graphs using *CommonGraph* algorithms. We first present an overview of the framework and then describe the core processing engine in more detail.

4.1 Framework overview

The framework consists of the storage and memory representation of the evolving graphs, as well as the set of primitives to manage the storage and the version control of the snapshots in memory during execution. We discuss these aspects of the framework next.

We use the common graph as the basis for storing the graph as a series of common graphs for each range of snapshots as well as the set of edges in the Triangular Grid for each of them. Thus, there is a minimal cost for generating common graphs while the updates are stored as sets of Δ edges corresponding to the Triangular Grid edges. For Kickstarter [44], we use its storage format based on Ligra [41].

The key feature of the data structures that we use to support evolving graphs is that we avoid the cost of updates to the graph as it evolves. As discussed in Section 3, the common graph and Δ edges are stored separately, and different versions of the graph are reached by including different subsets of the Δ edges as specified by the TG. The representation is space optimal as each edge in the system only gets represented once. In addition to avoiding expensive mutations, this organization also significantly reduces both memory footprint and memory access overheads. To access the graph, we support the primitives shown in Table 1. These primitives are used to retrieve a snapshot or find the difference between snapshots. The overhead in querying a snapshot is much lower than that in Aspen [12], which is a multi-version storage system for evolving graphs; the overhead for Aspen has up to 2× overhead than Ligra+ on static graphs.

When a new snapshots are to be created by a stream of batches, the system uses the batches to update the common graph and the TG. Specifically, the new edges (both the additions and deletions) will be removed from the common graph and additional nodes representing the new snapshot will be added to the TG.

Table 1: Common graph main primitives for query computation and graph update.

Version control API	Description
get_version(number)	Retrieve a snapshot
diff(snapshot, snapshot)	Identifies difference between two snapshots
new_version(Δ_+, Δ_-)	Create a new snapshot and update common graph
Query API	**API function**
edge_function(edge)	Algorithm specific edge function
schedule(vertex_id, mode)	Schedule active vertex
update(vertex_id)	Atomic update function

Algorithm 2 Mutation-Free Incremental Algorithm

```
1:  Inputs: common graph; streaming batches, query algorithm;
2:  Output: Query result ↪ vertex_value_array[].
3:  function INCREMENTAL COMPUTATION
4:      for edge in streaming batches do
5:          if edge_function(edge) == True; then
6:              update(destination);
7:              schedule(dst, mode);
8:          end if
9:      end for
10:     while scheduler is not empty do
11:         for vertex in scheduler do
12:             for edges ∈ common graph+update batch do
13:                 if edge_function(edge) == True; then
14:                     schedule(vertex, mode);
15:                 end if
16:             end for
17:         end for
18:     end while
19: end function
```

4.2 Execution Engine

The system executes a query targeting multiple snapshots in two steps: *a scheduling phase*; and *a computation phase*. The scheduler derives the query execution plan following Algorithm 1, this step is not needed for direct hop. The query execution phase is divided to two parts: initial computation of the query on the common graph and incremental update to add the batches to reach the next graph, and eventually all the snapshots. In the initial stage, computation happens on the common graph only, with active vertices pushing information to neighbors. The computation iterates till the graph stabilizes. For the second phase, we extend the Kickstarter streaming algorithm as shown in Algorithm 2. Specifically, the system first starts with streaming batches (lines 4-9), the destination vertices are updated and scheduled based on the edge function. Next, the scheduled vertices repeatedly push updates to their outgoing neighbors and new vertices are scheduled and updated (Algorithm 2,

lines 10-18). The steps in the algorithm shown in red are APIs for programming the engine.

A major difference between common graph engine and other streaming engines is it takes a graph with batches of streaming edges as input instead of doing computation on just single version. The edge function in Table 1 is applied both on common graph and streaming batches. This strategy avoids expensive graph mutation operation with reasonable overhead.

4.3 Scheduler design

Scheduler referenced in Algorithm 2 on lines 10-11 incorporates another policy in our system. The key idea is to switch between synchronous and asynchronous mode. For large streaming batch, the scheduler will work in synchronous mode, and vertex updates will take effective in next iteration. For a small streaming batch, the scheduler is set to asynchronous mode and updated vertex value will be available in the current iteration.

5 PERFORMANCE EVALUATION

All experiments are conducted on a shared memory system, which contains 56 Intel Xeon E5-2680 processor and 520GB memory. The **CommonGraph** system is compiled by g++ 7.3.1 and runs on CentOS Linux 7.

We evaluate *CommonGraph* on five benchmarks (all monotonic algorithms). The benchmarks are shown in Table 3, along with their push operations, which is the primary difference between the benchmarks. We use the four input graphs shown in Table 2. Update batches consisting of edge additions and deletions are generated for each benchmark to transition from one snapshot to the next, and the evaluation targets a number of snapshots specified with each experiment.

Table 2: Edges and Vertices of the Input Graphs.

Input Graph	| Edges |	| Vertices |	|Avg degree|
LiveJournal (LJ) [3]	70M	4M	28.26
DBpediaLinks (DL) [2]	170M	18M	18.85
WikipediaLinks (Wen) [27]	400M	13M	64.32
Twitter (TTW) [28]	1.5B	41M	70.51

Our first experiment tracks the execution time for evaluating a query of each of the five benchmarks on 50 consecutive snapshots. Each snapshot is separated from the next by a batch of 75,000 edge changes split *evenly* between additions and deletions, which represents approximately 0.01% of the number of edges in LiveJournal, the smallest benchmark among our input graphs. The first row for each benchmark in the table is baseline *KickStarter*: we start from the initial snapshot and stream in the batches to reach the next snapshot repeatedly until we reach the final snapshot. The second row for each benchmark uses *CommonGraph* but *Direct-Hop*s the additions to reach each snapshot in a single batch; although the snapshots can be processed in parallel, we report the total sequential time. Finally, the last row introduces the maximal *Work-Sharing* algorithm based upon building the TG and solving the Steiner tree to identify the paths that minimize the overall work to be able to reach all the snapshots. Snapshots that share subsets of their path, share the processing to reach the node of the Triangular grid where they diverge; we are computing each addition batch once for the snapshots that share it.

Table 3: Benchmarks and their Push Operations. CASMIN(a; b) sets a = b if b < a atomically; CASMAX is similarly defined. The algorithms are Breadth First Search (BFS), Single Source Widest Path (SSWP), Single Source Narrowest Path (SSNP), Single Source Shortest Path (SSSP), and Virterbi.

Algorithm	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	$CASMAX(Val(v), Val(u)/wt(u,v))$

In Table 4, we can see the speedup for *CommonGraph* with Direct-Hop (direct traversal to each snapshot); it outperforms the baseline KickStarter 1.02×-7.91×; even though it processes a higher number of edges compared to KickStarter, all these edges are additions and benefit also from parallelism among additions since they are processed in a single batch. Moreover, some of the benefits come from avoiding the cost of graph mutation through our graph representation. Additional speedup is achieved using work sharing, for an overall speedup of 1.38x-8.17x over baseline KickStarter.

The next set of experiments evaluate the scalability of the direct hop and work sharing algorithms with respect to two different scaling parameters, number of snapshots and the batch size. We used our biggest graph (TTW) and four benchmarks (BFS, SSSP, SSWP, and SSNP) for the scalability experiment. In the first experiment we fix the batch size to 75K graph updatesand varied the number of snapshots from 5 to 50. As we can see from Figure 8, the execution time of *CommonGraph* based algorithms is superior to that of Kickstarter; the execution time for all three algorithms increases linearly with the number of snapshots.

We also observe that for fewer snapshots the direct-hop algorithm works better than work-sharing. At a smaller number of snapshots, the degree of work sharing is small as each batch is reused by at most 2 snapshots. At the same time, having to stabilize the solution at an intermediate common graph reduces the amount of parallelism available in executing a larger number of updates concurrently. However, work sharing significantly outperforms direct hop when we increase the number of snapshots beyond 23 to 35 for different benchmarks.

In Figure 9, we show the second analysis for the scalability. In this analysis, we fix the total number of graph updates and vary the batch size – smaller batch size corresponds to more snapshots and more accurate picture of changes in query results. As shown in the Figure 9, we start with 75K batch sizes for 50 snapshots and then increase the batch size to 375K for 10 snapshots. For the bigger batch size, the direct-hop algorithm works better compared to the work-sharing, and for the smaller number of the batch size, the work-sharing works better. This is because for smaller batch size we have a greater number of snapshots and the benefit of TG grows due to increased opportunities for sharing.

We also varied the ratio of additions and deletions to show that across different ratios our common graph representation provides

Table 4: Average Execution Times in Seconds for KickStarter with Both Additions and Deletions, and the speedup of CommonGraph Direct Hop and CommonGraph Work-Sharing Algorithm over KickStarter for 50 Snapshots.

Graph	Query Evaluation Algorithm	BFS	SSSP	SSWP	SSNP	Viterbi
LJ	KickStarter Time	3.43s	3.88s	3.69s	3.75s	5.17s
	Direct-Hop Speedup	1.58×	1.07×	1.23×	1.18×	1.02×
	Work-Sharing Speedup	1.86×	1.43×	1.38×	1.43×	1.62×
DL	KickStarter Time	27.22s	27.64s	27.91s	27.51s	31.87s
	Direct-Hop Speedup	7.09×	7.45×	7.3×	6.7×	7.91×
	Work-Sharing Speedup	7.17×	8.17×	7.64×	7.21×	8.17×
Wen	KickStarter Time	4.65s	4.59s	4.72s	4.20s	2.03s
	Direct-Hop Speedup	4.53×	1.32×	2.73×	2.08×	3.24×
	Work-Sharing Speedup	4.68×	2.42×	3.31×	2.40×	3.8×
TTW	KickStarter Time	10.91s	11.73s	11.32s	11.31s	15.30s
	Direct-Hop Speedup	3.09×	2.36×	2.52×	1.85×	2.85×
	Work-Sharing Speedup	3.35×	2.94×	3.14×	2.62×	3.42×

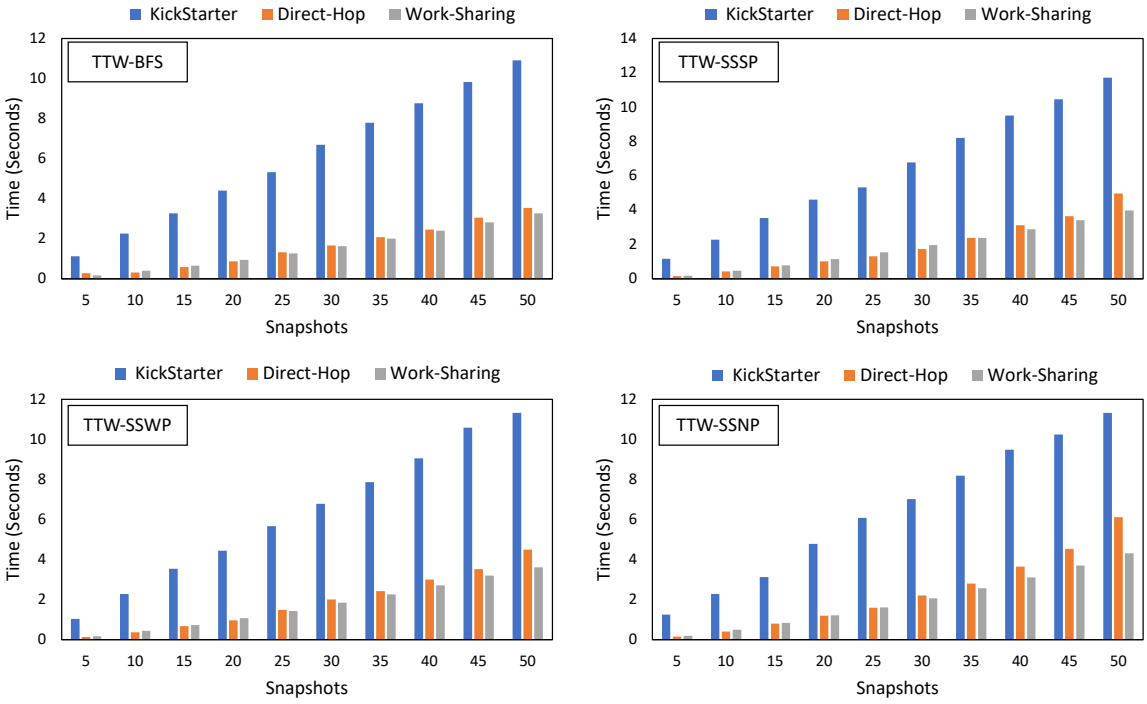

Figure 8: Execution time for KickStarter, CommonGraph Direct-Hop, and Work-Sharing.

gains over KickStarter. In Figure 10 we show the speedups by varying batches from from 150K additions and 50K deletions to 50K additions and 150K deletions. As we can see, as greater percentage of deletion updates are considered, the speedup of Direct-Hop over KickStarter increases.

CommonGraph also exposes opportunities for parallelism that are difficult to realize in streaming only systems such as Kickstarter. Specifically, we can execute the direct hop algorithm in parallel to reach each of the snapshots independently from the common graph. In contrast, Kickstarter processes the snapshots sequentially, making it difficult to parallelize the processing. In Table 5, we show the time for the longest direct hop evaluation to reach any of the 50 snapshots. Given a system with sufficient cores, this is an estimate of the overall run time of these embarrassingly parallel evaluations. We can see that there is an opportunity to achieve speedups upwards of 2 orders of magnitude compared to Kickstarter. We note that it is also possible to parallelize the work sharing version of common graph, resulting in a more work efficient algorithm.

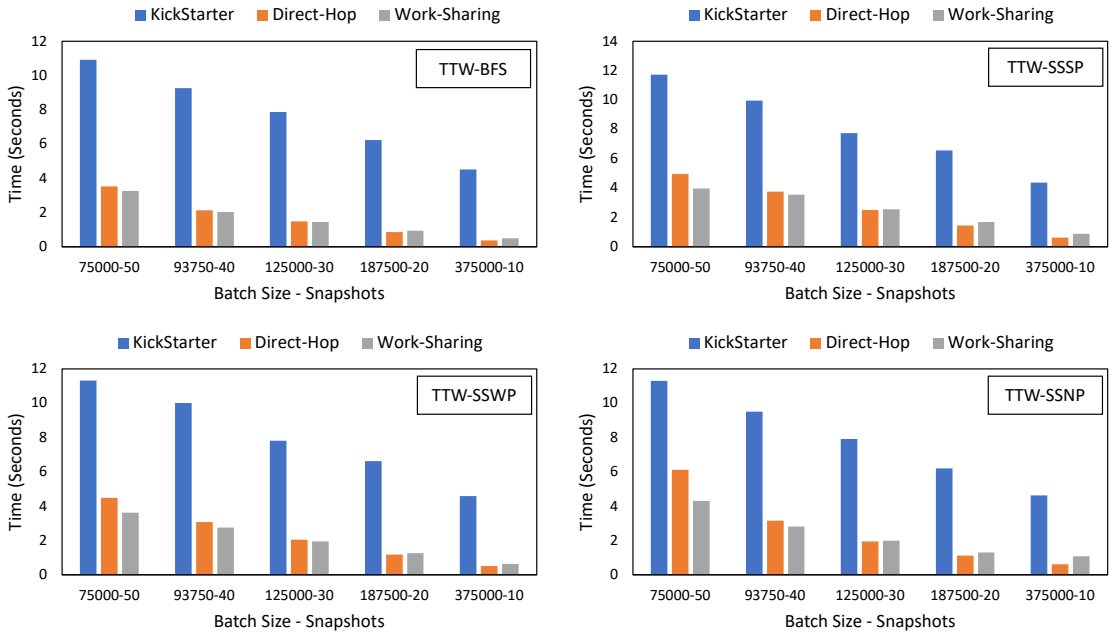

Figure 9: Execution times while batch size is varied, total number of graph updates is fixed.

Table 5: Execution times (seconds) and speedups of parallel implementation of the Direct-Hop algorithm over KicKStarter.

G	Direct-Hop	BFS	SSSP	SSWP	SSNP	Viterbi
LJ	Time	0.044s	0.072s	0.06s	0.063s	0.101s
	Speedup	78.76×	53.73×	61.56×	59.25×	51.14×
DL	Time	0.077s	0.074s	0.076s	0.082s	0.08s
	Speedup	354.93×	372.52×	365.37×	335.19×	395.64×
Wen	Time	0.021s	0.07s	0.034s	0.04s	0.041
	Speedup	226.17×	66.06×	136.89×	104.12×	161.85×
TTW	Time	0.071s	0.099s	0.09s	0.122s	0.107s
	Speedup	154.54×	118.09×	126.11×	92.5×	142.39×

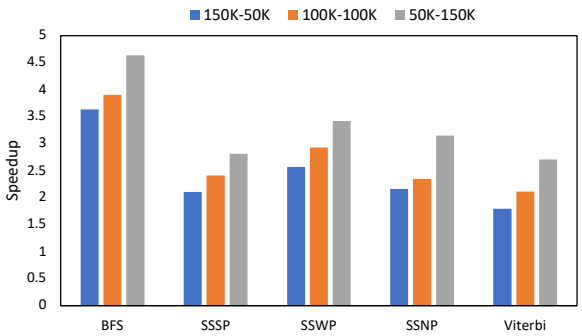

Figure 10: Sensitivity of performance to the ratio of additions and deletions.

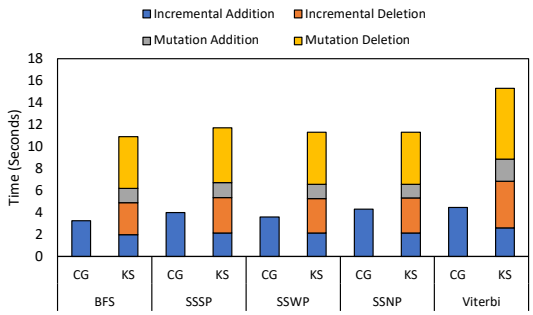

Figure 11: Breakdown of the execution time for TTW graph for the baseline KickStarter (KS) and CommonGraph (CG) Work-Sharing algorithm.

Figure 11 shows the breakdown of the execution time for the TTW graph both for KickStarter and *CommonGraph* . As we can see, the *CommonGraph* approach eliminates the mutation time for additions and deletions as well as the incremental deletion time. The incremental addition time for *CommonGraph* is lower than the combined incremental addition and deletion time in KickStarter. Thus, we observe that benefits of *CommonGraph* result from both reductions in computation cost and mutation cost.

6 RELATED WORK

In this section we summarize the existing dynamic graph systems and accelerators that aim at providing flexible graph storage and fast incremental concurrent graph querying.

6.1 Evolving & Streaming Graphs Frameworks

Among the most recent works on rapid analysis of evolving graphs are RisGraph [16] and Tegra [22]. RisGraph achieves impressive query evaluation speeds by developing a new data structure to support fast edge insertions and deletions. However, this is achieved at the expense of 3.25× to 3.38× increased memory costs via precomputed indexes that are necessary to support both fast insertions and deletions. Tegra [22] provides a novel API for performing ad-hoc queries on arbitrary time windows of the graph. It accelerates query evaluation using a compact in-memory representation for both graph and intermediate computation state. Both RisGraph and Tegra leverage existing algorithms developed for streaming systems to support incremental computation for handling edge additions and deletions. Other storage systems to support evolving and streaming graphs include GraphOne [26] and Aspen [12] while systems that amortize the cost of memory accesses and computation include Chronos [19] and FA+PA [43].

A number of systems for streaming graphs have been proposed. These algorithms maintain a single version of the graph and a standing query's results that are incrementally updated when a batch of updates are applied to the graph. The focus of these works is on incremental computation, i.e. how to efficiently update query results. Early streaming systems (such as Kineograph [9], Naiad [33], Tornado [40] and Tripoline [23]) only support incremental computations for edge additions while more recent systems (such as Kickstarter [44] and GraphBolt [32]) also support edge deletions.

Note that even though many of the above evolving and streaming systems support both edge additions and deletions, they pay a high cost for supporting deletions, as we showed in the comparison to Kickstarter. *CommonGraph* is the first system to convert deletions into additions for evolving graph analysis and thus reduce the cost of graph mutation as well as incremental computation (via work sharing) significantly.

6.2 GPUs and Other Accelerators

Recent works have begun to exploit accelerators to speedup up graph algorithms. Much of this work is aimed at static graphs (e.g., Gunrock [46], CuSha [24, 25],Tigr [36], Subway [39] etc.) and the problem addressed is to map irregular graph computation to regular GPU architectures. Specialized graph accelerators have also been developed for both static graphs [1, 11, 34, 37] and streaming graphs [4, 38, 47]. Some recent works have begun to support dynamic graphs in accelerators such as GraSu [45]. However, to our knowledge, no work has been done to exploit single accelerators to address evolving graph analysis. We expect that the tremendous memory and computational demands of evolving graphs will require development of multi-accelerator systems.

7 CONCLUDING REMARKS

Graph analytics on a dynamic graph that evolves over large time scales is a challenging problem. A user is typically interested in queries that span potentially large time windows, which translates into having to solve sub-queries targeting snapshots of the graph within those windows. We propose new algorithms that significantly improve the performance of evolving graphs compared to state-of-the-art streaming graph systems. In particular, we observe that deletions are significantly more expensive than additions, and that streaming from one original snapshot limits opportunities for work sharing. We propose *CommonGraph*, a representation of an evolving graph that captures the part of the graph that is common among a group of snapshots. Moving from this graph to any snapshots can be accomplished by adding the missing edges needed for the particular snapshot. We also show that *CommonGraph* exposes opportunities for work sharing among snapshots that share groups of edges, and capitalize on this opportunity using a Triangular Grid structure, that enables optimal work sharing when computing queries across a sequence of snapshots. Finally, we observe that streaming implementations incur substantial cost to mutate the graph as it changes, and come up with a representation that enables composing representations in place to represent the different snapshots without mutation. Taken together, our ideas result in 1.38× to 8.17× improvement in the evaluation of five query types, an advantage that grows with the number of snapshots being analyzed.

We believe that *CommonGraph* offers additional opportunities and advantages. It breaks the sequential dependency in streaming algorithms since we are able to move to each snapshot independently of the prior ones. This offers opportunities for parallel execution to further improve performance. It also enables efficiencies in storage and query execution: for example, it enables efficient range queries without having to start from an initial stored snapshot that can be far from the start of the range, and therefore requires substantial overhead just to reach the first snapshot. We intend to pursue these ideas in future work

ACKNOWLEDGMENTS

We thank all the reviewers for their valuable feedback. This work is supported in part by National Science Foundation Grants CNS-1955650, CNS-2053383, CCF-2028714, CCF-2002554 and CCF-2226448 to the University of California, Riverside.

REFERENCES

[1] Junwhan Ahn, Sungpack Hong, Sungjoo Yoo, Onur Mutlu, and Kiyoung Choi. 2015. A Scalable Processing-in-memory Accelerator for Parallel Graph Processing. *SIGARCH Comput. Archit. News* 43, 3 (June 2015), 105–117. https://doi.org/10.1145/2872887.2750386

[2] Sören Auer, Christian Bizer, Georgi Kobilarov, Jens Lehmann, Richard Cyganiak, and Zachary Ives. 2007. DBpedia: A Nucleus for a Web of Open Data. In *The Semantic Web*, Karl Aberer, Key-Sun Choi, Natasha Noy, Dean Allemang, Kyung-Il Lee, Lyndon Nixon, Jennifer Golbeck, Peter Mika, Diana Maynard, Riichiro

Mizoguchi, Guus Schreiber, and Philippe Cudré-Mauroux (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 722–735.

[3] Lars Backstrom, Dan Huttenlocher, Jon Kleinberg, and Xiangyang Lan. 2006. 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* (Philadelphia, PA, USA) *(KDD '06)*. ACM, New York, NY, USA, 44–54. https://doi.org/10.1145/1150402.1150412

[4] Abanti Basak, Zheng Qu, Jilan Lin, Alaa R. Alameldeen, Zeshan Chishti, Yufei Ding, and Yuan Xie. 2021. Improving Streaming Graph Processing Performance Using Input Knowledge. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture* (Virtual Event, Greece) *(MICRO '21)*. Association for Computing Machinery, New York, NY, USA, 1036–1050. https://doi.org/10.1145/3466752.3480096

[5] Maciej Besta, Marc Fischer, Vasiliki Kalavri, Michael Kapralov, and Torsten Hoefler. 2021. Practice of streaming processing of dynamic graphs: Concepts, models, and systems. *IEEE Transactions on Parallel and Distributed Systems* (2021).

[6] E. Bullmore and O.Sporns. 2009. Complex brain networks: graph theoretical analysis of structural and functional systems. In *Nature Reviews Neuroscience, 10(3)*. 186–198.

[7] P. Burnap, O. F. Rana, N. Avis, M. Williams, W. Housley, A. Edwards, J. Morgan, , and L. Sloan. 2015. Detecting tension in online communities with computational Twitter analysis. In *Technological Forecasting and Social Change, 95*. 96–108.

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

[9] Raymond Cheng, Ji Hong, Aapo Kyrola, Youshan Miao, Xuetian Weng, Ming Wu, Fan Yang, Lidong Zhou, Feng Zhao, and Enhong Chen. 2012. Kineograph: Taking the Pulse of a Fast-Changing and Connected World. In *Proceedings of the 7th ACM European Conference on Computer Systems* (Bern, Switzerland) *(EuroSys '12)*. Association for Computing Machinery, New York, NY, USA, 85–98. https://doi.org/10.1145/2168836.2168846

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

[11] Vidushi Dadu, Sihao Liu, and Tony Nowatzki. 2021. *PolyGraph: Exposing the Value of Flexibility for Graph Processing Accelerators*. IEEE Press, 595–608. https://doi.org/10.1109/ISCA52012.2021.00053

[12] Laxman Dhulipala, Guy E Blelloch, and Julian Shun. 2019. Low-latency graph streaming using compressed purely-functional trees. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 918–934.

[13] M. De Domenico, A. Lima, P. Mougel, and M. Musolesi. 2013. The anatomy of a scientific rumor. In *Scientific Reports, http://dx.doi.org/10.1038/srep02980*.

[14] David Ediger, Rob McColl, Jason Riedy, and David A Bader. 2012. Stinger: High performance data structure for streaming graphs. In *2012 IEEE Conference on High Performance Extreme Computing*. IEEE, 1–5.

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

[16] Guanyu Feng, Zixuan Ma, Daixuan Li, Shengqi Chen, Xiaowei Zhu, Wentao Han, and Wenguang Chen. 2021. RisGraph: A Real-Time Streaming System for Evolving Graphs to Support Sub-millisecond Per-update Analysis at Millions Ops/s. In *Proceedings of the 2021 International Conference on Management of Data*. 513–527.

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

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

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

[20] Frank K Hwang and Dana S Richards. 1992. Steiner tree problems. *Networks* 22, 1 (1992), 55–89.

[21] H. Isah, P. Trundle, , and D. Neagu. 2014. Social media analysis for product safety using text mining and sentiment analysis. In *14th UK Workshop on Computational Intelligence (UKCI)*.

[22] Anand Padmanabha Iyer, Qifan Pu, Kishan Patel, Joseph E. Gonzalez, and Ion Stoica. 2021. TEGRA: Efficient Ad-Hoc Analytics on Evolving Graphs. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. USENIX Association, 337–355. https://www.usenix.org/conference/nsdi21/presentation/iyer

[23] Xiaolin Jiang, Chengshuo Xu, Xizhe Yin, Zhijia Zhao, and Rajiv Gupta. 2021. 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*. ACM, 17–32. https://doi.org/10.1145/3447786.3456226

[24] Farzad Khorasani, Rajiv Gupta, and Laxmi N. Bhuyan. 2015. Scalable SIMD-Efficient Graph Processing on GPUs. In *Proceedings of the International Conference on Parallel Architectures and Compilation (PACT '15)*. 39–50. https://doi.org/10.1109/PACT.2015.15

[25] Farzad Khorasani, Keval Vora, Rajiv Gupta, and Laxmi N. Bhuyan. 2014. CuSha: vertex-centric graph processing on GPUs. In *Proceedings of the 23rd International Symposium on High-Performance Parallel and Distributed Computing (HPDC '14)*. ACM, 239–252. https://doi.org/10.1145/2600212.2600227

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

[27] Jérôme Kunegis. 2013. KONECT – The Koblenz Network Collection. In *In Proceedings of International Conference on World Wide Web Companion, May 13–17, 2013, Rio de Janeiro, Brazil*. ACM, 1343–1350.

[28] Haewoon Kwak, Changhyun Lee, Hosung Park, and S. Moon. 2010. What is Twitter, a social network or a news media?. In *WWW '10*.

[29] Aapo Kyrola, Guy E. Blelloch, and Carlos Guestrin. 2012. GraphChi: Large-Scale Graph Computation on Just a PC. In *10th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2012, Hollywood, CA, USA, October 8-10, 2012*. USENIX Association, 31–46. https://www.usenix.org/conference/osdi12/technical-sessions/presentation/kyrola

[30] N. Laptev and S. Amizadeh. 2015. Yahoo anomaly detection dataset S5. In *http://webscope.sandbox.yahoo.com/catalog. php?datatype=s&did=70*.

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

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

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

[34] L. Nai, R. Hadidi, J. Sim, H. Kim, P. Kumar, and H. Kim. 2017. GraphPIM: Enabling Instruction-Level PIM Offloading in Graph Computing Frameworks. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 457–468. https://doi.org/10.1109/HPCA.2017.54

[35] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. 2013. A Lightweight Infrastructure for Graph Analytics. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (Farminton, Pennsylvania) *(SOSP '13)*. 456–471. https://doi.org/10.1145/2517349.2522739

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

[37] S. Rahman, N. Abu-Ghazaleh, and R. Gupta. 2020. GraphPulse: An Event-Driven Hardware Accelerator for Asynchronous Graph Processing. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 908–921. https://doi.org/10.1109/MICRO50266.2020.00078

[38] Shafiur Rahman, Mahbod Afarin, Nael Abu-Ghazaleh, and Rajiv Gupta. 2021. JetStream: Graph Analytics on Streaming Data with Event-Driven Hardware Accelerator. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture* (Virtual Event, Greece) *(MICRO '21)*. Association for Computing Machinery, New York, NY, USA, 1091–1105. https://doi.org/10.1145/3466752.3480126

[39] Amir Hossein Nodehi Sabet, Zhijia Zhao, and Rajiv Gupta. 2020. Subway: minimizing data transfer during out-of-GPU-memory graph processing. In *Proceedings of the Fifteenth EuroSys Conference (EuroSys '20)*. 12:1–12:16. https://doi.org/10.1145/3342195.3387537

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

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

[42] L. Takac. 2012. DATA ANALYSIS IN PUBLIC SOCIAL NETWORKS.

[43] K. Vora, R. Gupta, and G. Xu. 2016. Synergistic Analysis of Evolving Graphs. In *ACM Transactions on Architecture and Code Optimization (TACO), Volume 13, Issue 4, Article No. 32, 27 pages*.

[44] Keval Vora, Rajiv Gupta, and Guoqing Xu. 2017. Kickstarter: Fast and accurate computations on streaming graphs via trimmed approximations. In *Proceedings of the twenty-second international conference on architectural support for programming languages and operating systems*. 237–251.

[45] Qinggang Wang, Long Zheng, Yu Huang, Pengcheng Yao, Chuangyi Gui, Xiaofei Liao, Hai Jin, Wenbin Jiang, and Fubing Mao. 2021. GraSU: A fast graph update library for FPGA-based dynamic graph processing. In *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 149–159.

[46] Yangzihao Wang, Yuechao Pan, Andrew A. Davidson, Yuduo Wu, Carl Yang, Leyuan Wang, Muhammad Osama, Chenshan Yuan, Weitang Liu, Andy T. Riffel, and John D. Owens. 2017. Gunrock: GPU Graph Analytics. *ACM Transactions on Parallel Computing* 4, 1 (2017), 3:1–3:49. https://doi.org/10.1145/3108140

[47] Jin Zhao, Yun Yang, Yu Zhang, Xiaofei Liao, Lin Gu, Ligang He, Bingsheng He, Hai Jin, Haikun Liu, Xinyu Jiang, and Hui Yu. 2022. TDGraph: A Topology-Driven Accelerator for High-Performance Streaming Graph Processing. In *Proceedings of the 49th Annual International Symposium on Computer Architecture* (New York, New York) *(ISCA '22)*. Association for Computing Machinery, New York, NY, USA, 116–129. https://doi.org/10.1145/3470496.3527409

[48] Xiaowei Zhu, Wentao Han, and Wenguang Chen. 2015. 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*. USENIX Association, 375–386. https://www.usenix.org/conference/atc15/technical-session/presentation/zhu

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