# LLAMA: Efficient graph analytics using Large Multiversioned Arrays

## The Harvard community has made this article openly available. **Please share** how this access benefits you. Your story matters

# LLAMA: Efficient Graph Analytics Using Large Multiversioned Arrays

Peter Macko [#*1], Virendra J. Marathe [*2], Daniel W. Margo [#3], Margo I. Seltzer [#*4]

[#] *School of Engineering and Applied Sciences, Harvard University*
*33 Oxford Street, Cambridge, MA, USA*
[1] `pmacko@eecs.harvard.edu`
[3] `dmargo@eecs.harvard.edu`
[4] `margo@eecs.harvard.edu`

[*] *Oracle Labs*
*35 Network Drive, Burlington, MA, USA*
[2] `virendra.marathe@oracle.com`

*Abstract*—We present LLAMA, a graph storage and analysis system that supports mutability and out-of-memory execution. 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. LLAMA 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 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 5x on breadth-first-search.

## I. INTRODUCTION

Graph-structured data are everywhere. From biological to social networks, from roads to the Internet, from recommendations to history, graph-structured data are growing in both size and complexity. Graph analytics have become almost synonymous with "big data", and graph data are often processed using distributed systems. Recent efforts tackle both scaling *up*, using large machines with dozens to hundreds of cores [1], and scaling *out*, using a cluster of machines each of which processes part of a dataset [2]. There are two distinct efforts: *main-memory* analytic systems require that the entire graph fit in memory [1], [3], [4], [5], and *out-of-memory* systems use persistent representations accommodating graphs that exceed memory [6], [7], [8]. The problem with main-memory systems is that they require costly shared-memory machines to process large graphs, while the problem with existing out-of-memory systems is that they frequently do not perform well when the graph fits in memory (as shown in Section V-B).

We present LLAMA (Linked-node analytics using Large Multiversioned Arrays), a system designed to provide the best of both worlds. Its performance is comparable to in-memory analytics systems when the graph fits in the memory of a single machine and comparable to or better than existing out-of-memory analytic engines on graphs that exceed memory.

LLAMA's in-memory representation is based on the compressed sparse row representation (CSR) used in most main-memory systems [9]. While most such systems operate on immutable data, LLAMA provides high-performance analytics in the presence of incremental data ingest and mutation.

LLAMA stores graphs as time series of snapshots. When first loaded, a graph resides in a single LLAMA snapshot; then each following incremental load produces another snapshot. This makes it easy to support updates, as each batch of updates produces a new snapshot; it also provides the ability to analyze graphs temporally [10], [11]. This structure is also ideal for applications that receive a steady stream of new data, but need to perform whole-graph analysis on consistent views. We leverage the virtual memory system and the parallel nature of SSDs to support out-of-memory execution.

We compare LLAMA to several of the most popular graph analytics systems, using graphs ranging from millions to billions of edges. We consider three widely used benchmarks: PageRank [12], an embarrassingly parallel vertex-centric computation; breadth-first search (BFS), which exercises graph traversal; and triangle counting, which is the building block of many graph centrality metrics. We show that LLAMA's mutability introduces modest overheads of 3–18% relative to an immutable CSR for in-memory graphs and that it significantly outperforms state-of-the-art out-of-memory systems in most cases, with a best case improvement of 5x on BFS.

The contributions of this work are:

- The design and implementation of a multi-version CSR representation that performs almost as well as a traditional CSR, while also supporting updates.
- The design and implementation of a CSR representation that works on datasets that exceed memory.
- A system that provides competitive performance for in-memory graphs.
- A system that outperforms state-of-the-art out-of-memory systems.

The rest of this paper is organized as follows. Section II describes LLAMA's architecture, focusing on its read-optimized storage, and Section III then discusses some of the more crucial design decisions. Section IV describes how to use LLAMA. Section V evaluates LLAMA's performance, comparing it to several popular graph analytic systems. Section VI places LLAMA in the broader research context, and Section VII concludes.

## II. 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 multiversioned 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 containing the newly loaded vertices and edges. A vertex's adjacency list can thus 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 is described in more detail in the following sections:

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

The vertex table is a Large Multiversioned Array or LAMA, as described in Section II-A.

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 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 a pair of incoming and outgoing edges. 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.

Figure 1 shows a simple multiversioned 3-node graph and its LLAMA representation. The first snapshot (0) contains the original graph, while the second snapshot (1) illustrates the addition of one edge and the removal of another. The vertex table (Section II-A) is shared between the two snapshots, while each snapshot has its own edge tables and deletion vectors (Section II-B).

### A. Vertex Table: Large Multiversioned Array (LAMA)

LLAMA implements the vertex table using multiversion arrays, which we call LAMAs. From the programmer's perspective, a LAMA is a mutable array with support for snapshotting. In principle one could treat snapshots as writable clones and create a branching version tree, but we currently do not.

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 we have $b$-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.

We create a snapshot by copying the indirection array. 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 change the new page as necessary. The top of Figure 1b shows a LAMA vertex table with two snapshots 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 the entire indirection table fits in the L3 cache or higher in the memory hierarchy.

*Vertex Records:* We store the following four fields for each vertex in the vertex table:

- *Snapshot ID*: the snapshot containing the first adjacency list fragment that belongs to the given vertex
- *Offset*: the index into the edge table of the given snapshot
- *Fragment Length*: the length of the given adjacency list fragment
- *Degree* (optional): the total vertex degree

Although the copy-on-write vertex table for a snapshot conceptually contains entries for all vertices, the data pages comprising the adjacency list fragments for that snapshot contain only small number of vertices. Therefore, we have to store adjacency list fragment lengths explicitly, because there is no guarantee that consecutive vertices reference consecutive adjacency list fragments, so the typical trick of subtracting the offsets from consecutive elements in the vertex table does not work.

*Adding and Deleting Vertices:* Different versions of the vertex table can have different sizes. We add new vertices to a snapshot by creating a larger vertex table for that snapshot. We delete vertices by replacing their vertex table entries with special *nil* records.

*On-Disk Format:* 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 easily reclaim space from deleted snapshots while keeping the number of files manageable.

The on-disk representation is nearly identical to the in-memory representation. 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 a previous snapshot (i.e., there are no modifications between the two snapshots),
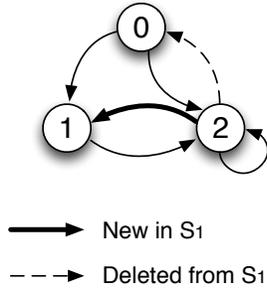
(a) Modified Graph Structure
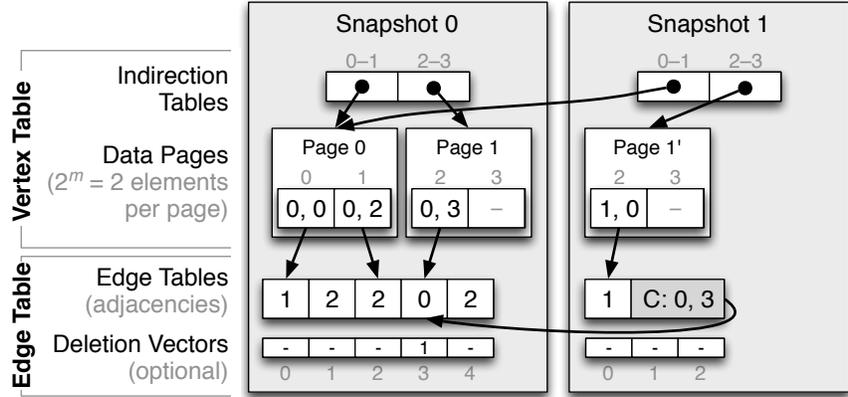
(b) LLAMA Representation

Fig. 1: **Example of a LLAMA with Two Snapshots** with one added and one removed edge in Snapshot 1. The elements in the vertex table contain the snapshot number and index into the corresponding edge table, omitting adjacency list fragment lengths and degrees for brevity. The adjacency list fragment in the edge table of Snapshot 1 ends with a continuation record that points to where the adjacency list continues.

we store a reference to the previous snapshot rather than storing a duplicate indirection table. We store a snapshot's data pages contiguously in the file.

*Data Load:* 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.

*Large Initial Capacity:* We can easily allocate a LAMA with more capacity than is currently needed with only a small space penalty. We set entries in the indirection array to refer to not-yet-populated pages, all referencing a single zero-filled page, which we update copy-on-write. For example, even with small 4 KB data pages, reserving space for one million eight-byte elements requires only 16 KB in memory (one 8-byte pointer per data page) or 24 KB on disk (one 4-byte snapshot ID + one 8-byte offset per data page), plus the zero page.

### B. Edge Tables and Deletion Vectors

*Edge Tables:* An edge table is a fixed-length `mmap`-ed array that contains adjacency list fragments stored consecutively; each fragment contains edges added in the given snapshot. We represent an edge by the vertex ID corresponding to its target vertex. By default, LLAMA allocates 64 bits for vertex IDs, but users can alternately request 32-bit vertex IDs. We use 64-bit IDs in the remainder of the paper unless stated otherwise.

Each adjacency list fragment in a delta snapshot (a snapshot with ID greater than 0) ends with a 16-byte *continuation record* that points to the next fragment or a special NULL record if there are no more edges. The continuation record of vertex $v$ that points to snapshot $s$ is a copy of $v$'s per-vertex structure from the vertex table's version $s$.

For example, consider finding vertex 2's adjacency list from

Snapshot 1 in Figure 1. The vertex table entry indicates that the adjacency list starts in edge table 1 at index 0. The adjacency list fragment of this snapshot {1} indicates that there is one edge in this snapshot, and the continuation record $C : 0, 3$ reveals that the adjacency list continues in edge table 0 at index 3. As edge table 0 corresponds to the oldest snapshot, we do not need a continuation record at the end of the adjacency fragment in that snapshot.

*Deletion Vectors:* LLAMA supports two methods for edge deletion. By default, LLAMA deletes edge $u \rightarrow v$ by writing $u$'s entire adjacency list (without the deleted edge) into a new snapshot with a NULL continuation record. This approach is time-efficient but space-inefficient.

When deletions are frequent or adjacency lists are sufficiently large that copying them takes up too much space, the user can choose *deletion vectors* instead. Deletion vectors are arrays, logically parallel to the edge tables, encoding in which snapshot an edge was deleted. In Figure 1, 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, 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. When using 32-bit vertex IDs, LLAMA stores deletion vectors in a separate array.

### C. Properties

LLAMA implements graph element properties using LAMAs. We store each property in its own LAMA, parallel to either the vertex or edge table. (The index into the LAMA is the same as the index into the table.) Figure 2 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
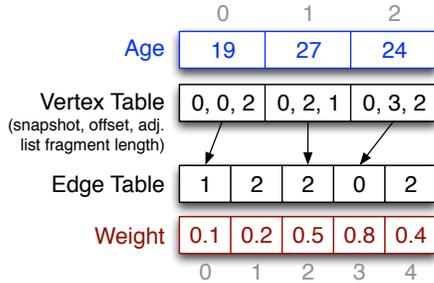
Fig. 2: **Example of a Single-Snapshot LLAMA with a Vertex and an Edge Property**. The figure contains just one snapshot and just the high-level representation (omitting the internal structure) of the LAMAs – the vertex and the property tables – for clarity.

clarity. A LAMA can store only fixed-size elements; we use a separate key/value store for variable-sized properties.

### D. Loading Data

LLAMA's bulk loader accepts a variety of edge-list based formats. 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.

### E. 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. We only support merging the most recent snapshots. Deleting older snapshots requires updating the vertex tables and continuation records in newer snapshots; we have left implementing this for future work.

LLAMA's operation is similar to an LSM tree [13], [14]: the system always writes data into a new snapshot and eventually merges the snapshots together, except where multiple versions are specifically desired.

### F. Write-Optimized Delta Map

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

The user can run queries across both the delta map and the read-optimized graph storage, but querying the delta map is slower. LLAMA thus excludes the delta map from analytics workloads by default, treating it instead as a buffer for updates. Snapshot creation is sufficiently fast that we expect a query needing the most up-to-date data to create a snapshot and then compute on the snapshot.

## III. DESIGN RATIONALE

One of our primary design goals was to be as efficient as in-memory analytics engines for in-memory data. We benchmarked the three most common state-of-the-art in-memory representations: CSR, which is used in a large number of graph analytic systems [1], [3], [15], [16]; storing adjacency lists in separate objects [17], [18], [19]; and representing the graph as a collection of compressed sparse bitmaps [20]. We chose CSR because it was the most efficient. For example, breadth-first search on a 9 million node, 45 million edge Barabási graph ran 34% faster on an immutable CSR than on a representation with separate adjacency lists and 46% faster than when using bitmaps. (We obtained these results on the "commodity machine" described in Section V-A.)

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, 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 [8], [21], 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.

### A. Mutability and Snapshots

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 [7], while others treat the CSR as a log, appending a modified adjacency list to the end of an edge table [1], [15]. The first approach requires rebuilding the entire CSR to merge changes from the delta map, because delta maps are typically too slow for analytics workloads. 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 destroys the sequentiality of the adjacency lists. Consequently, the entire CSR must occasionally be rebuilt.

LLAMA chooses a different approach. We tried three different methods for storing adjacency lists in snapshots: 1) write new edges only into a new snapshot, 2) write the entire adjacency list, or 3) choose one or the other based on a simple policy. The first of these performs best overall. We found that writing complete adjacency lists (and the hybrid approach)

performs marginally better when the graph is entirely memory resident, but the graph grows rapidly when updated and falls out of memory. Performance then plummets.

### B. Properties

Separating storage of properties from the graph structure draws on column-store philosophy: load only the needed parts of the data. For example, many algorithms need only outgoing edges. Loading only the edges and properties necessary for an algorithm leads to superior performance relative to systems that must load all data associated with a vertex or an edge.

However, if the user frequently runs analyses that require certain properties, using columnar storage might backfire. We thus plan to implement a row-store like materialized views in the future that will combine the graph structure with user-specified vertex and edge properties in a single data structure.

### C. The Cost of the Indirection Table

To understand the overhead of accessing data through the indirection table in LAMA when all data fits 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 10 million node and 50 million Erdős-Rényi edge random graphs. The first algorithm, triangle counting, consists of a tight inner loop; this highlights the effect of additional instructions in element access. On this highly CPU-sensitive workload, LAMA's overhead was 5.0%. On PageRank [12], which has no such tight inner loop, the performance difference between LAMA and an immutable array was statistically insignificant. We also compared LAMA to a copy-on-write 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.

### D. Buffer Management

As we transitioned from a purely in-memory representation to one that could gracefully and efficiently handle datasets larger than memory, we implemented and evaluated several alternative strategies for caching data. Our final implementation uses `mmap`, which produces effectively zero overhead once data are memory resident. This was our highest priority design requirement.

We also experimented with maintaining our own buffer pool and transferring data between disk and memory explicitly. We tried two different approaches for pinning pages in memory – reference counting using atomic increment/decrement operations and hazard pointers [22], but found the in-memory overhead prohibitive in both cases. Relative to the `mmap`-based implementation, the overhead ranged from nearly 50% for PageRank to nearly 200% for triangle counting.

This result is not entirely unsurprising. Moderately high overheads due to page latching have already been observed in relational databases [23], but the results for graph analytics are even worse. To better understand this, we disassembled our triangle counting routine; we found that the LAMA array is accessed in a loop of 12–16 instructions. Adding even a small number of instructions significantly affects execution time.

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 always access vertices in order (e.g., GraphChi [7], X-Stream [8]), in which case the operating system does a good job prefetching and evicting. The second is to simply ignore out-of-memory execution (e.g., GraphLab [24], Pregel [25]), and the third is to use `mmap` as we do (e.g., Lin et al. [21]). Should the need arise for more specific eviction control, we have experimented with `madvise` and `mlock`; preliminary results suggest that they can make `mmap` to behave as we want.

### IV. USING LLAMA

LLAMA is a C++ library that users can easily embed in their programs. After opening a database, the user can iterate over all vertices in the database using a for loop, potentially using OpenMP [26] for parallelism, or can address vertices directly using their IDs. LLAMA thus supports both whole-graph queries and targeted queries such as BFS that start from a specific vertex. Then, given a vertex, the user can open an iterator to iterate 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 the adjacency lists occupy a single fragment. LLAMA additionally provides a BFS construct and syntactic sugar to simplify many common operations such as iterating over all nodes or iterating over the out-edges of a given node.

LLAMA provides a general purpose programming model instead of restrictive vertex- or edge- centric models, such as GAS [27]. 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 more sequential access patterns. Unlike GraphChi [7] or X-Stream [8], 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.

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; we plan to support dynamic adaptation.

### V. EVALUATION

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 [12] (PR), breadth-first search (BFS), and triangle counting (TC). The goal of

| System | Load | PageRank | BFS | TC |
|--------|------|----------|-----|-----|
| 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 | PageRank | BFS | TC |
|--------|------|----------|-----|-----|
| 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 I: **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.

| System | Load | PageRank | BFS | TC |
|--------|------|----------|-----|-----|
| 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 | PageRank | BFS | TC |
|--------|------|----------|-----|-----|
| 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 II: **BigMem Performance** for LiveJournal and Twitter datasets, elapsed times shown in seconds.

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 [3] and GraphLab v. 2.2 (PowerGraph) [27], and two out-of-core systems, GraphChi [7] and X-Stream [8]. 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 [18], but we do not include the results in the paper. 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 [28].

### A. 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:

- *LiveJournal* [29]: 4.8 mil. nodes, 69.0 mil. edges
- *Twitter* [30]: 41.7 mil. nodes, 1.47 bil. 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.

### B. Cross-System Comparison

Tables I and II 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 V-C.) The LiveJournal dataset fits in memory in both cases, while Twitter fits in the memory only on the BigMem machine.

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.

*GreenMarl:* We used nearly identical implementations of PageRank, BFS, and triangle counting on both LLAMA and GreenMarl. GreenMarl's runtime likewise uses OpenMP

| System | Time (s) | | | CPU Time Breakdown (%) | | | I/O (GB) | |
|---|---|---|---|---|---|---|---|---|
| | 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 III: **PageRank on Twitter: Performance Breakdown** on the Commodity platform.

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 0.23% slower on triangle counting, 3% on PageRank, and 18% on BFS. 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 most workloads. The only outlier is triangle counting on the BigMem platform, where the overhead is nearly 60%. This is because 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 in order to improve cache locality. This is a widely-used optimization for triangle counting on very 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.

*GraphLab:* LLAMA significantly outperforms GraphLab running on a single machine for all cases except for 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, 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 unsorted 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 5).

*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 III 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 [23].

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.

*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 Green-Marl'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 auto-detects 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.

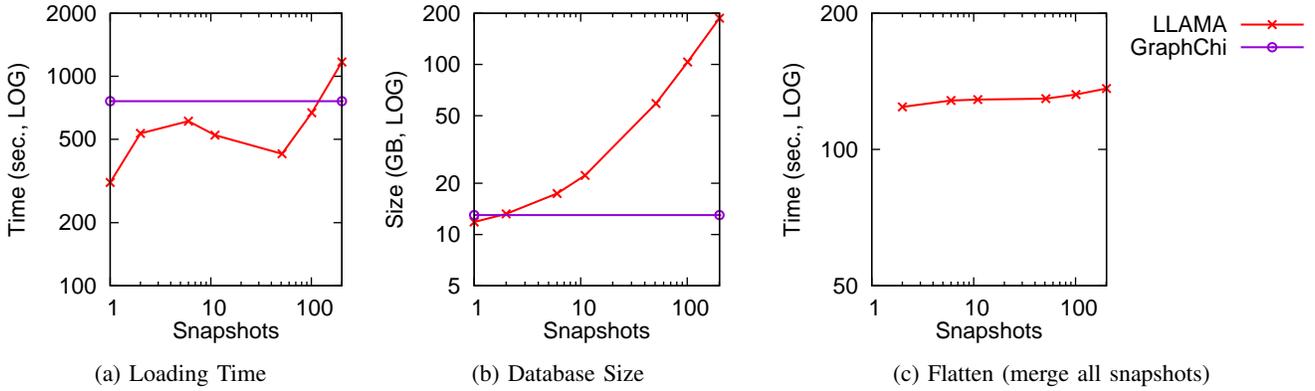(a) Loading Time  (b) Database Size  (c) Flatten (merge all snapshots)

Fig. 3: **Loading and Flattening Multiple Snapshots** of the Twitter graph on the Commodity platform. The GraphChi preprocessing time is shown for reference.

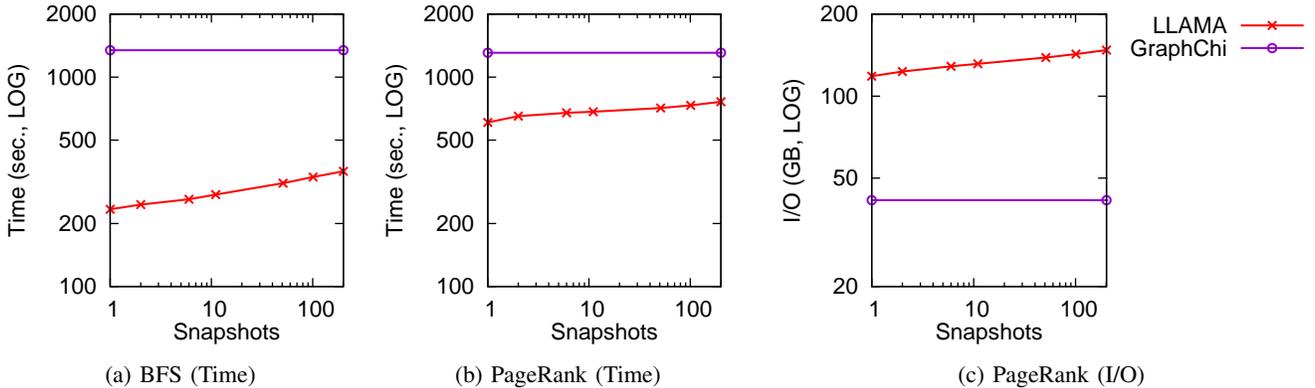

(a) BFS (Time)  (b) PageRank (Time)  (c) PageRank (I/O)

Fig. 4: **Computing Across Multiple Snapshots** of the Twitter graph on the Commodity platform. The GraphChi preprocessing time is shown for reference.

## C. 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 V-B. 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 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 3 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 almost 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, which users can use if they do not require multiversion access to all snapshots. 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 4 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
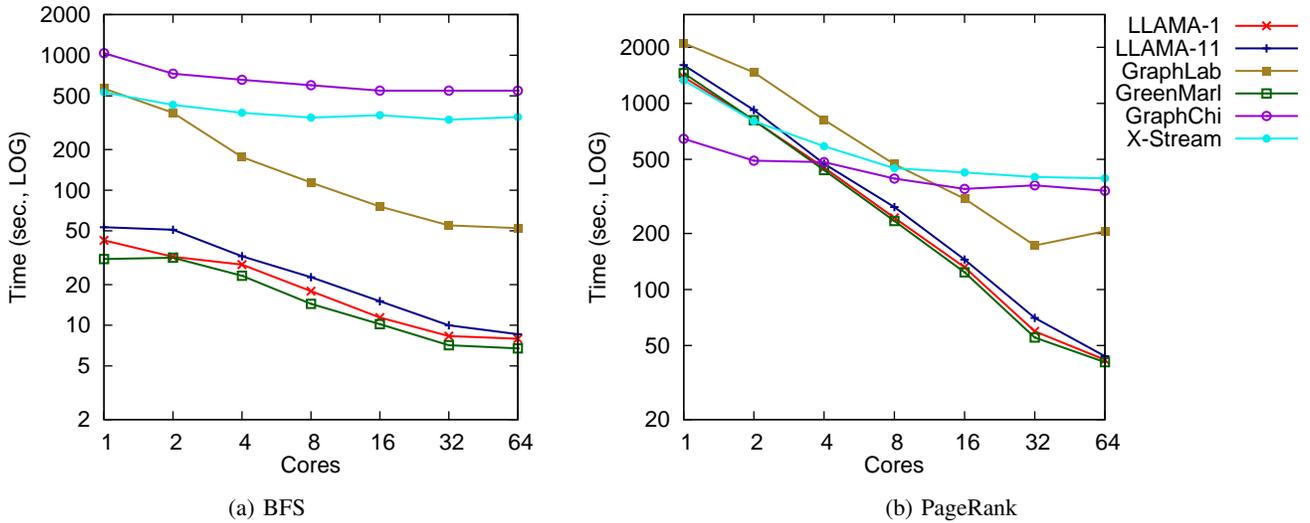
(a) BFS

(b) PageRank

Fig. 5: **BFS and PageRank Scaling as a Function of Core Count** on the BigMem platform for the Twitter graph.

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 3c) 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 [31]), we find that merging less 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.

### D. Scaling in the Number of Cores

We now turn to scalability. Figure 5 shows the performance of BFS and PageRank on the Twitter graph as a function of the number of cores on the BigMem platform.

The plot includes two lines for LLAMA: LLAMA-1, with the entire graph in a single snapshot, and LLAMA-11, which loads it into 11 snapshots as explained in Section V-C (we
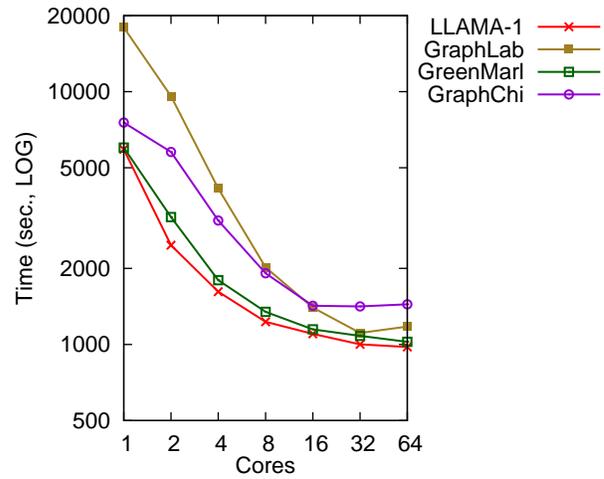


Fig. 6: **Triangle Counting as a Function of Core Count** on the BigMem platform for the Twitter graph.

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 III, 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 [32].

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
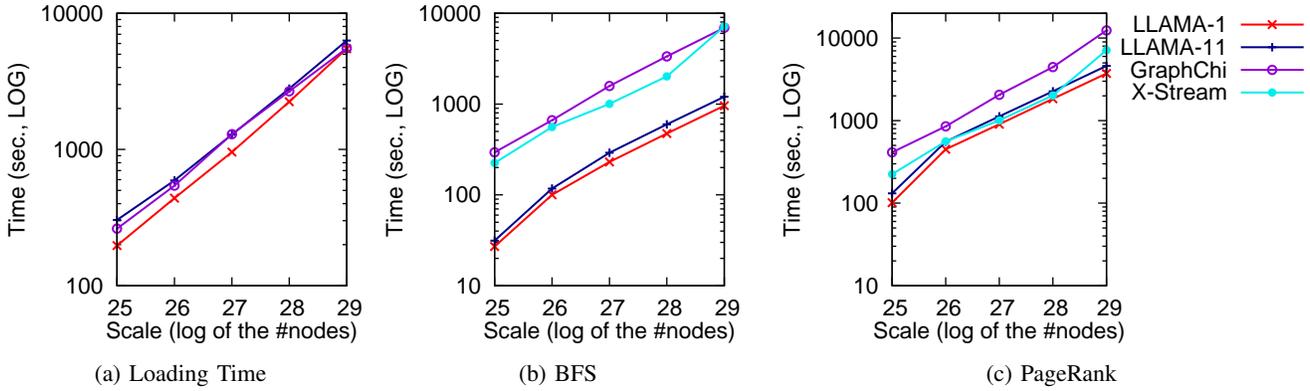
(a) Loading Time      (b) BFS      (c) PageRank

Fig. 7: **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.
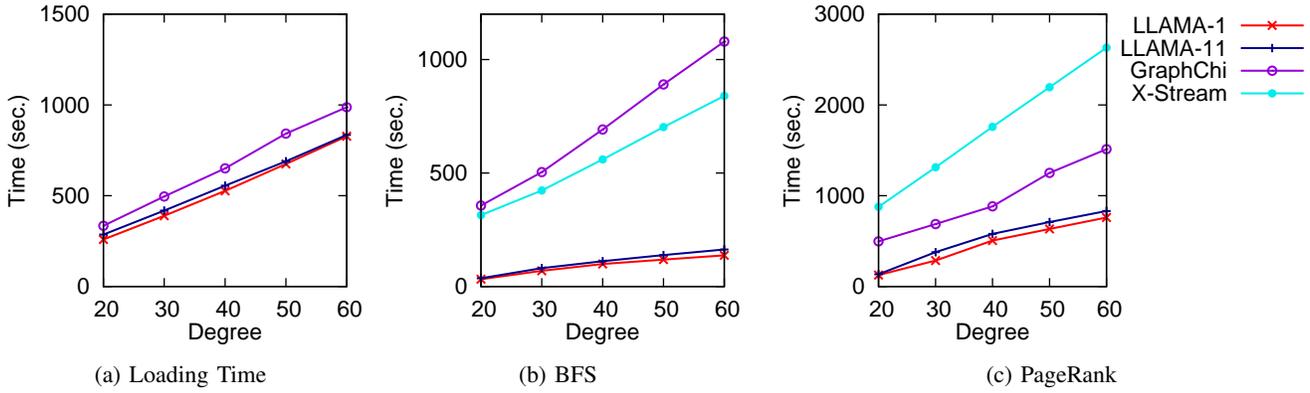


(a) Loading Time      (b) BFS      (c) PageRank

Fig. 8: **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.

of the page cache scalability limitation. This in turn affects the performance of the rest of the system that depends on the loaded data.

*Triangle Counting:* Figure 6 shows the performance of triangle counting on the BigMem platform for LLAMA, Green-Marl, 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.

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!

*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 Commod-

ity 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.

### E. 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 7, we fix the average vertex degree to 16 as recommended by Graph500 [33] 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 graph sizes ranging from 4.5 GB to 72.0 GB. In Figure 8, 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.

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 V-B.

## VI. Related Work

We now place LLAMA's contributions in the context of existing graph analytic engines. LLAMA is currently a single-machine graph analytics system, so we begin by discussing other single-machine systems. We then expand to include distributed analytic engines that deal with graphs exceeding the memory of a single machine by sharding across multiple machines. Finally, we compare LLAMA to other multiversioned data structures.

### A. Single Machine Graph Analytics

GraphChi [7] was one of the first analytic systems able to process out-of-memory graphs using commodity hardware. GraphChi and LLAMA employ two different design philosophies: GraphChi shards the data into subsets small enough to fit in memory and then reads the subgraphs from disk in a mostly sequential access pattern, pushing information to a user-provided vertex program. In contrast, LLAMA programs pull data on demand, leaving access locality to the user, thereby enabling, but not requiring sequential access. Additionally, LLAMA uses `mmap` rather than file I/O which incurs less overhead, especially when data mostly fit in memory.

TurboGraph [6], with its Pin and Slide computation model, is a more recent vertex-centric graph processing framework designed for SSDs. It improves on GraphChi by extracting more parallelism, fully overlapping CPU processing and I/O. Both TurboGraph and LLAMA reduce CPU time (relative to systems like GraphChi), but each in different ways: Turbo-Graph pins whole pages and processes them, while LLAMA leverages the operating system's virtual memory mechanism and access locality. We were not able to compare to Turbo-Graph, because it does not run on our Linux platforms.

X-Stream [8] uses an edge-centric gather-and-scatter model that takes as input a binary formatted edge-list requiring no preprocessing. X-Stream sequentially streams through the edges, which makes it useful for whole graph analysis. In contrast, LLAMA efficiently supports point queries as well as whole-graph analysis.

Lin et al. [21] argued for, and demonstrated, the feasibility of using a simple `mmap`-based approach for graph processing, in which a 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, improving upon it by using a mutable, Compressed Sparse Row representation, that supports efficient incremental ingest and point-queries, while sharing a simple programming model. A 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.

Galois [4], Grace [1], Green-Marl [3], and Ligra [5] 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. Of these, Galois can also run out-of-core by executing a GraphChi-like vertex program iterating over a persistent CSR, but it does not currently support all the features of GraphChi.

### B. Distributed Graph Analytics

An alternative to out-of-core execution is to leverage memory from multiple machines. GraphLab [24], [2] and Pregel [25] are the most widely-known examples of such systems. Both use vertex-centric computation. PowerGraph [27], now GraphLab v. 2.2, runs more restricted gather-and-scatter (GAS) vertex programs but improves on the original GraphLab by sharding the graph by edges rather than vertices. PE-GASUS [34] is a recent example of a Hadoop-style graph processing framework. Kineograph [19] adds the ability to incrementally ingest data into a distributed system and produce timely, consistent results. While LLAMA is currently limited to a single machine, we believe that we can leverage its out-of-core execution to "page" data to remote machines instead of to the persistent store, turning it into a distributed engine.

### C. Multiversioned Graphs

LLAMA's mutability provides multiversioning functionality not found in the engines discussed so far. DeltaGraph [35] is an index structure that handles mutation by creating snapshots, represented by a set of deltas and event-lists. Like DeltaGraph, Chronos [36] optimizes for efficient queries across snapshots. Instead of deltas, it batches versions of vertices together. In contrast to optimizing for running queries spanning multiple snapshots, LLAMA is optimized for efficient incremental ingest.

## VII. Conclusion

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.

REFERENCES

[1] V. Prabhakaran, M. Wu, X. Weng, F. McSherry, L. Zhou, and M. Haridasan, "Managing large graphs on multi-cores with graph awareness," in *USENIX ATC*. Berkeley, CA, USA: USENIX Association, 2012.

[2] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein, "Distributed GraphLab: A framework for machine learning in the cloud," *CoRR*, 2012.

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

[4] D. Nguyen, A. Lenharth, and K. Pingali, "A lightweight infrastructure for graph analytics." in *SOSP*, M. Kaminsky and M. Dahlin, Eds. ACM, 2013, pp. 456–471.

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

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

[7] A. Kyrola, G. Blelloch, and C. Guestrin, "GraphChi: large-scale graph computation on just a PC," in *OSDI*. Berkeley, CA, USA: USENIX Association, 2012, pp. 31–46.

[8] A. Roy, I. Mihailovic, and W. Zwaenepoel, "X-Stream: edge-centric graph processing using streaming partitions." in *SOSP*, M. Kaminsky and M. Dahlin, Eds. ACM, 2013, pp. 472–488.

[9] J. Dongarra, P. Koev, X. Li, J. Demmel, and H. van der Vorst, *10. Common Issues*. SIAM, 2000, ch. 10, pp. 315–336. [Online]. Available: http://epubs.siam.org/doi/abs/10.1137/1.9780898719581.ch10

[10] J. Leskovec, J. M. Kleinberg, and C. Faloutsos, "Graphs over time: densification laws, shrinking diameters and possible explanations." in *KDD*. ACM, 2005, pp. 177–187.

[11] L. Yang, L. Qi, Y.-P. Zhao, B. Gao, and T.-Y. Liu, "Link analysis using time series of web graphs." in *CIKM*. ACM, 2007, pp. 1011–1014.

[12] L. Page, S. Brin, R. Motwani, and T. Winograd, "The PageRank citation ranking: Bringing order to the web," Stanford University, Tech. Rep., 1998.

[13] H. V. Jagadish, P. P. S. Narayan, S. Seshadri, S. Sudarshan, and R. Kanneganti, "Incremental organization for data recording and warehousing," in *VLDB*. Morgan Kaufmann, 1997, pp. 16–25.

[14] P. E. O'Neil, E. Cheng, D. Gawlick, and E. J. O'Neil, "The log-structured merge-tree (LSM-tree)," *Acta Informatica*, vol. 33, no. 4, pp. 351–385, 1996.

[15] K. Bharat, A. Z. Broder, M. R. Henzinger, P. Kumar, and S. Venkatasubramanian, "The connectivity server: Fast access to linkage information on the web." *Computer Networks and ISDN Systems*, vol. 30, no. 1-7, pp. 469–477, 1998.

[16] "SNAP: Stanford network analysis platform," http://snap.stanford.edu/snap, January 2014.

[17] D. A. Bader, J. Berry, A. Amos-Binks, D. Chavarría-Miranda, C. Hastings, K. Madduri, and S. C. Poulos, "STINGER: Spatio-temporal interaction networks and graphs (STING) extensible representation," Georgia Institute of Technology, Tech. Rep., 2009.

[18] "Neo4j: the graph database," http://neo4j.org, April 2014.

[19] B. Shao, H. Wang, and Y. Li, "Trinity: A distributed graph engine on a memory cloud," in *SIGMOD*. New York, NY, USA: ACM, 2013, pp. 505–516.

[20] N. Martínez-Bazan, M. A. Aguila-Lorente, V. Munts-Mulero, D. Dominguez-Sal, S. Gmez-Villamor, and J.-L. Larriba-Pey, "Efficient graph management based on bitmap indices." in *IDEAS*. ACM, 2012, pp. 110–119.

[21] Z. Lin, D. H. P. Chau, and U. Kang, "Leveraging memory mapping for fast and scalable graph computation on a PC." in *BigData Conference*, 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, and R. Nambiar, Eds. IEEE, 2013, pp. 95–98.

[22] M. M. Michael, "Hazard pointers: Safe memory reclamation for lock-free objects." *IEEE Trans. Parallel Distrib. Syst.*, vol. 15, no. 6, pp. 491–504, 2004.

[23] S. Harizopoulos, D. J. Abadi, S. Madden, and M. Stonebraker, "OLTP through the looking glass, and what we found there." in *SIGMOD*, J. T.-L. Wang, Ed. ACM, 2008, pp. 981–992.

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

[25] G. Malewicz, M. H. Austern, A. J. C. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: a system for large-scale graph processing." in *SIGMOD*, A. K. Elmagarmid and D. Agrawal, Eds. ACM, 2010, pp. 135–146.

[26] "The OpenMP API specification for parallel programming," http://openmp.org/wp, April 2014.

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

[28] D. Chakrabarti, Y. Zhan, and C. Faloutsos, "R-MAT: A recursive model for graph mining." in *SDM*, M. W. Berry, U. Dayal, C. Kamath, and D. B. Skillicorn, Eds. SIAM, 2004.

[29] L. Backstrom, D. Huttenlocher, J. Kleinberg, and X. Lan, "Group formation in large social networks: membership, growth, and evolution," in *SIGKDD*, 2006, p. 4454.

[30] H. Kwak, C. Lee, H. Park, and S. Moon, "What is Twitter, a social network or a news media?" in *WWW*. New York, NY, USA: ACM, 2010, pp. 591–600.

[31] "Data never sleeps 2.0," http://www.domo.com/learn/data-never-sleeps-2, April 2014.

[32] D. Zheng, R. Burns, and A. S. Szalay, "A parallel page cache: IOPS and caching for multicore systems," in *HotStorage*. USENIX, 2012.

[33] "Graph 500 benchmark," http://www.graph500.org/specifications, October 2010.

[34] U. Kang, C. E. Tsourakakis, and C. Faloutsos, "PEGASUS: mining peta-scale graphs." *Knowl. Inf. Syst.*, vol. 27, no. 2, pp. 303–325, 2011.

[35] U. Khurana and A. Deshpande, "Efficient snapshot retrieval over historical graph data." in *ICDE*, C. S. Jensen, C. M. Jermaine, and X. Zhou, Eds. IEEE Computer Society, 2013, pp. 997–1008.

[36] W. Han, Y. Miao, K. Li, M. Wu, F. Yang, L. Zhou, V. Prabhakaran, W. Chen, and E. Chen, "Chronos: A graph engine for temporal graph analysis." in *EuroSys*. ACM, 2014.