LLAMA: A Persistent, Mutable Representation for Graphs

LLAMA: A Persistent, Mutable Representation for Graphs

A DISSERTATION PRESENTED
BY
PETER MACKO
TO
THE SCHOOL OF ENGINEERING AND APPLIED SCIENCES

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY
IN THE SUBJECT OF
COMPUTER SCIENCE

HARVARD UNIVERSITY
CAMBRIDGE, MASSACHUSETTS
DECEMBER 2014

Dissertation Advisor: Professor Margo I. Seltzer Peter Macko

LLAMA: A Persistent, Mutable Representation for Graphs

Abstract

Graph-structured data is large, ever-changing, and ubiquitous. These features demand that graph analytic applications both compute on and modify large graphs efficiently, and it is also beneficial for analysts to be able to focus just on the last few hours or days of data. We present LLAMA, a graph storage and analysis system that supports mutability and out-of-memory execution, and SLOTH, a sliding window extension of LLAMA that efficiently maintains a view of just the most recently added parts of a graph.

LLAMA performs comparably to immutable main-memory analysis systems for graphs that fit in memory and significantly outperforms existing out-of-memory analysis systems for graphs that exceed main memory. It bases its implementation on the compressed sparse row (CSR) representation, which is a read-only representation commonly used for graph analytics. We augment this representation to support mutability and persistence using a novel implementation of multi-versioned array snapshots, making it ideal for applications that receive a steady stream of new data, but need to perform whole-graph analysis on consistent views of the data. We leverage the multi-versioned nature of this representation to build SLOTH, a sliding window data structure that advances the window by creating a snapshot from new data and aging out old data by deleting the corresponding snapshots.

We compare LLAMA to state-of-the-art systems on representative graph analysis workloads, showing that LLAMA scales well both out-of-memory and across parallel cores. Our evaluation shows that LLAMA's mutability introduces modest overheads of 3–18% relative to immutable CSR

for in memory execution and that it outperforms state-of-the-art out-of-memory systems in most cases, with a best case improvement of a factor of 5 on breadth-first-search. We evaluate SLOTH with various sliding window configurations and demonstrate that LLAMA is a good building block for sliding window computation.

Contents

Listing of figures

Acknowledgments

This work would not have been possible without the support of many people who have helped me with my research, advised me, prayed for me, and encouraged me. I have been blessed to work with many great people and to have many meaningful relationships. I cannot possibly list everybody, so I apologize in advance to anyone whom I may have missed.

I would like to start by thanking my research advisor Margo Seltzer who mentored me throughout my PhD program at Harvard, giving me great advice on my research as well as many other issues, ranging from career advice to eating gluten-free. I would also like to thank the rest of my thesis committee, Jim Waldo, Stephen Chong, and Stratos Idreos, for their comments and valuable advice, both on my thesis and other topics.

I would also like to thank my collaborators on LLAMA and SLOTH. I start by thanking Daniel Margo with whom I collaborated the most during my time in the graduate school. He helped me with evaluating LLAMA, but we also worked together on several other graph-related projects, such as local clustering on provenance graphs and performance introspection for graph databases. Moreover, I appreciate him as a friend. I am also very grateful to Virendra Marathe from Oracle Labs, with whom I collaborated on LLAMA and SLOTH from the beginning of this project. He and Margo came up with the concept, and he gave me valuable advice throughout my research.

I would like to thank James Cuff for his help with benchmarking LLAMA on the big-memory machines, Adam Welc for his insightful discussions about LLAMA especially during the first few months of the project, and Tim Harris for his helpful comments. I am also grateful to Albert Wu,

David Ding, and Yaniv Yacoby for their help with investigating the use of memory mapping for prefetching in LLAMA. I would also like to thank Oracle's Scalable Synchronization Research Group, the Green-Marl team, and the Spatial team for their feedback and support.

I am grateful to my research group members: Amos Waterland, David Holland, Diana MacLean, Elaine Angelino, Kiran-Kumar Muniswamy-Reddy, Marc Chiarini, Nicholas Murphy, Robin Smogor, and Uri Braun. I am grateful for their collaboration and for our discussions, related to research and otherwise. I especially thank Kiran, with whom I worked closely during my first two years in graduate school; he was a great mentor and friend, who made a big difference in my growth as a researcher. I would also like to especially thank David and Uri, for their help with my research, their valuable advice, and their friendship.

I would also like to thank all of my collaborators on other projects with whom I worked with during my graduate program: Keith Smith of NetApp and Alexei Colin on adding back-references to file systems, Wenguang Wang with whom I worked closely at Apple, and Chelsea Yeh, Hanspeter Pfister, Krzysztof Gajos, Madelaine Boyd, and Michelle Borkin with whom I collaborated on provenance visualization. I would also like to thank my managers during my internships, who ensured that I had truly great experiences: Audrey Van Belleghem at NetApp, Deric Horn at Apple, and Karl Haberl at Oracle.

I would like to thank Oracle Labs and the National Science Foundation (NSF) for funding my research. I am especially indebted to Oracle, where I started my dissertation project, and I am grateful for their generous support.

This work would not have been possible without all of my family members, who are too many to list here. I am especially indebted to my wife Jill who supported me, encouraged me, and prayed for me tirelessly, and to my parents who supported me and sacrificed for me to receive a great education. I would also like to specifically thank both of my grandmothers for their love, encouragement, and unceasing prayers.

I am blessed to have many good friends during my graduate studies, out of which I would like to mention at least Gregory Malecha, Kenneth Arnold, Marek Hlaváč, Michael Kester, and Shaolong Wang. I would especially like to thank Marek, whom I knew since elementary school, for playing a major role in my early development as a computer scientist before I entered college.

I am also very blessed to have had many spiritual leaders, who encouraged me, cared for me, prayed for me, and advised me in all sorts of matters: Pastor Paul Kim & Rebekah JDSN, Pastor David Um & Angela SMN, Pastor Thomas Chen & Peggy SMN, Pastor James Lee & Donna SMN. Also, I thank Pastor Sang, Pastor Heechin, and Pastor Roy. Also, I thank Rev. Milan Devečka and Rev. Irena Devečková of my church in my hometown Liptovský Mikuláš, Slovakia.

I would also like to thank all of the Antioch church members, and especially the Young Adult department, for their love, care, prayers, and support. I cannot possibly list everyone, but I would at least like to mention some specific brothers with whom I have developed meaningful relationships: Arthur Huang, Brandon Yoshimoto, Edward Kao, Hendrata Dharmawan, John Kim, Justin Lai, Ken Zhou, Laurence Tai, Sam Hui, Sung Taek On, Tony Qian, and Vallent Lee.

Finally, I would like to thank God, my Lord Jesus, without whom truly nothing would be possible and who made all the difference in my life!

Every two days now we create as much information as we did from the dawn of civilization up until 2003.

Eric Schmidt (Google)

1 Introduction

From biological networks to social networks, from the interstate highway system to the Internet, graph-structured data is ubiquitous, and it continues to grow in both size and complexity. Every minute Twitter users send over 270, 000 tweets, Yelp users post 26, 000 reviews, and Facebook users give 3, 200, 000 likes[18,22]. Due to the massive size of the data and the number of connections, simple questions such as "What is the most influential web page?"[57] and "What is the most controversial hashtag on Twitter?"[62] are next to impossible to answer without taking advantage of the graphi-

cal structure of these data sets. It is no wonder that graph analytics have become almost synonymous with "big data."

The sizes and growth rates of graph-structured data present a challenge in how to efficiently analyze large graphs, process updates, and if desired, to also just focus on the most recently added edges. This dissertation addresses all of these: We present LLAMA, which supports updates and processes graphs larger than RAM on a single machine, and then we present SLOTH, an extension of LLAMA that provides a sliding window abstraction over the stream of new edges, making it easy to focus on only the most recent edges.

The following three approaches have emerged for analyzing large graphs:

- *Distributed platforms* that shard (distribute) a graph over a potentially large number of computers, fitting the graph entirely in the cumulative memory of the cluster,
- *Single-machine shared-memory systems* that store the entire graph in the memory of a single, large multi-processor system, and:
- *Single-machine out-of-memory systems* that perform analysis using a single machine, storing the graph on a disk or other persistent memory.

The problem with the first approach is that it requires managing a potentially large computer cluster, while the second approach requires expensive machines. Single-machine out-of-memory systems lower cost and are often simpler to manage, however they are frequently much slower than dedicated in-memory systems even when the graph fits entirely in memory, and they are often not easier to program than distributed systems.

Another common problem is that many state of the art analytics systems do not store persistent representations of graphs, so they ingest data each time they run. Users who run multiple computations on the same data thus pay the ingest cost repeatedly. Furthermore, graph-structured data loaded in many analytic engines is immutable. Systems that do support mutation typically do so as a

part of a graph algorithm (for example, some algorithms add edges, and others delete edges to avoid visiting them twice).

We present a single representation that provides high-performance analytics both in and out of memory on a single machine. Our solution performs as well as shared-memory systems when the graph fits entirely in memory, performs better than existing out-of-memory solutions when the graph is larger than memory, and enables efficient incremental ingest and updates. It further lessens the need for hardware over-provisioning for handling unexpected spikes in graph growth in shared-memory applications, since our performance degrades gracefully as the graph outgrows memory. We base our approach on data structures optimized for main-memory, read-only analytics, adding support for updates, and then making it work out-of-core.

Our approach can be also used as a building block to develop faster and more power-efficient distributed systems. The two main factors contributing to the performance of a distributed system are single-node performance and the cost of network communication. Our approach addresses both: Improving single-node performance directly results in improving the overall performance of a distributed system, and enabling a single node to process a larger fraction of the graph results in requiring fewer nodes in a distributed system, thus lowering the amount of network communication and the overall power consumption.

We also address another class of graph analytics with rising importance: sliding-widow, time-series computation. For example, questions like the one mentioned above, "Who is the most influential Twitter user within the last hour?" fall into this class. Answering such queries allows analysts to spot interesting trends in a stream of incoming graph edges, such as new tweets or likes.

As our first objective is fast computation on data that fits in memory, we choose the compressed sparse row representation[19] (CSR) as our starting point, due to its acceptance as the gold standard for in-memory graph analytics. CSR is the most common data structure for in-memory graph analytics, primarily because of its compact form and great cache behavior[9]. We also found it to perform

best in our experimentation (Section 2.5). A few examples of systems built on top of the CSR and its variants are the Connectivity Server[7], Grace[60], GraphChi[43], Green-Marl[32], and SNAP[66]. The disadvantage of CSR is that at its core it is immutable and entirely memory-resident. CSR is thus not a natural fit for quickly growing datasets and datasets larger than memory. There are several existing approaches that address these issues, but they either come with noticeable performance and/or space penalties, or they sacrifice useful features, such as the ability to store multiple versions of a graph. However, we show that this need not be the case: with our approach we achieve rich features, fast incremental ingest, and fast graph analytics, even for graphs larger than memory.

> The compressed sparse row representation is an effective building block for graph analytics applications requiring mutability, persistence, and/or sliding window computation.

1.1 Contributions

The contributions of this dissertation are as follows:

- *Quantitative comparison of most current approaches to CSR mutability in a single framework*: Many of the existing approaches to making CSR mutable fall into two categories: delta maps[43] and log-based CSR[7,60]. By comparing them in our framework, we show that running analytics on log-based CSR performs with only 3–6% overhead on top of traditional, immutable CSR for many common algorithms, but at more than five times the memory cost for incrementally ingested graphs with more than ten snapshots. Delta maps (depending on their implementation) can be easily even 100% slower than traditional CSR and occupy two to three times more memory (or even more).
- *Fast, memory efficient approach to CSR mutability*: We propose and evaluate a novel way to make CSR mutable that performs nearly as well as log-based CSR but with a significantly smaller space overhead.

- *LLAMA – efficient graph analytics on graphs larger than memory*: We leverage the aforementioned space-efficient, mutable CSR representation to prototype a graph analytics solution for graphs larger than memory. We achieve analytics performance faster than the state of the art by using `mmap` to implement a larger-than-memory CSR backed by an SSD and ensuring that processing vertices in ID order results in an access pattern that is as close to sequential access as possible, even in the presence of updates, with a best-case improvement of a factor of 5 on breadth-first-search.

- *SLOTH – sliding window graph analytics on a time series of graph edges*: We further leverage our mutable CSR design to efficiently maintain a sliding window over a stream of incoming graph updates.

1.2 Dissertation Overview

In the next chapter, we survey the four most common data structures for representing graphs in memory. We present this in three parts: Section 2.1 introduces the compressed sparse row (CSR) representation and surveys its common variants and existing approaches to implementing mutability. Sections 2.2 to 2.4 describe storing adjacency lists in separate arrays and in compressed bitmap indices, and storing the adjacency matrix explicitly. The final section then compares these representations on simple graph analytic workloads and shows that CSR outperforms the alternatives by 34–46%. We defer comparing the various approaches to CSR mutability to the later chapters.

Chapter 3 introduces LLAMA, a graph storage and analysis system that supports mutability and out-of-memory execution. It then presents the programmer's view of LLAMA. The next two chapters describe LLAMA in detail, discussing and comparing the various design options, followed by Chapter 6 that presents a comprehensive evaluation.

Chapter 4 explores various options for making CSR mutable when the entire graph fits in memory. The chapter starts by providing a high-level overview of LLAMA's mutable graph data structure based on CSR. The chapter then presents two approaches to implementing a mutable vertex table: a simplistic flat-array approach and an approach based on a multiversioned copy-on-write array. The next section revisits two common state of the art approaches to CSR mutability: adding a writable delta map on top of the read-only CSR and treating the CSR as a log; and describes how to represent them in LLAMA. Section 4.4 introduces the novel approach that we use in LLAMA, which has performance comparable to the state of the art for a majority of workloads while representing the graph more compactly. Section 4.5 unifies our space-optimized approach with the approach that treats the CSR as a log by describing them as two extremes of a single parameterizable approach; other possible approaches fall along the spectrum. Section 4.6 discusses mutable properties, Section 4.7 describes our implementation of the write-optimized graph store (i.e., a delta map), and Section 4.8 presents further details about LLAMA operation. The final section presents a quantitative comparison of the various approaches to CSR mutability on in-memory workloads, showing that our space-optimized approach results in high performance and low memory cost.

Chapter 5 describes and discusses the persistent graph storage in LLAMA. The first two sections detail the persistence aspects of the vertex and edge tables. We use a vertical partitioning design (i.e., storing each attribute in a separate file) and memory map all files to lower buffer management overhead. The last section justifies our design decisions in the context of alternative approaches.

Chapter 6 presents a comprehensive evaluation of LLAMA both in and out of memory. After describing our experimental setup, the chapter dives into cross-system comparison, in which we compare LLAMA's performance to two state of the art in-memory systems, GreenMarl[32] and GraphLab[45], as well as to two state of the art out-of-memory systems, GraphChi[43] and X-Stream[63]. We show that LLAMA's persistence and mutability add a modest (3–18%) overhead on top of immutable CSR on common tasks such as breadth-first search (BFS), PageRank[57], and triangle count-

ing, and that it outperforms the state of the art out-of-memory systems with a best-case improvement by a factor of five on BFS. We further demonstrate that LLAMA scales well with the number of snapshots, CPU cores, and the graph size.

Chapter 7 presents SLOTH, an extension of LLAMA for maintaining a sliding window over a time-series of edges. Section 7.1 presents a high-level overview of the system, followed by Section 7.2 that explains how to construct and advance a multiversioned sliding window using LLAMA. Section 7.3 presents an evaluation, and Section 7.4 summarizes the results.

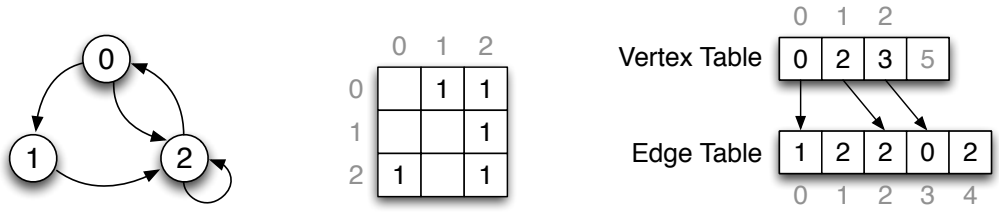
Chapter 8 surveys related work and compares LLAMA and SLOTH to existing and proposed systems. The chapter starts by surveying the field of graph analytic systems, both single-machine and distributed. Section 8.2 then briefly describes graph databases. Finally, Section 8.3 summarizes the related work in the area of sliding windows in graphs.

Chapter 9 discusses conclusions and future opportunities. We review what we have accomplished and compare it to our design requirements. Finally, we highlight the implications of this work and investigate avenues for future work.

2

Survey of Graph Data Structures

We begin with an overview of the widely-used compressed sparse row representation (CSR) on which we base our implementation. We then discuss other graph representations, such as those used in systems designed for a more OLTP-like setting, and conclude the chapter with a brief quantitative comparison of the various representations on a simple analytics workload.



(a) Graph

(b) The equivalent sparse matrix

(c) The CSR representation

Figure 2.1: Compressed Sparse Row Representation. Every graph (a) can be represented as a sparse matrix (b), which can be encoded using the CSR representation (c).

2.1 Compressed Sparse Row Representation

We can represent a graph structure with n nodes as an $n \times n$ sparse matrix M, in which element M_{ij} is nonzero if and only if there is an edge from node i to node j (i is the *source* or *tail* vertex, and j is the *target* or *head* vertex). The undirected edge between node i and node j is represented with nonzero values in both M_{ij} and M_{ji}. The compressed sparse row (CSR) representation is a more efficient encoding of the rows of this sparse matrix, which facilitates rapid identification of the *outgoing* edges of each node. A compressed sparse column (CSC) representation is an identical implementation that encodes the *incoming* edges of each node.

A typical implementation consists of two tables, each implemented as an array:

- *Vertex Table* (V) is an array of $n + 1$ elements that maps a vertex ID (between 0 and $n - 1$) to the ID of its first outgoing edge, which doubles as an index into the edge table, where the corresponding adjacency list starts. The number of items of the i-th adjacency list is $V[i + 1] - V[i]$, and the last element in the vertex table is set to the size of the edge table.
- *Edge Table* (E) stores each node's adjacency list consecutively in a single array. Each entry in the array maps an edge ID to its corresponding destination vertex ID.

Figure 2.1 shows an example of a graph (a) represented as a sparse matrix (b) and encoded as CSR (c). Consider node 2: The vertex table indicates that its adjacency list starts at index 3 in the edge table and that its size is $V[3] - V[2] = 5 - 3 = 2$. The edge table shows that the endpoints of the outgoing edges from node 2 are nodes 0 and 2. The IDs of these edges are 3 and 4, which are their indices in the edge table.

While there are many approaches to storing properties, node properties are most commonly stored in a flat array parallel to the vertex table, so that the property of node i is at index i. Edge properties are usually stored similarly in an array parallel to the edge table.

Many graph analytics systems, such as the Connectivity Server[7], Grace[60], GraphChi[43], Green-Marl[32], and SNAP[66], use CSR or one of its variants. We discuss the common variants of CSR below.

2.1.1 Common CSR Variants

Green-Marl[32] and SNAP[66] both follow the typical CSR design, but other variants are possible: For example, the Connectivity Server[7] and Grace[60] do not guarantee that the adjacency lists belonging to consecutively numbered vertices are stored next to each other. In this case, the number of items in the i-th adjacency list cannot be determined from the $(i + 1)$-th element in the vertex table as is usually done in CSR. The Connectivity Server[7] instead determines the end of an adjacency list by setting the highest-order bit in the last item. Grace[60] instead stores the length of the adjacency list explicitly in the edge table before the adjacency list itself.

We found in our experimentation that different methods of determining the end of an adjacency list can have significant performance implications. We evaluated four methods on two common graph algorithms: triangle counting and 10 iterations of PageRank[57] (a pull-based implementation in which a vertex update function reads weights of its neighbors and updates its own weight). The four evaluated methods were:

Variant	Triangle Counting		PageRank (10 iterations)	
Standard CSR	1206.9 ms		5821.4 ms	
Degree in vertex table	1224.7 ms	(1.5%)	5712.8 ms	(−1.9%)
Degree in edge table	1329.0 ms	(10.1%)	8462.7 ms	(45.4%)
Set the high-order bit	1360.9 ms	(12.8%)	20522.9 ms	(252.5%)

Table 2.1: Performance implications of alternatives for identifying the end of adjacency lists evaluated on two common graph algorithm on a 10 million node, 50 million edge random graph, running on our Intel Core i5 platform. All times are in ms, and the percentages show the difference relative to the standard CSR method.

1. The standard CSR method,
2. Storing the node degree explicitly in the vertex table,
3. Storing the node degree explicitly in the edge table (used in Grace[60]), and:
4. Setting the high-order bit of the last item in the adjacency list (used by the Connectivity Server[7])

We used a 10 million node, 50 million edge Erdős–Rényi random graph[21] for this evaluation. Table 2.1 shows the results on a dual-core Intel Core i5 + HT, 3.30 GHz, equipped with 16 GB RAM.

The standard CSR method and storing the node degree in the vertex table both perform well relative to the other two approaches. The overheads for triangle counting are statistically significant but not large. Storing the degree explicitly increases the data structure sizes and thus worsens cache behavior, and checking for the bit requires additional bit manipulation instructions. We confirmed that the additional instructions can indeed cause such a dramatic slowdown by comparing the disassembled triangle counting routines for the various tested approaches: The inner loop in our implementation is only 8–10 assembly instructions long, so adding even two bit-manipulation instructions is sufficient to cause a significant slowdown.

The overheads were much more pronounced when running PageRank. Our implementation computes the vertex degree in its inner loop but does not otherwise access the vertex's adjacency

list. Storing the degree in the edge table thus requires an edge table access, which is not necessary when the degree can be determined just from the vertex table. Using a bit flag requires the vertex degree to be computed explicitly by traversing the adjacency list. Storing the degree in the vertex table resulted in faster execution than when computing it by subtracting two consecutive indices in the vertex table. The result is statistically significant, and we confirmed by analyzing our results that using one instruction to compute the degree versus three instructions does indeed account for the performance improvement.

2.1.2 Mutable CSR

There are currently three ways to build a mutable graph store on top of CSR: adding delta maps, treating the CSR as a log, and using dynamic block linked lists. We focus primarily on the first two approaches as they are the most common. For now we introduce these techniques but leave evaluation for Chapter 4, after we have introduced our analytics system.

Mutability using Delta Maps

The most straightforward way to add mutability to CSR is by augmenting it with a delta map [43], a writeable data structure that stores the difference between the CSR and the current state. One way to implement the delta map is as a hash table that maps node IDs to growable arrays of recently added or removed edges, but many other implementations are possible.

When we need to compute analytics on the most recent data, the system must run the computation on the combined delta map and CSR; when out of date data suffice, the system can run the computation exclusively on the CSR. Computation on the combination of delta map and CSR is, unsurprisingly, slower. While computation on the CSR is significantly faster, achieving that speedup on the most up-to-date data necessitates rebuilding the entire data structure.

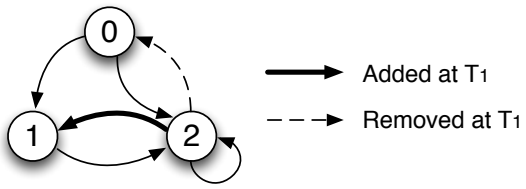


Figure 2.2: A Modified Graph for future examples. It contains one added and one removed edge at time T_1 to the original graph from Figure 2.1(a).

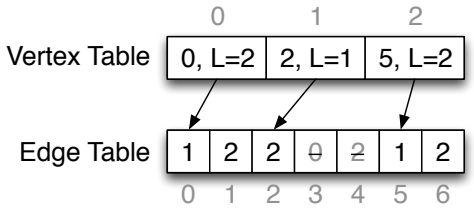


Figure 2.3: Log-Based CSR: Adding or removing an edge is implemented by appending an updated adjacency list at the end of the edge table and updating the vertex table accordingly. Adjacency list lengths are stored in the vertex table, but other approaches to determining the ends of adjacency lists are possible.

Log-Based CSR

The Connectivity Server[7] and Grace[60] modify the CSR directly by treating the CSR's edge table as a log. They preallocate the edge table with free space at the end. Whenever the user modifies an adjacency list, such as by adding or removing an edge, the system appends the entire updated adjacency list to the end of the edge table and updates the pointer in the vertex table accordingly. The system reallocates the edge table if there is no more space. Updating the graph creates "holes" in the edge table, which waste space. The system must periodically compact the structure to keep memory overhead under control in the presence of frequent updates.

Systems built on top of log-based CSR often do not have a separate delta map to batch updates, especially if they were designed for data ingest in large batches. In some systems, frequent updates to the same adjacency list result in copying a large amount of data and creating many holes in the edge

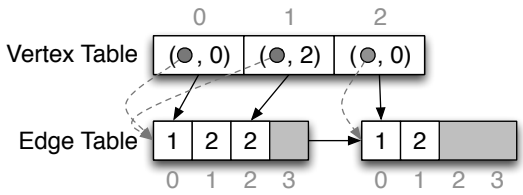


Figure 2.4: Dynamic Block Linked Lists. The vertex table contains pointers to the adjacency lists in the edge table stored on top of a block linked list.

table. This can be easily solved by adding a delta map. An alternative solution is to reserve some empty space at the end of each adjacency list so that the CSR can absorb a small number of updates per vertex without having to rewrite the adjacency list at the end of the edge table each time. The disadvantage of this approach is that it leaves a large number of small holes between the adjacency lists, which increases the total data structure size and adversely affects cache behavior.

Figure 2.2 takes the graph from Figure 2.1 and shows the addition of an edge from node 2 to node 1 and the removal of the edge from node 2 to node 0. Figure 2.3 shows how these modifications would be implemented in a log-based CSR. It is important to note that in this representation we cannot calculate the length of the adjacency list by the difference between two vertices, because sequentially numbered vertices are no longer necessarily consecutive in the edge table. We described and evaluated some alternatives in Section 2.1.1.

Grace[60] uses a software-based copy-on-write technique to create and maintain multiple versions of the vertex table, enabling efficient storage of multiple versions of the data. We use a similar approach in LLAMA.

Dynamic Block Linked Lists

GraphLab[24,45] implements their dynamic CSR by storing the edge table in a dynamic block linked list – a linked list of arrays, called "dynamic blocks," which makes it easy to dynamically insert new elements in the middle of the data structure. Figure 2.4 illustrates the graph in Figure 2.2 stored in

CSR built on top of a dynamic block linked list. The arrays in the block linked lists contain "next" pointers and a size in addition to the data elements. The default array size in GraphLab is 4 KB.

An element in the vertex table consists of a pointer to the corresponding array and an offset within the block where the adjacency list starts. The adjacency lists are stored in order of vertex ID, so the end of the adjacency list can be easily inferred from the start of the adjacency list of the next vertex in the vertex table.

While this data structure supports efficient reads when in memory, we do not expect it to work well out of core: Blocks might not be entirely filled, which would result in more I/O to read the same number of "useful" bytes, and splitting blocks would break the sequential access pattern typical for CSR. Additionally, adding and removing edges requires shifting data and splitting or merging potentially many dynamic blocks with the vertex table updated accordingly.

2.2 Adjacency Lists

Now we will turn our discussion from CSR to other data structures for graphs.

Systems that prioritize updates and/or optimize for online transaction processing (OLTP) workloads usually store adjacency lists in separate (growable) arrays. A few notable examples are the object cache in Neo4j [52], a popular graph database, STINGER [4], a data structure optimized for streaming applications, and Trinity [64], whose goal is efficiency for both analytics and OLTP-like workloads simultaneously.

Trinity [64] stores adjacency lists in separate objects called "cells" that the system stores compactly next to each other in a circular buffer. It grows cells by reallocating them at the head of the buffer, and it compacts the buffer by periodically moving objects from the tail to the head.

Neo4j's object cache [52] and STINGER [4] have a more traditional implementation of separate adjacency lists. An adjacency list is implemented as a linked list of arrays (or even an array of arrays),

so that more edges can be efficiently added on the end without having to reallocate the entire list. This also improves concurrency, because it makes it unnecessary to acquire a lock when reading the list to protect the reader from the system moving the list in memory at the same time.

Properties can be stored directly in vertex and edge objects or completely separately from the graph structure.

2.3 Compressed Bitmap Indices

DEX[48] represents graphs using compressed, partitioned bitmaps, which enable the database to efficiently combine multiple adjacency lists using set operations. The bitmaps are chunked into 32-bit words, and all nonzero chunks are stored in a balanced tree that maps the offset of a chunk to the chunk itself. The result is a compact data structure optimized for out-of-core computation with reasonable in-memory performance, but it is not as fast as dedicated in-memory systems.

DEX uses the following three groups of mappings, each implemented as two B+ trees[14], one for forward lookup (get data associated with a given ID) and one for reverse lookup (find IDs corresponding to the given data):

- Objects (vertices and edges): Mapping object IDs to their vertex and edge types (labels)
- Relationships: Mapping edge IDs to the IDs of tail and head nodes
- Properties: Mapping IDs to their properties

All strings are dictionary encoded.

2.4 Adjacency Matrix

Dense graphs, in which the number of edges $|E|$ approaches the square of the number of vertices $|V|^2$, are often stored using an uncompressed adjacency matrix[15]. However, in our work we rarely see such graphs; social networks and road networks are rarely that dense.

Data Structure	Undirected BFS (ms)
CSR	501.6
Adjacency Lists	703.6
Compressed Bitmap Indices	885.0

Table 2.2: Comparison of Various Graph Data Structures on graph traversal benchmarks on a scale-free 9 million node, 45 million edge Barabási-Albert graph[5].

2.5 Quantitative Comparison

Before committing to CSR (Section 2.1), we conducted a preliminary study, comparing it to adjacency lists (Section 2.2) and compressed bitmaps indices (Section 2.3). We implemented all three data structures in the same custom C++ framework (not LLAMA) to evaluate them on a level playing field, unaffected by other differences in the state of the art systems that implement them, such as buffer management, synchronization, and API. We used similar array-based vertex table implementations for all three representations to further minimize confounding variables. Our results thus do not reflect runtimes from any particular graph analytics system, but they are indicative of their relative performance.

We evaluated the representations on a synthetically generated, scale-free 9 million node, 45 million edge Barabási-Albert graph[5] on a dual-core Intel Core i3, 3.00 GHz, and 8 GB RAM (herein after referred to as a "small commodity" machine). We compared the performance of graph traversal using breadth-first search configured to traverse edges in both direction and to cap the maximum depth to $k = 5$ hops. We ran each operation $10,000$ times on randomly selected vertices (the same on all three representations) and report the average runtimes. Other variants, such as traversing the edges in only one direction or capping at a different number of hops, follow the same pattern.

Table 2.2 shows the results, highlighting CSR as the most efficient representation, especially for read-only, traversal-oriented workloads.

CSR is faster than storing adjacency lists in separate arrays due to its excellent cache behavior[9] and compact size. This is fundamental to these data structures, because CSR is essentially a densely-packed collection of adjacency lists sorted by the source vertex ID, which is important for the performance of a large class of algorithms that iterate over all vertices, such as PageRank[57].

CSR is also faster than compressed bitmaps; it is compact and it requires relatively little computation to reconstruct adjacency lists. Compressed bitmaps can represent large adjacency lists compactly, but do not order or densely pack individual adjacency lists. Our result that CSR performs better than the compressed bitmaps is not a fundamental relationship between the two data structures, since compressed bitmaps might theoretically outperform CSR if they represent the graph significantly more compactly than CSR.

Moreover, CSR's read-only nature is both a blessing and a curse: It does not require any synchronization for read-only workloads, which is great for performance, but it can efficiently support only static graphs. Another disadvantage of CSR is that it does not enable the graph analytics system to selectively page individual adjacency lists in and out of memory, since they are not represented as separate objects. Nonetheless, we show in the remainder of the dissertation that CSR is, in fact, a fine building block for implementing a high-performance out-of-memory graph analytics system that supports incremental load and updates.

3

LLAMA Overview

We introduce LLAMA, a scalable and high-performance graph storage and analysis system that works well both in and out of memory. LLAMA supports efficient incremental ingest, mutation, multiversion data access, and out-of-memory execution. This chapter presents a high-level overview of LLAMA; the next two chapters describe the system in detail and discuss and evaluate the various design options. Chapter 6 then presents a comprehensive evaluation of LLAMA, compares it to several state-of-the-art systems (GreenMarl[32], GraphLab[45], GraphChi[43], and X-Stream[63]), and

evaluates its scalability with respect to the number of CPU cores, graph size, and the number of snapshots.

LLAMA consists of the following two core components:

- *Multiversioned read-optimized graph store* (ROS), based on a compressed sparse row representation and tuned for high-performance graph analytics, and:
- *Write-optimized graph store* (WOS, also known as a *delta map*), which buffers incoming updates in memory and might or might not participate in execution of a graph analytics algorithm depending on the workload requirements.

This is similar to Log-Structured Merge (LSM) trees[35,54] and some relational databases, such as column stores[1], which also consist of multiple read-optimized stores and a single write-optimized store.

The write-optimized graph store maintains the most up-to-date version of the graph and processes all update requests, but it is slower to query and less memory efficient than the read-optimized graph store. We implement the write-optimized store (Section 4.7) by storing adjacency lists in separate arrays, which is slower than CSR for read-only queries as we saw in Section 2.5. Querying the read-optimized store does not require locks, which makes it particularly well-suited for long-running queries. LLAMA periodically freezes the contents of the write-optimized store, creating a new snapshot in the multiversioned read-optimized store (Section 4.1). Read-only workloads that do not need the most up-to-date data can be run on the read-optimized store without consulting the write-optimized store, resulting in high performance. LLAMA periodically merges snapshots to decrease memory footprint and improve performance.

The read-optimized store is the primary focus of this dissertation. It stores the graph structure in a persistent, multiversioned variant of CSR, and it stores vertex and edge properties in persistent, multiversioned arrays parallel to the vertex and edge tables of the CSR.

A read-optimized store is traditionally implemented as multiple trees[35,54] (or sorted lists), each corresponding to a different snapshot in the database, and a query typically examines every tree in the database. A common optimization is to associate a Bloom filter[8] with each snapshot containing all keys found in its tree or a sorted list, so that a query can quickly narrow down the set of trees that need to be examined – but it still needs to examine each Bloom filter. LLAMA's read-optimized store replaces trees by CSR-like data structures, improving the key lookup time complexity in each examined snapshot from $O(\log n)$ to $O(1)$. LLAMA's representation differs from a traditional CSR by the addition of an indirection array that lets us use copy-on-write to copy the vertex table for snapshot n to the vertex table for snapshot $n + 1$ (Section 4.2). This copy-on-write approach also subsumes the functionality of the Bloom filters: Every vertex table entry contains a direct reference to the edge list of the most recent snapshot containing any edges for the vertex, and the system can efficiently determine the next snapshot to examine either by using a "next" pointer (Section 4.4.2) or by examining an older version of the vertex table (Section 4.4.1).

The read-optimized store does not update data directly, but it has the capacity to create new snapshots, which can be used to implement coarse-grained mutability, i.e., applying a batch of updates all at once. This behavior is often sufficient for algorithms that require changes to the graph to be applied only at the end of a phase in a computation, such as between the gather and scatter phases of the gather-apply-scatter (GAS) model[24]. The users can additionally take advantage of the multi-versioned nature to run queries on different versions of the graph and to study graph evolution over time.

The next section introduces the programmer's perspective of LLAMA. Chapter 4 then explains the multiversioned nature of the read-optimized store in detail and explores various design options, followed by a discussion of persistence in Chapter 5. We describe the implementation of our write-optimized store in Section 4.7.

3.1 The Programmer's View of LLAMA

LLAMA is a C++ library that users can easily embed in their programs. It exports a C++ API for loading, accessing, and modifying graphs, suitable for writing C-like imperative programs without having to conform to any particular execution framework, such as gather-apply-scatter (GAS)[24].

After opening a database, the user can iterate over all vertices in the database using a for loop, potentially using OpenMP[55] for parallelism, or can address vertices directly using their IDs. LLAMA thus supports both whole-graph queries and ad hoc queries such as a breadth-first search (BFS) starting from a specific vertex. Once given a vertex, the user can use an iterator to loop over the vertex's adjacency list, using either the most recent data or data from a specific snapshot. LLAMA does not expose the CSR adjacency lists directly to the user, except in the special case in which the graph is stored in a single snapshot, in which case all adjacency lists are contiguous in memory. LLAMA additionally provides a BFS construct and syntactic sugar to simplify common operations such as iterating over all vertices or iterating over the out-edges of a given vertex.

LLAMA provides a general-purpose programming model instead of the more limited vertex- or edge- centric models, such as GAS[24]. It is nonetheless possible to write LLAMA programs that exhibit such access patterns, and it is indeed advantageous to do so when processing graphs significantly larger than memory, as this produces sequential access patterns. Unlike GraphChi[43] or X-Stream[63], LLAMA enables rather than enforces sequential access patterns. We found that writing programs that exploit sequential access was straightforward: process vertices in the order in which they are stored in the vertex table (i.e., in vertex ID order).

We implemented two alternative implementations for some algorithms, such as BFS; one is optimized for in-memory graphs, and the other is optimized for out-of-core graphs. Currently the user has to choose between the two manually, but it would be easy to build this into the library.

4

Mutable CSR

Mutable CSR is the core of LLAMA's read-optimized graph store. This chapter describes and compares several techniques for making CSR mutable focusing entirely on its in-memory performance as this is one of our main design goals, and the next chapter then discusses persistence aspects. The objective of this chapter is to explore the design space and to recommend a specific approach to CSR mutability for use in LLAMA.

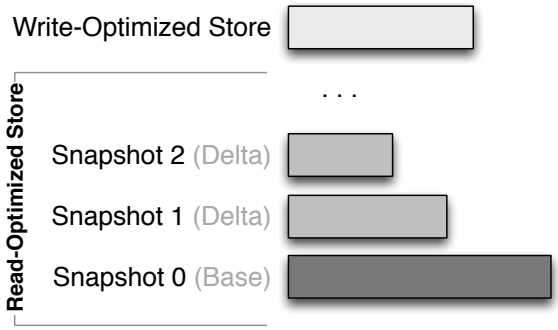


Figure 4.1: High-Level LLAMA Representation: LLAMA consists of a multi-snapshot, read-optimized store and a single-snapshot, write-optimized store. Snapshot 0 is the base snapshot, and the other read-optimized snapshots are delta snapshots.

We start with a high-level overview of LLAMA's read-optimized store. Section 4.2 briefly describes an implementation of a simple vertex table that does not support multiversion access, followed by a detailed description of our multiversioned vertex table based on copy-on-write. We refer to these two implementations in the following sections. Section 4.3 then revisits two common methods for making CSR mutable and explains how to represent them in LLAMA. We then describe our technique and its variants in Section 4.4, which combines good performance with good memory efficiency. Section 4.5 then shows that our space-optimized approach and our representation of log-based CSR revisited in Section 4.3.2 represent two extremes in a single design space, and we can describe the "in-between" designs using parameterization. Sections 4.6 to 4.8 then briefly describe mutable properties, our implementation of the write-optimized store (i.e., the delta map), and LLAMA's log-merge operation. Section 4.9 presents a quantitative comparison of the various approaches to CSR mutability, followed by a summary of the key points in Section 4.10.

4.1 Overview of the Multiversioned Read-Optimized Graph Storage

LLAMA represents a graph using a variant of the Compressed Sparse Row (CSR) representation that supports versioned data, which we use to provide efficient incremental ingest and fast multi-versioned access. When a graph is first loaded into LLAMA, it resides in a single base snapshot, and each subsequent incremental load creates a new "delta" snapshot encoding the difference between the previous and the new state of the graph. LLAMA operates in the log-merge fashion[35,54]: Ingest and updates create new delta snapshots, and the system periodically merges snapshots when enough of them are created or when instructed to do so by the user.

Figure 4.1 shows a conceptual example of LLAMA, showing a write-optimized store and a read-optimized store with multiple snapshots. Snapshot 0 is fully self-contained, while all the other snapshots (plus the write-optimized store) store the differences from the previous snapshot.

A delta snapshot can contain added and deleted edges only, complete updated copies of adjacency lists, or a mixture of the two, depending on the policy selected by the user. A vertex's adjacency list can potentially be spread across several snapshots. We call the part of an adjacency list contained in a single snapshot an *adjacency list fragment*.

A graph in LLAMA consists of the following data structures, each of which we describe in more detail in the following sections and chapters:

- multiple *edge tables*, one per snapshot, each storing consecutive adjacency list fragments, and:
- a single *vertex table* shared by all snapshots, mapping vertex IDs to per-vertex structures.

LLAMA implements the vertex table using a Large Multiversioned Array (or LAMA) by default, which we describe in Section 4.2.1, but it can also use a non-versioned vertex table if requested by the user. We discuss the various options for the edge table implementations in Sections 4.3–4.5.

Having multiple edge tables also changes the definition of an edge ID from the standard one used by CSR. In a conventional CSR, an edge ID is just the index into its edge table, but in LLAMA, we

uniquely reference an edge by the tuple $(snapshot\_id, index)$, where $index$ is the index into the edge table of a particular snapshot. Most applications do not need more than $2^{48} \approx 2.8 \times 10^{14}$ edges and $2^{16} \approx 6.5 \times 10^4$ snapshots, so we pack the tuple into a single 64-bit word by using the most significant 16 bits for the $snapshot\_id$ and the other 48 bits for the $index$. If the workload requires more snapshots or more edges per snapshot, the user can configure the number of bits used by each component of the edge ID at compile time.

The vertex table elements consist of the following three fields:

- $first\_edge$ (8 bytes) – the edge ID of the first edge in the adjacency list,
- $fragment\_length$ (4 bytes) – the length of the adjacency list fragment referenced by the $first\_edge$, and:
- $degree$ (4 bytes, optional) – the vertex degree.

We include the vertex degree because it is frequently used in inner loops of many algorithms, such as in some implementations of PageRank[57]. We do not need to include the vertex degree if the edge table always stores complete adjacency lists, in which case $fragment\_length = degree$.

4.1.1 Edge Storage Modes

By default, LLAMA stores only the out-edges of a graph, as this suffices for many analytics applications; the user can optionally include in-edges. LLAMA stores out- and in- edges in separate read-optimized stores.

It is important to note that an out-edge and the corresponding in-edge have different edge IDs, because edge ID in CSR correspond to specific locations in their respective edge tables. LLAMA optionally maintains a map of in-edge IDs to out-edge IDs implemented as a special edge property in the "in-edges" read-optimized store (and optionally also a map of out-edge IDs to in-edge IDs if desired by the user).

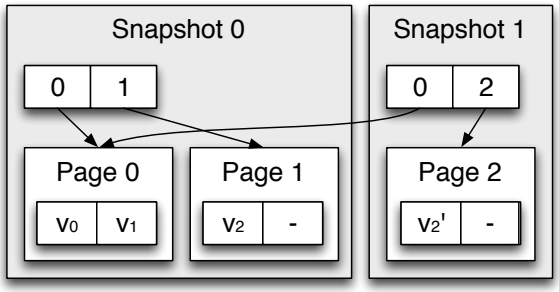


Figure 4.2: LAMA Vertex Table: An example of a two-snapshot vertex table in which the data associated with vertex 2 have been modified in Snapshot 1. The example uses $2^m = 2$ elements per page, but m is significantly larger in practice.

LLAMA additionally supports other views of the graph that can be useful for specific applications. For example, an undirected graph can be loaded in "doubled" mode, which transforms it to a directed graph by representing each edge as both an incoming and an outgoing edge. The user can also specify "post-order edges" mode, which inverts some edges so that all edges lead from a tail ID to a numerically greater (or equal) head ID. This mode is useful for triangle counting.

4.2 Mutable Vertex Table

The simplest way to implement a mutable vertex table is to maintain a flat array and update it in place. While this results in great performance, it provides neither multiversion data access nor the ability to create a snapshot while computing on the latest stable snapshot.

The following section describes the multiversion arrays that we use in LLAMA. While multiversioned arrays provide good performance, users can choose a flat vertex table at compilation time.

4.2.1 Multiversioned Vertex Table – Large Multiversioned Array (LAMA)

LLAMA implements the multiversioned vertex table on top of multiversioned arrays, which we call LAMAs. From the programmer's perspective, a LAMA is a mutable array with support for snap-

shotting. In principle one could treat snapshots as writable clones and create a branching version tree, but we currently do not do so.

The LAMA implementation uses a software copy-on-write technique: We partition the array into equal-sized data pages, each of which holds 2^m elements for some fixed value m. We then construct an indirection array containing pointers to the data pages. We access the i^{th} element of the array in two steps: first, assuming that the LAMA array uses b-bit indices, locate the data page by using the $b - m$ most significant bits of i to index into the indirection array, then, use the remaining m bits to select the appropriate entry from the data page. Each page has an associated reference count.

We create a snapshot by copying the indirection array and incrementing the reference counts for each page. To modify a data page belonging to a previous snapshot, we first copy the page, update the indirection array to reference the new page, and then modify the new page as desired. The top of Figure 4.2 shows a LAMA vertex table with two snapshots, three vertices, and $2^m = 2$ elements per page. In practice, we choose m to be high enough so that the physical page size is an integer multiple of the file-system and virtual memory page sizes and so that the entire indirection table fits in the L3 cache or higher in the memory hierarchy. At the same time, choosing a value that is too large can cause an unnecessarily large copy-on-write in response to even a small number of randomly distributed updates. We found 4 and 8 KB pages to work well in practice.

Large Initial Capacity

Note that we can easily allocate a LAMA with more capacity than is currently needed with a small space penalty. We set entries in the indirection array to refer not-yet-populated pages, all referencing a single zero-filled page, which we update copy-on-write. For example, with 4 KB data pages, reserving space for one million eight-byte elements (that is, $10^6 \times 8/4096 \approx 2000$ pages) requires only 16 KB in memory ($2000 \times 8 = 16000 \approx 16$ KB, assuming 8-byte pointers) plus the zero page.

The Cost of the Indirection Table

To understand the overhead of accessing data through the indirection table in LAMA when all data fit in memory (i.e., when data access is not I/O bound), we compared LAMA to an immutable flat array implementation on two common graph algorithms on the LiveJournal graph [3] with 4.8 million nodes and 69.0 million vertices, which is a skewed real-world social network graph. We found that the cost of the indirection table is 4.0–5.1% for BFS, which has a tight inner loop, and only 0.2–1.1% for push-based PageRank [57], whose inner loop is dominated by the cost of atomic operations (see Section 4.9 for more detail).

In a preliminary study on an Erdös-Rényi graph [21] with 10 million vertices and 50 million edges, we also compared LAMA to a COW implementation in which we modify the operating system's page tables directly to share common pages, but we found that this produced sufficiently many TLB misses to be approximately 5% slower than LAMA.

4.3 Representing CSR + Delta Map and Log-Based CSR in LLAMA

Now, we revisit the two most common methods for making CSR mutable and show how to represent them in LLAMA. This lets us conduct a fair quantitative comparison between the different approaches on a single platform, unaffected by differences between the analytics systems that use them. Note that while we do not have to achieve an exact reimplementation of these approaches for the conclusions to hold, we need to ensure that the performance impact of any implementation difference is negligible.

4.3.1 CSR + Delta Map

LLAMA represents CSR with a delta map (Section 2.1.2) by restricting the read-optimized store to use only a single snapshot, choosing the flat array vertex table, and enabling the use of the write-

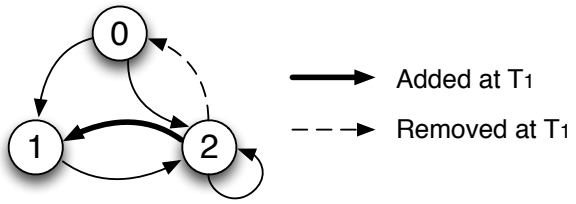


Figure 4.3: Example Graph with one added and one removed edge in Snapshot 1.

optimized store (Section 4.7) in the analytics computation. Note that there are many ways to implement a delta map, so different systems with different delta map implementations exhibit different performance, but the overall takeaways still hold.

The biggest difference between LLAMA's representation of CSR + Delta Map and its canonical implementation is an extra indirection when translating an edge ID into the corresponding edge table to a pointer, which happens once per iterator creation. Fortunately, this has minimal impact: There is only one edge table, because we constrained the read-optimized store to use only one snapshot. The array that maps snapshot IDs to edge tables has only one element in it, and it comfortably fits into a single cache line and remains in the L1 cache throughout the computation.

4.3.2 Log-Based CSR (Performance-Optimized Edge Table)

We represent log-based CSR (Section 2.1.2) in LLAMA by storing only complete adjacency lists in the edge tables of the delta snapshots. We refer to LLAMA's implementation of this approach as the *performance-optimized* design because, intuitively, using only complete adjacency lists (as opposed to storing only differences) sacrifices space for performance.

Some modern systems use copy-on-write vertex tables[60] to support multiversioned access and transactional semantics, while others do not[7]. We use the multiversioned vertex tables with the performance-optimized design unless stated otherwise.

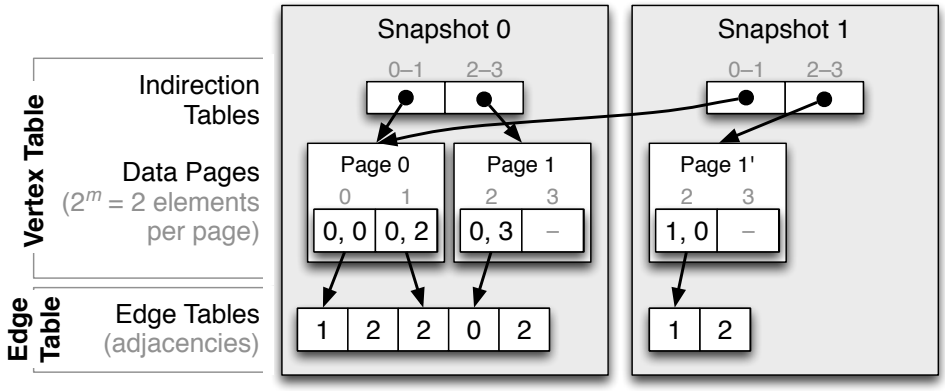


Figure 4.4: Performance-Optimized Design with the copy-on-write (LAMA) vertex table that represents the graph from Figure 4.3. Delta snapshots contain complete adjacency lists. The elements in the vertex table contain the snapshot number and index into the corresponding edge table, omitting the adjacency list fragment length for brevity.

Figure 4.4 shows how LLAMA's performance-optimized design represents the two-snapshot graph from Figure 4.3, which has one added and one removed edge, both incident to vertex 2.

Log-based CSR systems often keep all adjacency lists together in a single edge table and grow it using `realloc`, but it is also possible to use growable arrays implemented using an indirection table. Note that either method results in efficient allocation, because modern operating systems, such as Linux, implement efficient `realloc` by manipulating page tables[61]. The former approach results in marginally better performance, while the latter enables the system to easily grow its edge table in the presence of concurrent long-running computations.

The only potential difference between the canonical log-based CSR that uses `realloc` and our implementation is again the extra indirection table translating snapshot IDs to edge tables, just like in our representation of CSR augmented with a delta map. This indirection table is small as long as we do not use more than a few tens or even hundreds of snapshots, and it remains in the cache throughout the computation, resulting in minimal overhead. On the other hand, our implementation behaves just like log-based CSR implementations that use growable arrays instead of `realloc`.

4.4 Mutable, Space-Optimized Edge Table

As we will see in Section 4.9, although the performance-optimized design has performance comparable to immutable CSR, its memory footprint grows quickly with the number of snapshots. Creating new read-only snapshots as a consequence of many graph changes results in having multiple copies of a large number of adjacency lists. While we can solve this with frequent merging, there is, in fact, a better alternative: a *space-optimized design* that trades a small performance penalty for a much lower memory overhead that does not increase significantly with the number of snapshots. Merging becomes less necessary, even to the point of being completely unnecessary for many workloads.

As before, snapshot 0 is fully self-contained and consists of complete adjacency lists. The edge tables of delta snapshots store only newly added edges (i.e., adjacency list fragments) instead of complete adjacency lists. This saves space but requires that we examine multiple snapshots to reconstruct an adjacency list for a given node.

We have two options for linking adjacency list fragments belonging to the same vertex:

- *Implicitly* by leveraging the multiversioned vertex table, and:
- *Explicitly* using next pointers similarly to a linked list.

The former approach is more space efficient, but the latter approach is often faster, and thus it is the default in LLAMA. These approaches also differ in how they implement edge deletion. We now explain both of these alternatives.

4.4.1 Implicit Linking

Figure 4.5 shows an example of space-optimized design with implicit linking that represents the graph in Figure 4.3. The delta snapshot contains only the new edges, and a *deletion vector* takes care of deletions.

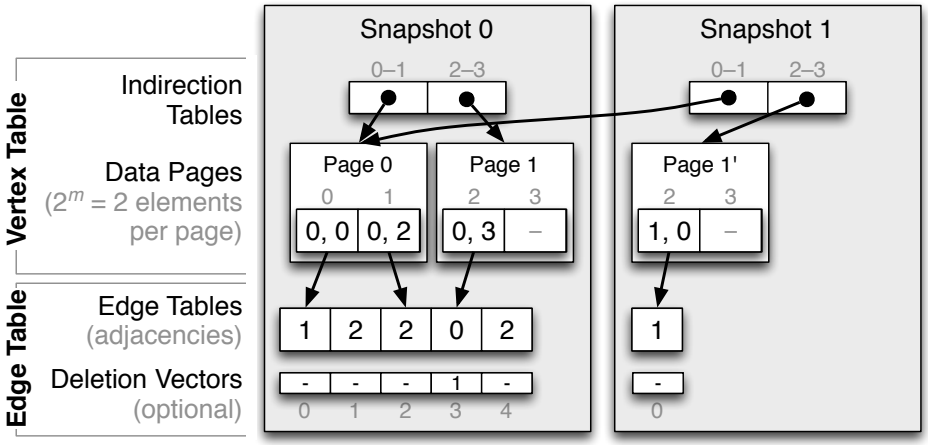


Figure 4.5: Space-Optimized Design with Implicit Linking and the copy-on-write (LAMA) vertex table that represents graph in Figure 4.3. The deletion vector marks the snapshot number at which the given edges were deleted; this is optional for workloads that do not need to support deletions.

Deletion vectors are arrays logically parallel to the edge tables that encode in which snapshot an edge was deleted. In Figure 4.5, snapshot 0's deletion vector indicates that edge $2 \rightarrow 0$ was deleted in snapshot 1. If using 64-bit vertex IDs, we can embed the deletion vector values in the top d bits of the vertex ID representing the head in the snapshot's edge table, where 2^d is the maximum number of snapshots required. This trades vertex ID space for deletion vector space. By default, we allocate 16 bits for the deletion vector and 48 bits for the actual vertex ID. If the user has specified 32-bit vertex IDs, then we store the deletion vector separately.

Recall that the vertex table entry for vertex v contains the location of the first adjacency list fragment as tuple (S, I), where S is the snapshot and I is the index. Note also that the referenced snapshot S can be smaller than the snapshot ID of the vertex table that contains this record. We start reconstructing the corresponding adjacency list by first reading this fragment. We check the deletion vector value for each edge and include the edge in the result only if the corresponding value is NIL, or in the case of querying on an older snapshot, if it is greater than the snapshot ID in which the

```
function OutEdges(G, v, query_snapshot)

    adj_list ← {}
    S ← query_snapshot
    repeat
        // Get v's entry in the vertex table of Snapshot S
        E ← G.VertexTable[S, v]
        if E is NIL then
            break
        end if

        // v's adj. list continues at index E.index of the edge table of snapshot E.snapshot
        S ← E.snapshot
        for i = 0 ... (E.length − 1) do
            k ← E.index + i
            d ← G.DeletionVector[S, k]
            if d is NIL or d > query_snapshot then
                adj_list ← adj_list ∪ {G.EdgeTable[S, k]}
            end if
        end for
        S ← S − 1
    until S < 0
    return adj_list
end function
```

Figure 4.6: Reconstructing an Adjacency List from multiple snapshots using the space-optimized design with implicit linking.

user is interested. When we are done with the given adjacency list fragment, we get v's record from the vertex table from the next lowest snapshot, $S - 1$, which points to the next adjacency list fragment at a snapshot with ID smaller than or equal to $S - 1$. The reconstruction is complete when we find a NIL entry for v or when we reach snapshot 0. The algorithm in Figure 4.6 summarizes this process. v is the vertex ID and $query\_snapshot$ is the ID of the snapshot in which the user is interested (usually the most recent snapshot). For example, if we want to reconstruct the adjacency list for node 2 as of snapshot 1 from Figure 4.5, we first examine snapshot 1. We see that node 2 has one

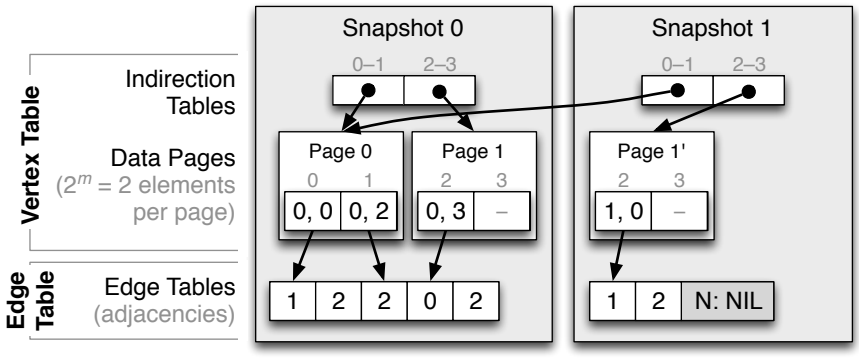
element in its list and that that element has NIL for its deletion vector, so we add it to the adjacency list. Next we examine snapshot 0. At snapshot 0, there are two entries for node 2. For the first (0), the deletion vector value is set to 1 which is equal to the query-snapshot, so we ignore it. The next entry (2) has a NIL for its deletion vector value, so we include it. There are no further snapshots, so our result is the list $\{1, 2\}$.

If we wanted to issue the query as of the time of snapshot 0, we would have examined only the entries at snapshot 0. Since the query-snapshot would be 0 and the deletion vector value 1, which is greater than 0, we would see that the delete took place after the query time. We would thus add the edge to the adjacency list and end up with the list $\{0, 2\}$.

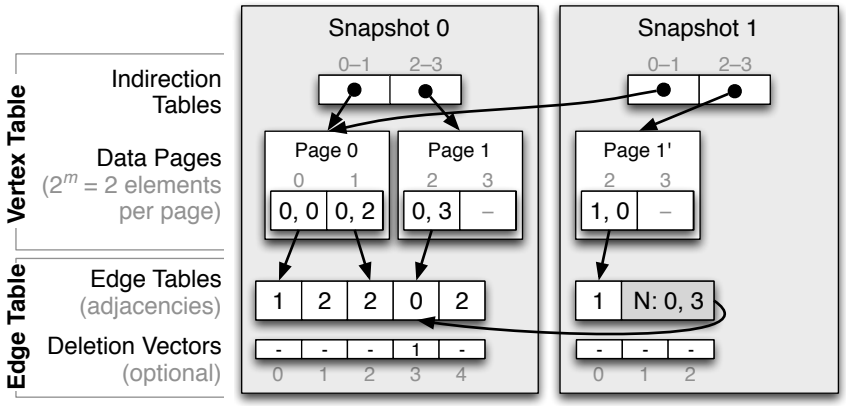
An important property of this algorithm, not illustrated by these examples, is that if the adjacency list of the vertex is spread over n snapshots, at most $n + 1$ vertex tables and n edge tables are examined, even if the total number of snapshots in the graph is much larger. For example, consider a graph with 10 snapshots, in which the adjacency list fragments of some node are in snapshots 5 and 8. The algorithm first accesses the vertex table of snapshot 10, which will tell it that the adjacency list starts at some index in snapshot 8. After processing this list, the algorithm descends to snapshot 7, which will then point it to the continuation of the adjacency list in snapshot 5. After finishing this part, the algorithm will then examine the vertex table in snapshot 4 and terminate, because it does not contain any information about the vertex.

4.4.2 Explicit Linking with Next Pointers

Instead of examining multiple versions of the vertex table, we can add *next pointers* to the end of all adjacency list fragments in snapshots other than snapshot 0, in which the NIL pointer marking the end of the adjacency list is implicit. The next pointer contains the information needed to find and traverse the next fragment: the snapshot ID, the index into the corresponding edge table, and the length of that fragment. The adjacency list fragments form a linked list.



(a) Without Deletion Vector (Default)



(b) With Deletion Vector

Figure 4.7: Space-Optimized Design with Explicit Linking, the LAMA vertex table without deletion vectors (a) and with deletion vectors (b). The figure depicts the graph from Figure 4.3.

If the adjacency list of the vertex is spread over n snapshots, only 1 version of the vertex table and n edge tables are examined. This is an improvement from implicit linking, saving us $n - 1$ to n accesses to the multiversioned vertex table, This makes a significant difference especially on large graphs. Next pointers increase the total data structure size, making it less space efficient, but determining where the adjacency list continues becomes more cache-efficient.

By default, LLAMA deletes edge $u \rightarrow v$ by writing u's entire adjacency list (without the deleted edge) into a new snapshot followed by a NIL next pointer. This approach has essentially no perfor-

mance overhead as long as the copied adjacency lists do not significantly increase the database size. When deletions are frequent or adjacency lists are sufficiently large that copying them takes up too much space, or if explicit linking is disabled, the user can instead choose deletion vectors.

Figure 4.7 shows the space-optimized design with explicit linking and the two methods for implementing deletions on the graph from Figure 4.3.

4.5 Parameterized (Hybrid) Edge Table Designs

Having now presented a representation optimized for performance and a similar one optimized for space, it is only natural to ask if it is possible to develop a representation that works well in both dimensions. Our hybrid design is just such a representation, improving performance of algorithms that do not work particularly well in the space-optimized design and having lower space overhead than the performance-oriented design.

The performance and space optimized designs occupy two extreme points of the spectrum ranging from "pass by copy" to "pass by reference" applied to adjacency lists. The performance optimized design implements adjacency lists by copying them; the space-optimized design implements adjacency lists by chaining references. We can create points along that spectrum by dynamically choosing when to copy and when to reference.

We chose two interesting points along this spectrum to investigate in detail:

- Represent small adjacency lists by reference and large adjacency lists with copies.
- Represent small adjacency lists with copies and large adjacency lists with references.

Implementing these is relatively straightforward, and if we ignore deletions, we can base our decision of copy versus reference on the degree of the node, which is readily available in the vertex table.

We have a similar option of using implicit or explicit linking as in the space-optimized design. We use explicit linking by default: using NIL pointers for copied adjacency lists and valid next pointers

```
for all Vertex v ∈ the writable representation do
    if (v's adjacency list contains any deleted edges
        or (copy-large and the vertex degree ≥ cutoff)
        or (copy-small and the vertex degree ≤ cutoff))
     then
        Copy the entire adjacency list into the new snapshot
    else
        Copy the modified edges into the new snapshot (and add the next pointer or set
        the "to be continued" bit)
    end if
end for
```

Figure 4.8: Creating a New Snapshot in the Parameterized (Hybrid) Design.

for incomplete adjacency lists. The other option is to use implicit linking with an additional bit per vertex that indicates whether the vertex's adjacency list is complete (it was copied) or whether it is a by-reference list and is composed by traversing multiple snapshots of our representation.

We use a similar approach to deleting edges as in the space-optimized designs: copy the adjacency list in the presence of a delete (regardless of the node degree), or use deletion vectors if we expect too many deletions. We use the former by default.

Putting this all together, we end up with a representation parameterized by two values: a degree cutoff and a bit that indicates copy-large versus copy-small. We create a new snapshot using the algorithm in Figure 4.8, which copies the entire edge list or just the modified edges into the new snapshot, depending on the policy.

4.6 Mutable Properties

LLAMA implements graph element properties using LAMAs – the same large multiversioned arrays that we use to implement the multiversioned vertex table. We store each property in its own LAMA, parallel to either the vertex or the edge table. (The index into the LAMA is the same as the

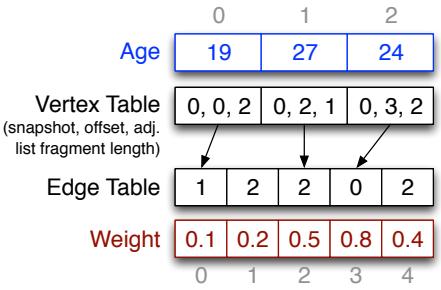


Figure 4.9: Example of a Single-Snapshot LLAMA with a Vertex and an Edge Property. The figure contains just one snapshot and just the high-level representation (omitting the internal structure) of the LAMAs – the vertex and the property tables – for clarity.

index into the table.) Figure 4.9 shows the LLAMA representation of a 3-node graph with one vertex property "Age" and one edge property "Weight". The snapshots and internal structure of the vertex table are omitted for clarity. A LAMA can store only fixed-size elements; we use a separate key/value store for variable-sized properties.

We can alternatively implement edge properties as flat arrays (instead of LAMAs) parallel to the corresponding edge table arrays and update the properties by deleting the edge and then re-adding it with the new property value. This improves performance as it allows us to bypass a level of indirection, but it comes at the cost of space unless the property updates are rare. (LLAMA does not currently use this design by default.)

Separating storage of properties from the graph structure draws on the column-store philosophy from relational databases: load only the needed parts of the data. For example, many algorithms need only outgoing edges. Loading only the necessary edges and properties leads to superior performance relative to systems that must load all data associated with a vertex or an edge. On the other hand, this representation is less efficient for cases in which the algorithm needs all attributes.

4.7 Implementation of the Write-Optimized Store (Delta Map)

LLAMA buffers incoming updates in a write-optimized store, also called the delta map. We represent each modified vertex by an object containing a list of new and deleted out- and in- edges. We store these vertex objects in a custom high-throughput in-memory key-value store that supports only fixed-length keys.

By default, LLAMA excludes this delta map from analytics workloads, treating it instead as an update buffer. It is frequently more efficient to create a new LLAMA snapshot from the delta map before running an analysis than to access the delta map during analysis. For example, as we will see in Section 4.9, running PageRank[57] on a LiveJournal graph[3] in which 80% of edges are loaded in CSR and 20% in a delta map takes 11.2 seconds on our platform (dual-core Intel Core i3, 3.0 GHz, 8 GB RAM). Creating a read-optimized snapshot takes approximately 1.2 seconds, and running PageRank then takes 5.1–6.5 seconds depending on the LLAMA configuration. The whole operation takes 6.3–7.7 seconds, which is significantly faster than 11.2 seconds for CSR + Delta Map.

The write-optimized store consists of a vertex table where each element contains either NIL (if the vertex has no modified edges) or a vertex object containing a growable array of added and removed incoming and outgoing edges. Like LAMA, the vertex table consists of both an indirection table and data pages. The indirection table starts out empty, demand allocating pages as needed. To support the addition of vertices, we allocate the indirection table larger than immediately necessary; allocating an extra 1 MB is enough to add approximately 15 million new vertices.

Figure 4.10 shows an example of a write-optimized store recording that edge $2 \longrightarrow 1$ was created in graph from Figure 4.3. Edge deletion is implemented using deletion vectors in the read-optimized store.

The writable store does not require locks for reading, and we use a fast fine-grain latching scheme to coordinate concurrent updates. We insert elements into the vertex table using compare-and-swap,

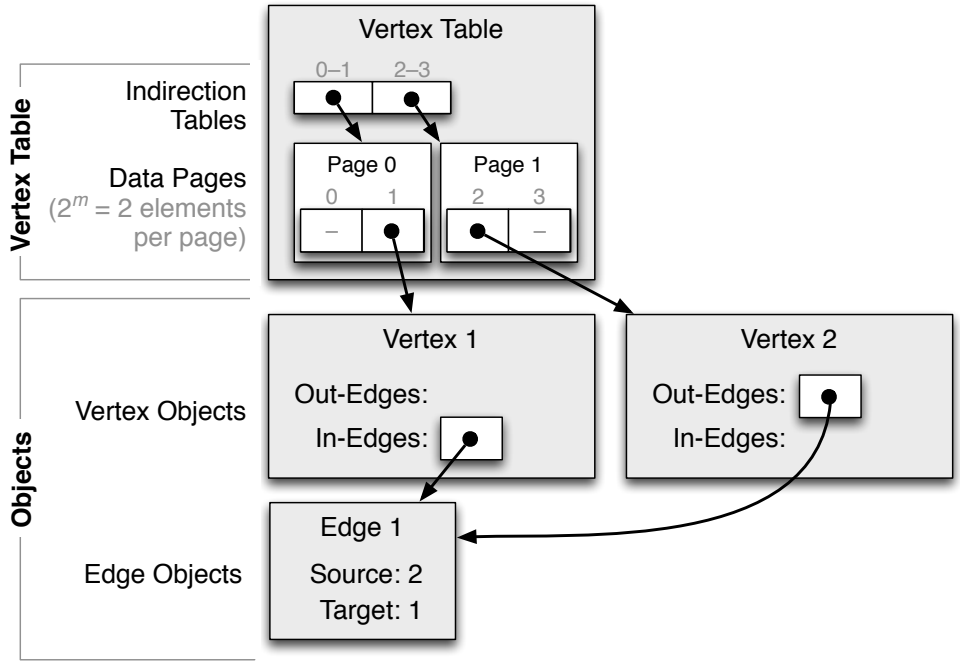


Figure 4.10: Write-Optimized Store (i.e., the Delta Map), representing the addition of edge $2 \longrightarrow 1$ in graph from the Figure 4.3.

both when inserting new data pages into an indirection table and when inserting new vertex objects into the data pages. If the original value in the table was NIL, we replace it with a special value, which we call BLOCK. In the case of the indirection table, we then proceed to allocate a corresponding data page, and once the allocation succeeds, we replace BLOCK with the valid pointer. Inserting demand-allocated objects into data pages works similarly.

A BLOCK value in the vertex table means that another thread is allocating memory, so concurrent accesses loop until the value becomes valid, thus treating it as a spin-lock. Read-only operations treat BLOCK as NIL (recall that the only time we see BLOCK is during allocation) and thus do not require locks.

We implement the growable arrays using a thin indirection layer that allows us to grow them without having to move any of their contents in memory, and we delete items from these arrays by

setting a flag instead of shifting the contents. When we add an edge, we latch both endpoints, append the new edge information to the ends of their adjacency lists, update the counters (such as the length of the adjacency list and the precomputed degree), and release the latches. We acquire locks in vertex ID order to prevent deadlocks. Again, we do not need to acquire any locks to read the adjacency lists. We leverage x86 atomicity of word accesses to read node metadata (e.g., the precomputed degree) without locking.

As with our other design decisions, this design was the result of experimentation with alternatives, until we found one that was both highly concurrent and performant.

4.8 LLAMA's Operation

4.8.1 Loading Data

LLAMA's bulk loader accepts two edge-list based formats:

- a text file with one edge per line represented as two white-space separated vertex IDs (used by SNAP[66])
- X-Stream[63]'s binary file format in which an edge is stored as two unsigned integers representing the endpoint vertices followed by a weight

These can be either complete graphs or a list of new edges to be added as a new snapshot.

Ingest is most straightforward with sorted edge lists, so if necessary, we first sort. Then, we scan the input edges to build an array of vertex degrees; the cumulative sum of this array provides each vertex's index into the edge table. We create an edge table of the appropriate size and then read the file a second time, copying the target vertex IDs directly into the edge table. Because we sorted the edges, this writes the edge table sequentially.

4.8.2 Freezing the Write-Optimized Store

Another way to load data into the read-optimized store is by creating a new read-only level from the current contents of the write-optimized store, which buffers incoming updates and supports queries that require the most recent data. The edges in the write-optimized store are already grouped by their source vertex ID, and the number of new edges for each vertex is thus likewise readily available.

We then proceed the same way as the bulk-loader: create the new version of the vertex table from the cumulative prefix sum of the array of the counts of the new edges for each vertex, and then sequentially copy the target vertex IDs directly to the edge table. We clear the contents of the vertex table when finished.

4.8.3 Merging Snapshots

We do not normally delete snapshots; instead we merge all snapshots into a single new LLAMA. Merging is a fast, straightforward operation: for each vertex, read its adjacency list, sort it by the target vertex ID, and write it out to the new edge table.

Deleting a single snapshot or a group of snapshots proceeds similarly: we merge into the next oldest snapshot. This involves reading data from each of the merged snapshots and writing it out to a new snapshot. We support only merging the most recent snapshots. Deleting older snapshots requires updating the vertex tables and continuation records in newer snapshots; implementing this is left for future work.

LLAMA's operation is similar to an LSM tree[35,54]: the system always writes data into a new snapshot and eventually merges the snapshots together, except where multiple versions are specifically desired.

4.9 Quantitative Design Evaluation

Throughout our design process, we evaluated alternatives by quantifying performance and space consumption. Regardless of whether we were comparing to our own ideas or existing designs, we implemented everything in the same framework to isolate each decision. Memory footprint is important for in-memory systems, because it allows us to process larger graphs on smaller machines, but it is also important for out-of-core processing. Efficient memory usage in combination with good cache locality can significantly reduce I/O. We describe the out-of-core aspects pertaining to LLAMA in the next chapter and present an in-depth evaluation on both in-memory and out-of-core workloads in Chapter 6.

We begin by evaluating the various approaches to CSR mutability on the *LiveJournal* graph[3], a real-world graph with 4.8 million nodes and 69.0 million edges. This graph occupies approximately 600 MB of memory when loaded into a single snapshot in the read-optimized representation in LLAMA, irrespective of a specific approach to CSR mutability (with the exception of delta maps). When broken into 11 snapshots and loaded into the least memory-efficient implementation, it can grow as large as 4 GB.

We first evaluate the non-parametric methods and then move on to evaluate the parameterized (hybrid) approach. We answer the following questions for each evaluated method: 1) What is the memory usage, and how does it grow with the number of snapshots?, 2) What is the performance, and how does it scale with the number of snapshots?, and: 3) How long does it take to ingest a graph into one or more snapshots?

We start by loading an initial graph, then incorporating a stream of updates, running analytics, and merging periodically. We evaluate each phase in isolation to better understand the various design decisions we have made. We use two main analytic benchmarks: ten iterations of PageRank[57] (PR) and breadth-first search (BFS), starting from the most well connected node in the graph. We

chose these benchmarks as they represent the two main types of access patterns: PageRank processes the whole graph in vertex order, while BFS (implemented using a standard queue-based approach) results in a more random access pattern.

We then follow this initial evaluation by an in-depth study of the performance and space optimized variants on synthetically generated R-MAT graphs[10] of varying size (expressed as an average vertex out-degree) and skew. This allows us to answer the following question: What is the relationship between the graph size and skew and the relative performance of the two parameterless approaches? (We focus on the parameterless approaches as they represent two extremes in the spectrum of the parameterized approaches.)

We run all benchmarks on a commodity machine with a dual-core Intel Core i3, 3.0 GHz and 8 GB RAM with Hyper-Threading (for a total of four hardware threads). We use all four hardware threads unless stated otherwise.

4.9.1 Methodology for Creating Multiversioned Graphs

We load graphs into 1, 2, 6, and 11 snapshots, which we create as follows: For $n > 1$ snapshots, we first load a random 80% subset of edges as the first snapshot and then used a uniform random distribution to assign the remaining edges into the other $(n - 1)$ snapshots. This simulates an initial large ingest followed by multiple smaller ingests. This also describes a system in which we create snapshots in regular time intervals with different update rates: The scenario with 2 snapshots models a case in which all 20% of the updates arrive in the same time interval, while the scenario with 11 snapshots models a case in which the updates arrive ten times more slowly, with only 2% of the updates arriving per time interval. The number of snapshots is therefore inversely related to the update rate.

We choose which edges to treat as updates uniformly at random as this produces a worst-case scenario (without specifically constructing adversary workloads). Bursty updates would only im-

prove locality, as more edges originating from the same vertex would be more likely to appear in the same snapshot. This would in turn decrease the average number of snapshots in which each vertex appears. This would also lower memory usage, as we would need to copy-on-write fewer vertex table pages in each snapshot when using LAMA, write fewer next pointers when using the space-optimized design with explicit linking, and rewrite fewer adjacency lists in the performance-optimized and hybrid designs. This would also increase our performance, which is a function of the number of snapshots in which each vertex appears.

We run each experiment five times and report the averages. Most standard deviations for PageRank are less than 1% and less than 2% for BFS.

4.9.2 Evaluation on the LiveJournal Graph

We evaluated the following configurations to evaluate the differences between the performance-optimized design, the space-optimized design with the two types of adjacency list linking, and their variants – plus two baselines:

1. Immutable CSR – the entire graph loaded in the standard CSR data structure. This is our baseline, against which we evaluate the overhead of adding mutability
2. CSR + 20% graph in Delta Map – a second baseline, showing the performance of a standard implementation that has 80% of its edges in CSR and 20% in a delta map
3. Perf. + Flat – the performance-optimized design with the flat vertex table implemented as a single modifiable array
4. Perf. + LAMA – the performance-optimized design with the multiversioned LAMA vertex table
5. Space + Expl. + Flat – the space-optimized design with explicit adjacency list linking using the next pointers and with the flat vertex table

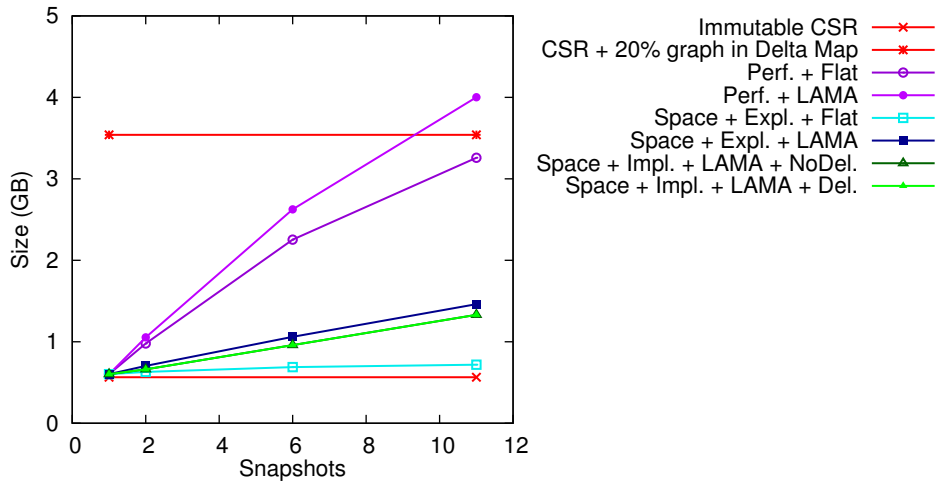


Figure 4.11: The In-Memory Size of the LiveJournal Graph for the various mutable CSR configurations, plus two baselines: immutable CSR, and 80% of edges in CSR plus 20% of edges in delta map.

6. Space + Expl. + LAMA – the same as above, but with the LAMA vertex table
7. Space + Impl. + LAMA + NoDel. – the space-optimized design with implicit linking and LAMA vertex table, but with no support for deletions
8. Space + Impl. + LAMA + Del. – the same as above but with a deletion vector

In-Memory Size

Figure 4.11 shows the in-memory size of the LiveJournal graph when it is loaded to 1, 2, 6, and 11 snapshots. Storing 20% of the edges in the delta map consumes almost 6 times the memory as does the 80% of the edges in the immutable CSR. While there are certainly more steps that we could take to decrease the memory footprint, this illustrates that storing adjacency lists in separate lists has a fundamentally higher memory footprint than CSR.

The in-memory footprint of a graph loaded into LLAMA is approximately 6.6% larger than that of an immutable CSR, when the entire graph is contained in a single snapshot, showing little

variance between the different approaches (except the CSR + Delta Map, which we did not load in a single-snapshot mode). The extra level of indirection in the LAMA vertex table accounts for less than 0.1%, which is the overhead of the "+ LAMA" data points relative to the corresponding "+ Flat" data points for 1 snapshot (this small difference is impossible to see in the plot). Storing additional information in the vertex table, such as the vertex degree, accounts for the remaining 6.5%; this applies to all data points other than immutable CSR.

The memory size then grows at different rates for the different approaches. The rates break down into the following three groups (all numbers are expressed as the percentage overheads relative to the immutable CSR baseline):

- Space-optimized design with the "Flat" vertex table grows approximately 3.3% per additional snapshot due to the cost of the "next" pointers at the ends of the adjacency list fragments in the delta snapshots at 16 bytes per next pointer.
- Space-optimized designs with the "LAMA" vertex table grow 12.0% per snapshot for implicit and 16.5% for explicit adjacency list linking, most of which is the cost of adding new versions to the LAMA vertex table. Note that this is indeed the worst-case behavior, since we selected edges for the delta snapshots uniformly at random, causing the adjacent vertex IDs to be spread throughout the entire ID space producing the copy-on-write behavior on most vertex table pages.
- The performance-optimized design grows by 57.4% per snapshot for the "Flat" vertex table and 70.5% for the LAMA vertex table. This steep growth is almost entirely due to copying complete adjacency lists instead of storing just the differences as in space-optimized designs.

The in-memory size of a snapshot depends on the number of edges in the snapshot, the number of updated adjacency lists, the number of COW-ed LAMA pages (which in turn depends on the locality of updated vertices), and the number of copied edges (especially in the performance-optimized

design, but also in the space-optimized design if deletion vectors are not used). We evaluate memory sizes for various degree skews later in this section when studying LLAMA on R-MAT graphs.

A vertex table entry consumes 16 bytes, and each edge consumes 8 bytes corresponding to the destination vertex IDs, so the size of a single-snapshot graph in bytes is simply:

$$16V + 8E$$

In the case of the LiveJournal graph with 4.8 million vertices and 69 million edges, the memory cost is: $16 \times 6.8 \times 10^6 + 8 \times 69 \times 10^6 = 660.8 \times 10^6 \approx 630$ MB. LLAMA uses 8 KB pages by default, each holding 512 vertex table elements, so the cost of an indirection table is then $(6.8 \times 10^6/512) \times 8 \approx 0.1$ MB for each snapshot (including Snapshot 0).

Now we consider multi-snapshot graphs. The number of edges in the space-optimized designs is identical regardless of the number of snapshots, because each edge is stored only once. The extra space overhead comes from COW-ed LAMA pages and from next pointers when using explicit linking. The number of COW-ed vertex table pages depends on the number of updated vertices and on how collocated they are. The number of new edges in an 11-snapshot graph with 2% edges per delta snapshot is $0.02 \times 69 \times 10^6 = 1.38 \times 10^6$, while there are only $4.8 \times 10^6/512 \approx 0.013 \times 10^6$ vertex table pages. In the worst case, we need to COW all pages; doing so costs $16V = 16 \times 6.8 \times 10^6 = 108.8 \times 10^6 \approx 104$ MB per snapshot. We COW-ed on average 72% pages per snapshot in our experiments, resulting in 74.5 MB of COW-ed pages per snapshot.

A next pointer in LLAMA occupies 16 bytes. If E_i is the number of edges in snapshot i, the maximum number of adjacency list fragments in the snapshot is $\min\{E_i, V\}$, so the maximum memory cost is $16 \times \min\{E_i, V\}$. In the case of the 11-snapshot LiveJournal graph, this would be $16 \times \min\{1.38 \times 10^6, 4.8 \times 10^6\} = 22.08 \times 10^6 \approx 21$ MB. We determined that there are actually 0.88×10^6 adjacency list fragments per snapshot on average. The corresponding memory cost of next pointers is 13.4 MB per snapshot.

More formally, if η is the average fraction of vertices updated in a snapshot, these vertices belong on average to fraction π of pages, and S is the number of snapshots, the size of space-optimized LLAMA with explicit linking and multiversioned vertex table is:

$$16V + 8E + 16\pi SV + 16\eta SV = 16V(1 + \pi S + \eta S) + 8E$$

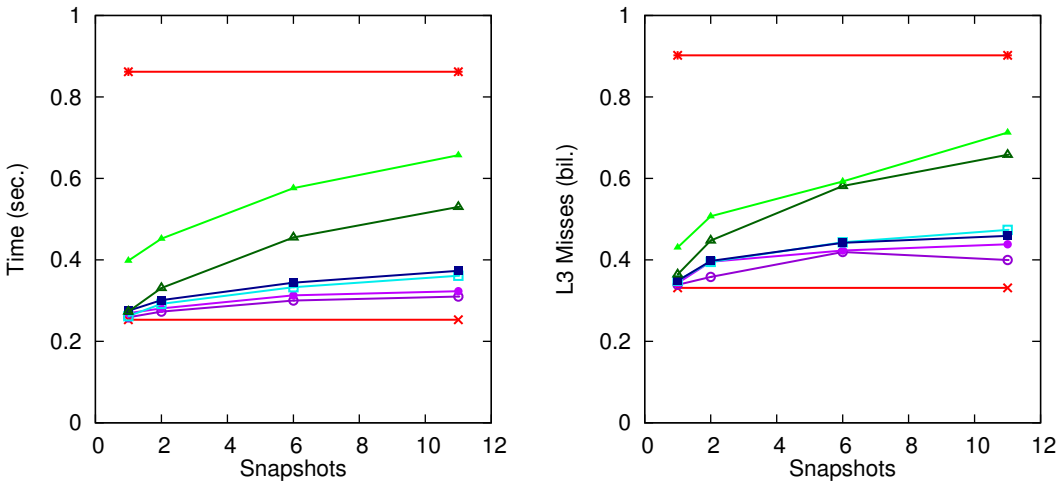
The cost of space-optimized LLAMA with explicit linking and flat vertex table is simply $16V(1 + \eta S) + 8E$ bytes, and the cost of LLAMA with implicit linking is $16V(1 + \pi S) + 8E$ bytes.

In the most degenerate case of the performance-optimized design, we would copy all edges in an edge table when creating a new snapshot. If Snapshot 0 contains 80% of edges, this would copy $8 \times 0.8 \times 69 \times 10^6 = 441.6 \times 10^6 \approx 421$ MB to Snapshot 1. In the case of a 11-snapshot graph, we copy 35.6×10^6 edges on average (approximately 65% of edges in Snapshot 0), which costs 272 MB per snapshot. If χ is the fraction of edges copied per snapshot, the memory size is approximately:

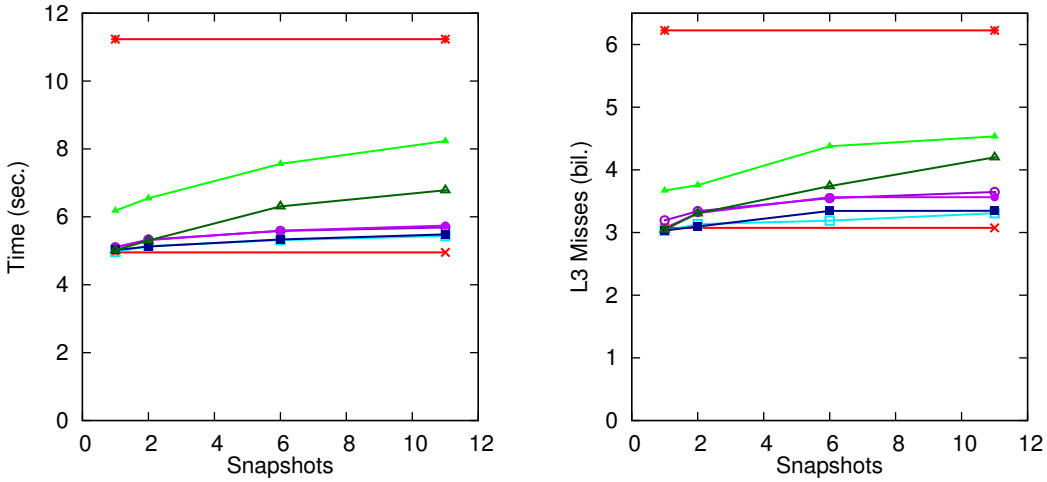
$$16V + 8E + 16\pi SV + 8\chi SE = 16V(1 + \pi S) + 8E(1 + \chi S)$$

The performance-optimized design has a significant memory cost, which can lead to frequent merges to maintain a reasonable bound on memory usage. The cost of the space-optimized design grows much more slowly, making it preferable.

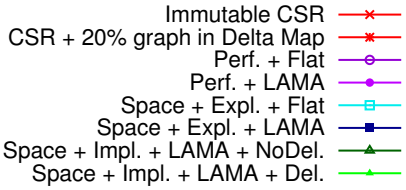
If memory usage is a concern and multiversioned access is not needed, the user should choose the space-optimized design with explicit linking and the flat vertex table, as it has the lowest and slowest-growing memory cost. The user pays only for the "next" pointers, which are relatively small compared to the total size of the edges in a snapshot, and occasional copies of adjacency lists due to deletions, which have an acceptable cost if deletions are not common. If deletions are more frequent, the user can still choose this design but may want to use deletion vectors instead of copy-on-delete, which has similar memory costs (not shown in the figure).



(a) BFS, elapsed time (left) and cache misses (right)



(b) PageRank, elapsed time (left) and cache misses (right)



(c) Legend

Figure 4.12: Scaling with the Number of Snapshots on the LiveJournal Graph on the Commodity platform, plus two baselines: Immutable CSR and CSR + delta map.

The Runtime Cost of Supporting Mutability in a Single-Snapshot Graph

Figure 4.12 shows the performance and the number of L3 cache misses for BFS and PageRank on the LiveJournal graph as we vary the graph representation from 1 to 11 snapshots.

When the entire graph resides in a single snapshot, the elapsed time is close to that of the immutable CSR baseline in all but few cases: BFS ran 2.4–8.7% slower than immutable CSR for all cases except for the deletion vector representation, which ran 57.3% slower. The overheads for PageRank were lower: 0.3–3.2% slower for most cases and 24.8% slower for the space-optimized design with the deletion vector. (We omit the CSR + Delta Map baseline from this discussion, since it does not apply to the single-snapshot case.) These numbers quantify the cost of enabling mutability without actually using it, since we are considering the single snapshot case.

The cost of mutability decomposes into the following components:

- A level of indirection when accessing the edge table while opening an iterator (negligible cost)
- Accessing a larger vertex table due to storing the length of the adjacency list fragment (see Sections 2.1.1) and the vertex degree (Section 4.1), which costs 0.3% for PageRank and 3.6% for BFS (this is the difference between the performance of the immutable CSR and the performance of the two mutable CSR configurations that use the "Flat" vertex table)
- In the case of multiversioned LAMA vertex table, the extra level of indirection when accessing the vertex table, which costs 0.2–1.1% for PageRank and 4.0–5.1% for BFS (the difference between the "Flat" and "LAMA" numbers)
- When using the deletion vector, the cost to check it for each edge, which costs 23.2% and 45.8% respectively

The cost of the deletion vector is especially significant because it requires four additional instructions, including a conditional branch, per each accessed edge. The overheads pertaining to adding

the extra levels of indirection are comparably insignificant because their costs are "charged" once per adjacency list fragment or once per vertex table access, not once for each edge.

The Runtime Cost of Multi-Snapshot Graphs

The performance curves for all evaluated variants, for both BFS and PageRank, and for all tested numbers of snapshots, are always between our two baselines, slower than immutable CSR and faster than CSR plus the delta map. Space-optimized designs with explicit linking ("Space + Expl.") and the performance-optimized designs ("Perf.") both perform significantly better than either of the space-optimized designs with implicit linking ("Space + Impl."), but there is no clear winner between the two.

The choice of the vertex table implementation for these two designs does not affect the results significantly: using LAMA instead of flat arrays adds only a small constant overhead irrespective of the number of snapshots, because the vertex table is accessed the same number of times regardless of the number of snapshots.

The performance-optimized design performs better for BFS, which has a more random access pattern, while the space-optimized design performs better for PageRank, which processes the graph strictly in vertex order. Processing the graph in vertex order is well suited for the space-optimized design, because its access pattern decomposes into n parallel sequential edge table scans for a graph with n snapshots. This results in great cache behavior, because the adjacency lists are stored consecutively in the order in which they are processed, which is great for prefetching. For example, consider a graph with three snapshots and two vertices: v_1 has adjacency list fragments in all three snapshots, and v_2 has its fragments in snapshots 0 and 2. Reading v_1's adjacency list for the first time (starting with a cold cache) generates a cache miss for each of the three edge tables. Accessing v_2's adjacency list fragment in snapshot 2 is likely to be satisfied from the cache, because the data immediately be-

Variant	BFS	PageRank
Perf. + Flat	1.94%	1.08%
Perf. + LAMA	2.12%	1.19%
Space + Expl. + Flat	3.67%	0.86%
Space + Expl. + LAMA	3.69%	0.90%
Space + Impl. + LAMA + NoDel.	9.88%	3.59%
Space + Impl. + LAMA + Del.	10.01%	4.10%

Table 4.1: Latency Growth with the Number of Snapshots – the additional overhead over computing on an immutable CSR for each additional snapshot for the first 10 delta snapshots. (These numbers are slopes for the linear regression of the performance curves in Figure 4.12 as percentages of the latency corresponding to the immutable CSR.)

forehand were recently accessed and are therefor likely still in the cache. Accessing the next adjacency list fragment from snapshot 0 is then also likely to be satisfied from the cache.

The access pattern in the performance-optimized design is less sequential, since the system reads adjacency lists of different vertices from different snapshots, skipping over large regions used by old versions of the adjacency lists. This results in more cache misses as illustrated by Figure 4.12. The performance-optimized design performs better when the processing order is more random, as it is in the case of BFS. Accessing an adjacency list is less likely to be satisfied from the cache, so having the entire adjacency list contiguous in memory is advantageous. The space-optimized design has to reconstruct a fragmented list from multiple pieces, resulting in an even more random access pattern and more cache misses.

The space-optimized design with implicit linking is slower than explicit linking, because it requires additional accesses to the vertex table to determine where an adjacency list continues. The explicit linking design gets this information from the "next" pointer in the edge table, which is likely to be satisfied from the cache. In comparison, the implicit linking design accesses an older version of the vertex table, which is less likely to be in cache, in addition to requiring an extra level of indirection.

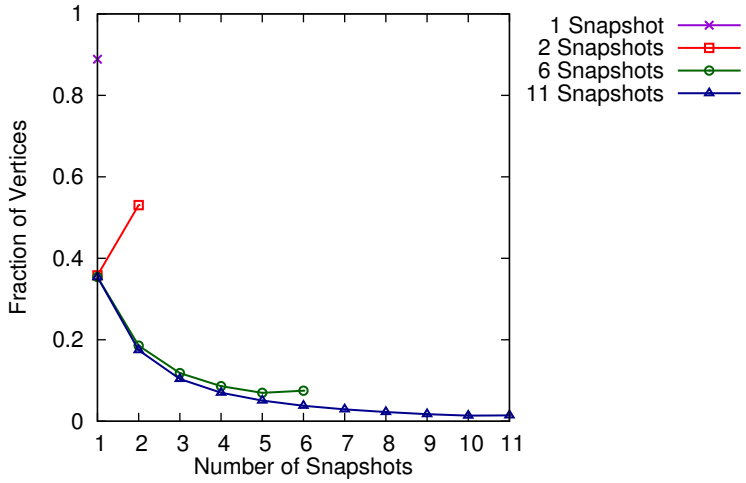


Figure 4.13: Vertex Spread for the LiveJournal Graph. The graph shows how many vertices (or adjacency lists) appear in the given number of different snapshots, omitting the fraction of vertices appearing in 0 snapshots.

Table 4.1 shows the percentage by which the latencies of BFS and PageRank grow with the addition of each snapshot, computed as slopes of linear regressions that fit the curves in Figure 4.12, expressed as percentages of the immutable CSR baseline. The table focuses on the first 10 delta snapshots, in which the slowdown is most visible. (These numbers should be interpreted as additive, not multiplicative; i.e., adding 10 snapshots in "Perf. + Flat" would make it $10 \times 1.08\%$ slower than the immutable CSR, not $1.08^{10}\%$.)

Performance degrades slowly over time in all cases as the in-memory layout becomes more fragmented, requiring the system to occasionally merge snapshots to bound system performance. The performance of BFS with a more random access pattern degrades faster than that of PageRank. The performance-optimized design and the space-optimized design with explicit linking degrade significantly more slowly than with implicit linking, which requires, on average, a larger number of vertex table accesses for each additional snapshot.

The performance decreases most dramatically with the addition of the first delta snapshot, and

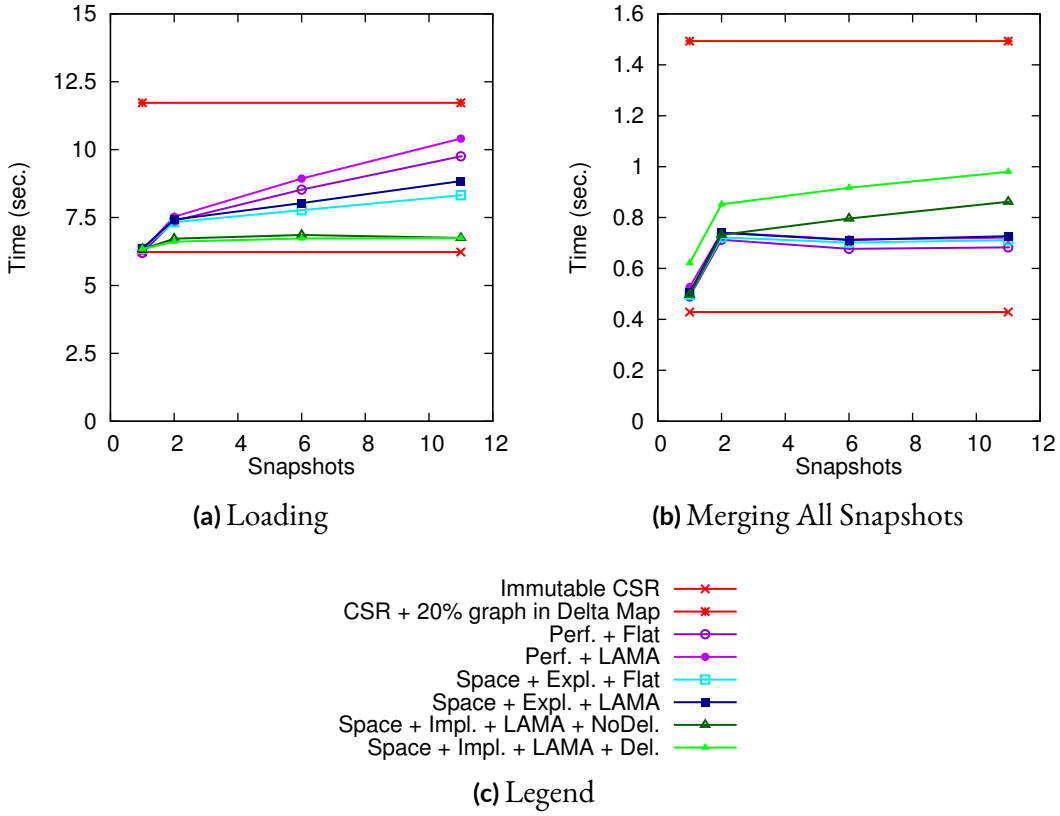
then the degradation is smaller with each subsequent snapshot. Observe that the performance curves in Figure 4.12 relating elapsed time and the number of snapshots exhibit a decreasing growth rate. (Fitting linear regressions as in Table 4.1 is thus an oversimplification, although a useful one.)

Figure 4.13 shows the numbers of vertices that have their adjacency lists in a given number of snapshots for the LiveJournal graph and the various number of snapshots into which we split the dataset. The curves in the plot are probabilities that selecting a vertex uniformly at random will select a vertex with out-edges in a particular number of snapshots. 11.1% of vertices do not have any outgoing edges (not shown in the figure), and 35.8% vertices appear in just one snapshot even if the graph is stored in multiple snapshots. Moving from a one-snapshot graph to a two-snapshot graph, 59.7% of adjacency lists are now spread across two snapshots, resulting in a noticeable (although not significant) performance degradation. Then moving from a two- to six- and eleven-snapshot graphs, those 59.7% adjacency lists now span multiple snapshots, but only a few lists span all snapshots. We distribute the same number of edges across a larger number of snapshots, so the number of adjacency lists that span more snapshots increases less with each additional snapshot.

Loading and Merging Performance

Figure 4.14 shows the cost of loading a graph from a file into various numbers of snapshots, starting with a cold buffer cache, and the cost of merging all snapshots into a single snapshot (i.e., flattening the entire database). Our input files are in X-Stream[63]'s binary edge-list format.

Unsurprisingly, load performance correlates with the amount of data that needs to be written (compare to Figure 4.11). Loading a single snapshot graph takes similar times in all evaluated configurations with the exception of CSR + Delta Map, and the loading time grows with the number of snapshots at rates that correlate with the resulting in-memory sizes. Loading a graph requires the same amount of I/O independent of the number of snapshots into which it is loaded, so the actual variance in the loading performance is due to the amount of data read from and written to mem-



(a) Loading

(b) Merging All Snapshots

(c) Legend

Figure 4.14: Loading and Merging of the LiveJournal Graph. The immutable CSR baseline and the data points for one snapshot in (b) just show the cost of copying the CSR data structure using the same mechanism used in merging, i.e. read each adjacency list, sort it, and write it to the newly constructed CSR.

ory – but the two are correlated. For example, a LAMA vertex table is larger than a flat array, and creating it requires more memory loads due to copy-on-write. The performance-optimized design likewise occupies more memory than the space-optimized design, and it also requires a longer load time due to copying adjacency lists.

The cost of merging correlates with the time it takes to read the entire graph (compare to Figure 4.12), since the size of a single-snapshot graph – and thus the amount of data written – is almost the same for each method (with the exception of CSR + Delta Map), all within 0.1% of each other.

In the case of the performance-optimized design and the space-optimized design with explicit linking, the merging cost does not increase with the number of snapshots, since the amount of data read grows slowly with the number of snapshots.

Merging decreases memory usage (especially for the LAMA vertex table and the performance-optimized design) and improves the performance of analytic applications. Users can compare the expected cost of merging with the expected overhead of computing across multiple snapshots to determine whether to merge. For example, it takes 0.7 seconds to merge all snapshots under the space-optimized design with explicit linking, and BFS takes 0.1 seconds longer when running across 11 snapshots than when running on one snapshot. In this case it makes sense to merge the snapshots into one if the user plans to run at least 7 instances of BFS.

Merging is fast compared to loading, since it is purely an in-memory operation, while loading requires disk access. (Note that this difference will not apply when we work with data that exceeds memory in the next chapter.) In the evaluated scenario we can merge the graph ten times in less time than it takes to load the graph into 11 snapshots! It is thus cheap to frequently merge all snapshots.

We expect users will want to continuously merge the graph in the background while incrementally loading more data, but LLAMA does not currently support concurrent loading and merging. Loading while merging requires us to copy-on-write data structures that are not yet created by the merge, specifically the LAMA vertex table and vertex and edge property tables. In the case of the performance-optimized design, it also requires us to copy not-yet merged adjacency lists, and the space-optimized design with explicit adjacency list linking requires us to know the final positions of merged adjacency lists to write next pointers. We can address these issues by merging specific adjacency lists and parts of the property tables on demand while the rest of the merge proceeds in the background. The final issue is maintaining consistent edge ID maps between different notions of edge IDs, especially out-edge IDs and in-edge IDs, which we plan to do by deferring relevant ID-related operations to the end of the merge.

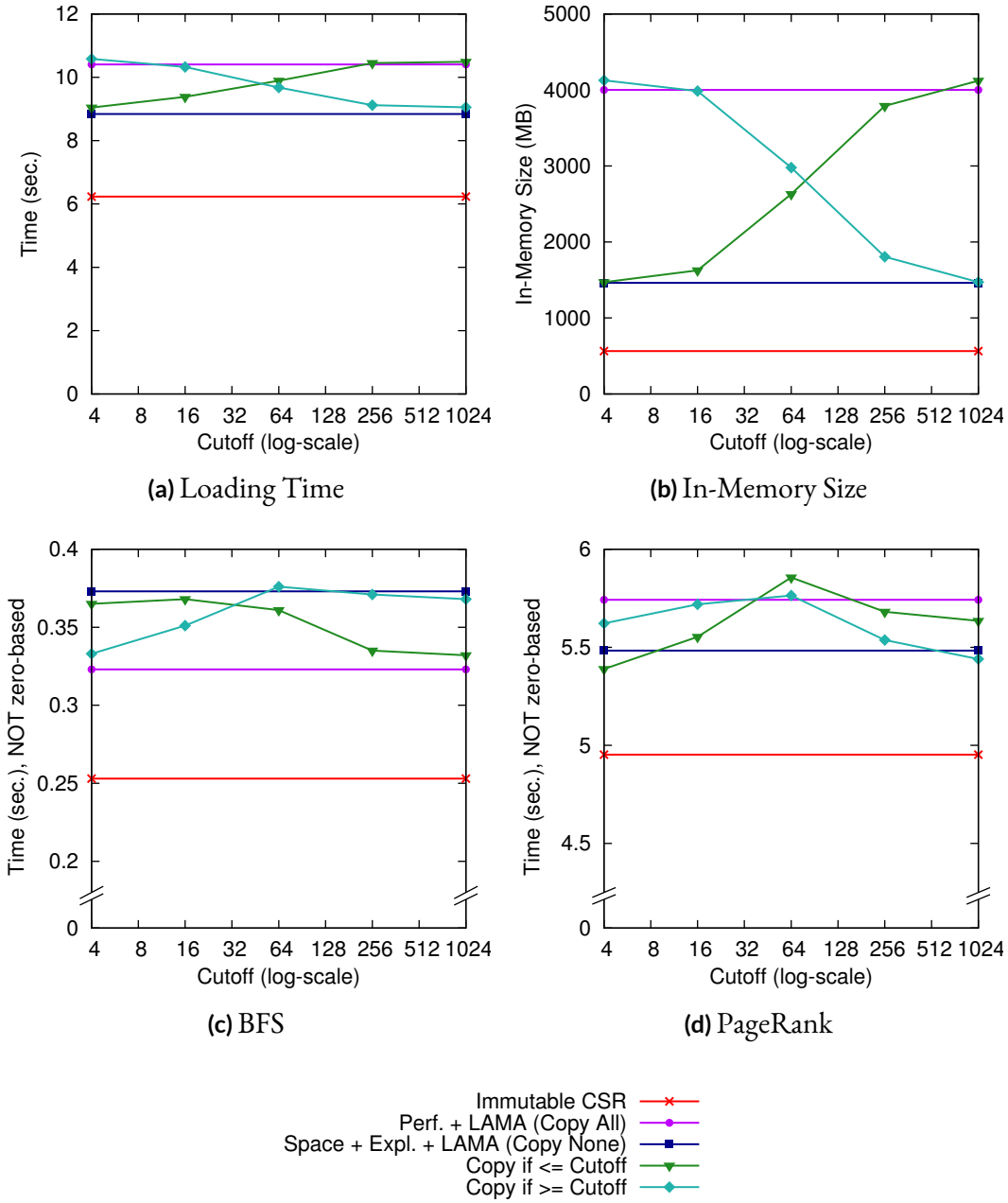
Parameterized (Hybrid) Representation

Figure 4.15 shows the loading time, memory size, and the performance of LLAMA under two hybrid design policies plus the baselines for the LiveJournal graph [3] loaded into 11 snapshots (in the same way as described in Section 4.9.1). The baselines and hybrid policies are:

1. Immutable CSR – storing the entire graph in a single snapshot using the traditional CSR design (the same as above)
2. Perf. + LAMA – the performance-optimized design with the multiversioned LAMA vertex table, which copies all updated adjacency lists
3. Space + Expl. + LAMA – the space-optimized design with the multiversioned LAMA vertex table and explicit adjacency list fragment linking, which does not copy updated adjacency lists
4. Copy if <= Cutoff – copy adjacency lists smaller than or equal to the given cutoff (the X axis), using the LAMA vertex table
5. Copy if >= Cutoff – copy adjacency lists larger than or equal to the given cutoff (the X axis), also using the LAMA vertex table

The "Copy if <= Cutoff" and "Copy if >= Cutoff" curves provide gradual transitions in loading time, memory size, and performance between "copy all" (Perf.) and "copy none" (Space) approaches, depending on how many adjacency lists are copied or stored as deltas.

The transitions in the in-memory size are especially smooth and symmetric, since these policies produce continuous functions in the number of copied edges. The LiveJournal dataset used in this evaluation follows the preferential attachment model, so there are a large number of small-degree vertices and a few large-degree vertices. The "Copy if <= Cutoff" policy thus copies a large number of small-degree vertices, and "Copy if >= Cutoff" copies fewer large-degree vertices.



(a) Loading Time

(b) In-Memory Size

(c) BFS

(d) PageRank

(e) Legend

Figure 4.15: Various Parameterized (Hybrid) Representation Policies on the LiveJournal graph stored in 11 snapshots on the Commodity platform, plus the Immutable CSR baseline.

If M is the size of the space-optimized design and $n_{s,d}$ is the number of updated vertices with degree d in snapshot s, the memory cost under the "Copy if <= Cutoff" policy for cutoff c is, without loss of generality, simply $M + 8 \sum_s \sum_{d \leq c} n_{s,d} d$ plus any copying due to deletions, which was zero in this experiment (recall that an edge occupies 8 bytes). The number of next pointers is identical to that of the space-optimized design irrespective of the policy and the cutoff, because the number of updated adjacency lists remains the same. Observe that copying all adjacency lists results in higher memory usage than that of the performance-optimized design because hybrid policies have next pointers while the performance-optimized design does not.

The transitions in the loading times in the figure are smooth and symmetric just like the memory sizes, since the two are correlated.

The transitions in analytic performance are less well-behaved, but they follow the same general trends. In the case of BFS, which has a more random-access pattern, the elapsed time increases with the number of adjacency lists that span multiple snapshots. Spanning fewer snapshots improves cache behavior especially when accessing data in a less predictable order, while spanning more snapshots results in accessing data from a larger number of distinct locations in memory that are difficult to predict by the prefetcher.

The case of PageRank is particularly interesting, since the space-optimized design performs better than the performance-optimized design, and the transition between the two is not well-behaved. The performance is worst at cutoff 64 for both "Copy if <= Cutoff" and "Copy if >= Cutoff," but it nonetheless follows the same pattern as in the other plots for most other cutoffs. PageRank accesses edges in source vertex ID order, which results in an efficient parallel sequential scan over all edge tables as mentioned in previous sections. Copying adjacency lists disrupts this pattern and worsens the cache behavior. This acts in opposition to the CPU prefetcher improving with the length of sequential reads, which improves performance due to reads of large continuous adjacency lists.

Most hybrid design policies do not provide performance that is consistently better than the space-optimized design with the exception of "Copy if <= 4" and "Copy if >= 1024", both of which produce 10.7–10.9% improvement in BFS and 0.8–1.7% improvement in PageRank. The first result follows a straightforward intuition that it makes sense to copy small adjacency lists, and it does not significantly disrupt the parallel sequential scan over the edge tables.

The second result shows that if the adjacency list is long enough, the savings of not having to jump around when reading it outweigh the benefits of the more cache-optimal access pattern of algorithms like PageRank in the space-optimized design. This makes sense in graphs that exhibit preferential attachment, in which case a large-degree vertex is more likely to get new edges – and thus its adjacency list will be spread over a large number of snapshots. The policy "Copy if >= 1024" thus applies specifically to this data set; a more generally applicable policy is "Copy a few largest adjacency lists." There are only 60 adjacency lists with 1024 or more elements in this dataset, and they account for 0.15% of edges, so copying them does not significantly increase the in-memory size.

We thus expect that a combined policy "Copy if <= 4 $\vee$ Copy if it is one of the few largest adjacency lists" would be closer to optimal.

4.9.3 Evaluation on Various R-MAT Graphs

In the next phase of our evaluation, we focus on the two most performant non-parametric designs – the performance-optimized design and the space-optimized design with explicit adjacency list linking – evaluating them on synthetic R-MAT[10] graphs of varying size and skew. We answer the following questions: How does in-memory size change with the skew? How does the performance of PageRank and BFS change? And finally, under which conditions does each of these two representations do better than the other? We use LAMA vertex tables and load the graph into 11 snapshots unless stated otherwise.

We fix the number of vertices to $2^{22} \approx 4.19 \times 10^6$, as this is the closest power of 2 to the LiveJournal graph from the previous part of the evaluation, and vary the size of the graph by varying the average vertex degree from 4 to 64. We define skew for the purposes of this evaluation as a number between 0 and 1, where 0 produces R-MAT probabilities describing an Erdös-Rényi random graph [21], and 1 produces R-MAT probabilities from the Graph500 specification [27] resulting in a skewed graph with preferential attachment. The numbers in the middle produce graphs with intermediate skew. We produce R-MAT weights with skew σ using the following linear combination, transitioning from Erdös-Rényi to Graph500 weights as σ goes from 0 to 1:

$$\begin{bmatrix} a \\ b \\ c \\ d \end{bmatrix} = (1-\sigma) \begin{bmatrix} 0.25 \\ 0.25 \\ 0.25 \\ 0.25 \end{bmatrix} + \sigma \begin{bmatrix} 0.57 \\ 0.19 \\ 0.19 \\ 0.05 \end{bmatrix}$$

We run each benchmark on five different graphs with the same R-MAT parameters, at least five times on each of the graphs.

In-Memory Size

Figure 4.16 shows the in-memory sizes of LLAMA with performance- and space-optimized designs with LAMA vertex table on R-MAT graphs of various skews and sizes (expressed as the average degree) loaded into 11 snapshots.

The memory usage of performance-optimized LLAMA increases with the skew due to preferential attachment: Nodes with a large degree are more likely to get new edges more often, which means that LLAMA is more likely to copy large adjacency lists each time it creates a new snapshot.

In comparison, the memory usage in the space-optimized design with explicit linking decreases slightly with the skew also as a consequence of preferential attachment. It follows from this graph

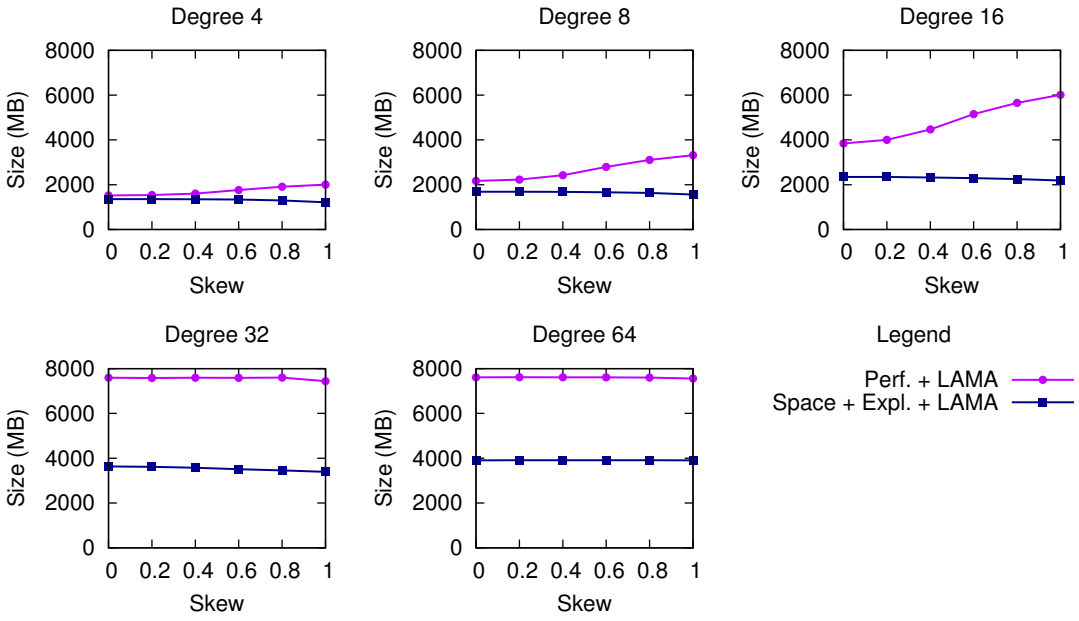


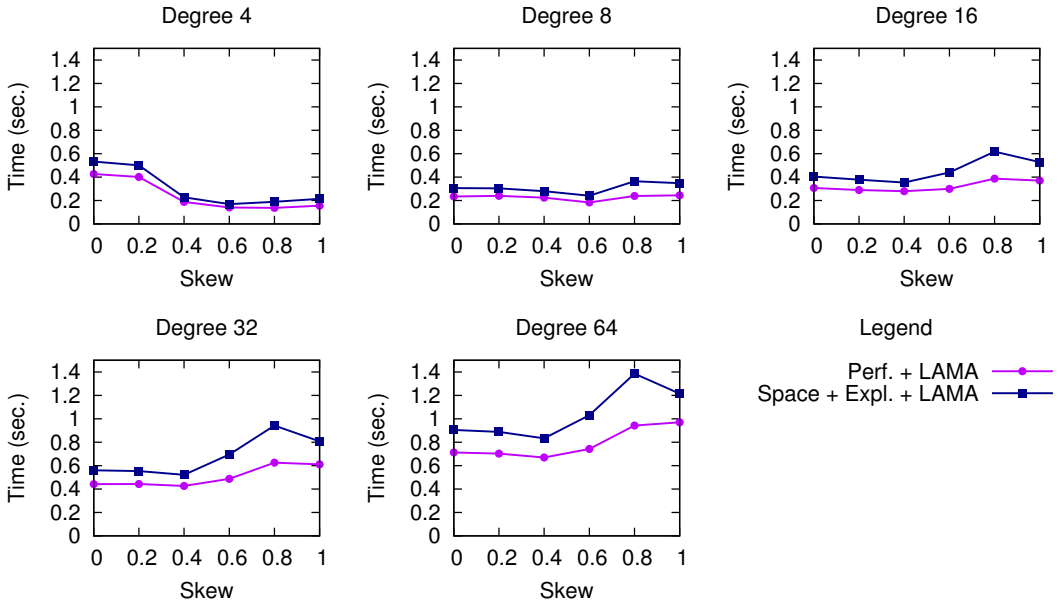
Figure 4.16: In-Memory Size of R-MAT Graphs of various size and skew loaded into 11 snapshots for performance- and space-optimized designs with LAMA vertex table.

update model that we are more likely to add edges to vertices that already have large degrees, which results in updating fewer adjacency lists. Having fewer adjacency list fragments results in writing fewer next pointers and updating fewer vertex records in the vertex table, which in turn translates into copying fewer LAMA array pages.

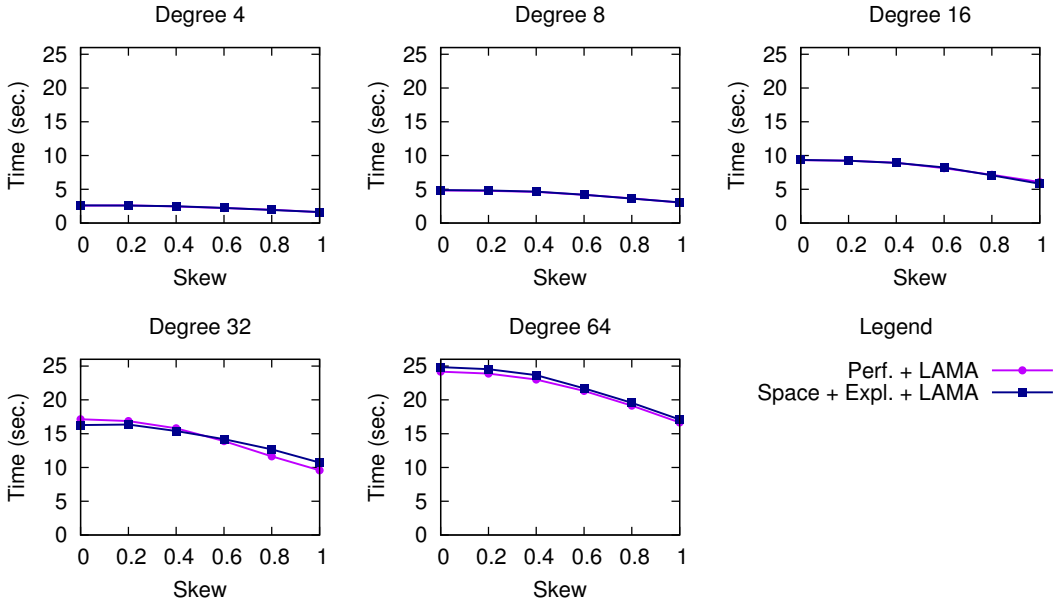
Graph Analytics Performance

Figure 4.17 shows the performance of BFS and PageRank on R-MAT graph with various skews and sizes loaded into 11 snapshots.

The performance-optimized design outperforms the space-optimized design in all cases on BFS, just as expected from our evaluation on the LiveJournal graph (Section 4.9.2), because it exhibits a



(a) BFS Performance



(b) PageRank Performance

Figure 4.17: Performance of BFS and PageRank on R-MAT Graphs of various size and skew loaded into 11 snapshots for performance- and space-optimized designs with LAMA vertex table.

more random access pattern. The average runtime increases with the skew in all cases but the smallest graphs, but this is just an artifact of our methodology of starting the traversal from one of the most well-connected nodes in the graph – and graphs with higher skews have larger longest adjacency lists. The only exceptions are graphs with the smallest tested average vertex degree (4), where the average elapsed time decreases as the skew increases. Skewed graphs with such small degrees have a large proportion of vertices with no out-going edges, thus causing BFS to terminate earlier.

The performance of PageRank is heavily influenced by the average lengths of adjacency lists. Graphs with a higher skew have a larger proportion of vertices with zero out-degree and thus larger adjacency list lengths for those vertices that do have edges, which can be more easily taken advantage of by the prefetcher, especially in the case of graphs with higher edge density.

The differences between the runtimes of performance- and space-optimized designs for sparse graphs (with smaller average vertex degrees – 4, 8, and 16) are not statistically significant for most data points, with p-values > 0.1 computed using paired t-tests. The few data points with smaller p-values all show that the performance-optimized design outperforms the space-optimized design by 0.4–2.7% for some sparse graphs with low skew.

The trends in graphs with larger vertex degrees (32 and 64) are all statistically significant with the exception of a single data point (PageRank performance for degree 32, skew 0.2, in which one of the five runs produced an outlier result). The space-optimized design outperforms the performance-optimized design by up to 5.1% on skews 0–0.4 when the average degree is 32 for skew 0, and the performance-optimized design then outperforms the other for the remaining skews by up to 12% for skew 1. The performance-optimized design outperforms the space-optimized design by 1.8–2.8% for all skews and average vertex degree 64.

Skewed graphs have, on average, longer adjacency lists, which results in better cache behavior especially in the case of the performance-optimized design, especially in graphs with higher edge

density. In contrast, long adjacency lists in the space-optimized design are more likely to be spread over multiple snapshots due to the preferential-attachment model, as a high-degree node is more likely to accumulate more new edges more frequently. This breaks up long sequential reads into a large number of smaller reads, which adversely affects the prefetcher.

The space-optimized design reads more data than the performance-optimized design due to next pointers, but this by itself does not degrade its performance, as the next pointer reads are already in the cache after processing the preceding adjacency list fragment. In contrast, while the performance-optimized design does not read any next pointers, it does not necessarily transfer less data from the memory to the cache: The memory transfers occur at the granularity of cache lines, but the performance-optimized design does not necessarily use all data from the cache lines, ignoring edge records that belong to old versions of adjacency lists. The number of such incompletely used cache lines increases with the number of snapshots – as the proportion of outdated adjacency lists increases.

4.10 Summary

The main disadvantage of CSR is immutability. Existing systems address immutability using two basic approaches. Some add a write-optimized delta map directly on top of a CSR [43], while others treat the CSR as a log, appending a modified adjacency list to the end of an edge table [60,7]. The first approach requires rebuilding the entire CSR to merge changes from the delta map, because delta maps are slow relative to CSR for analytics workloads (see Section 4.9.2). The second approach is not much better: even a moderate number of updates causes the edge table to grow quickly and leaves holes in the middle. This also destroys the sequentiality of the adjacency lists. Consequently, the entire CSR must occasionally be rebuilt.

LLAMA chooses a different approach that leverages multi-snapshot read-optimized storage. We tried three different methods for storing adjacency lists in snapshots: 1) write the entire adjacency list, 2) write new edges only into a new snapshot, or 3) choose one or the other based on a simple policy. They all provide great performance, adding at most a modest overhead relative to immutable CSR. The graph grows rapidly when we always write entire updated adjacency lists or even when writing them frequently under a hybrid policy. We get the best overall performance and good memory characteristics at the same time when always writing deltas into new snapshots, or when writing deltas for all but the smallest and largest adjacency lists under a simple hybrid policy.

5

Persistent, Larger-than-Memory CSR

After exploring options to make CSR mutable, this chapter focuses on enabling LLAMA to analyze graphs that exceed memory.

The main advantages of CSR are its compact size and excellent cache behavior, both due to its dense packing of both edges and vertices. Sorting the edges first by their tail vertex IDs and then by their head IDs produces sequential access for many graph algorithms, such as PageRank, some implementations of BFS, and single-source shortest path (SSSP). CSR was originally designed for

in-memory computation, but we expected CSR to also perform well in out-of-memory scenarios for these same reasons.

While many systems operate on edge-lists [44,63], a CSR-like approach uses about half the space. In an edge-list representation, each edge requires both head and tail vertex IDs; CSR requires only a head vertex ID – the tail is implied. A smaller data footprint means that more of the graph fits in memory, requiring fewer I/Os.

Our primary goal is to provide analytic performance comparable to dedicated in-memory solutions when the graph fits entirely in memory and competitive with dedicated out-of-core solutions for graphs larger than memory. Our secondary goal is to ensure that the loading phase occurs only once instead of loading the data for each analysis (such as X-Stream [63]) or having to reload it for different algorithms (such as GraphChi [43]). We just allow creating other materialized views of the data with different sort orders, such as in-edges or post-order edges.

We solve the first challenge by using an on-disk format that is nearly identical to the in-memory format, placing the graph files on an SSD, and memory mapping them into the address space of the analysis engine. Addressing the latter challenge is straightforward: We just make all relevant data persistent.

LLAMA does not currently provide strong transactional guarantees other than ensuring that new data is durable as soon as snapshot creation completes. Turning LLAMA into a graph database with full ACID guarantees is left for future work.

The rest of the chapter is organized as follows: Section 5.1 discusses and justifies our approach to persistence, and Section 5.2 compares various approaches to CSR mutability from the previous chapter on selected out-of-core workloads. The next chapter then presents a comprehensive evaluation of LLAMA.

5.1 Low Overhead Persistence via Memory Mapping

Many graph algorithms have tight inner loops, which means that adding even a few instructions for buffer management can prove fairly expensive. This is unacceptable given our goal of providing zero overhead persistence when the graph fits in memory.

We experimented with two alternatives before settling on memory mapping: reference counting and hazard pointers[50], but both proved to be prohibitively expensive. Relative to the `mmap`-based implementation, the overhead of page latching ranged from nearly 50% for PageRank to nearly 200% for triangle counting. This result is not entirely surprising. Moderately high overheads due to page latching have already been observed in relational databases[31], but the results for graph analytics are even worse. For either approach, opening an edge iterator for a vertex requires:

1. Checking whether the containing page is physically present in main memory and loading it if it is not,

2. Incrementing the corresponding reference count or acquiring a hazard pointer to pin the page in memory, and

3. Performing bookkeeping necessary to implement the desired eviction policy, such as setting a bit for CLOCK or moving the page to the end of a queue for LRU.

Adding this functionality significantly increases the amount of work done by the iterator's constructor, making it quite expensive. We also need to decrement the reference count or release a hazard pointer by the corresponding destructor.

Lowering CPU cost is necessary even for dedicated out-of-core systems: We found that both GraphChi[43] and X-Stream[63] were CPU-bound even when running out-of-core (see Section 6.2).

TurboGraph[30] mitigates the increased CPU cost by pinning large pages instead of individual adjacency lists, which amortizes the cost of buffer management over a larger number of adjacency list

accesses. This comes at a minor inconvenience that the graph algorithms must either conform to a particular vertex- or edge-program framework or they must manage buffers explicitly.

LLAMA instead uses an alternative approach based on memory mapping, which was demonstrated to work well both in the areas of graph analytics[44] and key-value stores[13]. `mmap` produces effectively zero overhead once data are memory resident.

This approach allows programmers to more easily transfer their implementations from in-memory solutions, such as Green-Marl[32], provided that they exhibit enough locality, so that once the system brings an on-disk page to memory, the graph algorithm uses as much data on the page as possible before proceeding. LLAMA's and TurboGraph's approaches are fundamentally similar, as they both require locality, except that while TurboGraph enforces it through appropriate scheduling of vertex programs, LLAMA depends on the algorithm to order its accesses appropriately.

LLAMA furthermore requires all of its graph files to be stored on an SSD instead of a spinning disk to take advantage of SSD's efficient parallel access. It is possible, although not recommended, to run LLAMA in a single-threaded mode on a graph stored on a hard-drive.

5.1.1 On-Disk Format

The on-disk representation is nearly identical to the in-memory representation, both because it is a prerequisite for memory mapping and because it avoids costly transformations between the two formats. The latter is known to be a significant bottleneck both in RDBMS[17,31] and key-value stores[13], and we make a similar observation in the area of graph analytics (Section 6.2).

On persistent storage, we default to storing 16 consecutive snapshots in one file – i.e., we store the first 16 snapshots in the first file, and then we start a second file in response to creating the 17th snapshot. Limiting the number of snapshots per file lets us reclaim space easily from deleted snapshots while keeping the number of files manageable.

Each file has a header containing metadata for each snapshot, which includes the offset of the

corresponding indirection table and range of data pages containing the snapshot. We store the indirection table as an array of structures, each containing two fields: a *snapshot number*, a 4-byte integer identifying the snapshot (and therefore file) that owns the page, and an *offset*, an 8-byte integer indicating the offset within the appropriate file in which the page resides. If the indirection table is identical to that of one of the previous snapshots (i.e., there are no modifications between the two snapshots), we store a reference to the previous snapshot rather than storing a duplicate indirection table. We store a snapshot's data pages, both vertex and edge tables, contiguously in the file.

We do not reference-count pages, because we currently allow deleting only the most recent snapshot; we can do this because deleting a snapshot does not delete any pages used by an older snapshot.

We load data into memory in ascending order of snapshot ID. For each snapshot, we `mmap` the appropriate data pages from the file and build the in-memory indirection array containing pointers to the corresponding mapped regions. Accessing an element using its index translates into just two pointer dereferences.

5.1.2 Memory Mapping and Buffer Management

In theory, the disadvantage of using `mmap` is that we give up control over which data pages are evicted from memory. In practice, we find that other systems take one of only three approaches to this issue. One is to have the graph engine always access vertices in order by appropriately scheduling vertex and edge programs (e.g., GraphChi [43], X-Stream [63]), in which case the operating system does a good job selecting pages to prefetch and evict. The second is to simply ignore out-of-memory execution (e.g., GraphLab [45], Pregel [47]), and the third is to use `mmap` as we do, relying on the implementation of the graph algorithm to order its accesses to exhibit enough locality so that the operating system does a good job prefetching and evicting pages (e.g., Lin et al. [44]). Should the need arise for more specific eviction control, experimentation suggests that we can use `madvise` to make `mmap` behave as we want.

5.1.3 Parallel Sequential Scans over Snapshots

Many graph algorithms are iterative in nature, processing one vertex at the time following access patterns similar to:

```
for all Vertex v ∈ G do
    for all Edge e ∈ OutEdges(v) do
        Do something with the other endpoint of e
    end for
end for
```

If the graph fits in a single snapshot, this trivially translates into a sequential access pattern on the edge table with occasional lookups in the vertex table, because CSR stores adjacency lists consecutively in source vertex ID order.

If the graph is stored in $n > 1$ snapshots, this access pattern decomposes into n parallel sequential scans over the corresponding n edge tables, plus some less sequential vertex table lookups, as explained in Section 4.9.2. This access pattern can be easily handled by the prefetcher due to its semi-sequential nature. It is also a good fit for SSDs, which can efficiently handle multiple reads in parallel.

5.1.4 BFS Implementation

LLAMA contains two implementations of BFS:

- A standard queue-based implementation intended for in-memory execution, and:
- An implementation inspired by iterative deepening[41] intended for on-disk execution on graphs with small diameter (such as social network graphs).

The algorithm in Figure 5.1 summarizes the second approach. It starts by setting the distances of all vertices except the root vertex to infinity, and then it iterates through the graph computing depths using an unweighted single-source shortest path computation. The user program can then visit the

```
procedure BFS-ITERATIVE(G, r)

    // Initialize the distance vector D; r is the root vertex for the BFS traversal
    for all Vertex v ∈ G do
        D[v] ← ∞
    end for
    D[r] ← 0

    repeat
        done ← true
        for all Vertex v ∈ G do
            if D[v] ≠ ∞ then
                for all Edge e ∈ OutEdges(v) do
                    w ← the other endpoint of e
                    if D[v] + 1 < D[w] then
                        D[w] ← D[v] + 1
                        done ← false
                    end if
                end for
            end if
        end for
    until done
end procedure
```

Figure 5.1: BFS Implementation Inspired by Iterative Deepening, which is more optimal for on-disk execution on small-diameter graphs.

vertices in depth order, one level at the time. The algorithm iterates through the graph multiple times, so it is suitable primarily for small-diameter graphs, such as social network graphs. Each phase accesses vertices in ID order, which makes it more efficient for on-disk execution. We use this variant of BFS for on-disk execution and the queue-based variant for in-memory execution unless stated otherwise.

5.2 Quantitative Comparison Revisited

We now revisit the quantitative comparison section from the previous chapter (Section 4.9) and run BFS and PageRank on the same Commodity machine – but this time with out-of-core data. We equip our commodity machine (dual-core Intel Core i3 + HT, 3.0 GHz, and 8 GB RAM) with a 256 GB Samsung 840 Pro SSD and conduct all experiments on the Twitter graph [42] with 41.7 million nodes and 1.47 billion edges. The graph occupies 11.9 GB when loaded into a single-snapshot LLAMA, which is almost 50% larger than memory size. We split the graph into 11 snapshots the same way as in Section 4.9.1 (load a random 80% of the edges into the first snapshot and evenly distribute the rest among the remaining snapshots), since we have already established that the performance on a single-snapshot LLAMA is similar for most configurations in Section 4.9.

In the first part of the evaluation, we focus on the performance-optimized design and the space-optimized designs with explicit and implicit linking, both using a multiversioned LAMA vertex table. We consider the space-optimized design with implicit linking without the deletion vector to focus on comparing the effects of the different I/O patterns on performance without being additionally slowed down by the deletion vector. We revisit the parameterized (hybrid) designs in the following section. (LLAMA currently requires a LAMA vertex table for persistence; mutating an existing vertex table is left for future work.)

The goal of this quantitative comparison is to answer the following two questions: 1) How much does the graph grow under different configurations? and: 2) Which configuration results in the best out-of-core performance? We present a more comprehensive evaluation of LLAMA in the next chapter.

We run each experiment five times and report the averages. The standard deviations were less than 1% for PageRank and less than 3% for BFS unless stated otherwise.

Configuration	Load Time (s)	Size (GB)
Performance-Optimized + LAMA	4864.24	99.09
Space-Optimized + Expl. + LAMA	362.11	22.26
Space-Optimized + Impl. + LAMA	187.87	21.01

Table 5.1: Loading Time and Database Size on the Commodity Machine for the Twitter dataset loaded into 11 snapshots.

	PageRank		BFS	
Configuration	Time (s)	I/O (GB)	Time (s)	I/O (GB)
Performance-Optimized + LAMA	4027.47	698.20	2008.13	94.14
Space-Optimized + Expl. + LAMA	672.26	131.33	272.55	45.91
Space-Optimized + Impl. + LAMA	1101.66	209.97	515.56	79.71

Table 5.2: Out-of-Core Commodity Machine Performance for the Twitter dataset loaded into 11 snapshots.

5.2.1 Space- and Performance-Optimized Designs

Table 5.1 shows the time it takes to load the 11-snapshot version of the Twitter graph [42] and the resulting database size for both the performance- and space-optimized designs. Both designs occupy 11.9 GB when the entire graph is loaded into a single snapshot, but the table shows that the database size grows much more dramatically with the performance-optimized design because it always copies adjacency lists in response to updates. The growth of the space-optimized design with implicit linking from 11.9 to 21.0 GB (corresponding to 1 and 11 snapshots, respectively) is due to copy-on-write of LAMA vertex table pages. The space-optimized design with explicit linking is 1.25 GB larger due to the next pointers.

The loading time is significantly larger for the performance-optimized LLAMA, not only because it writes five-times more data than the space-optimized design, but also because it needs to read previous versions of adjacency lists when updating them. The space-optimized LLAMA with explicit

linking needs to examine only the previous versions of the vertex table to write proper next pointers, but it does not need to read any of the old edge tables. The loader needs to access only the most recent version of each vertex table page, which is small enough to remain in memory throughout most of the loading phase. Finally, the space-optimized LLAMA with implicit linking does not even need to access old vertex tables during load, which makes its loader the fastest of all.

Table 5.2 shows the performance of ten iterations of PageRank[57] and BFS from one of the most well-connected nodes in the database, both on an 11-snapshot Twitter graph. It is easy to see that both space-optimized designs perform significantly better on both workloads, because they transfer less data between the SSD and main memory. The iterative nature of these algorithms naturally decomposes into parallel sequential scans over the 11 edge tables, which is perhaps the next most efficient access pattern for SSDs after a purely sequential scan. The space-optimized design with explicit linking is faster than the variant with implicit linking, because it examines the vertex table only once per opening an iterator over the out-edges of a vertex.

The space-optimized designs process slightly more data than the performance-optimized design due to the next pointers, but this does not incur a noticeable performance penalty, because the next pointers are placed immediately after the corresponding adjacency list fragments. In contrast, the performance-optimized design does not access a large fraction of the bytes on the disk pages it reads, since the unneeded old versions of adjacency lists are interspersed between the most recent versions of adjacency lists. The performance-optimized design therefore performs more I/O, because it needs to read a larger number of pages to use the useful bytes and also because it can hold fewer useful bytes in a buffer cache of the same size.

The space-optimized design with explicit linking is strongly preferable over both the variant with implicit linking and the performance-optimized design in out-of-core scenarios.

Configuration	BFS-Queue Time (s)	I/O (GB)
Performance-Optimized + LAMA	3582.56	91.56
Space-Optimized + Expl. + LAMA	2051.64	59.94
Space-Optimized + Impl. + LAMA	9168.06	205.53

Table 5.3: Out-of-Core Performance of the In-Memory BFS Algorithm on the commodity machine for the Twitter dataset loaded into 11 snapshots.

5.2.2 Out-of-Core Performance of the In-Memory BFS Construct

Table 5.3 shows the execution times and the corresponding amounts of I/O of the standard queue-based BFS algorithm optimized for in-memory execution but running out of core. The standard deviations for the runtimes are less than 3%, but the standard deviations for the I/O are as large as 10%, which is probably due to differences in thread interleaving.

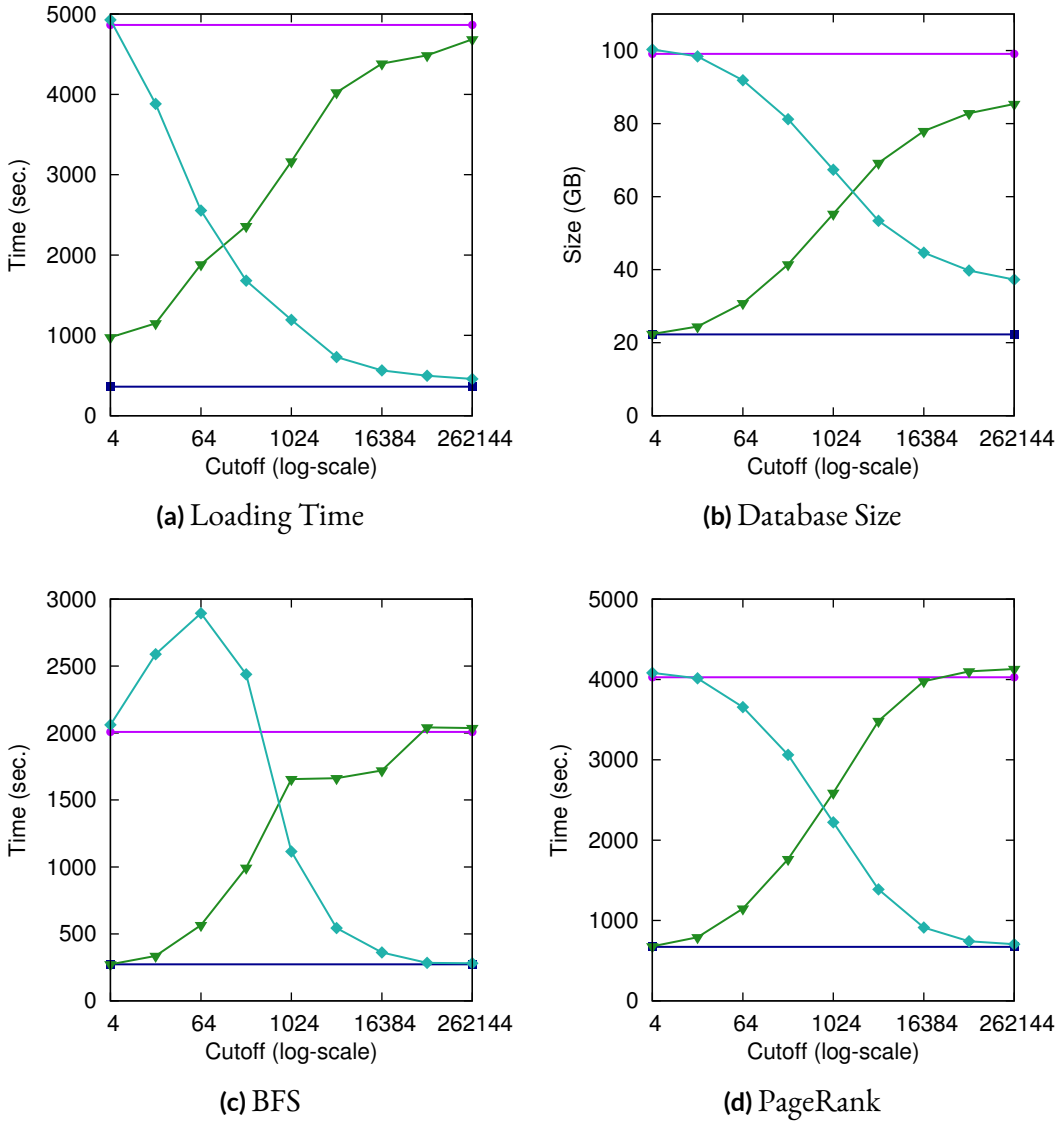
This implementation of BFS accesses vertices in a more random order than the algorithm designed for out-of-core execution, which makes it less efficient, and its performance is (unsurprisingly) worse.

5.2.3 Hybrid Designs

Figure 5.2 compares the loading times, database sizes, and performance of the two hybrid designs to the performance-optimized design and space-optimized design with explicit linking on the Twitter graph loaded into 11 snapshots.

As we saw in Section 4.9, the two hybrid designs produce performance that transitions between the two nonparametric designs. BFS performance for "Copy if >= Cutoff" is likewise not well-behaved, having the worst performance also at cutoff 64. But unlike the in-memory case examined in the previous chapter, no policy improves upon the space-optimized design with explicit linking.

Considering that hybrid policies help only marginally in the in-memory case and that they do not work as well out-of-core, we conclude that the space-optimized design with explicit linking is most appropriate to support both in-memory and out-of-core workloads.



(a) Loading Time

(b) Database Size

(c) BFS

(d) PageRank

(e) Legend

Figure 5.2: Various Parameterized (Hybrid) Representation Policies on the Twitter Graph stored in 11 snapshots on the Commodity platform.

6

Evaluation of LLAMA

Our evaluation has four parts, analyzing LLAMA's: 1) performance relative to competing engines, 2) multiversion support, 3) scalability by number of cores, and 4) scalability with respect to graph size and composition. We use three main benchmarks: ten iterations of PageRank[57] (PR), breadth-first search (BFS), and triangle counting (TC). The goal of the evaluation is to evaluate different parts of a system that starts by loading an initial graph, then keeps receiving updates, running analytics, and merging periodically – all in isolation to better understand each one of these parts.

First, we compare LLAMA's performance to two in-memory systems, GreenMarl[32] and GraphLab version 2.2 (PowerGraph)[24], and two out-of-core systems, GraphChi[43] and X-Stream[63]. We chose GreenMarl because it exposes a flexible programming model as does LLAMA. GraphLab and GraphChi use a vertex-centric model, and X-Stream provides an edge-centric model. We also ran some of our workloads on Neo4j[52], but we do not include the results here. Neo4j handles graphs larger than memory, and is one of the most widely used graph databases, but it targets a different point in the graph processing design space. Even on the smallest datasets, its performance was orders of magnitude worse than the other systems.

Second, we evaluate LLAMA's multiversion support, by examining incremental data ingest performance and the overhead of running analytics workloads across multiple snapshots.

Third, we evaluate scalability in the number of cores.

Fourth, we analyze LLAMA's scalability on synthetically generated R-MAT graphs, varying the number of vertices and the average vertex degree. R-MAT is a recursive matrix model for generating realistic graphs[10].

We run each experiment three to five times.

6.1 Setup

We run our evaluation on two platforms, each representing a design point we target:

- *BigMem*, a large shared-memory multiprocessor on which data all fit in memory: 64-core AMD Opteron 6376, 2.3 GHz with 256 GB RAM.
- *Commodity*, a small personal machine on which we may need to execute out-of-core: Intel Core i3, 3.0 GHz with two HT cores (4 hardware threads total), 8 GB RAM, and a 256 GB Samsung 840 Pro SSD.

We compute on the following real-world graphs in addition to R-MAT:

System	Load (s)	PageRank (s)	BFS (s)	TC (s)
LLAMA	7.74	6.48	0.35	9.97
GraphLab	48.80	24.30	6.60	21.02
GreenMarl	6.75	5.30	0.27	9.79
GraphChi	26.00	39.54	38.84	45.81
X-Stream	–	12.74	5.65	–

(a) LiveJournal (in memory, 4 cores)

System	Load (s)	PageRank (s)	BFS (s)	TC (s)
LLAMA	311.1	607.6	233.8	2875.0
GraphLab	–	–	–	–
GreenMarl	–	–	–	–
GraphChi	760.5	1260.9	1334.9	3975.2
X-Stream	–	1942.9	1124.7	–

(b) Twitter (larger than memory, 4 cores)

Table 6.1: Commodity Machine Performance for LiveJournal and Twitter datasets, elapsed times shown in seconds. Note that neither GreenMarl nor GraphLab can run when the graph exceeds main memory. X-Stream does not implement an exact triangle counting algorithm, only an approximation, so we do not include its results.

- *LiveJournal*[3]: 4.8 million nodes, 69.0 million edges
- *Twitter*[42]: 41.7 million nodes, 1.47 billion edges

In LLAMA, LiveJournal uses 601 MB. Twitter is a 11.9 GB file; this is almost 50% larger than the memory of the commodity machine.

For GraphLab tests in memory, we loaded the entire graph into a single partition to avoid inter-partition communication overhead. Note that while GraphLab is a distributed system, it has good support for shared-memory computation.

System	Load (s)	PageRank (s)	BFS (s)	TC (s)
LLAMA	12.68	2.42	0.25	8.58
GraphLab	120.26	15.63	4.33	8.51
GreenMarl	10.56	2.11	0.23	5.24
GraphChi	37.03	14.43	18.21	48.73
X-Stream	–	263.00	258.18	–

(a) LiveJournal (in memory, 64 cores)

System	Load (s)	PageRank (s)	BFS (s)	TC (s)
LLAMA	260.0	41.9	7.9	1033.8
GraphLab	2909.0	205.7	52.4	1132.5
GreenMarl	226.4	40.7	6.7	1031.4
GraphChi	1234.8	339.4	545.9	1441.0
X-Stream	–	395.5	348.1	–

(b) Twitter (in memory, 64 cores)

Table 6.2: BigMem Performance for LiveJournal and Twitter datasets, elapsed times shown in seconds.

6.2 Cross-System Comparison

Tables 6.1 and 6.2 compare the performance of LLAMA to the four other systems on the commodity and BigMem machines, respectively. We store the entire graph in a single LLAMA snapshot. (We evaluate LLAMA with multiple snapshots in Section 6.3.) The LiveJournal dataset fits in memory in both cases, while Twitter fits in the memory only on the BigMem machine.

Most standard deviations on the Twitter graph are less than 5% with the exception of X-Stream's, some of which are as high as 11%. The standard deviations on the LiveJournal graph are less than 15%. Comparisons between LLAMA and other systems are statistically significant (despite the high standard deviations) unless stated otherwise.

We did not run triangle counting on X-Stream; X-Stream only implements an approximate algorithm, and did not produce a close-enough result even when run as long as exact triangle counting on the other systems. This is an artifact of the fact that X-Stream is designed to work directly on its input data files without preprocessing them first. X-Stream also has no separate loading phase, so its performance on each benchmark could reasonably be compared to the sum of the load time and the run time on the other systems. However, in practice load times are amortized if a system runs multiple workloads.

LLAMA outperforms all systems except GreenMarl both in and out of memory, and GraphLab matches it only for triangle counting on BigMem.

6.2.1 GreenMarl

We use nearly identical implementations of PageRank, BFS, and triangle counting on both LLAMA and GreenMarl. GreenMarl's runtime likewise uses OpenMP and CSR, although its CSR is in-memory and immutable. Therefore, the differences between LLAMA's and GreenMarl's runtimes approximate the overhead of supporting persistence and multiple snapshots. This difference is heavily workload-dependent and tends to be smaller on large datasets as illustrated by the Twitter graph results. LLAMA performs less than 3% slower on PageRank and 18% slower on BFS on the Twitter graph. The difference between the triangle counting performances of LLAMA and GreenMarl is not statistically significant. The three major differences relative to Green-Marl's CSR that account for this disparity are: LLAMA accesses an element in the vertex table through an extra level of indirection, LLAMA's vertex table is larger due to including the adjacency list fragment length, and the iterator contains extra code for processing continuation records.

LLAMA's "overhead" on small graphs with short runtimes on the order of seconds (as shown by the LiveJournal results) ranges from 3% to 30% for all workloads with the exception of triangle counting on the BigMem platform, where the overhead is nearly 60%. These results are statistically

significant for all workloads on the commodity platform, but only the triangle counting result on the BigMem platform is significant.

The high overhead of triangle counting on the BigMem platform is a consequence of an artifact of LLAMA's (and GreenMarl's) triangle counting implementation: when the graph is sufficiently large, both LLAMA and GreenMarl sort the vertices by degree before counting to improve cache locality. This is a widely-used optimization for triangle counting on large graphs, but the LiveJournal graph is not large enough to benefit from it: sorting takes longer than the expected gain. However, the resulting cache penalty hurts LLAMA more than GreenMarl, because it applies to every indirection in the LAMA vertex table. For graphs that are large enough to benefit from degree-sorting, the access pattern is better behaved and the difference between LLAMA and GreenMarl shrinks.

6.2.2 GraphLab

LLAMA significantly outperforms GraphLab running on a single machine for all cases except triangle counting on BigMem, where they perform similarly. There are two factors that account for this. The first is memory footprint: on the BigMem platform, GraphLab uses almost 8 GB for the LiveJournal graph and almost 200 GB for Twitter, while LLAMA uses only 0.6 GB and 18 GB respectively. Profiling the execution of PageRank shows the second reason: While LLAMA's PageRank function accounts for 98% of the CPU time, GraphLab's scatter, apply, and gather functions account for a total of 37%. The remaining 63% goes towards the framework supporting the vertex-centric computational model. Even discounting the framework overhead, LLAMA still outperforms GraphLab, but the results are much closer.

Triangle counting is an exception to this trend due to the caching effects observed in GreenMarl discussed above. Because the LiveJournal graph is too small to benefit from degree-sorting, its access pattern is highly random and the triangle counting algorithm scales poorly in the number of cores. GraphLab, however, always scales well in the number of cores by design, so for the smaller and un-

	Time (s)			CPU Time Breakdown (%)			I/O (GB)	
System	Wall	CPU	CPU%	PageRank	Buffer Mgmt.	Other	Read	Write
LLAMA	607.6	1088.5	179	98.0	< 0.1	2.0	118.4	0.0
GraphChi	1260.9	3463.3	274	11.9	86.6	1.5	38.7	0.2
X-Stream	1942.9	7746.2	398	24.8	27.0	48.2	306.3	121.0

Table 6.3: PageRank on Twitter: Performance Breakdown on the Commodity platform.

sorted LiveJournal data on the 64-core BigMem machine GraphLab is able to catch up to LLAMA. For larger graphs, such as Twitter, that benefit from sorting, the vertex access pattern behaves better and LLAMA scales well in the number of cores (similar to Figure 6.3).

6.2.3 GraphChi and X-Stream

LLAMA also significantly outperforms both of these systems, both in and out of memory. Performing better while in memory is not surprising due to LLAMA's use of CSR and `mmap`.

Table 6.3 shows the performance breakdown for one of the out-of-memory cases – PageRank on Twitter on the Commodity platform. LLAMA reads more data than GraphChi because it uses 64-bit IDs instead of GraphChi's 32-bit IDs, and thus the graph structure is larger on disk. LLAMA nonetheless outperforms GraphChi because the computation is CPU bound. We confirmed this by repeating the same computation on a hard disk instead of the faster SSD and found that LLAMA (now limited to one thread in order to avoid seek thrashing) still outperformed GraphChi but by less: only by 35% instead of 200%.

While LLAMA spent 98% of its CPU time in the PageRank function, GraphChi spent less than 12% of its CPU time in the algorithm's update function. Most of the remaining CPU time is spent on buffer management: reading data and transforming it into an in-memory representation. Harizopoulos et al. made a similar observation for relational databases[31].

X-Stream consumes both more I/O and more CPU. Profiling shows that it spends less than 25%

in PageRank; the rest of the time is divided between buffer management and reshuffling the intermediate data between the scatter and gather phases of the computation. This is in part because X-Stream does not have a load phase. Note that if load times are included X-Stream slightly outperforms LLAMA on in-memory LiveJournal on the commodity machine, but falls behind for larger datasets and if multiple workloads are run with one load phase.

6.2.4 Data Ingest

We ported LLAMA's loader to GreenMarl, which allows us to evaluate the overhead of building LLAMA's CSR with the extra indirection table relative to loading GreenMarl's CSR. Our "overhead" ranges from 14.7% to 20.1%.

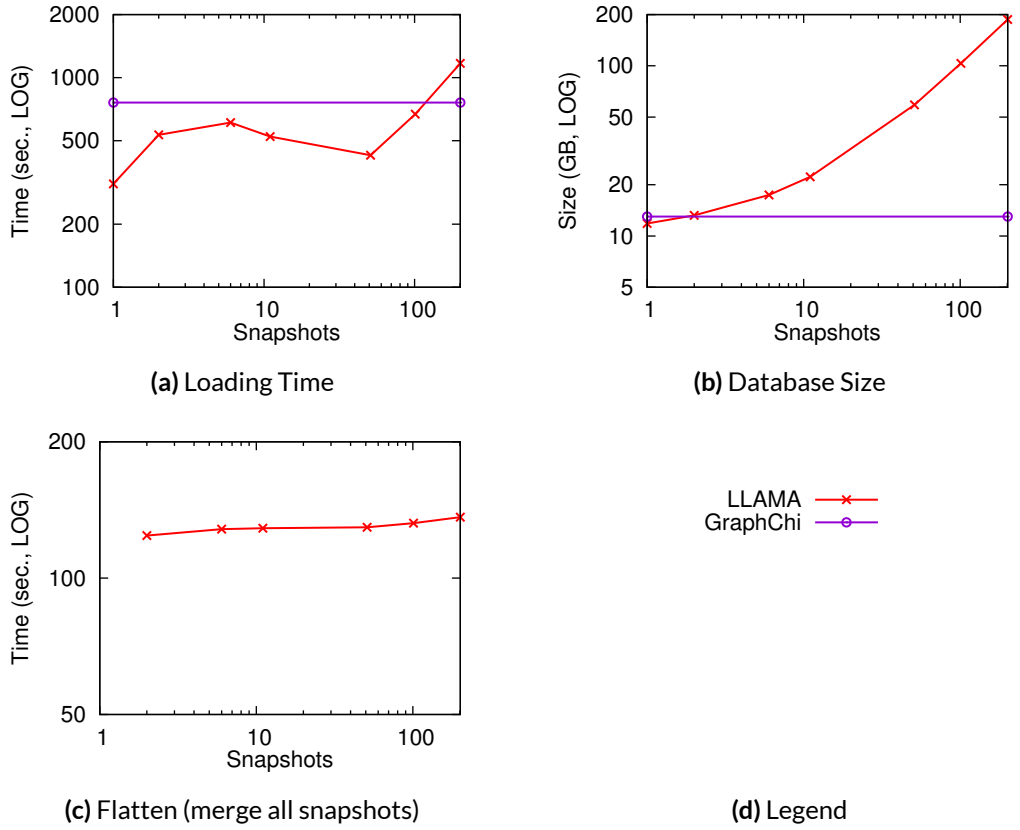
LLAMA loads data 2.5–10x faster than GraphLab and GraphChi when the input file is already sorted, which we found to be the case for many real-world datasets we encountered. LLAMA autodetects if a file is already sorted and then loads it using a simple data transformation. GraphChi and GraphLab always shard and sort the data.

If the data are not sorted, LLAMA marginally outperforms GraphChi, as both do the external sort, but GraphChi must then shard the data, which LLAMA does not need to do.

6.3 Multiversion Access

Next, we evaluate LLAMA's multiversion support consisting of incremental ingest, merging, and computation across multiple versions. We present out-of-core results on the Twitter graph only, as the other datasets show similar trends, even when running in memory.

We ran seven experiments, breaking the data into 1, 2, 6, 11, 51, 101, and 201 snapshots. The single snapshot case is the same as in Section 6.2. For $n > 1$ snapshots, we first loaded a random 80% subset of edges as the first snapshot and then used a uniform random distribution to assign



(a) Loading Time

(b) Database Size

(c) Flatten (merge all snapshots)

(d) Legend

Figure 6.1: Loading and Flattening Multiple Snapshots of the Twitter Graph on the Commodity platform. The GraphChi preprocessing time is shown for reference.

the remaining 20% of the edges to the other $n - 1$ snapshots. This simulates an initial large ingest followed by multiple smaller ingests.

We chose to pick which edges to treat as updates uniformly at random as this provides us with the worst-case scenario (without specifically constructing adversary workloads). Bursty updates would only improve our locality and thus our performance, which is a function of the number of snapshots in which each vertex appears.

Figure 6.1 shows the load time, resulting database sizes, and the merge time. We include the GraphChi numbers for reference where appropriate. The database size grows steeply, from al-

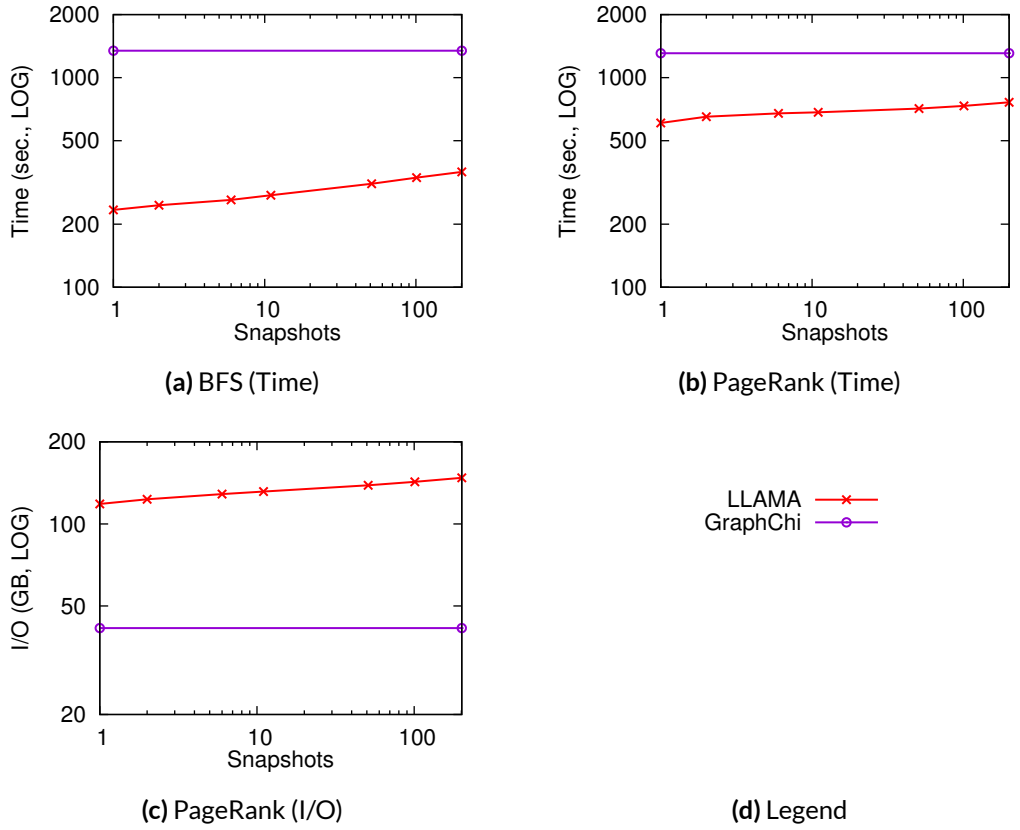


Figure 6.2: Computing Across Multiple Snapshots of the Twitter Graph on the Commodity platform. The GraphChi preprocessing time is shown for reference.

most 12 GB for one snapshot to 22 GB for 11 snapshots, and 187 GB for 201 snapshots. Most of this growth is due to the multiversion vertex table; the continuation records account for only 2.52 GB for 201 snapshots. We plan to implement a tool to delete unneeded vertex tables in persistent LLAMA databases, which users can use if they do not require multiversion access to all snapshots (the function that does this currently works only for in-memory LLAMAs). This significantly decreases the total database size; for example, deleting all but the most recent versions of the vertex table decreases the database size to approximately 14.5 GB for 201 snapshots, yielding only 21% overhead relative to a single snapshot.

The total load times increase on average but not with a clear monotonic pattern, because there are two competing forces: writing out the database files takes longer as they get larger, but since the total amount of data is constant, with more snapshots each delta is smaller and sorts faster. In particular, the dip in the load time reflects the point at which the delta snapshots became small enough to sort in memory.

Figure 6.2 shows that the computational performance decreases slightly as we introduce more snapshots, but it does so quite gradually. The most dramatic slowdown occurs in the first 11 snapshots – that is, the slow down between 1 and 11 snapshots is greater than the slowdown between 11 and 201 snapshots: PageRank slows down on average by 1.2% per snapshot from 1 to 11 snapshots, then by 0.08% for the next 100 snapshots, and then by 0.04% for the remaining 100 snapshots. BFS slows down first by 1.8% per snapshot for the first 10 snapshots, then by 0.25% per snapshot for the next 100, and then only by 0.09% for the final 100 snapshots.

The computation must read more data, because adjacency lists are fragmented and contain continuation records, requiring more random I/O. The increase in size of the multiversion vertex table does *not* contribute to decreased performance. Scanning the vertex table requires the same number of I/Os regardless of the number of snapshots as the number of pages read stays the same. Although the access becomes less sequential, this does not impact performance significantly on modern SSDs with high random I/O speeds.

The time to merge all the snapshots into one (Figure 6.1c) increases only slightly with the number of snapshots. This is an I/O bound operation; the amount of I/O does not increase significantly with the number of snapshots, so the performance curve is nearly flat.

For write intensive workloads, we assume a model where we are constantly merging snapshots to bound the total number. For example, if we create a snapshot every 10 seconds (which would correspond to $50,000$ new edges in a Twitter graph or $550,000$ in a Facebook graph[18]), we find that merging fewer than 100 snapshots takes 120 seconds on our platform. Then in the steady state,

we expect to have no more than 24 snapshots (the 12 being merged and the 12 newly created ones while the merge takes place), which occupies 45 GB if all vertex table versions are present.

6.4 Scaling in the Number of Cores

We now turn to scalability. Figure 6.3 shows the performance of BFS, PageRank, and triangle counting on the Twitter graph as a function of the number of cores on the BigMem platform.

The plot includes two lines for LLAMA for BFS and PageRank: LLAMA-1, with the entire graph in a single snapshot, and LLAMA-11, which loads it into 11 snapshots as explained in Section 6.3 (we chose 11 snapshots for this experiment as the most dramatic slowdown in multiversion access occurs within the first 11 snapshots). Both LLAMA and LLAMA-11 scale as well as GreenMarl and similarly to GraphLab, the two other in-memory systems.

GraphChi scales well for a small number of cores for both BFS and PageRank, but the curve then levels off. Profiling GraphChi shows that preloading vertices before computation improves dramatically from one to four cores and then only minimally afterwards – and as shown in Table 6.3, this buffer management is the dominant part of GraphChi's execution. This is expected since the Linux page cache does not scale well past 4 to 8 cores[69].

X-Stream likewise scales only for a small number of cores on the BigMem platform. Profiling shows that the runtime of its routines, now hitting the buffer cache instead of the disk, improves for up to four cores and then levels off, also because of the page cache scalability limitation. This in turn affects the performance of the rest of the system that depends on the loaded data.

Figure 6.3c shows the performance of triangle counting on the BigMem platform for LLAMA, GreenMarl, GraphLab, and GraphChi. We did not run LLAMA-11 because our triangle counting implementation requires all snapshots to be merged into one, which then results in the same performance as LLAMA-1 displayed in the figure.

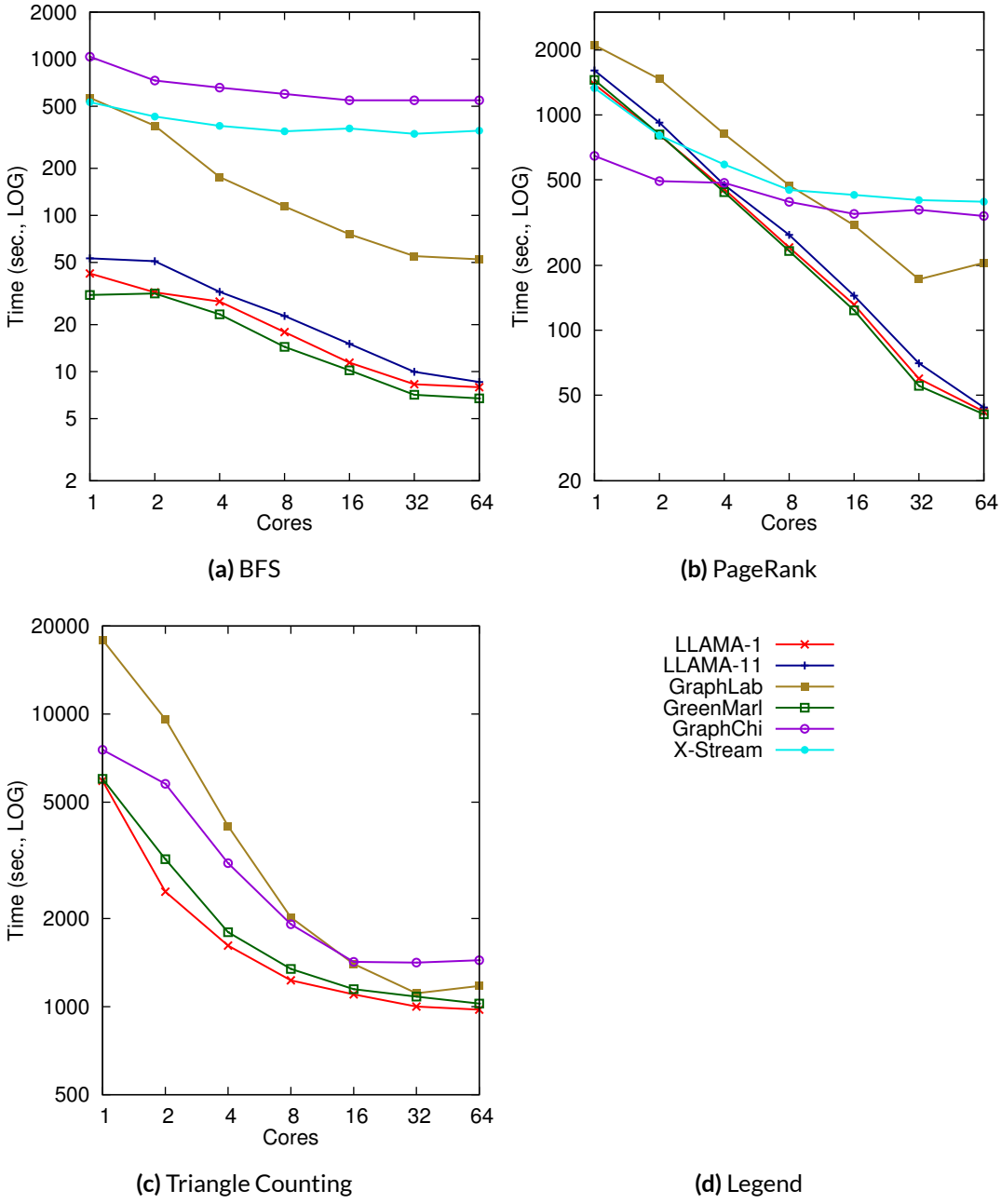


Figure 6.3: Performance Scaling as a Function of Core Count on the BigMem platform for the Twitter graph.

All four systems scale well, LLAMA, GreenMarl, and GraphLab converging at comparable runtimes when using all 64 cores, and GraphChi performing 40% slower. Single-core GraphLab is slower than LLAMA, but it then scales 10x. GraphChi scales only 5.2x; LLAMA and GreenMarl scale 6x. The culprit is sorting the vertices by their degree, which is not very scalable – the actual triangle counting runtime not including this step scales by a factor of 20!

6.4.1 Out-of-Core Scalability

LLAMA's and LLAMA-11's out-of-core performance on the Commodity platform both improve on average by 25% as they scale from 1 to 4 cores (not shown in a figure). GraphChi scales well on the Commodity platform, halving the runtime from 1 to 4 cores, but its runtime at 4 cores is still more than twice LLAMA's runtime.

In contrast, X-Stream does not scale at all on the Commodity platform. X-Stream performs significantly more I/O than either LLAMA or GraphChi, and it is I/O bound (despite its high CPU usage, which is partly due to spin-locking while waiting for I/O). Thus, adding more cores does not help improve performance.

6.5 Scaling in the Graph Size

Finally, we evaluate how LLAMA's performance scales as a function of the number of vertices and the average degree in the out-of-core scenario. We study both of these trends separately on two series of synthetically generated R-MAT graphs, focusing on PageRank and BFS on the commodity machine.

In Figure 6.4, we fix the average vertex degree to 16 as recommended by Graph500[27] and vary the number of vertices from $2^{25} \approx 33.6 \times 10^6$ to $2^{29} \approx 536.9 \times 10^6$, which produces LLAMA

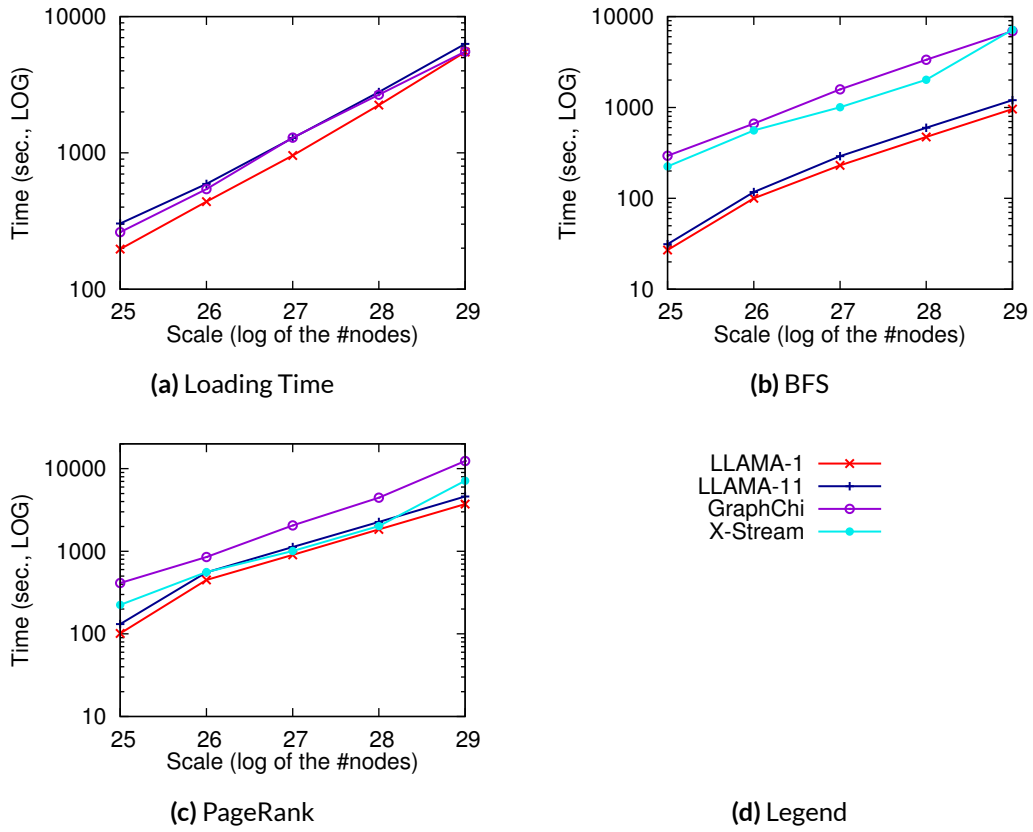


Figure 6.4: R-MAT, varying the number of vertices on the Commodity platform using. Only the graph with 2^{25} vertices fits in memory; the sizes of the remaining graphs range from 100% to 900% of memory.

graph sizes ranging from 4.5 GB to 72.0 GB. In Figure 6.5, we fix the number of vertices to 2^{25} and vary the degree from 20 to 60, producing LLAMA graphs ranging in size from 5.5 GB to 15.5 GB. We chose these ranges so that the leftmost point on the graph fits in memory, while the other points (potentially far) exceed memory. We therefore only compare to other out-of-core systems. We generated the graphs using probabilities $a = 0.57, b = 0.19, c = 0.19$ (recommended by Graph500) to produce graphs with the scale-free property found in many real-world graphs. The figures also include two lines for LLAMA, one for the graph loaded into a single snapshot and one for 11 snapshots.

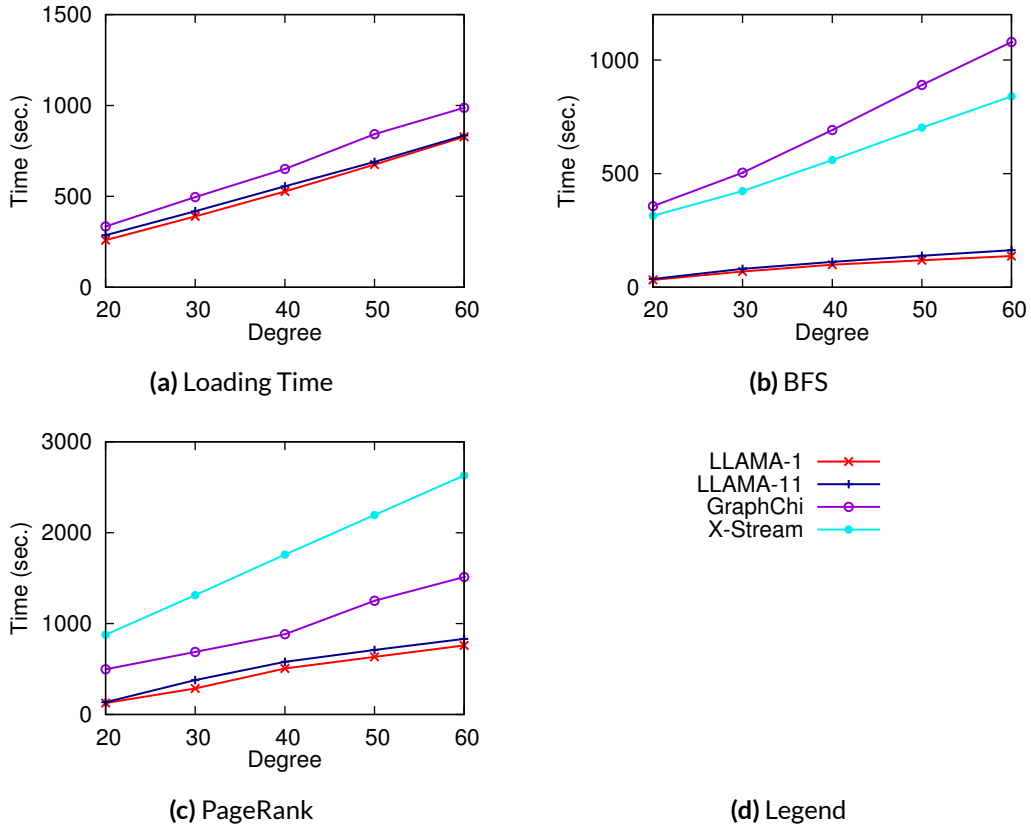


Figure 6.5: R-MAT, varying the average vertex degree on the Commodity platform using. Only the graph with 20 edges per vertex fits in memory; the sizes of the remaining graphs range from 100% to 200% of memory.

LLAMA-1 and LLAMA-11 scale well both with the number of vertices and with the vertex degree, even to graphs that are 9 times the size of memory, consistently outperforming the other tested systems, for all the same reasons discussed in Section 6.2.

6.6 Summary

LLAMA demonstrates that a modified CSR representation, capable of supporting updates and working on datasets that exceed memory, can provide performance comparable to in-memory only

solutions and better than existing out-of-memory solutions. LLAMA's ability to efficiently produce snapshots makes it ideal for applications that receive a steady stream of new data, but need to perform whole-graph analysis on consistent views of the data. Its ability to handle relatively large graphs on a small, commodity machine further extends its utility.

7

SLOTH – Sliding Window Management over a Stream of Edges using LLAMA

The multiversion nature of LLAMA makes it well-suited for maintaining a sliding window over a quickly evolving graph, which is useful for answering queries such as "Who is the most influential user of Twitter within the last hour?" Answering such questions requires the system to maintain a view of the graph containing only those edges inserted during the last t_L seconds, or in other appli-

cations, by the last L inserted edges. Deleting edges is implicit: The system considers edges deleted as soon as they fall out of the sliding window.

There are two approaches to answering such queries:

- Leverage streaming algorithms, or
- Explicitly maintain a sliding window over the stream of edges and use a standard, general-purpose algorithm

Both of these approaches were pioneered only recently. Crouch et al.[16] initiated a theoretical study of streaming algorithms for sliding windows, in which they described several algorithms that work in $O(n \text{ polylog } n)$ space, where n is the number of nodes in the graph. The difficulty with this approach is that it requires users to invent new streaming algorithms for each new application, and streaming algorithms are often only approximations.

It is therefore valuable to have a system that can maintain a sliding window explicitly in $O(L)$ space, enabling its users to run standard, unmodified algorithms on the graph. Many algorithms can work directly on the graph as is, while others require the graph to not have any duplicate edges. In some cases it is beneficial to maintain edge weights instead of keeping duplicate edges.

STINGER[4] is the first system that enables sliding window computation on graphs. It provides a data structure with time-stamped edges and efficient enough inserts and deletes, on top of which it is possible to build a sliding window management system. Cheng et al.[12] proposed another sliding window system, but left its implementation as future work. SLOTH (short for "sliding over things"), the sliding window extension to LLAMA, implements sliding window management using CSR and is, to the best of our knowledge, the first solution to properly maintain edge weights instead of having duplicate edges.

SLOTH accepts an incoming stream of edges, maintains a sliding window over them, and runs unmodified algorithms written for LLAMA on consistent snapshots of the data. It supports three

variants of sliding windows, each of which adds complexity to the preceding variant:

1. Basic sliding window
2. Sliding window without duplicate edges
3. Sliding window in which duplicate edges translate into incrementing edge weights

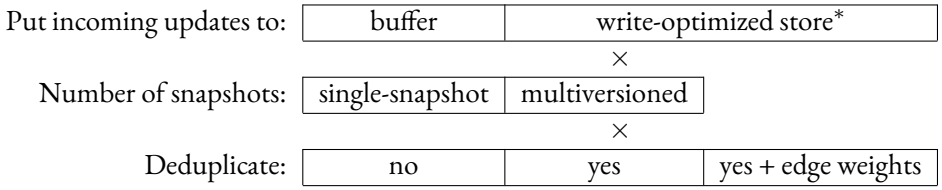
SLOTH provides two implementations for each:

1. The single-snapshot implementation, in which the system maintains a queue of incoming updates and builds a single-snapshot CSR representation on demand.
2. The multiversioned implementation, in which the sliding window consists of a fixed number of snapshots created at fixed time intervals or at a fixed number of inserted edges. For example, a user can configure SLOTH to compute a graph algorithm every minute on a sliding window describing the last hour of data, creating a new read-optimized snapshot every minute and maintaining a window of the last 60 snapshots.

The rest of the chapter is organized as follows: Section 7.1 describes SLOTH at a high level. Section 7.2 then describes the process of maintaining a multiversioned sliding window, including aging out edges and their weights. Section 7.3 presents experimental results.

7.1 System Overview

SLOTH is built on top of space-optimized LLAMA with explicit adjacency list linking and multiversioned LAMA vertex table (Section 4.4), because it presents the best overall combination of memory efficiency and runtime performance while providing multiversion support. We handle deletions in the multiversioned sliding window using a deletion vector (Section 4.4.1) instead of copy-on-delete even though we use explicit linking, so that an edge remains associated with the original snapshot in which it was inserted and can therefore be aged out easily.



* Currently available only for single-snapshot sliding windows.

Figure 7.1: Summary of SLOTH Configurations

SLOTH currently works only in memory; extending it to compute on sliding windows larger than RAM is left for future work.

SLOTH buffers incoming updates either in a simple non-queryable buffer or in LLAMA's write-optimized graph store (Section 4.7), depending on 1) the sliding window type, 2) whether applications need immediate access to the current state of the graph, and 3) the required features, such as updating properties. Storing updates in a buffer has lower memory and runtime overheads, but it currently only supports edge insertion. However, it is currently also the only option for single-snapshot sliding windows. Multiversioned windows can use either the buffer or the write-optimized store. The write-optimized store has a larger memory footprint than the buffer and requires more CPU cycles to process each update, since it has to process each update twice: first apply it to the write-optimized store and then when creating a new snapshot. The advantage of using the write-optimized store is that makes the new data immediately queryable, and it enables SLOTH to support the same types of updates as LLAMA, such as deleting edges, adding and deleting vertices, and updating properties – not just inserting edges. While we believe that such features are useful, most streaming applications do not need them, so we configure SLOTH to store incoming updates in the non-queryable buffer unless stated otherwise.

Figure 7.1 summarizes the different configuration options of SLOTH: two options for where to buffer the incoming data, two options for the number of snapshots in the sliding window, and three options for deduplication.

In a typical mode of operation, SLOTH allocates one thread to process incoming updates and insert them into the buffer or the write-optimized store, while the rest of the cores run analytic algorithms each time the sliding window advances. The system can advance the sliding window either in the loader thread or in the graph analytic threads before running an analytic task. We use the former approach by default, since it overlaps loading new data and advancing the sliding window with graph computation. The latter approach is better suited for scenarios with a high rate of incoming updates but relatively little analytic computation.

7.2 Maintaining a Multiversioned Sliding Window for Graphs

A multiversioned sliding window consists of multiple snapshots, each corresponding to the time interval between two advances of the sliding window. We advance the window by creating a new read-optimized snapshot and aging out the oldest snapshot.

The following sections describe the three variants of sliding windows with increasing complexity: a basic sliding window, and two types of sliding windows with special duplicate handling.

7.2.1 Basic Sliding Window (Type 1)

Maintaining a basic multiversioned sliding window is straightforward. We modified LLAMA to ignore adjacency list fragments in snapshots below S_L, where S_L is a configurable parameter. To advance a sliding window consisting of w snapshots:

1. Increment S_L.
2. Create a new read-optimized snapshot from the new data. If the user configured LLAMA to store precomputed degrees in the vertex table, then for each vertex that has an adjacency list in the aged-out snapshot, subtract from its precomputed degree in the new vertex table the number of aged-out edges. If deletions are disabled, the number of aged-out edges is just the

length of the adjacency list fragment in the aged-out snapshot. If deletions are enabled, we do a single pass over the aged-out edge table to count the number of edges that still exist.

3. Delete the oldest snapshot(s) $S_j < S_L$ that are currently not in use.

To delete a snapshot:

1. Decrement reference counts for all pages that comprise the vertex and property tables of the given snapshot, freeing all pages whose reference count drops to zero.
2. Free the LAMA indirection table for the vertex table and for all property tables.
3. Free the edge table.

It is not necessary to update pointers to aged-out snapshots, since LLAMA ignores references to them.

The time complexity of advancing the sliding window differs depending on whether deletions are allowed. With deletions, it is linear in the number of new edges plus aged-out edges. Without deletions, it is linear in the number of new edges plus the number of LAMA array pages, which is a relatively small number.

By reclaiming space, LLAMA maintains constant memory usage, but consumes snapshot ID space. If there is a risk of exhausting the ID space, we can simply increase the ID size. Recall that LLAMA stores the deletion vector in the 16 most significant bits of the edge table cells, allowing up to 2^{16} snapshots. Creating a snapshot every minute exhausts the snapshot ID space in 45 days. But, for example, if we need no more than 36 bits to represent a vertex ID (i.e., there are no more than 2^{36} vertices in the graph), we can use $64 - 36 = 28$ bits for snapshot IDs and thus create up to 2^{28} snapshots. Even if we create a snapshot every minute, we will not exhaust the ID space for 510 years! We can of course always wrap the ID space but have not yet found it necessary to do so.

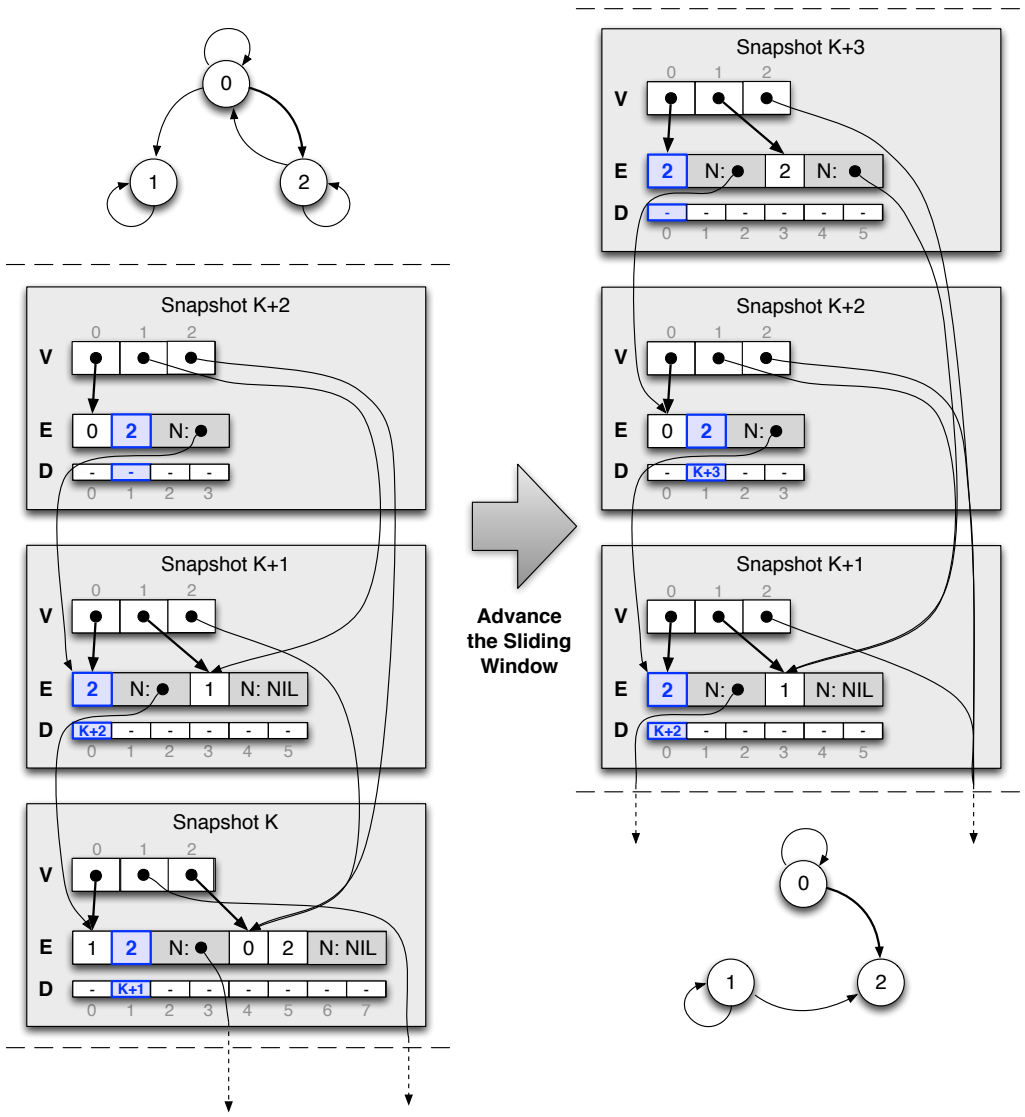


Figure 7.2: Advancing a Sliding Window without Duplicate Edges spanning 3 snapshots with edge $0 \longrightarrow 2$ added multiple times (highlighted in blue). "V" stands for the vertex table, "E" for the edge table, and "D" for the deletion vector. We omitted the internal representation of the vertex table from the figure for clarity.

7.2.2 Sliding Window without Duplicate Edges (Type 2)

To maintain a sliding window without duplicate edges, we modify the first approach to search the sliding window for an existing edge before inserting an identical one. If we find a duplicate edge, we delete it from the older snapshot by writing the ID of the newly created snapshot to the corresponding element in the deletion vector. This ensures that the deleted duplicate edge is still visible to computation running on older snapshots, while computation executed on the new snapshot sees only one edge, the newly added one.

We do not need to do anything special when aging out a snapshot: Duplicate edges in an aged-out snapshot are already deleted, and their corresponding newer duplicates are in newer snapshots, so aging out the snapshot does not delete the duplicate edges.

Figure 7.2 shows a simple example of advancing a sliding window spanning three snapshots, in which edge $0 \longrightarrow 2$ was added multiple times (highlighted in blue). Each time the edge was added, SLOTH deleted the old version of the edge and added it again in the new snapshot. The older duplicate is thus already deleted in the aged-out snapshot.

7.2.3 Sliding Window with Edge Weights (Type 3)

In the final variant of sliding windows, SLOTH increases the weight of an edge to add a duplicate, and it decreases the weight to age out a copy of an edge.

We modify the aforementioned deduplicated sliding window approach so that it inserts new edges with weight 1, or if it finds a duplicate, with weight 1 + the weight of the duplicate. We additionally maintain a "forward pointer" edge property that stores the edge IDs of the corresponding newer duplicate edges or NIL if the edge does not have a newer duplicate. We use this property when aging out snapshots to decrease the weights of the newer duplicates.

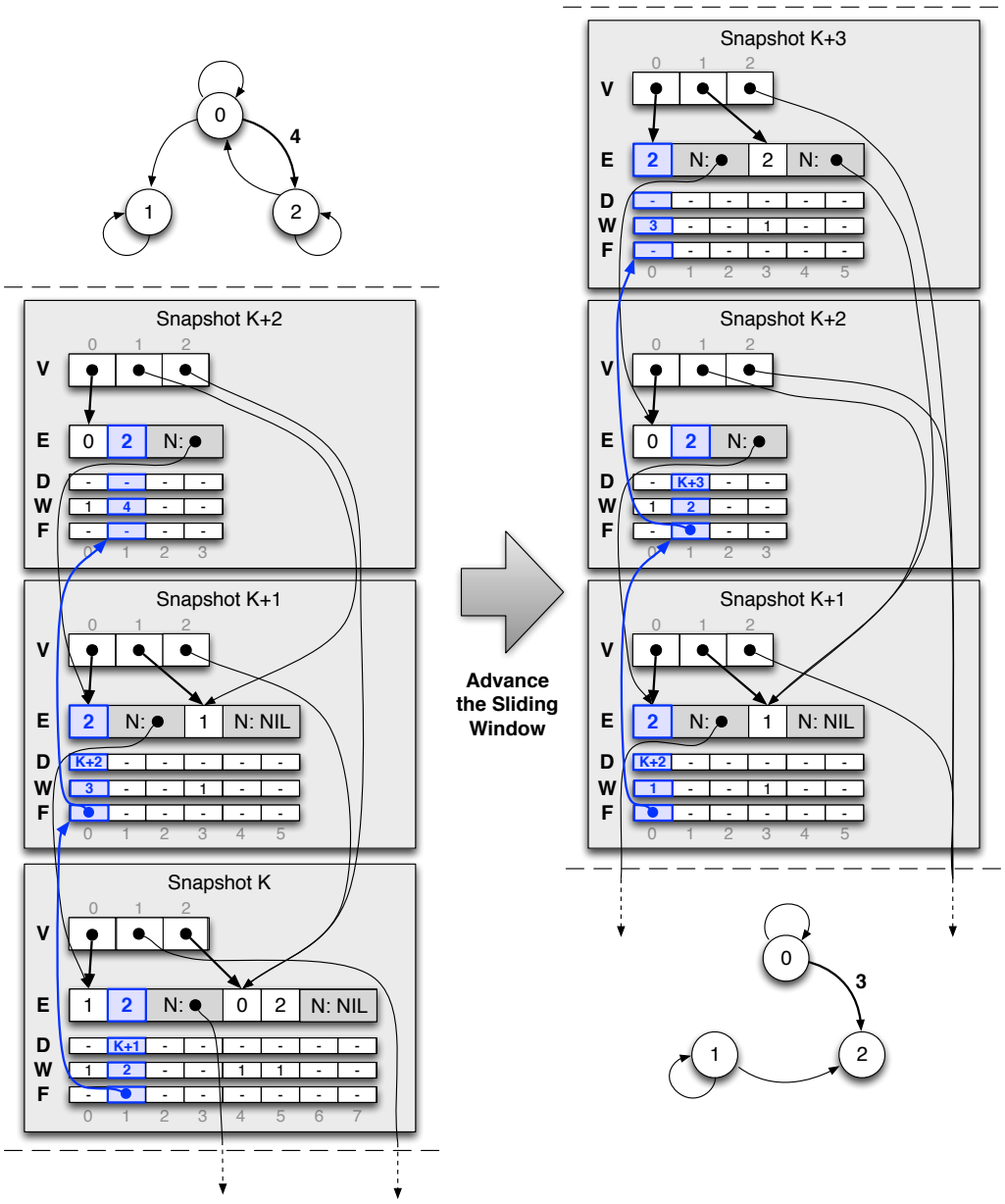


Figure 7.3: Advancing a Sliding Window with Edge Weights spanning 3 snapshots with edge $0 \longrightarrow 2$ added multiple times (highlighted in blue) and with initial weight 2. "V" stands for the vertex table, "E" for the edge table, "D" for the deletion vector, "W" for the weight property, and "F" for the forward pointer. We omitted the internal representation of the vertex table and of the edge properties from the figure for clarity.

For example, consider Figure 7.3, in which edge $0 \longrightarrow 2$ was added multiple times, and it had an initial weight of 2. When we age out this edge, we subtract its weight from *all* of its newer duplicates, which we get by chasing the forward pointers until we reach NIL, which corresponds to the most recently added duplicate edge.

7.2.4 Practical Considerations

Memory Footprint

SLOTH creates a new snapshot (and ages out an old snapshot) each time it advances the sliding window, which is a prerequisite to computing a graph algorithm on a consistent snapshot of the graph. The number of snapshots contained in the window grows with the frequency of advancing the window, in addition to growing with the size of the window, so a sliding window in SLOTH can contain a large number of snapshots. We cannot merge these snapshots, because doing so would prevent SLOTH from properly aging out individual snapshots.

Not being able to merge presents a challenge, since the memory footprint of LLAMA, and thus also of SLOTH, grows steeply with the number of snapshots, almost entirely due to the multiversioned nature of the vertex table (see Section 6.3). In the worst-case scenario, the updated vertices are evenly distributed throughout the vertex ID space, which causes the vertex table to copy-on-write a large number of pages, most of which only update one or two vertices. We unfortunately observe this behavior even when the stream of new edges follows the preferential attachment model, unless the vertices are clustered so that vertices that are likely to be updated together have similar IDs, or only few vertices are updated within any given time interval between two advances of the sliding window.

The easiest way to decrease the memory footprint is to decrease the size of the vertex table pages, but this has an adverse effect on performance of the graph analytic applications as well as on the

time it takes to create a new read-optimized snapshot. For example, decreasing the page size from 8 KB to 1 KB in our experiments on the Twitter graph [42] reduced the memory footprint by more than a factor of three, but decreased overall system performance by more than 30%. Smaller data pages produce a larger indirection table, but more importantly, it makes the vertex table layout in memory less sequential over time as SLOTH reuses pages from deleted snapshots when building new snapshots.

We cannot simply delete old versions of the vertex tables either, because we still need to scan each aged-out snapshot to update vertex degrees and edge weights.

We solve this problem by converting old versions of vertex tables to the corresponding *sparse* representations before deleting them. A sparse representation of a version of a vertex table is simply two parallel arrays containing the same information as the vertex table:

- *Vertex IDs*: The array of IDs of all vertices modified in the given snapshot
- *Vertex Data*: The array of the corresponding vertex structures copied from the vertex table

These arrays contain sufficient information to age out the corresponding snapshot, because it only requires a sequential scan through the modified vertices, which is efficient.

This representation also preserves the ability to access vertices using their IDs, but we do not currently use this feature. The vertex ID array is sorted, so we can find vertices in it using binary search, yielding time complexity $O(\log n)$ instead the $O(1)$ for the original vertex table.

Another alternative for decreasing the memory footprint is to store a time stamp with each edge. This enables us to merge old snapshots while preserving the ability to age out only the appropriate edges, especially if we do not need to update edge weights (as that would require us to keep multiple versions of weights for duplicate edges). We currently do not use this approach, but we leave implementing and evaluating it for future work. This approach would marginally slow down the process of aging out old snapshots. It is also unclear whether it would be more memory efficient than our

current approach, because we expect the extra memory cost of storing timestamps for each edge to be larger than the space consumed by the sparse representations, duplicate edges, and next pointers. On the other hand, having fewer snapshots would improve the performance of graph algorithms. It is also unclear whether the benefit of this would be significant, since as we saw in Sections 4.9 and 6.3, the growth of performance overhead for graph computation slows down with the number of snapshots.

Multiple Sliding Windows

SLOTH currently supports maintaining only one sliding window at the time, but it would be straightforward to support multiple sliding windows of different granularities. Such an implementation would provide the ability to compute over the last minute, hour, and day of the incoming stream simultaneously. We could support this by merging selected snapshots when they fall out of a smaller sliding window into a larger window. For example, if we advance the sliding window every 10 seconds and compute over the last minute and last hour, we would merge each 6 snapshots that fall out of the "last minute" window, so that the sliding window over the "last hour" advances once a minute.

7.3 Evaluation

We evaluate SLOTH to answer the following questions about both single-snapshot and multiversioned sliding windows:

1. How does the streaming rate, the window size, and the interval between two advances of the sliding window affect the performance of graph algorithms?

2. What is the cost of advancing different types of sliding windows?

3. What is the memory overhead for the different window types, streaming rates, window sizes, and advance intervals?

We answer these questions on a graph with a negligible number of duplicates, which results in the worst-case behavior, as eliminating duplicates lowers the amount of data in the sliding window and thus improves our performance.

And then, fixing the streaming rate, window size, and advance interval, we answer the following two questions:

1. How does the duplicate rate and the streaming window type affect the performance of graph algorithms?

2. How does the duplicate rate and the window type affect the cost of advancing the sliding window?

7.3.1 Setup

We run all benchmarks on a commodity machine with a dual-core Intel Core i3, 3.0 GHz and 8 GB RAM with Hyper-Threading. We use one thread to process incoming updates, place them in a buffer, and advance the sliding window. The remaining three threads run graph computations.

We use existing datasets to model a stream of updates by first shuffling edges in the dataset and then streaming them in a random order at a specified rate. This produces an almost worst-case scenario, since this spreads updated vertices throughout the entire ID space. Collocating edges belonging to the same vertex would only improve performance as discussed earlier in Section 4.9.1.

We run each configuration for five times the window length. We use the first one-fifth of the run as warm up and report averages for the remaining four-fifths of the run. The standard deviation corresponding to most of the reported averages is less than 2%.

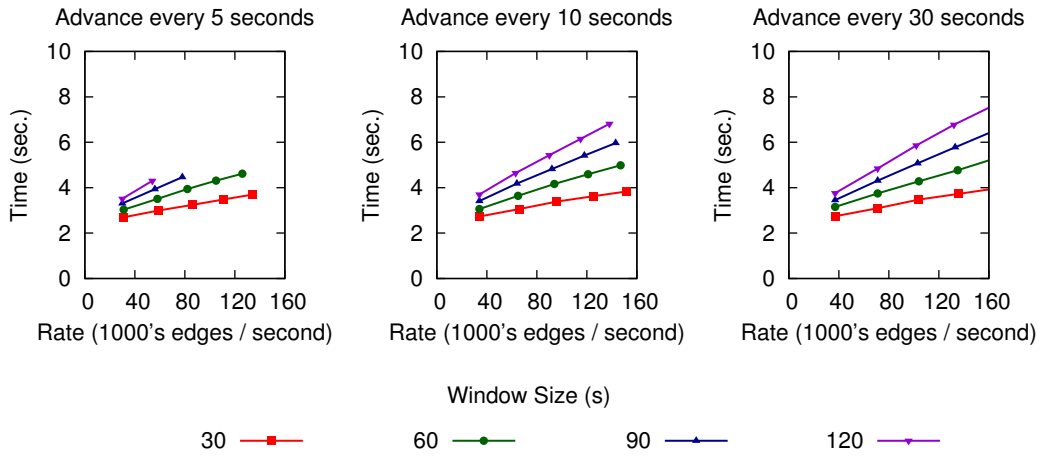


Figure 7.4: PageRank Performance on the Basic Single-Snapshot Sliding Window (Type 1) on the Twitter graph for various advance intervals, window sizes, and stream rates. PageRank performance on the other two types of windows is nearly identical.

7.3.2 Single-Snapshot Sliding Windows, Negligible Number of Duplicates

Graph Algorithm Performance in Single-Snapshot Sliding Windows

Figure 7.4 shows the performance of PageRank[57] on the Twitter graph[42] on the basic single-snapshot sliding window (type 1) and various stream rates, window sizes, and advance intervals. We focus our evaluation on PageRank, a representative whole-graph analysis algorithm, which can be used, for example, to determine the top k most influential users of Twitter in the last t seconds. We vary the rate of incoming updates from 10×10^3 to over 150×10^3 new edges per second on four different window sizes ranging from 30 to 120 seconds, advancing the window every 5, 10, and 30 seconds.

The plots do not include runs in which the time to compute the graph algorithm and/or the time to advance the sliding window was larger than the interval between two consecutive advances of the sliding window. SLOTH currently fails runs in which computations or advancing takes too long.

The analytic performance decreases with the stream rate and the window size, because both of them are proportional to the numbers of vertices and edges in the window. The performance is

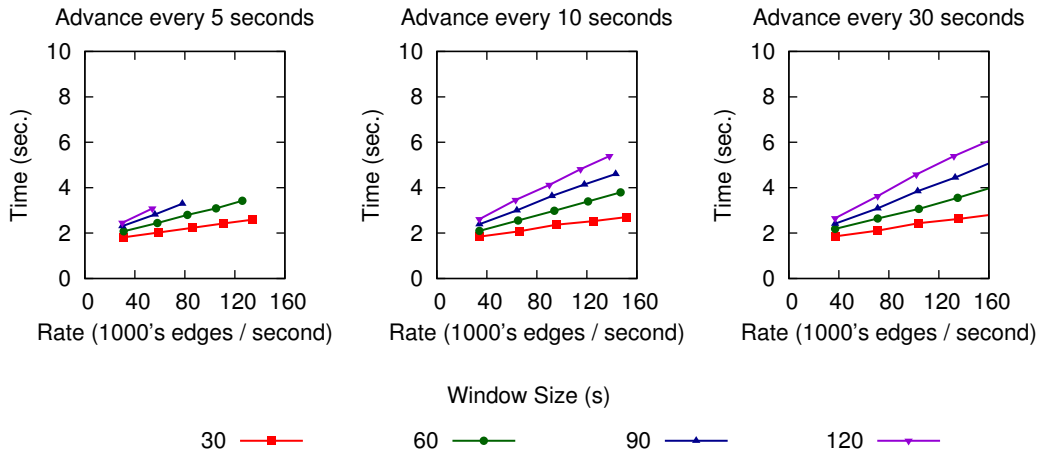


Figure 7.5: The Cost of Advancing the Basic Single-Snapshot Sliding Window (Type 1) on the Twitter graph for various advance intervals, window sizes, and stream rates. The costs on the other two types of windows follow exactly the same trends.

independent of the length of the advance interval, since that has no effect on the amount of data in the sliding window. The graph computation runs 13–18% slower compared to the case without an active loading thread due to interference between the two.

PageRank performance in the two window configurations that explicitly address duplicates is nearly identical, within approximately 1%. (Our PageRank implementation does not currently take advantage of edge weights, but we speculate that there are interesting opportunities there.)

The Cost of Advancing a Single-Snapshot Sliding Window

Figure 7.5 shows the cost of advancing the sliding window on the Twitter graph[42] on the basic single-snapshot window (type 1) and various stream rates, window sizes, and advance intervals.

The cost corresponds entirely to the cost of loading a single-snapshot LLAMA from an edge-list, which increases with the number of input edges. The number of edges in turn increases with the stream rate and the length of the sliding window, but it is independent of the advance rate.

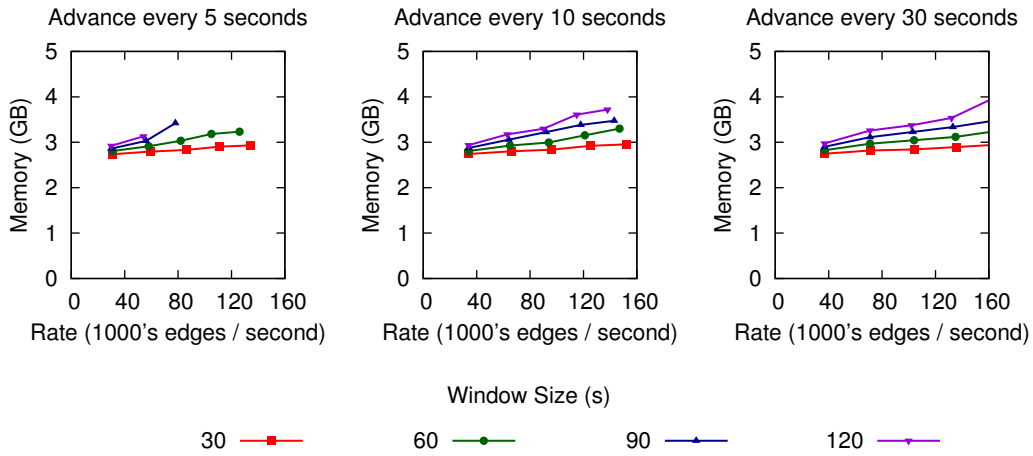


Figure 7.6: The In-Memory Size of the Basic Single-Snapshot Sliding Window (Type 1) on the Twitter graph for various advance intervals, window sizes, and stream rates. The costs on the other two types of windows follow exactly the same trends, except that the variant with edge weights incurs an additional 4 byte overhead for each edge.

The cost of advancing the sliding window without duplicates (type 2) is similar to the cost of advancing the basic window. The overhead of edge weights in the type 3 window increases with the number of edges and is approximately 11% in the worst measured case.

The In-Memory Size of a Single-Snapshot Sliding Window

Figure 7.6 shows the maximum resident set size (RSS) of the SLOTH process using the basic single-snapshot sliding window. The resident set size consists of a fixed memory cost and a variable part that depends on all of these parameters. The fixed cost includes, but is not limited to, the program executable, arrays that hold the computed PageRank data and the intermediate arrays used by the computation (about 0.5 GB for the given number of vertices), and the internal buffers used by the sort function (sized to a constant fraction of the available memory size). The variable part consists of the queue that stores the edges in the sliding window and the LLAMA data structures that store the graph and the properties.

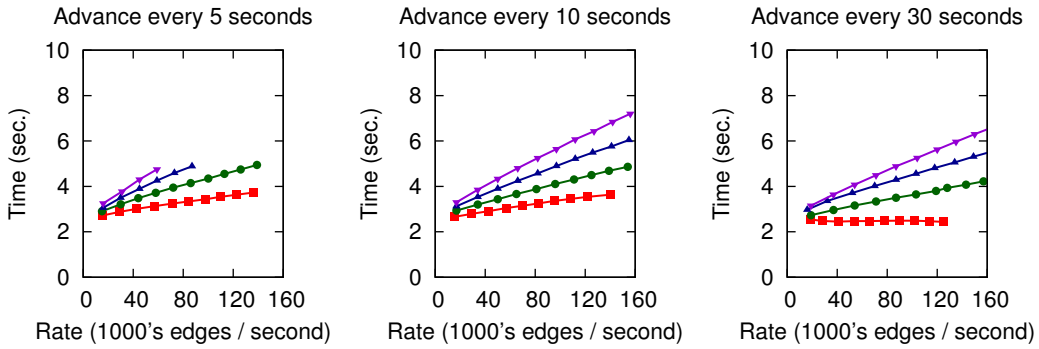
The in-memory cost of SLOTH increases with the number of edges in the sliding window, which increases with the stream rate and the window size but is independent of the advance interval. The sliding window without duplicates (type 2) has almost the same size as the basic window as long as the number of duplicates is small. Maintaining edge weights (sliding window type 3) incurs an additional four byte overhead for each edge to store its weight.

If the graph has V vertices, edges are streaming at rate r edges per second, and if the sliding window is t seconds long, it contains on average $E = rt$ edges. Recall that a vertex in LLAMA occupies 16 bytes and an edge 8 bytes. The size of an edge weight is 4 bytes. The in-memory size of the sliding window is then $16V + 8E = 16V + 8rt$ bytes for the two types of windows without edge weights and $16V + 12E = 16V + 12rt$ bytes for the weighted window. There are two sliding windows in memory: one used for graph computation and one that is concurrently loaded by the loading thread. The queue that stores the edges in the sliding window occupies $16E = 16rt$ bytes, where an edge is represented by two 8-byte vertex IDs. The total memory cost of single-snapshot SLOTH is thus $2(16V + 8rt) + 16rt = 32V + 32rt$ bytes without weights or $2(16V + 12rt) + 16rt = 32V + 40rt$ with weights on top of the aforementioned fixed part of the resident set size (RSS).

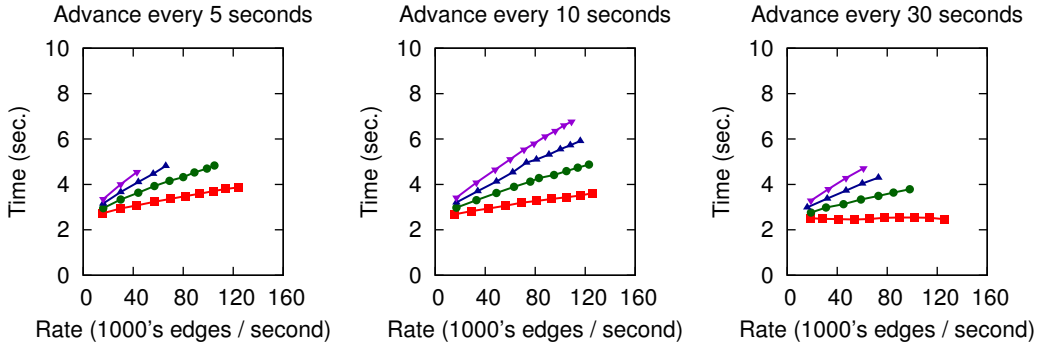
7.3.3 Multiversioned Sliding Windows, Negligible Number of Duplicates

Graph Algorithm Performance in Multiversioned Sliding Windows

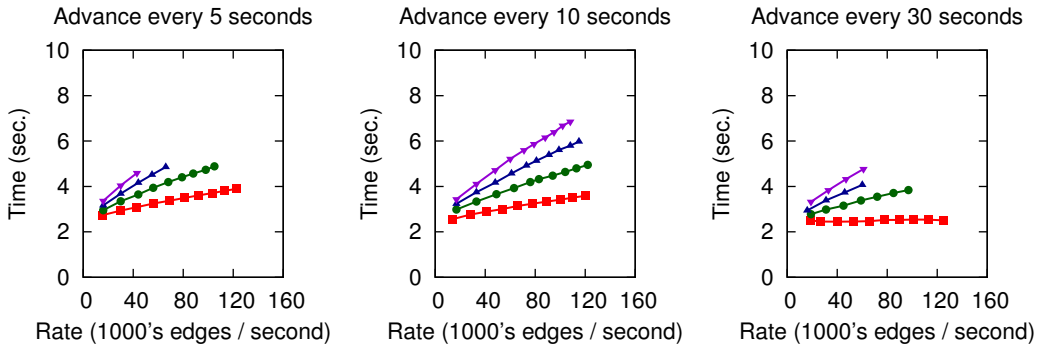
Figure 7.7 shows the performance of PageRank[57] on the Twitter graph[42] for various types of sliding windows, stream rates, window sizes, and advance intervals. The basic (type 1) sliding window has deletions disabled. The plots do not include any runs in which the graph algorithm or advancing the sliding window took more time than the length of the advance interval.



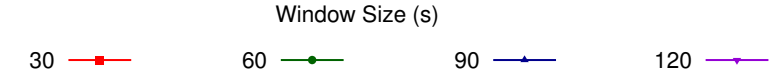
(a) Basic Sliding Window (Type 1) without Deletions



(b) Sliding Window without Duplicates (Type 2)



(c) Sliding Window with Edge Weights (Type 3)



(d) Legend

Figure 7.7: PageRank Performance on Multiversioned Sliding Windows of Various Types on the Twitter graph for various advance intervals, window sizes, and stream rates.

The analytic performance in the basic multiversioned sliding window without deletions decreases as the window size increases when fixing the other two variables, because the window size is proportional to the number of snapshots and also to the number of vertices and edges. The performance gradually decreases with a higher streaming rate, because the rate is proportional to the numbers of vertices and edges in the window, and the more edges in the window the more adjacency lists span multiple snapshots. The performance also slightly decreases as the advance interval gets smaller, which in turn increases the number of snapshots without increasing the total amount of data in the window.

PageRank performs on the basic multiversioned sliding window (type 1) up to 4% faster on the basic single-snapshot sliding window. SLOTH overlaps advancing the sliding window with computation, and the culprit behind this counterintuitive result is the interference between the thread that loads more data and advances the window and the threads that perform the graph analytics computation. The "advancing" thread and the PageRank threads run concurrently for less time in the multiversioned case. The corresponding thread in the single-snapshot case is significantly more data intensive and takes a relatively long time to complete, while the "advancing" thread in the multiversioned case finishes before PageRank completes on the remaining threads. On the other hand, PageRank on multiversioned sliding windows with special handling of duplicates (types 2 and 3) runs slower than on the corresponding single-snapshot sliding windows, both because the deletions are enabled and because the "advancing" thread takes much longer to run due to checking for duplicates.

Adding a deletion vector to the basic sliding window incurs a non-negligible overhead that increases sub-linearly with the streaming rate, starting at 0.4% for $10,000$ edges per second and reaching up to 20% in the worst tested scenario corresponding to the fastest stream rate, longest sliding window, and longest advance interval. Figure 7.8 shows the overhead extrapolated to stream rate 5×10^6 edges/second, which flattens at approximately 36%. We computed it from data that we

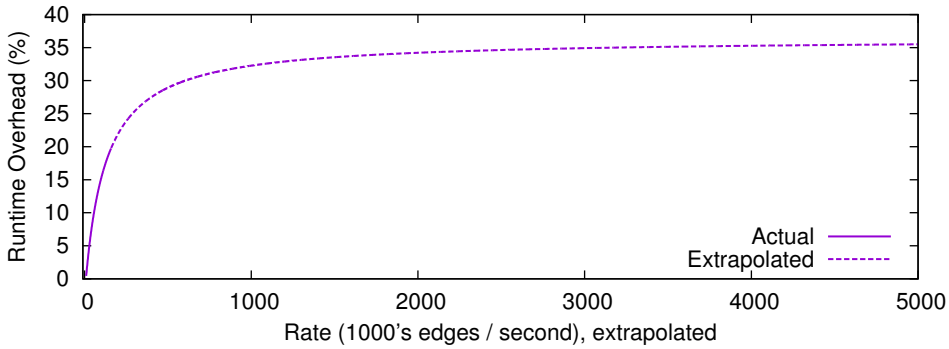


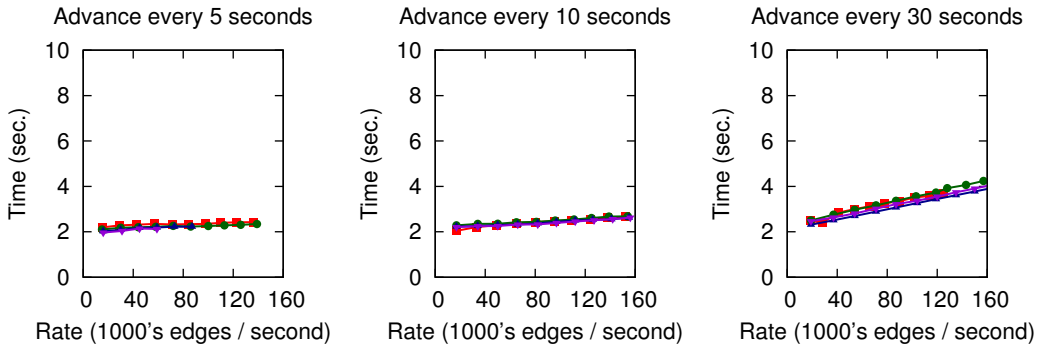
Figure 7.8: Deletion Vector Overhead on analytic performance for the worst tested scenario, extrapolated to stream rate 5×10^6 edges/second (dashed line).

extrapolated using linear regression on the corresponding curve in Figure 7.7a and similarly extrapolated data from an experiment with enabled deletion vectors.

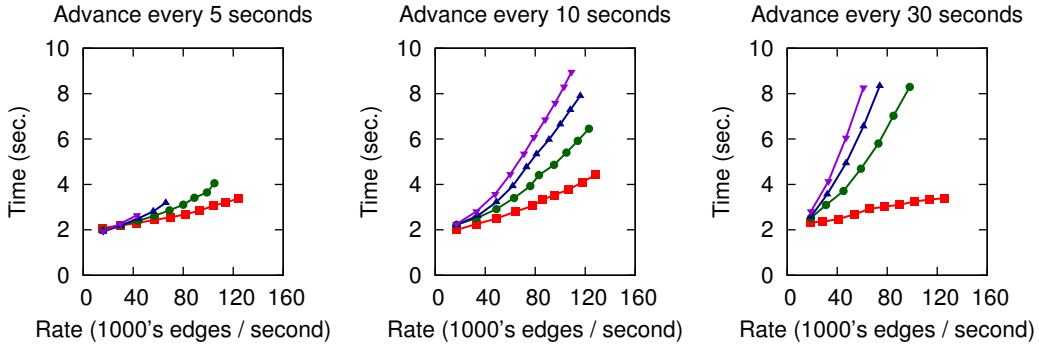
The overhead increases with the advance interval, which is inversely proportional to the number of snapshots comprising the window, resulting in a smaller overhead due to having fewer snapshots. This makes the overhead due to deletions more observable. The overhead is likewise more observable as the stream rate and the window size increase, both of which are proportional to the number of edges in the sliding window.

The two sliding window types that remove edge duplicates require deletion vectors, so they both provide analytic performance comparable to the basic window with deletions, provided that the number of duplicate edges is relatively small (they differ only in the ingest phase).

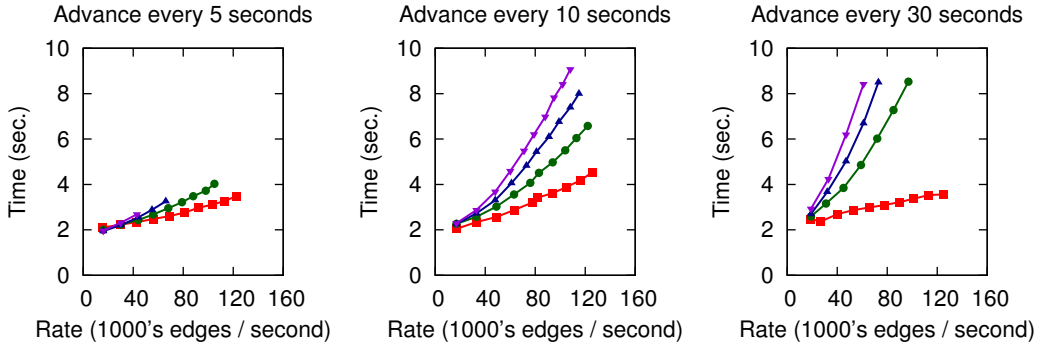
These two types of multiversioned sliding windows perform by up to 16% more slowly than their single-snapshot variants, primarily due to the deletion vectors.



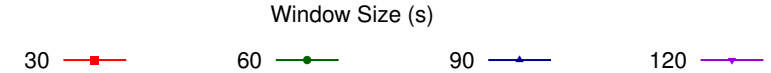
(a) Basic Sliding Window (Type 1) without Deletions



(b) Sliding Window without Duplicates (Type 2)



(c) Sliding Window with Edge Weights (Type 3)



(d) Legend

Figure 7.9: The Costs of Advancing Multiversioned Sliding Windows of Various Types on the Twitter graph for various advance intervals, window sizes, and stream rates.

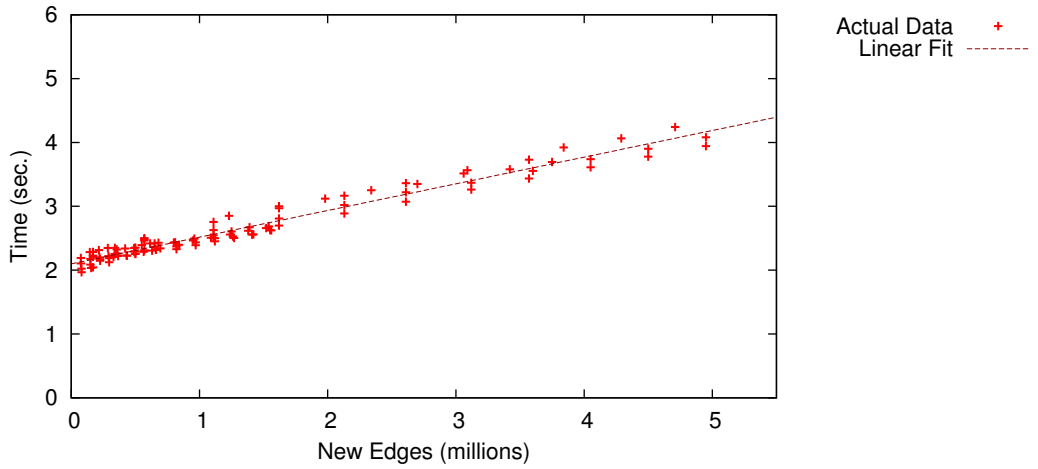


Figure 7.10: The Cost of Advancing the Basic (Type 1) Sliding Window as a function of the number of new edges loaded between two advances of the sliding window.

The Cost of Advancing a Multiversioned Sliding Window

Figure 7.9 shows the cost of advancing a sliding window for the different types of windows and various stream rates, advance intervals, and window sizes. The plots again do not include any runs in which the graph algorithm or advancing the sliding window took more time than the length of the advance interval.

The cost of advancing increases with the number of new edges, which is a multiple of the streaming rate and the length of the advance internal. Figure 7.10 shows this relationship for the basic (type 1) sliding window.

In the case of deduplicated sliding windows, the cost also increases with the number of snapshots that comprise the sliding window. Figure 7.11 shows the cost of advancing the deduplicated sliding window without weights (type 2) as a function of both the number of new edges and the number of snapshots. The cost of advancing of the deduplicated window with weights (type 3) grows similarly, but it is up to 2.3% slower.

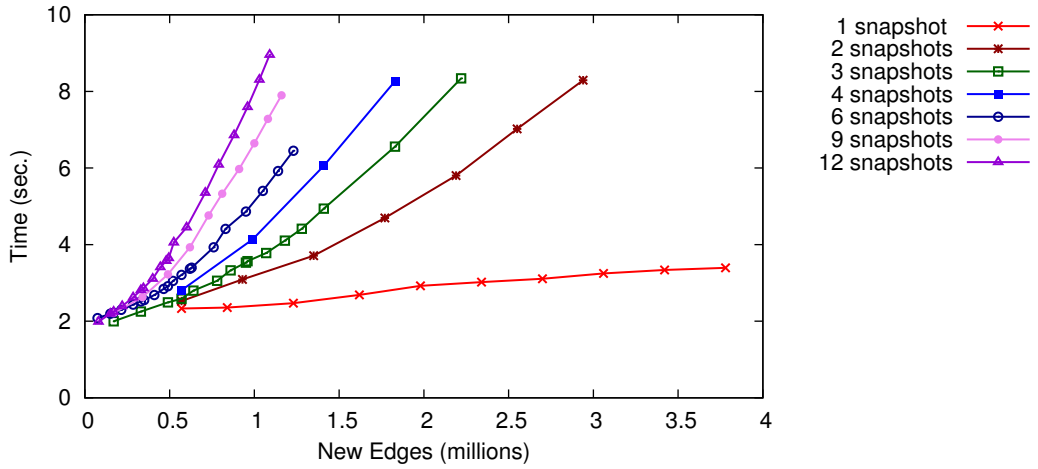


Figure 7.11: The Cost of Advancing the Deduplicated (Type 2) Sliding Window without Weights as a function of the number of new edges loaded between two advances and of the number of snapshots that comprise the sliding window. The cost of advancing of the deduplicated window with weights (type 3) follows similar patterns, but it is up to 2.3% slower.

The relationship between the cost of advancing the sliding window and the length of the advance interval is in fact complicated. Increasing the length of the advance interval decreases the number of snapshots that comprise the sliding window and increases the size of the individual snapshots, which have opposite effects on the cost of advancing:

- Having larger snapshots means that the system ages out more edges when advancing the window. This involves updating the precomputed degrees, and in the case of the sliding window with edge weights (type 3), also updating the edge weights.
- Having fewer snapshots decreases the cost of checking for duplicates and consequently decreases the advancing cost. This is because fewer snapshots imply longer adjacency list fragments and fewer next pointers to follow, both of which improve the cache behavior of checking for duplicates.

The former applies to all types of multiversioned sliding windows, while the latter only applies to

deduplicated windows. The latter is also significantly more important than the former.

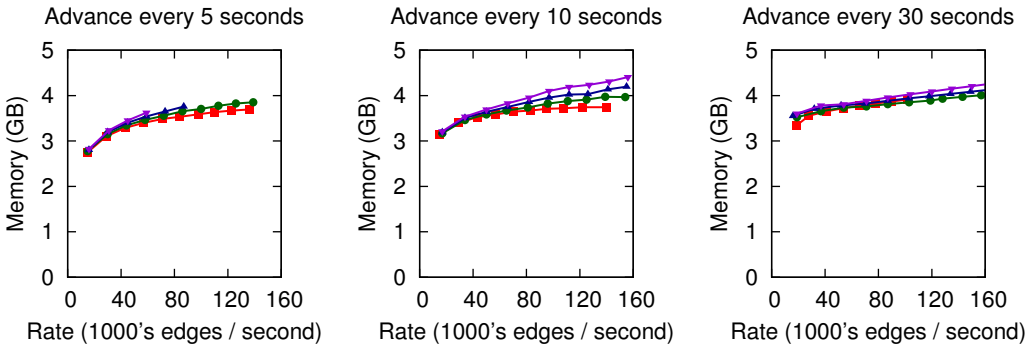
The cost of advancing is independent of the length of the sliding window (i.e., the number of seconds, not snapshots) in the case of windows without duplicates (type 1), but it grows with the window size for the two types of windows with special handling for duplicates (types 2 and 3). Larger sliding windows contain more data, so checking for duplicates requires reading more data. Larger windows also consist of more snapshots, which results in slower checking for duplicates due to more fragmentation of adjacency lists as just mentioned.

As already mentioned, maintaining edge weights costs up to 2.3% on top of checking for duplicates (comparing window types 2 and 3). This overhead is mostly independent of the stream rate, but it increases slightly with the number of snapshots in a sliding window: Having more snapshots results in edge weights being distributed over a slightly larger number of pages due to the wasted space in the representation that parallels the space occupied by the next pointers in the edge table. This simply means that more pages need to be allocated while advancing the window, resulting in a small runtime overhead.

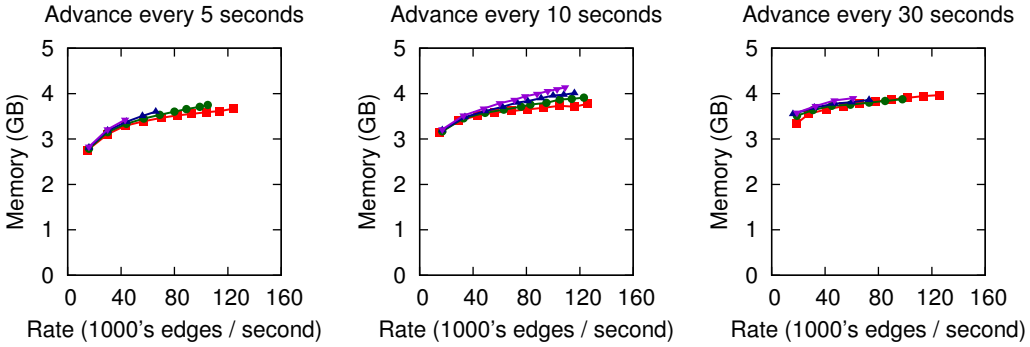
Sliding windows that consist of only one snapshot, as it is the case when we advance a 30-second sliding window every 30 seconds, have low advance cost, because they are both faster to load with new data and faster to age out. Aging out is faster primarily since it does not need to transform old vertex tables to their sparse representations, because it can just delete them.

The In-Memory Size of a Multiversioned Sliding Window

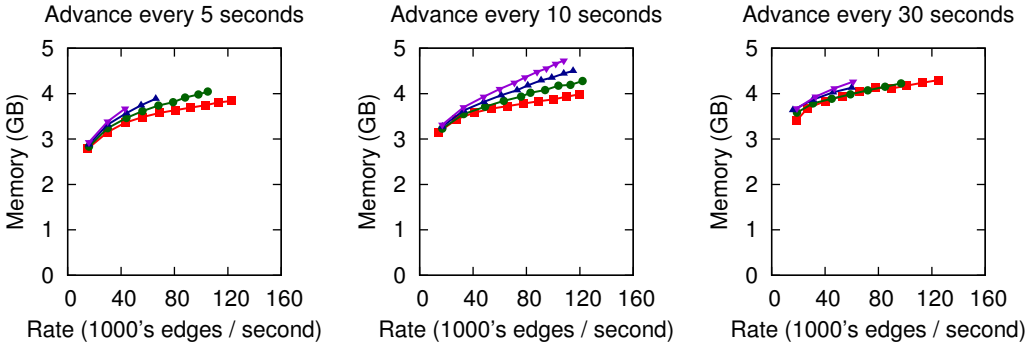
Figure 7.12 shows the total memory cost of SLOTH, reported as the maximum resident set size (RSS) of the process, for the various types of multiversioned sliding windows, advance intervals, window sizes, and stream rates. As mentioned earlier, the resident set size consists of a fixed and a variable component. The variable part in this case consists of the buffers that store the incoming updates and the LLAMA data structures that store the graph structure and properties.



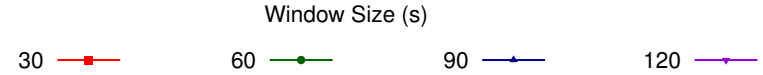
(a) Basic Sliding Window (Type 1) without Deletions



(b) Sliding Window without Duplicates (Type 2)



(c) Sliding Window with Edge Weights (Type 3)



(d) Legend

Figure 7.12: The Memory Costs of Multiversioned Sliding Windows of Various Types on the Twitter graph for various advance intervals, window sizes, and stream rates.

The memory consumption increases with the stream rate and the window size, both of which are proportional to the number of edges in the sliding window and the number of staged updates in the temporary buffers (that will be added to the sliding window the next time it advances). The memory cost also increases with the length of the advance interval, which correlates with the amount of data in the temporary buffers.

The basic sliding window and the sliding window type without duplicates (types 1 and 2) occupy similar amounts of memory in this case, because the fraction of duplicates in the dataset is negligible. The sliding window with edge weights (type 3) costs on average a bit more than four bytes per edge for storing the edge weight property, plus an eight byte forward pointer.

Let V be the number of vertices in the graph, r the streaming rate in edges per second, t the sliding window length in seconds, and a the advance interval in seconds. The total number of edges in the sliding window is then $E = rt$, the number of snapshots is $S = t/a$, and each snapshot consists of ra edges. The memory size of a snapshot in the sliding window is the sum of the sizes of COW-ed vertex table pages, edges, and next pointers. If η is the average fraction of vertices updated in a snapshot, and these vertices are stored on fraction π of vertex table pages, the size of a snapshot is $16\pi V + 8ra + 16\eta V = 16V(\pi + \eta) + 8ra$ bytes. If we replace the vertex table by a sparse representation, in which each vertex is stored as an 8 byte ID and a 16 byte payload, the memory cost is $24\eta V + 8ra + 16\eta V = 40\eta V + 8ra$. Note that $\pi \geq \eta$ and that these per-snapshot probabilities are inversely related to the number of snapshots S.

SLOTH maintains S snapshots available for the graph computation, out of which it keeps only one vertex table and converts the rest to the sparse representation. There is also one snapshot that is concurrently build by the loading thread, plus up to ra new edges in a buffer. An edge in the buffer is represented as two 8 byte vertex IDs.

The memory cost of SLOTH is then:

$$
\begin{aligned}
&2(16V(\pi+\eta)+8ra)+(S-1)(40\eta V+8ra)+16ra \\
&\quad = 32V(\pi+\eta)+16ra+40\eta V(S-1)+8ra(S-1)+16ra \\
&\quad = V\left(32(\pi+\eta)+40\eta(S-1)\right)+ra(8S+24)
\end{aligned}
$$

This is the variable part of the memory cost on top of the aforementioned fixed component of the resident set size (RSS) for the basic (type 1) sliding window, and it is also the upper bound for the deduplicated sliding window without weights (the actual size depends on the number of duplicates). Weighted sliding windows cost additional $(S+1)\times 12(ra+2\eta V)$ bytes for weights (4 bytes per edge) and forward pointers (8 bytes per edge). $2\eta V$ accounts for the wasted space "parallel" to the space occupied by next pointers in the corresponding edge table.

Recall that the size of single-snapshot SLOTH is $32V + 32rt$ for the unweighted cases. After substituting $t = Sa$, we get $32V + 32raS$. Observe that the first component of the sum is more important than the other for short sliding windows, but the latter is more dominant for longer windows. For example, the Twitter graph has $V = 41.7 \times 10^6$ vertices, while streaming at $100,000$ edges per second and keeping a window of 120 seconds results in $raS = rt = 12 \times 10^6$.

By rearranging the corresponding expression for multiversioned sliding windows we get:

$$
\begin{aligned}
&V\left(32(\pi+\eta)+40\eta(S-1)\right)+ra\left(8S+24\right) \\
&\quad = 32V\left((\pi+\eta)+1.25\eta(S-1)\right)+32ra\left(0.25S+0.75\right) \\
&\quad = 32V\left(\pi+1.25\eta S-0.25\eta\right)+32raS\left(0.25+\frac{0.75}{S}\right)
\end{aligned}
$$

By comparing it to the cost of single-snapshot windows, $32V + 32raS$, we see that multiversioned sliding windows store fewer edge data when $S > 1$ (right components of the sums), but the com-

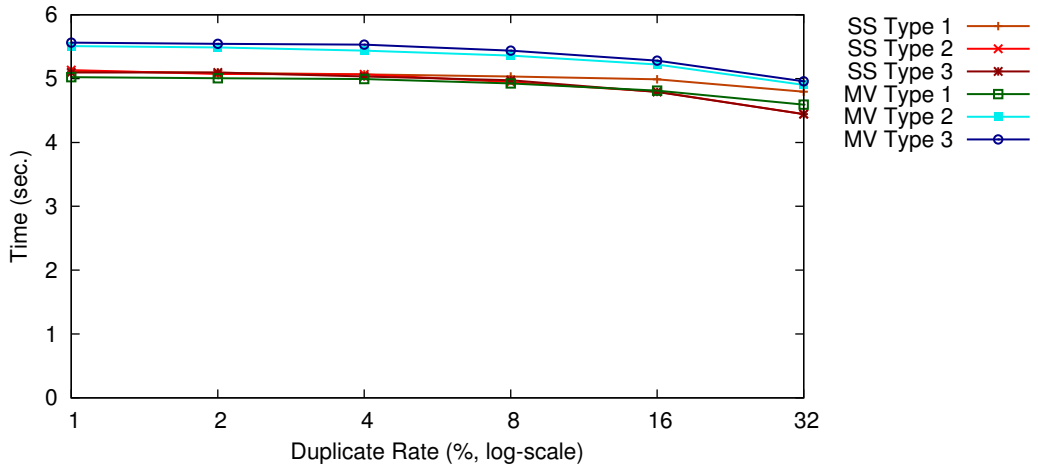


Figure 7.13: PageRank Performance versus the Duplicate Rate on the three types of single-snapshot sliding windows and the corresponding three types of multiversioned windows.

parison in terms of vertex data and next pointers is non-trivial (left components). The result of the comparison depends on S, π, and η. If the fraction of updated vertices, as modeled by π and η, is sufficiently small, multiversioned sliding windows are more likely to be memory-efficient. If the fraction of updated vertices is large – and this is indeed a common case – multiversioned windows are more likely to consume more memory especially when the number of snapshots S is small, such as it is the case when the sliding window is short or the advance interval is long.

In our experiments, SLOTH's resident set size with multiversioned sliding windows is typically up to 35% larger than for single-snapshot windows.

7.3.4 Sliding Windows in the Presence of Duplicates

Next, we analyze the impact of duplicates on the different types of sliding windows. We again use the Twitter graph [42], but we artificially introduce duplicate edges with probabilities ranging from 1% to 32%. We fix the streaming rate at 100, 000 edges per second, the window size at 90 seconds,

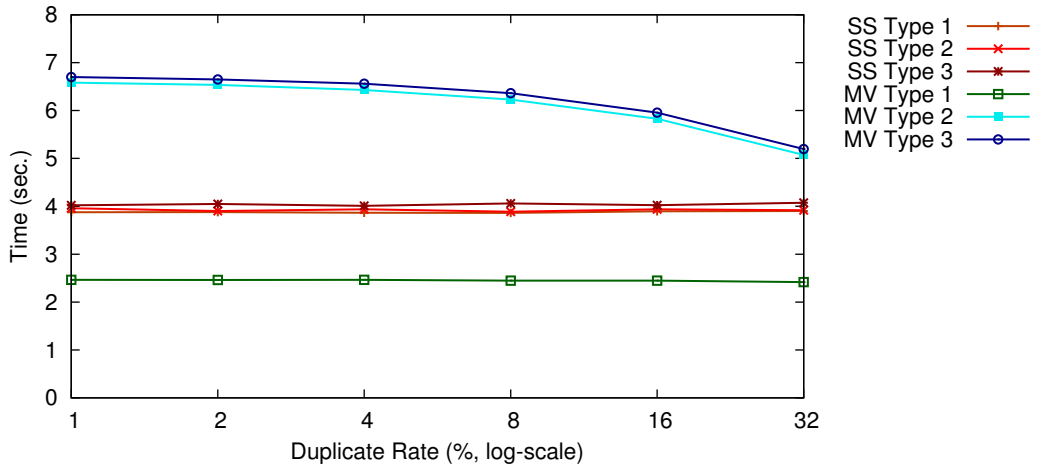


Figure 7.14: The Cost of Advancing versus the Duplicate Rate on the three types of single-snapshot sliding windows and the corresponding three types of multiversioned windows.

and the advance interval at 10 seconds, which are in the middle range of the evaluated parameters in the previous two sections.

Figure 7.13 shows the performance of PageRank as a function of the duplicate rate. The performance improves slowly for all sliding window types. The improvement is more significant for deduplicated sliding windows, where the graph algorithm has fewer edges to process. The impact on sliding windows that do not explicitly deal with duplicates is small but not negligible. A higher duplicate rate results in edges belonging to fewer vertices and consequently also in longer adjacency lists, which results in better cache performance.

Figure 7.14 shows the cost of advancing the sliding window as a function of the duplicate rate. The duplicate rate does not affect the performance of single-snapshot sliding windows, because loading them into a snapshot involves sorting and scanning through the same amount of data, regardless of whether duplicates are eliminated at the end. The duplicate rate likewise does not affect the performance of the basic multiversioned sliding window, because it does not deal with duplicates. The higher duplicate rate improves the advancing cost only in the case of the two deduplicated

multiversioned sliding windows, because a higher duplicate rate lowers the number of edges that are examined when checking for duplicates.

7.4 Summary

SLOTH enables efficient graph computation on different types of sliding windows over a time-series of edges by 1) periodically building CSR from the series of edges or by 2) maintaining a multiversioned sliding window. Both of these implementations demonstrate that LLAMA is a good building block for sliding window computation.

8

Related Work

The primary contributions of this dissertation are in the areas of graph analytics (LLAMA, Chapters 3-5) and streaming (SLOTH, Chapter 7). We present the related work in these areas to illustrate how our work fits in the existing landscape, how it is different from other work, and what its contributions are. We will then complete the picture of graph analytics landscape by presenting a brief overview of graph databases, which optimize for a different design point by focusing pri-

marily on efficient OLTP queries instead of whole-graph (OLAP-style) analysis. Graph algorithms, domain-specific languages (DSL's), and indexing are outside of the scope for this work, so we mention them only briefly when relevant.

8.1 Graph Analytics

As mentioned in the Introduction, we categorize graph analytics solutions into the following main three categories:

- Distributed platforms
- Single-machine shared-memory systems
- Single-machine out-of-memory systems

We examine each of these in more detail, after which we will briefly describe multiversioned graph stores.

8.1.1 Distributed Analytics

The first approach to large scale graph analytics requires that data fit in memory – either on a single large memory machine or distributed across the memory of multiple machines. We discuss the former in the next section and focus on the latter here. LLAMA solves the problem of computing on large graphs using an SSD instead of distributing them across multiple machines, but one could, in principle, build a distributed system on top of LLAMA that would allow distribution of the graph over RAMs and SSDs of multiple networked machines. Most of the following discussion is thus orthogonal to our work.

Pregel[47] is one of the first distributed approaches for graph analytics, designed to run large computational jobs on a cluster of commodity machines. It uses *vertex-centric computation*, which proceeds as follows: The computation consists of a sequence of iterations, called supersteps. During

each superstep, Pregel calls a user-defined function for each vertex. The function can read messages that it received during the previous superstep, modify the state of the vertex, mutate the graph topology, and send messages to other vertices that will be delivered to them at the beginning of the next superstep. Pregel thus implements the bulk-synchronous parallel (BSP) model. A vertex generally knows only about its outgoing edges, but it can also learn about other vertices. Pregel further supports global aggregators. The computation stops when all vertices declare themselves inactive. GPS[26], Giraph[23], and Apache Hama[28] are open-source Pregel implementations.

We could, in theory, run such computations on LLAMA. LLAMA provides a mutable, multi-versioned graph representation, so it would be possible for vertex programs to mutate the graphs and let the changes be visible only during the next superstep. The persistent representation could be leveraged for checkpointing the computation. We found in practice that while we do not write vertex programs explicitly, thinking in terms of vertex-centric computation is often helpful when writing efficient LLAMA programs with good cache locality.

GraphLab[45,46] is a distributed analytics platform primarily targeting machine learning and data mining applications. Its computational model is similar to Pregel's[47], except that it additionally supports looser consistency guarantees. LLAMA programs can be written in this model as well. Most of GraphLab's contributions are orthogonal to our work, especially with regard to scheduling, locking, and distribution. PowerGraph[24] runs more restricted gather-and-scatter (GAS) vertex programs, improving on GraphLab by sharding the graph by edges rather than vertices.

PEGASUS[38,39] and GBase[36,37] are early examples of Hadoop-style graph processing frameworks. PEGASUS implements highly optimized Generalized Iterated Matrix-Vector multiplication (GIM-V), which is a building block for many common graph analytics algorithms (such as PageRank[57], random walk, or connected components analysis), on top of the Hadoop framework. PEGASUS decomposes the adjacency matrix block-wise instead of using traditional row-wise (or column-wise) decomposition, such as that employed by CSR. Decomposing the graph into blocks enables the

system to store only one set of files to answer queries regardless of whether they require in- or out-edges. While this works well for whole-graph computation, it is inefficient for point-queries, since it breaks each adjacency list into a large number of fragments.

GBase[36,37] likewise uses generalized matrix-vector multiplication and computes on an adjacency matrix decomposed into blocks, but it compresses them using relatively heavy-weight compression techniques. This significantly reduces the size of the graph and results in significant speedups. Such compression methods reduce the I/O cost in exchange for CPU time, which is effective in I/O-bound scenarios, such as when loading data from a distributed file system. However, single-node computation is, in our experience, frequently CPU bound even when running out-of-core. It is nonetheless possible that compression in LLAMA would speed up I/O bound graph computation, such as when the graph algorithm produces a suboptimal random access pattern or when it uses only a small part of each disk block that it loads on graphs that are significantly larger than memory.

GraphX[25,67] combines graph-parallel and data-parallel computation in a single system, implementing both Pregel/GraphLab-style graph processing and MapReduce data processing on top of Apache Spark[68], a MapReduce-like data parallel engine that computes on in-memory resilient distributed datasets (RDDs). GraphX's API enables its users to view the data as a graph or as collections of vertices and their properties, edges and their properties, or triples and the corresponding vertex and edge properties. Users can run MapReduce tasks on these collections, but they are not necessary in LLAMA, because it is not distributed, and its programming model allows its users to directly access and iterate over vertices, edges, and triples, together with the corresponding properties. Other features of GraphX, such as join optimizations or data compression, are orthogonal to our work.

8.1.2 Single Machine Shared Memory Systems

Galois[53], Grace[60], Green-Marl[32], and Ligra[65] are a few recent examples of graph analytics systems that target primarily in-memory analysis. Many of their contributions, such as Galois' fine-grained priority scheduler and Ligra's automatic switching between different implementations of operators, are orthogonal to our work. Boost Graph Library (BGL)[6] and the SNAP library[66] include popular open-source CSR implementations.

The Connectivity Server[7] is an early – if not first – implementation of log-based CSR (Section 2.1.2). It also implements an efficient property store optimized for storing URLs, which dictionary-encodes all strings and stores each string in the dictionary as a delta from the previous entry; this contribution is orthogonal to our work.

Green-Marl[32] is a domain specific language (DSL) for graph analytics. Its compiler optimizes to graph algorithms written in the DSL and produces efficient, parallel C++ code. The resulting program runs on a standard in-memory CSR. The Green-Marl language, compiler, and its optimizations are orthogonal to our work.

Grace[60] implements mutable CSR by treating it as a log (Section 2.1.2), which we represented in LLAMA as performance-optimized design (Section 4.3.2) and determined that it has a significant memory overhead but good performance when the graph is smaller than memory. LLAMA instead uses a space-optimized design, which enables it to scale better with the number of updates at minimal runtime overhead. Grace improves multi-core analytic performance by partitioning the graph to allow simpler concurrency and parallelization between multiple workers. Within each partition, it groups edges according to the partitions of the destination vertices and sorts edges within each group by the source vertices, which improves the cache behavior of propagating updates along the edges during an iterative graph computation. Its placement mechanism strives to place proximate vertices near each other within a partition to further improve cache behavior. Grace's partitioning

and placement mechanisms are orthogonal to our work and could be probably used to improve LLAMA's multi-core performance.

Galois[53] is a general-purpose implementation of an amorphous data-parallelism[59] (a generalized form of data-parallelism), in which a graph algorithm is decomposed to tasks (activities) scheduled to run on "active" nodes. This programming model subsumes both vertex-centric and edge-centric models. Galois implements an efficient, fine-grained priority scheduler, because tasks in graph algorithms are often at most on the order of a few hundred instructions. It uses a sequence of concurrent bags distributed in a machine-topology-aware way that minimizes the coherence traffic between cores and between the packages of cores. Bags correspond to different priority levels and tasks within each bag have the same priority and can be executed in any order. The scheduler is not required to execute tasks strictly in the priority order. This results in some wasted work (such as pursuing a higher-cost path in SSSP before some lower-cost paths), but it is outweighed by the performance improvements resulting from a lower scheduling overhead. Galois programming model and scheduler are orthogonal to our work.

Ligra[65] efficiently executes algorithms decomposed into applying functions to subsets of vertices and edges, and many common graph algorithms indeed naturally decompose in such ways. Its vertex-map function takes a user-specified vertex function that returns a boolean, applies it to the specified set of vertices, and returns a subset of vertices for which the user-provided function returned true. The edge-map function applies a user-specified function to all outgoing edges from the given set of source vertices and returns a set of destination vertices for edges for which the function returned true. Ligra does not enforce consistency by ordering updates, but it instead requires the user-specified functions to do their own synchronization using atomic instructions, such as compare-and-swap. An implementation of a graph algorithm in Ligra can switch between out-edges and in-edges by swapping its data structures that represent out- and in-edges. Ligra contains two types of implementations of the vertex-map and edge-map functions, one for dense vertex sets and

the other for sparse sets, and it automatically switches between the two based on a simple threshold. Ligra's programming model and the automatic switching of implementations of graph operators are orthogonal to our work.

8.1.3 Single Machine Out-of-Core Systems

LLAMA falls in this category of graph analytics systems, but unlike many of them, it supports mutability, incremental ingest, and multi-version access.

GraphChi[43] is one of the first analytic systems able to process out-of-memory graphs using commodity hardware. It uses a vertex-centric computation model similar to GraphLab. GraphChi partitions a graph into shards, each of which is small enough to fit in memory and each of which contains all the edges with the same set of target vertices within the corresponding interval. It processes shards one at the time using a Parallel Sliding Window (PSW), which results in a small number of large sequential I/Os, enabling it to work well with both spinning disks and SSDs.

GraphChi and LLAMA employ two different design philosophies: GraphChi reads the subgraph from disk in an order that results in a mostly sequential I/O and pushes it to the user-provided vertex program. The graph computation iterates over the entire graph until a stopping criterion has been reached. In contrast, LLAMA programs pull data on demand, leaving the program the ability to express locality in access, but not requiring it. That being said, writing LLAMA programs with sequential access and good cache behavior is no more difficult than doing so for GraphChi. The advantage of pulling data instead of pushing is that it makes it easy to implement efficient point-queries and algorithms that operate on only a part of a graph. The user programs to can easily access only the data that they need without the overhead of a fine-grained scheduler. LLAMA uses `mmap` to access data, which incurs less overhead than GraphChi's explicit POSIX I/O, especially when data fits almost entirely in memory.

TurboGraph[30] improves on GraphChi by extracting more parallelism, leveraging I/O parallelism in SSDs, and overlapping CPU processing with I/O. It likewise uses a vertex-centric computational model, but it uses generalized matrix-vector multiplication[36,37] instead of typical vertex programs. TurboGraph implements the Pin and Slide computational model, which proceeds as follows: At the beginning of a superstep, it determines which pages are needed to perform the computation and schedules them to be read asynchronously. As soon as pages are read, TurboGraph computes on them and frees them again, making space for more pages to be read.

Like LLAMA, TurboGraph programs do not place any artificial restrictions on the processing order of vertices within a page. Unlike TurboGraph, LLAMA does not explicitly overlap CPU processing and I/O, but considering that most LLAMA programs exhibit sequential access, the operating system's read-ahead can easily predict which pages are needed next. TurboGraph decreases its buffer management overhead by pinning whole pages and processing them instead of pinning individual adjacency lists, which we found to be very expensive in our experimentations. LLAMA solves this issue using `mmap` and thus letting the operating system to take care it, instead of pinning down pages explicitly like TurboGraph.

While both LLAMA and TurboGraph support updates, they do so differently. While LLAMA implements updates using multiversioned CSR, TurboGraph leaves some empty space in its data pages for new edges, thus making the data structure less compact. If TurboGraph needs to insert more edges than will fit on a page, it splits the page and adds pointers to the original page for the moved records, so that they can be accessed using their original IDs.

X-Stream[63] uses an edge-centric, gather-and-scatter model that takes as input a binary formatted edge-list requiring no preprocessing. It shards the graph so that all of the vertex data fit in the "fast" memory (RAM or even just the CPU cache). Each superstep proceeds in two rounds. First, the X-Stream program processes the edges sequentially, producing a stream of updates. The system then sorts the updates by the corresponding vertex ID and aggregates them. This makes X-Stream

well-matched for whole-graph analysis, but less well-suited for point queries. In contrast, LLAMA efficiently supports both point queries and whole-graph analysis.

Lin et al.[44] argued for, and demonstrated the feasibility of, using a simple `mmap`-based approach for graph processing, in which the program iterates over an `mmap`-ed binary edge list file. They achieved execution times far superior to both GraphChi and TurboGraph. LLAMA uses `mmap` in a similar fashion, but it improves upon it by using a mutable compressed sparse row representation that supports efficient incremental ingest and point-queries, while sharing a simple programming model. CSR occupies approximately only half of the space of an edge list – which means that more of the graph fits in memory, producing significantly less I/O, so we expect to perform noticeably better on larger graphs. MDB[13] (later renamed to LMDB) demonstrates similar orders-of-magnitude performance improvement in the realm of key-value stores.

Pearce et al.[58] demonstrated efficient graph traversal, SSSP, and connected components analysis on graphs larger than memory using semi-sorted, multithreaded visitor queues to extract parallelism across both the CPU and the SSD. The relaxed vertex processing order causes these algorithms to do some redundant work, but it is outweighed by the overall benefits of parallelism. Pearce's contributions are orthogonal to our work.

8.1.4 Multiversion Graph Stores

DeltaGraph[40] is a distributed index structure that stores the entire history of a graph and enables efficient retrieval of multiple graphs from arbitrary points in time in addition to maintaining the current state of the graph. It compactly represents snapshots as sets of deltas and event-lists, but the user must explicitly retrieve a snapshot from the system before querying it. In contrast, all snapshots in LLAMA are immediately queryable.

Like DeltaGraph, Chronos[29] optimizes for efficient queries across snapshots by collocating dif-

ferent versions of vertices. Chronos focuses on optimizing queries comparing multiple snapshots, while LLAMA optimizes for efficient queries on individual snapshots and for incremental ingest.

8.2 Graph Databases

Graph databases optimize for a different design point than graph analytic systems, focusing more on point queries without specializing on whole-graph analysis. Many graph databases have at least partial support for ACID transactions. Most contributions in this space are orthogonal to our work.

The space of graph databases is large, especially if we include RDF stores. We limit our discussion to the most relevant systems.

Neo4j[52] is perhaps the most widely used disk-based graph database, featuring full ACID support. Similar to LLAMA, Neo4j does not guarantee that edges belonging to the same vertex are stored contiguously on disk and uses next pointers to link them, but it does so at the granularity of individual edges instead of adjacency list fragments. It also does not sort or cluster edges in any way, unlike LLAMA, which guarantees that edges are stored in source ID order within a snapshot. Neo4j's on-disk format is thus not well-suited for whole-graph analysis. Its in-memory format stores adjacency lists in separate arrays, similarly to LLAMA's write-optimized graph store.

DEX[48,49] (now Sparksee) is a high-performance graph database built on top of compressed bitmap indices[48] (Section 2.3). It leverages this data structure to implement efficient set operators on adjacency lists and sets of vertices, such as union and intersection. Adjacency list fragments in LLAMA can be sorted if configured by the user, which likewise enables efficient set computations, even if they might not be as efficient as on compressed bitmaps. We leverage this, for example, in our triangle counting implementation.

Trinity[64] is a distributed graph database that stores data in a RAM cloud. Each vertex is represented as a cell combining its adjacency list, properties, and temporary data specific to the given

graph algorithm. This represents a row-store-like approach, unlike the column-store-like approach used by LLAMA.

InfiniteGraph[33] is an example of a distributed graph database that uses persistent storage. Turning LLAMA to a distributed system that spreads the graph across memories and SSDs of the cluster is interesting future work.

HyperGraphDB[34] features support for hypergraphs, in which an edge can connect more than two vertices, thus providing richer semantics for edges than LLAMA. OrientDB[56] and CouchDB[51] are examples of distributed document-graph databases, in which vertices are structured documents with references to other documents. The documents are thus first-class entities, providing richer semantics than LLAMA's vertex properties.

8.3 Sliding Windows in Graphs

Sliding windows in graphs is a relatively young field. Crouch et al.[16] carried out the first (and to the best of our knowledge, the only) theoretical study of sliding windows, developing several single-pass streaming algorithms that compute on a sliding window of the last L edges in the stream, but without materializing it. Instead of explicitly maintaining the sliding window, they only maintain data in memory specific to the given algorithm. Most of their algorithms run in $O(n \text{ polylog } n)$ space, where n is the number of nodes in the graph. SLOTH instead maintains the sliding window explicitly and uses normal, unmodified graph algorithms. A sliding window in SLOTH occupies $O(L)$ bytes, which is often in practice less than or comparable to $O(n \text{ polylog } n)$.

In the rest of the section, we survey approaches for maintaining sliding windows explicitly. We first focus on methods specifically tailored for graphs, followed by a brief discussion of generic sliding windows and their applicability to graph analytics.

8.3.1 Sliding Window Management for Graphs

Kineograph[12] proposes (but to the best of our knowledge, does not yet implement) a straightforward approach for maintaining a sliding window, in which they maintain $n + 1$ graphs to maintain a sliding window of n snapshots, plus 1 for the snapshot that is currently being built. Then they propose computing on a weighted combination of these n snapshots. This approach enables them to efficiently add new data and drop old data, but at the cost of having to compute over the union of multiple snapshots – thus requiring the system to examine all n snapshots each time to get a complete adjacency list. Like SLOTH, Kineograph would enable incremental computation by updating the computation results based on recent changes in the graph reflected in new snapshots.

STINGER[4] makes the opposite design decision: The entire sliding window is stored in a single data structure to speed up computation, but at the cost of doing more work when ingesting and aging out data. STINGER stores different adjacency lists in separate growable arrays and timestamps each edge, allowing the graph algorithm to compute just on the correct set of edges, thus providing a fine-grained method of defining the sliding window. Adding new edges requires traversing the corresponding adjacency list to find an empty slot for the new edge. Deleting old edges requires a scan of the entire data structure.

STINGER can also check if a new edge is a duplicate, in which case it increases its weight. But unlike LLAMA, STINGER does not store old values of edge weights, so it cannot properly age them out. Both adding and deleting edges is an embarrassingly parallel problem[20], suitable especially for massive shared memory multiprocessors.

We found in our preliminary study (Section 2.5) that storing adjacency lists in separate arrays is slower than using the compressed sparse row representation, which is our motivation behind SLOTH. Ingesting edges into SLOTH requires computing parallel prefix sums, but unlike STINGER, we need to examine the adjacency lists only if we need to check for duplicate edges.

8.3.2 Generic Sliding Windows

Sliding window implementations in relational stream processing systems are implemented as traditional row-store tables optionally augmented with unclustered indexes (such as SteM in TelegraphCQ[11]) or as a clustered B-tree (Aurora[2], now commercialized as StreamBase). In either case, using such an implementation for a sliding window of edges requires an index to efficiently determine the list of adjacent edges to a given vertex. Deleting old data requires a sequential scan over the entire data structure. In the case of B-trees, adding, deleting, and querying a vertex has time complexity $O(\log n)$ rather than $O(1)$, which is typical for graph analytics frameworks.

9

Conclusion

We demonstrated that a modified compressed sparse row (CSR) representation is indeed a good building block for graph analytic applications requiring mutability, persistence, and/or computation using sliding windows. LLAMA and SLOTH are two systems that embody this design, and both deliver competitive performance and functionality across a broader spectrum of use cases than other systems.

We compared several approaches to implementing updates in CSR in a single system, LLAMA, to eliminate as many confounding variables as possible. We showed that a variant of multiversioned, space-optimized CSR, which stores differences between multiple versions of adjacency lists and links them using next pointers, provides the best balance between performance and memory usage. It provides only marginal performance and memory overheads over immutable CSR when the graph is stored entirely in a single snapshot, and both overheads increase slowly with each additional snapshot.

When the graph fits in memory, this representation performs marginally more slowly than the best state of the art mutable CSR representation, which treats CSR as a log, and whose implementation in LLAMA we call the performance-optimized design, but at only a fraction of its memory usage. A space-optimized design without next pointers consumes even less memory than the variant with next pointers, but it has higher runtime cost. All three representations perform better and have smaller memory cost than adding a write-optimized delta map to a standard CSR representation, which is another state of the art approach to making CSR mutable. Parameterized (hybrid) designs that write updated adjacency lists or store differences based on simple policies do not generally improve upon the space-optimized design, and if so, only marginally and only in some cases.

The space-optimized design significantly outperforms the performance-optimized design and all hybrid designs when the graph is larger than memory. We implement efficient larger-than-memory graph computation using memory mapping, in which we delegate the buffer management to the operating system. This saves us both the cost of checking whether a page is loaded in memory before accessing it and loading it if it is not and the cost of pinning pages and implementing an eviction policy. We demonstrated that this performs well on an SSD when we process vertices in ID order, which is a common access pattern in many graph analytics algorithms. Processing vertices in order on multiversioned CSR results in parallel sequential access over the different snapshots, which has good cache behavior and is efficient when executed on an SSD. Memory mapping further enables us

to devote almost the entire CPU time to graph computation, which is important considering that many instances of out-of-core computation are, in fact, CPU bound.

Finally, the space-optimized design enables efficient multiversioned sliding window maintenance. We advance a sliding window by creating a new snapshot with new data, and age out old data by deleting the oldest snapshot. We also showed that we can easily extend the sliding window to maintain edge weights instead of having duplicate edges.

9.1 Future Work

The compressed sparse row representation is a good starting point for mutable graph representations, out-of-core execution, and sliding windows. It also has potential in other scenarios, such as OLTP-like workloads and distributed systems, because it delivers better performance than other graph data structures, giving us enough "headroom" to implement additional features on it and still potentially outperform the state of the art. It is also possible that we can further improve the out-of-core performance of CSR using compression.

9.1.1 Graph Compression in CSR

The compressed sparse row representation is already more memory efficient than many alternatives, such as edge lists or storing adjacency lists in separate arrays, but it can be made even smaller by employing variable-length integer encoding techniques or more heavy-weight compression methods. It is possible that compression can improve the analytic performance, especially in out-of-core situations, even though graph computation is already CPU intensive. Decreasing the size of the data structure will increase the fraction that fits in memory, decreasing I/O.

The encoding scheme needs to be sufficiently cheap, so that it will not cost more than the savings obtained from manipulating smaller data sizes. Examples of relatively light-weight compres-

sion schemes are null-suppression, delta-encoding, and compressed bitmaps[48]. We could also offset the cost of decompression by computing directly on the compressed graph. For example, DEX[48] computes unions and intersections of node-sets directly on its compressed representation. We also expect compression to speed up cases in which graph computation is I/O-bound, such as when the graph algorithm produces a random access pattern or when it uses only a small fraction of disk blocks that it loads, especially if the graph is significantly larger than memory.

9.1.2 CSR-Like Write-Optimized Graph Store

While CSR is normally not suitable for updates, perhaps we can turn it into a write-optimized graph data structure with good analytic performance. This would, for example, enable us to use the same code for both write- and read-optimized snapshots, saving the cost of conditional branches in iterators that execute the appropriate code depending on the representation they reference (an iterator in the write-optimized store also iterates over the unmodified edges in the read-optimized store).

One idea is to leave space for new edges instead of tightly packing the adjacency lists, and when we run out of space, allocate a new block of edges at the end of the edge table, using next pointers to link them. We can allocate larger blocks for new edges each time we run out of space. We could handle deletions either by shifting contents within the same block of edges, using a deletion vector, or by replacing a deleted edge by a special value. Such an approach would be more compact than storing adjacency lists in separate growable arrays, but less compact than CSR, and it would enable us to reuse the same code for both read-optimized and write-optimized CSR.

Developing an efficient write-optimized store would also enable us to build a graph database on top of CSR supporting both efficient OLAP and OLTP queries. We can easily guarantee durability of transactions in the write-optimized store using a write-ahead log and creating read-optimized snapshots when checkpointing. We can use locking or multiversion concurrency control to implement the remaining ACID properties.

9.1.3 Efficient Distributed System

Increasing single-node performance and enabling the system to efficiently process graphs larger than memory is a fundamental building block upon which we can build a faster distributed system. Better single-node performance will expose the network overhead, but we can decrease the number of network messages by simply letting the individual nodes work on graph partitions larger than memory.

We expect the overall analytic performance of the distributed system to increase with the number of nodes as the work is spread out over a larger number of compute resources, but it will later flatten out and eventually get worse as the number of network messages increases, and the cost of network communication becomes more dominant. This presents a challenge in being able to predict the optimal number of nodes for the given graph and algorithm.

Another challenge is the programming model: Distributed systems currently require a graph algorithm to be decomposed into vertex- or edge-centric programs, but according to our experiments, such frameworks have high CPU overheads. We might need to decompose the algorithm to instead work on larger chunks of the graph, which presents additional challenges both in the ability to easily express graph algorithms as well as in the framework's ability to maintain and ensure consistency of the distributed state of the graph.

References

[1] Abadi, D., Boncz, P. A., Harizopoulos, S., Idreos, S., & Madden, S. (2013). The design and implementation of modern column-oriented database systems. *Foundations and Trends in Databases*, 5(3), 197–280.

[2] Abadi, D. J., Carney, D., Çetintemel, U., Cherniack, M., Convey, C., Lee, S., Stonebraker, M., Tatbul, N., & Zdonik, S. (2003). Aurora: a new model and architecture for data stream management. *The VLDB Journal*, 12(2), 120–139.

[3] Backstrom, L., Huttenlocher, D., Kleinberg, J., & Lan, X. (2006). Group formation in large social networks: membership, growth, and evolution. In *SIGKDD* (pp. 44–54).

[4] Bader, D. A., Berry, J., Amos-Binks, A., Chavarría-Miranda, D., Hastings, C., Madduri, K., & Poulos, S. C. (2009). *STINGER: Spatio-Temporal Interaction Networks and Graphs (STING) Extensible Representation*. Technical report, Georgia Institute of Technology.

[5] Barabási, A. L. & Albert, R. (1999). Emergence of scaling in random networks. *Science*, 286, 509–512.

[6] BGL (2014). The Boost graph library (BGL). http://www.boost.org/doc/libs/1_57_0/libs/graph/doc/index.html.

[7] Bharat, K., Broder, A. Z., Henzinger, M. R., Kumar, P., & Venkatasubramanian, S. (1998). The connectivity server: Fast access to linkage information on the web. *Computer Networks and ISDN Systems*, 30(1-7), 469–477.

[8] Bloom, B. H. (1970). Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7), 422–426.

[9] Buluç, A., Gilbert, J., & Shah, V. B. (2011). 13. Implementing sparse matrices for graph algorithms. In *Graph Algorithms in the Language of Linear Algebra* (pp. 287–313). Society for Industrial and Applied Mathematics.

[10] Chakrabarti, D., Zhan, Y., & Faloutsos, C. (2004). R-MAT: A recursive model for graph mining. In M. W. Berry, U. Dayal, C. Kamath, & D. B. Skillicorn (Eds.), *SDM*: SIAM.

[11] Chandrasekaran, S., Cooper, O., Deshpande, A., Franklin, M. J., Hellerstein, J. M., Hong, W., Krishnamurthy, S., Madden, S., Raman, V., Reiss, F., & Shah, M. A. (2003). TelegraphCQ: Continuous dataflow processing for an uncertain world. In *CIDR*.

[12] Cheng, R., Hong, J., Kyrola, A., Miao, Y., Weng, X., Wu, M., Yang, F., Zhou, L., Zhao, F., & Chen, E. (2012). Kineograph: taking the pulse of a fast-changing and connected world. In P. Felber, F. Bellosa, & H. Bos (Eds.), *EuroSys* (pp. 85–98).: ACM.

[13] Chu, H. (2012). MDB: A memory-mapped database and backend for OpenLDAP. In *UKUUG*.

[14] Comer, D. (1979). The ubiquitous b-tree. *ACM Comput. Surv.*, 11(2), 121–137.

[15] Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2001). *Introduction to algorithms*. The MIT press, second edition edition.

[16] Crouch, M. S., McGregor, A., & Stubbs, D. (2013). Dynamic graphs in the sliding-window model. In H. L. Bodlaender & G. F. Italiano (Eds.), *ESA*, volume 8125 of *Lecture Notes in Computer Science* (pp. 337–348).: Springer.

[17] DeBrabant, J., Pavlo, A., Tu, S., Stonebraker, M., & Zdonik, S. B. (2013). Anti-caching: A new approach to database management system architecture. *PVLDB*, 6(14), 1942–1953.

[18] DOMO (2014). Data never sleeps 2.0. http://www.domo.com/learn/data-never-sleeps-2.

[19] Dongarra, J., Koev, P., Li, X., Demmel, J., & van der Vorst, H. (2000). *10. Common Issues*, chapter 10, (pp. 315–336). SIAM.

[20] Ediger, D., McColl, R., Riedy, E. J., & Bader, D. A. (2012). STINGER: High performance data structure for streaming graphs. In *HPEC* (pp. 1–5).: IEEE.

[21] Erdös, P. & Rényi, A. (1959). On random graphs. *Publications Mathematicae*, 6, 290.

[22] Facebook (2013). Facebook's growth in the past year. https://www.facebook.com/media/set/?set=a.10151908376636729.1073741825.20531316728&type=3.

[23] Giraph (2014). Welcome to Apache Giraph! http://giraph.apache.org.

[24] Gonzalez, J. E., Low, Y., Gu, H., Bickson, D., & Guestrin, C. (2012). PowerGraph: Distributed graph-parallel computation on natural graphs. In *OSDI* (pp. 17–30). Berkeley, CA, USA: USENIX Association.

[25] Gonzalez, J. E., Xin, R. S., Dave, A., Crankshaw, D., Franklin, M. J., & Stoica, I. (2014). Graphx: Graph processing in a distributed dataflow framework. In *OSDI* (pp. 599–613). Broomfield, CO, USA: USENIX Association.

[26] GPS (2014). A graph processing system. http://infolab.stanford.edu/gps.

[27] graph500 (2010). Graph 500 benchmark. http://www.graph500.org/specifications.

[28] Hama (2014). A general BSP framework on top of Hadoop. https://hama.apache.org.

[29] Han, W., Miao, Y., Li, K., Wu, M., Yang, F., Zhou, L., Prabhakaran, V., Chen, W., & Chen, E. (2014). Chronos: A graph engine for temporal graph analysis. In *EuroSys*: ACM.

[30] Han, W.-S., Lee, S., Park, K., Lee, J.-H., Kim, M.-S., Kim, J., & Yu, H. (2013). TurboGraph: a fast parallel graph engine handling billion-scale graphs in a single pc. In I. S. Dhillon, Y. Koren, R. Ghani, T. E. Senator, P. Bradley, R. Parekh, J. He, R. L. Grossman, & R. Uthurusamy (Eds.), *KDD* (pp. 77–85).: ACM.

[31] Harizopoulos, S., Abadi, D. J., Madden, S., & Stonebraker, M. (2008). OLTP through the looking glass, and what we found there. In J. T.-L. Wang (Ed.), *SIGMOD* (pp. 981–992).: ACM.

[32] Hong, S., Chafi, H., Sedlar, E., & Olukotun, K. (2012). Green-Marl: a DSL for easy and efficient graph analysis. In *ASPLOS* (pp. 349–362).: ACM.

[33] InfiniteGraph (2014). Distributed graph database. http://www.objectivity.com/infinitegraph.

[34] Iordanov, B. (2010). HyperGraphDB: A generalized graph database. In *WAIM Workshops*, volume 6185 of *Lecture Notes in Computer Science* (pp. 25–36).: Springer.

[35] Jagadish, H. V., Narayan, P. P. S., Seshadri, S., Sudarshan, S., & Kanneganti, R. (1997). Incremental organization for data recording and warehousing. In *VLDB* (pp. 16–25).: Morgan Kaufmann.

[36] Kang, U., Tong, H., Sun, J., Lin, C.-Y., & Faloutsos, C. (2011a). GBASE: a scalable and general graph management system. In *KDD* (pp. 1091–1099).: ACM.

[37] Kang, U., Tong, H., Sun, J., Lin, C.-Y., & Faloutsos, C. (2012). GBASE: an efficient analysis platform for large graphs. *VLDB J.*, 21(5), 637–650.

[38] Kang, U., Tsourakakis, C. E., & Faloutsos, C. (2009). Pegasus: A peta-scale graph mining system. In W. W. 0010, H. Kargupta, S. Ranka, P. S. Yu, & X. Wu (Eds.), *ICDM* (pp. 229–238).: IEEE Computer Society.

[39] Kang, U., Tsourakakis, C. E., & Faloutsos, C. (2011b). PEGASUS: mining peta-scale graphs. *Knowl. Inf. Syst.*, 27(2), 303–325.

[40] Khurana, U. & Deshpande, A. (2013). Efficient snapshot retrieval over historical graph data. In C. S. Jensen, C. M. Jermaine, & X. Zhou (Eds.), *ICDE* (pp. 997–1008).: IEEE Computer Society.

[41] Korf, R. E. (1985). Depth-first iterative-deepening: An optimal admissible tree search. *Artificial Intelligence*, 27, 97–109.

[42] Kwak, H., Lee, C., Park, H., & Moon, S. (2010). What is Twitter, a social network or a news media? In *WWW* (pp. 591–600). New York, NY, USA: ACM.

[43] Kyrola, A., Blelloch, G., & Guestrin, C. (2012). GraphChi: large-scale graph computation on just a PC. In *OSDI* (pp. 31–46). Berkeley, CA, USA: USENIX Association.

[44] Lin, Z., Chau, D. H. P., & Kang, U. (2013). Leveraging memory mapping for fast and scalable graph computation on a pc. In X. Hu, T. Y. Lin, V. Raghavan, B. W. Wah, R. A. Baeza-Yates, G. Fox, C. Shahabi, M. Smith, Q. Yang, R. Ghani, W. Fan, R. Lempel, & R. Nambiar (Eds.), *BigData Conference* (pp. 95–98).: IEEE.

[45] Low, Y., Gonzalez, J., Kyrola, A., Bickson, D., Guestrin, C., & Hellerstein, J. M. (2010). GraphLab: A new framework for parallel machine learning. In P. Grünwald & P. Spirtes (Eds.), *UAI* (pp. 340–349).: AUAI Press.

[46] Low, Y., Gonzalez, J., Kyrola, A., Bickson, D., Guestrin, C., & Hellerstein, J. M. (2012). Distributed graphlab: A framework for machine learning in the cloud. *PVLDB*, 5(8), 716–727.

[47] Malewicz, G., Austern, M. H., Bik, A. J. C., Dehnert, J. C., Horn, I., Leiser, N., & Czajkowski, G. (2010). Pregel: a system for large-scale graph processing. In A. K. Elmagarmid & D. Agrawal (Eds.), *SIGMOD '10* (pp. 135–146).: ACM.

[48] Martínez-Bazan, N., Aguila-Lorente, M. A., Muntés-Mulero, V., Dominguez-Sal, D., Gómez-Villamor, S., & Larriba-Pey, J.-L. (2012). Efficient graph management based on bitmap indices. In *IDEAS* (pp. 110–119).: ACM.

[49] Martínez-Bazan, N., Muntés-Mulero, V., Gómez-Villamor, S., Nin, J., Sánchez-Martínez, M.-A., & Larriba-Pey, J.-L. (2007). Dex: high-performance exploration on large graphs for information retrieval. In *CIKM* (pp. 573–582).: ACM.

[50] Michael, M. M. (2004). Hazard pointers: Safe memory reclamation for lock-free objects. *IEEE Trans. Parallel Distrib. Syst.*, 15(6), 491–504.

[51] Mondal, J. & Deshpande, A. (2012). Managing large dynamic graphs efficiently. In *SIGMOD Conference* (pp. 145–156).: ACM.

[52] Neo4j (2014). The graph database. http://neo4j.org.

[53] Nguyen, D., Lenharth, A., & Pingali, K. (2013). A lightweight infrastructure for graph analytics. In M. Kaminsky & M. Dahlin (Eds.), *SOSP* (pp. 456–471).: ACM.

[54] O'Neil, P. E., Cheng, E., Gawlick, D., & O'Neil, E. J. (1996). The log-structured merge-tree (LSM-tree). *Acta Informatica*, 33(4), 351–385.

[55] openmp (2014). The OpenMP API specification for parallel programming. http://openmp.org/wp.

[56] OrientDB (2014). Document-graph NoSQL database. http://www.orientechnologies.com.

[57] Page, L., Brin, S., Motwani, R., & Winograd, T. (1998). *The PageRank citation ranking: Bringing order to the web*. Technical report, Stanford University.

[58] Pearce, R. A., Gokhale, M., & Amato, N. M. (2010). Multithreaded asynchronous graph traversal for in-memory and semi-external memory. In *SC* (pp. 1–11).: IEEE.

[59] Pingali, K., Nguyen, D., Kulkarni, M., Burtscher, M., Hassaan, M. A., Kaleem, R., Lee, T.-H., Lenharth, A., Manevich, R., Méndez-Lojo, M., Prountzos, D., & Sui, X. (2011). The tao of parallelism in algorithms. In *PLDI* (pp. 12–25).: ACM.

[60] Prabhakaran, V., Wu, M., Weng, X., McSherry, F., Zhou, L., & Haridasan, M. (2012). Managing large graphs on multi-cores with graph awareness. In *USENIX ATC* Berkeley, CA, USA: USENIX Association.

[61] Roche, X. (2014). A story of realloc (and laziness). http://blog.httrack.com/blog/2014/04/05/a-story-of-realloc-and-laziness.

[62] Romero, D. M., Meeder, B., & Kleinberg, J. M. (2011). Differences in the mechanics of information diffusion across topics: idioms, political hashtags, and complex contagion on Twitter. In S. Srinivasan, K. Ramamritham, A. Kumar, M. P. Ravindra, E. Bertino, & R. Kumar (Eds.), *WWW* (pp. 695–704).: ACM.

[63] Roy, A., Mihailovic, I., & Zwaenepoel, W. (2013). X-Stream: edge-centric graph processing using streaming partitions. In M. Kaminsky & M. Dahlin (Eds.), *SOSP* (pp. 472–488).: ACM.

[64] Shao, B., Wang, H., & Li, Y. (2013). Trinity: A distributed graph engine on a memory cloud. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13 (pp. 505–516). New York, NY, USA: ACM.

[65] Shun, J. & Blelloch, G. E. (2013). Ligra: a lightweight graph processing framework for shared memory. In A. Nicolau, X. Shen, S. P. Amarasinghe, & R. Vuduc (Eds.), *PPOPP* (pp. 135–146).: ACM.

[66] SNAP (2014). Stanford network analysis platform. http://snap.stanford.edu/snap.

[67] Xin, R. S., Crankshaw, D., Dave, A., Gonzalez, J. E., Franklin, M. J., & Stoica, I. (2014). Graphx: Unifying data-parallel and graph-parallel analytics. *CoRR*, abs/1402.2394.

[68] Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauly, M., Franklin, M. J., Shenker, S., & Stoica, I. (2012). Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *NSDI* (pp. 15–28). San Jose, CA, USA: USENIX Association.

[69] Zheng, D., Burns, R., & Szalay, A. S. (2012). A parallel page cache: IOPS and caching for multicore systems. In *HotStorage*: USENIX.