IncBoost: Scaling Incremental Graph Processing for Edge Deletions and Weight Updates

Xizhe Yin
University of California, Riverside
Riverside, California, USA
xyin014@ucr.edu

Zhijia Zhao
University of California, Riverside
Riverside, California, USA
zhijia@cs.ucr.edu

Rajiv Gupta
University of California, Riverside
Riverside, California, USA
rajivg@ucr.edu

ABSTRACT

Incremental query evaluation is key to efficiently processing rapidly changing graph data. By focusing on the parts of the query results affected by updates, it avoids unnecessary computations, allowing for faster query evaluation. While this technique works well in the cases of edge insertions, its benefit quickly diminishes when the volumes of *edge deletions* and *edge weight updates* increases.

To address the above scalability issue, this work introduces several techniques for handling large update batches that include many edge deletions and weight updates. First, for edge deletions, this work introduces a *bottom-up dependency tracing* method to identify the affected vertices. Unlike the existing top-down tracing, it completely avoids traversing the underlying graph, thus more scalable for large deletion batches. Second, for edge weight updates, existing graph systems treat each weight change as an edge deletion (with old weight) followed by an edge insertion (with new weight). This "two-round" method is computationally excessive. This work shows that it is, in fact, possible to handle weight updates *directly*. Finally, this work shows the benefits of adjusting the processing strategy according to the update volume. We integrated the above ideas into a graph system called `IncBoost`. Based on our evaluation, `IncBoost` can scale incremental query evaluation to large update batches that represent 30-60% of the graph size. By contrast, the state-of-the-art streaming graph system (`RisGraph`) typically fails to yield benefits when the batch size reaches 5-15% of the graph size. Regarding the absolute processing time, `IncBoost` consistently outperforms `RisGraph` with 3.1× and 5.2× speedups for edge deletions and weight updates on large batches, respectively.

SoCC '24, November 20–22, 2024, Redmond, WA, USA

ACM ISBN 979-8-4007-1286-9/24/11.
https://doi.org/10.1145/3698038.3698524

CCS CONCEPTS

• **Theory of computation** → **Dynamic graph algorithms**; • **Information systems** → *Computing platforms*; • **Computing methodologies** → *Parallel computing methodologies.*

KEYWORDS

graph processing, incremental query evaluation

ACM Reference Format:
Xizhe Yin, Zhijia Zhao, and Rajiv Gupta. 2024. IncBoost: Scaling Incremental Graph Processing for Edge Deletions and Weight Updates. In *ACM Symposium on Cloud Computing (SoCC '24), November 20–22, 2024, Redmond, WA, USA.* ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/3698038.3698524

1 INTRODUCTION

Graph processing is essential for many applications where data naturally forms graph-like structures, such as social networks and web analytics [18]. While most existing graph system research has focused on static graphs, real-world graphs are often dynamic. For example, on social networks, users join, connect, and interact with each other over time. New friendships, status updates, and interactions constantly change the graph structure [6]. In online recommendation systems, as users rate, review, or interact with items, the graph that represents the user-item relationships evolves [46]. More obviously, transportation networks, like road networks or airline networks, undergo constant changes due to factors like traffic patterns, road closures, and flight schedules [25].

Motivated by the dynamic nature of real-world graphs, a series of systems have been proposed recently for changing graphs, such as `Kineograph` [5], `Chronos` [13], `Tornado` [40], `KickStarter` [45], `Aspen` [7], `GraphBolt` [27], `Ingress` [11], `Tripoline` [16], and more recently `RisGraph` [10]. Instead of re-evaluating the graph queries from scratch, most of these systems incrementally update query results in response to the changes to the graph.

For path-based algorithms like single-source shortest path (SSSP), the state-of-the-art incremental approaches, such as `RisGraph` [10], have shown great scalability—handling large batches of edge insertions up to 30-50% of the graph size (see Figure 1). However, it remains a fundamental challenge

Figure 1: Scalability of Incremental Evaluation. The maximum update batch relative to the full graph size, where incremental evaluation is faster than full query evaluation, is demonstrated using the single-source shortest/widest path (SSSP/SSWP) algorithms and the LiveJournal graph.

to scale the incremental evaluation for edge deletions and weight updates. As shown in Figure 1, the existing solution can only scale the batches of edge deletions and weight updates up to 10-15% of the graph size.

Efficient handling of substantial graph updates, especially edge deletions and weight updates, is crucial for real-world analytics. For example, in dynamic communication networks, link weights signify key attributes like latency, bandwidth, reliability, or cost. These attributes fluctuate over time based on user demands and network conditions, causing frequent graph updates [14, 17, 35]. In evolving graph analysis, when comparing two temporally distant snapshots of the same graph, the extent of changes between them can be huge, leading to a large update batch (e.g., 30% edges of the Stack Overflow temporal network [24]).

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

- *Expensive dependency tracing.* When edges are deleted, the graph system needs to identify all affected vertices. Intuitively, the system can maintain the dependencies among vertices and trace down them from the deleted edges [10, 45] (see Section 2.2). However, we find that existing systems suffer from a mismatch between the *top-down* dependency tracing and the commonly used bottom-up representation of dependencies (e.g., parent array), making it the dominating cost of edge deletion handling (70-80%). Moreover, the overhead of *top-down* tracing is exacerbated by the high communication cost in a distributed environment.
- *Two-round weight update handling.* The existing graph systems [10, 45] handle edge weight changes in two rounds: delete the edges with their old weights, then reinsert them with the new weights. While being an intuitive workaround, this solution often involves a large amount of unnecessary computations.
- *Unawareness of workload.* The state-of-the-art graph systems employ a single processing strategy for batch updates of varying sizes. However, large changes in batch size lead to different computation characteristics, potentially resulting in sub-optimal performance.

To concur each of the aforementioned challenges, this work introduces three new techniques correspondingly:

- First, to align with the bottom-up representation of dependency data, this work introduces a novel ***bottom-up dependency tracing*** strategy. Unlike the top-down tracing, this new strategy totally avoids accessing the graph data during the dependency tracing.
- Second, this work proposes a ***direct weight change handling*** method to avoid the two-round handling. The key is to separate weight increases and decreases and treat them like edge insertions and deletions. For correctness, this work designs a monotonicity-based test to find out the right treatment to a weight increase or decrease across different graph algorithms.
- Finally, ***workload-adaptive evaluation*** is introduced to address the changing behaviors of computations due to changes in the workload volume. It automatically selects the dependency tracing strategy and the data representation to realize the best performance.

To integrate the above techniques, we implemented a new graph system for incremental query evaluation—`IncBoost`. Our evaluation focuses on comparing the performance of `IncBoost` against the state-of-the-art system, `RisGraph`. Our results show that `IncBoost` can boost the update batch size from 10-15% to 50-60% of the graph size for edge deletions and weight changes (as shown in Figure 1) without losing the benefits of incremental evaluation. More specifically, for large update batches, our results indicate up to 1.6× speedup in dependency tracing with the bottom-up approach, while the direct weight update handling delivers 2.1× speedup over the two-round approach. We also demonstrate that, in the distributed environment, the bottom-up approach provides more performance benefits by reducing communication costs. Overall, `IncBoost` achieves up to 3.1× and 252× speedups for edge deletions, 5.2× and 345× speedups for edge weight updates over `RisGraph` and `KickStarter`, respectively.

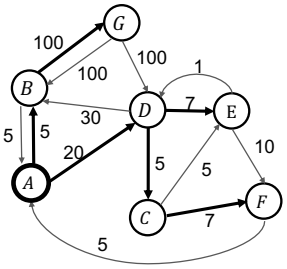
2 BACKGROUND

2.1 Vertex-Centric Programming

In a directed weighted graph $G(V, E)$, where V is a set of vertices and E is a set of edges, an edge e is in the form of (u, v, w), where u is the source vertex, v is the destination

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

(a) vertex function

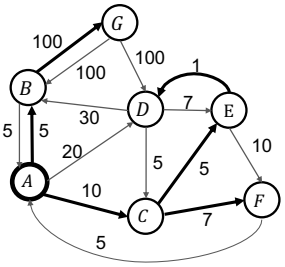


(b) a directed graph

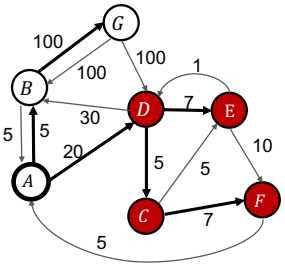
Iter#	A	B	C	D	E	F	G	Frontier
0	0	∞	∞	∞	∞	∞	∞	{A}
1	0	5	∞	20	∞	∞	∞	{B, D}
2	0	5	25	20	27	∞	105	{C, E, G}
3	0	5	25	20	27	32	105	{F}
4	0	5	25	20	27	32	105	{}

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

Figure 2: Full Evaluation of Query *SSSP*(A). Thick edges are dependent edges for the given query.



(a) after inserting edge AC



(b) after removing edge AC

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

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

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

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

Figure 3: Incremental Evaluation of *SSSP*(A). Vertices affected by the deletion of edge AC are in red.

vertex, and w is edge weight. An undirected graph can be represented by a directed graph with edges in both directions. Thus, we only consider directed graphs in this paper.

In vertex-centric programming [41], $f(v)$ is a *vertex function* that specifies the logic to compute the property values of vertex v. If $f(v)$ computes the property value of v based on values of v's in-neighbors, it is referred to as the *pull* model; Otherwise, if $f(v)$ computes the property values of v's out-neighbors based on v's value, it is called the *push* model. In this work, we assume the push model for its better efficiency for most iterative graph algorithms [41]. Taking SSSP as an example, Figure 2-(a) shows the pseudocode of its vertex function written using the push model.

Given a graph and a source vertex v_0, the evaluation of $SSSP(v_0)$ proceeds in iterations, as shown in Figure 2-(c). In each iteration, the vertex function $f(v)$ is applied to a subset of vertices known as the *frontier*. Initially, only the source vertex v_0 is added to the frontier. The vertices whose values are changed in the iteration would be added to the frontier for the next iteration. These iterations terminate once vertices values have stopped changing. These converged values are the shortest distances from vertex v_0 to all the other vertices.

2.2 Existing Incremental Methods

To avoid the expensive full evaluation each time after the graph is updated, incremental query evaluation has been proposed [7, 10, 11, 16, 27, 40, 45]. Next, we present the basic ideas of incremental query evaluation with respect to the three types of graph updates[1]: (i) edge insertions, (ii) edge deletions, and (iii) weight updates. We will focus on the widely studied monotonic path-based graph algorithms and use the example in Figure 2 to help explain the ideas.

Edge Insertion Handling. Assume a new edge $(A, C, 10)$ is inserted to the graph in Figure 2-b. The edge creates a new way to reach C through A which may result in a better value for C. To find it out, we can apply the vertex function on A but limit its scope to only the out-neighbor C (like an edge function). Based on vertex A's prior result, which is 0 (see Figure 2-c) and the weight of the new edge "10", a new best value "10" is found for C. Next, we need to propagate this new value of C to the other vertices in the graph. To achieve this, we can put C to the frontier and resume the iterative query evaluation, as illustrated by Figure 3-c. Once all values are converged again, the latest shortest distances are found.

Edge Deletion Handling. To handle edge deletions, the graph system needs to maintain the dependencies among vertices that capture how the vertex values are computed. Consider the example in Figure 3-a, thick edges reflect the dependencies among the final values of vertices. Take vertex

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

D as an example, its final value "16" is computed based on the final value of E, "15", so D depends on E. For path-based graph queries, the dependencies form a tree rooted at the query's source vertex (more details in Section 3). Figure 3-a shows a dependency tree (thick edges) rooted at vertex A. In general, there are three steps in handling an edge deletion.

① *Dependency tracing.* If a deleted edge does NOT dictate any dependenices, then no vertices are affected; Otherwise, it requires finding the affected vertices. Consider the graph in Figure 3-a, deleting edge DE has no effects on the value of any vertex. However, deleting edge AC may impact the values of vertices that depend on this edge. First, the directly impacted vertex is C. Without edge AC, C's prior value "10" is no longer valid, so do the values of other vertices that depend on C, including E, F, and D (see Figure 3-a). To ensure correctness, their values need to be *reset* to ∞ (see Figure 3-d).

② *"Jump-start".* This step finds a *safe approximation* value—no better than the best value, for each reset vertex. One way is to "pull" values from their in-neighbors and use them to update the values of these reset vertices [10]. In Figure 3-d, the "**init**" row shows the initial values of reset vertices after applying a *pull* operation. Note that at the "**init**" iteration, a pull operation is applied to each of four reset vertices (C, D, E, and F) to get a safe approximation. These pull operations are performed in parallel. As some of the vertices are adjacent, the approximation results depend on the order in which these pull operations are performed. The example shows where vertices C, E, and F are evaluated before D, resulting in their values being set to infinity. However, alternative evaluation orders are possible, such as when D is processed before C, which would assign vertex C an approximate value of 25 at the "**init**" iteration.

③ *Re-convergence.* The graph system then resumes the iterative evaluation until all values re-converge. During this time, the value propagation only occurs to the reset vertices as others do not depend on the deleted edge(s).

Among the above three steps, dependency tracing often dominates the total handling time in the existing systems (about 70-80% for large deletion batches).

For both edge insertion and deletion, the correctness is ensured by the safe approximation of affected vertices' values and the monotonicity of the iterative graph algorithms [45]. Taking SSSP as an example, during an iterative evaluation of $SSSP(u)$, the value of every vertex—the shortest path distance from the source u to this vertex—never increases.

Edge Weight Change Handling. Existing graph systems [10, 27, 45] treat an edge weight change as two separate updates: an edge deletion and an edge insertion and process them in two rounds. While simplifying the design, the two-round method may incur a lot of unnecessary computations.

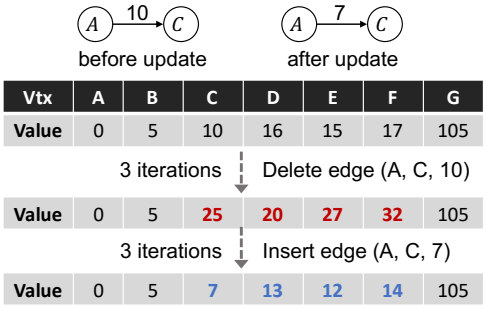


Figure 4: Two-Round Handling of Weight Update.

At the high level, the two-round handling always "takes a detour" to reach the final convergence. As illustrated by the example in Figure 4, when the weight of edge AC is changed from 10 to 7, the two-round handling first makes the values of affected vertices worse (larger values in SSSP) in the first round, then re-converges them to better values in the second round. This "detour" greatly limits the efficiency of incremental edge weight change handling.

One Configuration for All Cases. Finally, existing graph systems with incremental evaluation [10, 27, 45] use the same configuration—the same representation of the frontier and dependency tracing direction, regardless of the workload. Based on our observation, the lack of workload adaption often leads to sub-optimal performance.

In the upcoming sections, we address these limitations. In Section 3, we introduce the new dependency tracing strategy for large deletion batches. In Section 4, we present the direct method for handling weight updates. Lastly, in Section 5, we discuss the adaptive processing scheme based on batch size.

3 DEPENDENCY TRACING

Dependency tracing is necessary in handling edge deletions for incremental query evaluation. It helps the graph system identify vertices affected by the deletion of a set of edges. In this section, we discuss the data representation options for dependency, then introduce the strategies for dependency tracing, including the existing *top-down tracing* and our new design—*bottom-up tracing*. We first define the dependency in our context more formally:

Definition 1 (Dependency). *Given a monotonic iterative graph algorithm and a graph, when the algorithm converges, if the value of a vertex v_i is determined solely by the value of one of its in-neighbors v_j [2], vertex v_i depends on v_j. We refer to v_i as the dependent child while v_j as the parent.*

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

Table 1: Average-case Complexities (d_{in} and d_{out} are the average in-degree and out-degree, respectively).

Representation	Children (per vtx)	Parent (per vtx)	Maint. (per vtx)	Space (all)
Sparse children vec.	$O(1)$	$O(d_{in})$	$O(d_{out})$	$O(\|V\|)$
Dense children vec.	$O(d_{out})$	$O(d_{in} * d_{out})$	$O(d_{out})$	$O(\|E\|)$
Children hashtable	$O(1)$	$O(d_{in})$	$O(d_{out})$	$O(\|V\|)$
Children in nbr. vec.	$O(1)$	$O(d_{in} * d_{out})$	$O(d_{out})$	$O(1)$
Parent vec.	$O(d_{out})$	$O(1)$	$O(d_{in})$	$O(\|V\|)$
Depen. discovery[3]	$O(d_{out})$	$O(d_{in})$	-	-

For vertex-centric graph queries, the dependency relations among vertices form either a tree structure (e.g., in the case of SSSP) or a forest (e.g., in weakly connected components). Hereinafter, we assume a single dependency tree scenario without loss of generality.

3.1 Dependency Representation

Although the data structure for dynamic graphs has been widely discussed [7, 10, 45], there is limited discussion on the data representation for vertex dependency.

Our discussion covers the lookup costs for dependent children and parents, the maintenance cost, and the space overhead. The children lookup is used for identifying affected vertices during dependency tracing, while the parent lookup could be used when changing the parent of a vertex. The maintenance is to update the dependency tree to reflect new dependencies after the graph is updated and the query is incrementally evaluated. In total, we examined six design choices for storing dependency, under the assumption that the dynamic graph is stored in an adjacency vector:

- *Sparse children vector* stores the dependent children (their indices in the neighbor vector) in a vector.
- *Dense children vector* combines a boolean vector with the neighbor vector to indicate dependent children.
- *Children hashtable* stores the dependent children of a vertex in a separate hashtable (unordered set).
- *Children in neighbor vector* puts the dependent children at the beginning of the neighbor vector, separated from the rest neighbors with a pointer.
- *Parent vector* stores the dependent parent (index in the vertex array) of each vertex in a vector.
- *Dependency discovery* discovers a vertex's dependency online by checking the values of in-neighbors and finds the one that determines its value.

Table 1 lists the average time and space complexities based on the average in-degree d_{in} and out-degree d_{out} of the graph. In general, children-based representations offer constant access time to the dependent children (except for the dense children vector), but take longer to find the parent of a vertex as they have to examine each in-neighbor of the vertex to check if it has this vertex as a child (i.e., $O(d_{in} * d_{out})$). In comparison, parent vector offers constant access time to the parent of a vertex, but takes longer to find the children of a vertex as it needs to scan the out-neighbors of the vertex and check which one has this vertex as the parent. Dependency discovery does not explicitly store the dependency, so its maintenance and space costs are zero, but it requires extra computations to find dependent children. Specifically, for each vertex v, it needs to pull values from its in-neighbors and identify the specific in-neighbor v_n^* that determines the current value of v (e.g., dist[v] == dist[v_n^*] + w(v_n^*, v) in SSSP)—v_n^* is the parent of v, and the complexity is $O(d_{in})$. Similarly, the children of vertex v can be discovered by examining its value relation with the values of its out-neighbors.

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

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

Offline vs. Online Maintenance. The maintenance costs listed in Table 1 assume an offline approach, that is, the dependency tree is updated after the incremental evaluation finishes. For children-based representations, this can be done by traversing the out-neighbors of each vertex to identify the new dependent children. Likewise, for parent vector, it needs to traverse the in-neighbors of each vertex to identify the new dependent parent. In both cases, the offline approach needs to access the graph structure which could be costly.

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

3.2 Top-down Dependency Tracing

Existing systems, like KickStarter [45] and RisGraph [10], follow a top-down dependency tracing strategy to identify

[3]Unlike other cases where the operations are simple graph accesses, this requires accessing the vertex values and perform re-computations.

Algorithm 1 Top-down Dependency Tracing

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

all the vertices affected by edge deletions, as outlined in Algorithm 1. They use the *parent vector* as the dependency representation for easier dependency maintenance and low parallelization cost, as discussed in previous work [10].

When edges are deleted from the graph, there is a set of vertices whose values are directly impacted, denoted as S_0. Top-down tracing starts from vertices in S_0 and traverses the dependency tree downwards till reaching its leaves. At last, the algorithm outputs the visited vertices as the full set of impacted vertices S. Figure 6-a illustrates the top-down dependency tracing, where S_0 = {B, D}. After the tracing, the impacted vertex set $S = \{B, C, D, E\}$.

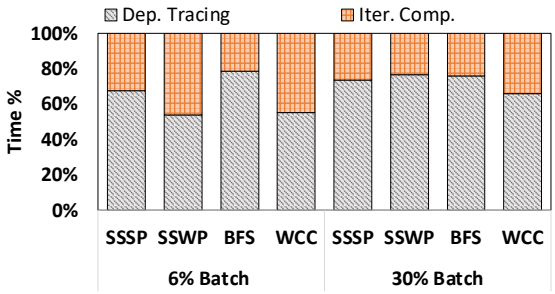


Figure 5: Time Breakdown: Dependency Tracing and Iterative Computation. RisGraph on TW graph, batch sizes are 6% and 30% of graph edges.

Scalability Issues. Top-down tracing works well when the directly impacted vertex set S_0 is relatively small. When S_0 becomes larger, we observed significant performance degradation—causing the dependency tracing to take up to 80% of the total handling time, as illustrated in Figure 5.

We found the key issue limiting the scalability of top-down tracing is the *mismatch between top-down tree traversal and the use of a parent vector*. The former needs the dependent children of vertices, but given the parent vector, it has to access the graph—scanning the out-neighbors of a vertex to find its dependent children (Line 7-8 in Algorithm 1), which takes $O(d_{out})$ time, where d_{out} is the average out-degree of the graph. When there are many directly impacted vertices, the cost of graph traversing becomes significant.

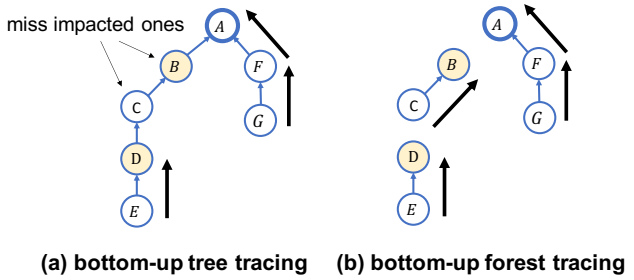


Figure 6: Complexities in Bottom-up Tracing. Vertices directly impacted by edge deletions are highlighted.

Children-based representations avoid the above issue by explicitly storing dependent children for each vertex. While this offers constant time to access the dependent children, it takes more time to remove a dependent child from the dependent children list, and requires locks during parallel children updates. Overall, the costs outweigh the benefits, hence the prior systems [10, 45] still chose to use the parent vector despite the mismatch.

3.3 Bottom-up Dependency Tracing

To address the mismatch mentioned above, we propose a *bottom-up* dependency tracing strategy that aligns well with the parent vector-based dependency representation, and it requires no accessing to the graph, making it a promising solution for large edge deletion batches.

However, it is less intuitive to traverse from the leaves of a dependency tree "backwards". To achieve that, we need to address two key questions: ① how to identify the leaf vertices of the dependency tree? ② how to correctly find impacted vertices starting from the leaves?

① *Identifying leaves.* In fact, leaves are the vertices with no dependent children. If we know the count of dependent children for every vertex, it becomes trivial to identify the leaves. However, to collect the dependent children count for a vertex v, it seems that we may still need to know which out-neighbors are the dependent children of v—going back to the same situation as in the top-down dependency tracing.

The workaround for the above issue is to maintain the dependent children counts of vertices incrementally, instead of computing them from scratch. We keep a copy of the old

parent vector before graph updates and compare it with the new parent vector afterwards. If the parent of a vertex v is changed from p_1 to p_2, we decrement the count of p_1 and increment the count of p_2. For parallel count updates, atomic operations (like `fetch_add` and `fetch_sub`) are required.

The high-level idea of our solution may look similar to children-based representations—instead of maintaining the list of dependent children, our solution maintains the count of dependent children. However, this difference brings a few critical advantages for an efficient implementation:

- First, the count is a single integer that can be easily stored in a vector for all the vertices;
- Second, maintaining the counts of dependent children does not need to access the graph;
- Lastly, the safety of parallel updates can be achieved with atomic operations, instead of employing locks as in the children-based representations.

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

② *Finding impacted vertices.* An intuitive idea is to start from the leaves of the dependency tree and traverse the tree bottom up. A traversal path stops if it reaches a vertex directly impacted by the edge deletions (i.e., a vertex from S_0 in Algorithm 1) or the root of the tree.

However, there is a caveat to the above idea—along one bottom-up traversal path, there could be multiple vertices directly impacted by edge deletions, as shown in Figure 6-(a). Stopping the traversal at the place where the first directly impacted vertex is found might miss out other impacted ones that appear even higher up in the path.

To address the above issue, we first remove the edges in the dependency tree that correspond to the deleted edges in the graph. As a consequence, the tree is broken into a set of smaller trees—a forest, as illustrated by Figure 6-(b). Meanwhile, we update the counts of dependent children to reflect the latest set of leaves—the leaves in the forest, denoted as S_{leaves}. If we traverse the forest from S_{leaves} bottom up, it would cover all the impacted vertices: $S = \cup_{v \in S_{leaves}} S_v$ where S_v denotes the vertices along the path from v to a directly impacted vertex in S_0.

There is another complexity in the bottom-up tracing: at the beginning of the traversal, it is unknown if it will finally reach a directly impacted vertex in S_0 or the original root of the dependency tree. As a result, it cannot decide if a visited vertex should be marked (i.e., added to S_v). To address this, we first assume every traversal path can eventually reach the original root, so no visited vertices are marked during this traversal. In cases where the assumption fails—the traversal did encounter a directly impacted vertex, our algorithm would re-traverse this path from the leaf, and this time it marks the vertices visited along the path as the impacted ones. The process is outlined in Algorithm 2.

Algorithm 2 Bottom-up Dependency Tracing

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

Correctness of Bottom-up Tracing. Assume that T_{dep} is the dependency tree before edge deletions and S_{top} is the set of all impacted vertices by top-down dependency tracing. Then, we have $S_{top} = \cup_{v \in S_0} N_v$, where N_v consists of all the vertices of a sub-tree in T_{dep} rooted at v.

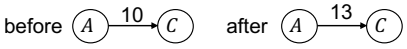
The bottom-up dependency tracing starts from all the leaves in the T_{dep}, denoted as L_{tree}, and the new "leaves" created by detaching the directly impacted vertices from their parents, denoted as L_{new}. (i) If there is one and only one directly impacted vertex appearing on the path from a leaf v, $v \in L_{tree}$, to the root of T_{dep}, the bottom-up tracing would stop at v and a follow-up re-traversing would include all the vertices along the path into the impacted vertex set S_{bottom}. (ii) If there are K, $K > 1$, directly impacted vertices on the path from a leaf v, $v \in L_{tree}$, to the root of T_{dep}, the parent detaching in bottom-up tracing would break the path into K path segments and include the bottom vertex of the last K-1 segments into L_{new}, which is also covered by the bottom-up tracing, similar to L_{tree}. (iii) If there are no directly impacted vertices on the path from a leaf v, $v \in L_{tree}$, to the root of T_{dep}, the bottom-up tracing would traverse all the way back to the root of T_{dep}, leaving no impact on S_{bottom}. In sum, the impacted vertex set of bottom-up tracing S_{bottom} covers exactly the same vertices as in S_{top}.

4 WEIGHT UPDATES HANDLING

Existing graph systems [10, 11, 45] simulate an edge weight update with an edge deletion followed by an edge insertion, which takes a "detour" to reach the final convergence. In this

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

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



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

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

Figure 7: Direct Handling of Weight Changes.

section, we show that it is possible to directly handle weight changes in a single round. We will start the discussion using SSSP, then generalize the ideas to other graph algorithms.

4.1 Case Study: SSSP

The key to directly handle weight updates is to distinguish between "weight increases" and "weight decreases". Let's revisit the example in Figure 3-a, if the weight of edge AC is decreased from 10 to 7, because AC is a dependent edge, we can tell that vertex C's value should also be updated from 10 to 7. Then, we can propagate C's new value to other vertices in the graph by resuming the iterative evaluation from vertex C. Figure 7-a illustrates this process, which is similar to the handling of an edge insertion (see Figure 3-c).

Using the same example, this time, assume the weight of edge AC is increased from 10 to 13. Again, because AC is a dependent edge, the old value of vertex C becomes invalid. Since the edge weight was increased, the new value of C may come from another different in-neighbor. So, we check all the in-neighbors of C to find out the new best value. However, some of its in-neighbors may also be impacted by this weight increase, as a result, their values should not be valid at the moment. In this case, to be safe, we need first to reset the value of C and the values of all the vertices depending on C to ∞. After this, for each reset vertex, we can safely "pull" the values from its in-neighbors to get a new *approximated* value. Finally, we need to resume the iterative evaluation starting from all the affected vertices. Figure 7-b illustrates this process, which, in fact, is similar to the 3-step handling (tracing→jump-start→re-convergence) of an edge deletion outlined in Section 2 (see Figure 3-d).

In summary, in the case of SSSP, edge weight decreases can be handled similarly to edge insertions, whereas edge weight increases can be addressed similarly to edge deletions. Also, similar to how correctness was ensured in handling edge insertions and edge deletions [45], the correctness of handling weight increases and decreases is ensured by the *safe approximation* of affected vertices' values and the inherit *monotonicity* of the SSSP algorithm. Next, we generalize this insight to some other graph algorithms.

4.2 Generalization

While the insight from SSSP is intuitive, it is non-trivial to generalize them to other graph algorithms. Taking Viterbi as an example, it is not obvious whether weight increase or decrease should be treated similarly to the edge deletion case. To address this, we again turn to the *monotonicity*—a common property shared by many iterative graph algorithms.

Definition 2 (Monotonicity). *A weight-based iterative graph algorithm is monotonic if the value of every vertex varies in such a way that it either never decreases or never increases.*

Besides SSSP, some other weight-based iterative graph algorithms that exhibit monotonicity include Single-Source Widest Path (SSWP), Single-Source Narrowest Path (SSNP), and Viterbi. Our goal here is to extend the optimization for SSSP to the other monotonic graph algorithms.

The pivotal question is: *for a weight-based monotonic graph algorithm, which of the two scenarios, weight increase or weight decrease, violates the monotonicity?* Next, we present a test to check if a weight change violate the monotonicity for a given monotonic graph algorithm.

Definition 3 (Local Monotonicity Test). *Consider a weight-based monotonic graph algorithm and a weight change (either a weight increase or a weight decrease) on edge (u, v), if (u, v) is a dependent edge, the test resets the value of the directly impacted vertex v to the initial value INIT. Then, it applies the vertex function on this edge $f(u, v)$. If the new value of vertex v is* closer *to INIT than the old value, the monotonicity is violated; otherwise, the change passes the test.*

Algorithm 3 summarizes the local monotonicity test. It returns `true` only when the weight change conforms with the monotonicity of the given graph algorithm. Table 2 lists the results of this test to the aforementioned weight-based monotonic graph algorithms using some offline synthesized examples, where the last column indicates the weight change direction that passes the test.

Correctness of Local Monotonicity Testing. The rationale behind this testing is to determine if a weight change would

Algorithm 3 Local Monotonicity Test

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

Table 2: Local Monotonicity Test Results

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

disrupt the algorithm's monotonic progression. It does so by checking whether the weight change makes the vertex value closer to the initial value (INIT). For a monotonic graph algorithm, the vertex values should never get closer to the INIT value as iterations proceed. By comparing the distances of the new and old vertex values to the INIT value (Line 6 in Algorithm 3), the test can decide if the monotonicity is violated. Upon detecting a violation, the change handling method ensures correctness by resetting the values of all impacted vertices, so that the outdated or incorrect values would never get propagated through the graph.

The above discussion is analogous to the monotonicity discussion for edge insertions and edge deletions [45], that is, edge insertions preserve the monotonicity as they may only improve the results (e.g., making the vertex values smaller in SSSP), while the edge deletions may violate the monotonicity by breaking the established value dependencies. Therefore, if a weight change passes the local monotonicity test, it can be handled like edge insertions; otherwise, it has to be treated like edge deletions (requiring dependency tracing).

With the help of local monotonicity testing, the direct weight change handling idea discussed in Section 4.1 can be generalized to all weight-based iterative graph algorithms that exhibit monotonicity, as outlined by Algorithm 4. In particular, each weight update needs to be tested (Line 5). Based on the test result, it is handled accordingly, either like edge insertion (Line 6) or edge deletion (Line 10). Note that alternatively, one may pre-determine the handling strategies for each given graph algorithm, like those in Table 2. While this can avoid some runtime testing overhead, it may limit the generality of the system.

Algorithm 4 Direct Handling for Weight Updates

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

5 WORKLOAD-ADAPTIVE EVALUATION

This section discusses the selection of the dependency tracing direction and its associated data representation based on the workload, in particular, the volume of graph updates.

5.1 Selection of Tracing Strategy

There are two major costs associated with the top-down tracing: (i) the cost of traversing the dependency tree C_{dep}^{top}; and (ii) the cost of accessing the graph (adjacency list) C_{gra}. In comparison, bottom-up tracing only needs to traverse the dependency tree. Assume its cost is C_{dep}^{btm}.

- When the update batches are small, C_{dep}^{top} and C_{gra} are relatively low as top-down tracing begins with a small set of directly impacted vertices (S_0). By traversing the dependency tree downwards, it tends to visit only a small portion of the dependency tree[4]. By contrast, C_{dep}^{btm} is a much higher cost as bottom-up tracing always starts from all the leaves of the dependency forest, its traversal covers the entire forest ($O(|V|)$).
- As the size of the graph update batches grows, so does the tracing costs C_{dep}^{top}, C_{gra}, and C_{dep}^{btm}. However, the latter grows at a much lower pace than the former two, as the bottom-up tracing anyway traverses from all the dependency tree leaves. Its cost still increases as its re-traversal cost depends on the set impacted vertices (see Lines 11-15 in Algorithm 2).

[4]The actual portion of the dependency tree that top-down tracing traverses depends on the locations of the directly impacted vertices; The closer these vertices are to the root, the larger the portion of the tree that is affected.

- When the update batches become sufficiently large, the total cost of top-down tracing, that is, $C_{dep}^{top} + C_{gra}$, would eventually surpass the cost of bottom-up tracing C_{dep}^{btm}, making the former a less promising choice.

Based on the above reasoning, we propose to select the dependency tracing strategy based on the given workload. Specifically, we define a threshold H_r for the ratio between the directly impacted vertices $|S_0|$ and the graph size $|V|$: $r_{tracing} = |S_0|/|V|$. If $r_{tracing} > H_r$, bottom-up tracing is used; otherwise, top-down tracing is employed. According to our experimental results, $H_r = 0.015$ works well in general.

Note that we chose $|S_0|$ rather than the batch size because the latter may not reflect the actual cost for dependency tracing. As pointed out by prior work [10, 45], most edges in the deletion batch are not dependent edges, thus carrying no computations. In addition, factors like the locations of deleted edges and the degrees of impacted vertices, may also affect the performance of dependency tracing. However, it is impractical to derive guidelines based on such fine-grained factors. In this work, we choose a simple and practically working policy (see Section 7 for results).

5.2 Selection of Data Representation

Another key design factor is the data representation. For bottom-up dependency tracing (see Algorithm 2), there are two ways to store the set of (deletions or weight changes) impacted vertices S: ① *Dense representation*, which uses a boolean array whose size equals to $|V|$ to indicate which vertices are impacted; ② *Sparse representation*, which directly stores the impacted vertices in a set (or a vector), whose size equals to the number of impacted vertices $|S|$.

Similar discussions can be found for selecting the frontier representation in static graph processing [19, 32, 41, 47–49]. The selection is based on an empirical threshold considering the frontier size and the number of outgoing edges [41]. In general, the sparse representation works better when the frontier size is relatively small, while the dense representation works better for relatively large frontiers. However, there is no discussion on data representations for affected vertices in dynamic graph processing. RisGraph chose a fixed scheme—sparse representation, while KickStarter chose the dense one. We use a dense representation for bottom-up tracing and a sparse one for top-down tracing. With this fixed coupling, the graph system only needs to determine the tracing strategy, simplifying the decision-making.

6 INCBOOST IMPLEMENTATION

We implemented the above proposed ideas in a new graph system, called `IncBoost`. `IncBoost` extends `Ligra` [41]—an in-memory static graph processing framework. To support dynamic graphs, we replaced the Compressed Sparse Row (CSR) [42] format used in `Ligra` with *indexed adjacency lists* (from `RisGraph` [10]). Basically, for high-degree vertices ($d(v) > 512$), their edges are indexed by a hashtable where the key is the destination vertex's ID and the value is the position of the vertex in the edge adjacency list. `IncBoost` provides a set of APIs for common graph updates, which include `EdgeDeletions`, `EdgeInsertions`, `WeightUpdates`, `VertexDeletions`, and `VertexInsertions`.

When inserting an edge (v, u), `IncBoost` appends it to the end of the source vertex v's adjacency list, then updates v's index if it is high-degree. When an edge (v, u) is deleted, `IncBoost` first swaps this edge and the last one in v's edge list, then updates v's degree and index (if applicable). In addition, `IncBoost` supports batch insertions and deletions using *batch reordering* [3] to ensure lock-free edge mutations: edges are clustered by the source vertex upon the arrival, then edges of the same source are applied serially. `RisGraph` applies all edge deletions in parallel as it does not require the swap operation but keeps deleted (tomb) edges.

`IncBoost` uses either *bottom-up* or *top-down* dependency tracing to handle edge deletions, based on the workload H_r (set to 0.015). `IncBoost` handles the edge weight updates directly with the local monotonicity test. For very small batches, to match `RisGraph`'s performance, we implemented a highly tuned sparse frontier representation.

In addition, RisGraph replaces OpenMP parallel primitives with macros when the loop size is below $8K$, avoiding the overhead associated with OpenMP parallel primitives for small batches. IncBoost adapts a similar optimization. It sets the batch size threshold for parallelism to be $8K$, below which a sequential implementation is adopted to avoid the overhead associated with parallel primitives. Regarding space usage, `IncBoost` requires an additional array of size $|V|$ for storing the number of dependent children for each vertex.

Finally, we noticed that `Aspen` [7] offers high-throughput graph mutations thanks to its uses of the compressed tree structure. But its current release only supports unweighted graphs and it does not store the in-neighbors of a vertex. In order to make it support weighted graphs, the compressed tree structure would need to be extended. Also, with only out-neighbors, `Aspen` cannot only perform pull operations, which is needed to assign safe approximated values to the affected vertices. Without pull support, `Aspen` has to perform push operations from all vertices, which would be inefficient. Although we see potential for implementing our techniques in `Aspen`, given the above limitations, it requires significant updates to `Aspen` itself before integrating our techniques.

7 EVALUATION

We compare `IncBoost` with two streaming graph systems, `KickStarter` [45] and `RisGraph` [10]. Although `RisGraph`

Table 3: Graph Statistics ("D" for directed and "T" for temporal)

Graph	Abbr.	D	T	|V|	|E|	Avg. deg.
LiveJournal [2]	LJ	✓	✗	4.8M	69M	14.2
Orkut [24]	OR	✗	✗	3.1M	234M	76.3
Wikipedia [1]	WP	✓	✗	13.5M	437M	32.2
StackOverflow [24]	SO	✓	✓	2.6M	63.5M	24.4
Wiki-Dynamic [20]	WD	✓	✓	2.2M	43.3M	19.7
Twitter [21]	TW	✓	✗	41.7M	1.5B	35.3
UK-2007 [20]	UK	✗	✗	105.2M	3.3B	31.4

focuses on processing small batches, we found that it delivers state-of-the-art performance across a wide range of batch sizes, from tiny to very large. For `KickStarter`, we chose its most recent version with graph mutation optimizations (`DZig` [26]). Both systems were configured according to the instructions from their repositories. In addition, we report the scalability of `IncBoost` and the detailed performance of different dependency tracing methods.

We conducted all the experiments on a 32-core machine with Intel Xeon E5-2683 v4 CPU and 512GB memory, running CentOS 7.9. All source code were compiled with g++ 7.3. To avoid the NUMA impacts, we used a single socket with 16 physical cores and 32 hyper-threads.

We evaluated graph systems with six path-based graph algorithms, including BFS, SSSP, SSWP, SSNP (single-source narrowest path), WCC (weakly connected components), and Viterbi[5]. Except for BFS and WCC (from Ligra [41]), all the other algorithms operate on weighted graphs.

We chose seven real-world graphs as listed in Table 3. All edge weights are integers between 1 and $\log_2 |V|$. The size of the update batch varies from small (1K edges) to medium (6% of total edges) and large (15% to 40% of total edges). The updates in the batches are sampled randomly. To perform weight updates experiments, the new weights are selected randomly within ±50% range of the old weights. For temporal (timestamped) graphs (SO and WD), the sampled batches may contain mixed updates (insertions, deletions, and weight updates), thus we report their results for mixed batches only. We chose non-trivial sources for vertex-specific queries. The reported times are the average over three runs.

We report the overall performance results in Section 7.1, the scalability results with varying batch sizes in Section 7.2, the dependency tracing performance in Section 7.3, and the performance of bottom-up tracing in a distributed graph processing system in Section 7.4.

[5]The Viterbi algorithm [23] finds the most likely sequence of hidden states (i.e., the Viterbi path) in a Hidden Markov Model (HMM), which is widely used in speech recognition [34], code decoding [44], etc.

7.1 Performance

Now we compare `IncBoost` against two existing systems in terms of incremental processing times for update batches of three representative sizes: small (1K edges), medium (6% edges), and large (30% edges), upon edge insertions, deletions, and weight change updates.

Edge Insertions. Table 4 presents the incremental query evaluation time for edge insertion across different systems. Overall, we find that `IncBoost` achieves similar performance as `RisGraph` for small batches but scales better by switching to the dense representation (for both dependency tracing and iterative evaluation). The average speedups that `IncBoost` achieves over `RisGraph` are 0.98×, 1.30×, and 2.0× for small, medium, and large batches, respectively. .

Edge Deletions. Table 5 reports the query evaluation time for incrementally handling edge deletions. Thanks to its use of workload-adaptive evaluation, `IncBoost` evaluated all small batches using *top-down* tracing while employing a *sparse representation*, and medium and large batches using the *bottom-up* tracing with the *dense representation*.

For small batches, `IncBoost` exhibits similar performance to `RisGraph`, with speedups from 0.9× to 1.3×. For medium and large batches, the speedups of `IncBoost` over `RisGraph` become more significant, ranging from 1.2× to 1.8× and from 1.9× to 3.2×, respectively, thanks to its use of the bottom-up dependency tracing. `KickStarter` is the least competitive among the three systems for all three batch sizes.

We noticed that `KickStarter` performs relatively worse for smaller batches. This is because of its use of the dense data representation for dependency tracing and iterative evaluation. As discussed earlier, for smaller update batches, the sparse data representation offers better efficiency.

Weight Updates. Table 6 reports the total handling time for edge weight updates, which covers four algorithms that require weighted graphs. On average, we found `IncBoost` is 2.6× (1.3×-5.1×) faster than `RisGraph` and 87.0× (6.8×-299×) faster than `KickStarter`. In general, the speedups follow the same trends as those in the edge deletion case.

The primary reason for the speedups of `IncBoost` is its adoption of a direct approach to handling weight changes rather than the two-round approach used by the existing graph systems. To show a more direct comparison, we also implemented the two-round approach in `IncBoost`, which we denote as `IB-2R`. Table 8 reports the detailed profiling results on batches of medium size (6% batch), including the ratio of directly impacted vertices ($|S_0|/|V|$), the ratio of all impacted vertices ($|S|/|V|$), the number of vertex activations during iterative computation (Tot. Act.), the tracing time (Tr. Time), and the iterative computation time (Iter. Time). For

Table 4: Performance of Incremental Processing (Insertion Batch)

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

		SSSP			SSWP			SSNP			Viterbi			BFS			WCC		
		S.	M.	L.	S.	M.	L.	S.	M.	L.	S.	M.	L.	S.	M.	L.	S.	M.	L.
LJ	KickStarter	5.0E-3	0.10	0.59	4.5E-3	0.07	0.45	3.0E-3	0.17	0.57	3.4E-3	0.04	0.25	4.5E-3	0.03	0.07	4.1E-3	0.03	0.11
	RisGraph	6.1E-5	0.07	0.36	7.2E-5	0.04	0.29	5.0E-5	0.04	0.19	5.6E-5	0.04	0.23	3.1E-5	0.02	0.13	3.6E-5	0.01	0.05
	IncBoost	6.9E-5	0.04	0.14	7.0E-5	0.03	0.16	5.7E-5	0.02	0.06	5.8E-5	0.03	0.09	2.9E-5	0.02	0.05	3.3E-5	0.01	0.02
OR	KickStarter	2.7E-3	0.11	0.57	2.7E-3	0.07	0.30	2.6E-3	0.05	0.27	3.0E-3	0.12	0.69	2.8E-3	0.02	0.12	2.3E-3	0.02	0.09
	RisGraph	3.0E-5	0.08	0.51	2.6E-5	0.03	0.15	2.6E-5	0.03	0.13	2.7E-5	0.04	0.19	2.3E-5	0.03	0.15	1.6E-5	0.01	0.07
	IncBoost	3.4E-5	0.04	0.24	2.4E-5	0.02	0.07	2.5E-5	0.02	0.06	2.5E-5	0.02	0.07	2.3E-5	0.02	0.10	1.2E-5	0.01	0.05
WP	KickStarter	1.3E-2	0.29	1.05	1.1E-2	0.60	2.16	1.2E-2	0.23	0.89	1.3E-2	0.28	1.22	1.08E-2	0.10	0.27	1.1E-2	0.08	0.23
	RisGraph	2.9E-4	0.15	0.85	3.2E-5	0.20	1.51	3.5E-5	0.10	0.51	3.5E-5	0.10	0.54	2.9E-5	0.07	0.38	2.0E-5	0.03	0.19
	IncBoost	2.8E-4	0.16	0.51	2.6E-5	0.20	0.89	4.1E-5	0.12	0.29	4.1E-5	0.11	0.29	3.3E-5	0.11	0.28	2.4E-5	0.05	0.21
TW	KickStarter	2.8E-2	0.29	8.38	3.6E-2	0.54	12.46	2.9E-2	0.46	9.51	2.8E-2	0.55	9.88	3.2E-02	0.39	6.75	2.9E-2	0.33	1.12
	RisGraph	4.4E-5	0.46	9.60	4.7E-5	0.22	4.23	1.6E-4	0.22	4.26	1.7E-4	0.42	9.76	1.4E-4	0.34	2.63	1.9E-4	0.14	0.71
	IncBoost	6.0E-5	0.36	3.52	5.4E-5	0.22	2.11	1.8E-4	0.23	2.51	1.8E-4	0.34	3.43	1.7E-4	0.32	2.28	2.2E-4	0.21	1.09
UK	KickStarter	3.0E-2	4.39	14.19	4.4E-2	2.76	11.21	2.8E-2	1.96	7.12	3.5E-2	3.79	12.91	2.7E-3	0.73	1.59	8.2E-3	0.81	2.41
	RisGraph	9.0E-5	2.45	9.62	8.3E-5	0.68	14.40	8.3E-5	0.75	3.30	9.4E-5	1.76	3.97	8.4E-5	1.41	5.24	5.8E-5	0.41	2.53
	IncBoost	8.7E-5	1.10	5.21	8.6E-5	0.36	2.61	8.0E-5	0.45	1.43	1.2E-4	1.11	1.57	7.7E-5	1.32	2.20	6.9E-5	0.37	1.58
Geo	vs. KS	**41.89×**	**1.87×**	**2.64×**	**253.34×**	**3.35×**	**3.80×**	**155.30×**	**3.26×**	**4.67×**	**158.87×**	**2.58×**	**4.90×**	**132.48×**	**1.01×**	**1.27×**	**176.44×**	**2.20×**	**1.72×**
	vs. Ris	**0.91×**	**1.54×**	**2.14×**	**1.03×**	**1.28×**	**2.37×**	**0.94×**	**1.27×**	**2.09×**	**0.92×**	**1.36×**	**2.47×**	**0.97×**	**1.04×**	**1.68×**	**0.97×**	**0.97×**	**1.22×**

Table 5: Performance of Incremental Processing (Deletion Batch)

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

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

IB-2R, the "Iter. Time" is the sum of iterative computation times for both deletion and insertion rounds.

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

Mixed Batches. IncBoost handles heterogeneous batches that contain mixed types of updates: edge insertions, edge deletions, and edge weight updates. RisGraph only supports batches containing a single type of update (homogeneous update batches). KickStarter supports batches of mixed insertions and deletions but prohibits batches with insertion

Table 6: Performance of Incremental Processing (Weight Update Batch)

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

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

Table 7: Performance of Incremental Processing (Mixed Updates Batch)

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

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

and deletion of the same edge. Fortunately, both systems can preprocess mixed batches into homogeneous sub-batches.

Table 7 presents the performance of evaluating mixed batches where the batch is configured as containing 50% edge insertions and 50% edge deletions when the graph is non-temporal. For temporal graphs where a timestamp is attached to each edge, we delete edges with older timestamps and insert newer ones. If an edge is inserted more than once in a batch, it is considered a weight update. Under this setup, a temporal graph update batch contains mixed updates. For example, a WD graph update batch contains 33% insertions, 51% deletions, and 16% weight updates. In general, we found that the speedups fall between those achieved in pure weight updates and pure deletion batches.

Graph Mutations. For completeness, we briefly compare the cost of graph mutations with RisGraph and Aspen [7]. Table 9 reports the throughput of the edge-related updates. While IncBoost and RisGraph show similar throughput for

Table 8: Profiling of Direct and Two-round on SSSP.

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

Table 9: Graph Mutation Throughput (graph TW)

Values are numbers of edge updates per second

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

edge insertions and deletions, both exhibit significantly lower throughput than Aspen, mainly because that Aspen utilizes the compressed tree data structure for the graph.

As to weight changes, Aspen does not support weighted graphs. IncBoost achieves a throughput roughly 2.6× higher than that of RisGraph, which is mainly due to its avoidance of the two-round handling.

Performance on Road Networks. We evaluated IncBoost on a non-power-law graph: roadNet-USA [38] ($|V|$: 24M, $|E|$: 58M). Unlike power-law graphs, road networks often have a higher vertex-to-edge ratio, and their edges are distributed more evenly across vertices. These properties lead to a high ratio of dependent edges. Thus, graph updates tend to affect a larger portion of the vertices.

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

7.2 Workload Scalability

Figure 8 reports the processing time for different batch sizes. The horizontal dotted lines are the full query evaluation times. Both systems deliver fast incremental evaluation when the batch size is relatively small (below 500K). IncBoost scales better than RisGraph—its incremental computation remains faster than the full query evaluation even for batches of sizes 30% - 40%. In contrast, RisGraph struggles to yield performance benefits when the batch size gets close to 20% for SSSP edge deletions and 15% for weight updates.

Note that although RisGraph shows superlinear behavior when processing batch sizes ranging from 6% to 30%, this does not imply that the system becomes more efficient by splitting a large batch into smaller ones. Evaluating multiple smaller sub-batches is more costly than processing a single large batch, as the former can cause many vertices to converge to unnecessary states, resulting in increased vertex activations. Additionally, a large batch may affect a significant number of vertices due to edge deletions, which do not scale linearly with the batch size.

Table 10: Dependency Tracing Time in Gemini.

	Small(1K)	Medium (6%)	Large (30%)
Top-down	0.03s	0.47s	1.27s
Bottom-up	0.07s	0.10s	0.18s

7.3 Dependency Tracing

Figure 9 shows the dependency tracing costs of IncBoost and RisGraph across different configurations. Overall, we observed the performance trend of IncBoost (top-down) mirrors that of RisGraph as both use top-down tracing.

For update batches with sizes less than 1% of the graph, we found IncBoost (bottom-up) performs the poorest. However, as the batch size approaches approximately 1.5% of the total graph size, a significant shift occurs—IncBoost (bottom-up) transforms into the fastest method. These results validate the necessity of workload-adaptive evaluation (see Section 5).

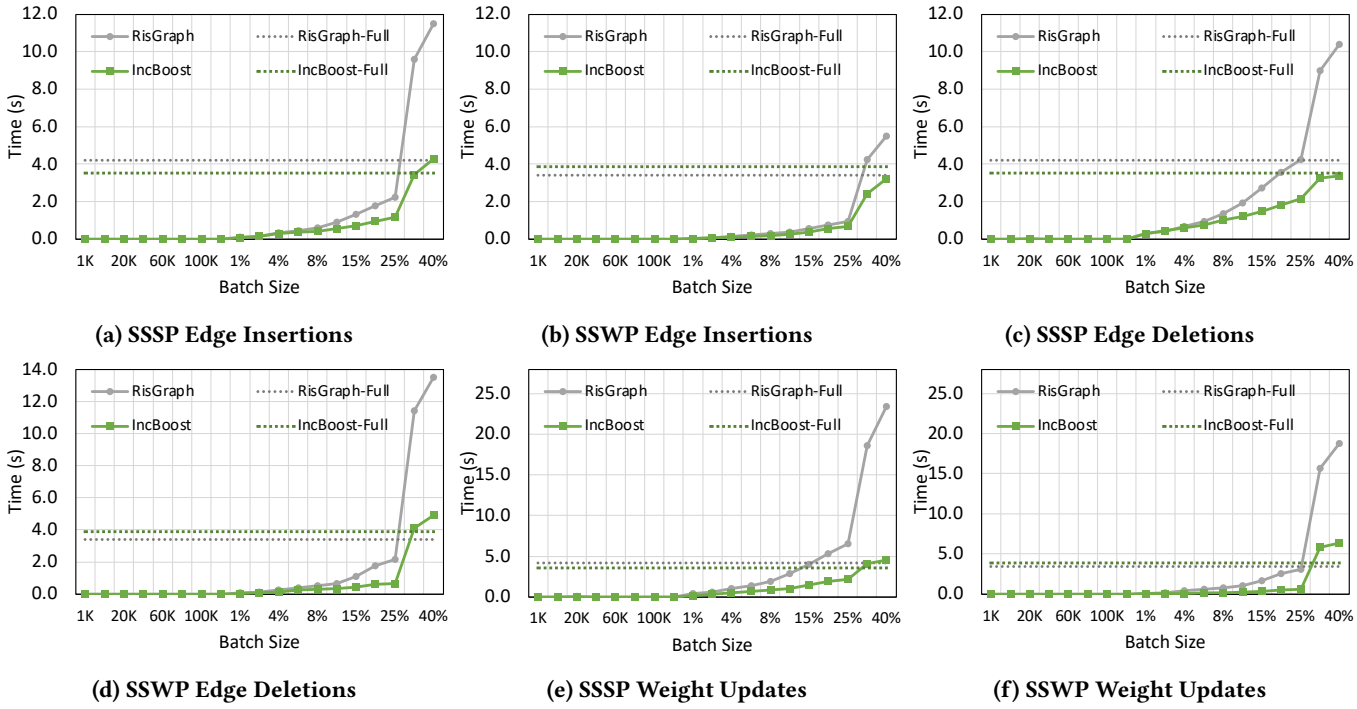
7.4 Bottom-up Tracing in a Distributed System

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

With a moderate level of effort, we integrated bottom-up dependency tracing into a state-of-the-art distributed graph processing system Gemini [48], which clearly demonstrates the applicability of our techniques in the distributed setting.

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

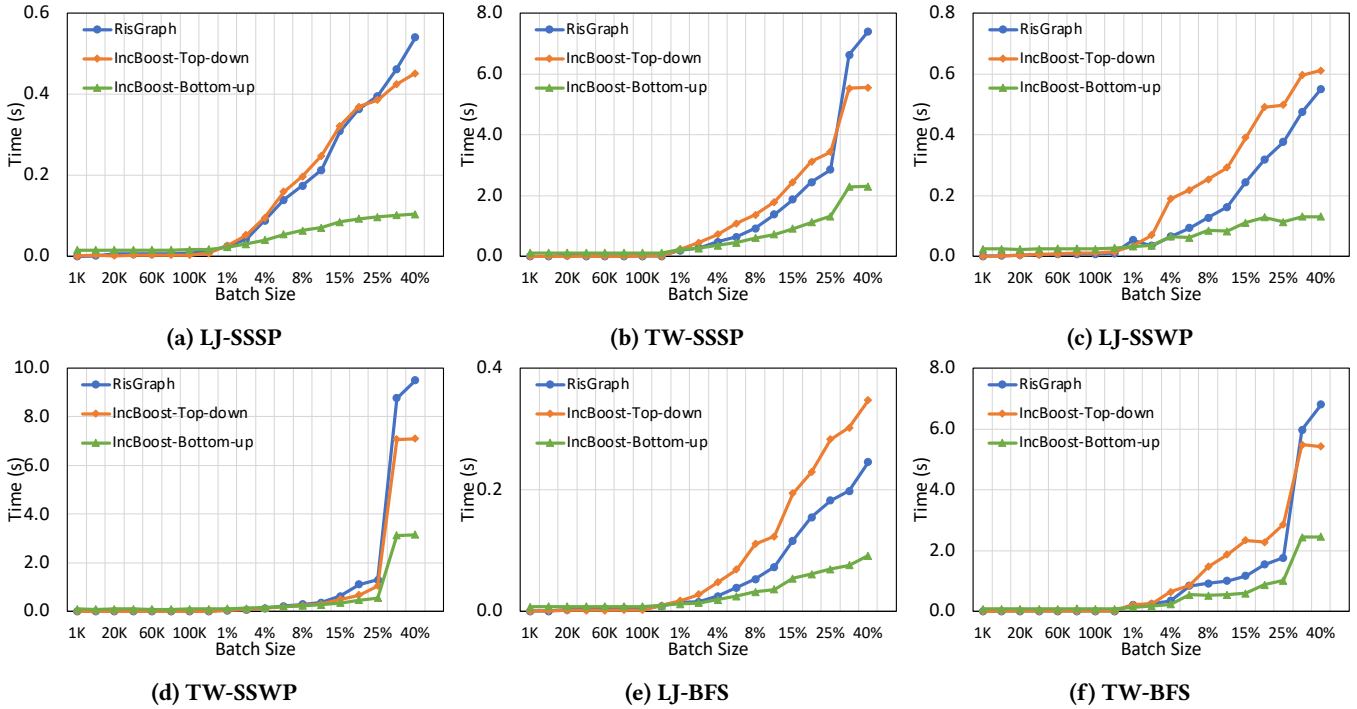
Table 10 reports the costs of dependency tracing in Gemini for SSSP on TW graph. The results cover three batch sizes (1k, 6%, and 30%). For larger batches, the bottom-up tracing



(a) SSSP Edge Insertions (b) SSWP Edge Insertions (c) SSSP Edge Deletions

(d) SSWP Edge Deletions (e) SSSP Weight Updates (f) SSWP Weight Updates

Figure 8: Scalability with Varying Batch Sizes on TW Graph.



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

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

Figure 9: Dependency Tracing Performance.

in Gemini delivers more performance improvements (4.7× to 7.1× speedup over the top-down one) than it does in a shared memory environment thanks to the additional savings in communication cost. On the other hand, top-down tracing outperforms bottom-up one when the batch size is small.

8 RELATED WORK

This section summarizes the existing systems, algorithms, and data structures relevant to querying dynamic graphs.

Graph Systems. Many existing systems for dynamic graphs employ incremental query evaluation. Earlier systems either only support incremental query evaluation in the presence of edge insertions (e.g., Chronous [13]) or compute approximate query results (e.g., Kineograph [5]). To handle edge deletions for incremental query evaluation, systems like KickStarter, RisGraph, and Tornado [40] record the value dependency for monotonic path-based algorithms (KickStarter also tracks each vertex's level in the dependency tree). Tornado utilizes Lamport Clocks [22] to guarantee consistency and correctness in a distributed environment. GraphBolt [27] and DZig [26] support both edge insertions and deletions for accumulative graph algorithms. Tripoline [16] proposes a generalized model to incrementally evaluate queries that are different from the standing queries. Ingress [11] is a system that automatically incrementalizes graph algorithms. The above systems do not address weight updates and cannot efficiently support large update batches.

Hardware accelerators for incremental computations have also been proposed [3, 36]. Differential Dataflow [28, 29] and Naiad [30] are generalized incremental computation models which are also capable of processing graph workloads.

Incremental Algorithms. To the best of our knowledge, the first incremental algorithm for handling SSSP edge deletions was described by Ramalingam et al. [37], which briefly points out the connection between the handling of weight changes and the handling of edge insertions and deletions. In this work, we generalize this connection and expand it to a wider range of path-based graph algorithms. Moreover, our work actually implemented these ideas in a graph system. Some recent works have focused on finding theoretical bounds for various incremental graph algorithms [8, 9], particularly when link weights undergo slow changes [14], and when incorporating temporal information [4].

Data Structures for Changing Graphs. There have been works for enhancing graph mutation performance. Aspen [7] supports low latency graph mutation by using a compressed tree-based graph representation. VCSR [15] leverages packed memory array (PMA) to enable graph mutation on CSR. In addition, the indexing method has been employed to improve graph mutation performance [10, 43]. Recently, Terrace [33] proposed to use a hierarchical data structure to store edges based on the vertex degree.

Handling Edge Weight Updates. Sallinen et al. [39] discussed supports for edge weight updating in SSSP, but the solution is limited only to reducing edge weight. Henzinger et al. [14] provided theoretical lower bounds for recomputing several algorithms (SSSP, maximum flow, matchings, etc.) with weight changes. It tried to answer if incremental computation can be significantly faster than recomputing from scratch given a small change of weights. Nasre et al. [31] studied the incremental betweenness centrality (BC) algorithms in graphs where edge weights are updated (decreases) or new edges are added. Gruenheid et al. [12] studied the record linkage clustering problem on changing graphs where insertion, deletion, and change (weight decreasing or increasing) operations are supported. The authors proposed that directly considering change could be more efficient than deletion and insertion for linkage clustering.

9 CONCLUSIONS

This work targets the scalability limitations in handling edge deletions and weight updates for incremental graph query evaluation. For edge deletions, it introduces a bottom-up dependency tracing strategy for handling very large update batches. For weight changes, it avoids simulating them with pairs of edge deletion and insertion by presenting a direct approach based on the monotonicity of graph algorithms. In addition, this work discusses an adaptive evaluation scheme that changes the tracing strategy based on the update volume. Finally, it demonstrates the effectiveness of the proposed ideas with a new graph system IncBoost. The evaluation results show that IncBoost is able to scale to very large update batches with sizes of 30% to 60% of the graph size.

ACKNOWLEDGMENT

We thank our paper shepherd Dr. Mingyu Gao for helping with the paper revision. This work was supported in part by National Science Foundation Grants CCF-2226448, CCF-2106383, CCF-2028714, CCF-2002554 and CCF-1813173 to the University of California, Riverside.

REFERENCES

[1] 2013. Wikipedia links, english network dataset. http://konect.cc/networks/wikipedia_link_en/. Accessed: 2022-01-02.

[2] 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*. 44–54.

[3] 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*. 1036–1050.

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

[5] 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.

[6] 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.

[7] 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.

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

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

[10] 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.

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

[12] Anja Gruenheid, Xin Luna Dong, and Divesh Srivastava. 2014. Incremental record linkage. *Proceedings of the VLDB Endowment* 7, 9 (2014), 697–708.

[13] 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.

[14] Monika Henzinger, Ami Paz, and Stefan Schmid. 2021. On the Complexity of Weight-Dynamic Network Algorithms. In *2021 IFIP Networking Conference (IFIP Networking)*. IEEE, 1–9.

[15] Abdullah Al Raqibul Islam, Dong Dai, and Dazhao Cheng. 2022. VCSR: Mutable CSR Graph Format Using Vertex-Centric Packed Memory Array. In *2022 22nd IEEE International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*. IEEE, 71–80.

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

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

[18] Jon M Kleinberg, Ravi Kumar, Prabhakar Raghavan, Sridhar Rajagopalan, and Andrew S Tomkins. 1999. The web as a graph: Measurements, models, and methods. In *International Computing and Combinatorics Conference*. Springer, 1–17.

[19] 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.

[20] Jérôme Kunegis. 2013. Konect: the koblenz network collection. In *Proceedings of the 22nd international conference on world wide web*. 1343–1350.

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

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

[23] Jüri Lember, Dario Gasbarra, Alexey Koloydenko, and Kristi Kuljus. 2019. Estimation of Viterbi path in Bayesian hidden Markov models. *Metron* 77 (2019), 137–169.

[24] Jure Leskovec and Andrej Krevl. 2014. SNAP Datasets: Stanford large network dataset collection.

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

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

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

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

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

[30] 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.

[31] Meghana Nasre, Matteo Pontecorvi, and Vijaya Ramachandran. 2014. Betweenness centrality–incremental and faster. In *International Symposium on Mathematical Foundations of Computer Science*. Springer, 577–588.

[32] 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*. 456–471.

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

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

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

[36] 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*. 1091–1105.

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

[38] Ryan A. Rossi and Nesreen K. Ahmed. 2015. The Network Data Repository with Interactive Graph Analytics and Visualization. In *AAAI*. https://networkrepository.com

[39] Scott Sallinen, Roger Pearce, and Matei Ripeanu. 2019. Incremental graph processing for on-line analytics. In *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 1007–1018.

[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*. 417–430.

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

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

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

[45] 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.

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

[47] Yunming Zhang, Mengjiao Yang, Riyadh Baghdadi, Shoaib Kamil, Julian Shun, and Saman Amarasinghe. 2018. Graphit: A high-performance graph dsl. *Proceedings of the ACM on Programming Languages* 2, OOPSLA (2018), 1–30.

[48] Xiaowei Zhu, Wenguang Chen, Weimin Zheng, and Xiaosong Ma. 2016. Gemini: A computation-centric distributed graph processing system.. In *OSDI*, Vol. 16. 301–316.

[49] Xiaowei Zhu, Guanyu Feng, Marco Serafini, Xiaosong Ma, Jiping Yu, Lei Xie, Ashraf Aboulnaga, and Wenguang Chen. [n. d.]. LiveGraph: A Transactional Graph Storage System with Purely Sequential Adjacency List Scans. *Proceedings of the VLDB Endowment* 13, 7 ([n. d.]).