DZiG: Sparsity-Aware Incremental Processing of Streaming Graphs

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

Joanna Che
School of Computing Science
Simon Fraser University
British Columbia, Canada
jca241@cs.sfu.ca

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

Abstract

State-of-the-art streaming graph processing systems that provide Bulk Synchronous Parallel (BSP) guarantees remain oblivious to the computation sparsity present in iterative graph algorithms, which severely limits their performance. In this paper we propose DZiG, a high-performance streaming graph processing system that retains efficiency in presence of sparse computations while still guaranteeing BSP semantics. At the heart of DZiG is: (1) a sparsity-aware incremental processing technique that expresses computations in a recursive manner to be able to safely identify and prune updates (hence retaining sparsity); (2) a simple change-driven programming model that naturally exposes sparsity in iterative computations; and, (3) an adaptive processing model that automatically changes the incremental computation strategy to limit its overheads when computations become very sparse. DZiG outperforms state-of-the-art streaming graph processing systems, and pushes the boundary of dependency-driven processing for streaming graphs to over 10 million simultaneous mutations, which is orders of magnitude higher compared to the state-of-the-art systems.

1 Introduction

Recent advances in graph-based analytics coupled with the increasing interest in analyzing the constantly-evolving graph data has led to the development of several dynamic graph management and analytics solutions including GraphBolt [32], Tornado [50], and others [11, 48, 57].

Continuous query analysis over fast changing graphs is achieved by streaming graph processing where the results of the query are continuously recomputed as graph gets updated in order to make the results consistent with the latest version of the graph structure. Systems designed for streaming graph processing, like GraphBolt and Tornado, rely on *incremental processing* techniques where the results that were already computed prior to graph mutation are adjusted based on the graph structure updates instead of restarting the entire computation from scratch. Such incremental processing reduces the amount of computation to be (roughly) in the order of changes to the graph structure by reusing results that were computed prior to graph mutation.

Recent streaming graph processing systems perform *dependency-driven incremental processing* [32, 57] where intermediate values are tracked as computations progress, and later upon graph mutation, these values are incrementally adjusted to reflect changes resulting from graph mutations. Depending on the nature of the graph algorithm, such dependency-driven processing can provide Bulk Synchronous Parallel (BSP) [54] guarantees (i.e., generates final results equivalent to a BSP execution that starts from scratch) or allow asynchronous execution. For example, KickStarter [57] leverages algorithmic properties like monotonic convergence to capture lightweight dependencies in the form of trees, and performs a *trimming* process during which results are transformed based on the monotonic relationship among neighboring values to compute useful intermediate and final results. GraphBolt [32], on the other hand, provides synchronous processing semantics that models BSP execution where computation is performed in a series of global supersteps. It does so by capturing dependencies across intermediate vertex values, and performing a *refinement* process that propagates (direct and transitive) changes iteration-by-iteration to reflect mutations in graph structure. Since the dependency-driven refinement process does not rely on algorithmic properties like monotonicity, it is broadly applicable to general class of graph algorithms.

However, the dependency-driven incremental refinement strategy proposed in literature [32] remains oblivious to sparsity coming from the convergent nature of iterative graph algorithms. Specifically, vertex values start stabilizing as iterations progress; this represents *computation sparsity* that is often leveraged by techniques like selective scheduling in static graph processing systems [39, 51, 67]. The incremental refinement process, however, identifies effects of graph mutations as changes between the values computed prior to

EuroSys '21, April 26–28, 2021, Online, United Kingdom

ACM ISBN 978-1-4503-8334-9/21/04…$15.00
https://doi.org/10.1145/3447786.3456230

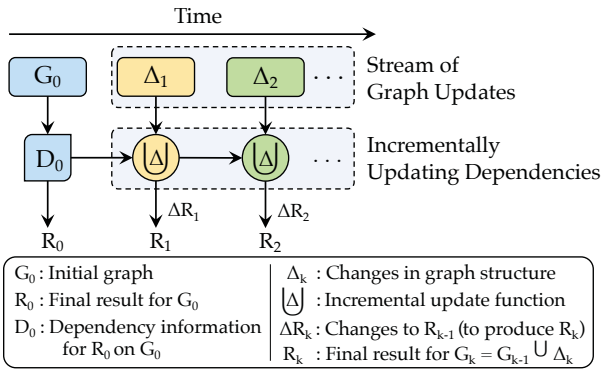


Figure 1. Dependency-driven incremental processing of streaming graphs. As graph updates get applied (either one at a time or in batches), the dependency information (i.e., intermediate values) gets incrementally adjusted to produce final results corresponding to those graph updates.

mutation and those that are incrementally computed during the refinement process. Therefore, even when vertex values stabilize across consecutive iterations, such a computation forces the refinement process to propagate the same change value for a given vertex across all iterations until the change itself diminishes naturally. This means, changes are propagated irrespective of whether the vertex value stabilizes (either during old execution prior to mutation, or during the refinement process), which limits the performance of incremental refinement. Furthermore, such change propagations cannot be directly pruned by simply verifying whether vertex values stop changing since those propagations are necessary to guarantee correctness; this is because of the fundamental way in which the refinement process identifies and operates on changes.

By not fully retaining the computation sparsity, the refinement strategy remains effective for only a few initial iterations where computing from scratch (i.e., incremental computation without dependency-driven refinement) would involve more work. For later iterations where computations become sparse, the refinement strategy ends up being relatively inefficient, and instead switching to the traditional incremental computation (without dependency-driven refinement) ends up being a better choice to retain end-to-end performance, as done by strategies like hybrid execution [32].

In this paper, we address the challenge of making the dependency-driven incremental refinement process an effective strategy across multiple iterations, especially when computation becomes sparse. We develop DZiG [1], a streaming graph processing system that retains efficiency in presence of sparse computations and guarantees Bulk Synchronous Parallel semantics. To do so, we first characterize the sparsity in convergence-based iterative graph computations, and then, we design a *sparsity-aware* incremental refinement technique by expressing computations *recursively* in terms of propagating 'updates to the changes' made in previous iterations. By doing so, the changes resulting from graph mutations can be directly captured as differences across consecutive iterations, which allows DZiG to safely identify and maintain sparsity throughout the refinement process.

Furthermore, we develop several important techniques that enable DZiG's sparsity-aware incremental processing to deliver high performance with easier programmability. First, we equip DZiG with a dependency tracking mechanism that tracks aggregation values corresponding to changes observed by outgoing neighbors, instead of just the aggregation values that get computed at any point; this guarantees accurate results in presence of custom propagation rules (e.g., propagate the value only if it is greater than a threshold value), which are common across several graph computations. Second, we design a simple change-driven programming model that naturally exposes sparsity in iterative computations to DZiG's underlying runtime, without exposing the dependency management/addressing issues to end users. Third, we develop an adaptive incremental computation strategy in DZiG that actively monitors the execution to control its overheads, and automatically switches the computation to propagate direct changes when computations become very sparse. And finally, we use dynamic adjacency lists in DZiG that not only enables fast graph mutation, but also retains high efficiency for parallel edge and vertex operations during the refinement process.

Our evaluation shows that our sparsity-aware refinement strategy pushes the boundary of effectiveness of dependency-driven incremental processing for streaming graphs to over 10 million simultaneous mutations (even when strategies like hybrid/adaptive switching are not used), which is orders of magnitude higher compared to the state-of-the art. Furthermore, our sparsity-aware refinement enables DZiG to outperform GraphBolt, the state-of-the-art system that provides BSP guarantees. Finally, DZiG also delivers high performance on traditional graph processing benchmarks like PageRank and shortest paths, hence significantly outperforming other streaming graph processing systems like Aspen [14], GraphOne [27], LLAMA [31] and Stinger [15].

2 Background and Motivation

We briefly review the streaming graph processing model and the relevant incremental processing techniques.

2.1 Streaming Graph Processing Model

A streaming graph is a graph whose structure keeps on changing via a continuous stream of graph updates (e.g., addition and deletion of vertices and edges). Streaming graph processing systems [32, 48, 50, 57] operate on streaming graphs to produce results consistent with the latest graph

[1] DZiG is incorporated in GraphBolt: https://github.com/pdclab/graphbolt

```
1  float[] prev_pr = {0, 0, ...};
2  float[] pr = {0.15, 0.15, ...};
3  float[] sum = {0, 0, ...};
4  Frontier curr = activateAll();
5  Frontier next = empty();
6  while(curr not empty) {
7    parallel_for u in curr {
8      float change = (pr[u] - prev_pr[u]) / numOutNbrs(u);
9      parallel_for v in outNeighbors(u) {
10       atomicAdd(&sum[v], change);
11       next.add(v);
12     }
13   }
14   swap(curr, next);
15   next.clear();
16   parallel_for v in curr {
17     float rank = 0.15 + 0.85 * sum[v];
18     if(fabs(rank - pr[v]) > ϵ) {
19       prev_pr[v] = pr[v];
20       pr[v] = rank;
21       next.add(v);
22     }
23   }
24   swap(curr, next);
25   next.clear();
26 }
```

Figure 2. Incremental PageRank example.

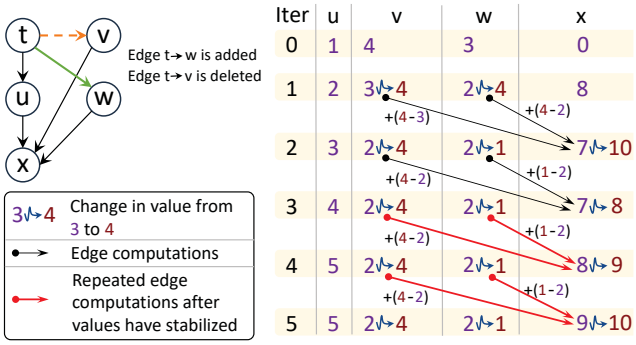


Figure 3. Incremental refinement in GraphBolt when edge (t, w) is added and (t, v) is removed. Note that edges (v, x) and (w, x) are active in iteration 4 (marked in red) even though both v and w stopped changing in iteration 3. This is because x's old value (prior to edge modifications) keeps changing till iteration 5 and its old value at iteration 4 does not yet have the new information from v and w. So, v and w have to push the difference to x at iteration 4. Similarly, these edges are active at iteration 5 even though v and w remain same in iteration 4.

structure. These systems rely on *incremental computation* where, upon graph mutation, they reuse the results that were computed before the graph structure mutated, so that the magnitude of computation remains (roughly) in the order of changes to the graph structure. Since different graph algorithms require different processing semantics to guarantee correctness, the incremental computation in these systems is tailored to guarantee synchronous processing (BSP [54]) semantics (e.g., in GraphBolt [32]), or enable asynchronous execution (e.g., in Tornado [50] and KickStarter [57]).

This work focuses on streaming graph processing that guarantees synchronous semantics, where the results produced by the incremental computation are the same as those produced by a BSP execution starting from scratch (i.e., BSP execution without reusing results).

With synchronous processing, computation is performed in a series of global supersteps such that values in a given superstep (or iteration) are computed based on values from the previous superstep. Figure 2 shows an example of PageRank for static graph that follows synchronous processing: during each iteration, `sum` is computed using `pr` from the previous iteration (lines 8-10), and the visibility of values is controlled by separating out the `pr` computation (lines 16-23) after the old `pr` values have been used (lines 7-13). Such clear separation of values being generated vs. values being used enables end users to easily develop complex graph-based analytics since the correctness/convergence properties can be clearly reasoned.

For streaming graphs, systems like GraphBolt [32] guarantee synchronous semantics using *dependency-driven incremental* computation, as described next.

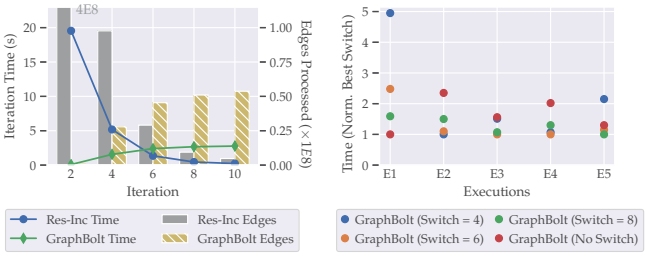
2.2 Dependency-Driven Incremental Computing

In this technique, *value dependencies* (i.e., information about how intermediate values affect each other to produce the final result) are tracked in memory as computation progresses, and later upon graph mutation, they are incrementally adjusted to produce correct final results (as shown in Figure 1). Since dependencies in graph computation are based on the structure of the input graph, the value dependencies are tracked in form of intermediate aggregation results on vertices instead of all the intermediate values propagating across edges, resulting in a much smaller amount of dependency information. Hence, in our PageRank example from Figure 2, the `sum` values are tracked instead of the individual `change` values that are pushed to outgoing edges.

Upon graph mutation, an *incremental refinement* process iteratively corrects the tracked aggregation values. In each iteration, changes in vertex values get propagated across edges, and they get merged into the aggregation values to correctly reflect the (direct and transitive) effects of graph mutation. The changes in values are captured based on results computed prior to graph mutation. This means, if a vertex's value at a given iteration gets updated due to incremental refinement, the difference between the updated value and its original value (prior to graph mutation) is propagated in the subsequent iteration.

2.3 Problem: Sparsity in Iterative Computations

Dependency-driven incremental refinement computes changes directly from the results computed prior to graph mutation. While such incremental computation is effective with dense computations, its benefits reduce as computations become sparse. We showcase this issue with the help



(a) (b)

Figure 4. Performance of dependency-driven incremental refinement [32] (GraphBolt) and incremental processing that restarts from scratch (Res-Inc) for Collaborative Filtering on Twitter [28] graph.

of an example. Figure 3 shows an example graph on the left where a new edge (t, w) is added and the existing edge (t, v) is deleted, and on the right it shows the resulting incremental refinement performed by [32]. Since u's value is never dependent on these two edges, its value does not change in any iteration (relative to its old value for the corresponding iteration prior to graph mutation). Hence, u does not propagate any difference to x throughout the refinement process. On the other hand, both v and w change in iteration 1 (v changes from 3 to 4, and w changes from 2 to 4), and hence, they propagate their differences to x in iteration 2, resulting in x's value to change from 7 to 10. In iteration 3, however, values for v and w remain exactly same as those in iteration 2; while this represents an opportunity to skip propagating the differences to x, the old value of x (8 in iter 4) is based on the old values of v (2 in iter 3) and w (2 in iter 3), and hence, the differences are still propagated to change the value of x from 8 to 9. This means, even though values stop changing at a given iteration, their effects are propagated (edge computations shown in red) across subsequent iterations until the outgoing neighbor's value stops changing.

We profiled the impact of this behavior on an incremental Collaborative Filtering computation [65] over a streaming Twitter [28] graph. Figure 4a shows the number of edges processed in each iteration by dependency-driven incremental computation [32] and by the traditional incremental computation that restarts from scratch [36, 51] (i.e., no dependency-driven refinement). As we can see, with dependency-driven refinement, the number of edges processed in each iteration increases as iterations progress; whereas it decreases (as expected) with the traditional incremental computation. In fact, after 5 iterations, traditional incremental computation processes fewer edges than using the refinement process.

The main reason why incremental refinement in [32] performs those unnecessary edge computations is because it computes changes between values before and after graph mutation, which prevents it from leveraging sparsity when values within an execution (either before or after graph mutation) start stabilizing. Solutions like hybrid execution (also used in [32]) eliminate the slowdowns in later iterations by performing dependency-driven incremental refinement only for few iterations and then dynamically switching to traditional incremental computation in the remaining iterations. However, the switching point needs to be manually tuned for each execution since different mutations result in different amounts of computation; Figure 4b shows that on different executions, different switch points result in drastically different performance, and there is no single switch point that gives the best performance across all the executions. Furthermore, relying on such a solution would fundamentally mean that dependency-driven incremental refinement is useful only when computations are dense.

In this work, we address the following question: *how do we perform dependency-driven incremental processing of streaming graphs that retains high performance across iterations where computations become sparse?*

3 Overview of DZiG

DZiG is a streaming graph processing system that guarantees synchronous (BSP) processing semantics and retains efficiency in presence of sparse computations. It builds over the design philosophy of GraphBolt [32]: as computation progresses, DZiG tracks value dependencies to capture the relationship between the intermediate and final results, and then upon graph mutation, it performs incremental refinement to efficiently compute final results.

DZiG uses dynamic adjacency lists that provide fast graph mutation while retaining high efficiency for parallel edge and vertex operations. The strength of DZiG lies in its novel *sparsity-aware* incremental refinement strategy, that aggressively reduces the amount of computation performed upon graph mutation. Furthermore, DZiG incorporates an *adaptive sparse incremental processing* strategy that automatically changes the manner in which incremental computation gets performed in order to manage the overheads of the sparsity-aware refinement strategy. DZiG uses a simple change-driven programming model, and captures the dependency information accurately so that differences can be correctly computed even in presence of custom constraints that block edge computations (e.g., floating-point based computations).

Section 4 formalizes the sparsity in iterative graph computations, and develops a refinement strategy that actively identifies and retains computation sparsity. Section 5 presents how sparsity-aware processing is achieved in DZiG.

4 DelZeros & Incremental Refinement

We first formalize DelZero updates, and then present the incremental refinement strategy based on DelZeros.

4.1 DelZero Semantics

Due to the convergent nature of iterative graph computations, vertex values stop changing as iterations progress.

Floating-point based graph algorithms like PageRank, rely on custom threshold checks (e.g., `1e-2` tolerance) to determine whether a newly computed vertex value is different enough compared to its previous value, so that minor value changes (ones that are smaller than the threshold) are not propagated further in the graph. We capture such kind of minor/no value changes as DelZeros.

Formally, let $\delta_i(v)$ be the value propagated by vertex v to its outgoing neighbors in iteration i+1. With incremental processing, $\delta_i(v)$ is based on change in v's value computed in iteration i. For example, in Figure 2, $\delta_i(u)$ is $\frac{pr[u]-prev\_pr[u]}{|out\_neighbors(u)|}$ which is computed on line 8 and propagated to v on line 10.

We define DelZero as:

$$\delta_i(v) = \varnothing \quad \text{iff} \quad |v_i - v_{i-1}| \leq \epsilon$$

where v_i and v_{i-1} are the values of v computed at iterations i and $i-1$ respectively.

For vertices that observe minor/no change in their values, the resulting $\delta(*)$ value is suppressed by $\varnothing$ to represent that they are not propagated to the neighboring vertices (similar to if-condition on line 18 in Figure 2). Hence, DelZeros enable sparse computations where vertex values are incrementally computed based on only the incoming values that change.

Our DelZero formulation does not enforce that vertex values remain same in the remainder of the execution. It simply captures those propagations across edges that do not happen in a given iteration due to no change in vertex values.

Generalization: Certain algorithms like Belief Propagation [24] compute $\delta(*)$ using both source and destination vertex values along with the corresponding edge weights. To capture such cases, δ should be parameterized by both the vertices, i.e., it becomes $\delta_i(v, w)$ where w is the destination vertex. To simplify exposition, we do not show w in $\delta_i(v)$ for the common case of algorithms.

4.2 DelZero-Aware Incremental Refinement

To exploit the sparsity coming from DelZeros, the incremental refinement process must be able to identify DelZeros so that their propagations (and the computations they cause on destination vertices) can be skipped. Since DelZeros are defined across consecutive iterations, we express our incremental computation *recursively* in terms of $\delta_i(*)$ values.

Formally, let $g_i(v)$ be the aggregated value of vertex v at iteration i (e.g., *sum* on line 17 in Figure 2). To reflect changes for the transformed graph G^T, our DelZero-aware refinement strategy computes $g_i^T(v)$ as:

$$g_i^T(v) = g_i(v) \bigoplus \Delta_i^T(v) \tag{1}$$

where $\Delta_i^T(v)$ is defined recursively as:

$$\Delta_i^T(v) = \Delta_{i-1}^T(v) \biguplus_{\substack{\forall e=(u,v)\in E_a \\ s.t.\ \delta_{i-1}(u)\neq\varnothing}} \delta_{i-1}(u) \biguplus_{\substack{\forall e=(u,v)\in E_d \\ s.t.\ \delta_{i-1}(u)\neq\varnothing}} \delta_{i-1}(u) \biguplus_{\substack{\forall e=(u,v)\in E^c \\ s.t.\ \delta_{i-1}(u)\neq\varnothing}} \delta_{i-1}(u) \biguplus_{\substack{\forall e=(u,v)\in E^c \\ s.t.\ \delta_{i-1}^T(u)\neq\varnothing}} \delta_{i-1}^T(u) \tag{2}$$

and $\bigoplus$ is the aggregation operator (e.g., `atomicAdd` in Figure 2), $\biguplus$ and $\biguplus$ are incremental aggregation operators that add and remove contributions respectively, and δ^T is value propagated based on the new values for G^T. Since $\Delta_{i-1}^T(v)$ recursively captures all the (direct and transitive) changes up to the previous iteration, the remaining four components in Eq. 2 deal with changes relative to the previous iteration only, i.e., in form of $\delta_{i-1}(*)$ values:

Additions. Newly added edges in E_a propagate $\delta_{i-1}(u)$ that would have been propagated if the edge was already present.

Deletions. Deleted edges in E_d retract $\delta_{i-1}(u)$ that would never have been propagated if the edge was never present.

Transitive Changes. E^c denotes the outgoing edges of vertices that are transitively affected (due to the refinement process) in the previous iteration. The incoming edges in E^c retract $\delta_{i-1}(u)$ that is based on old values, and propagate $\delta_{i-1}^T(u)$ that is based on new values.

Since all the above changes are across vertex values from consecutive iterations (either for the original graph G or for the transformed graph G^T), the preconditions on edges in Eq. 2 directly eliminate computations based on DelZeros, making our incremental refinement DelZero-aware.

Illustrative Example: Figure 5 illustrates how our DelZero-aware refinement changes x's values while maintaining sparsity as iterations progress for our example graph from Figure 3. For iteration 2, edge (w, x) retracts the change in w's value across iterations 1 and 0 in G (shown with negative symbol: $-(2-3)$) and propagates the change in w's value across iterations 1 and 0 in G^T (shown with positive symbol: $+(4-3)$). For iteration 3, edge (w, x) only propagates the change in w's value across iterations 2 and 1 in G^T ($+(1-4)$), and it does not retract any change in G since w's value remains same (w's value for G is 2 in iterations 1 and 2). Similarly in iteration 4, w does not propagate any changes as its value does not change across iterations 3 and 2 in both G and G^T. Thus, our DelZero-aware incremental refinement retains sparsity corresponding to both, G and G^T.

Correctness of DelZero-Aware Refinement: Our DelZero-aware refinement strategy guarantees synchronous processing semantics (same as GraphBolt [32]), i.e., it produces the same results as those computed by a BSP execution from scratch.

Theorem 4.1. *With DelZero-aware incremental refinement, $\forall i > 0$, $\forall v \in V$, and $\forall (u,v) \in E^T$, $g_i^T(v)$ is computed by incorporating value changes from iteration $i-1$ to $g_{i-1}^T(v)$.*

Proof. We prove this by induction over i.
— Base Case ($i = 1$): In Eq. 2, Δ_0^T is the identity value (e.g., 0 for sum aggregation) and E^c captures all the outgoing edges of vertices that are sources of edges in $E_a \cup E_d$. Hence, the contributions added and removed by the incremental aggregation are direct adjustments resulting from mutated edges.
— Induction Hypothesis ($i = k$):

$$g_k^T(v) = g_{k-1}^T(v) \biguplus_{\substack{\forall e=(u,v)\in E^T \\ s.t.\ \delta_{k-1}(u)\neq\varnothing}} \delta_{k-1}^T(u)$$

— Induction Step ($i = k+1$): Let $\ominus$ be the inverse operator of $\oplus$. From Eq. 1, we have:

$$\Delta_k^T(v) = g_k^T(v) \ominus g_k(v)$$

Substituting $\Delta_k^T(v)$ in the equation for $\Delta_{k+1}^T(v)$, we get:

$$g_{k+1}^T(v) = g_k^T(v) \oplus g_{k+1}(v) \ominus g_k(v) \biguplus_{\substack{\forall e=(u,v)\in E_a \\ s.t.\ \delta_k(u)\neq\varnothing}} \delta_k(u) \biguplus_{\substack{\forall e=(u,v)\in E_d \\ s.t.\ \delta_k(u)\neq\varnothing}} \delta_k(u) \biguplus_{\substack{\forall e=(u,v)\in E^c \\ s.t.\ \delta_k(u)\neq\varnothing}} \delta_k(u) \biguplus_{\substack{\forall e=(u,v)\in E^c \\ s.t.\ \delta_k^T(u)\neq\varnothing}} \delta_k^T(u)$$

Here, $g_{k+1}(v) \ominus g_k(v)$ represents the incremental computation that propagates all the changes from iteration k to iteration $k+1$. This means $g_{k+1}(v) \ominus g_k(v) = \biguplus_{\substack{\forall e=(u,v)\in E \\ s.t.\ \delta_k(u)\neq\varnothing}} \delta_k(u)$, and hence:

$$g_{k+1}^T(v) = g_k^T(v) \biguplus_{\substack{\forall e=(u,v)\in E \\ s.t.\ \delta_k(u)\neq\varnothing}} \delta_k(u) \biguplus_{\substack{\forall e=(u,v)\in E_a \\ s.t.\ \delta_k(u)\neq\varnothing}} \delta_k(u) \biguplus_{\substack{\forall e=(u,v)\in E_d \\ s.t.\ \delta_k(u)\neq\varnothing}} \delta_k(u) \biguplus_{\substack{\forall e=(u,v)\in E^c \\ s.t.\ \delta_k(u)\neq\varnothing}} \delta_k(u) \biguplus_{\substack{\forall e=(u,v)\in E^c \\ s.t.\ \delta_k^T(u)\neq\varnothing}} \delta_k^T(u)$$

The incremental aggregation operators above capture all the old contributions for G^T (first three terms: $\biguplus$, $\biguplus$ and $\biguplus$), and then adjust them for the new contributions based on the transitive effects of graph mutation (remaining two terms: $\biguplus$ and $\biguplus$). Hence:

$$g_{k+1}^T(v) = g_k^T(v) \biguplus_{\substack{\forall e=(u,v)\in E^T \\ s.t.\ \delta_k(u)\neq\varnothing}} \delta_k^T(u)$$

□

Sparsity-Awareness vs. GraphBolt: While we need to incorporate updated values relative to old values from G, our DelZero-aware strategy does not explicitly compare and compute differences between values for G and G^T. This is because we express the incremental computation relative to $\Delta_{i-1}^T(v)$, which recursively captures the difference in values for G and for G^T up to iteration $i-1$, and hence, we only need to incrementally incorporate the changes corresponding to the previous iteration, separately for G and for G^T. In comparison, the refinement process in GraphBolt [32] directly computes differences between values for G and G^T, which does not allow it to identify DelZeros across consecutive iterations in G and in G^T. Furthermore, it cannot skip propagating its changes by simply verifying whether vertex values stop changing in G or in G^T since the changes it operates on are necessary to guarantee correctness.

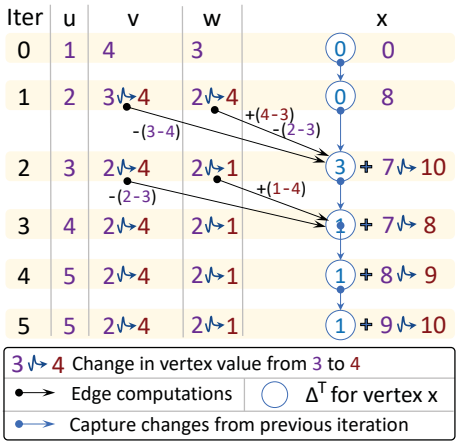


Figure 5. DelZero-aware incremental refinement of x for our example from Figure 3. Since v's value in iteration 2 remains same for G^T, it does not propagate the difference (+(4-4) ⟹ DelZero) in iteration 3. Similarly, no propagations in later iterations as v and w do not change.

5 DelZero-Aware Processing in DZiG

Using the above DelZero-aware refinement strategy, we develop DZiG system. In this section we focus on three key components of DZiG: first, the dependency tracking mechanism that enables accurate incremental computation; second, the incremental processing model that performs DelZero-aware refinement, along with its programming API to express graph computations; and third, the adaptive execution model that automatically switches the incremental processing strategy depending on computation sparsity.

5.1 Accurate Tracking of Value Dependencies

Since our DelZero-aware refinement strategy operates on aggregation values, DZiG tracks dependencies in the form of aggregation values for vertices. Depending on how DelZeros get defined in the user algorithm, the values visible to the target vertices of (inactive) edges may not be exactly equivalent to the aggregation values of the source vertices. For example, floating point computations often define DelZeros as values below a certain threshold (e.g., `1e-2` tolerance in PageRank); in such cases, minor changes in aggregation

Iter	2	4	5	6	7	8	9	10
Vertices	221K	632K	1.2M	1.6M	1.9M	2.0M	2.1M	2.1M

Table 1. PageRank on Wiki [60] graph with 1K edge mutations: number of vertices with over 1% relative error after each iteration.

values do not result in differences that get propagated across edges. If incremental refinement is done only based on aggregation values, the resulting changes that get propagated to outgoing neighbors may not be accurate. Furthermore, as the refinement process progresses, such inaccuracies increase since they get propagated throughout the graph. We measured this behavior for PageRank by calculating the relative error between vertex values resulting from refinement based on aggregation values alone, and those resulting from a computation without incremental refinement. As shown in Table 1 the number of vertices with relative error greater than 1% increases quickly as iterations progress, rising up to 32% of vertices within 10 iterations. While the magnitude of inaccuracies is small enough, the inaccuracies keep accumulating across multiple refinement passes (resulting from multiple mutation rounds), which further diverges the results as the graph structure keeps mutating.

To compute accurate final results that are equivalent to those generated by a BSP execution, our refinement process computes changes w.r.t. values whose effects were visible to outgoing neighbors. Hence, DZiG tracks the aggregation values corresponding to those that were propagated along with the aggregation values that capture the minor updates (similar to GraphBolt [32]). This allows the DelZero-aware refinement process to correctly compute the changes: the new values are computed using the aggregation values that contain minor updates (i.e., ones that may not be visible to neighbors) so that no updates are lost, while the changes are computed based on aggregation values corresponding to those that were propagated.

Tracking in Memory: Tracking aggregation values requires additional memory apart from the memory consumed by the dynamic graph data structure and the final vertex values. Traditional incremental processing techniques like [8] track all the individual values propagated across the edges which ends up demanding $O(|E| + |V|)$ memory. Since aggregation values in DZiG are tracked at vertex-level (similar to [32]), they require only $O(|V|)$ memory. As shown later in Section 6.6, maintaining aggregation values in DZiG consumes only 3-13% additional memory per iteration.

5.2 DelZero-Aware Incremental Refinement

The DelZero-aware incremental refinement iteratively propagates differences, as modelled in Eq. 2, using edge-parallel and vertex-parallel operations. In each iteration, first the direct changes resulting from edge additions and deletions are propagated, and then transitive changes are propagated, based on which final vertex value gets recomputed.

```
27 VValueType[] oldVVal = {...};    // Vertex values for oldG
28 VValueType[] newVVal = {...};    // Vertex values for newG
29 AggregationType[][] agg = {...};     // Aggregation values
30 DeltaType[] delta = {...};    // Changes due to refinement

32 void refine(EdgeList eAdditions, EdgeList eDeletions,
              Graph oldG, Graph newG) {
33   EdgeList eMutations = union(eAdditions, eDeletions);
34   Frontier vChanged = getTargets(eMutations);
35   Frontier vUpdated = getSources(eMutations);
36   for i in [1 .. k] {
37     // Direct changes
38     refineEdges(eAdditions, addChange, oldVVal, oldG, i);
39     refineEdges(eDeletions, removeChange, oldVVal,oldG,i);
40     // Transitive changes
41     refineOutEdges(vUpdated, addChange, newVVal, newG, i);
42     refineOutEdges(vUpdated, removeChange,oldVVal,oldG,i);
43     vDest = getTargets(E_update);
44     vChanged = union(vChanged, vDest);
45     vUpdated = refineVertices(vChanged, newG, i);
46   }
47 }

49 void refineEdges(EdgeList edges, AggregationOp aOp,
                  VValueType[] vVal, Graph G, int i) {
50   parallel_for e = (u, v) in edges {
51     if notDelZero(vVal[u][i], vVal[u][i-1]) {
52       DeltaType vChange = vertexChange(u, vVal[u][i],
                                          vVal[u][i-1], G);
53       DeltaType eChange = edgeChange(edge(u, v), vChange,
                             vVal[u][i], vVal[u][i-1], G);
54       aOp(&delta[v], eChange);
55     }
56   }
57 }

59 void refineOutEdges(Frontier activeV, AggregationOp aOp,
                     VValueType[] vVal, Graph G, int i) {
60   parallel_for u in activeV {
61     if notDelZero(vVal[u][i], vVal[u][i-1]) {
62       DeltaType vChange = vertexChange(u, vVal[u][i],
                                          vVal[u][i-1], G);
63       parallel_for v in outNgh(u) {
64         DeltaType eChange = edgeChange(edge(u, v),
                vChange, vVal[u][i], vVal[u][i-1], G);
65         aOp(&delta[v], eChange);
66       }
67     }
68   }
69 }

71 Frontier refineVertices(Frontier activeV, Graph G, int i){
72   Frontier vUpdated;
73   parallel_for v in activeV {
74     addChange(&agg[v][i+1], delta[v]);
75     new_value = computeVertex(v, agg, newVVal[v][i], G);
76     if notDelZero(new_value, newVVal[v][i]) {
77       newVVal[v][i+1] = new_value;
78       vUpdated.add(v);
79     } else { newVVal[v][i+1] = newVVal[v][i]; }
80   }
81   return vUpdated;
82 }
```

Figure 6. DelZero-aware incremental refinement in DZiG. The functions marked in blue are DZiG internal functions, and those in purple are callbacks to user functions. Figure 7 shows PageRank program written using the user functions.

Figure 6 shows the `refine()` function that performs DelZero-aware incremental refinement. The `refineEdges()` and `refineVertices()` functions invoke

```
83 bool notDelZero(VValueType oldVal, VValueType newVal) {
84   return fabs(newVal - oldVal) > epsilon;
85 }
86 DeltaType vertexChange(VId v, VValueType oldVal,
                          VValueType newVal, Graph G) {
87   return (newVal - oldVal) / G.out_degree[v];
88 }
89 DeltaType edgeChange(Edge e, DeltaType change,
       VValueType oldVal, VValueType newVal, Graph G) {
90   return change * G.edge_weight[e];
91 }
92 void addChange(AggregationType* aggr, DeltaType delta) {
93   atomicAdd(aggr, delta);
94 }
95 void removeChange(AggregationType* aggr, DeltaType delta){
96   atomicSubtract(aggr, delta);
97 }
98 VValueType computeVertex(VId v, AggregationType aggr,
                           VValueType oldVal, Graph G) {
99   return 0.15 + (0.85 * aggr);
100 }
```

Figure 7. Weighted PageRank program written on DZiG.

```
101 refineOutEdges(vUpdated, oldVVal, newVVal, oldG, newG,
                  addRemoveChange, i);
```

Figure 8. Merging Transitive Differences. Line 101 replaces lines 41-42 in Figure 6 to extract efficiency.

```
102 void addRemoveChange(AggregationType aggr,
       DeltaType oldDelta, DeltaType newDelta) {
103   atomicAdd(&aggr, newDelta - oldDelta);
104 }
```

Figure 9. Merged update for Weighted PageRank.

the user functions in parallel on edges and vertices respectively, and `refineOutEdges()` is similar to `refineEdges()` except that it operates on edges of active vertices instead of an edge list.

The refinement process iteratively call user-defined functions to identify changes, incrementally adjust the aggregation values, compute the vertex value, and more importantly identify values that are not DelZeros (`notDelZero()`). Figure 7 shows how weighted PageRank is implemented using the user-defined functions. To process the change for a given edge, the edge computation is split into two components: the first component computes the difference resulting from source vertex's value (lines 52 and 62), and the second component updates this difference based on edge weights or target vertex value (lines 53 and 64). For example, in PageRank (Figure 7), the vertex component in `vertexChange()` computes the difference in ranks that must be propagated to the outgoing edges, and then the edge component in `edgeChange()` multiplies the edge weight with the computed difference. Hence, the first component does not get recomputed multiple times for the outgoing edges (i.e., eliminates redundancy). Finally, the difference gets incrementally aggregated using `addChange()` and `removeChange()` calls.

The differences get computed only for updates that are not DelZeros. This is ensured by `notDelZero()` checks on lines 51 and 61, which eliminates DelZeros in both, the original graph (prior to mutation) and the updated graph. Also, after the vertex value gets computed using the differences, it is verified by `notDelZero()` (line 76), so that the vertex gets scheduled for the next iteration only if the new vertex value is significantly different.

Merging Transitive Differences: The transitive changes are performed by removing the old change and adding the new change (lines 41-42). Since these two steps are performed on the same set of edges (i.e., outgoing edges of vertices in `vUpdated`), they can be merged into a single step. By doing so, the same set of edges are not iterated twice, which improves performance.

Figure 8 shows the merged call to `refineEdges()` (replaces lines 41-42) that calls `addRemoveChange()` which is a combination of the two steps. While the default implementation of `addRemoveChange()` calls `addChange()` followed by `removeChange()`, it can be optimized to efficiently process the change. For example, Figure 9 shows how `addRemoveChange()` performs only a single atomic write instead of two separate atomic writes. While such merging brought little benefits (< 5% improvement) for most of our programs, we observed a big improvement (up to 40%) for PageRank due to the relatively simple operations involved in its edge and vertex functions.

5.3 Adaptive Sparse Incremental Processing

DelZero-Aware incremental refinement processes only those edges where changes need to be propagated. As iterations progress and sparsity increases, we observe that the number of edges processed by DelZero-Aware incremental refinement reduces to a similar magnitude as the number of edges processed without dependency-driven refinement (i.e., incremental processing that starts from scratch, similar to Figure 2). Figure 10 shows the performance of DelZero-Aware incremental refinement and incremental processing without dependency-driven refinement for Collaborative Filtering [65] on Twitter graph. As we can see, the difference between edge computations with and without incremental refinement reduces as iterations progress; in fact, after iteration 8, they both process nearly the same amount of edges. Since each edge update in the refinement process involves two sub-operations (removing old differences and adding new differences), propagating direct differences can be faster during those iterations (also shown in Figure 10). Even though the two sub-operations get fused using `addRemoveChange()`, it still requires computing two differences: `oldDelta` and `newDelta` (see Figure 9).

To further accelerate our incremental processing across those sparse iterations, we develop an adaptive incremental processing strategy that dynamically switches to propagate changes directly during those sparse iterations. Our strategy initiates the switch automatically by monitoring the time taken for each iteration, and estimating the time that the

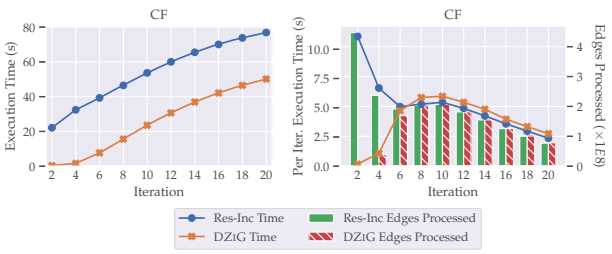


Figure 10. DelZero-aware incremental refinement (DZiG) vs. incremental computation without dependency-driven refinement that starts from scratch (Res-Inc) for Collaborative Filtering [65] on Twitter [28] graph. Left: cumulative execution times (in seconds) as iterations progress; Right: execution times per iteration (in seconds) and number of edge computations.

iteration would take without dependency-driven refinement. Since the processing time is dominated by the time taken to perform edge computations, the number of active edges (i.e., edges that need to be processed to propagate changes) in each iteration is linearly correlated with the time taken to process that iteration. Hence, we use linear regression over the number of active edges in each iteration along with least squares error for correlation to estimate the time required for a given iteration without dependency-driven refinement. Vertex computations take a very small amount of time (usually between 1-5% of iteration time), and so we use the average time for vertex operations as a constant weight in our model.

Our linear regression based model is lightweight, and hence, during execution we only need to track two variables: the number of active edges and the time taken for vertex operations. The number of active edges is computed using a parallel reduction over degrees of active vertices (i.e., edges are not traversed here) which does not consume much time (between 0.1-2% of the entire iteration time). Moreover, our model estimates the required time with high accuracy: it achieves the R^2 value [2] of ~0.99 with absolute error of only ~0.2 seconds (relative error between 0.1-10%). This is because the number of active edges dominates the execution time, which our linear regression based model captures directly.

With such a highly accurate estimation, our adaptive incremental processing strategy identifies the sparse iterations (estimated time lower than iteration time), and automatically switches to incremental processing without dependency-driven refinement. Once the strategy decides to switch, it needs to activate the set of edges corresponding to incremental computations that will propagate only a single change. It does so efficiently via a single `vertexMap` (i.e., parallel operation over vertices) that prunes out DelZeros and activates remaining vertices to be processed.

[2] R^2 is coefficient of determination which measures how well the observed outcomes are estimated. R^2 value of 1 indicates that the estimations perfectly fit the data, while an R^2 value of 0 indicates a poor fit.

6 Evaluation

We evaluated DZiG to study the performance of our DelZero-aware incremental refinement, and answer the following questions.

1. Is our DelZero-aware refinement strategy effective compared to the incremental refinement strategy developed in state-of-the-art streaming graph processing systems?
2. Does our DelZero-aware refinement strategy push the boundary of effectiveness of dependency-driven incremental processing compared to the existing solutions?
3. How does our DelZero-aware incremental refinement perform across different degrees of computation sparsity?
4. Does our adaptive sparse incremental strategy improve the performance of DZiG?
5. How does the performance of DZiG compare with other streaming graph processing systems that do not perform incremental refinement?

We first summarize the implementation details of DZiG, and then evaluate it thoroughly.

6.1 Implementation Details

DZiG is built on a similar underlying runtime as GraphBolt [32] (C++ using CilkPlus [12] for efficient parallelism and mimalloc [29] for high performance memory management) to retain the efficient parallelization model and various optimizations (e.g., vertex frontiers and atomic operations).

Dynamic Graph Data Structure: To maintain high locality for parallel vertex and edge operations during refinement while also ensuring fast insertions and deletions, DZiG uses dynamic adjacency lists to represent the graph. Each vertex maintains a contiguous list of incoming and outgoing edges, along with counters to hold its in-degree and out-degree information. These lists maintain empty slots at the end of the list depending on how graph mutation takes place, which relieves the pressure on memory management during high mutation rates. The adjacency lists get updated in parallel to handle (small and large) batches of vertex/edge updates. A batch of edge updates is first analyzed to verify whether the empty slots are sufficient to incorporate the new edges, based on which edge lists are expanded as required. Edge deletions are applied by swapping the deleted edge with the last edge in the corresponding edge list and decrementing the degree counter (resulting in empty slots at the end). Edge additions are performed by inserting the source/target vertex id in the first available empty slot.

Our dynamic adjacency lists allow efficiently handling both, single edge/vertex updates as well as large batches of simultaneous vertex/edge updates. For example, it takes only ~300 ms to perform 100K edge additions/deletions; detailed results in Figure 19 show that the graph mutation times are competitive.

The dependency information (aggregation values) is maintained separately from the graph data structure in the form of arrays. The data type of aggregation values gets supplied via a static template argument, which allows the framework to seamlessly track dependencies without exposing their memory management/addressing issues to the users.

6.2 Experimental Setup

To enable direct comparisons, we evaluate DZiG using the synchronous graph algorithms used in [32]: PageRank (PR) [43], Belief Propagation (BP) [24], Co-Training Expectation Maximization (CoEM) [40], Collaborative Filtering (CF) [65], and Label Propagation (LP) [66]. We did not use triangle counting since its incremental processing can be directly achieved in a single iteration. We used five real-world graphs for our evaluation, as shown in Table 2. Similar to [32, 50], we obtained an initial fixed point when 50% of edges were loaded, and streamed in the remaining edges to model edge insertions, while sampled edges from the loaded graph for edge deletions. To eliminate the effects of locality, we shuffled the edges while forming our edge streams.

To evaluate the effects of high mutation rates, we post batches of multiple edge insertions and deletions at the same time so that they get incorporated simultaneously. Similar to [32], we run all algorithms for 10 iterations on all inputs except YH, and for 5 iterations on YH, unless otherwise stated. The benefits of DelZero-aware incremental processing become clearly visible in those iterations, and with more iterations, computations become even more sparse which is anyways favorable for DelZero-aware processing.

We use two systems for our evaluation. For experiments on all inputs except YH, we use the machine with 32 cores (single socket) running at 2GHz and 231GB RAM. For the large YH graph, we use `r5.24xlarge` on Amazon EC2 which has 96 cores (dual socket, 48 cores per socket) running at 2.5GHz and 748GB RAM. The systems ran 64-bit Ubuntu 16.04 with compiler GCC 5.4 (compiled with `-O3` optimization).

Frameworks: We compare the performance of DZiG with state-of-the-art dynamic graph processing systems like GraphBolt [32], Aspen [14], GraphOne [27], LLAMA [31] and Stinger [15]. Since GraphBolt is the only dynamic graph processing system that performs incremental processing for our benchmarks while guaranteeing the same synchronous processing semantics, we thoroughly evaluate and compare against GraphBolt. Later, we briefly compare the performance with other systems (that do not perform incremental processing) on traditional graph processing benchmarks (PageRank and Shortest Paths).

While systems like Differential Dataflow [34] enable incremental processing for general-purpose input streams, their generality comes at a performance cost. Experiments in [32] show that an incremental PageRank computation on streaming graphs is an order of magnitude faster in GraphBolt than in Differential Dataflow. Hence, we primarily focus on the performance comparisons with GraphBolt.

Graph	Edges	Vertices
UKDomain (UK) [7]	1.0B	39.5M
Twitter (TW) [28]	1.5B	41.7M
TwitterMPI (TT) [10]	2.0B	52.6M
Friendster (FT) [16]	2.5B	68.3M
Yahoo (YH) [62]	6.6B	1.4B

Table 2. Real world graphs used in evaluation.

Throughout the evaluation, we use the following notations for different baselines:

- **GraphBolt-HP:** this is GraphBolt [32] with the best hand-picked switching (customized for each run) for its hybrid execution. This baseline captures the best performance that can be achieved by GraphBolt for every streaming input, and it is not a practical solution.
- **GraphBolt-K:** this is GraphBolt [32] with static switching at K^{th} iteration for its hybrid execution ($K \in \{3, 5, 7\}$).
- **Res-Inc:** this restarts execution (i.e., does not reuse results) upon graph mutation and performs incremental computation (i.e., propagates changes to enable selective scheduling, similar to *PageRankDelta* in [51]).

We follow the experimental methodology as used in GraphBolt [32] where the pending edge mutations to be processed by each technique is exactly same. Unless otherwise stated, adaptive processing is turned off in order to thoroughly evaluate our DelZero-aware incremental processing.

6.3 Performance

Figure 11 shows the execution times for DZiG, GraphBolt and Res-Inc across 1K and 1M edge mutations for various input graphs. As we can see, DZiG consistently achieves the best performance. Compared to GraphBolt-HP, DZiG performs better in most cases and nearly the same in the remaining few cases, and as expected, GraphBolt-HP is better than the statically picked GraphBolt-3/5/7, while Res-Inc is the slowest compared to DZiG and GraphBolt-HP. This is mainly because DZiG's DelZero-aware model processes the least amount of edges, as shown in Figure 12. GraphBolt-HP processes fewer edges compared to its statically picked variants, however, since it cannot directly identify DelZeros, it ends up processing more edges than DZiG. Also, Res-Inc does not reuse any prior results, and hence it ends up processing much more edges.

Without awareness of DelZeros, GraphBolt is useful only for a few iterations, which is visible with increasing execution time of GraphBolt-K as K increases. On the other hand, DZiG retains high performance throughout the execution.

As expected, processing 1K edge mutations is faster in DZiG than processing 1M edge mutations since the latter

	PR-1K				BP-1K				CF-1K				CoEM-1K				LP-1K			
	UK	TW	TT	FT	UK	TW	TT	FT	UK	TW	TT	FT	UK	TW	TT	FT	UK	TW	TT	FT
DZiG	0.578	9.59	11.2	21	1.4	4	4.77	1.23	4.71	10.5	11.1	4.25	0.73	5.65	7.64	5.43	0.609	6.62	8.05	6.03
GraphBolt-HP	1.5	10.1	11.7	21.4	2.7	5.36	6.37	5	7.83	11.9	12.9	8.39	0.983	5.53	7.52	5.63	3.18	9.62	12.3	9.25
GraphBolt-3	3.32	10.1	12.1	23.1	2.7	5.36	6.37	5	9.16	12.7	13.9	22.6	2.05	6.65	9.06	13.6	9.76	14.2	19	28.9
GraphBolt-5	1.91	10.4	11.8	21.5	12.8	91.9	112	87.7	7.83	11.9	12.9	11.3	0.983	5.53	7.52	5.63	5.36	11.7	13.8	9.25
GraphBolt-7	1.5	10.8	12.6	21.4	38.9	195	241	278	8.11	14.8	16.3	8.39	1.21	10.4	14	10	3.18	22	26.1	16.4
Res-Inc	4.38	12	15.8	29.6	35.8	91.8	114	164	10.2	21.5	26.5	43.1	5.5	15.8	22.9	31.9	9.11	24.1	31.6	49.8

	PR-1M				BP-1M				CF-1M				CoEM-1M				LP-1M			
	UK	TW	TT	FT	UK	TW	TT	FT	UK	TW	TT	FT	UK	TW	TT	FT	UK	TW	TT	FT
DZiG	2.39	10.6	13.5	25.7	8.65	31.3	35.7	22.5	9.13	16.4	18.2	23.4	2.34	8.33	11	15.1	4.06	12	14.8	21.2
GraphBolt-HP	3.74	11.2	14.3	26	7.66	27.2	31.4	22.2	9.88	16.4	18.5	24.6	2.29	8.1	10.8	14.7	8.14	16.3	19.8	29.8
GraphBolt-3	3.77	11.2	14.3	26.4	7.66	27.2	31.4	22.2	9.88	16.4	18.5	26	2.34	8.11	10.9	15	10	16.8	21.6	30.7
GraphBolt-5	3.74	11.4	14.9	26.2	44.8	128	156	206	10.1	19	21.9	24.6	2.33	8.11	10.9	15	8.14	21.1	25.3	38.9
GraphBolt-7	4.68	12.1	16.3	26.1	83.7	231	285	397	11.7	25.2	29.7	32.7	3.74	14	18.9	27.1	9.31	35.8	43.9	70.5
Res-Inc	4.29	12	15.8	29.8	36.6	91.7	114	164	10.5	21.5	26.4	43.1	5.48	15.8	22.9	31.9	9.1	24.1	31.8	49.7

Figure 11. Execution times (in seconds) for DZiG, GraphBolt and Res-Inc with 1K and 1M edge mutations. The cells are colored to easily compare the performance within each column: a darker shade in the column (green in colored mode and gray in gray-scale) indicates faster execution time in that column, and similarly a lighter shade indicates slower execution time in that column.

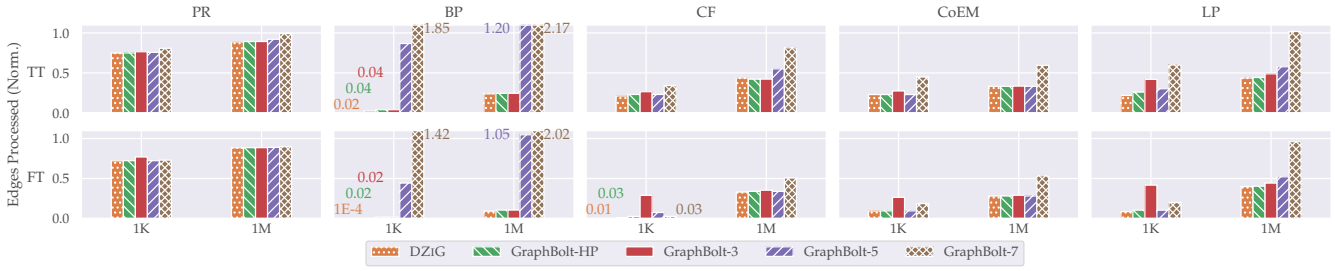


Figure 12. Number of edges processed by DZiG and GraphBolt normalized w.r.t. Res-Inc.

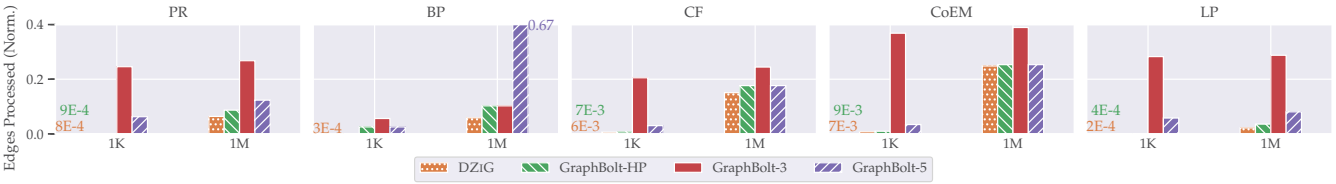


Figure 13. Number of edges processed by DZiG and GraphBolt on YH graph normalized w.r.t. Res-Inc.

	1K					1M				
	PR	BP	CF	CoEM	LP	PR	BP	CF	CoEM	LP
DZiG	1.21	4.51	6.89	3.54	2.46	1.71	10.2	9.45	4.2	2.82
GraphBolt-HP	3.73	14.1	12.3	3.05	3.68	4.49	84.2	15.3	5.47	4.34
GraphBolt-3	5.35	14.4	16.7	6.29	9.31	5.45	84.2	16.4	6.56	9.14
GraphBolt-5	4.54	14.7	13.5	3.05	5.41	4.66	87.6	15.6	5.47	5.96
Res-Inc	4.97	70.6	17.7	13.2	13.2	5.27	68.1	17.4	13	12.7

Figure 14. Execution times (in seconds) for DZiG, GraphBolt and Res-Inc on YH graph with 1K and 1M edge mutations on `r5.24xlarge`.

impacts more intermediate results than 1K edge mutations, demanding more edges to be processed. Furthermore, it is interesting to observe that the number of edge computations by DZiG in BP and CF with 1K edge mutations is much lower than that with 1M edge mutations; in fact, for BP on FT DZiG processes 0.01% of the edges processed by Res-Inc. This directly results in high performance for BP and CF that process 1K edge mutations in just 1.23 seconds and 4.25 seconds respectively. GraphBolt-HP on the other hand processes 1.7-2.66% of the edges, resulting in 5 seconds and 8.39 seconds respectively.

For the larger Yahoo graph (YH) we evaluate DZiG on `r5.24xlarge` with 96 cores. Figure 14 shows the execution times, and Figure 13 shows the corresponding number of edge computations. Our observations are consistent: DZiG is fastest across nearly all cases, and processing 1K mutations is faster than 1M mutations.

6.4 Scaling with Mutation Batch Sizes

Figure 15 shows the execution times for DZiG, GraphBolt and Res-Inc. Since performance appears very close, each plot is divided into two subplots: the top subplot shows performance of DZiG and GraphBolt-HP (here **DZiG-AE** represents DZiG with adaptive execution turned on); and the bottom subplot shows performance for different GraphBolt variants. Finally, Figure 16 shows the corresponding number of edges processed.

As the mutation batch size increases, more number of edges get processed which increases the execution times for both DZiG and GraphBolt. Since Res-Inc restarts from

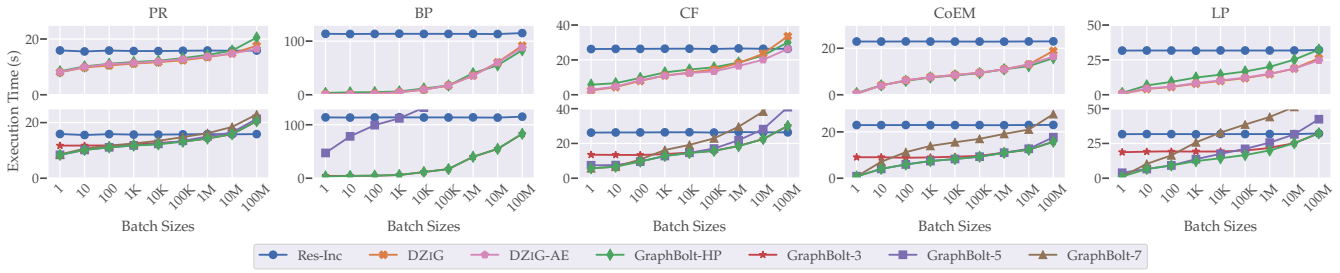


Figure 15. Scaling with mutation batch sizes. Execution times (in seconds) for DZiG, GraphBolt and Res-Inc on TT graph. DZiG-AE represents DZiG with adaptive execution turned on. Since performance of different executions appear very close to each other, the numbers are separated across two subplots for each benchmark: the top subplot shows performance of DZiG, DZiG-AE and GraphBolt-HP; and the bottom subplot shows performance for different GraphBolt variants.

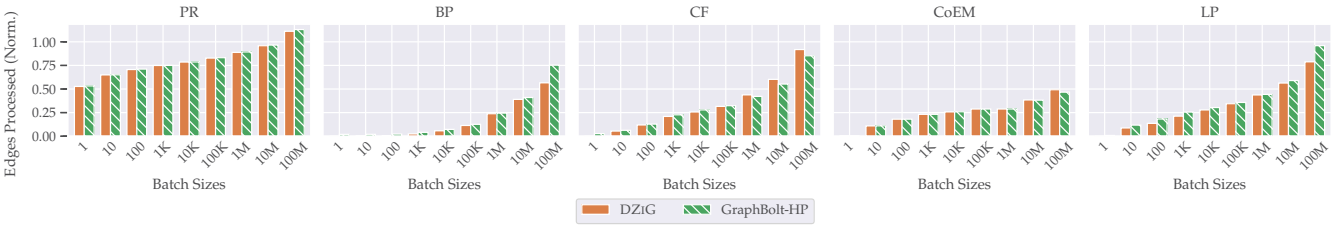


Figure 16. Number of edges processed by DZiG and GraphBolt normalized w.r.t. Res-Inc for different mutation batch sizes.

beginning, it remains unaffected with the number of edge mutations, and hence it exhibits steady performance. The key observation here is that *our DelZero-aware incremental refinement strategy pushes the boundary of effectiveness of dependency-driven processing for streaming graphs to over 10 million simultaneous mutations (at least a few orders of magnitude higher compared to GraphBolt)*: this is visible by comparing the intersection points of DZiG with Res-Inc vs. GraphBolt and Res-Inc. GraphBolt has to rely on hybrid execution which turns off its dependency-driven processing, and with static tuning it gives different (often low) performance. DZiG on the other hand delivers high performance even without adaptive execution, i.e., purely based on its DelZero-aware dependency-driven processing.

With 100 million edge mutations, the number of edge computations by DZiG rises compared to Res-Inc, mainly because of two reasons. First, DelZero-aware incremental processing requires two sub-operations per edge update (removing old difference and adding new difference, even when fused together). And second, each iteration processes the set of mutated edges (unless DelZeros) to propagate direct changes (recall direct changes from Section 4.2), and with 100 million edge mutations, this set becomes large. For example, DZiG on PR processes ~17% more edges compared to Res-Inc, while propagating direct changes itself takes ~20% edge computations.

Benefits of Adaptive Execution: Finally, the benefits of adaptive execution show up as the number of mutations per batch increase. Figure 15 shows that DZiG-AE (DZiG with adaptive execution turned on) is often faster than DZiG for 10 million and 100 million edge mutations. In fact, CF benefits most from adaptive execution which brings down the execution time below Res-Inc for 100 million edge mutations; this is mainly because edge computations in CF involve more operations than those in other benchmarks. For PR on the other hand, even though DZiG-AE gives better performance over DZiG for 100 million edge mutations, DZiG-AE does not noticeably bring down the execution time below Res-Inc. This is because our automatic switching strategy is conservative as it first observes whether the execution took more time than expected, and hence, it misses the switching point by one iteration which affects the final performance. Nevertheless, the adaptive execution accelerates DZiG's processing by up to ~1.2×, mainly because of its highly accurate and lightweight estimation model.

6.5 Sensitivity to Computation Sparsity

We study how DZiG performs across different degrees of computation sparsity. To control the computation sparsity, we vary the threshold value (ϵ) that defines DelZeros. A lower tolerance value represents the case where computations are more sensitive to value changes (i.e., less sparsity), whereas a higher value of tolerance means that computations are less sensitive to value changes (i.e., more sparsity). Figure 17 shows the performance of DZiG, GraphBolt and Res-Inc for PR and LP when the tolerance is varied from 0.2 (higher sparsity) to 0.001 (lower sparsity). DZiG reacts to

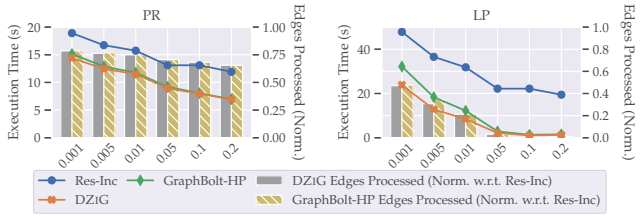


Figure 17. Sensitivity to value changes. Execution time (in seconds) for DZiG, GraphBolt and Res-Inc, and number of edges processed by DZiG and GraphBolt normalized w.r.t. Res-Inc with varying tolerances that define DelZeros.

changes the same way as GraphBolt and Res-Inc; when sparsity is lower (lower values of tolerance), minor changes need to get propagated across edges, which increases the number of transitive changes processed by DZiG. When sparsity increases, DZiG performs less work since only major changes get propagated.

6.6 Memory Overhead

DZiG tracks aggregation values at vertex-level which consumes memory in addition to the dynamic graph data structure. Figure 18 shows the increase in memory footprint per iteration by DZiG and GraphBolt compared to Res-Inc for various graph algorithms and inputs. As we can see, the increase for DZiG is only 3-13% across all the algorithms, which is similar to the increase for GraphBolt. DZiG shows a slightly lesser increase than GraphBolt because the dynamic adjacency lists in DZiG consume a bit more memory (to hold empty slots) compared to the compressed sparse row/column (CSR/CSC) representations used in GraphBolt.

Since the aggregation value types are different across different graph algorithms, the overall increase in memory footprint is also different depending on the size of the aggregation values. For instance, CF tracks 3× more information per vertex than PR, and hence it requires more memory compared to PR where memory increases by only 3-5%.

6.7 Comparison with Other Systems

Even though these systems do not perform incremental refinement that guarantees synchronous processing semantics (unlike DZiG and GraphBolt), we compare the performance of DZiG with Aspen, GraphOne, LLAMA and Stinger on traditional graph processing algorithms: PageRank and SSSP. For cases where benchmarks were not implemented in the their public repository, we implemented the fastest version (e.g., incremental PageRank from [51]); and for cases where multiple implementations of the same benchmark were available, we use the fastest version that gave correct results. Aspen's public repository only supports undirected graphs, which we evaluate.

	UK		TW		TT		FT	
	DZiG	GB	DZiG	GB	DZiG	GB	DZiG	GB
PR	4.8%	5.4%	3.8%	6.1%	3.9%	4.3%	4.1%	4.7%
BP	10.3%	10.3%	9.0%	8.9%	10.0%	9.1%	9.6%	8.8%
CF	12.9%	19.0%	11.4%	15.7%	11.4%	17.1%	12.1%	18.3%
CoEM	5.4%	7.7%	4.5%	4.4%	4.5%	4.4%	4.9%	4.9%
LP	7.9%	7.6%	6.7%	6.5%	5.6%	6.5%	7.0%	6.0%

Figure 18. Increase in memory for DZiG and GraphBolt (GB) w.r.t. Res-Inc.

As shown in Figure 19, both DZiG and GraphBolt are competitive mainly because of their dependency-driven incremental refinement. Furthermore, across all executions, DZiG performs fastest due to its DelZero-aware incremental refinement strategy. While DZiG uses a simple dynamic adjacency list based data structure, it is efficient enough to retain high end-to-end performance (i.e., including graph mutation and PageRank/SSSP processing time). This is because DZiG's data structure provides high locality for parallel vertex and edge operations during processing, while at the same time enabling fast graph ingestion. Finally, we observed that batched mutation with DZiG's data structure is faster than recent works like GraphOne.

7 Related Work

We group several dynamic graph processing solutions based on their support for continuous analysis over streaming graphs, and temporal analysis over static graph snapshots.

Streaming Graph Processing Systems: These systems allow continuous analysis over streaming graphs. While most of these systems perform incremental processing to compute based on changes in graph structure, only GraphBolt [32] and KickStarter [57] perform dependency-driven incremental processing. GraphBolt guarantees BSP semantics using its dependency-driven incremental refinement strategy (discussed in detail in Section 2.2). KickStarter focuses on monotonic graph algorithms like shortest paths and connected components that do not require BSP semantics to guarantee correctness. It maintains the dependency information in the form of dependency trees, and performs an incremental *trimming* process that adjusts the values based on monotonic relationships.

While other systems like Tornado [50], Kineograph [11] and GraphIn [48] perform incremental computation, they do so by triggering the user functions based on graph updates, and allowing the changes to propagate throughout the graph until convergence. Since they do not maintain correct dependencies during incremental computation, they cannot guarantee BSP semantics like DZiG. Tornado uses a bounded async iterations to process the user queries upon graph structure updates. GraphIn identifies the vertices that could be potentially impacted by graph updates using tag propagation, and restarts computation from scratch for those

	Batch Size = 1			Batch Size = 10			Batch Size = 100			Batch Size = 1K			Batch Size = 10K		
	G	PR	SSSP	G	PR	SSSP	G	PR	SSSP	G	PR	SSSP	G	PR	SSSP
DZiG	0.06	8.62	0.05	0.09	10.00	0.04	0.09	10.56	0.05	0.17	11.20	0.06	0.33	11.20	0.04
GraphBolt	1.15	9.04	0.05	1.13	10.55	0.04	1.17	11.14	0.05	1.27	11.70	0.06	1.42	12.25	0.04
Aspen	1E-4	29.48	3.31	3E-4	29.89	3.34	2E-3	29.29	3.31	6E-3	29.80	3.31	7E-3	29.68	3.34
GraphOne	4E-3	19.91	13.24	0.07	19.91	12.58	0.09	19.73	13.09	0.26	19.87	12.90	1.64	20.02	12.58
LLAMA	x	20.55	2.77	x	20.55	2.77	x	20.55	2.77	x	20.55	2.77	x	20.55	2.77
Stinger	0.02	431.18	145.15	0.03	422.19	146.55	2.88	437.11	145.12	2.02	417.95	144.39	7.06	488.78	146.55

Figure 19. Graph mutation (G) time (in seconds), and execution times (in seconds) for PageRank (PR) and Single Source Shortest Path (SSSP) with different mutation batch sizes. LLAMA does not provide graph mutation support (marked with ×).

identified vertices. [52] uses incremental GIM-V (generalized iterative matrix vector multiplication) to perform incremental computation. GraphInc [8] performs incremental processing by saving all messages across edges along with computed states (which requires large storage) and replaying the computation to incorporate the changes. Finally, systems like GraphJet [49] and [20] are targeted towards custom graph bases analytics like real-time content recommendations.

Systems like LLAMA [31], STINGER [15], Aspen [14] and GraphOne [27] focus on designing efficient dynamic graph data structures, and their processing units do not support incremental computation. LLAMA, STINGER and GraphOne use adjacency lists with blocks of edges for efficient insertion, and indexing strategies for efficient retrieval, while Aspen uses C-Trees for efficient fine-grained multiversioning. We compare the performance of DZiG with these systems in Section 6.7 and show that DZiG's dynamic adjacency lists enable fast graph mutation and allow DZiG to retain high end-to-end efficiency.

Finally, while graph databases like [38, 42] allow the graph structure to be modified, Graphflow [25] is an active graph database that supports continuous subgraph queries using a new incremental view maintenance algorithm.

Systems for Processing Graph Snapshots: Systems like GraphTau [22], ImmortalGraph [35], Chronos [21] and [56] operate on a group of temporally-related graph snapshots that capture the evolution of the graph structure over time. Since the graph snapshots are structurally similar to each other (due to being temporally correlated), these systems use incremental computation by reusing the results computed for a snapshot to compute those for another snapshot. They feed results across consecutive graph snapshots, and propagate the changes throughout the graph until convergence. Such style of incremental processing is suitable for certain kinds of self-fixing graph algorithms like shortest paths, but it does not guarantee BSP semantics. Since temporal graph snapshots can be processed individually as well (without incremental computation), static graph processing systems [17, 18, 26, 30, 36, 39, 45, 47, 51, 55, 58, 59, 61, 67] can also be used, but at the expense of high performance costs.

Finally, LiveGraph [68] is a recent graph storage system for both, transactional graph and graph analytics workloads. It employs Transactional Edge Log (TEL) structure and maintains low latency with high throughput.

Generalized Stream Processing Systems & Their Derivatives: Stream processing systems like [1, 2, 4, 5, 9, 37, 44, 46, 53, 63, 64] operate on generalized streams of data tuples to support continuous or real-time analysis. Differential Dataflow [34] extends Naiad's timely dataflow with incremental processing operators that capture value changes, thereby allowing incremental computation as the input tuples change. The generality of these systems allow development of streaming graph-based analytics on top of them. However, as shown in [32], this generality comes at a performance cost that graph systems avoid.

Custom Incremental Graph Solutions: Several incremental graph algorithms have been developed in literature that operate on changing graph structures. Since these works are problem-specific, they develop tailored solutions to efficiently solve the given graph problem. For example, [3, 13, 23, 33] propose incremental PageRank algorithms and use custom techniques to optimize for faster convergence. Incremental View Maintenance (IVM) algorithms [6, 19, 41], on the other hand, maintain a consistent view of the input by reusing computed results to handle general queries. As discussed in [34], they require heavy recomputation.

8 Conclusion

We developed DZiG, a streaming graph processing system that retains efficiency in presence of sparse computations and guarantees BSP semantics. DZiG's DelZero-aware refinement strategy expresses incremental computations recursively to safely maintain sparsity throughout the refinement process. Our evaluation showed that DZiG outperforms state-of-the-art systems, and our DelZero-aware refinement strategy pushes the boundary of dependency-driven processing for streaming graphs by orders of magnitude.

Acknowledgements

We would like to thank our shepherd Tim Harris 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 *Conference on Innovative Data Systems Research (CIDR '05)*, 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 *Proceedings of the ACM International Conference on Management of Data (SIGMOD '13)*, pages 577–588, 2013.

[3] Bahman Bahmani, Abdur Chowdhury, and Ashish Goel. Fast Incremental and Personalized PageRank. *Proceedings of the VLDB Endowment (PVLDB '10)*, 4(3):173–184, 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 *Proceedings of the International Middleware Conference (Middleware '14)*, pages 61–72, 2014.

[6] Jose A. Blakeley, Per-Ake Larson, and Frank Wm Tompa. Efficiently Updating Materialized Views. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD '86)*, pages 61–71, 1986.

[7] Paolo Boldi and Sebastiano Vigna. The WebGraph Framework I: Compression Techniques. In *Proceedings of the International Conference on World Wide Web (WWW '04)*, pages 595–601, 2004.

[8] Zhuhua Cai, Dionysios Logothetis, and Georgos Siganos. Facilitating Real-Time Graph Mining. In *Proceedings of the International Workshop on Cloud Data Management (CloudDB '12)*, pages 1–8, 2012.

[9] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. Apache Flink™: Stream and Batch Processing in a Single Engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 38:28–38, 2015.

[10] Meeyoung Cha, Hamed Haddadi, Fabricio Benevenuto, and Krishna P. Gummadi. Measuring User Influence in Twitter: The Million Follower Fallacy. In *Proceedings of the International AAAI Conference on Web and Social Media (ICWSM '10)*, pages 10–17, 2010.

[11] Raymond Cheng, Ji Hong, Aapo Kyrola, Youshan Miao, Xuetian Weng, Ming Wu, Fan Yang, Lidong Zhou, Feng Zhao, and Enhong Chen. Kineograph: Taking the Pulse of a Fast-changing and Connected World. In *Proceedings of the European Conference on Computer Systems (EuroSys '12)*, pages 85–98, 2012.

[12] CilkPlus: https://www.cilkplus.org/.

[13] Prasanna Desikan, Nishith Pathak, Jaideep Srivastava, and Vipin Kumar. Incremental Page Rank Computation on Evolving Graphs. In *Proceedings of the International Conference on World Wide Web (WWW '05)*, pages 1094–1095, 2005.

[14] Laxman Dhulipala, Guy E. Blelloch, and Julian Shun. Low-Latency Graph Streaming Using Compressed Purely-Functional Trees. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '19)*, page 918–934, 2019.

[15] David Ediger, Rob Mccoll, Jason Riedy, and David A. Bader. STINGER: High Performance Data Structure for Streaming Graphs. In *IEEE Conference on High Performance Extreme Computing (HPEC '12)*, pages 1–5, 2012.

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

[17] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. PowerGraph: Distributed Graph-parallel Computation on Natural Graphs. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI '12)*, pages 17–30, 2012.

[18] 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 *USENIX Symposium on Operating Systems Design and Implementation (OSDI '14)*, pages 599–613, 2014.

[19] Ashish Gupta, Inderpal Singh Mumick, and V. S. Subrahmanian. Maintaining Views Incrementally. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD '93)*, pages 157–166, 1993.

[20] Pankaj Gupta, Venu Satuluri, Ajeet Grewal, Siva Gurumurthy, Volodymyr Zhabiuk, Quannan Li, and Jimmy Lin. Real-Time Twitter Recommendation: Online Motif Detection in Large Dynamic Graphs. *Proceedings of the VLDB Endowment (PVLDB '14)*, 7(13):1379–1380, 2014.

[21] Wentao Han, Youshan Miao, Kaiwei Li, Ming Wu, Fan Yang, Lidong Zhou, Vijayan Prabhakaran, Wenguang Chen, and Enhong Chen. Chronos: A Graph Engine for Temporal Graph Analysis. In *Proceedings of the European Conference on Computer Systems (EuroSys '14)*, pages 1:1–1:14, 2014.

[22] Anand Padmanabha Iyer, Li Erran Li, Tathagata Das, and Ion Stoica. Time-Evolving Graph Processing at Scale. In *Proceedings of the International Workshop on Graph Data Management Experiences and Systems (GRADES '16)*, page 5, 2016.

[23] Sepandar Kamvar, Taher Haveliwala, and Gene Golub. Adaptive methods for the computation of PageRank. *Linear Algebra and its Applications*, 386:51–65, 2003.

[24] U Kang, Duen Horng, and Christos Faloutsos. Inference of Beliefs on Billion-Scale Graphs. In *Large-scale Data Mining: Theory and Applications (LDMTA '10)*, 2010.

[25] Chathura Kankanamge, Siddhartha Sahu, Amine Mhedbhi, Jeremy Chen, and Semih Salihoglu. Graphflow: An active graph database. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD '17)*, pages 1695–1698, 2017.

[26] Pradeep Kumar and H. Howie Huang. G-Store: High-Performance Graph Store for Trillion-Edge Processing. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '16, 2016.

[27] Pradeep Kumar and H. Howie Huang. GRAPHONE: A Data Store for Real-time Analytics on Evolving Graphs. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST '19)*, pages 249–263, 2019.

[28] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. What is Twitter, A Social Network or a News Media? In *Proceedings of the International Conference on World Wide Web (WWW '10)*, pages 591–600, 2010.

[29] Daan Leijen, Benjamin Zorn, and Leonardo de Moura. Mimalloc: Free List Sharding in Action. In *Asian Symposium on Programming Languages and Systems (APLAS '19)*, pages 244–265, 2019.

[30] Heng Lin, Xiaowei Zhu, Bowen Yu, Xiongchao Tang, Wei Xue, Wenguang Chen, Lufei Zhang, Torsten Hoefler, Xiaosong Ma, Xin Liu, Weimin Zheng, and Jingfang Xu. ShenTu: Processing Multi-Trillion Edge Graphs on Millions of Cores in Seconds. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis (SC '18)*, 2018.

[31] Peter Macko, Virendra J Marathe, Daniel W Margo, and Margo I Seltzer. LLAMA: Efficient graph analytics using Large Multiversioned Arrays. In *IEEE International Conference on Data Engineering (ICDE '15)*, pages 363–374, April 2015.

[32] Mugilan Mariappan and Keval Vora. GraphBolt: Dependency-Driven Synchronous Processing of Streaming Graphs. In *Proceedings of the European Conference on Computer Systems (EuroSys '19)*, pages 25:1–25:16, 2019.

[33] Frank McSherry. A Uniform Approach to Accelerated PageRank Computation. In *Proceedings of the International Conference on World Wide Web (WWW '05)*, pages 575–582, 2005.

[34] Frank McSherry, Derek G. Murray, Rebecca Isaacs, and Michael Isard. Differential Dataflow. In *Conference on Innovative Data Systems*

Research (CIDR '13), 2013.

[35] Youshan Miao, Wentao Han, Kaiwei Li, Ming Wu, Fan Yang, Lidong Zhou, Vijayan Prabhakaran, Enhong Chen, and Wenguang Chen. ImmortalGraph: A System for Storage and Analysis of Temporal Graphs. *ACM Transactions on Storage (TOS)*, 11(3):14:1–14:34, 2015.

[36] Svilen R. Mihaylov, Zachary G. Ives, and Sudipto Guha. REX: Recursive, Delta-Based Data-Centric Computation. *Proceedings of the VLDB Endowment (PVLDB '12)*, 5(11):1280–1291, 2012.

[37] Derek G. Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. Naiad: A Timely Dataflow System. In *ACM Symposium on Operating Systems Principles (SOSP '13)*, pages 439–455, 2013.

[38] Neo4J. www.neo4j.com.

[39] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. A Lightweight Infrastructure for Graph Analytics. In *ACM Symposium on Operating Systems Principles (SOSP '13)*, pages 456–471, 2013.

[40] Kamal Nigam and Rayid Ghani. Analyzing the Effectiveness and Applicability of Co-training. In *Proceedings of the International Conference on Information and Knowledge Management (CIKM '00)*, pages 86–93, 2000.

[41] Vivek Nigam, Limin Jia, Boon Thau Loo, and Andre Scedrov. Maintaining Distributed Logic Programs Incrementally. In *Computer Languages, Systems & Structures*, pages 125–136, 2011.

[42] OrientDB. https://orientdb.com/.

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

[44] Frederick Reiss, Kurt Stockinger, Kesheng Wu, Arie Shoshani, and Joseph M. Hellerstein. Enabling Real-Time Querying of Live and Historical Stream Data. In *International Conference on Scientific and Statistical Database Management (SSDBM '07)*, pages 28–, 2007.

[45] Amitabha Roy, Laurent Bindschaedler, Jasmina Malicevic, and Willy Zwaenepoel. Chaos: Scale-out Graph Processing from Secondary Storage. In *ACM Symposium on Operating Systems Principles (SOSP '15)*, pages 410–424, 2015.

[46] Pratanu Roy, Arijit Khan, and Gustavo Alonso. Augmented Sketch: Faster and More Accurate Stream Processing. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD '16)*, pages 1449–1463, 2016.

[47] Semih Salihoglu and Jennifer Widom. GPS: A Graph Processing System. In *International Conference on Scientific and Statistical Database Management (SSDBM '13)*, pages 22:1–22:12, 2013.

[48] 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 *Proceedings of the International European Conference on Parallel and Distributed Computing (Euro-Par '16)*, pages 319–333, 2016.

[49] Aneesh Sharma, Jerry Jiang, Praveen Bommannavar, Brian Larson, and Jimmy Lin. GraphJet: Real-time Content Recommendations at Twitter. *Proceedings of the VLDB Endowment (PVLDB '16)*, 9(13):1281–1292, 2016.

[50] Xiaogang Shi, Bin Cui, Yingxia Shao, and Yunhai Tong. Tornado: A System For Real-Time Iterative Analysis Over Evolving Data. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD '16)*, pages 417–430, 2016.

[51] Julian Shun and Guy E. Blelloch. Ligra: A Lightweight Graph Processing Framework for Shared Memory. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '13)*, pages 135–146, 2013.

[52] Toyotaro Suzumura, Shunsuke Nishii, and Masaru Ganse. Towards Large-scale Graph Stream Processing Platform. In *Proceedings of the International Conference on World Wide Web (WWW '14)*, pages 1321–1326, 2014.

[53] 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 *Proceedings of the ACM International Conference on Management of Data (SIGMOD '14)*, pages 147–156, 2014.

[54] Leslie G Valiant. A Bridging Model for Parallel Computation. *Communications of the ACM*, 33(8):103–111, 1990.

[55] Keval Vora. Lumos: Dependency-Driven Disk-based Graph Processing. In *USENIX Annual Technical Conference (USENIX ATC '19)*, pages 429–442, 2019.

[56] Keval Vora, Rajiv Gupta, and Guoqing Xu. Synergistic Analysis of Evolving Graphs. *ACM Transactions on Architecture and Code Optimization (TACO '16)*, 13(4):32, 2016.

[57] Keval Vora, Rajiv Gupta, and Guoqing Xu. KickStarter: Fast and Accurate Computations on Streaming Graphs via Trimmed Approximations. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '17)*, pages 237–251, 2017.

[58] Keval Vora, Sai Charan Koduru, and Rajiv Gupta. ASPIRE: Exploiting Asynchronous Parallelism in Iterative Algorithms Using a Relaxed Consistency Based DSM. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA '14)*, pages 861–878, 2014.

[59] 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 Annual Technical Conference (USENIX ATC '17)*, pages 507–522, 2016.

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

[61] 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 *Proceedings of the ACM Symposium on Cloud Computing (SoCC '15)*, pages 408–421, 2015.

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

[63] 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 *USENIX Workshop on Hot Topics in Cloud Computing (HotCloud '12)*, 2012.

[64] Erik Zeitler and Tore Risch. Massive Scale-out of Expensive Continuous Queries. In *Proceedings of the VLDB Endowment (PVLDB '11)*, pages 1181–1188, 2011.

[65] Yunhong Zhou, Dennis Wilkinson, Robert Schreiber, and Rong Pan. Large-Scale Parallel Collaborative Filtering for the Netflix Prize. In *International Conference on Algorithmic Applications in Management (AAIM '08)*, pages 337–348, 2008.

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

[67] Xiaowei Zhu, Wenguang Chen, Weimin Zheng, and Xiaosong Ma. Gemini: A Computation-Centric Distributed Graph Processing System. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI '16)*, pages 301–316, 2016.

[68] Xiaowei Zhu, Guanyu Feng, Marco Serafini, Xiaosong Ma, Jiping Yu, Lei Xie, Ashraf Aboulnaga, and Wenguang Chen. LiveGraph: A Transactional Graph Storage System with Purely Sequential Adjacency List Scans. *Proceedings of the VLDB Endowment (PVLDB '20)*, 13(7):1020–1034, 2020.