Version Traveler: Fast and Memory-Efficient Version Switching in Graph Processing Systems

Xiaoen Ju   Dan Williams[†]   Hani Jamjoom[†]   Kang G. Shin
University of Michigan   [†]*IBM T.J. Watson Research Center*

Abstract

Multi-version graph processing, where each version corresponds to a snapshot of an evolving graph, is a common scenario in large-scale graph processing. Straightforward application of existing graph processing systems often yields suboptimal performance due to high version-switching cost. We present *Version Traveler (VT)*, a graph processing system featuring fast and memory-efficient version switching. VT achieves fast version switching by (*i*) representing differences among versions as *deltas* and (*ii*) constructing the next version by integrating the in-memory graph representation of the current version with the delta(s) relating the two versions. Furthermore, VT maintains high computation performance and memory compactness. Our evaluation using multi-version processing workloads with realistic datasets shows that VT outperforms PowerGraph—running 23x faster with a 15% memory overhead. VT is also superior to four multi-version processing systems, achieving up to 90% improvement when jointly considering processing time and resource consumption.

1 Introduction

Multi-version graph processing is an important and common scenario in big data analytics. In such a scenario, each version corresponds to a snapshot of an evolving graph; a graph processing system iterates over all input versions and applies a user-defined algorithm to them, one at a time. Multi-version graph processing enables the analysis of characteristics embedded across different versions. For example, computing the shortest distance between two users across multiple versions of a social network captures the varying closeness between them [26]. Computing the centrality scores of scientific researchers across multiple versions of a co-authorship graph demonstrates their evolving impact [17].

A key element in multi-version graph processing is efficient *arbitrary local version switching*. Version switching refers to the preparation of the next to-be-processed version after computation completes on the current version. Such a procedure can be arbitrary, because the sequence of to-be-processed versions cannot be predetermined by the underlying system. It is local in that the next version commonly resides in the vicinity of the current version, demonstrating version locality.

Arbitrary local version switching has not been fully addressed before, in particular, from the perspective of the entire multi-version processing workflow. Due to version unawareness, mainstream systems, such as Pregel [23] and PowerGraph [11], perform version switching by discarding the in-memory representation of the current version and loading the next version in its entirety from persistent storage. Such an approach incurs substantial version switching time. Existing multi-version processing systems [10, 17, 19, 22] expedite the version-switching procedure by graph-delta integration. Specifically, the next version is constructed by integrating the current version with the delta representing the difference between the two versions. Albeit efficient, they either lack the support for arbitrary local switching [10, 19], incur high neighbor access penalty during computation [17], or lead to high memory overhead [22].

Towards efficient support for arbitrary local version switching, a system must balance contradicting requirements among three design dimensions: *extensibility*, *compactness*, and *neighbor access efficiency*. Extensibility refers to the easiness of creating a new graph version by extending the current one. Compactness refers to the memory overhead related to version-switching support. Neighbor access efficiency refers to the speed of neighbor access by a graph computing engine.

We present *Version Traveler (VT)*, a multi-version graph processing system enabling fast arbitrary local version switching. From a holistic view, VT balances the requirements in all three dimensions of the design space. VT consists of two novel components: (*i*) a *hybrid-com-*

pressed-sparse-row (CSR) graph supporting fast delta integration while preserving compactness and neighbor access speed during graph computation, and (*ii*) a *version delta cache* that stores the deltas in an easy-to-integrate and compact format. Conceptually, the hybrid-CSR graph represents the common subgraph shared among multiple versions in the CSR format. As a result, the subgraph is compactly stored in memory and yields high neighbor access speed—both known advantages of the CSR format. Differences among versions are absorbed by a hierarchical vector-of-vectors (VoV) representation and placed in the delta cache, leading to high version-switching speed thanks to its ability to overcome CSR's lack of extensibility.

We have implemented Version Traveler inside PowerGraph [11] by augmenting its graph representation layer with VT's hybrid-CSR graph and version delta cache. Our evaluation with realistic graphs shows that VT significantly outperforms PowerGraph in multi-version processing: VT runs 23x faster with a mere 15% memory overhead. VT also outperforms designs proposed in state-of-the-art multi-version processing systems, such as log delta, bitmap delta, and multi-version-array, achieving up to 90% improvement when jointly considering processing time and resource consumption.

The contributions of this paper include:

- the formulation of the arbitrary local version switching problem in the context of multi-version graph processing,
- a method for arbitrary local version switching with a holistic view, considering neighbor access speed, version switching speed, and compactness,
- the design of Version Traveler, a graph processing system balancing the above three requirements with two novel components, and
- the demonstration of VT's superior performance compared to state-of-the-art graph processing systems via extensive evaluation.

2 Multi-Version Graph Processing

In this section, we first discuss the characteristics of multi-version graph processing workloads, followed by a discussion on its workflow. We then summarize related work, analyze the design space for efficient multi-version graph processing systems, and discuss challenges.

2.1 Workload Characteristics

In multi-version graph processing, version switching commonly demonstrates randomness and locality. Version switching is *arbitrary*, in that the next version may precede or succeed the current version in the graph evolution.[1] Such a switching sequence may be dynamic, unable to be predetermined by the graph processing system. Version switching is *local*, in that the next version commonly resides within the vicinity—in terms of similarity—of the current one in the graph evolution.

We exemplify the demand for arbitrary local version switching with three examples. First, suppose we need to identify the cause of the varying distance between two users in a social network [26]. For simplicity, assume that the distances in versions i and k are different and that the distance changes only once along the evolution from versions i to k. If, after processing version $j = \frac{i+k}{2}$, a binary-search-style exploration algorithm finds that the distance in that version remains the same as that in version k but differs from that in version i, then the algorithm would invoke another iteration of shortest distance computation for version $m = \frac{i+j}{2}$. The switching from versions j to m is arbitrary for the supporting graph processing system. In terms of locality, although the search may oscillate between versions with high dissimilarity at the beginning, the version locality increases exponentially with the progress of the execution.

Second, in interactive big data analytics, an analyst may rerun an algorithm on a graph version, after digesting the results of the previous execution. Which version should be processed next depends on the analyst's understanding of the existing results, as well as his/her domain knowledge and intuition. This leads to arbitrary version switching from the perspective of the graph processing system. As for locality, such analysis commonly follows a refinement procedure, where significant efforts are required to zoom in and conduct in-depth analysis on a cluster of versions within the vicinity of each other.

Third, in a collaborative data analytics environment, both datasets and computing power are shared among users [5]. Individual tasks—each targeting a graph version—can be combined by the processing system, leading to multi-version graph processing. Since the next request may be enqueued during the processing of the current version and may target a version preceding or succeeding the current version in the graph evolution, version switching is arbitrary. Regarding locality, independently-submitted tasks may target similar versions. Such is the case, for example, where various algorithms are employed to capture and understand trending events in an evolving social network.

2.2 Workflow

A typical multi-version graph processing workflow is divided into multiple iterations. In each iteration, an arbitrary local graph version is processed. Systems de-

[1] More broadly, in a non-linear graph evolution scenario [5], the next version may reside in a different branch than the current version.

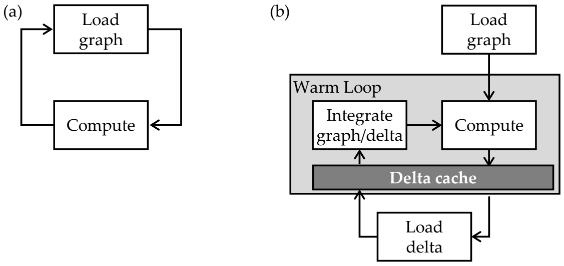

Figure 1: Version switching workflow, (a) with and (b) without the use of deltas

signed for individual graph processing tasks are unable to recognize or take advantage of the evolution relation among versions. Treating each version as a standalone graph, such systems first fully load the version from persistent storage into memory and then execute a user-defined graph algorithm over it (cf. Figure 1a).

When versions of a working set share a substantial common subgraph, working with deltas—representations of the differences between graph versions—can be more efficient. Figure 1b shows the multi-version processing workflow with deltas. After the first version is loaded and processed, switching to a subsequent version can be achieved by integrating the current version in memory with deltas relating the current and the next versions. In general, deltas are much smaller than full graphs [26,31]. As a result, they can be cached in memory, further improving the efficiency of version switching.

2.3 Related Work on Graph/Delta Designs

We focus the discussion of related work on in-memory graph and delta representations, because they determine the efficiency of the switching loop (cf. Figure 1b).[2] For graphs, we specifically focus on representations related to neighbors of vertices, because they differentiate graphs from regular table-form datasets.

As a result, we exclude other active research directions, such as programming paradigms [11, 14, 15, 20, 23, 24, 30, 32], out-of-core processing [19, 27], load balancing [16], failure recovery [28], streaming [10], dataflow-based processing [9,12], and performance evaluation [21]. We also exclude work within the broad scope of multi-version processing but not dedicated to in-memory graph/delta design in the context of arbitrary local version switching. Examples are streaming processing [10, 19],[3] parallel multi-version processing [13], multi-version algorithm design framework [26], and multi-version dataset management [5,6]. [4]

We study related work by asking three questions:

- Does it provide high computation performance? In particular, does it support fast access of the neighbors of a vertex?[5]
- Does it support fast version switching?
- Does it store graphs and deltas compactly?

Graph. We study two common graph representations: compressed sparse row (CSR), adopted in PowerGraph [11] and GraphX [12], and a vector-of-vectors (VoV) design, adopted in Giraph [1].

In CSR (cf. Figures 2b and 2c), all neighbors are packed in an array. A pointer array maintains the address of the first neighbor of each vertex. The set of neighbors for vertex i is thus marked by the values of vertices i and $i+1$ in the pointer array. This representation enables fast access to a vertex's neighbors. Version switching is slow, however. This is because modifying a vertex's neighbor affects pointers and neighbors of all vertices following the one being modified.

As for VoV (cf. Figures 2d and 2e), the first-level vector functions as the pointer array in CSR, locating the neighbors of a vertex according to the vertex id. Each second-level vector represents the neighbors of a vertex. This format also supports fast neighbor access. In addition, the neighbors of a vertex can be modified without affecting other vertices, thus enabling fast version switching. Its shortcoming is the memory overhead due to maintaining auxiliary information, such as the start and end positions of each vertex's neighbors.[6]

Delta. Previous work has used a compact log-format structure to represent deltas in streaming processing [10]. A log delta consists of an array of log entries, each specifying an edge via its source and destination vertex ids (and an optional edge id) and whether the edge should be added or removed (i.e., an opcode). Log deltas are

[2] Both graphs and deltas may have different representations in memory and on disk. We focus on in-memory representations, due to their significance in the warm loop of the version switching workflow.

[3] Streaming processing is a special case of multi-version graph processing, where version switching is always forward and versions are only processed once. They are insufficient for the general multi-version scenario, where switching is arbitrary and a cached version is repeatedly accessed by multiple algorithms.

[4] Such work investigates the tradeoff between storage space and recreation speed of a dataset version, focusing on organizing versions on disk instead of in-memory data structure optimization.

[5] In this paper, we equate computation performance with neighbor access efficiency for two reasons. First, computation related to graph algorithms affects all systems in the same way and is out of scope. Second, in the computation stage, a system supports neighbor access and vertex/edge data access. Assuming the storage of data in sequence containers and their identical impact on all systems, computation performance is determined only by neighbor access efficiency.

[6] Such overhead is non-trivial. For example, a 24-byte per-vector overhead (cf. Figure 2d, "start," "end of contents," and "end of storage" pointers each consume 8 bytes) amounts to a 40% overhead for representing the entire out-neighbor array for the Amazon08 dataset [7, 8], assuming 4-byte vertex/edge ids.

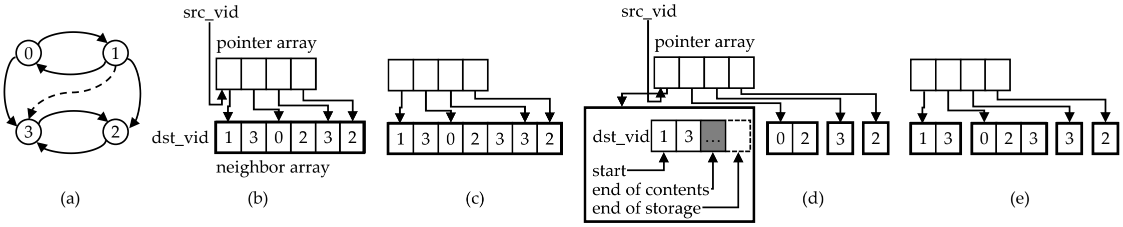

Figure 2: Graph representations: (a) illustrates two versions of a graph. A circle represents a vertex (vertex id inside) and an arrow represents an edge (edge id omitted). The first version consists of solid-arrow edges. The second version has one more edge (illustrated by a dashed arrow). (b) and (c) demonstrate the CSR representation of the out-edges of the two versions. (d) and (e) demonstrate the vector of vectors format. For clarity, each element in the neighbor array in (b)–(e) shows only the destination vertex id and omits the edge id.

compact and have no negative impact on the neighbor access efficiency during graph processing. This is because log deltas are conceptually applicable to all graph representations as-is. During graph-delta integration, log deltas are fully absorbed in the graph version. The cost of graph-delta integration is high, however, because all log entries in deltas relating the current and the next versions need to be applied during version switching.

Alternatively, a system could co-design graph and delta representations to minimize the integration cost. For example, affected neighbor vectors of a VoV graph may be copied and updated in a delta, reducing version switching to simple and fast vector pointer updates but losing compactness. LLAMA [22] partially mitigates the compactness loss by separating modified neighbors related to a version into a dedicated consecutive area in the neighbor array, avoiding copying unmodified neighbors. CSR's pointer array is transformed to a two-level translation table. The first level consists of per-version indirection tables, each bookkeeping a set of second-level pages associated with a version. A second-level page contains a series of vertex records—equivalent to a fragment of CSR's pointer array—with each record indicating the start of a vertex's neighbors.[7] LLAMA's version switching incurs nearly zero time cost: conceptually, only an indirection table pointer needs to be updated. Its use of page-level copy-on-write for the second-level pages holding vertex records, nevertheless, requires the copy of an entire page even if only one vertex in the page has a modified neighborhood, hindering its compactness.

GraphPool [17] maintains the union of edges across all versions in the graph. Its deltas are per-version bitmaps over the graph's edge array, where a bitmap's n-th bit indicates the existence of the corresponding edge in that version. Version switching is simple: a bitmap pointer is adjusted to point to the next version. In the computation stage, however, this approach requires bitmap checking for determining whether an edge exists in the current version, incurring neighbor access penalty.

2.4 Design Dimensions and Challenges

Summarizing lessons learned from related work, we point out that the design of graph and delta must balance between three dimensions: extensibility, compactness, and neighbor access efficiency.

Extensibility. Efficient version switching requires that a delta be easily integrated with a graph. From a data structure perspective, it requires that the neighbors of a vertex be easily extended to reflect the evolution from one version to another. This, in turn, requires that either the data structure representing the neighbors of a vertex support efficient modification (i.e., insertion and removal) or the collection of the neighbors of a version be easily replaced by that of another version.

Compactness. Compact graph and delta representations enable caching a large number of versions, leading to low delta cache miss rate and high version switching efficiency. Moreover, real-world large graphs commonly have millions of vertices and millions or billions of edges, making the compactness of the neighborhood data structure a primary requirement.

Access Efficiency. A common and crucial operation during computation is to access a complete collection of neighbors for a vertex. Fast neighbor access requires limiting the number of lookups in the integrated graph/delta data structure. Ideally, only one lookup is sufficient for locating the first neighbor of a vertex. The remaining neighbors can then be accessed sequentially.

The main design challenge is to carefully balance the requirements from the above three dimensions and co-design graph and delta representations such that they are extensible, compact, and efficient in neighbor access. Achieving the balance is difficult, as witnessed by exist-

[7] Neighbors belonging to the same vertex but stored in separate areas—each containing per-version modifications—are concatenated via *continuation records* such that only one start position needs to be maintained for a vertex's neighbors per version.

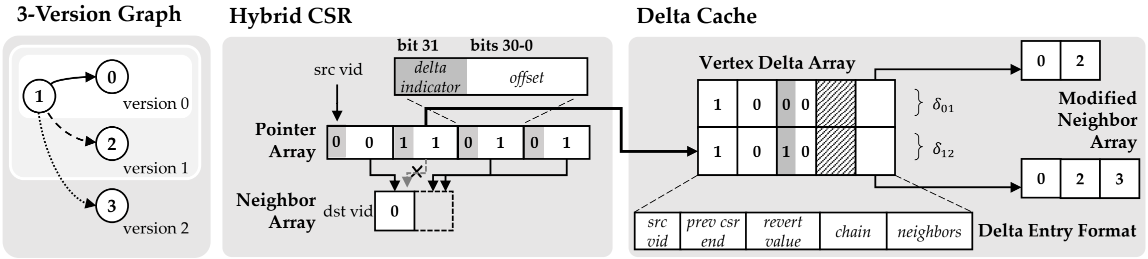

Figure 3: Hybrid graph representation

ing designs, because those requirements commonly lead to contradicting design choices.

3 Version Traveler

We introduce Version Traveler (VT), a graph processing system that features a graph/delta co-design achieving compactness, extensibility, and access efficiency. Residing in the core of VT are two innovative components—a hybrid graph and a hybrid delta cache—bringing together fast neighbor access and compactness of CSR and high extensibility of VoV.

3.1 Hybrid Graph

VT's hybrid graph augments CSR in a way that achieves extensibility while remaining compact and efficient in neighbor access (cf. Figure 3). It avoids costly in-place modification to CSR's neighbor array by storing vertices with a modified neighborhood in a version delta cache.

CSR's neighbor array is created during the loading of the first version—also referred to as the root version—and then remains constant. Each subsequent version is loaded into the delta cache, in the form of a series of *vertex delta entries*. Each entry contains information related to the updated neighbors of a vertex, as well as metadata to support neighbor access and version switching (cf. Figure 3). VT reserves a *delta indicator bit* in each entry of CSR's pointer array to indicate the placement of a vertex's neighbors for the current version: in CSR's neighbor array or in the vertex delta cache.

Neighbor Access. In a conventional CSR, neighbors of vertex *vid* are bounded by the pointers of vertices *vid* and $vid+1$. For VT's hybrid CSR, neighbor access may be directed to either CSR's neighbor array or the *neighbors* field of a delta entry, depending on whether the neighbors are stored (cf. `access_neighbors` in Algorithm 1). Each delta entry maintains the end position of the preceding vertex's neighbors in CSR's neighbor array (in the *prev_csr_end* field), such that the end position of *vid* can be determined regardless of whether neighbors of $vid+1$ are stored in CSR's neighbor array or the delta cache (cf. lines 4–7).

Algorithm 1 Neighbor Access and Delta Application

```
 1: procedure ACCESS_NEIGHBORS($vid$)
 2:     if csr_ptrs[$vid$].in_delta = true then
 3:         return cache[csr_ptrs[vid]].nbrs
 4:     else if csr_ptrs[$vid+1$].in_delta = false then
 5:         return csr_nbrs[csr_ptrs[$vid$], csr_ptrs[$vid+1$]]
 6:     else return csr_nbrs[csr_ptrs[$vid$],
 7:             cache[csr_ptrs[$vid+1$]].prev_csr_end]
 8: procedure DELTA_APPLICATION($\delta_{ij}, opcode$)
 9:     for $e$ in $\delta_{ij}$ do
10:         if $opcode = apply$ then                    ▷ apply $\delta_{ij}$
11:             csr_ptrs[$e.src\_vid$] ← offset($e$)
12:             csr_ptrs[$e.src\_vid$].in_delta ← true
13:         else csr_ptrs[$e.src\_vid$] ← $e.revert\_value$    ▷ revert $\delta_{ij}$
```

Delta Application and Reversion. VT performs version switching by applying or reverting deltas (cf. `delta_application` in Algorithm 1). When applying δ_{ij} to switch from versions i to j, VT iterates over entries belonging to δ_{ij} in the delta cache and, for each entry, updates the corresponding entry (according to the *src_vid* field) in the CSR pointer array with the delta entry's offset in the delta array and sets the delta indicator bit. Reverting δ_{ij} consists of restoring the *revert value* field—which contains the saved value for version i's CSR pointer entry—to the corresponding entry in the CSR pointer array for each entry in δ_{ij}.

Example. We use a 3-version graph in Figure 3 to illustrate neighbor access and version switching. The hybrid CSR in Figure 3 represents the state of version 2. The three out-neighbors of vertex 1 can be accessed from its delta entry (δ_{12}). For vertex 0, neighbor access requires obtaining the start position from its CSR pointer, due to its cleared delta indicator bit, and the end position from *prev_csr_end* of vertex 1's delta entry. The difference between the two is 0 ($0-0=0$), indicating that vertex 0 has no out-neighbors. In order to switch from versions 2 back to 1, VT reverts δ_{12}, which has only one entry related to vertex 1. Its *revert value* field, of which the delta indicator bit is set and the offset is 0, is restored to vertex 1's CSR pointer entry. After reversion, vertex

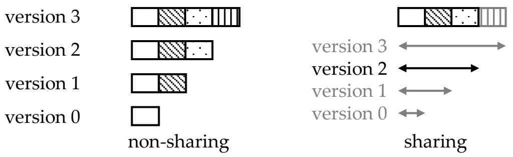

Figure 4: Illustration of the concept of Sharing

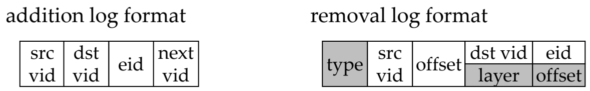

Figure 5: Delta log format

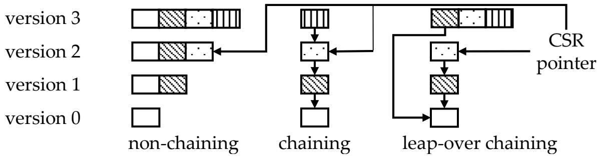

Figure 6: Illustration of the concept of Chaining

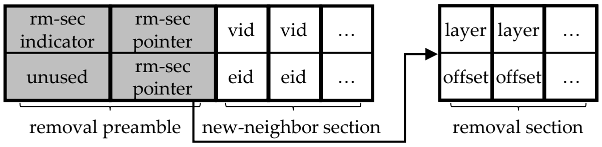

Figure 7: Neighbor vector format

1's CSR pointer entry will point to the first entry in the delta array, which corresponds to vertex 1's delta entry for version 1.

3.2 Hybrid Delta

For simplicity, in Section 3.1, we assume that a delta entry maintains the entire neighbors of a vertex (cf. Figure 3). This is memory-inefficient for vertices with a large number of neighbors and small amount of per-version modifications, due to numerous redundant copies of neighbors. To improve compactness, we propose two complementary solutions: *Sharing* and *Chaining*. Sharing preserves access efficiency and trades extensibility for compactness. Chaining preserves extensibility and trades access efficiency for compactness.

3.2.1 Sharing

Concept. Sharing reduces the memory footprint by merging a vertex's delta entries spanning multiple versions into one shared entry. Figure 4 shows an example with four versions of a vertex, each adding one neighbor to its base. When they share a delta entry, there exists only one neighbor vector, containing the neighbors of the vertex related to the current version being processed. The challenge is to compactly specify how the shared vector is modified during version switching. VT maintains this information in addition and removal *delta logs*.

Delta Representation. In the Sharing mode, VT does not create delta cache entries with copies of modified neighbor arrays. Instead, it creates log entries: specifying the neighbors it would have added to or removed from the neighbor array in an addition or removal log. Each entry in the addition log array (cf. Figure 5) contains the source and target vids of the added edge, as well as the edge id. Logs associated with the same vertex (the source vid in the out-neighbor case) are continuously stored. A *next vid* field indicates the start position of the logs associated with a subsequent vertex. The format of a removal log entry—gray fields apart—is similar. Its *offset* field refers to the offset within the neighbor array where the removal should take place.

Neighbor Access. Sharing has no effect on neighbor access. When multiple versions of a vertex share a neighbor vector, the CSR pointer header points to the same delta entry containing the vector for all versions.

Delta Application/Reversion. In the Sharing mode, version switching resorts to log-based delta application/reversion, similar to streaming processing [10, 19]. During delta application, for an neighbor addition, the neighbor is simply appended to the end of the neighbor vector. For a removal, VT removes the neighbor at the offset according to the *offset* value in the removal log entry. Delta reversion follows the inverse procedures.

3.2.2 Chaining

Concept. Chaining refers to the representation of a vertex's neighbors with a chain of vectors, each containing a subset of neighbors and capturing the difference between the version associated with it and its base version. In Figure 6, with each version chained onto its base, only one neighbor needs to be maintained per version. Redundant copies are eliminated, improving compactness. Extensibility remains the same: to switch from versions 1 to 2, for example, we need to adjust only the CSR pointer to version 2's delta entry, regardless of whether that entry's neighbor vector is chained onto another. Access efficiency decreases in Chaining, because of the need to switch among multiple neighbor vectors. Chaining imposes two new challenges to delta design: chaining beyond the base, called *Leap-Over Chaining*, and removal from ancestors, called *indirect removal*.

Delta Representation. Leap-Over Chaining intends to accelerate neighbor access. In Figure 6, with each version chained onto its base, the neighbor access for later versions leads to considerable performance hit, limiting Chaining to a small set of adjacent versions in the graph evolution relation. Leap-Over Chaining enables the chaining of a delta entry on an indirect ancestor version. For instance, version 3 in Figure 6 can be chained onto version 0.

To support Chaining, in particular Leap-Over Chaining, we introduce a *chaining* field to the delta entry format (cf. Figure 3). When Chaining is disabled,[8] the entry is a standalone entry with a complete copy of neighbors. When Chaining is in use, VT saves a pointer to the entry upon which the current one is based along the chain to the latter's *chaining* field. Similar to CSR pointers, a chaining pointer uses its most significant bit to indicate whether the offset is for CSR or for the delta array.

To support indirect removal, a neighbor vector is divided into two sections: a new-neighbor section and a removal section (cf. Figure 7). An element in the new-neighbor section represents a new neighbor added to the vertex in the current version. An element in the removal section corresponds to a removed element, with *layer* indicating the neighbor vector in the chain where the removal should take place and *offset* the position of the to-be-removed neighbor within the vector. To improve compactness, we overload the first element in the new-neighbor section: it is marked with a special flag if the removal section exists, in which case the second element contains a pointer to the removal section;[9] otherwise it contains the first added neighbor.

Effect on Removal Log. Since Chaining introduces the separation of new-neighbor and removal sections, Sharing's removal log format needs to be adjusted (cf. gray fields in Figure 5). Specifically, a *type* field is added to differentiate the two sections. A {layer, offset} pair in a removal section is stored similarly to a {vid, eid} pair in a new-neighbor section. During delta application, if a removal takes place in the current vertex's new-neighbor section, then the corresponding {vid, eid} pair is removed. Otherwise, the {layer, offset} pair is inserted to the current vertex's removal section. The inverse procedure achieves delta reversion.

Neighbor Access. Given a vertex, VT first locates its entry via the CSR pointer header. If the neighbors are stored in a delta array entry whose Chaining mode is turned on, then VT iteratively accesses neighbors stored in entries along the chain, skipping neighbors that no longer exist in the current version using the removal sections. Otherwise, it follows the common neighbor access procedure, as described in Section 3.1.

Delta Application/Reversion. Except for Chaining's effect on removal logs, both delta application and reversion follow the description in Section 3.1.

3.2.3 Relationship between Chaining and Sharing

Chaining and Sharing are similar in that they both aim at reducing memory consumption by storing only the difference among versions. Sharing is a good choice when the size of neighbors is large and the delta size is small. A large neighborhood leads to a considerable gain in compactness over a full-neighbor-copy approach, whereas a small delta entails a moderate cost for log-based version switching. Chaining is useful when both the sizes of neighbors and delta are large. Similar to Sharing, a large neighbor size leads to a substantial gain in compactness for Chaining. A large delta entails Chaining's superiority to Sharing, due to the avoidance of the latter's costly log-based version switching procedure.

Another way to compare the two is when the concatenation of neighbors occurs. Chaining performs the concatenation in a chain at the computation stage. Sharing performs the concatenation at the version switching stage. Due to the different delta formats used in Chaining and Sharing, the concatenation in Chaining is lighter-weight than that in Sharing. As for the number of concatenation performed for a vertex, the concatenation in Chaining needs to take place when a vertex's neighbors are accessed. The cost of concatenation in Chaining is thus magnified if a vertex is iteratively processed by an algorithm. The concatenation in Sharing is, in contrast, guaranteed to be once per vertex per version switching.

VT supports Sharing and Chaining as operation modes, complementing the default full-neighbor-copy mode (referred as Full mode). It enables them when the estimated cost of version switching and the potential impact on the computation stage are justified by the amount of memory saving. The current VT implementation supports flexible threshold-based policies: when creating a new delta for a vertex, VT feeds the number of neighbors in its base version and the current delta size related to that vertex to a configurable policy arbitrator function, which determines the activation of Sharing or Chaining.

3.3 Implementation

We implement VT by integrating it with PowerGraph [11], replacing the latter's graph representation with VT's hybrid CSR graph and delta/log arrays. VT operates seamlessly with PowerGraph's computation engine layer, thanks to its full support of the same computation-stage graph abstraction viewed from an engine. This also demonstrates VT's broad applicability to existing graph processing systems.

[8] That is, setting all bits in *chaining* to one.

[9] The first two elements are also referred to as *removal preamble*.

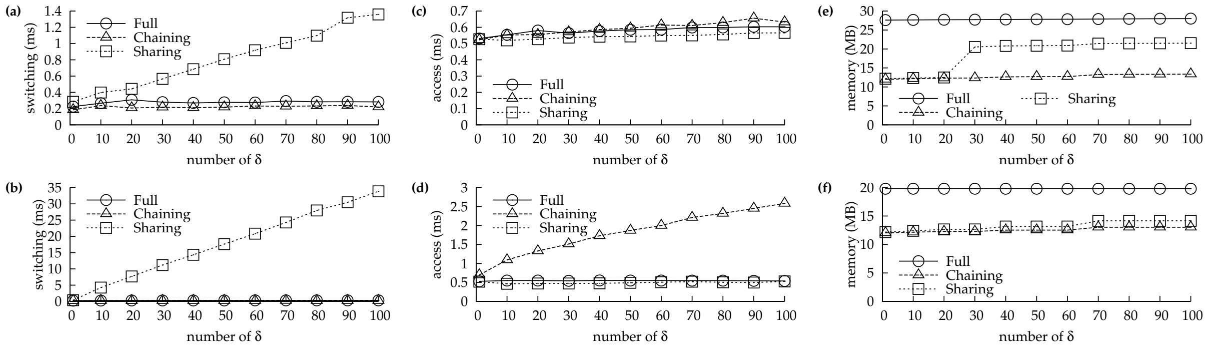

Figure 8: Microbenchmark results (top row for additions and bottom row for removals)

4 Evaluation

We first demonstrate the three-way tradeoff among extensibility, compactness, and access efficiency, showing the relative advantage of Full, Chaining, and Sharing. We then compare the performance of VT against PowerGraph and several multi-version reference designs.

4.1 Microbenchmark

Design. The goal of microbenchmarking is to evaluate the relative effectiveness of Full, Sharing, and Chaining in balancing the three-way tradeoff. Since the overall tradeoff on a graph is the accumulative effect of the same tradeoff on each vertex, we conduct microbenchmarking from a vertex's perspective. Trends in the microbenchmark results are applicable to varying graph sizes, given the accumulative nature of the per-vertex tradeoff.

We construct a graph with 1000 identical high-degree stars. For each star, only the *center vertex* has a nonempty set of out-neighbors—default to 1000. Each center vertex thus provides the opportunity for an in-depth study of the per-vertex tradeoff. To evaluate it in a multi-version scenario, we create two versions: a source version and a target version. The target version differs from the source by randomly adding or removing out-neighbors of center vertices.

Two key factors related to a delta are its size and the ratio of additions to removals. Prior work shows that the difference between consecutive versions is commonly within 1% of the graph size [26]. For each star, given the default 1000 edges in the base version, we vary the delta size from 1 to 100, corresponding to 0.1% to 10% of the size of the star. The total vertices and edges in the graph thus vary between 0.9 to 1.1 million. We also fix the operation types in a delta: a delta consists of either edge additions or edge removals.

We evaluate extensibility by measuring the version switching time from the source version to the target version, neighbor access efficiency by measuring the time for iterating through all the out-neighbors of center vertices in the target version, and compactness by measuring the memory used for maintaining the graph connectivity information of both versions. All measurements are conducted on a host with 8 3GHz vCPUs and 60GB memory.

Version Switching. Figures 8a and 8b compare the version switching performance. The performance of Full and Chaining is comparable and remains constant, regardless of the edge modification types in deltas or the delta size, because both approaches require adjusting only CSR pointer values for center vertices. The cost of Sharing linearly grows with the delta size, due to the need to parse a log array whose size is proportional to the delta size. Comparing edge additions with removals, the cost of the former is significantly lower than the latter. This is because, additions translate to appending neighbor records to the end of the neighbor vector and removals involve data movement within the vector.

Access. Figures 8c and 8d compare the neighbor access speed. Full and Sharing perform equally well for both additions and removals. Since the cost of neighbor access is proportional to the neighborhood size, it linearly increases and decreases with the size of delta in the cases of addition and removal, respectively. Chaining leads to the worst performance in both cases, due to its cost of indirection. The cost is moderate in the case of edge additions, because there is one and only one indirection during neighbor access—that is, the switching from the newly added neighbors to the existing ones—regardless of the delta size. The cost of indirection becomes significant for edge removals, because each removal separates a previously continuous neighbor range into two, introducing one more indirection during neighbor access. The cost thus linearly grows with the delta size.

Memory. Figures 8e and 8f show the memory footprint. In both addition and removal cases, Chaining and Sharing lead to significant memory savings comparing with

Table 1: Graphs, algorithms, and reference designs

dataset	V (M)	E (M)	description
Amazon08	0.7	5.2	similarity among books
Dblp11	1.0	6.7	scientific collaboration
Wiki13	4.2	101.4	English Wikipedia
Livejournal	5.4	79.0	friendship in LiveJournal social network
Twitter	41.7	1468.4	Twitter follower graph
Facebook	0.1	1.6	friendship in regional Facebook network
GitHub	1.0	5.7	collaboration in software development

algorithm	description
nop	access neighbor and return
bipart	max matching in a bipartite graph
cc	identify connected components
PageRank	compute rank of each vertex
sssp	single-source shortest path
tc	triangle count

ref. design	description
csr	use CSR graph and log delta
log	use VoV graph and log delta
bitmap	maintain union of neighbors in all versions in VoV graph and use bitmap delta
m-array	use multi-version-array graph/delta

Full. Intuitively, the cost of Full linearly grows with the delta size in the addition case and linearly decreases in the removal case. Our measurements, however, show mostly constant memory footprints in both cases, due to (1) the capacity doubling effect and (2) no capacity reduction upon removal in the vector implementation in our testbed (glibc 2.15). For Chaining and Sharing, the memory footprint grows with the size of delta, regardless of the type of edge modifications. This is because, for both additions and removals, Chaining needs to maintain the modifications either in the neighbor vector (for additions) or in the removal section (for removals). Similarly, Sharing maintains the modifications in the log arrays.

4.2 Macrobenchmark

Reference Designs. We compare VT with PowerGraph [11]—a high-performance system targeting individual graph processing—and four reference multi-version processing system designs (cf. Table 1) reflecting different combinations of graph and delta formats. Specifically, we evaluate CSR+log, VoV+log, VoV+bitmap, and multi-version-array. They mirror design choices made in PowerGraph, Giraph [1], GraphPool [17], and LLAMA [22], and are abbreviated to *csr*, *log*, *bitmap*, and *m-array*, respectively.

Workloads. Table 1 summarizes the datasets and algorithms. The Facebook [31] and GitHub graphs are collected as dynamically evolving graphs. The remaining five graphs are collected as static graphs [7, 8], for which deltas need to be created. Since deltas among consecutive versions are commonly within 1% of the graph size [26], we vary the delta size from 0.01% to 1%. We select $\delta = 0.1\%$ as a middle ground and show most of the evaluation results with this configuration. The total number of cached versions n varies broadly from 1 to 100. Unless otherwise specified, we use uniform add-only deltas: each delta consists of edge additions uniformly distributed over a graph. Graphs evolve linearly: version i is created by iteratively applying $\delta_{j,j+1}, j = 0 \ldots i-1$ to the root version (i.e., version 0). Version switching is local, in that all versions are within the range of $n\delta$ from the root version. Version switching is arbitrary. That is, the next version j is selected independently to the current version i, may precede or succeed i, and do not need to be consecutive to i.

Since machines with large memory and many cores become popular and affordable [25, 29], VT's evaluation focuses on single-host setting. The elimination of inter-host communication cost in graph processing stage further highlights the effect of neighbor access efficiency. All measurements except those related to the Twitter graph [18] are performed on a host with 8 3GHz vCPUs and 60GB memory. Twitter-related workloads run on a host with 32 2.5GHz vCPUs and 244GB memory.

Metrics. The requirements on extensibility, compactness, and access efficiency naturally lead to the use of time and memory consumption as two basic metrics. In addition, inspired by the resource-as-a-service model in the economics of cloud computing [4], we introduce a penalty function as a third metric: $p = (t_s + t_c)^\alpha \times m^\beta$. The penalty p is a function of the version switching time t_s, the computation time t_c, and memory consumption m. α and β are weights associated with time and memory resource. If the per-time-unit monetary cost is determined only by memory consumption, then assigning 1 to both parameters equates the penalty with the per-task monetary cost. We use $\alpha = 1$ and $\beta = 1$ in our evaluation. When appropriate, we report penalty score p in the form of *utility improvement*: the improvement of VT over a reference system *ref* is calculated as $\frac{p_{ref}-p_{vt}}{p_{ref}}$.

Delta Preparation. For each system/workload setting, deltas corresponding to versions accessed in that workload are populated in memory, according to the delta design employed by that system, before the start of the workload. For VT, we employ a threshold-based policy (cf. Section 3.2.3), determining the delta format according to the number of neighbors in its source version and switching from Full to Sharing to Chaining as the number increases. We sample the threshold space for Full-Sharing and Sharing-Chaining transitions and report the

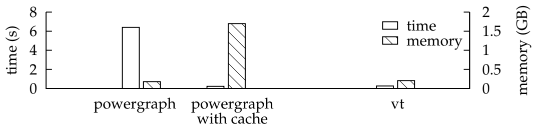

Figure 9: Comparison of VT and PowerGraph

lowest penalty score.

Comparison with PowerGraph. We evaluate the performance of PowerGraph by running SSSP on Amazon08 with ten 0.1% δs in two scenarios. First, we measure the performance of PowerGraph as-is, with a graph version loaded from persistent storage in its entirety at the beginning of each task. Second, we augment PowerGraph with full-version caching, storing each version in the working set as a full graph copy in memory.

Figure 9 shows that VT significantly outperforms PowerGraph in both scenarios. VT's processing speed is on a par with that of PowerGraph with full-graph caching and is 23x faster than that of PowerGraph without caching. This is due mainly to PowerGraph's substantial loading time when caching is disabled. VT's memory footprint is close to that of PowerGraph without caching (incurring a 15% overhead) and is only 12% of that of PowerGraph with full-graph caching—a 7.3x enhancement. Overall, VT improves utility by 86% and 95% over PowerGraph with and without caching, respectively.

Comparison with Multi-Version Designs. Figure 10 compares four multi-version designs with VT, executing nop on Amazon08 with ten 0.1% δs. *Csr* incurs prohibitive switching cost, due to CSR's low extensibility. It, nevertheless, yields the highest performance and has the smallest memory footprint. Both *log* and *bitmap* consume more memory than VT. *Bitmap* incurs a computation-stage penalty due to bitmap checking. *Log*'s switching cost is 7.4x that of VT.

Regarding *m-array*, its neighbor access time and version switching time are significantly shorter than the other designs. Its memory consumption, however, is much higher than the other. It is important to note that *m-array*'s superior performance is an outcome of efficient *implementation* of LLAMA, not a result of the multi-version-array design. This is because, after a version becomes ready for processing, all things being equal, *csr* should yield the highest neighbor access performance for the nop workload. The difference between *m-array* and *csr* is then due to the framework-related overhead: *m-array* is measured with LLAMA and *csr*—as well as the other designs—is measured with PowerGraph-based implementation. Had we ported *m-array* to PowerGraph, its performance would be at best on a par with *csr*, and thus also close to VT.

M-array's high memory consumption is a result of the multi-version-array design. For Amazon08 with 0.1% δs, each version contains 5.2K new edges. Uniformly distributed, those edges affect 5.2K vertices' neighborhood. In LLAMA, with a 16-byte vertex record[10] and a 4KB page, the entire vertex record array for the root version spans 2.7K pages, which is also the expected number of pages affected when the 5.2K vertices with modified neighborhood are uniformly distributed. This yields a 100% memory overhead in terms of per-version vertex record array—because the entire 2.7K pages containing the root multi-version array need to be copied for each version—and a 21.5% overhead when the entire graph connectivity structure (with neighbor arrays) is considered. Such an overhead is prohibitively expensive for large graphs. In contrast, VT has a smaller footprint for the root version and, more importantly, incurs only a 0.6% per-version overhead for the graph connectivity structure in its Chaining mode.

Figure 10 confirms our expectation on the advantages and shortcomings of existing designs. Given *csr*'s low extensibility and *m-array*'s high memory consumption, we focus on comparing VT with *log* and *bitmap* for the rest of the evaluation.

Comparison with *log* and *bitmap*. Figure 11 summarizes the results comparing VT with *log* and *bitmap*, each with 10 δs of size 0.1%. VT consistently outperforms both systems in all but one case. Except the Twitter-SSSP workload, VT runs 2–17% faster in average per-version processing time and achieves 17–34% memory saving and 19–40% utility improvement.

Running SSSP over the Twitter graph, VT runs 88% faster than *log* but 19% slower than *bitmap*. This is because, given the size of the dataset, the configuration of the supporting hardware, and the characteristics of the algorithm, the difference in version switching dominates the overall processing efficiency. *Log* falls far behind VT, due to the former's need of log replaying during version switching. VT's delta application, although efficient and highly parallelized, is still a heavier-weight operation compared to *bitmap*. Combining time and memory consumption, the net effect is that VT outperforms *log* by 90% and is on a par with *bitmap* in utility improvement.

Varying Deltas. We compare VT with *log* and *bitmap* by executing SSSP on Amazon08, varying the size of delta from 0.01% to 1% and the number of deltas from 10 to 100. Fixing the delta size to 0.1% and varying the number of deltas from 10 to 100, we observe that VT's utility gain remains high with respect to *log* and *bitmap* (cf. Figure 12a). Compared to *log*, VT's memory saving

[10] A vertex record consists of version id, offset into the neighbor array, number of new edges for the current version, and an optional outdegree, each occupying 4 bytes.

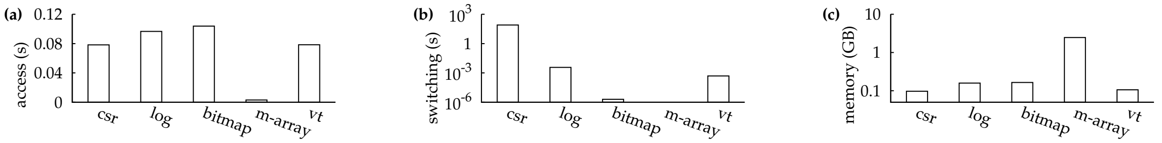

Figure 10: Comparison with existing graph/delta designs

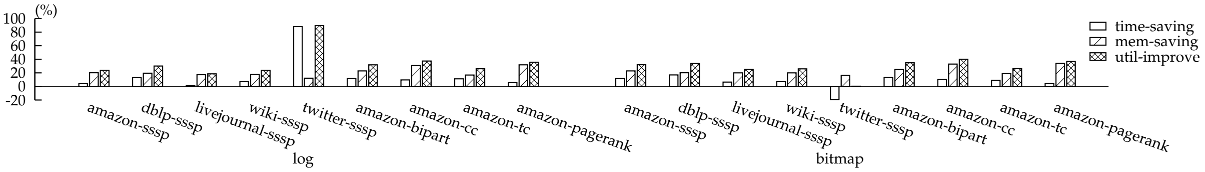

Figure 11: Comparison of VT with *log* and *bitmap* across all datasets and algorithms with 10 0.1% δs

reduces with the increasing number of deltas, because the memory consumption of the delta cache grows with the number of deltas, gradually neutralizing the benefit of the use of hybrid CSR. VT's gain due to the reduction of version switching time increases with the number of deltas, however. Overall, with these opposite trends, VT's utility gain remains high. Compared to *bitmap*, VT's memory saving remains high, because of *bitmap*'s need to maintain per-version bitmaps. VT's saving in processing time reduces, however, because the impact of *bitmap*'s saving in version switching time increases with the number of deltas, compensating for *bitmap*'s neighbor-access slowdown in the computation stage. The overall effect of these opposite trends is VT's constantly high utility gain with respect to *bitmap* across a wide range of versions.

Fixing the number of deltas to 10 and varying the size of delta from 0.01% to 1%, we observe that VT's utility gain gradually reduces (cf. Figure 12b). Compared to *log*, VT's gain peaks at $\delta = 0.01\%$, thanks to its efficient graph-delta representation. VT's gains for $\delta = 0.1\%$ and $\delta = 1\%$ are similar: larger deltas reduce VT's advantage in memory representation but amplify its reduction of version switching cost. Compared to *bitmap*, VT's utility gain remains high for $\delta = 0.01\%$ and $\delta = 0.1\%$, but drops significantly for $\delta = 1\%$. Note that, the maximum distances among versions—in terms of dissimilarity—are the same for 100 0.1% δ (in Figure 12a) and 10 1% δ (in Figure 12b). Yet, VT's gain with respect to *bitmap* is much higher in the former case. This is because VT's memory saving is more significant when *bitmap* needs to maintain a larger number of per-version bitmaps in order to track the neighbor-version relation.

Skewed and Add/Remove Workloads. Figure 13 compares VT's performance across three types of workloads, all with ten 0.1% δs. The first is a uniformly distributed add-only delta type, same as those used throughout the evaluation. The second is a skewed add-only delta, in which the probability of adding a new edge to a vertex is proportional to the latter's degree in the root version. The third is a mixed add/remove delta type, with each delta maintaining a removals/total operations ratio varying from 0.1% to 10%. VT consistently outperforms *log* and *bitmap* in all the three workloads.

Effectiveness of Optimization. Figure 14 summarizes the effectiveness of Sharing and Chaining. Reusing the workload of SSSP-Amazon with ten 0.1% δs, we first enforce a fixed delta format, measuring the performance of Full, Sharing, and Chaining individually. We then combine Full and one of the two optimization approaches and report the minimum achievable penalty. All results are then normalized to those of VT. The effectiveness of Sharing and Chaining is demonstrated by the superiority (in terms of penalty) of a combined delta preparation strategy (e.g., Full-Sharing) to both approaches when applied individually (e.g., Full and Sharing). It is also demonstrated by VT's superiority—with all three delta formats combined—to the five alternatives.

Realistic Evolving Workloads. We compare VT with *log* and *bitmap*, using two 10-version graphs generated from the evolving Facebook friendship and GitHub collaboration graphs,[11] respectively. Figure 15 shows their evolution trends. Specifically, we choose 10 consecutive days towards the end of the collected periods for the two graphs[12] and combine newly established friendship/ collaboration relations in each day into a delta. Friendship/collaboration relations existing before that 10-day

[11] In the GitHub graph, the collaboration (i.e., edge) between two users (i.e., vertices) is established when they start to work on at least one shared repository. The initial state of the graph is set to empty. Its evolution spans between March 2011 and July 2015. We generate this graph via the use of GitHub API [2] and GitHub Archive [3].

[12] For the Facebook graph, daily delta size reduces drastically towards the end of the collected period, which might be caused by limitations of the collection method. We avoid those anomalies when creating the multi-version graph for evaluation.

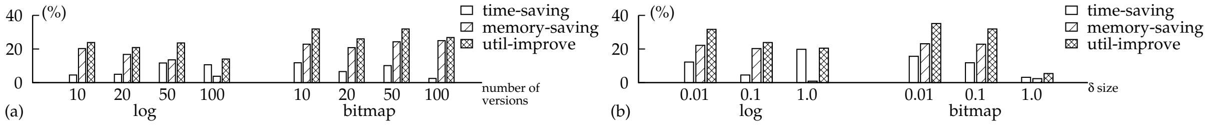

Figure 12: Varying number of δs and δ size. (a) Fixing $\delta = 0.1\%$ and varying number of versions between 10 and 100. (b) Fixing number of versions to 10 and varying δ between 0.01% and 1.0%.

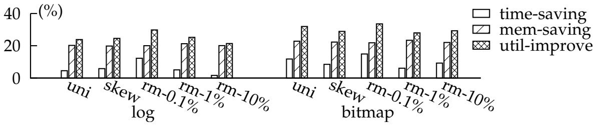

Figure 13: Uniform, skewed, and add/rm deltas

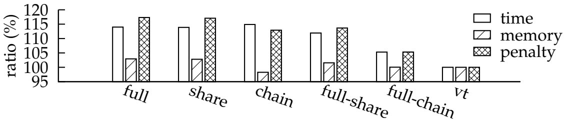

Figure 14: Effectiveness of optimization, with 10 0.1% δs (normalized to VT)

period then form the root versions of the two graphs. Figure 16 shows that VT outperforms both *log* and *bitmap* when executing SSSP over these two graphs, improving utility by 10.38–24.83%.

4.3 Discussion: Locality Revisited

At the core of VT lies the concept of locality. The effectiveness of VT depends on the high version access locality in multi-version workloads. Quantifying the locality of version access patterns, nevertheless, is difficult. In this paper, we express locality in terms of a range $n\delta$—defined by the number of deltas n and the size of delta δ—within which arbitrary version switching takes place. We have shown that VT achieves superior performance for a wide range of $n\delta$ configurations (cf. Figure 12), with respect to state of the art. Yet, VT's performance, as well as its relative gain with respect to other systems, needs careful investigation for other access patterns.

For example, for workloads featuring high computation-to-version-switching ratio and forward-only switching, we expect either *log* or a single-version system to perform the best. For such workloads, the significance of the computation-stage performance outweighs that of version switching. For example, when a large set of algorithms are applied to a loaded version, any version switching except the first one becomes a self-switching operation, incurring almost zero cost thanks to the memory management of supporting operating systems. In addition, forward-switching nullifies the need to preserve the graph representation of a version after it is processed.

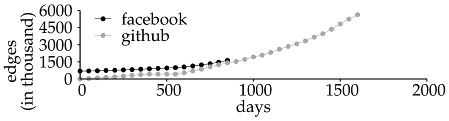

Figure 15: Evolution trends of a regional Facebook friendship graph and a GitHub collaboration graph

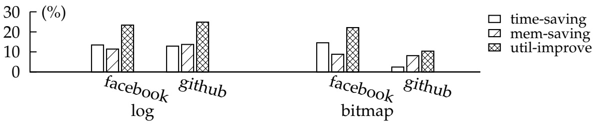

Figure 16: Performance of VT, *log*, and *bitmap* on Facebook and GitHub graphs

Thus a system optimized purely for high computation performance is favorable.[13] To efficiently handle such workloads, VT needs to be extended to support direct modification to the CSR, bypassing the shadowing effect of the delta cache. More importantly, the switching between existing operating modes of VT and this new direct modification mode, as well as other modes potentially devised in future work, requires a thorough investigation of the switching policies.

5 Conclusions

In this paper, we conducted a systematic investigation of the caching design space in multi-version graph processing scenarios, decomposing it into three dimensions: neighbor access efficiency, extensibility, and compactness. Our solution, Version Traveler, balances requirements from all three dimensions, achieving fast and memory-efficient version switching. It significantly outperforms PowerGraph and is superior to four multi-version reference designs.

Acknowledgments

We thank the anonymous reviewers and our shepherd, Indranil Gupta, for their feedback.

[13] The relative merit of *log* to a single-version system is determined, in this case, by whether constructing the next version by modifying the current one is less costly than building it from scratch.

References

[1] Apache Giraph. http://giraph.apache.org. Retrieved in Apr. 2016.

[2] Github api. https://developer.github.com/v3/. Retrieved in Apr. 2016.

[3] Github archive. https://www.githubarchive.org/. Retrieved in Apr. 2016.

[4] AGMON BEN-YEHUDA, O., BEN-YEHUDA, M., SCHUSTER, A., AND TSAFRIR, D. The resource-as-a-service (raas) cloud. In *Proceedings of the 4th USENIX Workshop on Hot Topics in Cloud Ccomputing* (Berkeley, CA, USA, 2012), HotCloud'12, USENIX Association.

[5] BHARDWAJ, A., BHATTACHERJEE, S., CHAVAN, A., DESHPANDE, A., ELMORE, A. J., MADDEN, S., AND PARAMESWARAN, A. G. DataHub: Collaborative Data Science & Dataset Version Management at Scale. In *7th Biennial Conference on Innovative Data Systems Research* (2015), CIDR '15.

[6] BHATTACHERJEE, S., CHAVAN, A., HUANG, S., DESHPANDE, A., AND PARAMESWARAN, A. Principles of dataset versioning: Exploring the recreation/storage tradeoff. *Proc. VLDB Endow. 8*, 12 (Aug. 2015), 1346–1357.

[7] BOLDI, P., ROSA, M., SANTINI, M., AND VIGNA, S. Layered label propagation: A multiresolution coordinate-free ordering for compressing social networks. In *Proceedings of the 20th international conference on World Wide Web* (2011), ACM Press.

[8] BOLDI, P., AND VIGNA, S. The WebGraph framework I: Compression techniques. In *Proc. of the Thirteenth International World Wide Web Conference (WWW 2004)* (Manhattan, USA, 2004), ACM Press, pp. 595–601.

[9] BU, Y., BORKAR, V., JIA, J., CAREY, M. J., AND CONDIE, T. Pregelix: Big(ger) graph analytics on a dataflow engine. *Proceedings of the VLDB Endowment 8*, 2 (2014), 161–172.

[10] CHENG, R., HONG, J., KYROLA, A., MIAO, Y., WENG, X., WU, M., YANG, F., ZHOU, L., ZHAO, F., AND CHEN, E. Kineograph: Taking the pulse of a fast-changing and connected world. In *Proceedings of the 7th ACM European Conference on Computer Systems* (New York, NY, USA, 2012), EuroSys '12, ACM, pp. 85–98.

[11] GONZALEZ, J. E., LOW, Y., GU, H., BICKSON, D., AND GUESTRIN, C. Powergraph: Distributed graph-parallel computation on natural graphs. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2012), OSDI'12, USENIX Association, pp. 17–30.

[12] GONZALEZ, J. E., XIN, R. S., DAVE, A., CRANKSHAW, D., FRANKLIN, M. J., AND STOICA, I. Graphx: Graph processing in a distributed dataflow framework. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)* (Broomfield, CO, Oct. 2014), OSDI '14, USENIX Association, pp. 599–613.

[13] HAN, W., MIAO, Y., LI, K., WU, M., YANG, F., ZHOU, L., PRABHAKARAN, V., CHEN, W., AND CHEN, E. Chronos: A graph engine for temporal graph analysis. In *Proceedings of the 8th ACM European Conference on Computer Systems* (2014), EuroSys '14.

[14] HOQUE, I., AND GUPTA, I. Lfgraph: Simple and fast distributed graph analytics. In *Proceedings of the First ACM SIGOPS Conference on Timely Results in Operating Systems* (New York, NY, USA, 2013), TRIOS '13, ACM, pp. 9:1–9:17.

[15] KANG, U., TSOURAKAKIS, C. E., AND FALOUTSOS, C. Pegasus: A peta-scale graph mining system implementation and observations. In *Proceedings of the 2009 Ninth IEEE International Conference on Data Mining* (Washington, DC, USA, 2009), ICDM '09, IEEE Computer Society, pp. 229–238.

[16] KHAYYAT, Z., AWARA, K., ALONAZI, A., JAMJOOM, H., WILLIAMS, D., AND KALNIS, P. Mizan: A system for dynamic load balancing in large-scale graph processing. In *Proceedings of the 8th ACM European Conference on Computer Systems* (New York, NY, USA, 2013), EuroSys '13, ACM, pp. 169–182.

[17] KHURANA, U., AND DESHPANDE, A. Efficient snapshot retrieval over historical graph data. In *Proceedings of the 2013 IEEE International Conference on Data Engineering (ICDE 2013)* (Washington, DC, USA, 2013), ICDE '13, IEEE Computer Society, pp. 997–1008.

[18] KWAK, H., LEE, C., PARK, H., AND MOON, S. What is twitter, a social network or a news media? In *Proceedings of the 19th International Conference on World Wide Web* (New York, NY, USA, 2010), WWW '10, ACM, pp. 591–600.

[19] Kyrola, A., Blelloch, G., and Guestrin, C. Graphchi: Large-scale graph computation on just a pc. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2012), OSDI'12, USENIX Association, pp. 31–46.

[20] Low, Y., Gonzalez, J., Kyrola, A., Bickson, D., Guestrin, C., and Hellerstein, J. M. Graphlab: A new parallel framework for machine learning. In *UAI* (2010), UAI '10, pp. 340–349.

[21] Lu, Y., Cheng, J., Yan, D., and Wu, H. Large-scale distributed graph computing systems: An experimental evaluation. *Proceedings of the VLDB Endowment 8*, 3 (2014).

[22] Macko, P., Marathe, V., Margo, D., and Seltzer, M. LLAMA: Efficient graph analytics using large multiversioned arrays. In *Proceedings of the 2015 IEEE International Conference on Data Engineering (ICDE 2015)* (Washington, DC, USA, April 2015), ICDE '15, IEEE Computer Society.

[23] Malewicz, G., Austern, M. H., Bik, A. J., Dehnert, J. C., Horn, I., Leiser, N., and Czajkowski, G. Pregel: A system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 2010), SIGMOD '10, ACM, pp. 135–146.

[24] Nguyen, D., Lenharth, A., and Pingali, K. A lightweight infrastructure for graph analytics. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2013), SOSP '13, ACM, pp. 456–471.

[25] Perez, Y., Sosič, R., Banerjee, A., Puttagunta, R., Raison, M., Shah, P., and Leskovec, J. Ringo: Interactive graph analytics on big-memory machines. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 2015), SIGMOD '15, ACM, pp. 1105–1110.

[26] Ren, C., Lo, E., Kao, B., Zhu, X., and Cheng, R. On querying historical evolving graph sequences. *Proceedings of the VLDB Endowment 4*, 11 (2011), 726–737.

[27] Roy, A., Mihailovic, I., and Zwaenepoel, W. X-stream: Edge-centric graph processing using streaming partitions. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2013), SOSP '13, ACM, pp. 472–488.

[28] Shen, Y., Chen, G., Jagadish, H., Lu, W., Ooi, B. C., and Tudor, B. M. Fast failure recovery in distributed graph processing systems. *Proceedings of the VLDB Endowment 8*, 4 (2014).

[29] Shun, J., and Blelloch, G. E. Ligra: A lightweight graph processing framework for shared memory. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (New York, NY, USA, 2013), PPoPP '13, ACM, pp. 135–146.

[30] Tian, Y., Balmin, A., Corsten, S. A., Tatikonda, S., and McPherson, J. From "think like a vertex" to "think like a graph". *Proceedings of the VLDB Endowment 7*, 3 (2013).

[31] Viswanath, B., Mislove, A., Cha, M., and Gummadi, K. P. On the evolution of user interaction in facebook. In *Proceedings of the 2nd ACM SIGCOMM Workshop on Social Networks (WOSN'09)* (August 2009).

[32] Zhou, C., Gao, J., Sun, B., and Yu, J. X. Mocgraph: Scalable distributed graph processing using message online computing. *Proceedings of the VLDB Endowment 8*, 4 (2014).