Controlling Memory Footprint of Stateful Streaming Graph Processing

Pourya Vaziri and Keval Vora, *Simon Fraser University*

https://www.usenix.org/conference/atc21/presentation/vaziri

This paper is included in the Proceedings of the
2021 USENIX Annual Technical Conference.

July 14–16, 2021

978-1-939133-23-6

Controlling Memory Footprint of Stateful Streaming Graph Processing

Pourya Vaziri
School of Computing Science
Simon Fraser University
British Columbia, Canada
pvaziri@cs.sfu.ca

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

Abstract

With growing interest in efficiently analyzing dynamic graphs, streaming graph processing systems rely on stateful iterative models where they track the intermediate state as execution progresses in order to incrementally adjust the results upon graph mutation. We observe that the intermediate state tracked by these stateful iterative models significantly increases the memory footprint of these systems, which limits their scalability on large graphs.

In this paper, we develop memory-efficient stateful iterative models that demand much less memory capacity to efficiently process streaming graphs and deliver the same results as provided by existing stateful iterative models. First, we propose a ✓ *Selective Stateful Iterative Model* where the memory footprint is controlled by selecting a small portion of the intermediate state to be maintained throughout execution. Then, we ✓ propose a *Minimal Stateful Iterative Model* that further reduces the memory footprint by exploiting key properties of graph algorithms. We develop incremental processing strategies for both of our models in order to correctly compute the effects of graph mutations on the final results even when intermediate states are not available. Evaluation shows our memory-efficient models are effective in limiting the memory footprint while still retaining most of the performance benefits of traditional stateful iterative models, hence being able to scale on larger graphs that could not be handled by the traditional models.

1 Introduction

Streaming graph processing systems [4, 6, 13, 16, 17, 27, 28, 35] aim to deliver real-time results as the graph structure changes via a continuous stream of edge and vertex mutations. These systems are equipped with efficient iterative processing models that are broadly classified into two types: the *stateless iterative model* and the *stateful iterative model*.

Stateless iterative models used in [6, 13, 27, 28] perform regular iterative processing without capturing additional intermediate values that describe the execution history. When the graph structure mutates (e.g., new edges/vertices get added or old vertices/edges get removed), they either continue the iterative computation without correctly incorporating the impact of mutations on already computed values (i.e., not guaranteeing accuracy of final results) [28]; or, they conservatively deduce the set of values that could be potentially affected due to mutation (using techniques like tag propagation [27]) and recompute those values from scratch.

On the other hand, stateful iterative models used in [16, 17, 35] capture the intermediate state (representing execution history) as computation progresses, often in terms of intermediate values computed for vertices in each iteration. When the graph structure mutates, only the change in values resulting from those mutations are iteratively propagated to correct the intermediate state and compute accurate final results. As expected, stateful iterative models are more efficient in generating correct final results compared to stateless iterative models simply because the stateful models operate on the precise set of intermediate values that get affected by the change. Stateless iterative models, on the other hand, need to be conservative while generating accurate results since they do not capture intermediate values representing the execution history, and hence, they end up demanding much more computation.

While traditional models like [4] maintain the intermediate state in the order of edges for each iteration, recent works [16, 17, 35] capture information in terms of intermediate values computed for vertices, which results in much lesser memory footprint compared to traditional models. In these systems, the amount of intermediate state saved by the stateful iterative models depends on the nature of the graph algorithms they support. For example, models that operate on asynchronous graph analytics algorithms like BFS, shortest paths, connected components, etc. leverage the monotonic relationship of the values in the algorithm to adjust them directly [35], which requires capturing only $O(|V|)$ intermediate state. On the other hand, models that support synchronous graph algorithms like Co-Training Expectation Maximization (CoEM), Collaborative Filtering (CF), etc. capture the

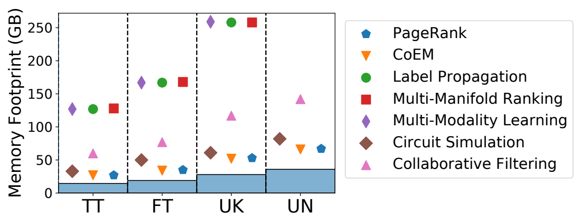


Figure 1: Memory footprint of stateful iterative model across different graph algorithms (shown as different points) and graph datasets (details in Table 2). Solid bars represent the memory consumed by the graph structure.

intermediate state at every iteration in order to incrementally recompute the values iteration-by-iteration and guarantee results equivalent to Bulk Synchronous Parallel (BSP) [32] execution from scratch [16, 17]. Furthermore, the intermediate state in each iteration often contains the aggregation results for vertices along with the vertex values, and their size (in bytes) depends on the graph algorithm being run. For example, CoEM requires 8 bytes per intermediate state of a vertex whereas CF consumes 16 bytes per intermediate state. Other graph algorithms like Multi-Modality Learning (MML) and Label Propagation (LP) have even larger intermediate states depending on the number of labels they operate on.

We profiled GraphBolt [17], a recent state-of-the-art streaming graph processing system, to measure the amount of memory consumed by the intermediate state across different graph algorithms and graph datasets. As shown in Figure 1, the intermediate state consumes at least twice the amount of memory required to hold the input graph itself; for the Collaborative Filtering algorithm this factor increases to over 4$\times$ whereas the intermediate state in algorithms like Multi-Manifold Ranking and Label Propagation consumes over an order of magnitude more memory compared to the input graph.

Such high memory footprint significantly limits the scalability of the stateful iterative models on large graphs. For instance we observed that three of the graph algorithms in Figure 1 ran out of memory (with 320GB memory capacity) for the UN graph. With graph datasets growing at a faster rate than memory capacity, it becomes crucial to develop stateful iterative models that do not demand large memory capacities just to hold the intermediate states.

In this paper, we develop novel memory-efficient stateful iterative models that demand much less memory capacity to efficiently process streaming graphs and provide the same BSP guarantees as existing state-of-the-art streaming graph processing systems. First, we propose a *Selective Stateful Iterative Model* where the memory footprint is controlled by selecting a small portion of the intermediate state to be maintained during execution. And then, we propose a *Minimal Stateful Iterative Model* that specializes the incremental processing for certain graph algorithms (depending on their update functions) to drastically reduce the memory footprint.

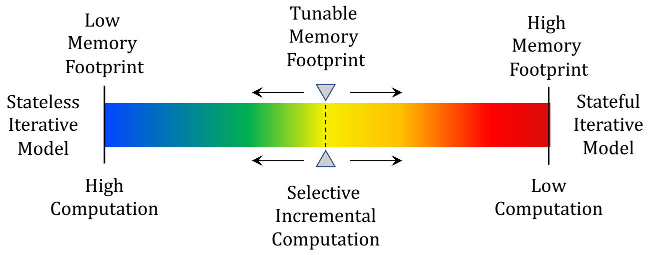


Figure 2: Sliding scale of memory and computation requirements between stateless iterative model and stateful iterative model. By capturing only partial intermediate state (selectively at a fine-grained level), memory consumption can be reduced at the cost of increased computation.

Selective Stateful Iterative Model. The key insight here is that vertex computations within an iteration are independent of each other, and hence, incrementally recomputing the value of a vertex is only dependent on the intermediate state saved for that vertex (i.e., not dependent on the states saved for other vertices). Moreover, the usefulness of different portions of the intermediate state (in terms of the amount of computation pruned out upon graph mutation) is different; this means, intermediate states for certain vertices would end up reducing more computation than those for other vertices.

Based on these insights, our selective stateful iterative model tunes its memory footprint depending on the available main memory by selecting the intermediate vertex states to be captured at a fine-grained level (illustrated in Figure 2). While this enables the model to scale on large streaming graphs, performing incremental computation using the partial intermediate state becomes challenging. This is because the value changes resulting from graph mutations are typically merged with the intermediate vertex states to propagate subsequent (transitive) changes throughout the graph, and how to compute the effects of value changes with missing intermediate states remains unclear. To address this, we develop a *selective incremental processing technique* that correctly computes the effects of changes even when intermediate vertex states are not available. Our strategy captures the nuances of the interaction between vertices with intermediate states and those without intermediate states to ensure that the vertices in the latter set correctly participate in the iteration-by-iteration incremental computation.

Minimal Stateful Iterative Model. In this model, we specialize the incremental processing for certain algorithms. Specifically, algorithms like CoEM and PageRank involve operations that are purely distributive, i.e., outgoing value changes from a given vertex can be directly computed based on the incoming value changes to the vertex. We formalize this characteristic as the novel *distributive update property* to understand the scope of graph computations that are purely distributive. Then, for algorithms that satisfy the distributive update property, we identify that the effects of graph mutations can be propagated iteration-by-iteration throughout the graph even without using the intermediate states for most of the vertices. Specifically, only the intermediate states for vertices that get directly affected by graph mutation are needed to perform incremental processing over the entire graph.

We use this insight to develop the minimal stateful iterative model, where the amount of intermediate state gets aggressively reduced to a known subset of vertices on which mutations take place, hence consuming a much smaller memory footprint that is dependent on the graph mutations instead of the original graph size. Upon graph mutation, the incremental computation strategy directly operates on changes even for vertices without intermediate states, while also carefully adjusting the available intermediate states for vertices that are directly impacted by mutation.

Results. Our proposed techniques are general, and can be incorporated in any streaming graph processing system to leverage incremental processing without incurring high memory footprint. We implemented both of our proposed memory-efficient models in GraphBolt [17]; since GraphBolt's existing stateful iterative model maintains intermediate states for all the vertices in the graph, it becomes a natural baseline to demonstrate the effectiveness of our proposed models.

Our evaluation with five real-world graphs and seven synchronous graph algorithms shows that our models demand significantly less memory capacity in comparison to GraphBolt, while still retaining most of its performance benefits. Specifically, the memory footprint of our selective stateful iterative model is dependent on the amount of intermediate state saved; for instance, by selecting only 20% of the intermediate state to be saved, the memory footprint is 35-70% smaller than that for GraphBolt. Furthermore, our minimal stateful iterative model reduces the memory footprint by 28-58% even when it is used for algorithms that have smaller intermediate state to begin with. This allows our memory-efficient stateful iterative models to scale to very large graphs that could not be handled by GraphBolt with the available memory capacity.

2 Background

We briefly review the streaming graph processing model and the stateful iterative processing that performs incremental computation.

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). The change in graph structure is also referred to as mutation of graph structure, and each individual update arriving from the stream is also called a mutation. Streaming graph processing systems [4, 16, 17, 27, 28, 35] operate on streaming graphs to continuously produce results consistent with the latest graph structure.

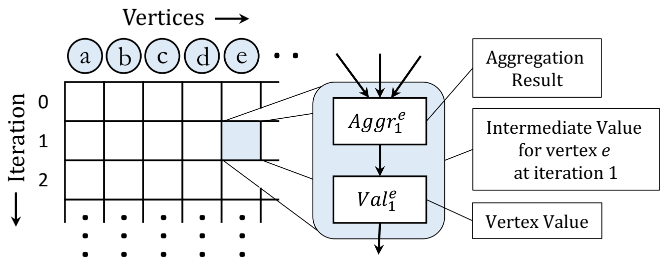


Figure 3: Intermediate state in terms of values relevant for vertices in each iteration. Each intermediate value consists of the aggregation value (to incrementally merge differences) and the vertex value (to compute outgoing differences).

Stateful Iterative Processing Model. *Stateful iterative processing models* used in recent systems like [16, 17, 35] reduce the amount of computation to be performed upon graph mutation using incremental processing. The main idea is to track the intermediate state that captures the necessary details of execution history so that when the graph structure mutates, only the relevant parts of execution history are adjusted or recomputed. To guarantee end results that are same as a Bulk Synchronous Parallel (BSP) execution [32] starting from scratch, GraphBolt employs a *dependency-driven incremental refinement* strategy [17], which incrementally recomputes (or *refines*) intermediate states by propagating changes or *differences* in values resulting due to the graph structure mutation. It propagates the changes in iteration-by-iteration manner, so that correctness guarantees (in form of BSP semantics) are retained for every iteration all the way till the end, hence ensuring that the final result are same as that from a BSP execution from scratch. DZiG [16] (built in GraphBolt) improves the dependency-driven incremental computation by developing a *DelZero-Aware incremental refinement* strategy that retains computation sparsity as iterations progress.

Tracking Intermediate State in Memory. The intermediate state in stateful iterative models is in form of values relevant for vertices at each iteration. As shown in Figure 3, these intermediate values often consist of two components: first, the result of aggregation (also called *aggregation value*) at each vertex that collects values from its incoming edges; and second, the value computed for that vertex. Maintaining both of these values as intermediate state becomes crucial because iterative algorithms often use selective scheduling mechanisms in order to suppress propagation of minor changes (e.g., change less than `1e-2` threshold). In such cases, the outgoing vertex value for a given iteration may not be based on its aggregation result since the latter can hold multiple minor changes which may not have been propagated to the outgoing neighbors. By explicitly tracking the two values, differences can be correctly computed and propagated based on values visible to the neighbors.

As expected, tracking this intermediate state increases the

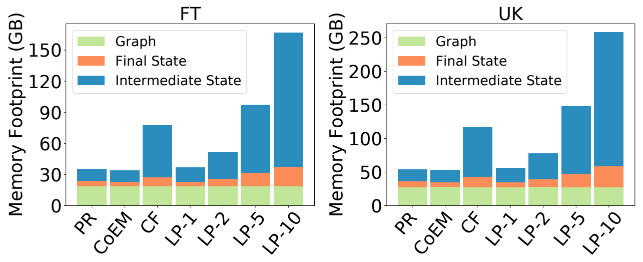


Figure 4: Memory consumption of different components in stateful iterative model: graph structure, final vertex results, and intermediate state. The intermediate state requires different amount of memory across different graph algorithms. On Label Propagation, the memory consumed by intermediate state increases as number of labels increase (indicated by LP-k where k is the number of labels).

memory footprint of such incremental processing techniques. While Figure 1 shows the high memory footprint for different graph algorithms on graph datasets, Figure 4 compares the size of intermediate state relative to the remaining memory consumption of the process (majority of which is taken by the input graph structure and then vertex frontiers). Even though PageRank and CoEM operate on scalar values, their intermediate state and final state consumes nearly as much memory as the remainder of the process. Collaborative Filtering operates on two factors per vertex (i.e., its vertex state is a size-2 vector), and hence, its intermediate state ends up consuming up to 80% additional memory compared to the stateless execution. Finally, Label Propagation operates on feature vectors; as shown in Figure 4, increasing the number of features directly increases the amount of memory consumed by the intermediate state. In fact, maintaining intermediate states with 10 features requires 129GB additional memory for FT and 198GB additional memory for UK, which increases the memory footprint by 3.44× and 3.3× respectively compared to their stateless executions. Such high amount of memory consumption significantly limits the applicability of the stateful iterative processing model on large graphs.

3 Selective Stateful Iterative Model

3.1 Intuition & Main Idea

Maintaining intermediate state essentially allows incremental processing where the effects of graph mutation are propagated in form of value changes throughout the graph. On the other extreme when intermediate state is not maintained, vertex values that are recomputed in a given iteration have to be pushed out since there is no way to determine whether the new values are different from ones computed prior to graph mutation. We observe that every single vertex computation, either in incremental manner with intermediate state or from scratch without intermediate state, is a local computation. This means the value for a given vertex can be computed as long as the right values arrive from its in-neighbors (either in form of value differences or actual values). Hence, we selectively trade off the benefits of incremental computation with reduced memory footprint at a fine-grained level.

Our selective stateful iterative model tracks the intermediate states of only a subset of vertices instead of all the vertices in the graph. For vertices whose intermediate states are not tracked, the model reconstructs their states on-the-fly so that changes resulting from graph mutation can be directly propagated. As illustrated in Figure 2, this allows us to limit the memory footprint by directly controlling the subset of vertices whose intermediate values are tracked, at the cost of performing more computation for vertices whose intermediate states are not tracked.

For simplicity, the vertices whose intermediate states are tracked are called *tracked vertices* whereas the remaining vertices are called *untracked vertices*.

3.2 Tracking Useful Vertex States

In order to selectively track intermediate states, we need to answer two main questions: first, how many vertices should be tracked, and second, which specific vertices should be tracked.

A) How many vertices should be tracked? To maximize the benefits of stateful incremental processing, tracking as many vertices as possible becomes an ideal choice. Hence, we can compute the number of tracked vertices based on the memory capacity (or budget) assigned for the process. Specifically, the size of the intermediate vertex states (which is algorithm dependent) can be determined during initialization, which can be used to bound the number of vertices to be tracked using the available memory budget:

$$mem\_budget \geq k \times state\_size \times t + base\_size$$

where k is the number of vertices whose states are tracked, mem_budget is the available memory capacity, $state\_size$ is the size of intermediate vertex state, t is the number of iterations for which intermediate state should be captured, and $base\_size$ is the memory consumed by other data structures in the system (e.g., input graph structure, vertex frontiers, stream buffers, etc.) along with additional capacity for the graph structure to grow over time. Hence, the above equation can be rewritten to maximize the number of tracked vertices k as:

$$\underset{k}{\text{argmin}}\ |mem\_budget - (k \times state\_size \times t + base\_size)|$$

Note that the majority of $base\_size$ is consumed by the graph data structure, whereas the remaining structures like vertex frontiers (which are simply boolean arrays) often consume less than 10% memory compared to the graph data structure.

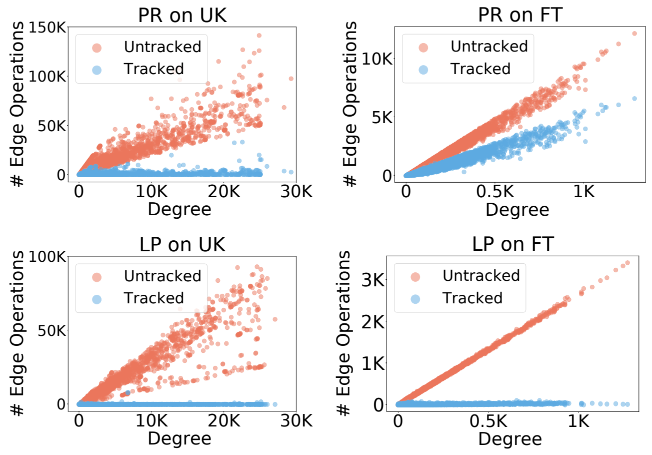


Figure 5: Number of edge operations performed for tracked and untracked vertices based on their in-degrees. Tracking high in-degree vertices reduces more edge operations compared to tracking low in-degree vertices.

B) Which vertices should be tracked? A naive way to select vertices to be tracked can be using random sampling, where tracking of intermediate states can be enabled for a random subset of vertices. While such a strategy easily allows selecting vertices, it remains oblivious of how incremental processing gets performed, and hence it fails to maximize the benefits of incremental processing. Since different vertices require different amount of computation depending on how values get propagated throughout the graph, we must ideally select those vertices that demand high computation so that most of their computation can be effectively eliminated by incremental processing. To do so, we consider the 'usefulness' of the intermediate state for each vertex, where the usefulness of an intermediate state is informally defined as the amount of computation it ends up reducing for that vertex. The usefulness of an intermediate state depends on several dynamic factors including the distance (in terms of number of hops) from the vertices where mutations got applied, and the sensitivity of the graph algorithm to changes in graph structure. Since accurately computing such a metric is infeasible for the general case, we approximate the usefulness of an intermediate state using a vertex-local heuristic.

To develop our heuristic, we profiled the amount of computation performed on each vertex when processing a given graph snapshot. The computation for each vertex is measured in terms of the number of edge operations performed for that vertex; this is because edge operations are expensive (often involve atomic writes and random accesses) and they are the primary candidates that incremental processing attempts to reduce [17]. Figure 5 correlates the number of edge operations for different vertices with their in-degrees for two executions: first, the execution where all vertices' intermediate states are tracked (i.e., GraphBolt's dependency-driven incremental processing); and second, the execution where computation is started from scratch (i.e., no intermediate state is tracked). As we can see vertices with higher in-degree demand more computation when their intermediate values are not tracked, and tracking their values reduces their computation requirements. For instance, the top 20% of the high in-degree vertices contribute to up to 34.1-94.9% of the total number of edge operations in Figure 5. Hence, tracking the intermediate states for high in-degree vertices is most useful. On the other hand, the savings for low in-degree vertices are small (visible from the gap between orange points and blue points for low in-degree vertices) since those vertices demand fewer computation even when their intermediate states are not tracked. Therefore, we track the intermediate states for top-*k* vertices ranked with highest in-degree.

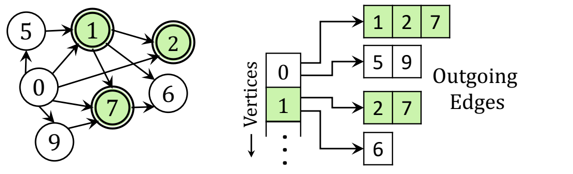
Efficiently Maintaining Tracked Vertex Set. When the graph snapshot gets initialized, a linear pass is performed over the vertices, and the vertex ids are maintained in a degree-ordered max-heap. The vertex ids whose intermediate states must be tracked are then selected in linear time from the heap.

As execution progresses and graph structure mutates, the heap is incrementally adjusted to reflect changes in vertex orderings based on their degree changes. Whenever graph mutations significantly impacts the top-*k* vertex ranking, the subset of tracked vertices can be refreshed to eliminate certain vertices and add new vertices. As the set of tracked vertices gets updated, the intermediate state for vertices that get newly added to the tracked set needs to become available for subsequent incremental processing. This is achieved during incremental refinement by keeping those newly added vertices active during the iterations so that their intermediate states get incrementally computed, which are then tracked. Vertices that get removed from the tracked set are handled by simply turning off their tracking during the next incremental refinement, and releasing the memory allocated to their tracked state.

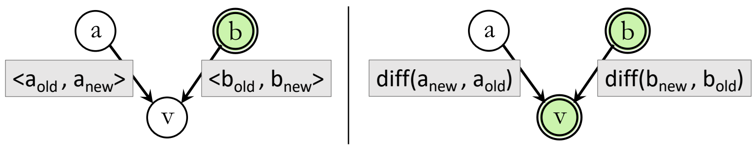
3.3 Incremental Processing upon Mutation

With intermediate states available for only a subset of vertices, propagating changes resulting from graph mutations becomes challenging. This is because values during the incremental refinement stage can flow across different vertices regardless of whether their intermediate states are tracked or not. Since we aim to guarantee BSP semantics (similar to systems like [17]), computation of vertex values cannot be deferred as they need to happen in an iteration-by-iteration manner.

We develop a *selective incremental processing* technique that operates on selective intermediate states, i.e., where a selected subset of vertices are tracked. Our technique effectively separates out the interactions between tracked and untracked vertices so that right values get propagated across the edges depending on whether their source and target vertices are tracked or untracked. We first summarize how the graph layout can be optimized when selective intermediate states are maintained, and then discuss the details of our selective incremental processing.



(a) Optimized graph layout. Vertices 1, 2 and 7 are tracked (highlighted), and remaining vertices are not tracked. Each vertex maintains two adjacency lists for outgoing edges: one for tracked neighbors and other for untracked neighbors.



(b) Values propagated based on whether the target vertex is tracked (highlighted) or untracked. Untracked vertices receive the old and new values, whereas tracked vertices directly receive difference in values.

Figure 6

Optimizing Graph Layout. Streaming graph processing systems use efficient dynamic graph representations to handle rapid graph mutation and enable efficient parallel operations on active edges and vertices [6, 7, 15, 17]. Since our selective incremental processing handles the interactions for tracked vertices in a different manner compared to the interactions for untracked vertices, the graph layouts can be improved to avoid expensive checks while propagating values during the refinement process. Specifically, the edges between tracked vertices and untracked vertices can be separated out in the graph layout itself; by doing so, all computations on edges whose target vertices are either tracked or untracked can be performed directly in form of parallel operations without verifying whether every individual target vertex is tracked or untracked.

Since we incorporate our technique in GraphBolt, which uses adjacency lists to hold graph snapshots, such a separation results in the graph layout shown in Figure 6a. Here, each vertex now holds two vectors for its outgoing neighbors: the first vector contains edges whose targets are tracked vertices, and the second vector contains edges whose targets are untracked vertices.

Propagating Differences upon Graph Mutation. When the graph structure mutates, the incremental computation must correctly propagate changes throughout the available intermediate states to compute final results. To retain BSP guarantees at the end of each iteration, our selective incremental processing propagates values iteration-by-iteration.

With only a subset of intermediate states available, processing for different vertices is handled differently. For tracked vertices, the intermediate states are incrementally refined in-place as processing progresses through iterations. Whereas for untracked vertices, a single vector is maintained to hold their latest values as processing progresses through iterations (similar to computing from scratch without intermediate states). In each iteration, the values propagated across edges are based on whether the target vertices are tracked or untracked, as shown in Figure 6b. If the target vertex is tracked, then the difference in value is propagated along its incoming edge similar to the traditional dependency-driven incremental processing [17]. On the other hand, if the target vertex is untracked, then both the old value (from before graph mutation) and the new value (resulting from graph mutation) are propagated along the edge. This allows the target vertex to compute the necessary differences for its outgoing neighbors in the subsequent iteration.

Algorithm 1 shows how our selective incremental processing propagates values upon graph mutation. In each iteration, the differences directly resulting from graph mutation are propagated (lines 6-13), and then the resulting differences are propagated for the tracked and untracked vertices (line 15-22). The tracked vertices acquire the difference between the previous value change and the new value change. The untracked vertices, on the other hand, receive two values: the previous change and the new change. This allows the untracked vertices to recompute the old aggregation along with the updated aggregation. Once the values arrive at the required active vertices, their vertex values are computed to identify the differences to be propagated in the next iteration (lines 24-33).

As we can see on lines 15 and 19, operations on edges based on their target being tracked or untracked are directly invoked in parallel without any checks per edge, mainly because of the optimized graph layout described above that separates the edges. Moreover, since the selective incremental processing recomputes the vertex values to identify differences, our model tracks only the aggregation values as intermediate states (i.e., it does not track the intermediate vertex values, which is also maintained as part of intermediate state in the traditional dependency-driven incremental refinement [17]).

4 Minimal Stateful Iterative Model

In this model, we aggressively eliminate the tracking of intermediate state by specializing the incremental processing for certain graph algorithms. Specifically, our model will directly operate on value differences without reconstructing the intermediate states so that effects of mutations get propagated only as value differences throughout the iterations.

We first formalize the *distributive update property* that enables this specialization, and then discuss the details of the minimal stateful iterative model.

4.1 Distributive Update Property

Computations in graph algorithms can be modelled as:

$$val(v) = A\Big(\bigoplus_{(u,v)\,\in\,E}\big(S(val(u))\big)\Big)$$

Algorithm 1 Selective Incremental Processing

```
1: par for $e \in E$ s.t. target($e$) is untracked do
2:   Activate source($e$) for propagation to untracked targets
3: end par for

4: for $i \in [1...]$ do
5:   /* Propagate changes directly resulting from mutations */
6:   par for $e \in$ mutated edges s.t. target($e$) is tracked do
7:     Propagate old change if $e$ is added; otherwise retract old change
8:     Activate target($e$) for vertex computation
9:   end par for
10:  par for $e \in E$ mutated edges s.t. target($e$) is untracked do
11:    Propagate old change if $e$ is removed; otherwise retract old change
12:    Activate target($e$) for vertex computation
13:  end par for

14:  /* Propagate transitive changes from active vertices */
15:  par for $e \in E$ s.t. target($e$) is tracked and (source($e$) is active or $e$ is mutated edge) do
16:    Propagate difference between old change and new change
17:    Activate target($e$) for vertex computation
18:  end par for
19:  par for $e \in E$ s.t. target($e$) is untracked and source($e$) is active do
20:    Propagate old change and new change
21:    Activate target($e$) for vertex computation
22:  end par for

23:  /* Compute vertex values and differences to push in next iter */
24:  par for $v \in$ active tracked vertices do
25:    Merge difference in $v$'s intermediate state
26:    Compute $v$'s old value and new value
27:    Activate $v$ for propagation if difference in value changes is not $\varnothing$
28:  end par for
29:  par for $v \in$ active untracked vertices do
30:    Merge old change in $v$'s old value and new change in $v$'s new value
31:    Compute $v$'s old value and new value
32:    Activate $v$ for propagation if difference in value changes is not $\varnothing$
33:  end par for
34: end for
```

where $\oplus$ is the aggregation function that combines incoming values to a vertex, S is the function that transforms the source's value to be aggregated (analogous to scatter operation in [8]), and A is the vertex function that computes the vertex value using the aggregation result. For instance, in PageRank $\oplus$ is the *sum* operation, S is the function that divides the rank value with outdegree (i.e., `pr(u) / out_degree(u)`), and A is the linear equation that computes rank value using the result of $\oplus$ and damping factor (i.e., `(1-d) + d * sum`).

The *distributive update property* states that the computation of vertex value can be distributed as sub-computations over its incoming neighbors' values, i.e.,

$$A\Big(\bigoplus_{k\in\{w,x,y,z\}}\big(S(k)\big)\Big) = \underset{k\in\{\{w,x\},\{y,z\}\}}{\gamma}\Big(\alpha\Big(\bigoplus_{k'\in k}\big(S(k')\big)\Big)\Big)$$

where γ and α are functions derived from A. This property is important because it allows directly computing the difference in the target value from the difference in the source value without reconstructing $\oplus\big(S(*)\big)$. This allows our minimal stateful iterative model to aggressively reduce the intermediate state by simply not tracking the results from $\oplus$. For instance, in our PageRank example if a source's rank value changes from u_1 to u_2, then the change in value of the destination vertex v gets directly computed as `d * (u`$_2$ `- u`$_1$`) / out_degree(u)`. Note this does not require explicitly reconstructing the `sum` variable for v.

Graph Algorithm	Distributive Update Property	Reason for Violation
PageRank	✓	-
Co-Training Expectation Maximization	✓	-
Katz Centrality	✓	-
Path-based / Monotonic Algorithms like: Breath First Search, Shortest Paths, Connected Components, Widest Paths, Minimal Spanning Tree	✓	-
Collaborative Filtering	✗	Matrix Inverse & Multiplication
Circuit Simulation	✗	Division
Label Propagation	✗	Value Normalization
Multi-Manifold Ranking	✗	
Multi-Modality Learning	✗	

Table 1: Algorithms whose computations satisfy (✓) or violate (✗) the distributive update property.

Applicability. The distributive update property is different from just the aggregation operation being distributive that is studied in static graph processing [39]. While most of the aggregation operations are distributive (which enables edge parallel incremental operations, as well-known in prior research), the distributive update property also requires the vertex functions to be distributive. For instance, the PageRank computation satisfies this property since its linear equation only operates on the aggregation value and constants, and hence, any change in rank value of source vertex can be directly incorporated in the destination value. On the other hand, even though algorithms like Collaborative Filtering [40] and Multi-Modality Learning [31] have *sum* aggregation (which is distributive), their vertex functions are not distributive since they involve operations like *normalization* and *matrix inverse*. Table 1 classifies various graph algorithms based on whether they satisfy or violate the distributive update property.

4.2 Tracking Minimal Vertex State

While the differences can be propagated without computing the intermediate states at each iteration, these differences need to be grounded w.r.t. some basis so that they are meaningful. Hence, we track the earliest intermediate state that initiates the incremental computation when graph structure mutates. These earliest intermediate states correspond to the states of the mutation points (e.g., vertices whose edges got mutated)

Algorithm 2 Incremental Processing with Minimal State

```
 1: for $i \in [1...]$ do
 2:   /* Propagate old values */
 3:   par for $e \in$ mutated edges do
 4:     Propagate old value if $e$ is added; otherwise retract old value
 5:     Activate target($e$) for vertex computation
 6:   end par for

 7:   /* Propagate transitive changes from active vertices */
 8:   par for $e \in E$ s.t. source($e$) is active or $e$ is mutated edge do
 9:     Propagate change
10:   end par for
11:   par for $v \in$ active vertices do
12:     if $v$ is tracked then
13:       Merge change in $v$'s intermediate state
14:     end if
15:     Activate $v$ for propagation if change is not $\varnothing$
16:   end par for
17: end for

18: /* Compute final vertex values */
19: par for $v \in V$ do
20:   Merge change in vertex value with old vertex value
21: end par for
```

since those points start propagating the changes directly based on the specific edge/vertex that gets added/deleted. Apart from the earliest states, no other intermediate state is captured because the computations purely operate on differences to propagate through the rest of the iterations.

Hence, the potential mutation points in the graph are the subset of vertices whose intermediate states are tracked. These vertices are identified based on application-specific insights like mutations occurring at certain important vertices, or what-if queries based on certain regions of the graph. When potential mutation points cannot be identified a priori, high in-degree vertices can be tracked (similar to Section 3.2) in order to increase the chances of mutations getting applied to tracked vertices. As we see next, the incremental processing can automatically handle the case when mutations occur on vertices that are not tracked as well.

4.3 Incremental Processing

With the distributive update property, the impact of mutations on tracked vertices can be computed directly using difference in values. When graph structure mutates, incremental computation is performed in iteration-by-iteration manner by purely operating on differences. Algorithm 2 shows how the differences are identified and propagated. Unlike the selective incremental processing technique, the distributive update property enables straightforward propagation of differences. The mutated edges propagate and retract the old values (lines 3-6), and their target vertices compute the differences. These differences are further propagated in subsequent iterations (line 9). If the target vertex is tracked, the differences are merged in the intermediate state as computation progresses. In the end, the cumulative differences are incorporated with the vertex values to generate the final result (line 20).

When mutation occurs on vertices that are not tracked, the processing can dynamically switch to selective incremental processing (Section 3.3) that correctly propagates value differences based on whether the target vertices are tracked or untracked. This switch happens seamlessly without performing additional computation since the selective incremental processing reconstructs the missing intermediate states on-the-fly as it progresses iteration-by-iteration.

5 Evaluation

We thoroughly evaluate our memory-efficient stateful iterative models and compare their performance with the dependency-driven incremental processing model from GraphBolt [17] (which delivers high performance at the cost of high memory consumption). Specifically, we answer the following questions:

1. How effective is our selective stateful iterative model in controlling the memory footprint?
2. How does the performance of our selective stateful iterative model vary as the number of vertices being tracked changes?
3. How effective is our minimal stateful iterative model in maintaining a small memory footprint while still delivering high performance?
4. How do our memory-efficient stateful iterative models perform when processing a large number of simultaneous graph mutations?

5.1 Implementation Details

We implemented our memory-efficient stateful iterative models in the GraphBolt system for two main reasons: first, it allows our models to utilize the efficient implementation of the underlying framework (e.g., parallelization strategy, atomics, frontiers, etc.); and second, it enables direct performance comparison of our models with GraphBolt's existing execution model.

We implemented the optimized graph layout for the adjacency list representation (discussed in Section 3.3) to replace the existing adjacency list data structure. The intermediate states for selected subset of vertices are tracked in their respective arrays. The state arrays get allocated vertically (per vertex) instead of horizontally (per iteration) so that arrays for untracked vertices are not allocated (hence reducing memory footprint), while at the same time the intermediate states for tracked vertices get addressed without using any hashmap.

5.2 Experimental Setup

We use seven synchronous graph algorithms. PageRank (PR) [23] computes the importance of web-pages based on incoming links to those pages. Collaborative Filtering (CF) [40] is a context-based technique used in recommender systems

Graph	Vertices	Edges	Graph Size	
			Without Final State	With Final State
TwitterMPI (TT)	52.6M	2B	14.2GB	19-30GB
Friendster (FT)	68.3M	2.5B	18.74GB	24-38GB
UK-2007-05 (UK)	105M	3.7B	27.75GB	36-59GB
UK-union (UN)	133M	5.5B	36.2GB	48-80GB
Clueweb (CWB)	978.4M	42.5B	130GB	197-240GB

Table 2: Real-world graphs used in experiments [1, 2, 3]

to classify associated items while Co-Training Expectation Maximization (CoEM) [22] is a semi-supervised learning algorithm for named object identification. Multi-Manifold Ranking (MMR) [37] is a ranking method that uses multiple image manifolds each constructed using a different image features. Multi-Modality Learning (MML) [31] and Label Propagation (LP) [41] are learning algorithms that disperse labels from a subset of vertices to assign label to the rest of the graph. Circuit Simulation (CS) [12] simulates flow in a circuit by solving partial differential equation.

The LP, MMR and MML algorithms compute vector of features for each vertex, whereas the remaining algorithms operate on scalar vertex values except for CF which operates on two factors per vertex. Computations in PageRank and CoEM follow the distributive update property (described in Section 4.1), and hence we evaluate our minimal stateful iterative model with these two benchmarks. Computations in the remaining benchmark do not follow the distributive update due to the complex sub-operations they involve: specifically, LP, MMR and MML normalize the feature vectors in every iteration, CF computes matrix inverse, and CS uses division on its aggregation values.

Table 2 lists the six real-world input graphs used for evaluation. Similar to [17, 28], we obtained an initial fixed point when 50% of edges were loaded, and streamed in the remaining edges to model edge insertions, while randomly sampled edges from the loaded graph were used for edge deletions. To eliminate the effects of locality, we shuffled the edges while forming our edge streams. Starting with 50% of edges is not a requirement for GraphBolt and other systems (i.e., one can start with an empty graph as well), but doing so allows us to evaluate the common scenario where a graph snapshot is already present and mutations are streamed in. For CWB graph which is significantly large, initial fixed point was obtained with 7% of edges so that at least stateless execution could successfully execute. The algorithms that operate on vectors consume high memory, and as expected, the memory footprint increases as the vector size increases. Unless otherwise stated, we use 10 features for LP, MMR and MML so that the GraphBolt baseline could hold the intermediate state without running out of memory, and 2 features are used for CWB graph to ensure that stateless execution could successfully execute. Similar to [17], we run all algorithms for 10 iterations. Unless otherwise stated, we apply 10K edge mutations to evaluate how quickly our models compute the final result; we also vary the mutation batch size from a single mutation to up 10 million edge mutations.

All experiments were performed on Oracle Cloud VM.Standard2.24 shape containing Intel(R) Xeon(R) 8167M processor with 24 physical cores (48 threads) and 320GB main memory running 64-bit Ubuntu 18.04.

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

- **Selective-k%:** this is our selective stateful iterative model that tracks k% of total vertices.
- **Minimal:** this is our minimal stateful iterative model.
- **GraphBolt:** this is GraphBolt's execution [16, 17], which tracks the intermediate state for all vertices.
- **Stateless:** this baseline does not track any intermediate state, and recomputes values from scratch upon graph mutation (same as GB-Reset in [17]).

5.3 Selective Stateful Model Performance

Figure 7 shows the memory footprint and execution time for our selective stateful iterative model when tracking the intermediate state for 20%, 40%, 60% and 80% of the vertices. The figure also compares the performance with GraphBolt and stateless executions. Figure 8 summarizes similar results for CWB graph. As we can see, our selective stateful iterative model is effective in controlling the memory footprint by tracking only the selected subset of vertices. For instance, it consumes 35-70% less memory than GraphBolt when tracking only 20% of vertices, while at the same time delivering 15-83% of the performance gains provided by GraphBolt over the stateless execution. In fact, GraphBolt runs out of memory for certain cases while the selective executions end up successfully executing and delivering high performance.

By tracking more intermediate states the memory footprint increases and the execution time decreases mainly because the intermediate state helps in incremental refinement; this is visible as decreasing number of edge operations in Figure 9 for FT graph as the number of tracked vertices increases from 20% to 80%. Since stateless execution does not track any intermediate state, its memory footprint is the lowest while the execution time is highest; on the other extreme, since GraphBolt tracks intermediate state for all vertices, its memory footprint is the highest while its execution time is the lowest. Our selective stateful iterative model trades off memory footprint for more computation, and hence delivers performance between the two extremes.

We observe that the increase in memory footprint from stateless to selective-20% is higher compared to the increase between consecutive selective variants (e.g., between selective-20% and selective-40%). This is because the selective incremental processing computes both the new value (after mutation) and the old value (prior to mutation) so that differences can be propagated from untracked values; holding

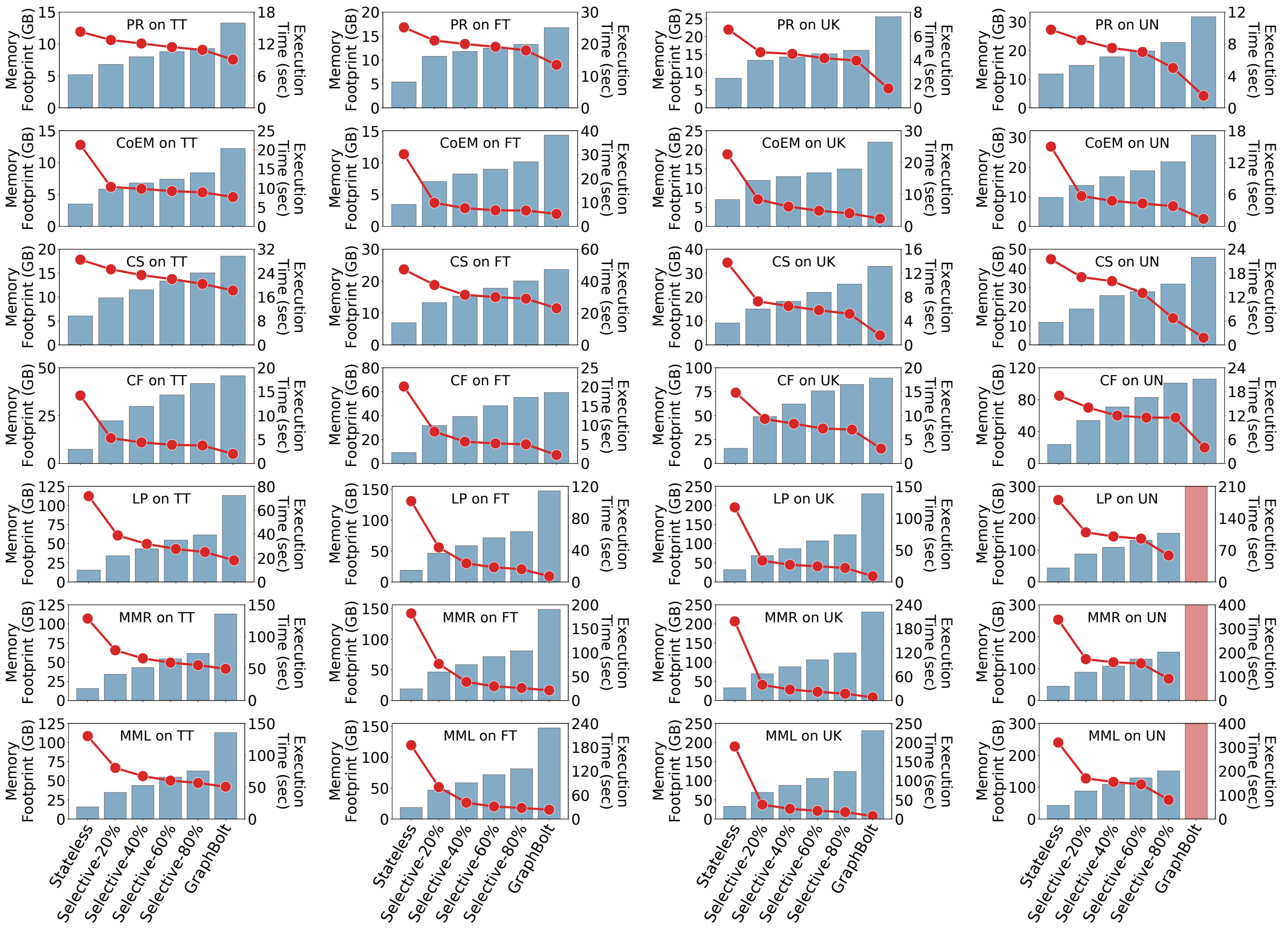


Figure 7: Performance of our selective stateful iterative model compared to the stateless iterative model and the stateful iterative model from GraphBolt. The memory footprints (in GB) are shown as bars (left y-axis) and the execution times (in seconds) are shown as points (right y-axis). Red bar indicates the execution ran out of memory.

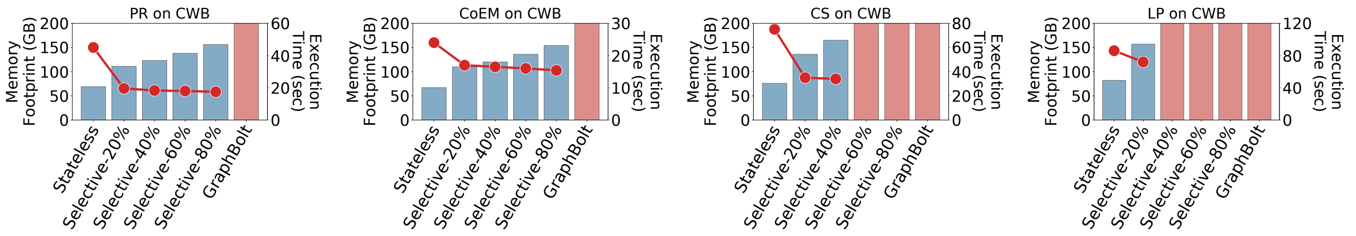


Figure 8: Performance of our selective stateful iterative model on CWB graph compared to the stateless iterative model and the stateful iterative model from GraphBolt. The memory footprints (in GB) are shown as bars (left y-axis) and the execution times (in seconds) are shown as points (right y-axis). Red bar indicates the execution ran out of memory.

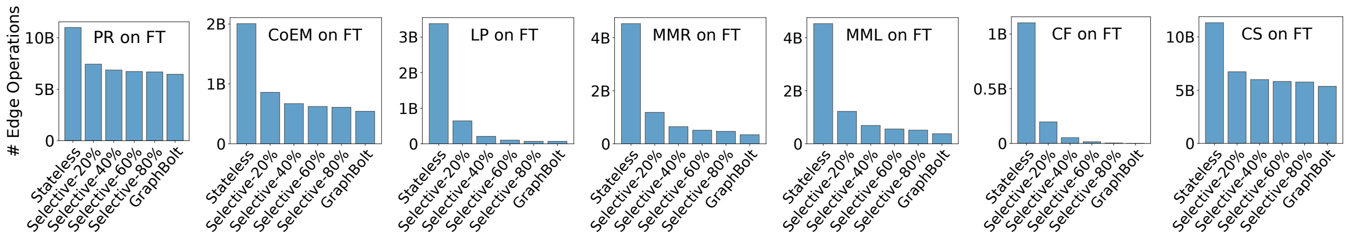


Figure 9: Number of edges operations executed by our selective stateless iterative model, compared to the stateless iterative model and the stateful iterative model from GraphBolt.

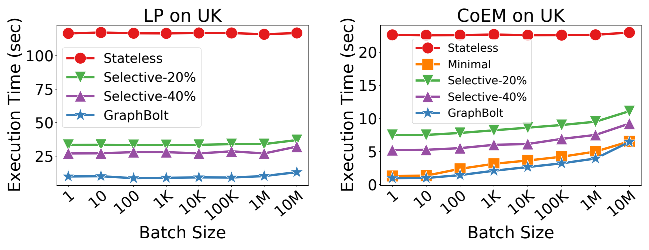


Figure 10: Execution times (in seconds) for different iterative models across varying number of mutations per batch.

these values in memory adds to the memory footprint, which is an overhead that stateless executions do not incur.

For a given graph, the memory footprint depends on the size of intermediate state in the benchmark. Hence, for each graph the memory footprints are lower for PageRank and CoEM since they operate on scalar values, while the footprints are higher for LP, MMR and MML due to their use of feature vectors. This is also the reason why GraphBolt runs out of memory for only LP, MMR and MML on UN graph; whereas on CWB graph only certain executions of selective variants run successfully.

The execution time, on the other hand, is dependent on how the values propagate across iterations which is dependent on the graph algorithm and the structure of input graph. Therefore, the performance benefit with saving selective intermediate state is different for different cases. For instance, selective-20% is 5× faster compared to stateless for MMR on UK, whereas it is only 1.63× faster for MMR on TT. On the other hand, selective-20% is 2.4× faster compared to stateless for CF on FT, but only 1.2× faster for CF on UN.

For most of the cases, we observe that tracking intermediate state for only 20% of vertices achieves 15-83% of performance gains that are achieved by GraphBolt. For instance selective-20% on MMR achieves 1.6-5× compared to stateless executions across different graphs, whereas GraphBolt achieves 2.6-24.4× compared to stateless execution. This is mainly because our model tracks the high-degree vertices that demand more computation if their states are not available, and hence incremental processing for those high-degree vertices ends up achieving high performance benefits. This is a benefit especially for skewed graphs since the memory consumption for selective-20% is much less compared to GraphBolt, while at the same time the performance gains are high. As expected, the performance gains from incremental processing reduce as the number of tracked vertices increases.

Finally, for cases like CF we observe that the difference in execution times between selective-80% and GraphBolt is higher compared to the difference between consecutive selective variants (e.g., between selective-60% and selective-80%). This is again because the selective incremental processing computes both the old values and the new values whereas GraphBolt directly computes the value changes. The effects of these additional computations become more visible for CF since it has relatively expensive vertex computations (involving a matrix inverse operation).

Scaling with Mutation Batch Sizes. Figure 10 shows the performance of our selective stateful iterative model with 20% and 40% tracked vertices as the mutation batch size increases from a single mutation to up 10 million edge mutations. The memory footprints of the stateful iterative models remain same as in Figure 7 since they are mainly dependent on the number of tracked vertices. However, we observe that the amount of computation performed increases as more mutations get simultaneously applied. Hence, the execution time for both, selective stateful model as well as GraphBolt increases as mutation batch size increases. Since the stateless execution simply recomputes from scratch, its execution time increases very slowly across different mutation batch sizes.

5.4 Minimal Stateful Model Performance

Figure 11 shows the memory footprint and execution times for our minimal stateful iterative model along with GraphBolt and stateless executions on PageRank and CoEM. Since the minimal stateful iterative model only tracks the earliest intermediate states that form the basis for differences, its memory footprint is 1.4-2.4× smaller compared to GraphBolt and only 1.1-1.3× higher than stateless execution. Moreover, the incremental computation in the minimal stateful iterative model purely operates on value differences, and hence, delivers high performance. Our minimal stateful iterative model is 1.1-8.2× faster than stateless executions, which results in 65-90% of the benefits delivered by GraphBolt.

Scaling with Mutation Batch Sizes. Figure 10 shows the performance of the minimal stateful iterative model for CoEM as the mutation batch size increases from a single mutation to up 10 million edge mutations. Similar to selective model, the execution time for the minimal stateful iterative model increases as the number of simultaneous mutations increases; nevertheless, it is 3.5-17.4× faster than the stateless execution.

Unlike the selective model, the memory footprint of our minimal stateful iterative model is sensitive to the number of updates. As the mutation batch size increased from 1 to 10M, the memory footprint of our minimal stateful iterative model increased from 0.5GB (1.06× higher than stateless, and 2.9× lower than GraphBolt) to 2GB (1.3× higher than stateless, and 2.4× lower than GraphBolt).

6 Related Work

Several dynamic graph processing systems have been developed in literature. We first discuss the systems with stateful iterative models, and then summarize the works that use stateless models.

Kickstarter [35], GraphBolt [17] and DZiG [16] develop efficient streaming graph processing solutions using stateful iterative models for incremental computation. Upon graph

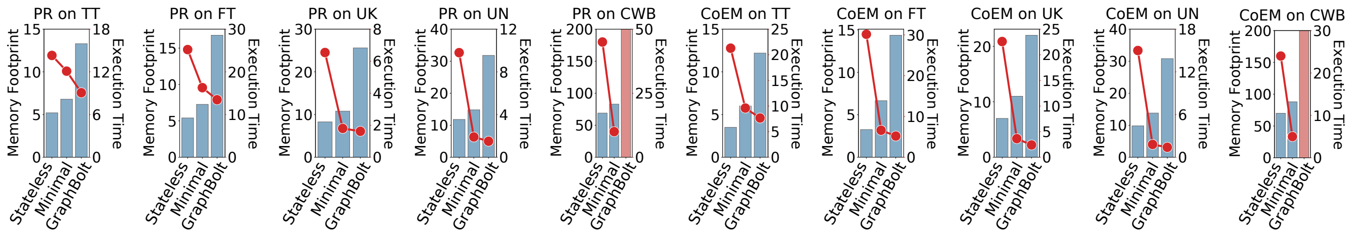


Figure 11: Performance of our minimal stateful iterative model compared to the stateless iterative model and the stateful iterative model form GraphBolt. The memory footprints (in GB) are shown as bars (left y-axis) and the execution times (in seconds) are shown as points (right y-axis). Red bar indicates the execution ran out of memory.

mutation, these systems use the intermediate state to quickly adjust the computed values and deliver final results corresponding to the latest graph version. KickStarter [35] focuses on graph algorithms like BFS and SSSP that use monotonic functions. Its runtime exploits the monotonic relationship between vertex values to capture dependencies between the latest computed values, resulting in only one intermediate state per vertex. Hence, its memory footprint does not drastically increase compared to stateless execution of those algorithms as the input graph consumes the most amount of memory. GraphBolt [17] and DZiG [16] focus on the broader class of graph algorithms that run in BSP manner. They capture the dependency information across intermediate vertex values as computation progresses, and not just across the latest values. DZiG [16] develops a DelZero-Aware incremental refinement strategy to efficiently handle computation sparsity during incremental processing. The intermediate state in both, GraphBolt and DZiG, requires much larger amount of memory compared to KickStarter, which drastically increases their memory footprint. Our memory-efficient stateful iterative models limit the memory footprint by reducing the amount of intermediate state that gets captured for efficient incremental computation.

GraphInc [4] saves all messages across edges along with computed states in order to replay the computation with incremental changes. This ends up demanding much larger memory capacity (intermediate state in the order of edges for each iteration) compared to recent works like GraphBolt. Differential Dataflow [18] is a general-purpose system that enables incremental computation by tracking states at operator level, i.e., for graph computations it also maintains intermediate state in the order of edges per iteration, which results in a much larger memory footprint than KickStarter and GraphBolt. Since our memory-efficient stateful iterative models prune out intermediate state at vertex-level, they can be easily applied to these systems by selectively tracking intermediate edge-level or operator-level states for only a subset of vertices.

Systems like [5, 6, 7, 13, 15, 27, 28, 30] use stateless iterative models which limits their efficiency in delivering accurate results upon graph mutation. Tornado [28], Kineograph [5] and GraphIn [27] perform incremental computation by triggering the user functions based on graph updates and allowing the changes to propagate throughout the graph. Hence, they cannot guarantee accurate results for BSP algorithms. GraphIn identifies the vertices that could be potentially impacted by graph updates using tag propagation, and restarts computation from scratch for those identified vertices. [30] uses GIM-V (generalized iterative matrix vector multiplication) to perform incremental computation. LLAMA [15], STINGER [7], Aspen [6], GraphOne [13] and LiveGraph [43] focus on designing efficient dynamic graph data structures and storage systems, and their processing models do not support incremental computation. However, incremental computation is crucial in delivering high end-to-end performance, as shown by detailed performance comparison of these systems in [16].

Solutions like [10, 11, 19, 34] operate on evolving graphs that contain a group of temporally-related graph snapshots capturing the evolution of the graph structure over time. These systems do not capture intermediate states and perform incremental computation by directly reusing the results computed for previous graph snapshots, making them suitable for self-fixing graph algorithms but not for general BSP algorithms.

Finally, static graph processing systems [8, 9, 14, 20, 21, 24, 25, 26, 29, 33, 36, 38, 42] focus on processing a given static graph input and do not natively handle dynamic graphs with incremental processing.

7 Conclusion

We presented two memory-efficient stateful iterative models for incremental processing of streaming graphs. The Selective Stateful Iterative Model controls the memory footprint by selecting a small portion of the intermediate state to be maintained throughout execution. The Minimal Stateful Iterative Model further reduces the memory footprint by exploiting the distributive update property in graph algorithms. Our results showed that our models are effective in limiting the memory footprint while still retaining most of the performance benefits of traditional stateful iterative models, hence being able to scale on larger graphs that could not be handled by the traditional models.

8 Acknowledgements

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

References

[1] Paolo Boldi, Marco Rosa, Massimo Santini, and Sebastiano Vigna. Layered Label Propagation: A MultiResolution Coordinate-Free Ordering for Compressing Social Networks. In Sadagopan Srinivasan, Krithi Ramamritham, Arun Kumar, M. P. Ravindra, Elisa Bertino, and Ravi Kumar, editors, *Proceedings of the International Conference on World Wide Web (WWW '11)*, pages 587–596, 2011.

[2] Paolo Boldi, Massimo Santini, and Sebastiano Vigna. A Large Time-Aware Web Graph. In *Special Interest Group on Information Retrieval (SIGIR '08)*, pages 33–38, 2008.

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

[4] 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.

[5] 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.

[6] 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)*, pages 918–934, 2019.

[7] 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.

[8] 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.

[9] 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.

[10] 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 1–14, 2014.

[11] 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)*, pages 1–6, 2016.

[12] Farzad Khorasani, Keval Vora, Rajiv Gupta, and Laxmi N Bhuyan. CuSha: Vertex-Centric Graph Processing on GPUs. In *Proceedings of the International Symposium on High-Performance Parallel and Distributed Computing (HPDC '14)*, pages 239–252, 2014.

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

[14] 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)*, pages 706–716, 2018.

[15] 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, 2015.

[16] Mugilan Mariappan, Joanna Che, and Keval Vora. DZiG: Sparsity-Aware Incremental Processing of Streaming Graphs. In *Proceedings of the European Conference on Computer Systems (EuroSys '21)*, pages 1–16, 2021.

[17] 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 1–16, 2019.

[18] Frank McSherry, Derek Gordon Murray, Rebecca Isaacs, and Michael Isard. Differential Dataflow. In *Conference on Innovative Data Systems Research (CIDR '13)*, 2013.

[19] 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 '15)*, 11(3):1–34, 2015.

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

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

[22] 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.

[23] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The PageRank Citation Ranking: Bringing Order to the Web. Technical report, Stanford InfoLab, 1999.

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

[25] Amitabha Roy, Ivo Mihailovic, and Willy Zwaenepoel. X-Stream: Edge-Centric Graph Processing Using Streaming Partitions. In *Proceedings of the Symposium on Operating Systems Principles (SOSP '13)*, page 472–488, 2013.

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

[27] 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 *European Conference on Parallel Processing (Euro-Par '16)*, pages 319–333, 2016.

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

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

[30] 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.

[31] Hanghang Tong, Jingrui He, Mingjing Li, Changshui Zhang, and Wei-Ying Ma. Graph based multi-modality learning. In *Proceedings of the International Conference on Multimedia (MULTIMEDIA '05)*, pages 862–871, 2005.

[32] Leslie G Valiant. A Bridging Model for Parallel Computation. *Communications of the ACM*, pages 103–111, 1990.

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

[34] Keval Vora, Rajiv Gupta, and Guoqing Xu. Synergistic Analysis of Evolving Graphs. *ACM Transactions on Architecture and Code Optimization (TACO '16)*, pages 1–27, 2016.

[35] 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.

[36] 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 International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA '14)*, page 861–878, 2014.

[37] Yang Wang, Muhammad Aamir Cheema, Xuemin Lin, and Qing Zhang. Multi-Manifold Ranking: Using Multiple Features for Better Image Retrieval. In *Pacific-Asia Conference on Knowledge Discovery and Data Mining (PAKDD '13)*, pages 449–460, 2013.

[38] 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 Symposium on Cloud Computing (SoCC '15)*, pages 408–421, 2015.

[39] Yanfeng Zhang, Qixin Gao, Lixin Gao, and Cuirong Wang. Maiter: An Asynchronous Graph Processing Framework for Delta-based Accumulative Iterative Computation. *IEEE Transactions on Parallel and Distributed Systems (TPDS '13)*, 25(8):2091–2100, 2013.

[40] 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.

[41] Xiaojin Zhu and Zoubin Ghahramani. Learning From Labeled and Unlabeled Data with Label Propagation. *Tech. Rep., Technical Report CMU-CALD-02–107, Carnegie Mellon University*, 2002.

[42] 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.

[43] 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)*, page 1020–1034, 2020.