GraphBolt: Dependency-Driven Synchronous Processing of Streaming Graphs

Mugilan Mariappan
School of Computing Science
Simon Fraser University
British Columbia, Canada
mmariapp@cs.sfu.ca

Keval Vora
School of Computing Science
Simon Fraser University
British Columbia, Canada
keval@cs.sfu.ca

Abstract

Efficient streaming graph processing systems leverage incremental processing by updating computed results to reflect the change in graph structure for the latest graph snapshot. Although certain monotonic path-based algorithms produce correct results by refining intermediate values via numerical comparisons, directly reusing values that were computed before mutation does not work correctly for algorithms that require BSP semantics. Since structural mutations in streaming graphs render the intermediate results unusable, exploiting incremental computation while simultaneously providing synchronous processing guarantees is challenging.

In this paper we develop GraphBolt which incrementally processes streaming graphs while guaranteeing BSP semantics. GraphBolt incorporates dependency-driven incremental processing where it first tracks dependencies to capture how intermediate values get computed, and then uses this information to incrementally propagate the impact of change across intermediate values. To support wide variety of graph-based analytics, GraphBolt provides a generalized incremental programming model that enables development of incremental versions of complex aggregations. Our evaluation shows that GraphBolt's incremental processing eliminates redundant computations and efficiently processes streaming graphs with varying mutation rates, starting from just a single edge mutation all the way up to 1 million edge mutations at a time. Furthermore, being specialized for graph computations, GraphBolt extracts high performance compared to Differential Dataflow.

Keywords Streaming Graphs, Incremental Processing

EuroSys'19, March 25–28, 2019, Dresden, Germany.

ACM ISBN 978-1-4503-6281-8/19/03…$15.00
https://doi.org/10.1145/3302424.3303974

1 Introduction

The continuously evolving nature of streaming graphs has led to the development of several systems like KickStarter [44], GraphIn [37], Tornado [38] and Kineograph [10] that enable efficient analysis over fast changing graphs. At the heart of these streaming graph processing systems is a dynamic graph whose structure changes rapidly via a stream of graph updates, and an incremental algorithm that reacts to the graph structure mutations to produce final results for the latest graph snapshot. The incremental algorithm aims to minimize redundant computations by reusing results that have been computed before the graph structure mutates.

While incremental processing makes the system more responsive, directly reusing intermediate values often causes the algorithm to produce incorrect results. Furthermore, these incorrect results become the basis upon which subsequent incremental processing gets performed, which results in increasing inaccuracies as time progresses. To address this issue, solutions like GraphIn and KickStarter incrementally transform the intermediate results to first make them consistent with the updated graph, and then use the transformed results to compute final answers. However, such transformations are often based on algorithmic properties like monotonic convergence (path lengths or component ids) which make them applicable to the selected subclass (e.g., monotonic) of graph algorithms. Generalized graph-based analytics solutions often rely on Bulk Synchronous Parallel (BSP) processing semantics since it enables easier programmability (correctness/convergence properties can be clearly reasoned with BSP), and the monotonic transformations do not apply for the general class of graph algorithms. Resorting to naive approaches based on tag propagation and values reinitialization end up tagging majority of values to be reset [44], limiting incremental reuse to only few vertices.

Generalized incremental processing systems, on the other hand, operate on unbounded structured and unstructured streams [1, 2, 4, 5, 31, 36, 42, 50, 51]. Differential Dataflow [25] incrementally processes generalized streams by enabling the impact of change in streams to be reflected via diffs (or changes). Its strength lies in its differential operators that capture and directly compute over diffs, and are general enough to support any iterative incremental computations. However, such generality comes at a performance cost when

applied to specific use-cases, like streaming graph processing; extracting performance for graph computations requires attention to detail that graph-aware runtimes can enable.

In this paper, we develop a dependency-driven streaming graph processing technique that minimizes redundant computations upon graph mutation while still guaranteeing synchronous processing semantics. To do so, we first characterize dependencies among values across consecutive iterations as defined by synchronous processing semantics, and then track these dependencies as iterations progress. Later when graph structure changes, we refine the captured dependencies iteration-by-iteration to incrementally produce the final result. Based on the above dependency-driven refinement strategy, we develop GraphBolt which incrementally processes streaming graphs and guarantees BSP semantics.

To ensure that GraphBolt's incremental processing scales well and delivers high performance, it incorporates several key optimizations. GraphBolt reduces the amount of dependency information to be tracked from $O(E)$ to $O(V)$ by translating the dependency information in form of aggregation values that reside on vertices and by utilizing the structure of input graph to deduce dependencies as needed. It further incorporates pruning mechanisms that conservatively prune the dependency information to be tracked, without causing additional analysis (e.g., backpropagation) when graph structure mutates. Finally, GraphBolt incorporates computation-aware hybrid execution that dynamically switches between dependency-driven refinement strategy and traditional incremental computation when dependency information is unavailable due to pruning.

Several analytics algorithms like Machine Learning & Data Mining (MLDM) algorithms, utilize complex aggregations that are difficult to update based on graph mutations. To support the broad class of graph algorithms beyond traditional traversal algorithms, GraphBolt provides a generalized incremental programming model that allows decomposing complex aggregations to incorporate incremental value changes. Our evaluation shows that GraphBolt processes up to 1 million edge updates in just few seconds and its incremental processing engine eliminates redundant computations for varying mutation rates, starting from just a single edge mutation all the way up to 1 million edge mutations at a time. GraphBolt compares favorably against Ligra which is a state-of-art synchronous graph processing system; furthermore, being specialized for graph computations, GraphBolt extracts high performance compared to Differential Dataflow.

2 Background & Motivation

We first discuss the semantics of synchronous execution and issues involved in incremental processing of streaming graphs, and then provide an overview of our dependency-driven incremental computation technique that accelerates processing while guaranteeing synchronous semantics.

Algorithm 1 Synchronous PageRank

```
1:  G = (V, E)                                  ▷ Input graph
2:  pr = {1, 1, ..., 1}                         ▷ Floating-point array of size |V|
3:  while not converged do
4:      newPr = {0, 0, ..., 0}                  ▷ Floating-point array of size |V|
5:      par-for (u, v) ∈ E do
6:          atomicAdd(&newPr[v], pr[u] / |out_neighbors(u)|)
7:      end par-for
8:      par-for v ∈ V do
9:          newPr[v] = 0.15 + 0.85 × newPr[v]
10:     end par-for
11:     swap(pr, newPr)
12: end while
```

2.1 Streaming Graph Processing

At the heart of efficient streaming graph processing systems, like KickStarter [44], GraphIn [37] and Tornado [38] is a dynamic graph whose structure changes rapidly via a stream of graph updates, and an incremental algorithm that reacts to the change in graph structure to produce final results for the latest graph snapshot. A streaming graph G is constantly modified by a stream of ΔG updates consisting of insertions and deletions of edges and vertices. An algorithm S iteratively computes over the latest snapshot of the graph to produce final results. To maintain consistency, updates are batched into ΔG when computations are being performed during an iteration, and they are incorporated in G before starting the next iteration.

Synchronous Processing. The Bulk Synchronous Parallel (BSP) (hereafter called *synchronous*) model is a popular iterative processing model that separates computations across iterations such that values in a given iteration are computed based on values from the previous iteration. We illustrate the synchronous processing semantics using PageRank [1] in Algorithm 1. The algorithm computes vertex values (*newPr*) using the ones computed in previous iteration (*pr*) as shown on line 6. The flow of values across iterations is explicitly controlled via swap( ) on line 11.

Such clear separation of values being generated v/s values being used in synchronous processing allows programmers to develop iterative graph algorithms more easily than with asynchronous execution since they can clearly reason about the important convergence and correctness properties. Hence, the synchronous processing model often becomes a preferred choice for large-scale graph processing [22, 27, 35, 39, 54] and in this paper we focus on efficient synchronous processing of streaming graphs for algorithms that require synchronous processing semantics.

Incremental Computation. Incremental computation enables fast processing as graph structure mutates since it reuses the results that have been computed prior to graph mutation. Such incremental processing is achieved as follows (visually depicted in Figure 1): let I be the initial values

[1] Algorithm 1 is simplified to eliminate details like selective scheduling.

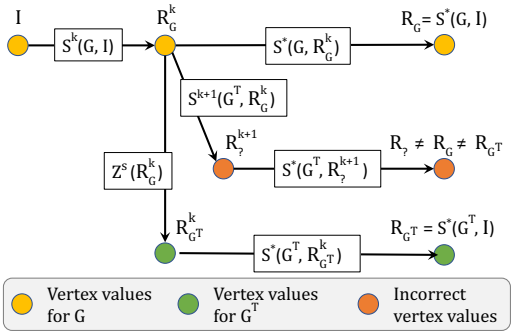


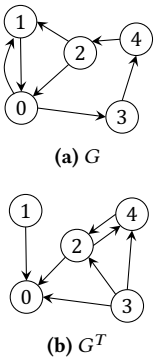
Figure 1. Incremental Processing of Streaming Graphs.

of vertices before processing starts. The iterative algorithm S computes over I and G to generate final results R_G. We use S^i to denote i iterations of S that produce intermediate results R^i_G, and S^* to denote processing until convergence to generate R_G. Hence, $R_G = S^*(G, I)$.

Assuming ΔG arrives during iteration k, incremental computation incorporates ΔG at the start of iteration $k+1$ while using intermediate results generated by S^k. Let $G^T = G + \Delta G$. Hence in iteration $k+1$, $R^{k+1}_{G^T} = S^{k+1}(G^T, R^k)$, and upon convergence, the final results become $R_{G^T} = S^*(G^T, R^k_G)$. Since incremental processing starts from R^k instead of I, it reuses the result of previous computations and converges quickly to the final results. Future updates that arrive after convergence get processed by starting from R_{G^T}. Typically, the amount of processing involved in incremental computation is aimed to be between $O(\Delta G)$ and $O(G)$ depending on the impact of ΔG on computed results. Such incremental processing has been shown to accelerate monotonic path-based algorithms over billion-scale graphs by 8.5-23.7× [44].

2.2 Problem: Incorrect Results

While incremental processing is efficient, directly reusing intermediate results often causes the algorithm to produce incorrect results. Although certain monotonic algorithms like shortest paths and breadth first search always produce correct results, algorithms that require synchronous processing semantics do not converge to correct values. Figure 2 shows a streaming graph that mutates from G to G^T, and the corresponding results for Label Propagation, a synchronous graph algorithm that we use in our evaluation. Without incremental processing, $S^*(G^T, I)$ converges to correct results; however, incrementally computing from $S^*(G, I)$ violates synchronous processing semantics and hence, converges to $S^*(G^T, R_G)$ which is incorrect. We also profiled the impact of such incremental processing using a real-world graph by streaming 10 batches of edge mutations with 100 mutations per batch. As shown in Table 1, incrementally computing from $S^*(G, I)$ causes 1.6M vertex values to be incorrect with relative error of $\geq$ 1% when only the first batch of 100 edge mutations is applied. Furthermore, this error propagates across subsequent batches and for the 10th batch, up to 59K values become incorrect by over 10% error.



(a) G

(b) G^T

	0	1	2	3	4
I	0.32	0.72	0.29	0.74	0.27
$S^*(G, I)$	0.32	0.326	0.329	0.317	0.327
$S^*(G^T, I)$	0.32	0.717	0.737	0.741	0.737

Incorrect incremental computation using values of G to compute for G^T

$S^*(G^T, R_G)$	0.32	0.326	0.322	0.317	0.321

(c) Label Propagation on G & G^T. Incorrect values are highlighted in red.

Figure 2. As G transforms to G^T, directly using results from G leads to incorrect results.

Error	B1	B2	B3	B4	B5	B6	B7	B8	B9	B10
>10%	790	2.9K	5.5K	9.3K	16.4K	24.5K	34K	43K	52K	59K
>1%	1.6M	2.3M	2.7M	2.8M	2.9M	2.9M	3M	3M	3M	3M

Table 1. No. of vertices with incorrect results (relative error $\geq$ 10% and $\geq$ 1%) for Label Propagation [53] on Wiki [47] graph for 10 batches (B1-B10) of edge mutations, 100 mutations per batch.

Directly using the intermediate values leads to incorrect results mainly because $S^*(G^T, R^k_G) \neq S^*(G^T, I)$ (as shown in Figure 1). In particular, since R^k_G represents values corresponding to G, it doesn't incorporate the impact of ΔG across previous k iterations. To address this issue, KickStarter and GraphIn incrementally transform R^k_G to make it consistent with G^T. With Z being the incremental transformation function, $S^*(G^T, Z(R^k_G))$ becomes equal to $S^*(G^T, R^k_{G^T})$. However, such transformations are often based on algorithmic properties like monotonic convergence and numerically comparable vertex values (path lengths or component ids) which allow $Z(R^k_G) \neq R^k_{G^T}$. Since Z does not guarantee equivalence between $Z(R^k_G)$ and $R^k_{G^T}$, such transformations only work for asynchronous (e.g., path-based) graph algorithms.

As shown in Figure 1, we need a special incremental transformation function Z^S that guarantees $Z^S(R^k_G) = R^k_{G^T}$, so that the subsequent computation guarantees synchronous semantics and converges to the correct final result. Since R^k_G doesn't include the information about how it got computed, Z^S cannot directly incorporate impact of ΔG in R^k_G. A straightforward Z^S is to identify the subset of values in R^k_G that need to be corrected by propagating tags (e.g., downstream vertices from addition/deletion points), and then compute their values to generate $R^k_{G^T}$, as done in GraphIn [37]; however, as shown in KickStarter [44], such tagging based approach ends up tagging majority of vertex values to be thrown out, hence limiting reuse of values to a very small fraction of vertices. This poses an important challenge: *how do we perform incremental processing of streaming graphs to minimize redundant computations upon graph mutation while still guaranteeing synchronous processing semantics?*

2.3 Overview of Techniques

To incrementally transform R_G^k, we develop a dependency-driven incremental processing model that captures how R_G^k was computed in form of value dependencies, and later uses the captured information to incorporate the impact of change in graph structure. To do so, we first characterize dependencies among values across consecutive iterations as defined by synchronous processing semantics, and then track these dependencies as iterations progress. When ΔG arrives, we refine the captured dependencies iteration-by-iteration to incrementally produce $R_{G^T}^k$ for iteration k, which is then directly used to compute forward in synchronous manner. By doing the above process, we guarantee synchronous processing semantics at the end of each iteration, hence ensuring correctness of final results.

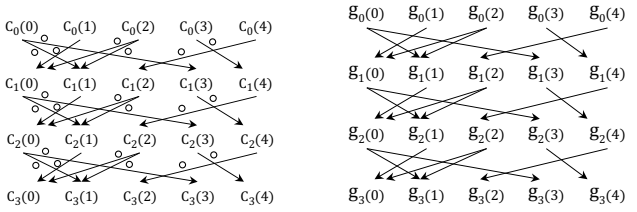
However, developing such a dependency-driven incremental processing poses several challenges. Firstly, tracking dependencies online can be very expensive since the amount of dependency information is directly proportional to $|E|$. We overcome this challenge by carefully translating the dependency information in form of *aggregation values* that reside on vertices, and by further recognizing that the structure of dependencies (i.e., how values impacts each other) can be derived from the input graph structure. This brings down the dependency information to order of $|V|$. Moreover, we observe that real-world graphs are sparse and skewed which results in aggregation values stabilizing as iterations progress. Hence, we incorporate pruning mechanisms that conservatively prune the dependency information to be tracked, without causing additional analysis (e.g., backpropagation) to recompute untracked values when ΔG arrives.

Secondly, dependency-driven incremental processing becomes difficult for complex aggregations (e.g., MLDM aggregations that operate on vectors) since their incremental counterparts are often not easy to deduce. To address this issue, we develop a generalized incremental programming model that allows decomposing complex aggregations to incorporate incremental value changes resulting from ΔG. Our programming model allows expressing the underlying workflow of decomposing complex aggregations and reproducing old contributions which get updated based on value changes, hence constructing incremental complex aggregations. Furthermore, simple aggregations like *sum* get directly expressed in our incremental programming model without going through the decomposition workflow.

Finally, we develop a computation-aware hybrid execution that dynamically switches between dependency-driven incremental processing and traditional incremental processing when dependency information is unavailable due to pruning.

3 Dependency-Aware Processing

We first characterize value dependencies and then develop our dependency-driven incremental processing technique.



(a) $\mathcal{D}_G = (\mathcal{V}_\mathcal{D}, \mathcal{E}_\mathcal{D})$ **(b)** $\mathcal{A}_G = (\mathcal{V}_\mathcal{A}, \mathcal{E}_\mathcal{A})$

Figure 3. Dependence graphs for G in Figure 2a.

3.1 Synchronous Processing Semantics

Synchronous iterative graph algorithms compute vertex values in a given iteration based on values of their incoming neighbors that were computed in the previous iteration. Since computations are primarily based on graph structure, such value dependencies can be captured via the graph structure as follows:

$$\forall (u, v) \in E, \quad u^t \mapsto v^{t+1} \tag{1}$$

where u^t and v^{t+1} represent values of vertex u in iteration t and vertex v in iteration $t+1$ respectively, and $\mapsto$ indicates that v^{t+1} is value-dependent on u^t. It is important to note that there are no dependency relationships among vertices that are not directly connected by an edge.

We can understand how synchronous processing semantics get violated as graphs mutate using value dependencies by carefully analyzing the impact of each edge mutation. If a new edge (u, v) gets added in iteration k, $\forall j < k$, $u^{j-1} \mapsto v^j$ was absent (while it should have been present), and hence v^k represents incorrect value under synchronous processing semantics. Furthermore, this incorrect v^k gets propagated via v's outgoing neighbors in subsequent iterations spreading inaccuracies across the graph. Similarly, if an existing edge (u, v) gets deleted in iteration k, $\forall j < k$, $u^{j-1} \mapsto v^j$ was present (while it should not have been present) making v^k inaccurate, which further propagates across the graph.

3.2 Tracking Value Dependencies

Let C_i represent the vertex values at the end of iteration i for the initial graph $G = (V, E)$, i.e., $\forall v \in V$, $c_i(v) \in C_i$ is the vertex value of v at the end of iteration i. Assuming G mutates to $G^T = (V^T, E^T)$ at the end of iteration L, C_L is no longer valid and must be refined to eliminate inaccuracies. Since the values that propagate through edges to satisfy dependencies together lead to values in C_L, values that have flown to satisfy prior dependencies must be examined to ensure that dependencies get correctly satisfied based on edge mutations. Hence, we aim to track the values that have contributed to computation of C_L, which we later use to correct C_L as graph mutates.

A simple way to track value dependencies is to save all the values that participate in satisfying dependencies defined in Eq. 1. Let $\mathcal{D}_G = (\mathcal{V}_\mathcal{D}, \mathcal{E}_\mathcal{D})$ be the dependency graph for computation over graph $G = (V, E)$ where $\mathcal{V}_\mathcal{D}$ captures all the intermediate values for vertices in V and $\mathcal{E}_\mathcal{D}$ captures

dependencies among intermediate values based on Eq. 1. Formally, at the end of iteration k:

$$\mathcal{V}_{\mathcal{D}} = \bigcup_{i \in [0,k]} c_i(v) \quad \Big| \quad \mathcal{E}_{\mathcal{D}} = \{ (c_{i-1}(u),\ c_i(v),\ e_i(u,v)) : i \in [0,k] \wedge (u,v) \in E \}$$

Figure 3a shows the dependency graph for G in Figure 2a over an execution of 4 iterations. As computation progresses through iterations, $\mathcal{D}_G$ increases by $|V|$ vertices and $|E|$ edges. While saving $\mathcal{D}_G$ exhaustively captures the entire execution history such that it enables incremental correction of C_L for subsequent graph mutations, such tracking of value dependencies leads to $O(|E|.t)$ amount of information to be maintained for t iterations which significantly increases the memory footprint, making the entire process memory-bound. To reduce the amount of dependency information that must be tracked, we first carefully analyze how values flowing through dependencies participate in computing C_L.

Tracking Value Dependencies as Value Aggregations. Given a vertex v, its value is computed based on values from its incoming neighbors in two sub-steps: first, the incoming neighbors' values from previous iteration are aggregated into a single value; and then, the aggregated value is used to compute vertex value for the current iteration. This computation can be formulated as [2]:

$$c_i(v) = \oint \Big(\bigoplus_{\forall e=(u,v) \in E} (c_{i-1}(u)) \Big)$$

where $\bigoplus$ indicates the aggregation operator and $\oint$ indicates the function applied on the aggregated value to produce the final vertex value. For example in Algorithm 1, $\bigoplus$ is AtomicAdd on line 6 while $\oint$ is the computation on line 9. Since values flowing through edges are effectively combined into aggregated values at vertices, we can track these aggregated values instead of individual dependency information. By doing so, value dependencies can be corrected upon graph mutation by incrementally correcting the aggregated values and propagating corrections across subsequent aggregations throughout the graph.

Let $g_i(v)$ be the aggregated value for vertex v for iteration i, i.e., $g_i(v) = \bigoplus_{\forall e=(u,v) \in E} (c_{i-1}(u))$. We define $\mathcal{A}_G = (\mathcal{V}_{\mathcal{A}}, \mathcal{E}_{\mathcal{A}})$ as dependency graph in terms of aggregation values at the end of iteration k:

$$\mathcal{V}_{\mathcal{A}} = \bigcup_{i \in [0,k]} g_i(v) \quad \Big| \quad \mathcal{E}_{\mathcal{A}} = \{ (g_{i-1}(u),\ g_i(v)) : i \in [0,k] \wedge (u,v) \in E \}$$

This allows us to separate out the structure of dependencies (i.e., u^{t-1} "impacts" v^t) from the values that participate in satisfying those dependencies (i.e., $c_{t-1}(u)$ and $c_t(v)$). Figure 3b shows the dependency graph $\mathcal{A}_G$ in terms of aggregation values. It is interesting to note that the structure of dependencies in $\mathcal{A}_G$ is directly based on the structure of input graph (see Eq. 1), i.e., $\mathcal{E}_{\mathcal{A}}$ in Figure 3b is based on G in Figure 2a. Since we are no longer tracking the values flowing through those dependency edges, we don't need to track the dependency structure as it can be later reconstructed during the refinement stage using the input graph structure. Hence, we only need to track aggregated values, i.e., $\mathcal{V}_{\mathcal{A}}$, which reduces the amount of dependency information to $O(|V|.t)$.

[2]Values residing on edges (i.e., edge weights) have been left out from equations for simplicity since they do not impact dependencies.

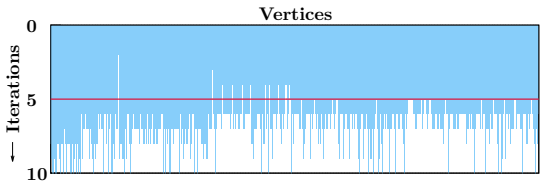


Figure 4. Change in vertex values across iterations for Label Propagation over Wiki graph. Blue pixels indicate change in vertex values.

Pruning Value Aggregations.
The skewed nature of real-world graphs [15] often cause synchronous graph algorithms to behave such that most vertex values keep on changing during the initial iterations and then the number of changing vertices decrease as iterations progress. For example, Figure 4 shows how vertex values change across iterations in Label Propagation over Wiki graph (graph details in Table 2) for a 10-iteration window. As we can see, the color density is higher during first 5 iterations indicating that majority of vertex values change in those iterations; after 5 iterations, values start stabilizing and the color density decreases sharply. As values stabilize, their corresponding aggregated values also stabilize. This provides a useful opportunity to limit the amount of aggregated values that must be tracked during execution.

We conservatively prune the dependence graph $\mathcal{A}_G$ to balance the memory requirements for tracking aggregated values with recomputation cost during refinement stage. In particular, we incorporate *horizontal pruning* and *vertical pruning* over the dependence graph that sparsify $\mathcal{A}_G$ across different dimensions. As values start stabilizing, horizontal pruning is achieved by directly stopping the tracking of aggregated values after certain iterations. For example, the horizontal red line in Figure 4 indicates the cut-off after which aggregated values won't be tracked. Vertical pruning, on the other hand, operates at vertex-level and is performed by not saving aggregated values that have stabilized. This eliminates the white regions above the horizontal red line in Figure 4. Hence, only the values corresponding to blue points are tracked after horizontal and vertical pruning. This captures the important region where changes in vertex values result in larger impact across their neighborhoods, i.e., the region where incremental processing will be most effective.

It is interesting to note that both horizontal and vertical pruning methods are conservative, i.e., they don't need further analysis about whether subset of values need to be

recomputed to be able to refine values upon graph mutation. While aggressive pruning can be performed (e.g., dropping certain vertices altogether), it would require backpropagation from values that get changed during refinement to recompute the correct old values for incremental computation.

3.3 Dependency-Driven Value Refinement

Let E_a and E_d be the set of edges to be added to G and deleted from G respectively to transform it to G^T. Hence, $G^T = G \cup E_a \setminus E_d$. Given E_a, E_d and the dependence graph $\mathcal{A}_G$, we ask two questions that help us transform C_L to C_L^T.

(A) What to Refine?

We dynamically transform aggregation values in $\mathcal{A}_G$ to make them consistent with G^T under synchronous semantics. To do so, we start with aggregation values in first iteration, i.e., $g_0(v) \in \mathcal{V}_\mathcal{A}$, and progress forward iteration by iteration. At each iteration i, we refine $g_i(v) \in \mathcal{V}_\mathcal{A}$ that fall under two categories: first, values corresponding to end points of E_a and E_d which are directly impacted by edge mutations; and second, values corresponding to outgoing neighbors of vertices whose values got refined in the previous iteration $i-1$, which captures the transitive impact of mutation. This means, we dynamically identify the aggregation values in $\mathcal{V}_\mathcal{A}$ that need to be refined as the process of refinement progresses. Note that performing incremental changes corresponding to transitive impact of edge mutations requires information about the structure of dependencies, i.e., $\mathcal{E}_\mathcal{A}$, which we directly infer by looking at the graph structure.

Figure 5 shows how the refinement process selects values to be incrementally computed for our dependency graph from Figure 3b upon addition of new edge (1, 2). In step 1, $g_1^T(2)$ is incrementally computed from $g_1(2)$ based on contribution of $g_0^T(1)$ (iteration 0 represents initial value) flowing from the new edge (solid edge). In step 2, the *change in contribution* of $g_1^T(2)$ (i.e., effect of $g_1^T(2) - g_1(2)$) gets propagated to vertex 2's outgoing neighbors 0 and 1 (dotted edges), which effectively allows $g_2^T(0)$ and $g_2^T(1)$ to compute based on $g_1^T(2)$. Since contribution of $g_1(1)$ was never propagated to $g_2(2)$, the contribution of $g_1^T(1)$ is also propagated to incrementally compute $g_2^T(2)$ (similar to in step 1). Similarly, $g_3^T(0)$, $g_3^T(1)$, $g_3^T(2)$ and $g_3^T(3)$ are incrementally computed in step 3 based on direct and transitive impact of the edge addition. As we can see, changes unroll dynamically based on: (a) the structure of $\mathcal{A}_G$; and, (b) the change in aggregation values resulting from edge mutations. Furthermore, computations during refinement process are far lesser than that involved while processing the original graph (as indicated by fewer edges in Figure 5 compared to Figure 3b) which showcases the efficacy of dependency-driven incremental processing.

(B) How to Refine?

As aggregation values in $\mathcal{V}_\mathcal{A}$ get identified to be refined, we incrementally update them based on change in values coming from incoming neighbors. Specifically, with L being the latest iteration before which graph mutates, we aim to update $g_i(v) = \bigoplus_{\forall e=(u,v)\in E} (c_{i-1}(u))$ to $g_i^T(v) = \bigoplus_{\forall e=(u,v)\in E^T} (c_{i-1}^T(u))$ for $0 \le i \le L$. This is incrementally achieved as:

$$g_i^T(v) = g_i(v) \underset{\forall e=(u,v)\in E_a}{\biguplus} (c_{i-1}(u)) \underset{\forall e=(u,v)\in E_d}{\biguplus} (c_{i-1}(u)) \underset{\substack{\forall e=(u,v)\in E^T \\ s.t.\, c_{i-1}(u)\neq c_{i-1}^T(u)}}{\biguplus} (c_{i-1}^T(u))$$

where $\biguplus$, $\biguplus$ and $\biguplus$ are incremental aggregation operators that add new contributions (for edge additions), remove old contributions (for edge deletions), and update existing contributions (for transitive effects of mutations) respectively. While $\biguplus$ often is similar to $\bigoplus$, $\biguplus$ and $\biguplus$ require undoing aggregation to eliminate or update previous contributions. We focus our discussion on incremental $\biguplus$ since its logic subsumes that for $\biguplus$. We generalize $\biguplus$ by modeling it as:

$$\underset{\substack{\forall e=(u,v)\in E^T \\ s.t.\, c_{i-1}(u)\neq c_{i-1}^T(u)}}{\biguplus} = \underset{\substack{\forall e=(u,v)\in E^T \\ s.t.\, c_{i-1}(u)\neq c_{i-1}^T(u)}}{\biguplus} (c_{i-1}(u)) \underset{\substack{\forall e=(u,v)\in E^T \\ s.t.\, c_{i-1}(u)\neq c_{i-1}^T(u)}}{\biguplus} (c_{i-1}^T(u))$$

$$= \underset{\substack{e=(u,v)\in E^T \\ s.t.\, c_{i-1}(u)\neq c_{i-1}^T(u)}}{\forall} \left(\bigoplus (c_{i-1}^T(u)) - \bigoplus (c_{i-1}(u)) \right)$$

The right-hand side of the above equation is referred to as *change in contribution* for each respective edge. Several aggregations like *sum*, *product*, etc. often simplify incremental aggregation by directly capturing the change in contributions; however, complex aggregations like operations on vectors require careful extraction of old values since differences cannot be directly formulated.

Complex Aggregations.

Machine Learning & Data Mining (MLDM) algorithms often involve complex aggregations that intricately transform vertex values, making them difficult to be computed incrementally. For example, algorithms like Belief Propagation and Alternating Least Squares operate on vectors or multi-valued variables that interact with elements of other complex variables during aggregations. We present a generalized incremental technique by explaining how such complex aggregations become incremental in two steps:

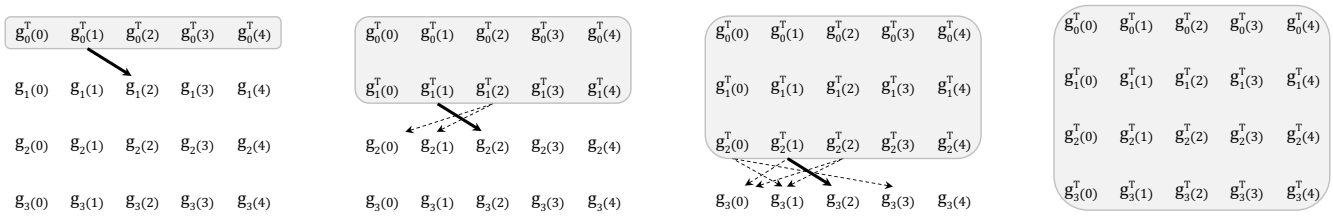
1. Static Decomposition to (Simple) Sub-Aggregations.

Complex aggregations can often be decomposed into multiple simple aggregations that act as sub-operations to perform the original complex aggregation. For example in Alternating Least Squares, the computation involving complex aggregation is:

$$c_i(v) = \left(\sum_{\forall e=(u,v)\in E} c_i(u).c_i(u)^{tr} + \lambda I_k \right)^{-1} \times \sum_{\forall e=(u,v)\in E} c_i(u).weight(u,v)$$

Ignoring the inverse operation and addition of identity matrix, the computation gets decomposed into a pair of sub-aggregations:

$$g_i(v) \quad = \langle \sum_{\forall e=(u,v)\in E} c_i(u).c_i(u)^{tr} \,,\, \sum_{\forall e=(u,v)\in E} c_i(u).weight(u,v) \rangle$$



(a) Step 1 **(b)** Step 2 **(c)** Step 3 **(d)** Final $\mathcal{V}_{\mathcal{A}}$

Figure 5. Dependency-driven value refinement for addition of new edge (1, 2). Edges indicate incremental flow of values to satisfy dependencies: solid edges indicate propagation of values while dotted edges indicate propagation of change in values.

While both sub-aggregations get decomposed to *addition*, the second sub-aggregation is simple, and hence, can be directly made incremental using *difference*, i.e., $c_i^T(u) - c_i(u)$. However, the first sub-aggregation requires further computation as described in the next step.

2. On-the-fly Evaluation of Discrete Contributions.
Since the first sub-aggregation in the above equation involves transformation of vertex values before they are summed together, we need to recompute the old contribution from vertex's value, which we then use to compute the difference in the contribution. Hence, we separately compute $c_i(u).c_i(u)^{tr}$ and $c_i^T(u).c_i^T(u)^{tr}$, and then compute their difference to be aggregated. By doing so, the combined aggregation gets incrementally computed as:

$$\biguplus_{\substack{\forall e=(u,v)\in E^T \\ s.t.\, c_{i-1}(u)\neq c_{i-1}^T(u)}} = \langle \sum_{\substack{\forall e=(u,v)\in E^T \\ s.t.\, c_{i-1}(u)\neq c_{i-1}^T(u)}} \left(c_i^T(u).c_i^T(u)^{tr}\right) - \left(c_i(u).c_i(u)^{tr}\right), \sum_{\substack{\forall e=(u,v)\in E^T \\ s.t.\, c_{i-1}(u)\neq c_{i-1}^T(u)}} (c_i^T(u) - c_i(u)).weight(u,v) \rangle$$

The process of breaking down complex aggregations is dependent on the arithmetic rules of underlying sub-operations. For example, the aggregation in Belief Propagation gets broken down based on products instead of additions.

Aggregation Properties & Extensions.
Our three incremental aggregation operators ($\biguplus$, $\biguplus^-$ and $\biguplus^{\triangle}$) allow capturing change in vertex values that propagate via edges. Firstly, the aggregation operator (and hence, these incremental operators) must be commutative and associative to relax the order in which values get combined and reverted during regular (non-incremental) and incremental computation. Furthermore, the domain of values visible to these operators includes the aggregation value at vertex v (i.e., either $g_i(v)$, or $g_i^T(v)$, or intermediate values between $g_i(v)$ and $g_i^T(v)$), and the values corresponding to a single edge (u, v) along with its source vertex u (i.e., $c_{i-1}^T(u)$, $c_{i-1}(u)$ and edge weights). This adds a restriction on the nature of aggregation that can be incrementally adjusted using our above formulation. Specifically, the aggregation must allow incrementally incorporating the impact of change from its single input (e.g., change coming from a single edge) to its final value. We classify such aggregations to be *decomposable*, examples of which include *sum* and *count*.

Contrary to these, we have *non-decomposable* aggregations like *min* and *max*, where certain changes cannot be directly adjusted if only the final aggregation result is maintained in $g_i(v)$. For example, if $c_{i-1}(u) = k$, $c_{i-1}^T(u) = p$, $p > k$ and $k = g_i(v)$, the *min* aggregation cannot be incrementally adjusted without re-examining its prior inputs. To directly support such *non-decomposable* aggregations, $g_i(v)$ must be represented by the set of values upon which it operates so that changes can be handled by re-examining the set. However, tracking aggregations by maintaining all edge dependencies is not desirable. Hence, we incorporate a re-evaluation strategy that pulls values from incoming edges on-the-fly and separates out tracking of $g_i(v)$ to single values (i.e., final results only). This modifies our formulation for *non-decomposable* aggregations to:

$$g_i^T(v) = \bigoplus_{\forall e=(u,v)\in E_{old}} (c_{i-1}(u)) \quad \biguplus_{\forall e=(u,v)\in E_{new}} (c_{i-1}^T(u))$$

$$E_{old} = E \setminus (E_d \cup \{(u,v) : (u,v) \in E \wedge c_{i-1}(u) \neq c_{i-1}^T(u)\})$$

$$E_{new} = E_a \cup \{(u,v) : (u,v) \in E^T \wedge c_{i-1}(u) \neq c_{i-1}^T(u)\}$$

4 GraphBolt Processing Engine

So far we discussed dependency-driven value refinement in a generalized context of processing streaming graphs without focusing on any particular system which makes our proposed techniques useful for several systems. We now discuss the important design details of GraphBolt system that incorporates our dependency-driven value refinement. We developed GraphBolt based on Ligra's processing architecture since its lightweight execution model enables high performance in supporting synchronous graph algorithms.

4.1 Streaming Graph & Dependency Layout

As discussed in §3.2, value dependencies are directly maintained as aggregation values, and are used along with the graph structure to perform refinement based on mutations. To enable fast manipulation of the graph structure, we maintain the structure separate from the aggregation values.

The core graph structure is maintained in compressed sparse row (CSR) and column (CSC) forms where edges are maintained as contiguous chunks indexed based on their

source and destination vertices. This allows GraphBolt to achieve efficient parallel mapping over vertices and edges in dense and sparse modes [39, 54]. To apply mutations, the structure is adjusted by making one sequential pass over vertex array and one parallel pass over edge array to ensure that indexes correctly reflect the mutation. The first pass over vertices computes offset adjustments in parallel while the second pass over edges shifts the edges and inserts/deletes edges in vertex-parallel manner. Mutations are supported in form of vertex and edge additions and deletions and can be applied either: a) one after the other as single edge/vertex mutation; or b) as a batch of multiple edge/vertex mutations. Value refinement begins instantly after the mutation batch gets applied. Mutations arriving during refinement are buffered to prioritize latency of the ongoing refinement step, and are applied immediately after refining finishes. Adjusting the structure across above two passes allows GraphBolt to respond quickly as graph structure changes: for example, adjusting the structure of UKDomain graph [7] (1B edges) with 10K mutations takes ~850 ms. Faster dynamic graph data-structures like STINGER [13] can be incorporated to improve the time taken to adjust the graph structure.

The aggregation values are maintained as arrays per-vertex to hold values across iterations. As computations progress and aggregation values get updated, the per-vertex aggregation arrays grow dynamically. To eliminate indirections during refinement, the aggregation values are maintained contiguously such that if $g_i(v)$ is to be saved because it reflects an updated value compared to $g_{i-1}(v)$, then $g_k(v)$ is also maintained $\forall k < i$ (i.e., holes reflecting no change are eliminated). These structures required are also pre-allocated for few iterations based on the available memory and optional user-defined parameters. With vertical pruning disabled, allocations are done per-iteration across all vertices.

4.2 Dependency-Driven Processing Model

We present how the aggregation values get iteratively refined via incremental computation using PageRank and Belief Propagation algorithms as our examples to contrast the technique for simple and complex aggregations. GraphBolt builds over the graph parallel interface to provide `edgeMap` and `vertexMap` functions, and also provides incremental execution models to simplify the refinement process.

Algorithm 2 shows the refinement process for Belief Propagation [20]. To focus our discussion on the incremental refinement process, details related to eliminating inbound contributions for Belief Propagation are left out from Algorithm 2. In BPMutable( ), lines 47-50 first compute direct impact of edge mutations (i.e., $\biguplus$ and $\bigsqcup$), and lines 53-60 compute the transitive impact (i.e., $\bigtriangleup$). The user functions repropagate( ) and retract( ) capture the incremental logic of $\biguplus$ and $\bigsqcup$ respectively. As we can see, the shape of these functions is similar since they are either contributing to or withdrawing from the saved aggregated values. It is interesting to note that the contribution made by each edge to the target aggregation is based on all of its states (lines 2-12). This means, the aggregation type is complex and cannot be directly undone. Hence, $\bigtriangleup$ gets further split across two functions (lines 55-56): retract( ) to withdraw old contribution followed by propagate( ) to contribute based on updated value for same set of edges. Algorithm 3 shows the refinement functions for PageRank whose aggregation type is simple. In this case, propagateDelta( ) directly captures the change in contribution (lines 8) which gets invoked using a single edgeMap( ) instead of two edgeMap( ) calls on lines 55-56 in Algorithm 2.

Note that to simplify exposition, Algorithm 2 and Algorithm 3 show that $\biguplus$ and $\bigsqcup$ get computed separately before computing $\bigtriangleup$. However, GraphBolt merges these computations together to make a single pass over the iteration space.

Non-Decomposable Aggregations.
While *min* and *max* are amenable to addition of new contributions (e.g., coming from edge additions) due to their monotonic nature, the aggregated value cannot be incrementally adjusted to remove an old contribution (e.g., to reflect edge deletions) since it doesn't implicitly hold all the original contributions. We classify such aggregations to be *non-decomposable* (discussed in §3.3). To support non-decomposable aggregations without increasing the amount of dependency information, GraphBolt's processing model can be enhanced such that it directly re-evaluates the aggregation with the entire updated input set (instead of the changed values only). The updated input set per vertex is reconstructed by pulling values from its incoming neighbors (directly available from CSC format). In §5.4 we will incrementally compute Single Source Shortest Paths algorithm using this re-evaluation strategy for *min* aggregation.

Selective Scheduling.
GraphBolt also supports selective scheduling which eliminates redundant computations by allowing vertex values to be recomputed only when its neighboring values change. Incremental refinement, if left oblivious of such selective scheduling, can violate semantics since it may cause spurious updates (e.g., retracting contribution that was never made). Our processing model ensures that change in contributions get correctly accounted based on how computations took place prior to mutations. GraphBolt also allows users to express the selective scheduling logic (e.g., comparing change with tolerance) which gets invoked on old values, based on which value retraction gets invoked.

Computation-Aware Hybrid Execution.
With horizontal pruning, aggregation values are available only until a certain iteration k. As computation progresses beyond iteration k, GraphBolt dynamically switches to incremental computation without value refinement. While the incremental computation directly pushes change in vertex values across edges, at iteration $k+1$ edges whose vertex

Algorithm 2 Dependency-Driven Refinement for Belief Propagation (Simplified)

```
1: S: set of states
2: function getContribution(e = (u, v), product)
3:    for s ∈ S do
4:       contribution[s] = 0
5:    end for
6:    for s ∈ S do
7:       for s′ ∈ S do
8:          contribution[s] += φ(u, s′) × ψ(u, v, s′, s) × product[s′]
9:       end for
10:   end for
11:   return contribution
12: end function

13: function repropagate(e = (u, v), i)
14:   contrib = getContribution(e, normOProd[u][i])
15:   for s ∈ S do
16:      atomicMultiply(&newProd[v][i + 1][s], contrib[s])
17:   end for
18: end function
19: function retract(e = (u, v), i)
20:   contrib = getContribution(e, normOProd[u][i])
21:   for s ∈ S do
22:      atomicDivide(&newProd[v][i + 1][s], contrib[s])
23:   end for
24: end function
25: function propagate(e = (u, v), i)
26:   contrib = getContribution(e, normNProd[u][i])
27:   for s ∈ S do
28:      atomicMultiply(&newProd[v][i + 1][s], contrib[s])
29:   end for
30: end function

31: function compute(v, i)
32:   ret = false
33:   for s ∈ S do
34:      normNProd[v][i + 1][s] = normal(newProd[v][i + 1][s])
35:      if normNProd[v][i+1][s] ≠ normOProd[v][i+1][s] then
36:         ret = true
37:      end if
38:   end for
39:   return (ret == true) ? v : ∅
40: end function
41: function computeBelief(v, i)
42:   for s ∈ S do
43:      belief[v][s] = k × φ(i, s) × normNProd[v][i][s]
44:   end for
45: end function
46: function BPMutable( )
47:   for i ∈ [0...k] do
48:      edgeMap(E_add, repropagate, i)
49:      edgeMap(E_delete, retract, i)
50:   end for
51:   V_updated = ∅
52:   V_change = getTargets(E_add ∪ E_delete)
53:   for i ∈ [0...k] do
54:      E_update = {(u, v) : u ∈ V_updated}
55:      edgeMap(E_update, retract, i)
56:      edgeMap(E_update, propagate, i)
57:      V_dest = getTargets(E_update)
58:      V_change = V_change ∪ V_dest
59:      V_updated = vertexMap(V_change, compute, i)
60:   end for
61:   vertexMap(V_change, computeBelief, k)
62: end function
```

Algorithm 3 Refinement functions for PageRank

```
1: function repropagate(e = (u, v), i)
2:    atomicAdd(&sum[v][i + 1], oldpr[u][i] / old_degree[u])
3: end function
4: function retract(e = (u, v), i)
5:    atomicSub(&sum[v][i + 1], oldpr[u][i] / old_degree[u])
6: end function
7: function propagateDelta(e = (u, v), i)
8:    atomicAdd(&sum[v][i + 1],
                newpr[u][i] / new_degree[u] − oldpr[u][i] / old_degree[u])
9: end function
```

values had changed in the original computation (prior to mutation) must also be processed, along with the set of edges (directly and transitively) impacted by edge mutations. In absence of vertical pruning, this is achieved by tracking the set of changed vertex values in a bit-vector at the end of iteration k in the original computation, and then updating the set with vertices impacted by mutations before starting the incremental processing.

4.3 Guaranteeing Synchronous Semantics

We directly reason about how $g_i^T(v)$ gets computed $\forall i, \forall v$. Let $c_0^T(v)$ denote the value of v before processing begins.

Theorem 4.1. *With dependency-driven value refinement, $\forall i > 0, \forall v \in V$, $g_i^T(v)$ is computed using $c_{i-1}^T(u)$ to satisfy dependencies based on E^T as defined in Eq. 1.*

Proof. We skip the proof due to limited space. We reason about the changes performed by $\biguplus$, $\bigsqcup$ and $\bigtriangleup$ incremental aggregators in terms of change in dependencies (subtracting dependencies from old values and adding dependencies from new values) to incrementally compute $g_i^T(v)$ from $g_i(v)$. □

5 Evaluation

We evaluate GraphBolt using six synchronous graph algorithms and compare its performance with Ligra [39], Differential Dataflow [25] and KickStarter [44].

5.1 Experimental Setup

We evaluate GraphBolt using six synchronous graph algorithms as listed in Table 4. PageRank (PR) [30] computes relative importance of web-pages while Belief Propagation (BP) [20] is an inference algorithm. Label Propagation (LP) [53]

Graphs	Edges	Vertices
Wiki (WK) [47]	378M	12M
UKDomain (UK) [7]	1.0B	39.5M
Twitter (TW) [21]	1.5B	41.7M
TwitterMPI (TT) [8]	2.0B	52.6M
Friendster (FT) [14]	2.5B	68.3M
Yahoo (YH) [49]	6.6B	1.4B

Table 2. Input graphs used in evaluation.

	System A	System B
Core Count	32 (1 × 32)	96 (2 × 48)
Core Speed	2GHz	2.5GHz
Memory Capacity	231GB	748GB
Memory Speed	9.75 GB/sec	7.94 GB/sec

Table 3. Systems used in evaluation.

Algorithm	Aggregation ($\bigoplus$)
PageRank (PR)	$\sum_{\forall e=(u,v)\in E} \frac{c(u)}{out\_degree(u)}$
Belief Propagation (BP)	$\forall s \in S : \prod_{\forall e=(u,v)\in E} (\sum_{\forall s'\in S} \phi(u,s') \times \psi(u,v,s',s) \times c(u,s'))$
Label Propagation (LP)	$\forall f \in F : \sum_{\forall e=(u,v)\in E} c(u,f) \times weight(u,v)$
Co-Training Expectation Maximization (CoEM)	$\sum_{\forall e=(u,v)\in E} \frac{c(u)\times weight(u,v)}{\sum_{\forall e=(w,v)\in E} weight(w,v)}$
Collaborative Filtering (CF)	$\langle \sum_{\forall e=(u,v)\in E} c_i(u).c_i(u)^{tr} , \sum_{\forall e=(u,v)\in E} c_i(u).weight(u,v) \rangle$
Triangle Counting (TC)	$\sum_{\forall e=(u,v)\in E} \lvert in\_neighbors(u) \cap out\_neighbors(v) \rvert$

Table 4. Graph algorithms used in evaluation and their aggregation functions.

is a learning algorithm while Co-Training Expectation Maximization (CoEM) [28] is a semi-supervised learning algorithm for named entity recognition. Collaborative Filtering (CF) [52] is a context-based approach to identify related items for recommendation systems. Triangle Counting (TC) computes frequencies of different triangles.

Table 2 lists the six real-world graphs used in our evaluation. Similar to [38, 44], we obtained an initial fixed point and streamed in a set of edge insertions and deletions for the rest of the computation. After 50% of the edges were loaded, the remaining edges were treated as edge additions that were streamed in. Edges to be deleted were selected from the loaded graph and deletion requests were mixed with addition requests in the update stream. In our experiments, we varied the rate of the update stream to thoroughly evaluate the effectiveness of incremental processing and scalability of GraphBolt. Unless otherwise stated, each algorithm (except TC which gets computed in a single iteration) was run for 10 iterations on all inputs except YH, and algorithms on YH were run for 5 iterations.

Table 3 describes the machines used in our evaluation. We used System A for all graphs except YH, and to further evaluate how GraphBolt scales, we used System B (`r5.24xlarge` on Amazon EC2) which has 3× the amount of memory and cores compared to that in System A. Both systems ran 64-bit Ubuntu 16.04 and programs were compiled using GCC 5.4, optimization `level -O3`.

To thoroughly evaluate GraphBolt, we compare the following three versions:

- **Ligra:** is the Ligra system [39] which restarts computation upon graph mutations.
- **GB-Reset:** is our GraphBolt system based on incremental computation during processing (i.e., propagates changes to enable selective scheduling), but restarts computation upon graph mutations. The processing model is similar to *PageRankDelta* in [39].
- **GraphBolt:** is our GraphBolt system based on dependency-driven incremental computation upon graph mutations as proposed in this paper.

To ensure a fair comparison among the above versions, experiments were run such that each algorithm version had the same number of pending edge mutations to be processed (similar to methodology in [44]). Unless otherwise stated, 100K edge mutations were applied before the processing of each version. While Theorem 4.1 guarantees correctness of results via synchronous processing semantics, we validated correctness for each run by comparing final results.

5.2 Performance

Table 5 shows the execution times for Ligra, GB-Reset and GraphBolt across 1K, 10K and 100K edge mutations. As we can see, both GB-Reset and GraphBolt outperform Ligra across all cases except TC where Ligra and GraphBolt-Reset are same (recall TC takes only single iteration to compute results). This is mainly due to selective scheduling that processes only those edges whose source vertex values change across iterations by propagating changes across aggregations. Furthermore, GraphBolt outperforms GB-Reset in all cases which indicates the effectiveness of our dependency-driven incremental computation to quickly react to changes in graph structure, even at a scale of 100K edge mutations. Figure 6 compares the amount of work performed by GraphBolt v/s GB-Reset in terms of number of edges processed. GraphBolt performs less than 50% edge computations compared to that in GB-Reset in most of the cases; while GraphBolt often performs 60-80% edge computations for PR (except UK), and for TT/TW on CoEM, the reduction in edge computations directly results in savings in Table 5.

It is interesting to observe that speedups are different across different algorithms; for example GraphBolt v/s GB-Reset for BP on TW is 10.48-14.39×, while for CF on TW is 6.17-8.79× even though Figure 6 shows that latter performs lesser edge computations compared to the former. This difference is because the remaining factors beyond edge computations (like `vertexMap` times, managing local copies, etc.) that had minor impact on GB-Reset's performance become significant enough in GraphBolt since edge work gets drastically reduced; nevertheless, this time is very low and

		WK			UK			TW			TT			FT		
		1K	**10K**	**100K**	**1K**	**10K**	**100K**	**1K**	**10K**	**100K**	**1K**	**10K**	**100K**	**1K**	**10K**	**100K**
	Ligra	2.81	2.84	2.81	4.52	4.34	4.36	19.45	19.58	19.26	45.50	45.38	45.76	47.65	47.36	47.57
	GB-Reset	1.65	1.66	1.67	4.76	4.61	4.63	9.87	9.85	9.81	13.80	13.89	13.79	20.22	20.29	20.33
PR	**GraphBolt**	1.12	1.22	1.36	0.26	0.37	0.55	7.25	7.43	7.60	8.61	9.20	9.74	11.77	14.44	15.72
	× Ligra	2.52×	2.32×	2.06×	17.40×	11.69×	7.87×	2.68×	2.64×	2.54×	5.29×	4.93×	4.70×	4.05×	3.28×	3.03×
	× GB-Reset	1.48×	1.36×	1.22×	18.35×	12.44×	8.36×	1.36×	1.33×	1.29×	1.60×	1.51×	1.42×	1.72×	1.40×	1.29×
	Ligra	49.89	49.91	49.84	98.66	98.24	98.28	256.15	256.25	256.86	348.26	321.20	321.03	521.56	521.49	522.12
	GB-Reset	12.36	12.39	12.36	33.27	33.36	33.14	60.88	60.85	61.08	78.83	78.78	78.76	116.91	116.84	116.72
BP	**GraphBolt**	0.76	1.05	1.45	2.11	2.17	2.23	4.23	5.04	5.83	3.41	4.90	9.19	3.62	3.73	4.09
	× Ligra	65.89×	47.61×	34.41×	46.76×	45.37×	44.02×	60.53×	50.82×	44.09×	102.14×	65.60×	34.93×	143.9×	139.7×	127.7×
	× GB-Reset	16.32×	11.81×	8.54×	15.77×	15.41×	14.84×	14.39×	12.07×	10.48×	23.12×	16.09×	8.57×	32.27×	31.29×	28.55×
	Ligra	14.20	14.15	14.14	27.65	27.04	27.75	81.93	81.61	81.84	109.79	109.87	110.08	170.06	169.25	169.01
	GB-Reset	3.60	3.59	3.63	7.76	7.54	7.59	20.20	20.13	20.17	26.99	26.91	26.91	40.80	40.48	40.71
CF	**GraphBolt**	0.47	0.51	0.82	1.49	1.50	1.54	2.30	2.68	3.27	3.15	4.92	7.08	2.57	2.62	2.76
	× Ligra	29.91×	27.60×	17.29×	18.50×	18.08×	18.07×	35.65×	30.47×	25.05×	34.89×	22.35×	15.54×	66.14×	64.61×	61.30×
	× GB-Reset	7.58×	7.00×	4.44×	5.19×	5.04×	4.94×	8.79×	7.52×	6.17×	8.58×	5.48×	3.80×	15.87×	15.45×	14.77×
	Ligra	12.69	12.74	12.66	24.01	24.09	24.08	77.79	78.07	77.84	103.62	103.69	103.75	159.96	160.33	160.35
	GB-Reset	3.46	3.46	3.44	6.33	6.35	6.40	20.39	20.44	20.43	27.19	27.25	27.33	39.96	39.94	39.97
CoEM	**GraphBolt**	0.61	0.73	0.98	0.27	0.34	0.46	4.59	5.37	5.84	7.19	8.36	9.41	0.69	3.41	6.26
	× Ligra	20.79×	17.37×	12.94×	90.36×	70.30×	51.91×	16.94×	14.55×	13.32×	14.40×	12.41×	11.03×	232.5×	47.06×	25.63×
	× GB-Reset	5.67×	4.71×	3.52×	23.82×	18.53×	13.79×	4.44×	3.81×	3.50×	3.78×	3.26×	2.91×	58.08×	11.72×	6.39×
	Ligra	22.40	22.39	22.32	38.15	38.10	37.93	135.62	135.75	135.86	167.08	166.77	167.20	287.51	287.76	287.77
	GB-Reset	7.63	7.61	7.55	16.49	16.47	16.45	43.12	43.20	43.21	54.26	54.40	54.11	91.81	91.82	91.92
LP	**GraphBolt**	0.70	0.96	1.48	1.78	1.81	1.85	13.59	16.25	17.45	15.74	20.05	24.05	3.72	8.81	15.64
	× Ligra	32.08×	23.32×	15.13×	21.40×	21.04×	20.52×	9.98×	8.35×	7.78×	10.62×	8.32×	6.95×	77.23×	32.67×	18.40×
	× GB-Reset	10.93×	7.92×	5.12×	9.25×	9.10×	8.90×	3.17×	2.66×	2.48×	3.45×	2.71×	2.25×	24.66×	10.42×	5.88×
	Ligra	10.02	9.00	8.94	6.14	6.15	5.96	273.56	273.84	274.82	335.18	334.90	335.48	59.31	59.34	59.44
	GB-Reset	10.02	9.00	8.94	6.14	6.15	5.96	273.56	273.84	274.82	335.18	334.90	335.48	59.31	59.34	59.44
TC	**GraphBolt**	0.03	0.20	2.12	0.05	0.06	0.15	0.15	0.22	0.41	0.58	5.34	14.58	0.08	0.08	0.21
	× Ligra	345.2×	43.96×	4.21×	124.9×	103.9×	40.21×	1815.7×	1268.2×	674×	578.2×	62.74×	23×	722.5×	710×	279×
	× GB-Reset	345.2×	43.96×	4.21×	124.9×	103.9×	40.21×	1815.7×	1268.2×	674×	578.2×	62.74×	23×	722.5×	710×	279×

Table 5. Execution times (in seconds) for Ligra, GB-Reset and GraphBolt across 1K, 10K and 100K edge mutations. The highlighted rows, i.e., × Ligra and × GB-Reset indicate speedups achieved by GraphBolt over Ligra and GB-Reset respectively.

hence, doesn't impact the benefits achieved from incremental processing. Alternately, GraphBolt on TC outperforms GB-Reset by two-three orders of magnitude mainly because the impact of edge mutations on TC is always local (e.g., edge addition/deletion only affect its end-points and their direct neighbors). Hence, set intersections involved in TC get incrementally adjusted to reflect changes in final result without propagating those changes across multiple iterations.

Finally, Table 6 presents the executions times for YH graph (on system B) and the corresponding amount of edge computations performed is shown in Table 7. It is interesting to note that GraphBolt performs significantly lower edge computations for YH compared to other graphs (shown in Figure 6); it is less than 0.5% for all cases except CoEM where it is less than 12% (i.e., 0.12 ratio). We also varied the parallelism by 3× from 32 cores to 96 cores in Table 6; as we can see, going to 96 cores reduces the execution time for GraphBolt (as expected), but also reduces the speedups mainly because GB-Reset has more work to be done compared to GraphBolt, and hence, GB-Reset leverages parallelism more compared to

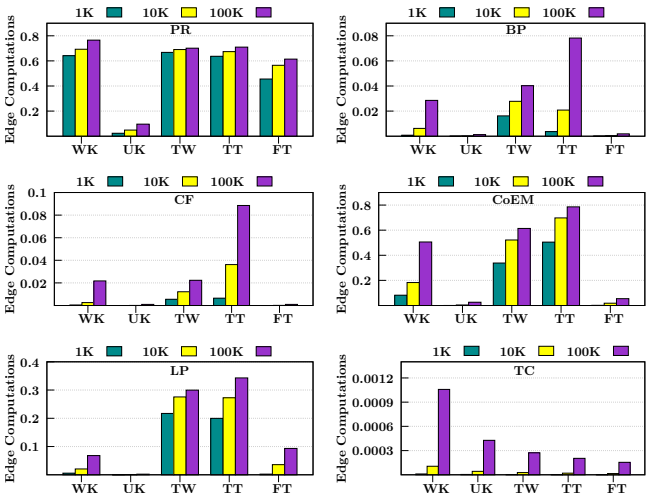


Figure 6. Ratio of edge computations performed by GraphBolt compared to that by GB-Reset.

		96 cores			32 cores		
		1K	10K	100K	1K	10K	100K
PR	Ligra	7.85	7.73	7.93	9.28	9.02	8.53
	GB-Reset	4.24	4.40	4.53	5.29	5.02	4.89
	GraphBolt	0.32	0.39	0.37	0.32	0.41	0.41
	× Ligra	24.19×	20.05×	21.52×	29.17×	21.74×	20.63×
	× GB-Reset	13.08×	11.40×	12.30×	16.63×	12.1×	11.82×
BP	Ligra	96.52	97.74	95.65	184.08	181.46	187.19
	GB-Reset	51.19	54.90	52.02	89.74	93.32	93.16
	GraphBolt	9.63	10.06	10.73	10.88	14.69	14.06
	× Ligra	10.02×	9.71×	8.91×	16.92×	12.35×	13.31×
	× GB-Reset	5.31×	5.46×	4.85×	8.25×	6.35×	6.62×
CF	Ligra	21.94	22.00	21.70	38.04	38.13	37.96
	GB-Reset	11.73	12.18	11.69	18.54	18.99	18.83
	GraphBolt	2.14	2.23	2.46	2.31	2.31	2.47
	× Ligra	10.25×	9.85×	8.80×	16.49×	16.52×	15.36×
	× GB-Reset	5.48×	5.46×	4.74×	8.04×	8.23×	7.62×
CoEM	Ligra	28.05	26.77	26.90	50.18	49.99	50.44
	GB-Reset	13.09	13.31	13.04	23.28	23.11	23.53
	GraphBolt	0.99	1.79	2.12	1.02	1.96	2.68
	× Ligra	28.25×	14.96×	12.68×	49.29×	25.51×	18.83×
	× GB-Reset	13.19×	7.44×	6.15×	22.87×	11.79×	8.79×
LP	Ligra	35.90	38.22	36.24	69.82	72.34	70.36
	GB-Reset	23.48	25.37	23.36	39.35	41.4	39.36
	GraphBolt	9.57	9.49	9.58	10.77	8.58	9.91
	× Ligra	3.75×	4.03×	3.78×	6.48×	8.43×	7.1×
	× GB-Reset	2.45×	2.67×	2.44×	3.65×	4.83×	3.97×
TC	Ligra	3.77	3.70	3.70	6.54	6.48	6.47
	GB-Reset	3.77	3.70	3.70	6.54	6.48	6.47
	GraphBolt	0.45	0.47	0.58	0.81	0.72	0.56
	× Ligra	8.37×	7.93×	6.42×	8.04×	8.96×	11.56×
	× GB-Reset	8.37×	7.93×	6.42×	8.04×	8.96×	11.56×

Table 6. Execution times (in seconds) on 96 and 32 cores for Ligra, GB-Reset and GraphBolt on YH graph across 1K, 10K and 100K edge mutations. The highlighted rows, i.e., × Ligra and × GB-Reset indicate speedups achieved by GraphBolt over Ligra and GB-Reset respectively.

	1K	10K	100K
PR	577.6K (0.013%)	7.3M (0.162%)	12.0M (0.267%)
BP	18.6K (0.000%)	0.4M (0.010%)	2.0M (0.051%)
CF	27.7K (0.002%)	1.8M (0.105%)	8.2M (0.481%)
CoEM	15.8M (1.801%)	57.8M (6.579%)	96.8M (11.009%)
LP	0.1M (0.002%)	2.0M (0.066%)	4.9M (0.164%)
TC	2.0K (0.000%)	20.0K (0.001%)	0.2M (0.012%)

Table 7. Edge computations performed by GraphBolt on YH. Numbers in parentheses indicate the percentage of edge computations performed by GraphBolt on YH compared to that by GB-Reset on YH.

GraphBolt. This is an indicator that GraphBolt's dependency-driven incremental computation is strong enough to reduce computations without relying on increasing compute capabilities to achieve high performance. Note that the impact of edge mutations varies based on the structure of the graph and also the nature of graph algorithm. For example, GraphBolt on PR achieves higher savings for UK compared to that for FT, while BP and CF achieve higher speedup for FT compared to that for UK.

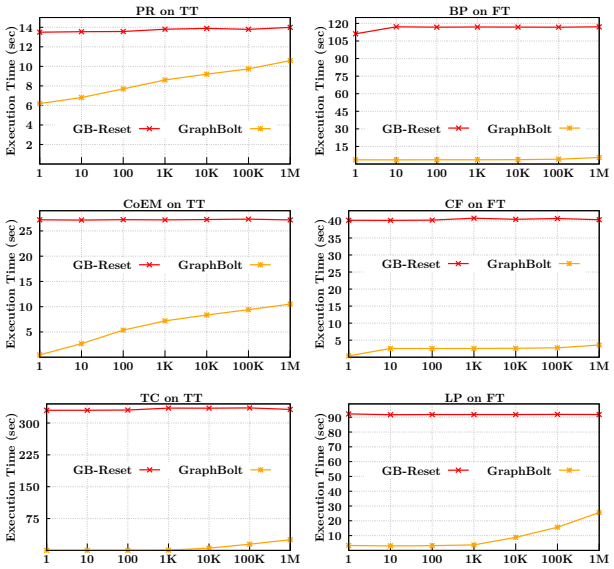


Figure 7. Execution times (in seconds) for TT and FT across different batch sizes: single edge to 1M mutations per batch.

5.3 Sensitivity Analysis

We study the sensitivity of our dependency-driven incremental processing to varying mutation batch size and workloads.

(A) Varying Mutation Batch Size.

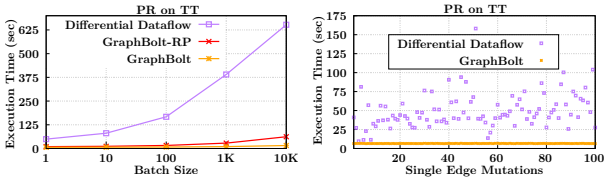
Figure 7 shows execution times for algorithms on FT and TT as we vary the mutation batch size between 1 to 1M edge mutations per batch. As expected, increase in edge mutations results in more work to be done, and hence, GraphBolt spends more time in doing increased work. It is interesting to note that even with 1M edge mutations, GraphBolt's incremental computation continues to be useful as it still manages to reduce work compared to the baseline GB-Reset. While GraphBolt outperforms GB-Reset for PR by 1.2-1.3× with 1M edge mutations, other benchmarks show higher improvements (between 1.9-42.5×). The increase in execution times is lowest for TC since the impact of edge mutations is relatively local in TC compared to that in other algorithms.

(B) Varying Mutation Workloads.

Since the impact of edge mutations also depend on where in the graph those mutations are being applied, we created two kinds of batches to reflect different mutation workloads: (1) a high workload (**Hi**) where mutations impact vertices with high outgoing degree (so that changes affect more vertices); and, (2) a low workload (**Lo**) where mutations impact vertices with low outgoing degree (to limit the impact of changes). Table 8 shows the execution times for GraphBolt on TT and FT with Hi and Lo workloads. As expected, executions times for Hi are higher than that for Lo across all cases. Nevertheless, GraphBolt still outperforms GB-Reset across all these cases due to its incremental computation.

	BP		CoEM		LP		TC		CF	
	Lo	Hi	Lo	Hi	Lo	Hi	Lo	Hi	Lo	Hi
TT	2.81	6.03	6.83	8.66	4.69	21.42	0.12	7.64	2.42	5.98
FT	3.62	4.76	1.26	9.26	3.15	27.44	0.08	0.12	2.59	3.08

Table 8. Execution times (in seconds) for GraphBolt with high workload (Hi) and low workload (Lo).



(a) Varying mutation batch size. **(b)** 100 single edge mutations.

Figure 8. Execution times (in seconds) for Differential Dataflow and GraphBolt.

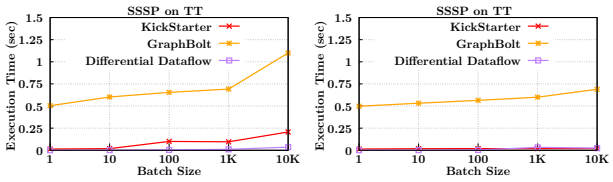
5.4 Comparison with Other Systems

We compare the performance of GraphBolt with that of Differential Dataflow [25] which is a general-purpose incremental dataflow system that operates on input streams, and KickStarter [44] which is a streaming graph processing solution for path-based monotonic graph algorithms.

(A) Differential Dataflow.

Graph computations can be expressed on Differential Dataflow in edge-parallel manner by joining edge tuples with rank values to be pushed across them, and then grouping them at destination vertices' rank tuples. In Figure 8, we evaluate PageRank on TT in Differential Dataflow and two versions in GraphBolt: first, expressed as PROPAGATEDELTA as shown in Algorithm 3 (called **GraphBolt**); and second, expressed as RETRACT and PROPAGATE as shown in Algorithm 2 (called **GraphBolt-RP**). GraphBolt-RP is typically performed for complex aggregations where change in values is captured explicitly using old and new values, i.e., edges propagate two values instead of one as done in GraphBolt (via PROPAGATEDELTA). We varied the mutation batch size from 1 to 10K and ran the experiments till convergence.

As shown in Figure 8a, Differential Dataflow retains high performance with 1 edge mutation, and scales as mutation batch size grows. Since GraphBolt is specialized towards processing streaming graphs as opposed to general streams, it delivers very high performance across all cases. GraphBolt-RP, as expected, gets impacted by propagating and computing over two values per change and hence is slower than GraphBolt. We observed very high variance for 1 edge mutation in Differential Dataflow; in Figure 8b, we show performance of 100 different 1 edge mutations for Differential Dataflow and GraphBolt. Differential Dataflow directly operates on changes and some of the changes impact the overall computation (i.e., intermediate results) more than others, which results in very high variance. GraphBolt processes single edge mutations efficiently and its variance is much lower even though work done per mutation is different; this is because of the constant overhead coming from the iteration-by-iteration



(a) Edge additions & deletions. **(b)** Edge additions only.

Figure 9. Execution times (in seconds) for KickStarter, Differential Dataflow and GraphBolt.

parallelization model and data-structure maintenance. We observe the impact of these overheads on throughput of processing single edge mutations such that results for each individual mutation needs to be reported: GraphBolt fully processes 100 consecutive single edge mutations (producing corresponding result for each individual mutation) in 409.34 seconds (compared to 7.89 seconds to process a single batch of 100 updates). Differential Dataflow, on the other hand, takes 1947.53 seconds for this case.

(B) KickStarter.

KickStarter enables streaming graph processing for monotonic graph algorithms by capturing light-weight dependencies across computed results and incrementally adjusting results upon edge mutations. Since it doesn't capture dependencies across intermediate vertex values, it doesn't guarantee synchronous processing semantics and is applicable to path-based monotonic algorithms. Figure 9 compares the performance of Single Source Shortest Path (SSSP) on TT in KickStarter and GraphBolt. As we can see in Figure 9a, even though GraphBolt efficiently computes shortest paths using *min* aggregation (*non-decomposable* type), KickStarter outperforms GraphBolt across all cases. This is due to two main reasons: a) GraphBolt maintains dependencies across all intermediate vertex values with the goal to guarantee synchronous processing semantics which is unnecessary for SSSP; and b) the *min* aggregation gets re-evaluated in GraphBolt with updated set of inputs which requires pulling values from incoming neighbors whenever edge deletions result in increase in path values. KickStarter, being a tailored solution for path-based monotonic algorithms, leverages asynchrony in SSSP in form of computation reordering which allows it to relax intermediate dependencies and propagate impact of changes quickly. Hence, KickStarter performs 14× lesser edge computations compared to GraphBolt upon edge mutation. To separate out the impact of re-evaluating *min*, we compare the performance with only edge additions in Figure 9b since edge additions in SSSP can be computed incrementally by *min* without re-evaluating it. As expected, both KickStarter and GraphBolt efficiently process edge additions and the difference in performance is purely due to the amount of dependencies maintained and updated in GraphBolt compared to that in KickStarter. Figure 9 also compares Differential Dataflow which is faster with edge deletions because it maintains an ordered map of path values and

counts for each vertex, which get quickly updated with value changes. Such a data-structure can be incorporated in GraphBolt to simulate faster incremental *min* (and *max*) at the cost of increased storage per vertex.

5.5 Memory Overhead

GraphBolt tracks aggregation values which requires space that is proportional to number of vertices. We measure the increase in memory used by GraphBolt compared to that by GB-Reset. Since this memory overhead gradually decreases with iterations, we measure the increase for the first iteration as a worst-case estimate; subsequent iterations will require lesser memory due to vertical pruning. As shown in Table 9, the increase is only up to 25% for all algorithms except CF and TC with TC requiring close to 2× memory. This overhead for TC comes from directly maintaining the structure of graph without mutation to incrementally adjust counts via addition/subtraction instead of resetting counts to 0 and recomputing for all edges within two-hop distance from mutated vertices. Nevertheless, TC runs for a single iteration only and hence, there is no further increase.

	WK	UK	TW	TT	FT	YH
PR	13.30%	15.90%	12.18%	11.69%	11.57%	12.49%
BP	16.87%	18.83%	15.95%	15.52%	15.42%	16.21%
CoEM	12.73%	15.10%	11.71%	11.25%	11.14%	11.99%
LP	20.30%	23.20%	18.97%	18.37%	18.23%	19.34%
CF	53.25%	59.06%	50.47%	49.18%	48.88%	51.25%
TC	92.13%	90.24%	92.90%	93.23%	93.31%	78.27%

Table 9. Increase in memory for GraphBolt w.r.t. GB-Reset.

6 Related Work

There exist several works on processing streaming graphs and generalized (structured and unstructured) data streams.

— *Streaming Graph Processing Frameworks.* Tornado [38] processes streaming graphs by forking off the execution to process user-queries while graph structure updates. KickStarter [44] uses dependence trees for incremental corrections in monotonic graph algorithms. GraphIn [37] incrementally processes dynamic graphs using fixed-sized batches. It provides a five-phase processing model that first identifies values that must be changed, and then updates them so that they can be merged back to previously computed results. It also maintains sparse dependence trees for path-based graph algorithms. Kineograph [10] enables graph mining using incremental computation along with push and pull models. [40] proposes the GIM-V incremental graph processing model based on matrix-vector operations. [32] constructs representative snapshots which are initially used for querying and upon success uses real snapshots. While these systems enable incremental computation, they lack dependency-driven incremental processing which guarantees synchronous processing semantics. STINGER [13] proposes dynamic graph data-structure and works like [12, 33] use the data-structure to develop algorithms for specific problems.

— *Batch Processing of Graph Snapshots.* These systems enable offline processing of well-defined temporal graphs snapshots. Chronos [18] uses incremental processing to compute values across multiple graph snapshots. [43] presents temporal computation and communication optimizations to process evolving graphs and its incremental processing leverages partially computed results across graph snapshots. GraphTau [19] maintains history of values over snapshots to rectify inaccuracies by reverting back the values. Finally, static graph processing systems [9, 15, 16, 27, 34, 35, 39, 45, 46, 48, 54] can also be used to process discrete graph snapshots.

— *Generalized Stream Processing.* Generalized stream processing systems [1, 2, 4, 5, 31, 36, 42, 50, 51] operate on unbounded structured and unstructured streams. Differential Dataflow [25] extends incremental computation in Timely Dataflow [26] by allowing the impact of change in streams to be reflected directly via diffs. Its strength lies in differential operators that enable capturing and computing over record changes. Since it operates over value changes only, it naturally incorporates selective scheduling which is useful for sparse computations. Its generality allows it to efficiently process streaming graphs as well (as shown in §5.4). GraphBolt, in comparison, is specialized towards processing streaming graphs, and hence extracts high performance.

— *Incremental Algorithms.* Incremental PageRank [24] reformulates computation where vertices propagate changes to extract optimizations for incremental updates and faster convergence. [3] analyzes Monte Carlo methods for incremental PageRank whereas [11] identifies vertices impacted by graph mutation and recomputes ranks for those vertices. For Triangle Counting, [23] presents a space-efficient algorithm to approximate counts while [41] develops cache-efficient parallel algorithm for counting on undirected streaming graphs.

Incremental View Maintenance (IVM) algorithms [6, 17, 29] reuse computed results to maintain a view consistent with input changes. They operate on different types of complex queries, however, they remain inefficient in terms of the amount of re-computation and storage required [25].

7 Conclusion

We presented GraphBolt to process streaming graphs while guaranteeing BSP semantics via dependency-driven incremental processing. GraphBolt's programming model allows decomposing complex aggregations to incorporate incremental value changes while also supporting direct incremental updates for simple aggregations. Our evaluation showed that GraphBolt efficiently processes streaming graphs with varying mutation rates, starting from just a single edge mutation all the way up to 1 million edge mutations at a time.

Acknowledgments

We would like to thank our shepherd Frank McSherry and the anonymous reviewers for their valuable and thorough feedback. This work is supported by the Natural Sciences and Engineering Research Council of Canada.

References

[1] Daniel J Abadi, Yanif Ahmad, Magdalena Balazinska, Ugur Cetintemel, Mitch Cherniack, Jeong-Hyon Hwang, Wolfgang Lindner, Anurag Maskey, Alex Rasin, Esther Ryvkina, et al. The Design of the Borealis Stream Processing Engine. In *CIDR*, volume 5, pages 277–289, 2005.

[2] Rajagopal Ananthanarayanan, Venkatesh Basker, Sumit Das, Ashish Gupta, Haifeng Jiang, Tianhao Qiu, Alexey Reznichenko, Deomid Ryabkov, Manpreet Singh, and Shivakumar Venkataraman. Photon: Fault-tolerant and Scalable Joining of Continuous Data Streams. In *SIGMOD*, pages 577–588, New York, NY, USA, 2013.

[3] Bahman Bahmani, Abdur Chowdhury, and Ashish Goel. Fast Incremental and Personalized PageRank. *VLDB*, 4(3):173–184, December 2010.

[4] Hari Balakrishnan, Magdalena Balazinska, Don Carney, Uğur Çetintemel, Mitch Cherniack, Christian Convey, Eddie Galvez, Jon Salz, Michael Stonebraker, Nesime Tatbul, et al. Retrospective on Aurora. *The VLDB Journal*, 13(4):370–383, 2004.

[5] Pramod Bhatotia, Umut A Acar, Flavio P Junqueira, and Rodrigo Rodrigues. Slider: Incremental Sliding Window Analytics. In *Middleware*, pages 61–72. ACM, 2014.

[6] Jose A. Blakeley, Per-Ake Larson, and Frank Wm Tompa. Efficiently Updating Materialized Views. In *SIGMOD*, pages 61–71, 1986.

[7] Paolo Boldi and Sebastiano Vigna. The WebGraph Framework I: Compression Techniques. In *WWW*, pages 595–601, 2004.

[8] Meeyoung Cha, Hamed Haddadi, Fabricio Benevenuto, and Krishna P. Gummadi. Measuring User Influence in Twitter: The Million Follower Fallacy. In *ICWSM*, pages 10–17, 2010.

[9] Rong Chen, Jiaxin Shi, Yanzhe Chen, and Haibo Chen. PowerLyra: Differentiated Graph Computation and Partitioning on Skewed Graphs. In *EuroSys*, pages 1:1–1:15, 2015.

[10] Raymond Cheng, Ji Hong, Aapo Kyrola, Youshan Miao, Xuetian Weng, Ming Wu, Fan Yang, Lidong Zhou, Feng Zhao, and Enhong Chen. Kineograph: Taking the Pulse of a Fast-changing and Connected World. In *EuroSys*, pages 85–98, 2012.

[11] Prasanna Desikan, Nishith Pathak, Jaideep Srivastava, and Vipin Kumar. Incremental Page Rank Computation on Evolving Graphs. In *WWW*, pages 1094–1095, 2005.

[12] David Ediger, Karl Jiang, Jason Riedy, and David A. Bader. Massive Streaming Data Analytics: A Case Study with Clustering Coefficients. In *IPDPSW*, pages 1–8, 2010.

[13] David Ediger, Rob Mccoll, Jason Riedy, and David A. Bader. STINGER: High Performance Data Structure for Streaming Graphs. In *HPEC*, pages 1–5, 2012.

[14] Friendster network dataset. http://konect.uni-koblenz.de/networks/friendster. KONECT, 2015.

[15] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. PowerGraph: Distributed Graph-parallel Computation on Natural Graphs. In *OSDI*, pages 17–30, 2012.

[16] Joseph E. Gonzalez, Reynold S. Xin, Ankur Dave, Daniel Crankshaw, Michael J. Franklin, and Ion Stoica. GraphX: Graph Processing in a Distributed Dataflow Framework. In *OSDI*, pages 599–613, 2014.

[17] Ashish Gupta, Inderpal Singh Mumick, and V. S. Subrahmanian. Maintaining Views Incrementally. In *SIGMOD*, pages 157–166, 1993.

[18] Wentao Han, Youshan Miao, Kaiwei Li, Ming Wu, Fan Yang, Lidong Zhou, Vijayan Prabhakaran, Wenguang Chen, and Enhong Chen. Chronos: A Graph Engine for Temporal Graph Analysis. In *EuroSys*, pages 1:1–1:14, New York, NY, USA, 2014. ACM.

[19] Anand Padmanabha Iyer, Li Erran Li, Tathagata Das, and Ion Stoica. Time-Evolving Graph Processing at Scale. In *GRADES*, page 5. ACM, 2016.

[20] U Kang, Duen Horng, and Christos Faloutsos. Inference of Beliefs on Billion-Scale Graphs. In *KDD-LDMTA*, 2010.

[21] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. What is Twitter, A Social Network or a News Media? In *WWW*, pages 591–600, 2010.

[22] Grzegorz Malewicz, Matthew H. Austern, Aart J. C. Bik, James C. Dehnert, Ilan Horn, Naty Leiser, Grzegorz Czajkowski, and Google Inc. Pregel: A System for Large-Scale Graph Processing. In *SIGMOD*, pages 135–146, 2010.

[23] Andrew McGregor, Sofya Vorotnikova, and Hoa T. Vu. Better Algorithms for Counting Triangles in Data Streams. In *PODS*, pages 401–411, 2016.

[24] Frank McSherry. A Uniform Approach to Accelerated PageRank Computation. In *WWW '05*, pages 575–582, 2005.

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

[26] Derek G. Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. Naiad: A Timely Dataflow System. In *SOSP*, pages 439–455. ACM, 2013.

[27] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. A Lightweight Infrastructure for Graph Analytics. In *SOSP*, pages 456–471, 2013.

[28] Kamal Nigam and Rayid Ghani. Analyzing the Effectiveness and Applicability of Co-training. In *CIKM*, pages 86–93. ACM, 2000.

[29] Vivek Nigam, Limin Jia, Boon Thau Loo, and Andre Scedrov. Maintaining Distributed Logic Programs Incrementally. In *PPDP*, pages 125–136, 2011.

[30] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The PageRank Citation Ranking: Bringing Order to the Web. Technical report, Stanford University, 1998.

[31] Frederick Reiss, Kurt Stockinger, Kesheng Wu, Arie Shoshani, and Joseph M. Hellerstein. Enabling Real-Time Querying of Live and Historical Stream Data. In *SSDBM*, pages 28–, 2007.

[32] Chenghui Ren, Eric Lo, Ben Kao, Xinjie Zhu, and Reynold Cheng. On Querying Historical Evolving Graph Sequences, 2011.

[33] Jason Riedy and Henning Meyerhenke. Scalable Algorithms for Analysis of Massive, Streaming Graphs. In *SIAM PP*, 2012.

[34] Amitabha Roy, Laurent Bindschaedler, Jasmina Malicevic, and Willy Zwaenepoel. Chaos: Scale-out Graph Processing from Secondary Storage. In *SOSP*, pages 410–424, 2015.

[35] Amitabha Roy, Ivo Mihailovic, and Willy Zwaenepoel. X-Stream: Edge-centric Graph Processing Using Streaming Partitions. In *SOSP*, pages 472–488, 2013.

[36] Pratanu Roy, Arijit Khan, and Gustavo Alonso. Augmented Sketch: Faster and More Accurate Stream Processing. In *SIGMOD*, pages 1449–1463, New York, NY, USA, 2016. ACM.

[37] Dipanjan Sengupta, Narayanan Sundaram, Xia Zhu, Theodore L Willke, Jeffrey Young, Matthew Wolf, and Karsten Schwan. GraphIn: An Online High Performance Incremental Graph Processing Framework. In *Euro-Par*, pages 319–333. Springer, 2016.

[38] Xiaogang Shi, Bin Cui, Yingxia Shao, and Yunhai Tong. Tornado: A System For Real-Time Iterative Analysis Over Evolving Data. In *SIGMOD*, pages 417–430, New York, NY, USA, 2016. ACM.

[39] Julian Shun and Guy E. Blelloch. Ligra: A Lightweight Graph Processing Framework for Shared Memory. In *PPoPP*, pages 135–146, 2013.

[40] Toyotaro Suzumura, Shunsuke Nishii, and Masaru Ganse. Towards Large-scale Graph Stream Processing Platform. In *WWW Companion*, pages 1321–1326, 2014.

[41] Kanat Tangwongsan, A. Pavan, and Srikanta Tirthapura. Parallel Triangle Counting in Massive Streaming Graphs. In *CIKM*, pages 781–786, 2013.

[42] Ankit Toshniwal, Siddarth Taneja, Amit Shukla, Karthik Ramasamy, Jignesh M Patel, Sanjeev Kulkarni, Jason Jackson, Krishna Gade, Maosong Fu, Jake Donham, et al. Storm @ Twitter. In *SIGMOD*, pages 147–156. ACM, 2014.

[43] Keval Vora, Rajiv Gupta, and Guoqing Xu. Synergistic Analysis of Evolving Graphs. *ACM TACO*, 13(4):32, 2016.

[44] Keval Vora, Rajiv Gupta, and Guoqing Xu. KickStarter: Fast and Accurate Computations on Streaming Graphs via Trimmed Approximations.

In *ASPLOS*, pages 237–251, 2017.

[45] Keval Vora, Sai Charan Koduru, and Rajiv Gupta. ASPIRE: Exploiting Asynchronous Parallelism in Iterative Algorithms Using a Relaxed Consistency Based DSM. In *OOPSLA*, pages 861–878, 2014.

[46] Keval Vora, Guoqing (Harry) Xu, and Rajiv Gupta. Load the Edges You Need: A Generic I/O Optimization for Disk-based Graph Processing. In *USENIX ATC*, pages 507–522, 2016.

[47] Wikipedia links, english network dataset. http://konect.uni-koblenz.de/networks/wikipedia_link_en. KONECT, 2017.

[48] Ming Wu, Fan Yang, Jilong Xue, Wencong Xiao, Youshan Miao, Lan Wei, Haoxiang Lin, Yafei Dai, and Lidong Zhou. GraM: Scaling Graph Computation to the Trillions. In *SoCC*, pages 408–421, New York, NY, USA, 2015. ACM.

[49] Yahoo! Webscope Program. http://webscope.sandbox.yahoo.com/.

[50] Matei Zaharia, Tathagata Das, Haoyuan Li, Scott Shenker, and Ion Stoica. Discretized Streams: An Efficient and Fault-Tolerant Model for Stream Processing on Large Clusters. In *HotCloud*, 2012.

[51] Erik Zeitler and Tore Risch. Massive Scale-out of Expensive Continuous Queries. In *VLDB*, pages 1181–1188, 2011.

[52] Yunhong Zhou, Dennis Wilkinson, Robert Schreiber, and Rong Pan. Large-Scale Parallel Collaborative Filtering for the Netflix Prize. In *AAIM*, pages 337–348. Springer, 2008.

[53] Xiaojin Zhu and Zoubin Ghahramani. Learning from Labeled and Unlabeled Data with Label Propagation. In *CMU Technical Report CALD-02-107*, 2002.

[54] Xiaowei Zhu, Wenguang Chen, Weimin Zheng, and Xiaosong Ma. Gemini: A Computation-Centric Distributed Graph Processing System. In *OSDI*, pages 301–316, 2016.