Synergistic Analysis of Evolving Graphs

KEVAL VORA and RAJIV GUPTA, University of California, Riverside
GUOQING XU, University of California, Irvine

Evolving graph processing involves repeating analyses, which are often iterative, over multiple snapshots of the graph corresponding to different points in time. Since the snapshots of an evolving graph share a great number of vertices and edges, traditional approaches that process these snapshots one at a time without exploiting this overlap contain much wasted effort on both data loading and computation, making them extremely inefficient. In this article, we identify major sources of inefficiencies and present two optimization techniques to address them. First, we propose a technique for *amortizing the fetch cost* by merging fetching of values for different snapshots of the same vertex. Second, we propose a technique for *amortizing the processing cost* by feeding values computed by earlier snapshots into later snapshots. We have implemented these optimizations in two distributed graph processing systems, namely, GraphLab and ASPIRE. Our experiments with multiple real evolving graphs and algorithms show that, on average fetch amortization speeds up execution of GraphLab and ASPIRE by 5.2× and 4.1×, respectively. Amortizing the processing cost yields additional average speedups of 2× and 7.9×, respectively.

CCS Concepts: • **Computing methodologies** → **Distributed computing methodologies**; • **Information systems** → *Information systems applications*

Additional Key Words and Phrases: Graph processing, temporal graphs, message aggregation

ACM Reference Format:
Keval Vora, Rajiv Gupta, and Guoqing Xu. 2016. Synergistic analysis of evolving graphs. ACM Trans. Archit. Code Optim. 13, 4, Article 32 (October 2016), 27 pages.
DOI: http://dx.doi.org/10.1145/2992784

1. INTRODUCTION

The importance of graph processing has grown due to the wide ranging use of graph mining and graph analytics. An important feature of real-world graphs is that they are constantly evolving (e.g., social networks, networks modeling the spreading of diseases, etc.) [Rossi et al. 2013; Watts and Strogatz 1998]. Such evolving graphs are useful to capture dynamic properties and interesting trends that change over time. For example, researchers in Leskovec et al. [2005] study how various structural characteristics like graph density and graph diameter evolve over time by analyzing multiple graph snapshots captured at different points in time. In Wilson et al. [2009], researchers analyze evolving graphs to study how user interactions and activities change over time. Techniques in Yang et al. [2007] operate on historical web graph

This work is supported in part by NSF grants CCF-1524852 and CCF-1318103 to UC Riverside, CNS-1321179, CCF-1409829, and CNS-1613023 to UC Irvine, and by ONR grants N00014-14-1-0549 and N00014-16-1-2913 to UC Irvine.
Authors' addresses: K. Vora and R. Gupta, Department of Computer Science & Engineering, University of California Riverside, Winston Chung Hall, Riverside, CA 92521; emails: {kvora001, gupta}@cs.ucr.edu; G. Xu, Computer Science Department, University of California Irvine, Donald Bren Hall, Irvine, CA 92617; email: guoqingx@ics.uci.edu.

DOI: http://dx.doi.org/10.1145/2992784

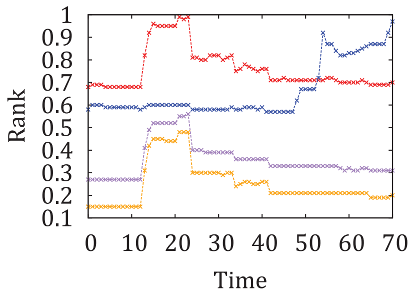


Fig. 1. Trend analysis of PageRanks in an evolving graph.

snapshots to model the dynamic nature of link analysis. Also, in Radicchi et al. [2009], researchers rank authors and papers in different graph snapshots to study how an author's influence changes in time.

Therefore, there is a great deal of interest in carrying out graph analytics tasks over evolving graphs, that is, repeatedly computing the results of analysis at various points in time to study how different characteristics of vertices (e.g., their PageRank, shortest path, widest path, community, etc.) change over time.

For example, Figure 1 shows the trend of PageRank values for different vertices in an evolving graph; by studying such a dynamic parameter, analysts can observe that popularity of the three web pages (red, purple, and yellow) that follow similar trends in their ranks are influenced by similar set of topics.

The analysis of an evolving graph is expressed as the *repeating* of graph analysis over multiple snapshots of a changing graph—different snapshots are analyzed independently of each other and their results are finally aggregated. Note that evolving graph processing is different from streaming graph processing [Ediger et al. 2012] where iterative processing continues over a dynamic graph structure that keeps on changing, hence terminating to the final solution for the most updated graph.

Due to the fast-changing nature of a modern evolving graph, the graph often has a large number of snapshots; analyzing one snapshot at a time can be extremely slow even when done in parallel, especially when these snapshots are large graphs themselves. For instance, one single snapshot of the Twitter graph—an input graph used in our experiment—has over 1 billion edges, and there are in all 25.5 billion edges in all its snapshots we analyzed.

In this article, we develop temporal execution techniques that significantly improve the performance of evolving graph analysis, based on an important observation that different snapshots of a graph often have large overlap of vertices and edges. By laying out the evolving graph in a manner such that this temporal overlap is exposed, we identify two key optimizations that aid the overall processing: first, we reorder the computations based on loop transformation techniques to amortize the cost of fetch across multiple snapshots while processing the evolving graphs; and second, we enable feeding of values computed by earlier snapshots into later snapshots to amortize the cost of processing vertices across multiple snapshots. Furthermore, the two optimizations are orthogonal, that is, they amortize different costs, and hence, we identify and exploit the synergy between them by allowing feeding of values from all vertices, including those that have not attained their final values, to amortize the processing cost, while simultaneously reordering computations to amortize the fetch cost.

There exists a body of work on speeding up mining of evolving graphs [Ren et al. 2011; Kan et al. 2009; Desikan et al. 2005; Sun et al. 2007]. While Chronos [Han et al. 2014] introduces a novel memory layout for evolving graphs to improve cache locality, none of these works amortize fetch and processing costs on clusters as we do; our

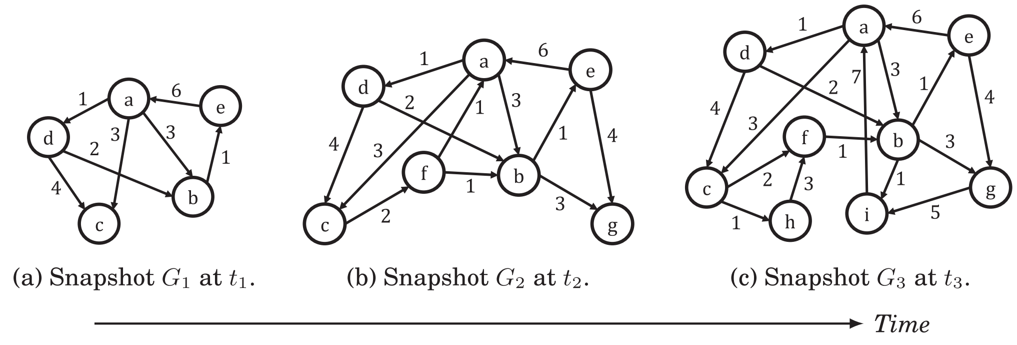

(a) Snapshot G_1 at t_1.
(b) Snapshot G_2 at t_2.
(c) Snapshot G_3 at t_3.
Time

Fig. 2. Example evolving graph $\mathcal{G} = \langle G_1, G_2, G_3 \rangle$.

techniques are complimentary and applicable to a wide variety of problems. Moreover, our optimizations are general and can be plugged into a variety of distributed or shared-memory graph processing systems.

In this article, we have experimented extensively with the implementations of these optimizations in GraphLab [Low et al. 2012] (a distributed graph-based machine learning framework) and ASPIRE (a Distributed Shared Memory (DSM) -based graph processing system [Vora et al. 2014]). Our experiments with multiple real evolving graphs and graph processing algorithms on a 16-node cluster demonstrate that, on average, fetch amortization speeds up the execution of GraphLab and ASPIRE by 5.2× and 4.1×, respectively. Amortizing the processing cost yields additional average speedups of 2× and 7.9×, respectively. Furthermore, our amortization techniques on average perform 16.5% faster than the optimizations incorporated in Chronos [Han et al. 2014].

2. EVOLVING GRAPH AND ITERATIVE PROCESSING

We formalize evolving graphs and discuss how they are iteratively processed.

2.1. Evolving Graph

An evolving graph $\mathcal{G}$ is a graph that undergoes structural changes over time. These structural changes take place via *addition and deletion of edges and vertices*. Formally, an evolving graph $\mathcal{G} = \langle G_1, G_2, \ldots, G_k \rangle$ is a sequence of *k graph snapshots* taken at different points in time. In general, the structural changes that cause a transition from snapshot G_{i-1} to snapshot G_i involve both addition and deletion of edges and vertices. Note that a change in edge weight can be viewed as a deletion of the edge followed by its insertion with a different weight. Figure 2(a), Figure 2(b), and Figure 2(c) together show an evolving graph consisting of three graph snapshots taken at t_1, t_2, and t_3, respectively.

2.2. Computation Over Evolving Graphs

We briefly discuss the iterative vertex-centric processing over a simple graph and then describe how it is performed over an evolving graph.

Iterative Vertex-Centric Processing. In this work, we focus on iterative vertex-centric graph algorithms, used in a wide range of modern mining and analytics tasks. In a vertex-centric graph algorithm, computation is written from the perspective of a single vertex. For a given vertex, all the neighboring vertex values are fetched and a new value is computed using these fetched values. If the value of a vertex changes, its neighbors become *active*, that is, they are scheduled to be processed in the next iteration. This process terminates when all the vertices in the graph become inactive.

ALGORITHM 1: Iterative Algorithm on an Evolving Graph.

```
1:  function EXECUTE($G_i$)
2:    for vid ∈ GET-ACTIVE-VERTEX($G_i$) do
3:      vertex ← FETCH(vid, $G_i$)
4:      nbrs ← ∅
5:      for nid ∈ GET-NEIGHBORS(vertex) do
6:        nbrs ← nbrs ∪ FETCH(nid, $G_i$)
7:      end for
8:      old-value ← vertex.GET-VALUE( )
9:      comp-value ← $f$ (vertex, nbrs)
10:     vertex.SET-VALUE(comp-value)
11:     if |old-value − comp-value| > $\epsilon$ then
12:       ACTIVATE-NEIGHBORS(vertex)
13:     end if
14:     STORE(vertex, $G_i$)
15:   end for
16: end function

17: function MAIN($\mathcal{G}$)
18:   for each graph $G_i \in \mathcal{G}$ do
19:     INITIALIZE($G_i$)
20:     ACTIVATE-VERTICES($G_i$)
21:     do
22:       parallel-for all threads do
23:         EXECUTE($G_i$)
24:       end parallel-for
25:       BARRIER( )
        /* Global termination condition */
26:     while there are active vertices
27:     OUTPUT-VERTEX-VALUES($G_i$)
28:   end for
29: end function
```

In a given iteration, vertices are processed in parallel such that a vertex is completely processed by the same set of threads on the same machine. This results in a simple and intuitive parallel algorithm shown in function `EXECUTE()` in Algorithm 1. Since the techniques presented in this work are general and independent of any specific graph processing environment, we present the execution plan using high-level load/store primitives—fetching/storing data from/to (local or remote) machines is achieved transparently using the `FETCH`/`STORE` operations (lines 3, 6, and 14), which can have different implementations on different platforms; for example, a message-passing-based system like Pregel [Malewicz et al. 2010] may use `SEND`/`RECEIVE` to transfer values across machines. Since only those vertices that are scheduled to be processed in a given iteration must be computed, `GET-ACTIVE-VERTEX` (line 2) returns the next vertex to be processed in the current iteration. While processing this vertex, the neighboring values are obtained using `GET-NEIGHBORS` (line 5) that internally fetches the neighboring vertex values residing on local and remote machines. Upon computation, if the change in the value of a vertex exceeds threshold ϵ (line 11), the neighbors of the vertex are activated for future processing by `ACTIVATE-NEIGHBORS` (line 12). It is interesting to note that the computations performed (line 9) are typically quite simple and threads are often seen waiting for values to be fetched (from local/remote memory/disk). In other words, these iterative graph algorithms are typically network bound when executed on a distributed environment.

Evolving Graph Processing. Analyzing an evolving graph involves *repeating* the iterative graph analysis computation over each graph snapshot. The `MAIN()` function in Algorithm 1 shows how an evolving graph is processed by invoking the same iterative `EXECUTE()` (line 23) function over different graph snapshots, one at a time. Since the results of iterative analysis are required for each of the graph snapshots, `OUTPUT-VERTEX-VALUES()` (line 27) is invoked immediately after processing of a given graph snapshot terminates.

As an example, the iterative Single Source Shortest Path (SSSP) computation for evolving graph snaphots G_1, G_2, and G_3 are shown in Tables I, II, and III, respectively. Computation on G_1 proceeds as follows. Initially, all the vertices are set to ∞ and the source vertex (vertex d) is set to 0. The outgoing neighbors of d are b and c, which are active in iteration 1. Their values are computed by fetching them and their incoming neighbors, that is, vertices a, b, c, and d. Since both b and c change, their outgoing neighbors (vertex e) become active in iteration 2. Processing terminates when there

Table I. Execution of SSSP on Snapshot G_1 (Figure 2(a))

Step	Vertex Distances					Active
	a	b	c	d	e	Vertices
0	∞	∞	∞	0	∞	-
1	∞	2	4	0	∞	b,c
2	∞	2	4	0	3	e
3	9	2	4	0	3	a
4	9	2	4	0	3	b,c,d

Table II. Execution of SSSP on Snapshot G_2 (Figure 2(b))

Step	Vertex Distances							Active
	a	b	c	d	e	f	g	Vertices
0	∞	∞	∞	0	∞	∞	∞	-
1	∞	2	4	0	∞	∞	∞	b,c
2	∞	2	4	0	3	6	5	e,f,g
3	7	2	4	0	3	6	5	a,b,g
4	7	2	4	0	3	6	5	b,c,d

Table III. Execution of SSSP on Snapshot G_3 (Figure 2(c))

Step	Vertex Distances									Active
	a	b	c	d	e	f	g	h	i	Vertices
0	∞	∞	∞	0	∞	∞	∞	∞	∞	-
1	∞	2	4	0	∞	∞	∞	∞	∞	b,c
2	∞	2	4	0	3	6	5	5	3	e,f,g,h,i
3	9	2	4	0	3	6	5	5	3	a,b,f,g,i
4	9	2	4	0	3	6	5	5	3	b,c,d

are no more active vertices. After processing completes for G_1, snapshot G_2, and later, snapshot G_3, is processed in a similar manner (Table II and Table III).

Note that evolving graph processing is inherently different from streaming graph processing [Ediger et al. 2012]; streaming graph processing does not rely on strict notion of graph snapshots and iterative processing continues while the graph structure changes rapidly, hence terminating to the final solution for the most updated graph. Evolving graph processing, on the other hand, involves processing all the intermediate graph snapshots and producing the final converged results for each of the snapshots.

Since graphs being processed are large, significant effort is involved both in fetching of values into memory of processing sites and carrying out the required computation. We recognize two opportunities (Section 4 and Section 5) to amortize these costs across multiple graph snapshots of an evolving graph. Note that the opportunities and proposed techniques are independent of any graph processing framework and any processing environment (distributed, shared-memory, out-of-core, etc.).

3. TEMPORAL LAYOUT OF EVOLVING GRAPHS

Computations performed over a static graph are typically vertex centric and hence, computation of a given vertex requires values of its neighboring vertices. To preserve locality in this case, graph processing systems typically lay out the graph structure using a variant of adjacency list format. Such a representation can be directly used to represent evolving graphs by laying out individual graph snapshots one after the other so that locality within each of the graph snapshots is preserved.

However, consecutive graph snapshots have high structural overlap that can be exploited to make the representation space efficient. Moreover, this structural overlap across snapshots can be used to expose the underlying temporal locality, which can be further used to accelerate processing. Hence, we extend the traditional adjacency list-based representation so that the represented graph structure is a union of all graph snapshots. This is analogous to using a *Structure-of-Arrays (SoA)* layout instead of an *Array-of-Structures (AoS)* layout where the graph snapshots collectively represent an array.

Figure 3 shows how we represent evolving graphs. The overall structure holds the union of all the captured graph snapshots, while the information about the individual

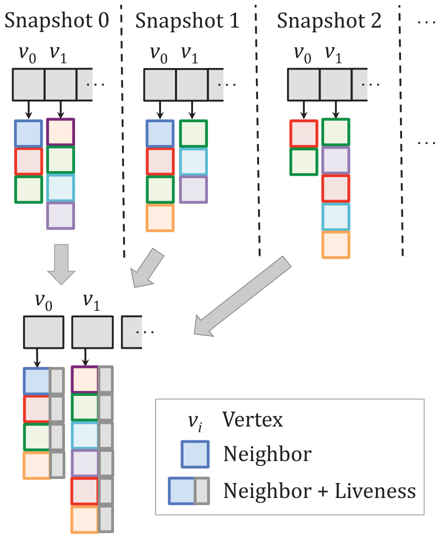


Fig. 3. Temporal layout of evolving graphs.

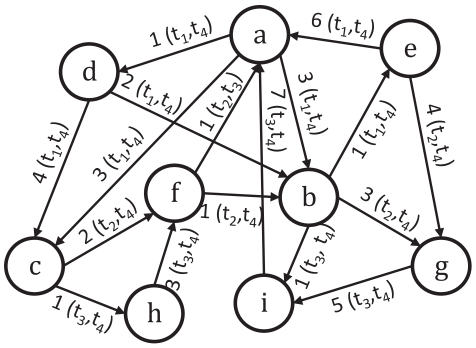


Fig. 4. Evolving graph $\mathcal{G}$ for example evolving graph in Figure 2.

snapshots is maintained using liveness intervals tagged over edges. A liveness interval is represented by a *begin* and *end* timestamp and each edge in the evolving graph has a vector of such liveness intervals that allows tracking multiple additions and deletions of same edges throughout the lifetime of the application. These vectors are sorted based on the begin timestamps to allow quick access of required data using a *forward-only pointer*. Note that vertices do not need liveness intervals because the presence of a vertex in a snapshot can be determined by the presence of its edges.

Every graph snapshot has an associated timestamp at which it was captured, that is, it is the time up to which all the updates have been applied to the snapshot. An edge is considered part of the snapshot if the snapshot's timestamp falls in between the begin and end timestamps for some pair in the edge's liveness vector. Every edge also has a vector of immutable values to capture different edge weights for different lifetimes.

Figure 4 shows an example of evolving graph $\mathcal{G} = \langle G_1, G_2, G_3\rangle$ with three graph snapshots that are shown in Figure 2. The individual snapshots G_1, G_2, and G_3 taken at t_1, t_2, and t_3 are shown in Figure 2(a), 2(b), and 2(c), respectively.

Space Complexity. An evolving graph with k snapshots, $\mathcal{G} = \langle G_1, G_2, \ldots, G_k\rangle$, consumes $O(k \times (|V| + |E|))$ space when it is represented as a sequence of separate snapshots. On the other hand, our unified representation consumes $O(|V| + (p \times |E|))$ space, where p is the maximum number of times any edge in the graph is updated. Typically, $p \ll k$ and hence, our unified representation is space efficient.

Compactly representing evolving graphs using lifetime intervals naturally exposes the temporal locality present across graph snapshots, which can be exploited by changing the order in which processing occurs, as discussed in subsequent sections.

4. FETCH AMORTIZATION

The high structural overlap across multiple snapshots leads to similar patterns in fetch requests for vertices when the individual snapshots are processed. Table IV provides the lists of *Fetched Vertices* for each iteration during the SSSP computation performed on three graph snapshots in Section 2. On comparing the *Fetched Vertices* columns for

Table IV. Fetched Vertices for SSSP on G_1, G_2, and G_3

Step	**Fetched Vertices**		
	G_1	G_2	G_3
1	a,b,c,d	a,b,c,d,f	a,b,c,d,f
2	b,e	b,c,e,f,g	b,c,e,f,g,h,i
3	a,e	a,b,d,e,f,g	a,b,d,e,f,g,h,i
4	a,b,c,d	a,b,c,d,f	a,b,c,d,f

Table V. Average % Fetch Overlap Across Consecutive Snapshots Over Different Datasets (Names Include Number of Snapshots) for SSSP Algorithm

Evolving Graph	**Overlap**
Twitter-25	66.0%
Delicious-100	58.8%
DBLP-100	42.4%
Amazon-100	44.8%
StackEx-100	53.5%
Epinions-100	51.2%
Slashdot-70	47.2%

the corresponding (same) iterations of consecutive graph snapshots, we observe that a significant number of vertices that are fetched are common.

The high degree of fetch overlap can also be observed in real datasets. Table V presents the degree of fetch overlap across consecutive snapshots when computing SSSP for real-world evolving graphs (see Table VIII in Section 7.1 for a description of graphs). As we can see, the fetch overlap is very high—over 49% of vertices are common among consecutive snapshots. This naturally leads us to the following question:

Can we reorder computations such that overlapping fetches can be batched together?

4.1. Fetch Amortization via Computation Reordering

The temporal layout of an evolving graph that exposes the high structural overlap across multiple snapshots can be efficiently utilized to aggregate the fetch requests of same vertices across different snapshots. We achieve this by computation reordering enabled using loop transformation techniques.

In the original processing model (Algorithm 1), the outer `for` loop (line 18) processes graph snapshots one after the other, while the inner `do-while` loop (line 21) processes a single snapshot. These two loops can be transformed such that the inner loop processes a batch of snapshots at the same time, while the outer loop iterates through different batches of snapshots to be processed. Now, while simultaneously processing the batch of snapshots, the fetch requests for different versions of the same vertex can be aggregated together by fusing multiple versions of the innermost loop which processes active vertices (line 2) such that processing of multiple versions of the same vertex belonging to different snapshots occur at the same time. This allows aggregating the fetch requests performed for multiple versions of same vertices.

The preceding idea is generalized by simultaneously processing Δ consecutive graph snapshots in parallel.

Fetch Amortization (FA). Fetch amortization *simultaneously processes* Δ snapshots ($G_{i+1}, G_{i+2} \dots G_{i+\Delta}$) so that fetches of vertices common among some of the Δ snapshots can be *aggregated* to amortize fetch cost across snapshots.

Algorithm 2 shows the iterative parallel algorithm that incorporates fetch amortization. Note that this algorithm processes Δ snapshots in parallel denoted by $\mathcal{G}\Delta$. When a thread invokes `EXECUTE` on a vertex, it performs an aggregate fetch via `MULTI-FETCH` (as opposed to `FETCH`) to get all its Δ snapshots (lines 3 and 6), computes the new values for the Δ snapshots (lines 8–15), and then performs an aggregate store via `MULTI-STORE` to update the Δ snapshots (line 16). When one graph snapshot is fully processed (G_{stable}), it is replaced by another snapshot (G_{next}) so that the algorithm is always processing Δ snapshots in parallel (lines 28–35).

The preceding computation reordering is legal because processing of individual snapshots occurs independently and hence, there are no dependencies between iterations of the outer `for` loop. This means, upon unrolling the outer `for` loop, the inner `do-while` loops can be fused together by maintaining multiple versions of vertex values and capturing individual snapshot's convergence and vertex activations as described in the following.

ALGORITHM 2: Incorporating Fetch Amortization.

```
 1: function EXECUTE( $\mathcal{G}\Delta$)
 2:    for vid ∈ GET-ACTIVE-VERTEX($\mathcal{G}\Delta$) do
 3:       vertices ← MULTI-FETCH(vid, $\mathcal{G}\Delta$)
 4:       nbrs ← ∅
 5:       for nid ∈
          GET-NEIGHBORS(vertices, $\mathcal{G}\Delta$) do
 6:          nbrs ← nbrs ∪
             MULTI-FETCH(nid, $\mathcal{G}\Delta$)
 7:       end for
 8:       for each vertex ∈ vertices do
 9:          old-value ← vertex.GET-VALUE( )
10:          comp-value ← $f$ (vertex, nbrs)
11:          vertex.SET-VALUE(comp-value)
12:          if |old-value − comp-value| > $\epsilon$
             then
13:             ACTIVATE-NEIGHBORS(vertex)
14:          end if
15:       end for
16:       MULTI-STORE(vertices,$\mathcal{G}\Delta$)
17:    end for
18: end function
19: function MAIN($\mathcal{G}$, $\Delta$)
20:    $\mathcal{G}\Delta \leftarrow \mathcal{G}[0 \ldots \Delta - 1]$
21:    INITIALIZE($\mathcal{G}\Delta$)
22:    ACTIVATE-VERTICES($\mathcal{G}\Delta$)
23:    do
24:       parallel-for all threads do
25:          EXECUTE($\mathcal{G}\Delta$)
26:       end parallel-for
27:       BARRIER( )
28:       for each $G_{stable} \in \mathcal{G}\Delta$ with
          no active vertex do
29:          OUTPUT-VERTEX-VALUES($G_{stable}$)
30:          $G_{next}$ ← GET-NEXT-GRAPH($\mathcal{G}$)
31:          INITIALIZE($G_{next}$)
32:          ACTIVATE-VERTICES($G_{next}$)
33:          $\mathcal{G}\Delta \leftarrow \mathcal{G}\Delta \setminus \{G_{stable}\}$
34:          $\mathcal{G}\Delta \leftarrow \mathcal{G}\Delta \cup \{G_{next}\}$
35:       end for
36:    while there are
       unprocessed graphs in $\mathcal{G}$
37: end function
```

4.2. Mutable Vertex Values

To incorporate Fetch Amortization (FA), the in-memory representation of the evolving graph should support simultaneous processing of Δ graph snapshots. Hence, mutable vertex values are vector expanded to store Δ values that are being computed simultaneously. If Δ snapshots are simultaneously processed in parallel, the memory consumption increases to $O((\Delta \times |V|) + (p \times |E|))$. This increase in memory consumption limits the degree up to which multiple graph snapshots can be processed together. As we will see in Section 7, we process 10 snapshots together for this reason.

4.3. Vertex Activations

During execution, not all Δ versions of the vertex may be activated to be processed since activations are dependent on computed values that may not be the same for different versions. Hence, vertex activations for different versions need to be explicitly tracked, which is done using additional bits in the worklist that maintains active vertices. While remote activation messages can be sent across machines as soon as they become available, multiple activations of the same vertex for different versions can also be aggregated together. Hence, we send out the remote activation messages after all the Δ versions of the vertex are processed, that is, at the end of the iteration that processes the vertex (line 15).

While a synchronous processing model requires that only active vertices in a given iteration are processed in that iteration, an asynchronous processing model relaxes this notion and tolerates computations on vertices even when they are not activated.

In this case, activations can be tracked at vertex level, which eliminates the need to maintain additional bits in the activation worklist.

4.4. Convergence Detection

To quickly determine whether or not processing has converged, individual machines either maintain a convergence flag that is set to false whenever vertices are activated to be processed in the next iteration, or rely on checking the activation queue. This convergence information is exchanged between machines, typically via parallel reduction, to determine whether all machines should stop processing. Since FA processes Δ snapshots in parallel, we maintain a vector of convergence flags, one for each snapshot being processed, and set a given flag to false whenever the corresponding version of any vertex is activated to be processed. Now, instead of exchanging the convergence information using scalar reduction, the vector-expanded convergence flags from all the machines are reduced together. Determining convergence separately for each of the Δ snapshots allows pipelined execution of snapshots where snapshots that are fully processed are replaced by new snapshots that are to be processed next (lines 28–35).

4.5. Caching and Message Aggregation

Distributed graph processing environments typically rely on caching to make remote values (vertices, etc.) locally available. When caches are used to store vertex values, the overall fetches from remote machines is reduced. However, this benefit does not come for free—the cache management protocol itself generates additional messages to maintain the data values coherent. A major part of the additional messages are *invalidates* with *piggy-backed updates* (as in the case of ASPIRE & GraphLab). Hence, once the stores of multiple snapshots of a vertex are aggregated by fetch amortization, the corresponding cache protocol messages for those versions of the same vertex, going to the same destination, are also aggregated. This reduces the overheads incurred by the caching protocol itself.

While fetch amortization can be directly incorporated in ASPIRE using the iterative execution algorithm presented in Algorithm 2, we have also integrated fetch amortization in GraphLab, the details of which will be discussed in Section 6.

5. PROCESSING AMORTIZATION

In our SSSP example from Section 2.2, comparing the final results for vertices of snapshots G_1 (Table I) and G_2 (Table II) reveals that values of vertices b, c, d, and e are the same for both snapshots. Vertex a is the only common vertex whose result value is not the same for the two snapshots; note vertices f and g are present only in G_2. Similarly, on comparing the results computed for G_2 (Table II) with G_3 (Table III) we observe that resulting values for vertices b, c, d, e, f, and g are the same.

To further confirm the preceding observation, we measured the average percentage of vertex values that are found to be the same across consecutive snapshots for seven iterative algorithms (listed in Table IX) on the *Slashdot* graph. The results in Table VI show that this is a high percentage—on average across consecutive snapshots, over 76% of vertices had values for a given snapshot identical to those for the previous snapshot. Hence, we explore the following question:

Can we leverage the results (potentially partially computed) for previous graph snapshots to accelerate processing of later snapshots?

Moreover, when we compare *unstable values*, that is, the intermediate results, from step 2 in G_2 (Table II) with the final results of G_3 (Table III), we observe that the values of most available vertices in G_1 (all except vertex a) are exactly the same. This

Table VI. Average % Overlap of Vertex Values Across Consecutive Graph Snapshots of *Slashdot* Input

Benchmark	Overlap
PageRank	87.9%
Single Source Shortest Path	99.3%
Single Source Widest Path	80.2%
Circuit Simulation	76.7%
Heat Simulation	99.8%
Graph Coloring	97.5%

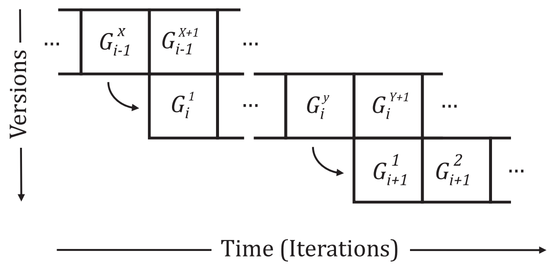


Fig. 5. Effect of processing amortization.

means, unstable results from previous snapshots may also be used in order to accelerate processing of later snapshots. This allows amortizing processing costs simultaneously while leveraging the first opportunity, that is, processing consecutive snapshots in batches by using unstable results from previous snapshots to process later snapshots.

5.1. Processing Amortization via Feeding

Based upon the previous observations, we propose *processing amortization* technique in which the computed values (from stable and unstable vertices) for a given snapshot can be used to accelerate processing of the next snapshot by avoiding repeated computation of same values. In the presence of fetch amortization, when processing of a snapshot finishes, the new snapshot that is included in the working set of snapshots being processed can be fed from its previous snapshot (analogous to pipelined processing), which might not have converged to its stable solution (see Figure 5). The advantage of doing this is twofold: it removes redundant computations so that the needed (nonredundant) processing can be performed sooner; and vertices stabilize faster because the computed values used from previous snapshots are already close to stability.

Processing Amortization (PA). While snapshot G_{i-1} is being processed, when processing of G_i is initiated for simultaneous processing, processing amortization *feeds* current vertex values from snapshot G_{i-1} into G_i as initializations to accelerate the processing of G_i.

Algorithm 3 shows the iterative processing with processing amortization. When processing of snapshot G_{stable} completely finishes (i.e., there are no active vertices), and the processing of a new snapshot G_{next} is initiated, instead of initializing vertices of G_{next} to their standard initial values, they are copied from the latest snapshot G_{prev}, which is still being processed (lines 14 and 15). This step involves copying the entire working set, which includes the following three components: the vertex values mapped to the machines; activations for vertices; and, any caches holding vertex values. Vertices that are active for G_{prev} potentially have unstable values and hence, it is necessary to activate them for G_{next}.

Note that the values that are fed from G_{i-1} to G_i may not have stabilized yet as G_{i-1} is in the midst of processing (see Figure 5). However, as our experiments show, this optimization yields benefits because G_{i-1} has often made enough progress toward stability that it helps accelerate G_i's termination. On the other hand, in absence of fetch amortization, the values fed by PA from G_{i-1} to G_i are always stable.

ALGORITHM 3: Incorporating Process Amortization.

```
1:  function MAIN($\mathcal{G}$, $\Delta$)
2:    $\mathcal{G}\Delta \leftarrow \mathcal{G}[0 \ldots \Delta - 1]$
3:    INITIALIZE($\mathcal{G}\Delta$)
4:    ACTIVATE-VERTICES($\mathcal{G}\Delta$)
5:    do
6:      parallel-for all threads do
7:        EXECUTE($\mathcal{G}\Delta$)
8:      end parallel-for
9:      BARRIER( )
10:     for each $G_{stable} \in \mathcal{G}\Delta$ with
        no active vertex do
11:       OUTPUT-VERTEX-VALUES($G_{stable}$)
12:       $G_{prev} \leftarrow$ GET-LATEST-GRAPH($\mathcal{G}\Delta$)
13:       $G_{next} \leftarrow$ GET-NEXT-GRAPH($\mathcal{G}$)
14:       INITIALIZE($G_{next} \leftarrow G_{prev}$)
15:       ACTIVATE-VERTICES($G_{next} \leftarrow G_{prev}$)
16:       $\mathcal{G}\Delta \leftarrow \mathcal{G}\Delta \backslash \{G_{stable}\}$
17:       $\mathcal{G}\Delta \leftarrow \mathcal{G}\Delta \cup \{G_{next}\}$
18:     end for
19:   while there are
      unprocessed graphs in $\mathcal{G}$
20: end function
```

5.2. Applicability and Correctness

Even though processing amortization shows the potential to accelerate evolving graph processing, it is important to ensure that the technique guarantees correct results for each of the graph snapshots. Note that any graph algorithm capable of executing over a streaming graph processing framework (like STINGER [Ediger et al. 2012]) can safely leverage PA mainly because the algorithm can handle structural changes. While this captures a large class of graph algorithms, to better understand the set of feasible algorithms, we identify the characteristics of graph algorithms and then reason about correctness of each of the algorithms considered in our evaluation.

5.2.1. Mutation Properties. Feeding of values across consecutive snapshots can be viewed as mutation in the graph structure while the overall computation progresses. Hence, we study the impact of PA on correctness of results by reasoning about the behavior of graph algorithms when the structure of the graph is mutated, that is, when vertices and edges are added and deleted to progress from one snapshot to the next. Determining properties in this manner is similar to the study performed in Burke and Ryder [1990] related to incremental dataflow analysis algorithms.

Note that vertex addition occurs when an edge with a new end vertex is added. Similarly, vertex deletion occurs when all its edges get deleted. Hence, we model vertex addition/deletion via addition/deletion of its edges.

(A) Global Mutation Property: We define two properties that directly characterize the results obtained by the iterative algorithms based on their behavior and the values using which they start computing the results.

[P-INIT] Convergence to correct results is independent of vertex initializations.
[P-MONO] Intermediate results exhibit monotonic trend under a given ordering.

For algorithms exhibiting [P-INIT], when the graph structure mutates in the middle of computation, subsequent processing operates on vertex values coming from

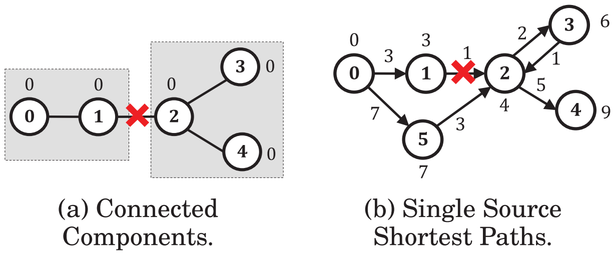


Fig. 6. Edge deletion examples.

computations performed prior to the mutation. These values can be considered as new vertex initializations using which the processing converges to stable values. This means, feeding values does not affect the correctness of results even when graph structure changes because the fed values are considered as vertex initializations that do not impact correct convergence; only the path to achieve final convergence changes by feeding different values. Hence, it is easy to see that [P-INIT] is a sufficient condition for an algorithm to be able to use PA correctly. However, note that [P-INIT] is not a necessary condition.

Algorithms exhibiting [P-MONO] can be reasoned about in a similar manner as monotonic dataflow functions [Burke and Ryder 1990] where the algorithms aim to achieve a maximum or a minimum fixed point based on a cost metric. Such algorithms need to be carefully evaluated to determine whether processing can recover from local maxima/minima and correctly converge to global maxima/minima when edges are added and deleted in the middle of processing. We illustrate such a reasoning with the Connected Components algorithm, which is based on iterative label propagation [Zhu and Ghahramani 2002] as shown next.

$$v.value = \begin{cases} v.id & \text{... when } iteration = 0 \\ \min(v.value, \min\limits_{e \in \text{edges}(v)} (e.other.value)) & \text{... otherwise.} \end{cases}$$

The preceding function starts with an approximation where every vertex is in a separate component and then iteratively merges components by picking the minimum component value available across the vertex's neighborhood. When edge addition occurs, the subsequent iteration computes over the vertices of the edge which incorporate the new neighbor's component value. If the two vertices have different component values, [P-MONO] ensures that the vertex with the larger value will get reassigned the smaller value from the new neighbor, the effects of which are propagated throughout the graph. When edge deletion occurs, the deleted edge can split a component into two separate components as shown in Figure 6(a). In this case, recomputing the component value for vertex 2 incorrectly results in value 0 because its remaining neighbors, that is, vertices 3 and 4, still exhibit the old component values. This means connected components can safely leverage PA only on growing graphs, that is, those that evolve purely using addition of new edges.

(B) Local Mutation Property: Contrary to the preceding properties, we define [P-EDGE] at the level of vertex functions.

[P-EDGE] Computation for a vertex only depends on values coming from its edges.

Algorithms exhibiting [P-EDGE] majorly compute values based on graph structure and hence, they rely lesser on previously computed values. This allows the computation to self-correct values and propagate the corrections throughout the graph in an iterative manner. We illustrate the effectiveness of [P-EDGE] using the Single Source Shortest Paths (SSSP) algorithm; next are two variants for expressing SSSP that correctly compute shortest paths on a given graph snapshot.

$$\text{(A) } v.value = \begin{cases} 0 & \text{... when } v \text{ is source} \\ \infty & \text{... when } iteration = 0 \\ \min(v.value, \min\limits_{e \in \text{inEdges}(v)} (e.weight + e.source.value)) & \text{... otherwise,} \end{cases}$$

$$\text{(B) } v.value = \begin{cases} 0 & \text{... when } v \text{ is source} \\ \infty & \text{... when } iteration = 0 \\ \min\limits_{e \in \text{inEdges}(v)} (e.weight + e.source.value) & \text{... otherwise.} \end{cases}$$

The preceding functions initially set source to 0 and start with an approximation where all vertices are unreachable from the source, that is, the path length is ∞. Then, they iteratively compute shorter paths for every vertex based on available shortest paths in its neighborhood.[1] The only difference between two variants is that while computing the new path value, variant A considers its previous path value, whereas variant B does not, that is, variant A does not exhibit [P-EDGE], whereas variant B does. When edge addition occurs, the target of newly added edges compute new values and if the newly added edge results in shorter paths, both the variants compute the corrected path, effects of which are iteratively propagated throughout the graph. When edge deletion occurs, however, the two variants behave differently, as illustrated in Figure 6(b). Upon deletion of edge (1, 2), variant A still incorrectly retains its shortest path as 4 (note that simply resetting its value to ∞ does not resolve the issue because of the back-edge from vertex 3). Variant B, on the other hand, corrects its path value to 10 ($0 \rightarrow 5 \rightarrow 2$) due to [P-EDGE]. This means, variant A can safely leverage PA only on growing graphs, whereas variant B can do so on evolving graphs that include deletion of edges too.

It is interesting to note that [P-EDGE] is neither a necessary nor a sufficient property to guarantee correctness of results when using PA. However, it is important to reason about [P-EDGE] because it defines a subset of vertex algorithms for which PA may be safely used.

5.2.2. Correctness of Algorithms. Using the properties described previously, we now study each of our benchmark algorithms considered in our evaluation (listed in Table IX) to discuss their correctness while incorporating PA. Table VII shows the vertex functions for each of the algorithms (vertex initializations are eliminated for simplicity).

(A) Shortest and Widest Paths: Our SSSP algorithm is the variant B from the preceding discussion that exhibits [P-EDGE]. As discussed previously, it can safely leverage PA. Single Source Widest Paths (SSWP), similar to SSSP, exhibits [P-EDGE]. However, as discussed previously, [P-EDGE] is not a sufficient condition to guarantee correctness while using PA and hence, we will see that edge deletions can impact the correctness of results of SSWP. When edge addition occurs, the target vertex recomputes its value based on the available edges and the newly added edge can increase the fed path value (due to monotonic nature of `max`), which is further observed by its neighbors to be propagated throughout the graph. Hence, correct results are guaranteed when PA is used with edge additions. However, when edge deletion occurs, the target vertex can incorrectly compute its path value to be same as the fed value from an alternate path that previously passed through itself, disallowing the vertex to

[1]Edge weights are positive for path problems.

Table VII. Various Vertex-Centric Graph Algorithms

Algorithm	**Vertex Function**
PR	$v.rank \leftarrow 0.15 + 0.85 \times \sum_{e \in \text{inEdges}(v)} e.source.rank$
SSSP	$v.path \leftarrow \min_{e \in \text{inEdges}(v)} (e.source.path + e.weight)$
GC	$conflict \leftarrow \bigvee_{e \in \text{edges}(v)} ((v.color = e.other.color) \text{ and } (v.id < e.other.id))$ $decrease \leftarrow \text{true if } \exists c < v.color \text{ s.t. } \forall_{e \in \text{edges}(v)} (e.other.color \neq c)$ if $conflict = true$ or $decrease = true$ then: $\quad v.color \leftarrow c$: where $\forall_{e \in \text{edges}(v)} (e.other.color \neq c)$
HS	$v.value \leftarrow \frac{\sum_{e \in \text{inEdges}(v)} C_1 \times e.source.value + C_2}{\lvert\text{inEdges}(v)\rvert}$
SSWP	$v.path \leftarrow \max_{e \in \text{inEdges}(v)} (\min(e.source.path, e.weight))$
CS	$v.value \leftarrow \frac{\sum_{e \in \text{inEdges}(v)} e.weight \times e.source.value}{\sum_{e \in \text{inEdges}(v)} e.weight}$

step out of its previous approximation. Note that this does not occur in SSSP because positive edge weights ensure that such incorrect cyclic feeding does not occur. Hence, we evaluate PA on SSWP with growing evolving graphs (i.e., allowing edge additions only).

(B) Graph Coloring: Graph Coloring (GC) is an interesting algorithm that effectively corrects vertex color values based on the neighboring values. In particular, when edge addition occurs, if the new neighboring vertices have the same color values, the vertex with the lower id updates its value to a new color value based on its neighbors, the effects of which iteratively propagate throughout the graph. When edge deletion occurs, the neighboring vertices are reset to default color values (to guarantee minimality) and recomputed to receive new color values based on the remaining neighborhood. Hence, even after structural mutations are observed across snapshots, the fed incorrect color values are treated as initializations using which correct coloring solution is computed, that is, GC exhibits [P-INIT] allowing PA to be safely used.

(C) PageRank, Heat Simulation, Circuit Simulation: It is well known that for algorithms like PageRank (PR), the initial vertex values do not matter [Farahat et al. 2005], that is, they exhibit [P-INIT]. This allows PR to safely leverage PA. Similarly, both Circuit Simulation (CS) and Heat Simulation (HS) do not require vertex initializations except for their source/ground vertices, which are not computed upon during processing; hence, they also exhibit [P-INIT] and can safely incorporate PA while processing.

Beyond the set of benchmarks considered in our evaluation, we studied many different vertex programs in order to ensure applicability of PA. Various algorithms including, but not limited to, K-Means, Wave Simulation, Reachability, Maximal Independent Set, Triangle Counting, NumPaths, BFS, and Diameter can safely incorporate PA, ensuring its wider applicability.

6. GRAPH PROCESSING SYSTEMS

The amortization techniques proposed in this work are independent of any specific graph processing system (and its internals) that can process a given graph snapshot one at a time. Various graph processing systems have been developed across different processing environments, some of which include Pregel [Malewicz et al. 2010], GraphLab [Low et al. 2012; Gonzalez et al. 2012], ASPIRE [Vora et al. 2014], GraphX [Gonzalez et al. 2014] and GraphChi [Kyrola et al. 2012]. To evaluate the efficacy of our proposed techniques, we incorporate our amortization techniques in two of those systems, namely, GraphLab and ASPIRE. We briefly discuss these two frameworks and how we incorporate the amortization techniques to process evolving graphs.[2]

6.1. ASPIRE

ASPIRE [Vora et al. 2014] uses an object-based DSM where the objects (vertices and edges) are distributed across the cluster such that there is exactly one global copy of each object in the DSM. The machine-level caches are kept coherent using a directory-based protocol that provides strict consistency [Goodman 1991]. Since the vertex values are usually small, invalidates are piggybacked with the updated data. Other standard optimizations like *Work Collocation* (or locality in Dean and Ghemawat [2008]) and *Message Aggregation* (similar to bulk transfer in Liang et al. [1996]) are enabled.

The graph is partitioned across machines to minimize edge cuts for reducing communication using the well-known ParMETIS [Karypis et al. 1997] partitioner. The runtime holds a flag per vertex indicating whether a vertex is active for the next iteration. For a particular vertex, the programmer can determine whether to activate neighbors on outgoing, incoming, or both edges. After each iteration, a check is made to see whether computation for a graph snapshot is complete. If not, the activation vectors are exchanged between the machines so that each machine aggregates the vertices it needs to work on in the next iteration. Before starting the next iteration, a check is made to ensure that all the invalidates have been acted upon. This guarantees that updated values are visible and correctly used in the next iteration.

In order to incorporate FA, we take advantage of ASPIRE's single-writer model by having the same thread process multiple snapshots of the same vertex at the same time. This allows remote fetches, stores, and all the cache protocol messages to be automatically aggregated across multiple snapshots of same vertices. To transfer objects to and from the DSM, the runtime provides `DSM-Fetch`, `DSM-MultiFetch` and `DSM-Store`, `DSM-MultiStore` APIs. This hides the internal details of the runtime and allows parallel algorithms to directly execute over the DSM.

6.2. GraphLab

GraphLab [Low et al. 2012; Gonzalez et al. 2012] is a popular distributed graph processing framework that provides shared memory abstractions to program over a distributed environment. It provides synchronous and asynchronous engines to process the graph and allows users to express computations in a gather-apply-scatter (GAS) model. Similarly to ASPIRE, boundary vertices are cached (named as *ghost vertices*) on different machines to make them locally available and the graph is partitioned using a balanced p-way partitioning strategy to minimize cuts and balance computation.

Since processing is done on a set of machines, FA is used to tolerate network latencies especially for ghost vertices, which need to be continuously synchronized. This is done by vectorizing the vertex values and updating the vertex functions to process multiple versions of the same vertex. PA is incorporated by feeding values from the most recent snapshot to the next snapshot; we only allow this feeding when all the executing snapshots become stable because GraphLab prevents dynamic changes to all vertex values and graph structure while the engine is processing.

[2]Details about system runtime, caching, programming, performance numbers, etc., for each of the frameworks can be found in their respective publications.

Table VIII. Real-World Evolving Graphs Taken from KONECT [Kunegis 2013] Repository for Experiments

Evolving Graph	V	E
Twitter-25 [Cha et al. 2010]	8.4B	25.5B
Delicious-100 [DeliciousUI 2014]	1.6B	15.2B
DBLP-100 [Ley 2002]	92.7M	963.8M
Amazon-100 [Lim et al. 2010]	144M	299.8M
StackEx-100 [StackExchange]	27.6M	70.7M
Epinions-100 [Massa and Avesani 2005]	9.4M	47.5M
Slashdot-70 [Gómez et al. 2008]	2.1M	7.5M

Table IX. Benchmark Programs Used in Experiments

Benchmark	Type
PageRank (PR)	Graph Analytics & Mining
Single Source Shortest Path (SSSP)	Graph Analytics & Mining
Single Source Widest Path (SSWP)	Graph Analytics & Mining
Graph Coloring (GC)	Graph Analytics & Mining
Circuit Sim. (CS)	PDE
Heat Sim. (HS)	PDE

6.3. Other Graph Processing Frameworks

Incorporating both amortization techniques in various graph processing frameworks requires knowledge about their processing engines and how computations are expressed (vertex centric, edge centric, etc.). This is because different frameworks are developed in order to exploit different processing environments and hence, they operate in different manners. However, we identify two simple techniques that can be easily used to incorporate both transformations in a new processing framework.

For FA, the values associated with graph structure (vertices and edges) can be vectorized to hold multiple versions in order to allow simultaneous processing of multiple snapshots. Also, the user defined processing function for a given algorithm can be vectorized so that it operates over a vector of vertex/edge values instead of traditional scalar values. For PA, the simplest technique is to orchestrate pipelined batch processing of graph snapshots by invoking the processing engine multiple times, each time using values from previous results. However, in order to fully leverage PA, the processing engine itself needs to be modified so that it becomes aware of this pipelined processing technique and feeding of values occurs internally as soon as processing of any graph snapshot ends.

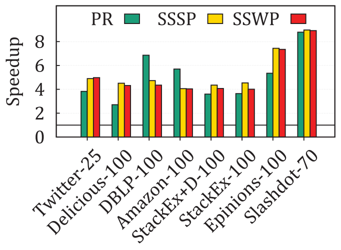
7. EXPERIMENTAL EVALUATION

7.1. Experimental Setup

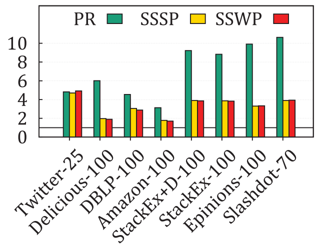
Real-World Datasets. Our experiments use real-world evolving graphs taken from the KONECT [Kunegis 2013] repository. Table VIII lists the input datasets and their sizes that range from 25.5 billion to 7.5 million edges. Similar to as in Han et al. [2014], we develop snapshots by first dividing the entire evolution into half; the first snapshot is this midpoint and then subsequent snapshots are created by batching the remaining edge additions and deletions. The number of snapshots is included in each input's name in Table VIII. We also create a synthetic graph called StackEx+D to simulate edge deletions using StackEx. In this case, the edges to be deleted are randomly selected from a given graph snapshot and the next snapshot is constructed using an equal number of edge deletions and additions.

Benchmarks. To evaluate our techniques, we use a wide range of modern applications (as listed in Table IX) that belong to important domains such as scientific simulation and social network analysis. These applications are based on the vertex-centric model where each vertex iteratively computes a value (e.g., colors for GC, ranks for PR [Page et al. 1999], and shortest paths for SSSP) and the algorithm terminates when values become stable, that is, all the algorithms were run fully until completion.

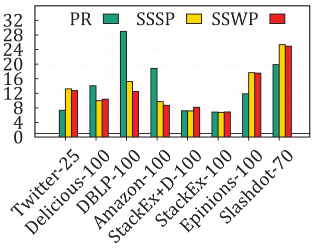
System. The experiments were run on a 16-node cluster running CentOS 6.3, Kernel v2.6.32 with each node having two 8-core Xeon E5-2680 processors (16 cores) operating



(a) Speedups by FA.



(b) Speedups by PA.



(c) Speedups by FA+PA.

Fig. 7. Speedups achieved by FA, PA, and FA+PA in GraphLab. The execution times (in seconds) for PR/SSSP on the original version of GraphLab (Baseline) are 4,117/2,900 for Twitter-25 and 7,148/1,490 for Delicious-100.

at 2.7GHz and 8X4GB (32 GB) DDR3-1600MHz memory. The nodes are connected via 56Gbit/s FDR InfiniBand interconnect.

The first set of performance results (i.e., entire Figure 7) are averaged (arithmetic mean) across 10 runs and since the observed deviation was typically less than 2%, the remaining results are averaged across three runs.

7.2. Performance of FA and PA in GraphLab/ASPIRE

We evaluate the benefits of the proposed techniques in distributed systems, ASPIRE and GraphLab,[3] by comparing the performance of the following versions of the benchmarks:

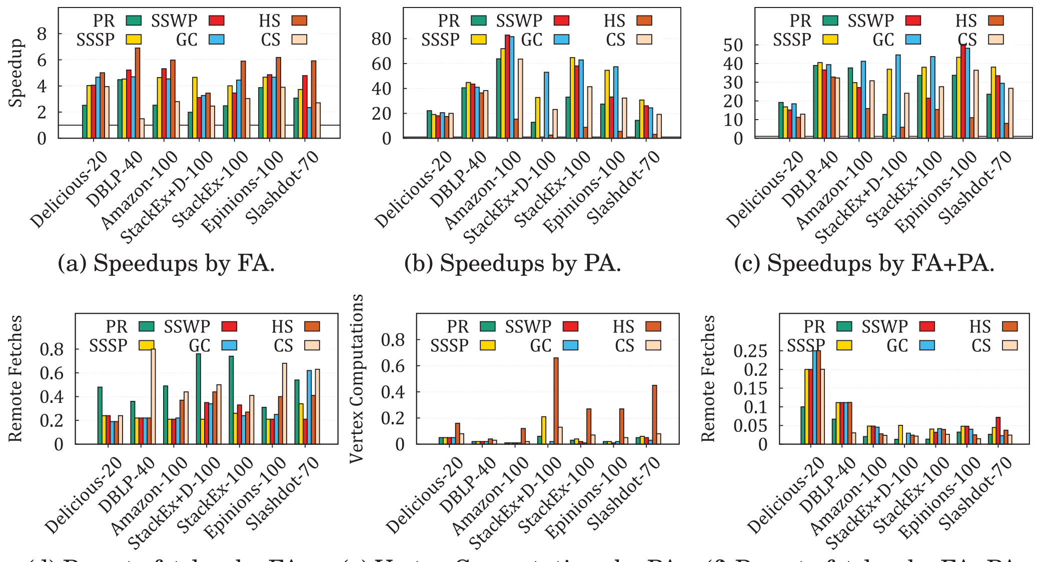
—**GraphLab/ASPIRE** is the original version that does not employ amortization techniques.
—**FA** is the version in GraphLab/ASPIRE that employs fetch amortization.
—**PA** is the version in GraphLab/ASPIRE that employs processing amortization.
—**FA+PA** is the version in GraphLab/ASPIRE that employs both techniques.

We carried out this performance comparison using all six benchmarks in ASPIRE and three benchmarks in GraphLab—PR, SSSP, and SSWP. For FA and FA+PA, we choose the number of snapshots to be simultaneously processed (Δ) to be 10 (more experiments with varying Δ are further presented in Section 7.4.). Since ParMETIS requires a high amount of memory to partition very large graphs, we ran ASPIRE with first 40/20 snapshots for DBLP/Delicious graphs and do not evaluate using the Twitter-25 graph. GraphLab's synchronous engine outperforms its asynchronous engine in all cases and hence, we report the times obtained from its synchronous engine.

Figure 7 and Figure 8 show the speedups achieved by FA, PA, and FA+PA when incorporated in GraphLab and ASPIRE, respectively. On average, FA in GraphLab achieves 5.2$\times$ speedup over original GraphLab and the further addition of PA yields an additional 2$\times$ speedup over FA in GraphLab. Similarly, on average, adding FA to ASPIRE yields a speedup of 4.06$\times$ over the original version of ASPIRE and the further addition of PA results in an additional 7.86$\times$ speedup over FA alone.

Figures 8(d)–8(f) show the reduction in remote fetches and vertex computations performed when amortizations are enabled. Overall, the speedups achieved by the amortization techniques vary across the benchmark-input combinations. However, it is interesting to note that the improvement from FA is usually higher for SSSP and SSWP than for PR even though the remote fetches saved by FA are more for PR than both SSSP and SSWP. On average, FA in ASPIRE gives 4.3/4.4$\times$ speedups over the

[3]We used the latest available version of GraphLab (v2.2).



(a) Speedups by FA. (b) Speedups by PA. (c) Speedups by FA+PA.

(d) Remote fetches by FA. (e) Vertex Computations by PA. (f) Remote fetches by FA+PA.

Fig. 8. Speedups achieved by FA, PA, and FA+PA in ASPIRE ((a)–(c)); reduction in Remote fetches for FA, Vertex Computations for PA, and Reduction in Remote fetches for FA+PA over FA ((d)–(f)). The execution times (in seconds) for PR/SSSP on the original version of ASPIRE (Baseline) are 16,245/7,244 for Delicious-20 and 9,282/2,340 for DBLP-40.

original versions for SSSP/SSWP, whereas for PR the corresponding speedup is 3×. This is because PR is more compute intensive compared to SSSP and SSWP; hence, even though reductions in remote fetches are higher for PR, vertices take more time to stabilize to their final values for PR compared to SSSP and SSWP, leading to more work to be done in each intermediate iteration. Similarly, speedups for HS are lower with PA compared to those with FA. This is because HS is more sensitive to structural changes in the graph and hence, the fed values are less useful, causing a fair amount of work to be performed (as can be seen in Figure 8(e)) to reach stability.

It is interesting to observe that in GraphLab, FA+PA performs better than PA, whereas the trend is reversed in ASPIRE on large graphs. This is due to the fundamental difference in how the remote vertex values are locally maintained in these two systems. In FA+PA, when a new snapshot gets included to be processed with the other snapshots, vertex values for the new snapshot get fed from the previous snapshot. In ASPIRE, this change in value is reflected on remote machines in the subsequent iteration (immediately after the inclusion of the new snapshot) when the values are accessed, which causes the computation to wait for the changed values to become available. For PA, on the other hand, the fed values are the same and hence, remote machines already have correct values in their software caches when processing of the new snapshot begins. The preceding behavior does not occur in GraphLab because remote vertex values are kept consistent with local copies as soon as new edges get added since they represent computation dependencies.

We achieve consistent speedups across all inputs, including both with and without edge deletions. On StackEx+D-100 (which includes edge deletions), on average, FA in GraphLab and ASPIRE gives 3.6× and 3.2× speedups, respectively, over their original versions and the further addition of PA achieves additional speedups of 1.8× and 7.9×, respectively. No data is provided for PA and FA+PA for SSWP on StackEx+D-100 because PA is not applicable on SSWP in the presence of edge removals.

While Figure 7 reports the overall speedups, we also separated the processing times from the evolving graph management times. When only the processing times are considered, on average, FA speeds up processing by 5.3× in GraphLab, while PA achieves additional speedups of 2.6×.

Note that simultaneously processing too few graph snapshots fails to fully realize the benefits of FA while processing excessive numbers of graph snapshots can also lead to suboptimal performance. Simultaneously processing too many graph snapshots causes many vertices to be active even though only some of the snapshots of those vertices are active. This delays processing for a given snapshot because different snapshots of other vertices need to be processed. Therefore, the number of graph snapshots that are simultaneously processed is a parameter that must be tuned for a given distributed environment. As discussed in Section 4.1, the degree up to which multiple graph snapshots can be processed together is bound by available memory. Based upon our experiments by varying the number of snapshots that are simultaneously processed, we found that simultaneously processing 10 snapshots gives good performance overall. Therefore, the experimental data reported previously used these settings when running FA and FA+PA.

From the results, we observe that amortization optimizations are very effective as they significantly improve the performance of both GraphLab and ASPIRE. The results for FA clearly show the need to tolerate communication latencies by amortizing fetches in both GraphLab and ASPIRE; the results for PA clearly show that the amount of computation is significantly reduced by feeding values across snapshots.

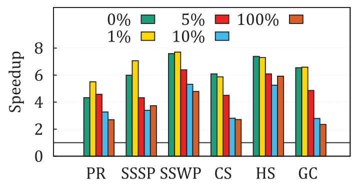
7.3. Sensitivity to Cache Size

Both caching and amortization optimizations reduce remote fetches. Hence, it is important to study their interplay and how the benefits achieved from the amortization techniques are affected when caching is available in the system. In order to study how the benefits of amortizations vary with caching, we experiment using ASPIRE and vary the machine-level software cache sizes to be 100%, 10%, 5%, and 1% of the number of vertices as well as when no cache is used. The ASPIRE runtime incorporates machine-level caches in order to make the remote vertices locally available for processing. The cache coherence protocol provides strict consistency [Goodman 1991] and cache invalidates are piggybacked with updated vertex values.

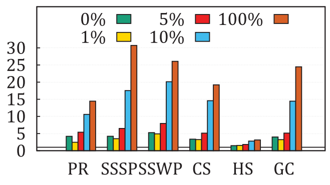
Since in the previous section we observed that the benefits achieved from FA and PA are similar across all inputs, we now select one of the inputs (Slashdot) to further perform this sensitivity study in detail. Finally, we ran all six applications in Table IX for each configuration.

From Figure 9(a), we first observe that, although both caching and FA can reduce remote fetches, they complement each other quite well. Consider the case where the cache size is 100%. When FA uses 10 snapshots plus caching, it provides speedups ranging from 2.35× to 5.92× over the ASPIRE Baseline[4] with caching enabled. For the 10%, 5%, and 1% cache size, the corresponding speedups range from 2.79× to 5.32×, 4.32× to 6.39×, and 5.5× to 7.7×. In other words, FA provides substantial speedups *beyond caching* by reducing remote fetches. The main reason for these speedups is the significant number of remote fetches saved by FA (shown in Figure 9(d)); as cache size decreases from 100% to 0%, the average savings increase from 54% to 82%. It is interesting to note that the speedups with FA are higher with 1% compared to other cases which is due to the inefficiency of the small cache which provides a higher miss rate; on average, cache with size 1% provides an 84.56% miss rate compared to the 34.78% to 10.12% miss rate provided by caches with size 5% to 100%. Hence, the

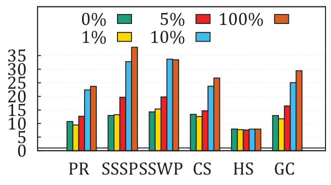
[4]Baseline refers to the ASPIRE version without FA and PA.



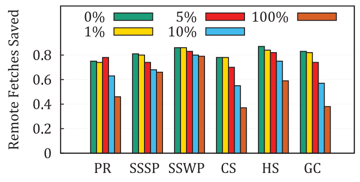
(a) Speedups by FA.



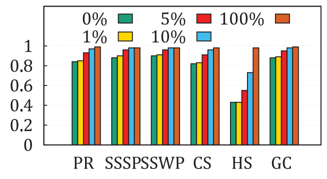
(b) Speedups by PA.



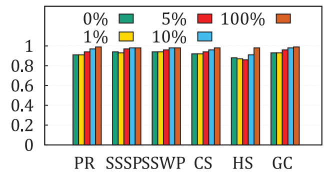
(c) Speedups by FA+PA.



(d) Remote fetches saved by FA.



(e) Remote fetches saved by PA.



(f) Remote fetches saved by FA+PA.

Fig. 9. Effect of varying cache size on FA, PA, and FA+PA ((a)–(c)); Remote fetches saved by FA, PA, and FA+PA ((d)–(f)).

overheads of maintaining a smaller cache of 1% size effectively enables a larger scope for savings that are captured by FA.

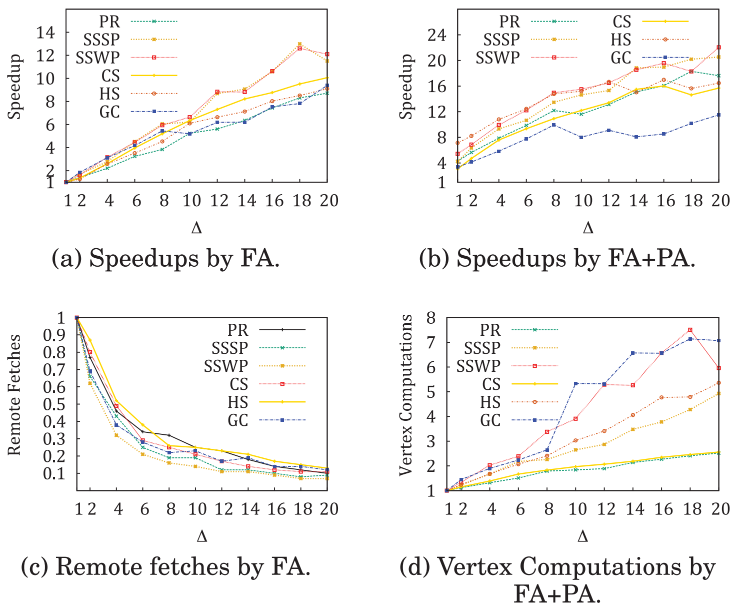
Alternatively, we also find that the 10% cache size is enough to capture most of the benefits of caching—further increasing the cache size from 10% to 100% would reduce the average miss rates only from 14.82% to 10.12% for the Baseline algorithm. The corresponding average speedups of FA plus caching over Baseline plus caching increase from 3.7× to 3.8× as the cache size decreases from 100% to 10%; moreover, this 10× increase in cache size does not yield many additional benefits. Note that the 5% cache size, on the other hand, is not enough to capture benefits comparable to that with the 10% cache size. The average hit rate for the Baseline algorithm with the 5% cache size is 34.78%, and hence, the average speedups of FA plus caching over Baseline plus caching increase to 5.12×.

Figure 9(b) shows the speedups achieved by PA with varying cache sizes. The speedups increase with increase in cache sizes; this is mainly because of the elimination of redundant computations by PA which makes the computation more communication bound, allowing larger caches to provide more speedups by saving more remote fetches (shown in Figure 9(e)). Again, the overheads of a smaller cache with size 1% along with a higher miss rate reduces the speedups achieved compared to other cases.

Finally, as seen in Figure 9(c) FA+PA provides the highest speedups with different cache sizes mainly because of the very high amount of remote fetches saved (as shown in Figure 9(f)) by both the amortization strategies.

7.4. Sensitivity to Number of Snapshots (Δ)

We also varied the number of snapshots processed simultaneously, that is, Δ from one to 20. As discussed in Section 4.1, the degree up to which multiple graph snapshots can be processed together is bound by available memory. Hence, we do not go beyond simultaneously processing 20 snapshots. We again select Slashdot and use all six benchmarks to perform this sensitivity study. In the first set of experiments, we disabled the machine-level caches to decouple the effects of caching from our amortization techniques. Later, we enabled the machine-level caches and vary both the number of snapshots and the cache sizes, to study the interplay between them.



(a) Speedups by FA. (b) Speedups by FA+PA.

(c) Remote fetches by FA. (d) Vertex Computations by FA+PA.

Fig. 10. Effect of varying Δ on FA (a) and FA+PA (b); Reduction in Remote Fetches for FA (c); and Increase in Vertex Computation for FA+PA (d).

The effects of varying Δ on the overall performance are shown in Figure 10. Clearly, our approach scales well with the number of snapshots. Figure 10(a) shows that as Δ increases, the benefits of FA show up. The reduction in remote fetches achieved by FA in those cases is shown in Figure 10(c); as we can see, the speedups are mainly achieved due to this reduction.

It is interesting to note that bulk of the reduction in remote fetches is achievable when $\Delta = 8$. Hence, we observe nearly linear speedup for FA up until $\Delta = 8$. Beyond $\Delta = 8$, some benchmarks exhibit a sawtooth trend where the benefits of increasing Δ by small steps are not clearly visible. The reason for this is the decline in reduction of remote fetches for corresponding intermediate Δ steps beyond $\Delta = 8$, which can be seen in Figure 10(c).

Figure 10(b) shows the benefits achieved by FA+PA with an increase in Δ. As we already know, PA eliminates redundant computations and FA with increased Δ reduces the overall remote fetches performed. It is interesting to note from Figure 10(d) that as Δ increases, the amount of processing required increases compared to PA with $\Delta = 1$. This is mainly because for $\Delta > 1$, unstable values are fed across snapshots, and these unstable values are further away from the final values for new snapshots, hence requiring more work to stabilize these newly fed snapshots. However, this increase in processing is overshadowed by savings in remote fetches, hence resulting in an overall increase in speedups with an increase in Δ.

Note that the speedups for GC nearly flatten out for higher Δ values because of the high impact of structural changes across snapshots that are far apart in evolution resulting in more color changes across the graph.

Finally, Figure 11(a) and Figure 11(b) show the execution times for PR for varying Δ with different cache sizes on FA and FA+PA, respectively, normalized with respect to $\Delta = 1$ without PA. As we can see, the performance benefits are achieved in all cases with different Δ and cache size combinations. As expected, increasing Δ reduces the overall execution time; again, the majority of performance benefits are achieved

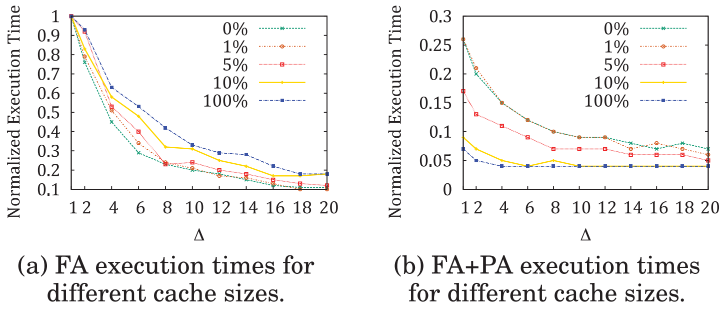


Fig. 11. Effect of varying Δ for different cache sizes on FA (a) and FA+PA (b).

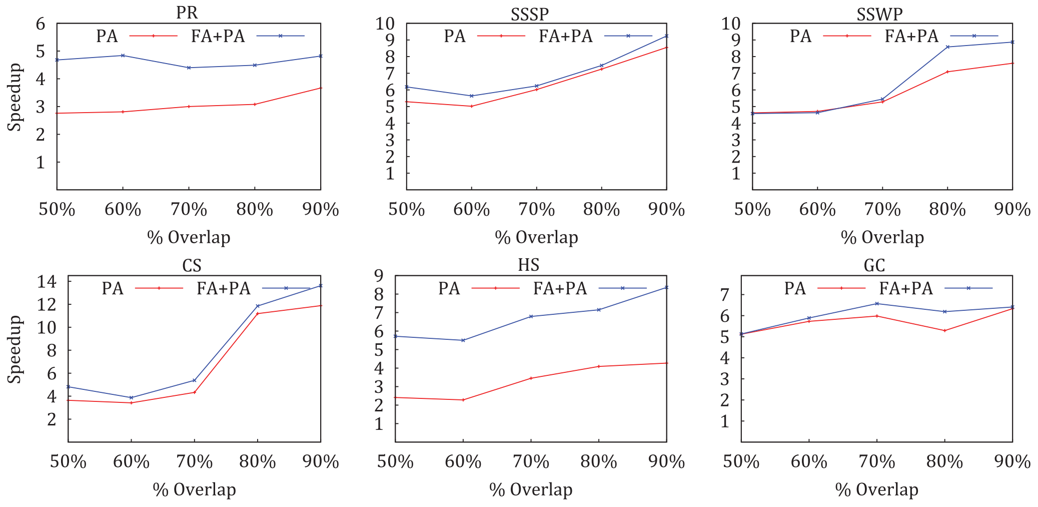


Fig. 12. Effect of varying the overlap across consecutive snapshots on PA and FA+PA.

when Δ is increased up to a certain point, after which, lesser additional benefits are seen—in particular, while the performance continues to improve beyond Δ = 8–10, the improvement is not significant. This means, the difference in execution times for higher values of Δ is much less and hence, we can achieve a majority of the performance benefits from our techniques without determining the ideal Δ value for each of the cache sizes.

7.5. Sensitivity to Similarity in Snapshots

While PA is useful to eliminate redundant computations by feeding values across snapshots, we study its effectiveness when consecutive graph snapshots are made structurally dissimilar. In this experiment, we vary the percentage of overlapping edges, that is, edges that are the same, across consecutive snapshots from 90% (high overlap) to 50% (low overlap) and measure the speedups achieved by PA with and without FA (Δ = 10). We again select Slashdot and use all six benchmarks to perform this sensitivity study. Since we are interested in the effectiveness of the amortization strategy itself, we enabled machine-level caches to hold the required remote vertices on each machine.

The effects of varying the percentage of overlapping edges are shown in Figure 12. As we can see, PA accelerates the overall processing in all cases; even if consecutive

Table X. Amortization Techniques vs. Chronos

Aspect	Chronos	Amortization Techniques
Temporal Layout	Enhance locality for cache performance	Expose structural similarity to aggregate remote fetches
Snapshots	Static	Dynamic
Batch Processing	Fixed batching	Variable batching using pipelining
Feeding Unstable Values	Absent	Present
Message Aggregation	Absent	Present

snapshots are structurally dissimilar by up to 50%, feeding of values provides a better initialization by eliminating redundant computations and hence, convergence is achieved faster. As expected, the speedups increase with an increase in structural overlap mainly because the fed values are closeer to the final results and hence, require less computation. The speedups do not increase as clearly for PR and GC as compared to those of SSSP, SSWP, CS, and HS when the overlap increases from 50% to 90%. This is because the number of remote fetches performed by PR and GC without the amortization techniques is high enough to minimize the difference between relative reduction achieved using the amortization techniques as similarity varies.

7.6. Comparison with Chronos

Table X highlights the major differences between our amortization techniques and Chronos's [Han et al. 2014] processing techniques. While Chronos incorporates batch scheduling and incremental processing, it is designed to improve the L1 data cache and last level cache performance in a shared memory environment. As shown in Han et al. [2014], the benefits are lower in a distributed setting (an average speedup of 4.06×) mainly because it is constrained by the network overheads. Our amortization techniques, on the other hand, collectively reduce the network overheads and provide average speedups of 28.9× and 10.3× in ASPIRE and GraphLab.

Since Chronos itself is a temporal graph processing system, we directly compare the performance of PA with the *Incremental Processing* (IP) technique. In the presence of FA, IP is performed by feeding stable values from the most recent snapshot in the previous batch to all the snapshots in the next batch. This requires Chronos to wait for all the snapshots in the previous batch to finish before the next batch can begin execution. PA, on the other hand, enables faster processing by feeding unstable (partially computed) values that are available from the most recent snapshot. This allows the batch of snapshots being processed to be dynamically adjusted as snapshots finish processing, eliminating the need to stall processing of future snapshots.

We incorporated IP in ASPIRE by processing snapshots batch after batch so that only final results from the most recent snapshots in the previous batch are fed to all the snapshots in the next batch. Figure 13 shows the execution times for PR using FA+PA normalized with respect to that using FA+IP across multiple evolving graphs created from StackEx input by varying the number of edge updates required to determine new snapshots. On average, FA+PA performs 16.5% faster than FA+IP mainly because PA does not stall processing of snapshots, which allows those snapshots to be computed in parallel as other snapshots are being processed.

8. RELATED WORK

The related works on evolving graphs consider two major scenarios: offline mining of evolving graphs representing historical data; and online processing of evolving graphs for continuous query processing.

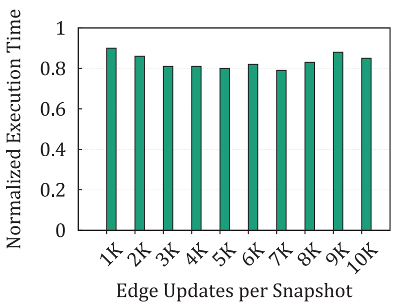


Fig. 13. Execution times for PR using FA+PA normalized with respect to FA+IP.

Mining of evolving graphs. This work, similar to our work, considers the scenario where an evolving graph is represented as a sequence of snapshots that are then analyzed. Chronos [Han et al. 2014] is a storage and execution engine to process temporal graphs. It employs a graph representation that places vertex data from different snapshots together. It exploits the cache locality created by the graph representation by interleaved (parallel) processing of snapshots in a single- and multithreaded environment leading to improved L1 data cache and last level cache performance. Both Chronos and FA exploit synergy between different graph snapshots by collocating vertex values, *in memory* versus *in messages*, respectively. In the presence of FA, Chronos performs *incremental processing* by feeding stable values from the most recent snapshot in the previous batch to all the snapshots in the next batch. This requires Chronos to wait for all the snapshots in the previous batch to finish before the next batch can begin execution. In comparison, our PA allows faster processing by feeding potentially unstable (partially computed) values that are available from the most recent snapshot. In addition, in our work we also consider the interplay between FA, PA, and caching in distributed and shared-memory environments. GraphInc [Cai et al. 2012] is a system that allows incremental processing of graphs by employing memorization of incoming messages to remove redundant vertex computations for same sets of messages.

GraphScope [Sun et al. 2007] focuses on the encoding of time-evolving graphs to efficiently perform community discovery and change detection. Kan et al. [2009] present indexing techniques for detecting simple spatiotemporal patterns in evolving graphs—unlike our work, these queries are simple and do not involve iterative algorithms. TEG [Fard et al. 2012] focuses on partitioning time-evolving graphs across nodes of a cluster for distributed processing and implements reachability and subgraph queries. In Khurana and Deshpande [2013], a distributed graph database system is developed to manage historical data for evolving graphs, supporting temporal queries and analysis. Many *algorithmic approaches* are being developed for improving efficiency. Ren et al. [2011] propose the *Find-Verify-Fix* approach where from a sequence of snapshots a representative snapshot is constructed. To process a query, first representative snapshot is used and if there is success, the real snapshots are queried to verify and fix the response. Like our work, the Find-Verify-Fix approach also recognizes the synergy between analysis performed on different snapshots. However, this approach is effective when only the structural properties of graphs are being considered. Desikan et al. [2005] propose an incremental PageRank algorithm for evolving graphs that partitions graph into a part whose PageRank values will remain unchanged and the remaining subgraph. This algorithm exploits the underlying principle of a first-order Markov model on which PageRank is based. In contrast, our Fetch Amortization is applicable and effective across a variety of mining tasks.

Online query processing for streaming graphs. Continuous real-time query processing over streaming data poses unique challenges, different from those faced while processing evolving graphs. Yuan et al. [2013] perform evolution aware clustering to partition a streaming graph that favors recent edges over older edges during splitting and merging of clusters. Suzumura et al. [2014] propose the GIM-V incremental graph processing model based upon matrix-vector operations. STINGER [Ediger et al. 2012] proposes an efficient representation for evolving graphs to enable fast real-time query processing on shared memory systems. Kineograph [Cheng et al. 2012] is a distributed in-memory graph storage system that supports incremental iterative propagation-based graph mining to operate on streaming graph. Finally, for real-time query processing, approaches providing approximate results are explored (e.g., techniques in Bahmani et al. [2012] provide approximate PageRank values for an evolving graph).

Other graph processing frameworks. Various other graph processing systems have been developed including Google's Pregel [Malewicz et al. 2010], GraphLab [Low et al. 2012], PowerGraph [Gonzalez et al. 2012], GraphX [Gonzalez et al. 2014], ASPIRE [Vora et al. 2014], Cyclops [Chen et al. 2014], GraphChi [Kyrola et al. 2012], X-Stream [Roy et al. 2013], Ligra [Shun and Blelloch 2013], Galois [Nguyen et al. 2013], and optimizations like Kusum et al. [2016] and Vora et al. [2016]. Beyond these, work has been done in the area of graph databases like Neo4J [2012], OrientDB [2012], FlockDB [Pointer et al. 2010], and InfoGrid [2016]. Moreover, similar techniques have been explored in other domains like in Liu et al. [2012] where an auxiliary system of equations is first solved to provide good initial guesses.

9. CONCLUSION

We accelerated evolving graph processing using two optimizations: *Fetch Amortization* reduces remote fetches while *Processing Amortization* accelerates termination of iterative algorithms. Our experiments with multiple real evolving graphs and graph algorithms on a 16-node cluster demonstrate that, on average fetch amortization speeds up the execution of GraphLab and ASPIRE by 5.2× and 4.1×, respectively. Amortizing the processing cost yields additional average speedups of 2× and 7.9×, respectively.

REFERENCES

Bahman Bahmani, Ravi Kumar, Mohammad Mahdian, and Eli Upfal. 2012. PageRank on an evolving graph. In *KDD*. 24–32.

Michael G. Burke and Barbara G. Ryder. 1990. A critical analysis of incremental iterative data flow analysis algorithms. *IEEE Trans. Softw. Eng.* 16, 7 (July 1990), 723–728.

Zhuhua Cai, Dionysios Logothetis, and Georgos Siganos. 2012. Facilitating real-time graph mining. In *CloudDB*. 1–8.

Meeyoung Cha, Hamed Haddadi, Fabrcio Benevenuto, and Krishna P. Gummadi. 2010. Measuring user influence in twitter: The million follower fallacy. In *ICWSM*.

Rong Chen, Xin Ding, Peng Wang, Haibo Chen, Binyu Zang, and Haibing Guan. 2014. Computation and communication efficient graph processing with distributed immutable view. In *HPDC*. 215–226.

Raymond Cheng, Ji Hong, Aapo Kyrola, Youshan Miao, Xuetian Weng, Ming Wu, Fan Yang, Lidong Zhou, Feng Zhao, and Enhong Chen. 2012. Kineograph: Taking the pulse of a fast-changing and connected world. In *EuroSys*. 85–98.

Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: Simplified data processing on large clusters. *Commun. ACM* 51, 1 (2008), 107–113.

DeliciousUI 2014. Delicious user-url network dataset. In *KONECT*.

Prasanna Desikan, Nishith Pathak, Jaideep Srivastava, and Vipin Kumar. 2005. Incremental page rank computation on evolving graphs. In *WWW*. 1094–1095.

David Ediger, Robert McColl, Jason Riedy, and David A. Bader. 2012. STINGER: High performance data structure for streaming graphs. In *HPEC*. 1–5.

Ayman Farahat, Thomas LoFaro, Joel C. Miller, Gregory Rae, and Lesley A. Ward. 2005. Authority rankings from HITS, PageRank, and SALSA: Existence, uniqueness, and effect of initialization. *SIAM J. Sci. Comput.* 27, 4 (2005), 1181–1201.

Arash Fard, Amir Abdolrashidi, Lakshmish Ramaswamy, and John A. Miller. 2012. Towards efficient query processing on massive time-evolving graphs. In *CollaborateCom*. 567–574.

Vicenç Gómez, Andreas Kaltenbrunner, and Vicente López. 2008. Statistical analysis of the social network and discussion threads in slashdot. In *WWW*. 645–654.

Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. 2012. PowerGraph: Distributed graph-parallel computation on natural graphs. In *OSDI*. 17–30.

Joseph E. Gonzalez, Reynold S. Xin, Ankur Dave, Daniel Crankshaw, Michael J. Franklin, and Ion Stoica. 2014. GraphX: Graph processing in a distributed dataflow framework. In *OSDI*. 599–613.

James R. Goodman. 1991. *Cache Consistency and Sequential Consistency*. University of Wisconsin-Madison, CS Department, 567–574.

Wentao Han, Youshan Miao, Kaiwei Li, Ming Wu, Fan Yang, Lidong Zhou, Vijayan Prabhakaran, Wenguang Chen, and Enhong Chen. 2014. Chronos: A graph engine for temporal graph analysis. In *EuroSys*. 1:1–1:14.

InfoGrid. 2016. Homepage. Retrieved from http://infogrid.org/.

Andrey Kan, Jeffrey Chan, James Bailey, and Christopher Leckie. 2009. A query based approach for mining evolving graphs. In *AusDM*. 139–150.

George Karypis, Kirk Schloegel, and Vipin Kumar. 1997. *Parmetis: Parallel Graph Partitioning and Sparse Matrix Ordering Library.* Department of Computer Science, University of Minnesota.

Udayan Khurana and Amol Deshpande. 2013. Efficient snapshot retrieval over historical graph data. In *ICDE*. 997–1008.

Jérôme Kunegis. 2013. KONECT: The Koblenz network collection. In *WWW Companion*. 1343–1350.

Amlan Kusum, Keval Vora, Rajiv Gupta, and Iulian Neamtiu. 2016. Efficient processing of large graphs via input reduction. In *HPDC*. 245–257.

Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. 2012. GraphChi: Large-scale graph computation on just a PC. In *OSDI*. 31–46.

Jure Leskovec, Jon Kleinberg, and Christos Faloutsos. 2005. Graphs over time: Densification laws, shrinking diameters and possible explanations. In *SIGKDD*. 177–187.

Michael Ley. 2002. The DBLP computer science bibliography: Evolution, research issues, perspectives. In *SPIRE*. 1–10.

Wen-Yew Liang, Chun ta King, and Feipei Lai. 1996. Adsmith: An efficient object-based distributed shared memory system on PVM. In *ISPAN*. 173–179.

Ee-Peng Lim, Viet-An Nguyen, Nitin Jindal, Bing Liu, and Hady Wirawan Lauw. 2010. Detecting product review spammers using rating behaviors. In *CIKM*. 939–948.

Xing Liu, Edmond Chow, Karthikeyan Vaidyanathan, and Mikhail Smelyanskiy. 2012. Improving the performance of dynamical simulations via multiple right-hand sides. In *IPDPS*. 36–47.

Yucheng Low, Danny Bickson, Joseph Gonzalez, Carlos Guestrin, Aapo Kyrola, and Joseph M. Hellerstein. 2012. Distributed GraphLab: A framework for machine learning and data mining in the cloud. *PVLDB* 5, 8 (April 2012), 716–727.

Grzegorz Malewicz, Matthew H. Austern, Aart J. C. Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. 2010. Pregel: A system for large-scale graph processing. In *SIGMOD*. 135–146.

Paolo Massa and Paolo Avesani. 2005. Controversial users demand local trust metrics: An experimental study on epinions.com community. In *AAAI*. 121–126.

Neo4J. 2012. Neo4J. *Graph NoSQL Database [Online]* (2012). https://neo4j.com/.

Donald Nguyen, Andrew Lenharth, and Keshav Pingali. 2013. A lightweight infrastructure for graph analytics. In *SOSP*. 456–471.

OrientDB. 2012. OrientDB. *Hybrid Document-Store and Graph NoSQL Database [Online].* http://orientdb.com/.

Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. 1999. The PageRank citation ranking: Bringing order to the web. Stanford InfoLab.

Robey Pointer, N. Kallen, E. Ceaser, and J. Kalucki. 2010. Introducing FlockDB. https://blog.twitter.com/2010/introducing-flockdb.

Filippo Radicchi, Santo Fortunato, Benjamin Markines, and Alessandro Vespignani. 2009. Diffusion of scientific credits and the ranking of scientists. *Phys. Rev. E* 80, 5 (2009), 056103.

Chenghui Ren, Eric Lo, Ben Kao, Xinjie Zhu, and Reynold Cheng. 2011. On querying historical evolving graph sequences. *PVLDB* 4, 11 (2011), 726–737.

Ryan A. Rossi, Brian Gallagher, Jennifer Neville, and Keith Henderson. 2013. Modeling dynamic behavior in large evolving graphs. In *WSDM*. 667–676.

Amitabha Roy, Ivo Mihailovic, and Willy Zwaenepoel. 2013. X-Stream: Edge-centric graph processing using streaming partitions. In *SOSP*. 472–488.

Julian Shun and Guy E. Blelloch. 2013. Ligra: A lightweight graph processing framework for shared memory. In *PPOPP*. 135–146.

StackExchange. Stack Exchange Inc. Stack Exchange Data Explorer. Retrieved from http://data.stackexchange.com/.

Jimeng Sun, Christos Faloutsos, Spiros Papadimitriou, and Philip S. Yu. 2007. GraphScope: Parameter-free mining of large time-evolving graphs. In *KDD*. 687–696.

Toyotaro Suzumura, Shunsuke Nishii, and Masaru Ganse. 2014. Towards large-scale graph stream processing platform. In *WWW Companion*. 1321–1326.

Keval Vora, Sai Charan Koduru, and Rajiv Gupta. 2014. ASPIRE: Exploiting asynchronous parallelism in iterative algorithms using a relaxed consistency based DSM. In *OOPSLA*. 861–878.

Keval Vora, Guoqing Xu, and Rajiv Gupta. 2016. Load the edges you need: A generic I/O optimization for disk-based graph processing. In *USENIX ATC*.

Duncan J. Watts and Steven H. Strogatz. 1998. Collective dynamics of small-world networks. *Nature* 393, 6684 (June 1998), 440–442.

Christo Wilson, Bryce Boe, Alessandra Sala, Krishna P. N. Puttaswamy, and Ben Y. Zhao. 2009. User interactions in social networks and their implications. In *Proceedings of the 4th ACM European Conference on Computer Systems*. 205–218.

Lei Yang, Lei Qi, Yan-Ping Zhao, Bin Gao, and Tie-Yan Liu. 2007. Link analysis using time series of web graphs. In *Proceedings of the 16th ACM Conference on Information and Knowledge management*. 1011–1014.

Mindi Yuan, Kun-Lung Wu, Gabriela Jacques-Silva, and Yi Lu. 2013. Efficient processing of streaming graphs for evolution-aware clustering. In *CIKM*. 319–328.

Xiaojin Zhu and Zoubin Ghahramani. 2002. *Learning from Labeled and Unlabeled Data with Label Propagation*. Technical Report CMU-CALD-02-107, Carnegie Mellon University.

Received March 2016; revised August 2016; accepted August 2016