KickStarter: Fast and Accurate Computations on Streaming Graphs via Trimmed Approximations

Keval Vora
University of California, Riverside
kvora001@cs.ucr.edu

Rajiv Gupta
University of California, Riverside
gupta@cs.ucr.edu

Guoqing Xu
University of California, Irvine
guoqingx@ics.uci.edu

Abstract

Continuous processing of a streaming graph maintains an approximate result of the iterative computation on a recent version of the graph. Upon a user query, the accurate result on the current graph can be quickly computed by feeding the approximate results to the iterative computation — a form of *incremental computation* that corrects the (small amount of) error in the approximate result. Despite the effectiveness of this approach in processing *growing graphs*, it is generally not applicable when *edge deletions* are present — existing approximations can lead to either incorrect results (*e.g.*, monotonic computations terminate at an incorrect minima/maxima) or poor performance (*e.g.*, with approximations, convergence takes longer than performing the computation from scratch).

This paper presents *KickStarter*, a runtime technique that can *trim the approximate values* for a subset of vertices impacted by the deleted edges. The trimmed approximation is both safe and profitable, enabling the computation to produce correct results and converge quickly. KickStarter works for a class of *monotonic* graph algorithms and can be readily incorporated in any existing streaming graph system. Our experiments with four streaming algorithms on five large graphs demonstrate that *trimming* not only produces correct results but also accelerates these algorithms by **8.5–23.7**×.

Keywords Graph Processing, Value Dependence, Streaming Graphs

1. Introduction

Real-world graphs that are constantly changing are often referred to as streaming graphs. Their examples include social networks and maps with real-time traffic information. To provide timely responses to online data analytic queries, these graphs are continuously processed via incremental algorithms as they change. The need to analyze streaming graphs has led to the development of systems such as Tornado [32], Kineograph [8], Stinger [12], Naiad [23], and others.

The core idea of these systems is to interleave iterative processing with the application of batches of updates to the graph. The iterative processing maintains an *intermediate approximate result* (intermediate for short) of the computation on the *most recent* version of the graph. When a query arrives, the accurate result for the *current version* of the graph where all batched updates have been applied is obtained by performing iterative computation starting at the intermediate results. In other words, computations at the vertices with edge updates are performed directly on their most recent intermediate values computed before the updates arrive.

This style of processing leverages *incremental computation* to achieve efficiency. The intuition behind it is straightforward: the values right before the updates are a better (closer) approximation of the actual results than the initial vertex values and, hence, it is quicker to reach convergence if the computation starts from the approximate values.

Problems However, the above intuition has an implicit assumption that is often overlooked: an intermediate value of a vertex is indeed closer to the actual result than the initial value *even when the graph mutates*. We observe that this assumption always holds for strictly growing graphs if the graph algorithm performs a *monotonic computation* (*e.g.*, SSSP, BFS, Clique, label propagation algorithms, *etc.*), because adding new edges preserves the existing graph structure on which intermediate values were computed. However, if graph is mutated via edge deletions, the graph structure changes may *break monotonicity* and invalidate the intermediate values being maintained.

It is not uncommon for real streaming graphs to have both edge additions and deletions. For example, analytics such as product recommendation or influential user tracking over social network graphs [8, 18] are typically performed over sliding windows of graph states, which may involve both addition and deletion of several edges. As another example, spatial-temporal road networks have time-dependent edge

ASPLOS '17, April 08-12, 2017, Xi'an, China.

DOI: http://dx.doi.org/10.1145/3037697.3037748

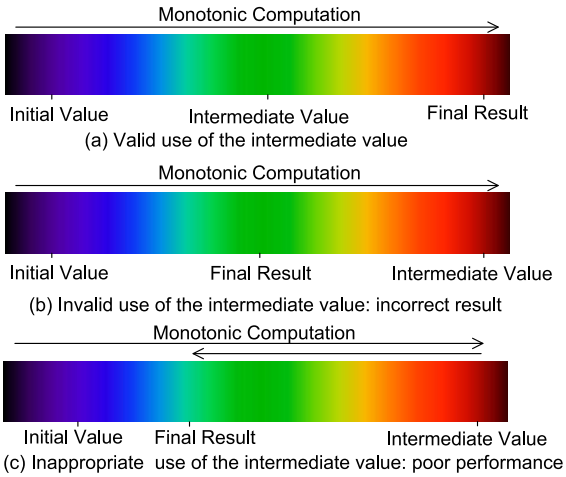


Figure 1: Three different scenarios *w.r.t.* the use of intermediate values after an edge update.

weights. Changing weight on an edge is typically modeled as an edge deletion followed by addition of the same edge with a different weight [9, 19]. How to correctly and efficiently process graphs in the presence of constant edge deletions is an important problem that none of the existing techniques have addressed.

Figure 1 depicts three scenarios *w.r.t.* the use of approximate results in the processing of streaming graphs. Since we focus on monotonic graph algorithms, each spectrum shows the unidirectional change of vertex values. Let us examine how the computation is impacted by the use of intermediate values. In Figure 1(a), the intermediate result is between the initial value and the final accurate result. This scenario is a valid use of intermediate result because it is closer to the final result than the initial value. Performing monotonic computations on a strictly growing graph falls in this category.

Figure 1(b), however, shows an opposite scenario where the final result is between the initial and the intermediate values. An edge deletion may fall into this category. To illustrate this situation, consider a path discovery algorithm where the intermediate path computed before the deletion (*i.e.*, intermediate result) no longer exists and the new path to be discovered (*i.e.*, final result) is "worse" than the intermediate path. If the algorithm only updates the vertex value (*i.e.*, path discovered) when a new "better" path is found, the algorithm will stabilize at the non-existent old (better) path and converge to an *incorrect* result.

Figure 1(c) shows a slightly different scenario than Figure 1(b) where the algorithm, despite being monotonic, is also self-healing. In this case, the computation goes "backward" after an edge deletion and finally stabilizes at the correct result. However, starting the computation at the intermediate result is clearly *unprofitable*. It would have taken much less effort to reach the correct result had the computation started at the initial value after edge deletion. Detailed examples illustrating these cases will be presented in §2.

It may appear that the problem can be solved by always resetting the value of a vertex to its initial value at the moment one of its incoming edges is deleted and making its computation start from scratch. This approach would still lead to incorrect results because computations at many other vertices are *transitively* dependent upon the deleted edge; thus, only resetting the value at the deleted edge does not handle these other vertices appropriately (*cf.* §2). Resetting *all* vertex values solves the problem at the cost of completely disabling incremental computation and its benefits.

Our Approach In this paper, we present a novel runtime technique called KickStarter that computes a *safe* and *profitable* approximation (*i.e.*, trimmed approximation) for a small set of vertices upon an edge deletion. KickStarter is the *first technique* that can achieve safety and profitability for a general class of monotonic graph algorithms, which compute vertex values by performing *selections* (discussed shortly). After an edge deletion, computation starting at the trimmed approximation (1) produces correct results and (2) converges at least at the same speed as that starting at the initial value.

The key idea behind KickStarter is to *identify* values that are (directly or transitively) impacted by edge deletions and *adjust* those values before they are fed to the subsequent computation. A straightforward way to do so is to *tag* the target vertices of the deleted edges and to carefully propagate the tags to the rest of graph. The values for all the tagged vertices are reset (to the initial value) to ensure correctness.

Although tagging guarantees correctness, it is performed conservatively as it is unaware of how the intermediate results are dynamically computed. Hence, it typically tags vertices excessively, leaving only a small set of vertices with usable approximate values. To overcome this drawback, KickStarter characterizes the *dependences* among values being computed and tracks them actively as the computation progresses. However, tracking dependences online can be very expensive; how to perform it efficiently is a significant challenge.

We overcome this challenge by making an observation on monotonic algorithms. In many of these algorithms, the value of a vertex is often *selected* from one single incoming edge, that is, the vertex's update function is essentially a *selection function* that compares values from all of the incoming edges (using max, min, or other types of comparisons) and selects one of them as the computed value of the vertex. This observation applies to all monotonic algorithms that we are aware of, including the eight algorithms listed later in Table 1. This feature indicates that the current value of a vertex only *depends on* the value of *one single* in-neighbor, resulting in simpler dependences that can be efficiently tracked.

Upon an edge deletion, the above dependence information will be used first to find a small set of vertices impacted by the deleted edges. It will also be used to compute safe approximate values for these vertices. The detailed explanation

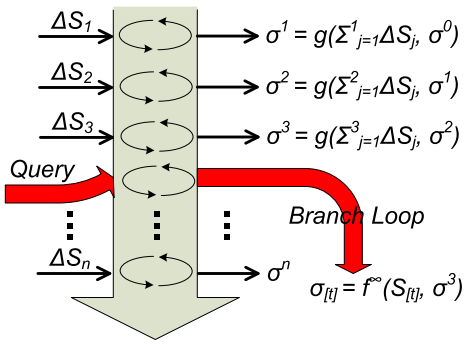


Figure 2: Streaming graph processing.

```
1: function SSWP(Vertex v)
2:     maxPath ← 0
3:     for e ∈ INEDGES(v) do
4:         p ← MIN(e.src.path,
5:               e.weight)
6:         if p > maxPath then
7:             maxPath ← p
8:         end if
9:     end for
10:    v.path ← maxPath
11: end function
```

(a) Single source widest path.

```
1: function SSSP(Vertex v)
2:     minPath ← ∞
3:     for e ∈ INEDGES(v) do
4:         p ← e.src.path+
5:               e.weight
6:         if p < minPath then
7:             minPath ← p
8:         end if
9:     end for
10:    v.path ← minPath
11: end function
```

(b) Single source shortest path.

Figure 3: Two path discovery algorithms.

of the dependence tracking and the trimming process can be found in §3. We have evaluated KickStarter using four monotonic algorithms and five large real-world graphs. Our results show that KickStarter not only produces correct results, but also accelerates existing processing algorithms such as Tornado [32] by **8.5–23.7**×.

2. Background and Motivation

Background In a typical streaming iterative graph processing system such as Tornado [32], the implementation employs a main loop that continuously and iteratively processes the changing graph to compute intermediate results for the most recent snapshot of the graph. Figure 2 illustrates this processing loop. While graph update requests constantly flow in, updates are batched (ΔS_i) and not applied until the end of an iteration. Upon a user query (for a certain property of the graph), the main loop forks a *branch loop* that uses the intermediate values computed at the end of the previous completed iteration of the main loop (*e.g.*, σ^3 in Figure 2) as its starting values. The branch loop then iteratively processes the graph until the computation converges. The final results are then returned to the user.

2.1 Problem 1: Incorrectness

We use a Single Source Widest Path (SSWP) example to show that naïvely using the above algorithm in presence of *edge deletions* can lead to incorrect results. SSWP solves the problem of finding a path between the designated source vertex and every other vertex in a weighted graph, maximizing the weight of the minimum-weight edge in the path. It has many applications in network routing where the weight of an edge represents the bandwidth of a connection between two routers. The algorithm can be used to find an end-to-end path between two Internet nodes that has the maximum possible bandwidth.

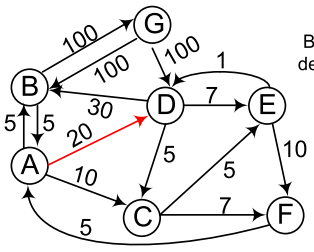
Figure 3(a) illustrates a vertex-centric implementation of SSWP. Next, we show that, for the simple graph given in Figure 4(a), feeding the computation with either the intermediate value or the initial value in presence of deletion of edge $A \rightarrow D$ generates incorrect results. Figure 4(b) reports the value of each vertex before and after deletion when the approximate value is used (*i.e.*, 20) for vertex D. Before the edge update, $A \rightarrow D$ is the key edge that contributes to the value 20 at D and G. After it is deleted, G and E become the only in-neighbors of D. Since G's value is still 20, D's value is not updated; so are the values of the other vertices. The computation stabilizes at the pre-deletion values, generating incorrect results.

Figure 4(c) shows that resetting the value of D to its initial value 0 does not solve the problem either. Clearly, despite the change, D's value will be incorrectly updated back due to the influence from G. The reason behind these problems is that the three vertices B, D, and G form a cycle and the computation of their values depends on each other. Only setting D's value is not enough to correct the wrong influence from the other nodes.

Precisely, for a given widest path $u \rightarrow v$, the vertex function maintains the invariant that $v.path \leq u.path$. Hence, for a cycle $v \rightarrow w \rightarrow ... \rightarrow k \rightarrow v$ in the graph, the maintained invariant is $k.path \leq ... \leq w.path \leq v.path$. Suppose the actual solution for this entire path is $u.path = v.path = w.path = k.path = m$. When the edge $u \rightarrow v$ is deleted, the vertex function computes a new value for v using its remaining incoming edges, one of which is $k \rightarrow v$. At this point, v would still receive value m from edge $k \rightarrow v$. If m is greater than the values coming from v's other in-neighbors, $v.path$ will still be set to m. This is incorrect since the value m of vertex k was originally computed from the value of v itself.

Similar incorrect behaviors can be observed for ConnectedComponents (CC) (see Table 3) — if there is a cycle, all vertices in the cycle can end up having the same component ID which would create wrong influence after edge deletions.

Motivation from Real Graphs Figure 5 shows the numbers of vertices that have wrong values in the query results for SSWP and CC on the LiveJournal and UKDomain graphs (see Table 2 for details of the graphs). Edge updates are batched in our experiments. At the end of each batch, we send a query that asks for the values of all vertices in the new version of the graph obtained by applying the current batch



(a) A simple graph.

	A	B	C	D	E	F	G
Before deletion →	∞	20	10	20	7	7	20
A→D deleted							
	∞	20	10	20	7	7	20
Correct Result							
	∞	5	10	5	5	7	5

(b) Computation using intermediate values after the deletion of A→D; shown in blue are the correct result.

A	B	C	D	E	F	G
∞	20	10	20	7	7	20
A→D deleted						
∞	20	10	0	7	7	20
∞	20	10	0	5	7	20
∞	20	10	20	5	7	20
∞	20	10	20	7	7	20

(c) Computation using the initial value for D; values changed are in red.

Figure 4: Using either the intermediate or the initial value for vertex D leads to incorrect results (which are underlined); the initial value for each vertex is 0.

of edge updates. The details of the experiment setup can be found in §4.

The vertices that have wrong results are identified by using the results of KickStarter as an *oracle*. Observe that the number of such vertices is noticeably high. Furthermore, the inaccuracies for each batch are carried over into the main processing loop, affecting future query results – this can be seen from the fact that there are increasing numbers of vertices with wrong values as more queries (batches) are performed (processed).

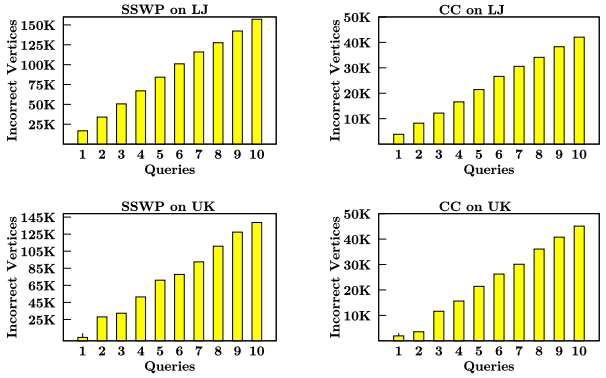
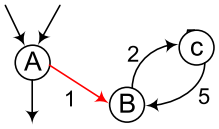


Figure 5: Numbers of vertices with incorrect results.

2.2 Problem 2: Degraded Performance

Consider the SSSP algorithm in Figure 3(b). While this algorithm produces the correct result, it would have severe performance problem if the approximate value is used upon the deletion of edge $A \rightarrow B$ in the graph shown in Figure 6(a). The deletion of the edge renders the vertices B and C disconnected from the rest of the graph. Using the intermediate values 6 and 8 for the forward computation would bump up these values at each iteration (Figure 6(b)); the process would take a large number of iterations to reach the final (correct) result (`MAX`). This is exactly an example of the scenario in Figure 1(c).



(a) A simple graph.

	A	B	C
Before deletion →	5	6	8
A→B deleted			
	5	6	8
	5	13	8
	5	13	15
	5	20	15
	...	...	...
	5	MAX	MAX

(b) Computation using the approximation after the deletion of A→B.

Figure 6: While using the intermediate value for vertex B yields the correct result, the computation can be very slow; the initial value at each vertex is a large number `MAX`.

2.3 How to Distinguish Algorithms

The above examples showed two types of monotonic algorithms, those that may produce incorrect results and others that produce correct results but may have significantly degraded performance in the presence of edge deletions. While the problems in both types are caused by cycles in the graph, different algorithm implementations may lead to different consequences (*i.e.*, incorrectness vs. performance degradation). We observe that the key difference between them is in their vertex update functions. In the first type of algorithms such as SSWP, the update function only performs *value selection* (*i.e.*, no computation on the value is done). Hence, when a cycle is processed, there is a potential for a value being propagated along the cycle without being modified and eventually coming back and inappropriately influencing the vertex that loses an edge, producing incorrect results.

The update function of the second type of algorithms performs computation on the selected value. For example, SSSP first selects a value from an in-neighbor and then adds the edge weight to the value. The addition ensures that when a value is propagated along the cycle, it appears as a different value at the vertex via which the original value had entered the cycle. In this case, the vertex function disallows cyclic

propagation and hence, upon deletion, it becomes impossible for the algorithm to stabilize at a wrong value. To summarize, whether the update function of an algorithm contains actual computation can be used as a general guideline to reason about whether the algorithm can produce incorrect values or would only cause performance problems.

2.4 Correcting Approximations using KickStarter

For both the correctness and performance issues, KickStarter trims the approximation such that correct results can be computed efficiently. For our SSWP example, KickStarter generates A(∞) B(5) C(10) D(0) E(5) F(7) G(5) using which, it would take the computation only one iteration to converge at the correct results. Similarly, for our SSSP example, KickStarter generates A(5) B(`MAX`) C(`MAX`) which is exactly the correct result. The detailed trimming algorithm will be presented in §3.

3. Trimming Approximations

This section describes KickStarter's trimming techniques. We begin with an overview and then discuss the algorithm.

3.1 KickStarter Overview

Given a graph $G = (V, E)$, let an approximation $A_G = a_0, a_1, ..., a_{n-1}$ be a set of values for all vertices in V, *i.e.*, $\forall a_i \in A_G$, a_i is the value for vertex $v_i \in V$ and $|A| = |V|$. For an iterative streaming algorithm S, a *recoverable approximation* A_G^S is an approximation with which the final correct solution on G can be computed by S. A simple example of a recoverable approximation is the set of *initial* vertex values, *i.e.*, values at the beginning of the processing. Note that due to the *asynchronous* nature of streaming algorithms, at any point during the computation, multiple recoverable approximations may exist, each corresponding to a distinct execution path leading to the convergence with the same correct result.

KickStarter Workflow For monotonic streaming algorithms (e.g., SSWP), the approximation maintained in the presence of edge additions is *always recoverable*. Hence, upon a query, running the computation on the updated graph with the approximation before the edge addition always generates the correct result. However, when edge deletion occurs, the approximation before a deletion point may be irrecoverable. Thus, to generate a recoverable approximation, we add a *trimming* phase right after the execution is forked (i.e., the branch loop in Figure 2) for answering a user query. In this phase, the current approximation from the main loop is trimmed by identifying and adjusting *unsafe* vertex values. The trimmed approximation is then fed to the forked execution.

After the query is answered, the result from the branch loop is fed back to the main loop as a new approximation to accelerate the answering of subsequent queries.

Technique Overview KickStarter supports two methods for trimming. The first method identifies the set of vertices s possibly affected by an edge deletion using a tagging mechanism that also exploits algorithmic insights. For the vertices in s, their approximate values are *trimmed off* and reset to the initial values (*e.g.*, a large value for SSSP and 0 for SSWP). This method guarantees *safety* by conservatively tagging values that *may* have been affected. However, conservative trimming makes the resulting approximation less *profitable*.

In the second method, KickStarter tracks dynamic dependences among vertices (*i.e.*, the value of which vertex contributes to the computation of the value of a given vertex) *online* as the computation occurs. While tracking incurs runtime overhead, it leads to the identification of a much smaller and thus a more precise set of affected vertices s. Furthermore, because the majority of vertices are unaffected by edge deletions and their approximate values are still valid, trimming uses these values to compute a set of safe and profitable approximate values that are closer to the final values for the vertices in s.

Our presentation proceeds in following steps: §3.2 presents the first approach where trimming is performed by tag propagation; §3.3 presents the second approach where dynamic value dependences are captured and trimming is performed by calculating new approximate values for the affected vertices. An argument of safety and profitability is provided in §3.5.

3.2 Trimming via Tagging + Resetting

A simple way to identify the set of impacted vertices is vertex tagging. This can be easily done as follows: upon a deletion, the target vertex of the deleted edge is tagged using a set bit. This tag can be iteratively propagated — when an edge is processed, KickStarter tags the target of the edge if its source is tagged. The value of each tagged vertex is set back to its initial value in the branch loop execution.

While tagging captures the transitive impact of the deleted edge, it may tag many more vertices than is necessary. Their values all need to be reset (*i.e.*, the computation done at these vertices is not reused at all), although the approximate values for many of them may still be valid. To illustrate, consider the example in Figure 4. Upon the deletion of edge $A \rightarrow D$, this approach will tag all the vertices in the graph, even though approximate values for at least C and F are valid.

To further reduce the number of tagged vertices, KickStarter relies on *algorithmic insights* to carefully propagate the tag across the vertices. The intuition here is to tag a vertex only if any of its in-neighbors that actually *contributes to* its current value is tagged. Since determining where the contribution comes from requires understanding of the algorithm itself, the developer can expose this information by providing a vertex-centric propagation function. For example, the following function determines how to tag vertices in SSWP:

$$tag(v) \leftarrow \bigvee_{\substack{e \in inEdges(v) s.t. \\ min(e.weight, e.source.value) = v.value}} tag(e.source) \quad (1)$$

Our insight is that in a typical monotonic algorithm, the value of a vertex is often computed from a single incoming edge. That is, only one incoming edge offers contributions to the vertex value. For example, in SSSP, the value of a vertex depends only on the smallest edge value. This observation holds for all monotonic graph algorithms that we are aware of, including the seven listed in Table 1. For these algorithms, the computation is essentially a *selection* function that computes the vertex value by selecting single edge value.

Algorithms	**Selection Func.**
Reachability	*or()*
ShortestPath, ConnectedComponents, MinimalSpanningTree, BFS, FacilityLocation	*min()*
WidestPath	*max()*

Table 1: Monotonic algorithms & their aggregation functions.

At the moment tagging is performed, an edge deletion has already occurred and all its impacted values have already been computed. Here we want to understand which edge contributes to the value of a vertex. The propagation function essentially encodes a "dynamic test" that checks "backward" whether applying the selection on a particular edge can lead to the value. However, since it is a backward process (*e.g.*, that guesses the input from the output), there might be multiple edges that pass this test. To guarantee safety, if the source of any of these edges is tagged, the vertex needs to be tagged. This is reflected by the $\vee$ (or) operator in Eq. 1.

Use of this technique in our example no longer tags vertices A and C. However, while propagation functions reduce the number of tagged vertices, tagging is still a "passive" technique that is not performed until a deletion occurs. This passive nature of tagging dictates that we must reset values for all the tagged vertices although many of them have valid approximate values at the time of tagging — KickStarter cannot determine safe approximate values for these vertices during tagging. For example, in Figure 4, even though vertex F has the correct approximate value 7, it gets tagged and then its value has to be reset to 0. In fact, F receives the same value from two different paths: $A \rightarrow C \rightarrow F$ and $A \rightarrow D \rightarrow E \rightarrow F$. Tagging does not distinguish these paths and hence, it ends up marking F regardless of the path the tag comes from.

Next, we discuss an alternative mechanism that *actively* captures the "which neighbor contributes to a vertex value" relationships, making it possible for us to compute *safe approximate values* for the impacted vertices.

3.3 Trimming via Active Value Dependence Tracking

In this approach we employ *active*, "always-on" dependence tracking, regardless when and where edge deletions occur. Through the recorded dependences, we can precisely identify the vertices impacted by a deleted edge. More importantly, the captured dependences form a data slice, following which safe approximate values can be computed. In contrast, tagging was not active but rather turned on only when a deletion occurred and thus it cannot *compute* approximate values.

3.3.1 Value Dependence

We first formalize a *contributes-to* relation ($\mapsto$) to capture the essence of this type of value dependences. Given two vertices u and v, $u \mapsto v$ if there is an edge from u to v and u's value individually contributes to the value of v. A transitive closure $\mapsto^*$ over $\mapsto$ thus involves all transitive contributes-to relationships. Based on $\mapsto^*$, we formalize a *leads-to* relation ($\xrightarrow{LT}$) to capture the desirable transitive dependences as described above. For two vertices u and v, $u \xrightarrow{LT} v$ iff. (1) $u \mapsto^* v$ and (2) $v \not\mapsto^* u$.

The second condition is important because it ensures that computation of v is not based on vertices whose values were computed using v's previous values (*i.e.*, a dependence cycle). Our goal is to guarantee safety: if a safe approximate value needs to be calculated for v upon an edge deletion, the calculation must avoid the effects of v's previous values by not considering any incoming neighbor u such that $v \mapsto^* u$. It is actually a property of a monotonic graph algorithm — to guarantee convergence, the algorithm often maintains monotonicity invariants across different values of each vertex and values of the neighbors. For example, computation in SSSP has the invariant that if $u \mapsto v$, then v's value is no smaller than u's value. All the algorithms listed in Table 1 maintain such monotonicity invariants.

Note that the *leads-to* relation is a subset of the set of normal *flow* data dependences. A flow dependence, induced by a write followed by a read operating at the same location, may or may not give rise to a *leads-to* relationship, since $\xrightarrow{LT}$ is defined at a high level over vertices of the graph.

While $\xrightarrow{LT}$ may be automatically computed by runtime techniques such as dynamic slicing [49, 50], such techniques are often prohibitively expensive, incurring runtime overhead that causes a program to run hundreds of times slower. Since graph algorithms are often simple in code logic, we leverage the developer's input to compute $\xrightarrow{LT}$. We design a simple API `contributesTo` that allows the developer to expose *contribute-to* ($\mapsto$) relationships when writing the algorithm. For example, for SSWP, the edge on which the vertex value depends is the last edge that triggers the execution of Line 7 in Figure 3(a). The developer can add an API call right after that line to inform KickStarter of this new relationship which led to computation of the new value.

Note that we only rely on the developer to specify direct $\mapsto$ relationships, which incurs negligible manual effort. The transitive closure computation is done automatically by KickStarter. The freedom from dependence cycles is actually provided by monotonicity.

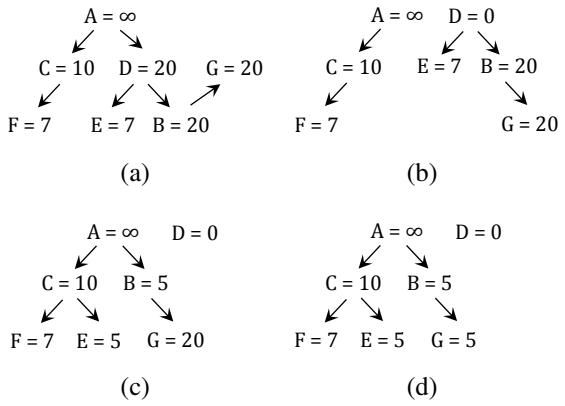


Figure 7: (a) Dependence tree for Figure 4(a) before the edge deletion; (b)-(d) trimming reorganizes the dependence tree.

Representing `Leads-to` Relation as Dependence Trees As individual *contributes-to* relationships are profiled, these relationships form a dependence graph $D = (V^D, E^D)$, which shares the same set of vertices as the original graph $G = (V, E)$. Each edge in D represents a $\mapsto$ relationship and the edge set E^D is a subset of E. We maintain one single dependence graph across iterations. However, dependence profiling is not accumulative — if a new value of a vertex is computed and it results from a different in-neighbor, a new dependence edge is added to *replace* the old dependence edge for the vertex.

This dependence graph D encodes the $\xrightarrow{LT}$ relation and has two crucial properties. (1) It is *acyclic* – this follows the definition of *leads-to* relation which ensures that if there is a directed path from u to v in D, then there must be no directed path from v back to u in D. (2) Every vertex $v \in V^D$ has at most one incoming edge, *i.e.*, if $u \mapsto v$, then $\forall w \in V^D \setminus \{u\}, w \not\mapsto v$. This can be derived from the fact that the selection function only selects one edge to compute the vertex value and the computation of a new value at the vertex replaces the old dependence edge with a new edge.

The above properties imply that D is a set of *dependence trees*. Figure 7(a) shows a simple dependence tree for the SSWP algorithm in Figure 4 before the edge $A \rightarrow D$ is deleted.

3.3.2 Computing New Approximate Values

Taking the set of dependence trees as input, KickStarter computes new approximate values for the vertices that are affected by deletions. First of all, KickStarter identifies the set of vertices impacted by a deleted edge. This can be done simply by finding the subtree rooted at the target vertex of the deleted edge.

To compute values for profitability, KickStarter employs three strategies: (1) it ignores deletions of edges which do not have corresponding dependence edges in D. This is safe because such edges did not contribute to the values of their target vertices; (2) if a deleted edge does have a corresponding dependence edge in D, KickStarter computes a *safe alternate value* for its target vertex. While resetting the approximate value of the vertex is also safe, KickStarter tries to compute a better approximate value to maximize profitability; and (3) once a safe alternate value is found for the vertex, KickStarter may or may not continue trimming its immediate children in D, depending on whether this new approximate value disrupts monotonicity.

Since the first strategy is a straightforward approach, we focus our discussion here on the second and third strategies.

Finding Safe Approximate Values Given a vertex affected by a deletion, KickStarter finds an alternate approximate value that is safe. Our key idea is to *re-execute* the update function on the vertex to compute a new value, starting from the target vertex of the deleted edge. One problem here is that, as we have already seen in Figure 4, there may be cycles (*e.g.*, B, G, and D in Figure 4(a)) in the actual graph (*not* the dependence graph) and, hence, certain in-neighbors of the vertex may have their values computed from its own value. This cyclic effect would make the re-execution still produce wrong values.

To eliminate the affect of cycles, KickStarter re-executes the update function at vertex v on a subset of its incoming edges whose source vertices do not depend on v. More precisely, we pass a set of edges e such that $v \not\mapsto^* e.src$ into the update function to recompute v's value. In Figure 4, after $A \rightarrow D$ is deleted, we first re-execute the SSWP function in Figure 3(a) at vertex D. The function does not take any edge as input — neither $G \rightarrow D$ nor $E \rightarrow D$ is considered since both the values of G and E depend on D in the dependence tree shown in Figure 7(a). D's value is then reset to 0.

Determining this subset of incoming edges for vertex v can be done by performing a dependence tree traversal starting at v and eliminating v's incoming edges (on the original graph) whose sources are reachable. However, such a traversal can be expensive when the sub dependence tree rooted at v is large. Hence, we develop an inexpensive algorithm that conservatively estimates reachability using the *level information* in the dependence trees.

For vertex v, let $level(v)$ be the level of v in a dependence tree. Each root vertex gets level 0. The tree structure dictates that $\forall w : v \mapsto^* w$, $level(w) > level(v)$. Hence, as a conservative estimation, we construct this subset of incoming edges by selecting every edge such that the level of its source vertex u is $\leq$ the level of v. This approach guarantees safety because every in-neighbor w of v such that $v \mapsto^* w$ is excluded from the set. It is also lightweight, as the level information can be maintained on the fly as the tree is built.

When to Stop Trimming Once a safe value is found for a vertex, KickStarter checks whether the value can disrupt monotonicity. For example, if this value is higher (lower) than the previous vertex value in a monotonically decreasing (increasing) algorithm, the monotonicity of the current value

is disrupted, which can potentially disrupt the monotonicity of its children vertices. In such a case, the resulting approximation may not be recoverable yet because cycles in the original graph can cause the effects of the previous value to inappropriately impact the computation at the children vertices. Hence, trimming also needs to be done for the children vertices.

On the other hand, if the monotonicity for the current vertex is not disrupted by the new value, the trimming process can be safely terminated because the approximate value for the current vertex is recoverable. Since the current vertex is the only one that contributes to the values of its children vertices, the values of the child vertices would become recoverable during the forward graph computation. As an example, for SSWP, if the old value for a vertex v is v_{old} and its new value is v_{new}, whether to continue the trimming process can be determined by the following rule:

$$continue \leftarrow \begin{cases} true & \text{... if } v_{new} < v_{old} \\ false & \text{... otherwise} \end{cases} \quad (2)$$

Example As new approximate values are computed, the dependence trees are adjusted to reflect the new dependences. Figure 7(b)-(d) show how the structure of the dependence tree in Figure 7(a) changes as trimming progresses. First, the subset of incoming edges selected for D, referred to as D^s, is an empty set, and hence, D's value is reset to 0 (*i.e.*, the initial value) and D becomes a separate root itself (Figure 7(b)) because it does not depend on any other vertex. Since this value is smaller than its old value (20), monotonicity is disrupted and thus D's immediate children, E and B, need to be trimmed. E^s consists of the incoming edges from C and D. The re-execution of the update function gives E a safe value 5, making E a child of C in the dependence tree (Figure 7(c)). Similarly, B^s consists of the incoming edges from A and D. B then receives the safe value 5, making itself a child of A in the tree (Figure 7(c)). Similarly, trimming continues to G, which receives a safe approximate value 5 from B (Figure 7(d)).

Putting It All Together: The Parallel Trimming Algorithm Trimming can be done in parallel on vertices since the computations involved in determining safe approximate values are confined to a vertex and its immediate neighbors. Hence, trimming itself can be expressed as vertex-centric computation and performed on the same graph processing engine.

Algorithm 1 presents the overall vertex-centric trimming algorithm. It first creates a subset of incoming edges (Lines 2-9) which can be used to generate a safe approximate value. Then it executes the same vertex function used to perform the actual graph computation to find a safe approximate value (Line 12). Note that when this is done, the value dependence exposed by the developer will be captured. Finally, the old and new vertex values are used to determine whether trimming should be done to the children of the vertex. The algorithm requires algorithmic insights which are provided by the developer using a comparator function (Line 15). Depending upon the result of the function, the immediate children in the dependence trees may or may not be scheduled to be processed.

Since multiple deletions can be present in the same update batch, trimming can be performed in such a way that it starts from the highest level of the dependence tree and gradually moves down, rather than starting at multiple deletion points which may be at different tree levels. This way multiple trimming flows get merged into a single flow.

Algorithm 1 Vertex-centric trimming algorithm.

```
1:  function TRIM(Vertex v)
2:      ▷ Construct the subset of incoming edges
3:      v^s ← ∅
4:      for e ∈ INCOMINGEDGES(v) do
5:          if e.source.level ≤ v.level then
6:              v^s.insert(e)
7:          end if
8:      end for
9:      incomingSet ← CONSTRUCTSUBSET(v.value, v^s)
10:
11:     ▷ Find safe alternate value
12:     v.newValue ← VERTEXFUNCTION(incomingSet)
13:
14:     ▷ Continue trimming if required
15:     continueTrim ←
16:             SHOULDPROPAGATE(v.value, v.newValue)
17:     if continueTrim = true then
18:         SCHEDULECHILDREN(v)
19:     end if
20:
21:     v.value ← v.newValue
22:     v.UPDATE(incomingSet)
23: end function
```

3.4 Trimming for Performance

As shown in Figure 6, certain deletions can render the approximate values of the affected vertices far away from their final results, causing the branch loop to take a long time to converge. The trimming technique described in §3 is automatically applicable in such cases to accelerate convergence. For example, in the case of SSSP (Figure 6), since the algorithm is monotonically decreasing, our dependence based trimming will keep trimming vertices when their new values are larger than their old values. In Figure 6 (a), after $A \rightarrow B$ is deleted, B's value is reset to `MAX` since the subset of incoming edges used to reexecute the update function at B is empty because C depends on B and thus $C \rightarrow B$ is not considered. This significantly accelerates computation since C's value can only be set to `MAX` as well (due to the influence from B).

3.5 Safety and Profitability Arguments

Safety It is straightforward to see that tagging + resetting provides safety because it resets vertex values conservatively. For the dependence-based trimming, as long as the developer appropriately inserts the dependence-tracking API call into the program, *all* value dependences will be correctly recorded. For monotonic algorithms that use one single incoming edge to compute the value of a vertex, these dependence relationships yield a set of dependence trees. When an edge is deleted, the subtree rooted at the target vertex of the deleted edge thus includes a *complete* set of vertices that are directly or transitively impacted by the deleted edge. Given this set of impacted vertices, we next show that trimming produces a safe (recoverable) approximation.

Theorem 3.1. *Trimming based on value dependence trees produces a safe approximation.*

Proof. We prove the safety of our trimming process by analyzing the set of values used for computing the new approximations and showing that these values themselves are not *unsafe approximations*, that is, they are not over-approximations for monotonic increasing algorithms and under-approximations for monotonic decreasing algorithms. Let us consider the deletion of an edge $a \rightarrow b$, which triggers the trimming process. We prove the theorem by contradiction. Suppose the approximation produced by trimming is unsafe and let v be the vertex whose approximate value becomes unsafe after trimming is performed. If we were to back-track how the unsafe approximation got computed during trimming, there must be an earliest point at which an unsafe value was introduced and propagated subsequently to v. Let c be such an earliest vertex. Since c's value was safe prior to the deletion of $a \rightarrow b$ but it became unsafe afterwards, c is dependent on $a \rightarrow b$. This means $b \xrightarrow{LT} c$, and now, $c \xrightarrow{LT} v$.

In this case, prior to the edge deletion, the dependence relationship $b \xrightarrow{LT} c$ must have been captured. When the deletion occurs, the trimming process considers the entire subtree rooted at b in the collected dependence trees, which includes the path from b to c. Our algorithm computes a new value for c if its predecessor's value changes against the monotonic direction. This leads to the following two cases:
CASE 1 — The value of c's predecessor indeed changes against monotonicity. In this case, c's value is recomputed. As described earlier, only those incoming values that are not in the subtree rooted at c are considered for computing c's new value, which ensures that all the vertices whose values were (directly or indirectly) computed using c's old value do not participate in this new computation. The incoming values selected for computation must have safe approximation themselves because c is the earliest vertex whose approximation became unsafe. Hence, using safe, but fewer, incoming values for c can only lead to a safe approximate value for c due to mononicity (*e.g.*, a value higher than the accurate value in a decreasing monotonic algorithm).

Graphs	#Edges	#Vertices
Friendster (FT) [13]	2.5B	68.3M
Twitter (TT) [6]	2.0B	52.6M
Twitter (TTW) [20]	1.5B	41.7M
UKDomain (UK) [5]	1.0B	39.5M
LiveJournal (LJ) [3]	69M	4.8M

Table 2: Real world input graphs.

CASE 2 — The value of c's predecessor does not change against monotonicity. In this case, c's old approximate value is already safe w.r.t. its predecessor.

Combining Case 1 and 2, it is clear to see that there does not exist any such vertex c whose value can become unsafe and flow to v under our algorithm. Simple induction on the structure of the dependence tree would suffice to show any vertex v's value must be safe. □

Profitability It is easy to see that any *safe* approximate value is at least as good as the initial value. Since the value already carries some amount of computation, use of the value would reuse the computation, thereby reaching the convergence faster than using the initial value.

4. Evaluation

This section presents a thorough evaluation of KickStarter on real-world graphs.

4.1 Implementation

KickStarter was implemented in a distributed graph processing system called ASPIRE [41]. In ASPIRE, graph vertices are first partitioned across nodes and then processed using a vertex-centric model. The iterative processing incorporates a vertex activation technique using bit-vectors to eliminate redundant computation on vertices whose inputs do not change.

Updates are batched in an in-memory buffer and not applied until the end of an iteration. Value dependence trees are constructed by maintaining the level information for each vertex along with "downward" pointers to children vertices that allow trimming to quickly walk down a tree. A query is performed after a batch of updates is applied. The query asks for the values of *all* vertices in the updated graph. Different types of edge updates are handled differently by KickStarter: edge deletion removes the edge and schedules the edge target for trimming; and standard treatment is employed for edge additions.

4.2 Experimental Setup

We used four monotonic graph algorithms in two categories as shown in Table 3: SingleSourceWidestPaths (SSWP) and ConnectedComponents (CC) may produce incorrect results upon edge deletions whereas SingleSourceShortestPaths (SSSP) and BreathFirstSearch (BFS) produce correct results with poor performance. VERTEXFUNCTION shows the algo-

Algorithm	Issue	VERTEXFUNCTION	SHOULDPROPAGATE
SSWP	Correctness	$v.path \leftarrow \max_{e \in \text{inEdges}(v)} (\min(e.source.path, e.weight))$	$newValue < oldValue$
CC	Correctness	$v.component \leftarrow \min(v.component, \min_{e \in \text{edges}(v)} (e.other.component))$	$newValue > oldValue$
BFS	Performance	$v.dist \leftarrow \min_{e \in \text{inEdges}(v)} (e.source.dist + 1)$	$newValue > oldValue$
SSSP	Performance	$v.path \leftarrow \min_{e \in \text{inEdges}(v)} (e.weight + e.source.path)$	$newValue > oldValue$

Table 3: Various vertex-centric graph algorithms.

		LJ	UK	TTW	TT	FT
SSWP	**RST**	7.48-10.16 (8.59)	81.22-112.01 (90.75)	94.18-102.27 (99.28)	170.76-183.11 (176.87)	424.46-542.47 (487.04)
	TAG	11.57-14.71 (13.00)	1.73-62.1 (21.42)	27.38-125.91 (71.26)	262.88-278.42 (270.29)	474.64-550.25 (510.52)
	VAD	3.51-5.5 (4.48)	1.17-1.18 (1.17)	21.54-34.38 (27.55)	66.85-130.84 (75.88)	113.3-413.51 (143.72)
CC	**RST**	6.43-7.93 (7.19)	133.92-166.33 (148.80)	105.16-111.46 (107.54)	113.92-126.35 (126.35)	212.43-230.26 (221.05)
	TAG	10.98-12.81 (11.86)	170.91-203.54 (183.93)	176.91-201.12 (185.84)	193.77-249.93 (208.90)	331.79-386 (360.34)
	VAD	4.89-5.85 (5.30)	1.81-7.75 (4.37)	31.78-33.24 (32.54)	21.98-22.58 (22.29)	38-39.36 (38.56)

Table 4: Trimming for correctness: query processing time (in sec) for SSWP and CC, shown in the form of min-max (average).

		LJ	UK	TTW	TT	FT
SSWP	**TAG**	3.1M-3.2M (3.1M)	8.9K-9.7M (4.1M)	1.1K-29.5M (13.4M)	28.5M-28.6M (28.6M)	49.5M-49.5M (49.5M)
	VAD	20.1K-90.8K (60.8K)	2.9K-93.0K (33.4K)	1.0K-4.5K (2.3K)	2.4K-1.1M (106.4K)	20.7K-13.6M (1.3M)
CC	**TAG**	3.2M-3.2M (3.2M)	25.9M-25.9M (25.9M)	31.3M-31.3M (31.3M)	32.2M-32.2M (32.2M)	52.1M-52.1M (52.1M)
	VAD	1.1K-3.1K (1.9K)	320-1.6K (1.0K)	116-463 (212)	241-463 (344)	294-478 (374)

Table 5: Trimming for correctness: # reset vertices for SSWP and CC (the lower the better) in the form of min-max (average).

rithms of vertex computation, while SHOULDPROPAGATE reports the termination conditions of the trimming process.

The algorithms were evaluated using five real-world graphs listed in Table 2. Like [32], we obtained an initial fixed point and streamed in a set of edge insertions and deletions for the rest of the computation. After 50% of the edges were loaded, the remaining edges were treated as edge additions that were streamed in. Furthermore, edges to be deleted were selected from the loaded graph with a 0.1 probability; deletion requests were mixed with addition requests in the update stream. In our experiments, we varied both the rate of the update stream and the ratio of deletions vs. additions in the stream to thoroughly evaluate the effects of edge deletions.

All experiments were conducted on a 16-node Amazon EC2 cluster. Each node has 8 cores and 16GB main memory, and runs 64-bit Ubuntu 14.04 kernel 3.13.

Techniques Compared We evaluated KickStarter by comparing the following four versions of the streaming algorithm:

- **TAG** uses the *tagging + resetting* based trimming (*cf.* §3.2).
- **VAD** uses our *value dependence* based trimming (*cf.* §3.3).
- **TOR** implements Tornado [32]. This approach is used as the baseline for BFS and SSSP, as **TOR** generates correct results for these algorithms.
- **RST** does not perform trimming at all and instead *resets* values of all vertices. This technique serves as the baseline for SSWP and CC because **TOR** does not generate correct results for them (as already shown in Figure 5 in §2).

To ensure a fair comparison among the above versions, queries were generated in such a way that each query had the same number of pending edge updates to be processed. Unless otherwise stated, 100K updates with 30% deletions were applied before the processing of each query.

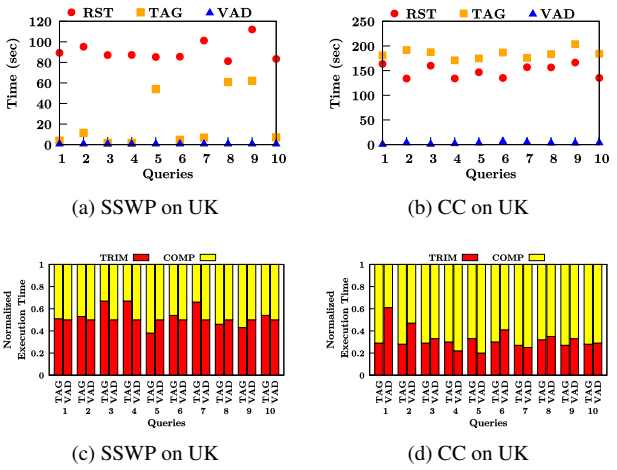
4.3 Trimming for Correctness

We first study the performance of our trimming techniques, **TAG** and **VAD**, to generate correct results for SSWP and CC — Table 4 shows the average, minimum, and maximum execution times (in seconds) to compute the query results using **TAG**, **VAD**, and **RST** (baseline).

We observe that **VAD** consistently outperforms **RST**. On average, **VAD** for SSWP and CC performs **17.7**× and **10**× faster than **RST**, respectively. These significant speedups were achieved due to the incremental processing in **VAD** that maximizes the use of computed approximate values, as can be seen in Table 5 — only a subset of vertices have to discard their approximate values (via resetting). Since **RST** discards the entire approximation upon edge deletions (*i.e.*, resetting all impacted vertices), computation at every tagged vertex starts from scratch, resulting in degraded performance.

Finally, for the UK graph, **VAD** performs noticeably better for SSWP than for CC mainly because safe approximate values computed for SSWP were closer to the final solution than those for CC. The reason is as follows. For CC, if the

component ID for a vertex in a component changes, this change is likely to get propagated to all the other vertices in that component. This means that when trimming finds a safe approximate value, the value may still be changed frequently during the forward execution. For SSWP, on the other hand, if the path value for a vertex changes, the change does not affect many other vertices. Hence, only small local changes may occur in vertex values before the computation converges. As a result, SSWP took less time than CC to finish the branch loop (less than a second in all cases for SSWP vs. between 1 and 4 seconds for CC).



(a) SSWP on UK (b) CC on UK

(c) SSWP on UK (d) CC on UK

Figure 8: Time taken to answer queries.

Next, we compare the tagging+resetting algorithm **TAG** with **VAD** and **RST**. In most cases, **TAG** outperforms **RST**. However, **TAG** performs worse than **RST** for CC because the overhead of using **TAG** outweighs the benefit provided by trimming — due to **TAG**'s conservative nature, a very large portion of the graph is tagged and their values are reset. This can be seen in Table 5 where there are millions of vertices whose values are reset in CC.

Under **TAG**, the number of vertices whose values are reset is significantly higher than that under **VAD** (see Table 5). Hence, **VAD** consistently outperforms **TAG**. Note that the reason why **VAD** works well for CC is that since CC propagates component IDs, many neighbors of a vertex may have the same component ID and thus trimming based on value dependence may have more approximate values to choose from. As a result, **VAD** resets far fewer vertices than **TAG**.

Figures 8a and 8b show the performance of **RST**, **TAG**, and **VAD** for the first 10 queries for SSWP and CC on UK. The performance of **TAG** is more sensitive to edge deletions than **VAD** – for SSWP, while the solutions for many queries were computed quickly by **TAG**, some queries took significantly longer processing time. **VAD**, on the other hand, is less sensitive to edge deletions because it is able to attribute the effects of the deletions to a smaller subset of vertices.

Finally, Figures 8c and 8d compare the performance of the two phases of the branch loop execution: *trimming* (TRIM) and *computation* (COMP). Since CC is more sensitive to edge additions and deletions compared to SSWP, it took longer processing time to converge to the correct result. Hence, for CC, the percentages of the time spent by both **TAG** and **VAD** on the trimming phase are lower than those on the computation phase. SSWP, on the other hand, needs less time to converge because the approximate values available for incremental processing are closer to the final values; hence, the time taken by the trimming phase becomes comparable to that taken by the computation phase. Nevertheless, as Table 4 shows, the trimming phase has very little influence on the overall processing time due to its lightweight design and parallel implementation.

4.4 Trimming for Performance

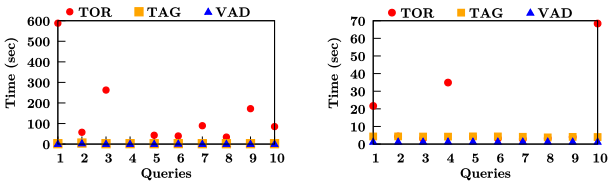
This set of experiments help us understand how different trimming mechanisms can improve the performance of query processing for BFS and SSSP. For these two algorithms, as explained in §2, although the baseline TOR (Tornado) produces correct results, it can face performance issues. Table 6 shows the average, minimum, and maximum execution times (in seconds) to compute the query results for SSSP and BFS by **TOR**, **TAG**, and **VAD**.

VAD consistently outperforms **TOR**. For example, **VAD** for SSSP and BFS are overall **23.7**× and **8.5**× faster than **TOR**, respectively. Figure 9a and Figure 9b show the performance for answering the first 10 queries for SSSP and BFS. Since **TOR** leverages incremental processing, its performance for some queries is competitive with that of **VAD**. However, different edge deletions impact the approximation differently and in many cases, **TOR** takes a long time to converge, leading to degraded performance.

While **TAG** consistently outperforms **TOR**, its conservative resetting of a larger set of vertex values (as seen in Table 7) introduces overhead that reduces the overall benefit it provides.

4.5 Effectiveness of the Trimmed Approximation

To understand whether the new approximate values computed by trimming are beneficial, we compare **VAD** with a slightly modified version **VAD-Reset** that does not compute new approximate values. This version still identifies the set of



(a) SSSP on UK (b) BFS on UK

Figure 9: Trimming for performance: time taken to compute answer queries by **TAG** and **VAD**.

		LJ	UK	TTW	TT	FT
SSSP	**TOR**	1.27-102.88 (39.10)	2.84-119.03 (24.90)	17.62-131.9 (112.57)	42.13-584.64 (190.78)	90.59-179.83 (163.99)
	TAG	3.25-4.49 (3.97)	2.03-2.94 (2.19)	46.06-52.5 (48.96)	98.59-118.23 (105.73)	131.22-150.16 (142.60)
	VAD	2.12-3.22 (2.55)	1.33-1.5 (1.41)	28.68-32.33 (30.21)	41.35-48.65 (44.19)	93.74-101.67 (97.22)
BFS	**TOR**	1.17-77.05 (7.17)	1.24-588.09 (142.55)	23.94-1015.76 (199.23)	55-283.71 (120.45)	190.52-2032.38 (881.17)
	TAG	3.47-4.43 (3.88)	1.81-5.14 (1.97)	51.08-58.3 (54.36)	110.75-192.71 (127.54)	143.21-334.07 (166.60)
	VAD	1.96-3.37 (2.59)	1.21-3.88 (1.42)	32.02-34.86 (32.96)	69.43-91.88 (74.27)	107.4-136.73 (114.56)

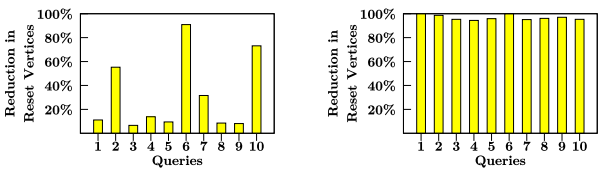
Table 6: Trimming for performance: query processing times (in sec) for SSSP and BFS in the form: min-max (average).

		LJ	UK	TTW	TT	FT
SSSP	**TAG**	8.2K-59.8K (25.9K)	4.1K-193.4K (36.4K)	19.7K-183.7K (89.4K)	6.2K-196.7K (51.5K)	19.8K-31.2K (25.4K)
	VAD	1.7K-40.1K (7.0K)	2.9K-52.2K (16.6K)	2.1K-77.7K (19.6K)	836-110.9K (11.1K)	4.5K-12.5K (8.0K)
BFS	**TAG**	10.8K-354.5K (79.0K)	1.3K-483.0K (35.5K)	20.9K-1.2M (457.6K)	44.2K-8.6M (1.1M)	19.1K-4.5M (469.8K)
	VAD	5.5K-116.6K (36.4K)	3.2K-469.9K (41.2K)	860-3.1K (1.6K)	742-1.4K (1.1K)	2.7K-5.2K (3.4K)

Table 7: Trimming for performance: number of reset vertices for SSSP and BFS in the form: min-max (average).

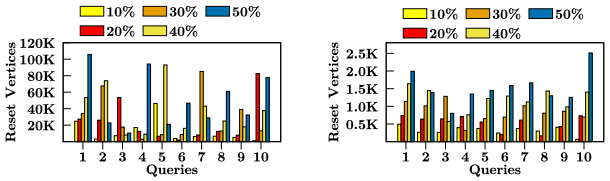
impacted vertices using dependence tracking, but simply resets the values of all vertices in the set. Figure 10 shows the reductions in the numbers of reset vertices achieved by **VAD** over **VAD-Reset**. The higher the reduction, the greater is the computation reused. This comparison was done on the first 10 queries for SSWP and CC over the UK graph. We observe that the reduction varies significantly between SSWP and CC. This is mainly due to the different shapes of the dependence trees constructed for CC and SSWP. For CC, the dependence trees are fat (*i.e.*, vertices have more children) and, hence, if a vertex's value is reset, the trimming process needs to continue resetting many of its children vertices, hurting performance significantly. In fact, 5.7K-25.9M vertices were reset for CC under **VAD-Reset**.

As CC propagates component IDs and the IDs in a vertex's neighborhood are often the same (because the vertices in a neighborhood likely belong to the same component), good approximate values are often available under **VAD**, which greatly reduces the number of reset vertices (to 320-1.3K). For SSWP, on the other hand, its dependence trees are thinner (*i.e.*, vertices have less children), and hence **VAD-Reset** does a reasonably good job as well, resetting only 3.4K-151K vertices. This number gets further reduced to 2.9K-85.2K when **VAD** is used. We have also observed that the benefits achieved for SSWP vary significantly across different queries; this is due to the varying impacts of deletions that affect different regions of the dependence trees and thus the availability of safe approximate values.



(a) SSWP on UK (b) CC on UK

Figure 10: Reduction in # of vertices reset by **VAD** compared to **VAD-Reset**.

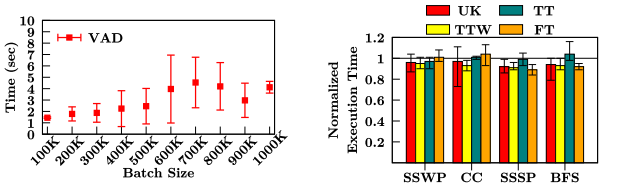


(a) SSWP on UK (b) CC on UK

Figure 11: Numbers of reset vertices with different deletion percentages in the batch.

4.6 Sensitivity to Edge Deletions & Batch Size

We study the sensitivity of the trimming effectiveness to the number of deletions performed in the update stream. In Figure 11, we vary the percentage of deletions in an update batch from 10% to 50% while maintaining the same batch size of 100K edge updates. While the trend varies across different queries and algorithms, it is important to note that our technique found safe approximations in many cases, keeping the number of reset values low even when the number of deletions increases.



(a) Time taken by queries with varying batch size for SSWP on UK (b) Dependence tracking overhead

Figure 12: Query time and dependence tracking overhead.

In Figure 12a, we varied the number of edge updates applied for each query from 100K to 1M while setting the deletion percentage to 30%. Clearly, the increase in the number of edge updates, and hence in the number of edge deletions, do not have much impact on the performance of the forked branch loop. This is mainly because the number of reset vertices remains low — it increases gradually from 31K to 230K.

4.7 Dependence Tracking Overhead

Finally, we try to understand the overhead of our dependence tracking technique by measuring the performance of the system with only edge additions. Since the handling of edge additions does not need trimming, the difference of the running time between **VAD** and **TOR** is the tracking overhead. Figure 12b shows the overall execution times for queries under **VAD** normalized *w.r.t.* **TOR** with only the edge addition requests. That is, the deletion percentage is set to 0%. In this case, all the four algorithms leverage the incremental processing in **TOR** and hence the maintained approximation is never changed. The error bars in Figure 12b indicate the min and max values to help us understand the performance variations. The overall performance is only slightly influenced (max overhead bars are slightly above 1 in most cases) and the query answering time under **VAD** increases by 13%.

5. Related Work

(A) Streaming Graph Processing *Custom Solutions.* These works develop specialized streaming algorithms to solve different problems. They incorporate correctness in the algorithm design either by relaxing the problem constraints or by dealing with edge mutations in specific ways.

STINGER [12] uses a novel data structure which enables quick insertions and deletions while allowing parallel traversal of vertices and edges. [11] develops an approximation method for maintaining clustering coefficients using bloom filters. [10, 26] incorporate techniques to correctly maintain connected components information using STINGER by using set intersection of neighborhood vertices to quickly determine connectivity and construct a spanning tree for each component to check for reachability up to root. While checking reachability is expensive, the algorithm relies on multiple concurrent graph traversals to maximize parallelism. In comparison, our trimming solution does not need expensive traversals since it relies on level checking. Other custom solutions for connectivity checks upon deletions are: [33] relies on two searches to find component splits whereas [17, 27] maintain graph snapshots for each iteration and use LCA and coloring technique. [45] presents a novel clustering algorithm which is aware of the evolution whereas Fennel [38] proposes a novel partitioning algorithm. [35] proposes greedy, chunking and balancing based heuristics to partition streaming graphs.

Generalized Streaming Graph Processing Frameworks. These systems allow users to express graph algorithms. When a query arrives in Tornado [32], it takes the current graph snapshot and branches the execution to a separate loop to compute results using incremental processing. Kineograph [8] is a distributed streaming graph processing system which enables graph mining over fast-changing graphs and uses incremental computation along with push and pull models. [36] proposes the GIM-V incremental graph processing model based upon matrix-vector operations. [25] constructs representative snapshots which are initially used for querying and upon success uses real snapshots. Naiad [23] incorporates differential data flow to perform iterative and incremental algorithms. Graspan [42] is a disk-based graph system that processes program graphs with constant edge additions.

(B) Evolving Graph Processing [40] presents two temporal optimizations to process evolving graphs: computation reordering to perform communication aggregation across graph snapshots; and incremental processing to leverage previously computed (potentially partial) results across graph snapshots. Chronos [16] is a storage and execution engine that also uses incremental processing. Spark [46], vertices maintain a history of values over time and inaccuracies are rectified by reverting back the values. Beyond these, static graph systems [7, 14, 21, 22, 28, 29, 31, 34, 39, 43, 44] can process graph snapshots one after the other.

(C) Data Stream Processing Various generalized data stream processing systems [1, 2, 4, 15, 24, 37, 47, 48] have been developed that operate on unbounded structured and unstructured streams which allow window operations, incremental aggregation and instant querying to retrieve timely results. They allow users to develop their own streaming algorithms; note that users must ensure correctness of algorithms. [30] identifies errors in data-stream processing and improves the accuracy of sketch-based algorithms like Count-Min, Frequency-Aware Counting, etc.

6. Conclusion

This paper presents KickStarter, a runtime technique that produces safe and profitable approximate values for vertices that are impacted by edge deletions. KickStarter can be used to augment an existing streaming graph processing engine, enabling it to process edge updates correctly and efficiently.

Acknowledgments

This work is supported by AWS Cloud Credits for Research to UC Riverside, NSF grants CCF-1524852 and CCF-1318103 to UC Riverside, NSF grants 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.

References

[1] D.J. Abadi, Y. Ahmad, M. Balazinska, U. Cetintemel, M. Cherniack, J-H. Hwang, W. Lindner, A. Maskey, A. Rasin, E. Ryvkina, et al. The design of the borealis stream processing engine. In *CIDR*, volume 5, pages 277–289, 2005.

[2] R. Ananthanarayanan, V. Basker, S. Das, A. Gupta, H. Jiang, T. Qiu, A. Reznichenko, D. Ryabkov, M. Singh, and S. Venkataraman. Photon: Fault-tolerant and scalable joining of continuous data streams. In *SIGMOD*, pages 577–588, 2013.

[3] L. Backstrom, D. Huttenlocher, J. Kleinberg, and X. Lan. Group formation in large social networks: Membership, growth, and evolution. In *KDD*, pages 44–54, 2006.

[4] H. Balakrishnan, M. Balazinska, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, E. Galvez, J. Salz, M. Stonebraker, N. Tatbul, R. Tibbetts, and S. Zdonik. Retrospective on aurora. *The VLDB Journal*, 13(4):370–383, 2004.

[5] P. Boldi and S. Vigna. The WebGraph framework I: Compression techniques. In *WWW*, pages 595–601, 2004.

[6] M. Cha, H. Haddadi, F. Benevenuto, and P.K. Gummadi. Measuring user influence in twitter: The million follower fallacy. *ICWSM*, 10(10-17):30, 2010.

[7] R. Chen, J. Shi, Y. Chen, and H. Chen. PowerLyra: Differentiated graph computation and partitioning on skewed graphs. In *EuroSys*, pages 1:1–1:15, 2015.

[8] R. Cheng, J. Hong, A. Kyrola, Y. Miao, X. Weng, M. Wu, F. Yang, L. Zhou, F. Zhao, and E. Chen. Kineograph: Taking the pulse of a fast-changing and connected world. In *EuroSys*, pages 85–98, 2012.

[9] U. Demiryurek, B. Pan, F. Banaei-Kashani, and C. Shahabi. Towards modeling the traffic data on road networks. In *International Workshop on Computational Transportation Science*, pages 13–18, 2009.

[10] D. Ediger, J. Riedy, D.A. Bader, and H. Meyerhenke. Tracking structure of streaming social networks. In *IEEE IPDPS Workshops and Phd Forum*, pages 1691–1699, May 2011.

[11] D. Ediger, K. Jiang, J. Riedy, and D.A. Bader. Massive streaming data analytics: A case study with clustering coefficients, In *IEEE IPDPSW*, pages 1-10, 2010.

[12] D. Ediger, R. Mccoll, J. Riedy, and D.A. Bader. Stinger: High performance data structure for streaming graphs, In *HPEC*, Sept. 2012.

[13] Friendster network dataset, 2015.

[14] J.E. Gonzalez, R.S. Xin, A. Dave, D. Crankshaw, M.J. Franklin, and I. Stoica. GraphX: Graph processing in a distributed dataflow framework. In *OSDI*, pages 599–613, 2014.

[15] STREAM Group et al. Stream: The stanford stream data manager. *IEEE Data Engineering Bulletin, http://www-db. stanford. edu/stream*, 2(003), 2003.

[16] W. Han, Y. Miao, K. Li, M. Wu, F. Yang, L. Zhou, V. Prabhakaran, W. Chen, and E. Chen. Chronos: A graph engine for temporal graph analysis. In *EuroSys*, pages 1:1–1:14, 2014.

[17] M.R. Henzinger, V. King, and T. Warnow. Constructing a tree from homeomorphic subtrees, with applications to computational evolutionary biology. *Algorithmica*, 24(1):1–13, 1999.

[18] A.P. Iyer, L.E. Li, T. Das, and I. Stoica. Time-evolving graph processing at scale. In *International Workshop on Graph Data Management Experiences and Systems*, pages 5:1–5:6, 2016.

[19] E. Kanoulas, Y. Du, T. Xia, and D. Zhang. Finding fastest paths on a road network with speed patterns. In *ICDE*, page 10, April 2006.

[20] H. Kwak, C. Lee, H. Park, and S. Moon. What is Twitter, a social network or a news media? In *WWW*, pages 591–600, 2010.

[21] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J.M. Hellerstein. Distributed graphlab: A framework for machine learning and data mining in the cloud. *Proc. VLDB Endow.*, 5(8):716–727, April 2012.

[22] G. Malewicz, M.H. Austern, A.J. C. Bik, J.C. Dehnert, I. Horn, N. Leiser, G. Czajkowski, and Google Inc. Pregel: A system for large-scale graph processing. In *SIGMOD*, pages 135–146, 2010.

[23] D.G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi. Naiad: A timely dataflow system. In *SOSP*, pages 439–455, 2013.

[24] F. Reiss, K. Stockinger, K. Wu, A. Shoshani, and J.M. Hellerstein. Enabling real-time querying of live and historical stream data. In *SSDM*, page 28, 2007.

[25] C. Ren, E. Lo, B. Kao, X. Zhu, and R. Cheng. On querying historical evolving graph sequences, In *Proc. VLDB Endow.*, Vo. 4, No. 11, 2011.

[26] J. Riedy and H. Meyerhenke. Scalable algorithms for analysis of massive, streaming graphs, In *SIAM Parallel Processing for Scientific Computing*, 2012.

[27] L. Roditty and U. Zwick. A fully dynamic reachability algorithm for directed graphs with an almost linear update time. *SIAM Journal on Computing*, 45(3):712–733, 2016.

[28] A. Roy, L. Bindschaedler, J. Malicevic, and W. Zwaenepoel. Chaos: Scale-out graph processing from secondary storage. In *SOSP*, pages 410–424, 2015.

[29] A. Roy, I. Mihailovic, and W. Zwaenepoel. X-Stream: Edge-centric graph processing using streaming partitions. In *SOSP*, pages 472–488, 2013.

[30] P. Roy, A. Khan, and G. Alonso. Augmented sketch: Faster and more accurate stream processing. In *SIGMOD*, pages 1449–1463, 2016.

[31] S. Salihoglu and J. Widom. GPS: A graph processing system. In *SSDBM*, pages 22:1–22:12, 2013.

[32] X. Shi, B. Cui, Y. Shao, and Y. Tong. Tornado: A system for real-time iterative analysis over evolving data. In *SIGMOD*, pages 417–430, 2016.

[33] Y. Shiloach and S. Even. An on-line edge-deletion problem. *Journal of the ACM*, 28(1):1–4, 1981.

[34] J. Shun and G.E. Blelloch. Ligra: A lightweight graph processing framework for shared memory. In *PPoPP*, pages 135–146, 2013.

[35] I. Stanton and G. Kliot. Streaming graph partitioning for large distributed graphs. In *SIGKDD*, pages 1222–1230, 2012.

[36] T. Suzumura, S. Nishii, and M. Ganse. Towards large-scale graph stream processing platform. In *WWW Companion*, pages 1321–1326, 2014.

[37] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J.M Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham, et al. Storm@ twitter. In *SIGMOD*, pages 147–156, 2014.

[38] C. Tsourakakis, C. Gkantsidis, B. Radunovic, and M. Vojnovic. Fennel: Streaming graph partitioning for massive scale graphs. In *WSDM*, pages 333–342, 2014.

[39] K. Vora, G. Xu, and R. Gupta. Load the edges you need: A generic I/O optimization for disk-based graph processing. In *USENIX ATC*, pages 507–522, 2014.

[40] K. Vora, R. Gupta, and G. Xu. Synergistic Analysis of Evolving Graphs. In *TACO*, Vol. 13, No. 4, pages 32:1–32:27, 2016.

[41] K. Vora, S-C. Koduru, and R. Gupta. ASPIRE: Exploiting Asynchronous Parallelism in Iterative Algorithms using a Relaxed Consistency based DSM. In *OOPSLA*, pages 861–878, 2014.

[42] K. Wang, A. Hussain, Z. Zuo, G. Xu, and A. A. Sani. Graspan: A single-machine disk-based graph system for interprocedural static analyses of large-scale systems code. In *ASPLOS*, 2017.

[43] K. Wang, G. Xu, Z. Su, and Y. D. Liu. GraphQ: Graph query processing with abstraction refinement. In *USENIX ATC*, pages 387–401, 2015.

[44] M. Wu, F. Yang, J. Xue, W. Xiao, Y. Miao, L. Wei, H. Lin, Y. Dai, and L. Zhou. Gram: Scaling graph computation to the trillions. In *SoCC*, pages 408–421, 2015.

[45] M. Yuan, K-L. Wu, G. Jacques-Silva, and Y. Lu. Efficient processing of streaming graphs for evolution-aware clustering. In *CIKM*, pages 319–328, 2013.

[46] M. Zaharia, M. Chowdhury, M.J Franklin, S. Shenker, and I. Stoica. Spark: cluster computing with working sets. In *HotCloud*, 2010.

[47] M. Zaharia, T. Das, H. Li, S. Shenker, and I. Stoica. Discretized streams: an efficient and fault-tolerant model for stream processing on large clusters. In *HotCloud*, 2012.

[48] E. Zeitler and T. Risch. Massive scale-out of expensive continuous queries, In *Proceedings of the VLDB Endowment*, Vol. 4, No. 11, 2011.

[49] X. Zhang, N. Gupta, and R. Gupta. Pruning dynamic slices with confidence. In *PLDI*, pages 169–180, 2006.

[50] X. Zhang, R. Gupta, and Y. Zhang. Precise dynamic slicing algorithms. In *ICSE*, pages 319–329, 2003.