Buffer Management for LLAMA: A Graph Analytics System

Albert Wu David Ding

***Abstract*—Graphs are ubiquitous in modern computing, and creating systems for efficient graph computation is a difficult problem. In particular, computing on graphs too large to fit in the memory of a machine requires special attention to memory management. We present a buffer management mechanism for LLAMA, a persistent graph analytics system. We show that our buffer management strategy can significantly improve performance on several query workloads, such as one-level neighborhood queries when recently queried nodes are queried with higher probability, friend-of-friends queries, and shortest path queries.**

I. Introduction

Graphs are everywhere in modern computing, modeling diverse phenomena such as social networks, the Internet, and roads. Many generic platforms for analyzing huge amounts of data, such as MapReduce, perform poorly in the analysis of graph-structured data [4][8], since real world graphs of interest are often large and interconnected in complex ways, and moreover, vertex degrees for these graphs tend to follow the power law [1], meaning that a nontrivial number of the vertices have huge degrees and a large proportion of edges lead to these vertices. Hence, efficient graph computation platforms must carefully take into account issues such as vertex placement to ensure locality of memory accesses that is necessary for performance.

Graph analytics systems can be divided into two major camps: in-memory distributed computing systems, and out-of-memory systems [4][7]. In-memory graph storage systems, such as Pregel and GraphLab, keep the entire graph in memory, distributing computation across a cluster of machines if necessary [6][8]. Because these systems have access to more resources, they run faster; however, large shared-memory machines tend to be fairly expensive, and performance on clusters of cheap machines is poor due to the communication and fault tolerance required [4][2]. Out-of-memory systems, such as LLAMA [7] or GraphChi [4], keep the graph on a persistent store, such as SSD, fetching portions of the graph into memory as needed. These systems run more slowly due to I/O overhead, but require fewer computational resources.

Effective buffer management is needed to alleviate the poor performance of persistent stores, and designing a buffer management system tailored for graph computation is an open problem [5][7]. This paper presents a buffer management strategy for LLAMA, one example of an out-of-memory system [7]. LLAMA memory-maps the entire graph file into the virtual address space, avoiding the overhead associated with userspace buffer management (such as extra copies from the kernel buffer cache into user space). LLAMA performs well on analytics algorithms that scan the graph in a predictable sequential manner, because many operating systems, including Linux, perform sequential prefetching by default; however, for other access patterns, the kernel's default prefetching mechanism is inefficient, as the kernel does not take into account the graph structure in order to make more informed prefetching decisions.

For instance, one common access pattern for graphs is to visit a node and immediately visit its neighbors, e.g., in a naïve implementation of breadth-first search or triangle counting. While triangle counting or breadth-first search can be rewritten to operate purely sequentially, these implementations are not as straightforward or as well-known as the typical algorithms for breadth first search and triangle counting taught in algorithms classes. Ideally, a graph analytics system should allow users to write simple algorithms they fully understand without forcing them to worry about performance.

Additionally, other workloads cannot be made sequential. For example, consider a graph database application that processes a series of queries for the neighbors of a queried node. In this case, the application cannot guarantee sequential access to the graph because the database must satisfy the online query stream and must respond as quickly as possible to the user (it cannot buffer up the queries to be processed in batch). However, as query sequences tend to follow recency patterns (recently-accessed nodes tend to be accessed again in the near future) and locality patterns (neighbors of recently-queried nodes are likely to be accessed in the future), a smarter prefetching strategy could potentially significantly improve performance.

The contributions of this paper are the following:

- A prefetching strategy that uses asynchronous `madvise` calls to instruct the kernel to prefetch data.
- Two prefetching policies for sequential and adjacency-local access patterns.
- Creation of three query-like workloads: *edge queries*, a workload that simulates recency-characteristics of queries to a real graph database; *friend of friends*, which computes neighbors of neighbors of queried nodes; and *shortest-path*, which computes the shortest unweighted directed path between two queried nodes.
- An evaluation of the impact of the two prefetching strategies on sequential and query-like workloads.

The rest of the paper is organized as follows: in Section II, we discuss previous work on graph analytics systems and prefetching. Section III discusses the mechanisms we use for our prefetching advisor. Then, Section IV discusses the workloads we use to test our mechanisms and Section V evaluates our mechanisms on these workloads. Finally, Section VI sums up our results and provides avenues for future work in improving buffer management for graph analytics systems.

II. Related Work

A. *Offline Graph Analytics*

Offline graph analytics systems are designed to handle large computations involving the whole graph, typically as a batch job. Examples of in-memory graph analytics systems are Pregel [8], GraphLab [6], and Ligra [13]. These systems introduced many of the graph computation paradigms in existence, including vertex-centric programs [8]; discussed ways of parallelizing computation [6]; and presented data structures for representing graphs, such as the compressed-sparse-row (CSR) representation [13]. This representation maintains a single vertex table and a single large edge table, which stores the adjacency lists of each vertex contiguously in memory. The entries in the vertex table are indexed by vertex ID and point to the start of that vertex's adjacency list in the edge table. These systems perform well for in-memory graphs, but because of the in-memory constraint, they can only be run on (relatively) small graphs, large-memory machines, or distributed systems. Nevertheless, the performance of these systems represents a goal that out-of-memory graph systems strive to reach and emulate.

Unlike in-memory graph systems, GraphChi [4] is meant to run out-of-memory. The authors argue that while in-memory systems are fast, they are expensive and hard-to-obtain. Their goal was to design a system that performs well on a regular personal computer, when the entire graph does not fit in memory. GraphChi uses both the CSR and the similar compressed-sparse-column (CSC) representations to store the graph on disk. GraphChi introduces Parallel Sliding Windows (PSWs) so that the graph is accessed sequentially and no redundant I/Os are performed. In this scheme, the vertices are partitioned into intervals, and each interval is associated with a corresponding shard, representing all edges with a destination in the interval, sorted by source. The intervals are chosen to balance the size of each shard, so that each shard can fit into memory. As GraphChi iterates over the vertices, PSWs sequentially read the necessary shards, creating a sequential access pattern that is amenable to the natural prefetching of the kernel.

Turbo-Graph improves upon the performance of GraphChi for SSDs by extracting more computation and I/O parallelism [2]. It uses a custom buffer manager which pins whole pages and processes them. Prefetching calls are issued asynchronously to maximize CPU and I/O parallelism; this is analogous to our approach of issuing asynchronous `madvise` calls. We were not able to compare to Turbo-Graph because it does not run on our Linux platforms. Unlike Turbo-Graph, we rely much more on the kernel's default virtual memory system for buffer management.

LLAMA is an out-of-memory database that supports mutable graphs and achieves similar performance to in-memory graphs when the graph fits in memory [7]. LLAMA uses a CSR representation for the graph to attain this performance, but LLAMA allows the graph to be updated. In a normal CSR representation, the adjacency lists are stored consecutively, so inserting or deleting from a CSR is difficult; LLAMA introduces levels, i.e., snapshots, to allow for the easy insertion of edges (create a new copy-on-write snapshot with the edge inserted), while maintaining the sequential layout of the CSR representation. To handle larger-than-memory graphs, LLAMA `mmap`s the graph into the virtual address space. `Mmap`ping performs better than explicit buffer pool management, especially when the graph could fit in memory: the performance of explicit buffer pool management suffered from a 50% to 200% performance overhead [7].

LLAMA performs significantly better than comparable systems such as GraphChi for sequential access patterns [7]. Overall, though, the biggest difference between GraphChi and LLAMA is the design philosophy [7]: while GraphChi separates data into small in-memory subsets and moves mostly sequentially through data, LLAMA allows the user to access the graph in an arbitrary manner. This flexibility in access patterns in LLAMA makes it a better candidate for experimentation with buffer management since there is more room for improvement in non-sequential accesses. Moreover, this flexibility allows LLAMA to handle database workloads as well as sequential offline batch computations. LLAMA provides this flexibility through a simple iterator interface to the graph, which allows the user to iterate over all vertices adjacent to a particular node. This simple primitive can be used to implement many diverse graph algorithms, such as PageRank, BFS, triangle counting, as well as query workloads such as neighborhood queries or friends-of-friends queries.

B. Graph databases

Graph databases represent another major way people use graphs and present many different challenges than those encountered by graph analytics systems. For instance, graph databases need to support modifiable graphs, something that many graph analytics systems do not support [4][7][8], and moreover, graph databases typically make some consistency guarantees (such as ACID transactions). Moreover, graph analytics systems deal with offline batch computations, which can often be rewritten to access the graphs sequentially; graph databases on the other hand deal with online queries, and hence have to access random portions of the graph [12]. Two popular graph databases used today include Neo4j [10] and Dex (recently renamed Sparksee) [9]. Their designs focus mostly on efficient index structures and query optimization as opposed to buffer management or graph layout [9]. Indeed, while these databases provide good performance when the graphs fit in memory, their performance is poor for out-of-memory graphs [5].

Nevertheless, graph databases and analytics systems share significant common functionality, and some systems aim to efficiently handle both offline processing and online queries. Trinity is one such example for distributed in-memory systems [12]. GraphChi-DB, an extension of GraphChi, is an out-of-memory system that provides support for online-queries. Its design is similar to the design of LLAMA. To support dynamic insertion, it uses a log-structured merge tree, similar to LLAMA's approach. Like LLAMA, it relies on `mmap` for buffer management. Instead of the CSR, GraphiChi-DB uses Partitioned Adjacency Lists (PAL) to represent graphs. In PAL, vertex IDs are split into continuous intervals, such that each interval is associated with an edge partition that stores edges sorted by source vertex ID. These continuous intervals can be of differing lengths but must satisfy the condition that each edge partition fits within memory. GraphiChi-DB does not provide its own query language, but instead provides a Scala-API to access and traverse the graph, similar to LLAMA's C++ API. The authors of the paper posed as a future research question whether a custom buffer manager that adapts to the structure and access patterns of the graph could provide better performance than the kernel's buffer cache; our paper shows that we can achieve better performance by issuing `madvise` calls without having to implement a complete buffer manager.

Another approach to optimize for the access patterns typical of graphs is to store the graph's vertices in some "optimal" reordering. For instance, G-store [11] is a storage-system for hard drives that can run underneath other graph analytics systems such as Pregel. It structures the graph so that adjacent vertices are stored close together, minimizing the seek time for neighborhood queries. While locality is less of an issue for SSDs, which have much better random access times, locality still is important, as it allows the kernel to sequentially prefetch relevant data into the buffer cache. Reordering the graph could hence significantly improve performance. One main drawback to G-store is that it cannot handle dynamic graphs, an important feature of graph databases. Moreover, each access pattern may have its own optimal ordering. Reordering the graph is time-consuming, and storing multiple different orderings of the graphs is space-inefficient. Hence, using reordering to improve performance is less flexible than buffer management strategies.

C. Madvise as a prefetching mechanism

The `madvise` command is a system call that provides the kernel with advice on memory-mapped pages. However, the operating system is not obligated to act on this advice, so until recently, `madvise` has not been used extensively for prefetching in performance-optimized systems [15]. The system call has a variety of options, including options for advising the kernel to prefetch or evict pages (e.g. `MADV_WILLNEED`, indicating that a certain range of memory will be accessed in the near future; and `MADV_DONTNEED`, indicating that a range of memory will not be accessed and thus can be evicted), as well as options to indicate manner of access (e.g. `MADV_SEQUENTIAL`, indicating that the memory is sequentially accessed and so should be prefetched sequentially and MADV_RANDOM, indicating that access is random so that prefetching does not improve performance). Indeed, Linux has implemented all the aforementioned options by setting flags for the manner of access and issuing asynchronous prefetching calls for MADV_WILLNEED (page allocation is handled synchronously, but I/O is asynchronous) [14].

An unpublished paper by Yacoby et al [15] presents preliminary experiments on the impact of madvise on performance for a particular SSD (256GB Samsung SSD 840 PRO, with a reported sequential read bandwidth of 540 MB/s and sequential write bandwidth of 520 MB/s) [15]. The paper demonstrates that madvise can significantly improve performance for both random and, surprisingly, sequential access on Linux, and it investigates the optimal way of issuing the madvise calls. For instance, one important factor to consider is the prefetching window, or how far ahead to prefetch. The paper indicates that for their particular SSD, the optimal prefetching window is around 3 MB, far above Linux's default size of 128 KB; this difference explains the increased performance of madvise for even sequential access. Although the specific numbers might vary from machine to machine, the general lesson is that current operating systems may not be optimized for SSDs and so madviseing even sequential accesses might be beneficial.

One difference between this work and previous work is that Yacoby et al's project considered a workload that simulates LLAMA, while we work with the real LLAMA system. Moreover, their work assumes that the order of accesses is completely known, and it is purely concerned with measuring the improvement that madvise can bring. In our work, the advising system does not explicitly know the sequence of queries generated by the user, so the main challenge is to predict vertices that will be accessed in the near future.

III. Buffer management mechanism and policy

The main graph-access mechanism in LLAMA is the *edge iterator*, which, for graph G and node $n \in G$, iterates through edges incident on n. Here is sample C++ code showing how the edge iterator is used:

```
ll_edge_iterator iter;
G.out_iter_begin(iter, n);
// iterate through n's edge table
FOREACH_OUTEDGE_ITER(idx, G, iter) {
    next_node = iter.last_node
    // Do computation on next_node
}
```

Our approach to buffer management is to use madvise calls to influence kernel prefetching. This approach is simple, and as Yacoby et al's work suggests, can be effective [15]. Initially, we tried issuing synchronous madvise calls: when the edge iterator touches a node, we issue an madvise call. After initial experiments, we found this approach was too slow, as it incurs the cost of a context switch for every iteration of the loop. Hence, we decided to issue madvise calls asynchronously. Before the computation, we spawn a madviseing-thread that issues madvise calls to the kernel. The madviseing thread and the computation thread communicate via a shared queue. When the computation thread accesses a vertex through the iterator specified above, the vertex is enqueued. The madviseing thread spins, waiting for a node to appear in the queue. When the node appears, it issues madvise calls according to some policy.

Initially, we implemented the queue using the C++ std::deque class with locks to protect the queue. However, performance using this queue was poor; examining the strace output, we found that this incurs significant memory allocation costs (brk accounted for more than 90% of all system time). We hence used a fix-sized circular buffer, with head and tail indices storing the head and tail of the queue. Older entries in the queue are overwritten if the queue is full. After this optimization, we found that futex calls (corresponding to locking) were the main bottleneck, so we removed the locks protecting the queue also. Without the lock, some vertices might get lost during enqueue; however, we implemented the data structure so that no memory corruption could occur, and all advised chunks of memory actually belong to the graph.

Using this queueing mechanism, we created two new prefetching iterators, called the neighborhood prefetcher, detailed in Section III-A, and the sequential prefetcher, detailed in Section III-B.

A. Neighborhood prefetcher

The *neighborhood prefetcher* is the prefetching strategy that optimizes for situations when previously-visited nodes and neighbors of visited nodes tend to be visited in the near future. In this policy, the madviseing thread will issue madvise calls for all the neighbors of an enqueued node.

We found that certain optimizations improved the performance gain achieved by this strategy. First, we associated each enqueued node with an *epoch*, representing the time when it was enqueued. The epoch is incremented on every enqueue. When our prefetcher `dequeues` a node, it skips `madvise`ing if the epoch of the node is too far behind the current epoch of the system (for our specific experimental setup, we found that skipping when the epoch difference is at least 4 maximizes performance). This optimization allows the `madvise` thread to catch up to the computation thread when it falls too far behind due to blocking system calls, etc. It also helps our system to avoid advising in stale data.

Another optimization we found helpful was preferentially loading the edge tables of higher-degree vertices if available. This optimization is helpful, as higher-degree vertices are neighbors of more nodes, so they tend to be reached more often. By preferentially `madvise`ing the edge tables of higher-degree vertices, we give the kernel more time to fetch their edge tables, so that when they are actually needed by the computation thread, there is a higher probability that the entire edge table is in memory.

Finally, we found it helpful to limit the size of the edge table to `madvise`. In some of our workloads, not all neighbors of visited nodes are actually visited. Then, asking the kernel to fetch a large, e.g., million-entry, edge table is wasteful if that edge table never gets accessed. We found that performance improved when our prefetcher only `madvise`ing a small portion of the edge table. Moreover, we hypothesize that when the edge table does eventually get accessed, the kernel's sequential prefetching can fetch the rest of the edge table while the computation thread works on the portions of the table already in memory[1].

B. Sequential Prefetcher

The *sequential prefetcher* optimizes for access patterns where edges immediately following recently-accessed edges in the CSR table are used in the near future. In this strategy, the madvising thread issues `WILL_NEED` calls for edges $e+s$ to $e+t$, where e is the first edge in the adjacency list for the dequeued node, and s and t are fixed constants representing the prefetching window. Previous work suggest that prefetching 1 megabyte chunks 3 megabytes in advance optimizes performance [15]. Since our edges are 8-byte integers, this suggests that $s = 3 \times 2^{12}$ and $t = 4 \times 2^{12}$. After experimenting with different parameters for s and t, we found that performance does not significantly depend on s and t, and we chose $s = 0$ and $t = 4 \times 2^{12}$, which seems to be optimal for our machine. While the kernel already does sequential prefetching, previous work mentioned earlier has indicated that `madvise`ing can still be beneficial in these instances due to inefficiencies with the kernel's sequential prefetcher [15].

C. General buffer manager

The goal of developing these prefetching strategies is to create a general buffer manager that takes care of prefetching in a way that is encapsulated from the user as much as possible. The original LLAMA interface provides iterator interfaces for nodes and edges connected to a particular node. We provide the user with different iterators corresponding to common access patterns: (1) a neighbor-edge iterator that prefetches in neighbor's edge tables, under the assumption that neighbors of recently-accessed nodes are more likely to be accessed and (2) a sequential-edge iterator that prefetches the next-indexed node's edge table, a common access pattern for the CSR representation and graph-analytic workloads.

We envision that when the LLAMA database is initialized, LLAMA spawns a prefetching thread, or potentially, a pool of prefetching threads. As specific types of iterators are used in graph workloads, they enqueue accessed nodes with a tag that indicates the access pattern. The thread, or pool of threads, continually dequeue these nodes one-by-one, and based on the tag corresponding to the dequeued node, uses `madvise` to prefetch either neighbors' edge tables or the subsequently-indexed nodes' edge tables. [2] An ideal buffer management system would be easy for users to use, while giving them the power and flexibility to optimize performance for their specific workload.

IV. Workloads

We measured the performance of our two prefetching strategies using the workloads described in this section.

A. Edge queries

The *edge queries* workload simulates queries to a graph database. The workload processes a stream of query nodes, and outputs the neighbors of each node. If the stream of query nodes is completely random, the best an `madvise`ing strategy can do is to `madvise` the edge table for the node itself. But because the node is already being accessed, this `madvise`ing can only improve performance for large-degree nodes.

However, in real-world applications, queries are not completely random. Recently-queried nodes, along with their neighbors, tend to be queried more often in the future. Moreover, higher-degree nodes tend to be queried with higher frequency. We created a model to generate a

[1]We have not fully verified that this is true.

[2]Not fully implemented yet

sequence of queries satisfying the above characteristics. With probability α, this model chooses a uniformly random node from the graph to query, and with probability $1 - \alpha$, the model chooses a uniformly random node from a queue of recently-seen nodes. Right after the generation of a query, the queried node and its neighbors are enqueued into the recently-seen queue. We cap the size of the recently-seen queue at some size L, and once the queue's size exceeds L, our implementation dequeues from the beginning of the queue, indicating that these nodes are no longer "recent."

Note that sequential prefetching would not do particularly well on this workload. The edge table in the CSR is contiguous, so accessing a particular node's edges would trigger prefetching of the edges corresponding to the node with the next ID. However, in this workload, we are more likely to access the edge tables of that node's *neighbors*, who could have arbitrary ID and thus could be in a completely different part of disk.

The queries generated by this model fit the access pattern for which our policy is optimized, except that not all neighbors of queried nodes will get accessed — in fact, only a small portion of these neighbors are actually going to be used, and without omniscience, there is no way for the `madvise`ing thread to know which of the neighbors will actually be used. Hence, any `madvise`ing strategy must either be overly conservative, failing to `madvise` many useful edge tables, or wasteful, issuing `madvise` calls for many unused edge tables. We found that `madvise`ing the high-degree nodes is a better solution than the two extremes, as high-degree nodes are accessed more frequently; however, we still find that many useless edge tables are prefetched, and we are still working on tuning our advising policy to maximize performance.

B. Friend of Friends

Another workload we considered was friends-of-friends queries. This workload processes a uniformly random stream of nodes and outputs all neighbors of neighbors of each node. The access pattern is non-sequential. While vertices' edge tables are contiguous when indexed by ID, a vertex's friend does not necessarily have a consecutive ID, and so its edge table could lie in an entirely different region of the graph's CSR representation. However, this workload fits well with the access pattern we investigated: all neighbors of accessed nodes will get accessed.

C. Shortest Path

The shortest-path query for an unweighted graph computes the directed shortest path distance between two randomly chosen vertices. We used a simple parallel two-end breadth first search (BFS). In one thread we perform a breadth-first search along out-edges from the source. We use the standard breadth-first search algorithm: we start with the source vertex in a queue local to the thread, and each iteration, we dequeue from the queue and update the distances of all unvisited neighbors of dequeued node. The other thread performs the analogous computation along in-edges from the destination. In both threads, after visiting each node, we check if that node has been visited from the other thread, and if so, we signal for both threads to terminate. The pseudocode is shown below.

Algorithm 1 Shortest Path Algorithm. The graph has N vertices.

```
function SHORTESTPATH(x, y)
    Initialize $N$-element array $d_s$ with -1
    Initialize $N$-element array $d_d$ with -1
    $found \leftarrow$ FALSE
    $dist \leftarrow$ MAXINT
    Source thread:
        Enqueue $x$ into $q_s$
        while $q_s$ is nonempty and not $found$ do
            Dequeue front element of $q_s$; store it in $n$.
            for all edges $(n, m)$ do
                if $d_s[m] \neq -1$ then
                    continue
                Critical
                    if $d_d[m] \neq -1$ and not $found$ then
                        $dist \leftarrow d_d[m] + d_s[n] + 1$
                        $found \leftarrow$ TRUE
                    $d_s[m] \leftarrow d_s[n] + 1$
                EndCritical
                Enqueue $m$

    Destination thread:
        // Same as above, swapping $d_s \leftrightarrow d_d$, $n \leftrightarrow m$.

    return $dist$
```

This algorithm in theory can perform significantly better than BFS from just the source: if the shortest-path distance between the source and destination is l, the BFS from just the source requires a depth-l BFS, whereas BFS from both the source and the destination requires, on average, two depth-$l/2$ BFS. Since the number of encountered nodes increases roughly exponentially with the level, two-end BFS encounters far fewer nodes before finding a path between the source and destination.

D. Sequential workloads

The original workloads in the LLAMA paper include implementations of breadth-first search (BFS), PageRank, and triangle counting [7]. These workloads can all be written in a purely sequential manner (with varying degrees of difficulty). We tested our sequential prefetcher on PageRank and triangle counting. We have not yet had the chance to experiment with a full BFS and plan to do so in the future. We used the same code as in the original LLAMA benchmarks [7]. As our prefetcher was not tested extensively for multiple computation threads, we removed parallelism from the original algorithms; in the future, we would like to see how our prefetcher works with multiple computation threads.

V. Evaluation

We ran these workloads on a Linux machine with a 3.2 GHz 6-core Intel i7 processor 12 GB of memory, and an SSD. We also have results for a MacBookPro; this data is presented in Appendix A.

We ran our workloads on a Twitter dataset from http://an.kaist.ac.kr/traces/WWW2010.html [3]. For each benchmark, we compare the performance of the original LLAMA system with our `madvise`-enhanced system. For all our workloads, the sequence of queries is generated ahead of time; in real world applications, the query stream is a given input for the server and the server does not have to do any computations to generate the query stream. The original LLAMA and our `madvise`-enhanced LLAMA receive exactly the same query sequence.

A. Edge Queries

We created a shell script that compared the query simulator workload with our asynchronous `madvise`ing strategy and without it. The script varied N, the number of vertices queried, between N ={1K, 5K, 10K, 20K, 50K, 100K, 200K, 500K, 1M}; and α, the probability of choosing a random node (as opposed to a recently-seen node), between $\alpha = \{0.1, 0.5, 1\}$. For each (N, α) pair, we perform 5 cold-cache trials, each time clearing the cache using `sh -c 'sync && echo 3 > /proc/sys/vm/drop_caches'`. For each trial, we create a query sequence of length N with parameter α using the query creator tool discussed in Section IV-A. Then, we run both the non-`madvise` and `madvise` versions of the database on that sequence and record the

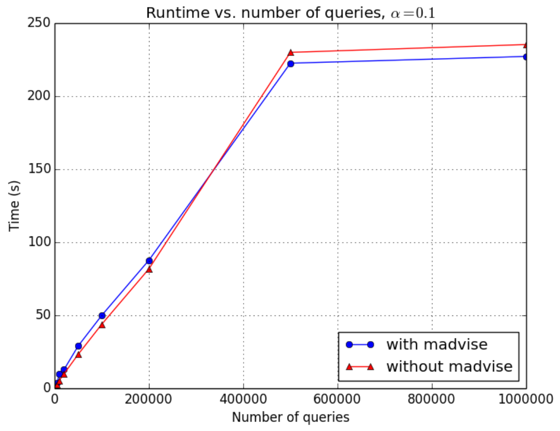

Fig. 1: Elapsed time for edge queries vs the number of queries. We set $\alpha = 0.1$. We see that in the first 200K nodes, i.e., the cold-cache period, the non-prefetching strategy outperforms the prefetching strategy. However, after this initial period, our prefetching strategy improves runtime by about 10%.

runtime, including the wallclock time and system / user time reported by `get_rusage`. Furthermore, in all the below results, the standard deviations in runtimes were quite low for all (N, α) pairs, ranging from 0 - 5%.

The results for $\alpha = 0.1$ represent a workload where users query a recently-queried node or its friends 90% of the time, and very seldom access a random node. Figure 1 shows the (wallclock) running times for the *with madvise* and *without madvise* strategies. From the graphs, it is apparent that up to $N = 200K$, the non-prefetching strategy performs better than the prefetching strategy. We refer to this period as the *cold-cache period*, when the buffer cache is initially filling up (this was verified by tracking memory usage explicitly while running the benchmarks). After the first 200K nodes, the `madvise`ing strategy become about 10% faster than the non-prefetching strategy.

The results for $\alpha = 0.5$ represent a workload where users query a recently-queried node or its friends 50% of the time and the other half of the time queries for a random node. The running times for the *with madvise* and *without madvise* strategies in Figure 2. Once again, for $N \leq 200K$, the *without madvise* strategy outperforms the *with madvise* strategy. However, for larger N, we see speedups of 36% for $N = 500K$ and 23% for $N = 1M$.

The results for $\alpha = 1$ represent a workload where users always query a random node. The running times for the *with madvise* and *without madvise* strategies are shown in Figure 3. We note that the *with madvise*

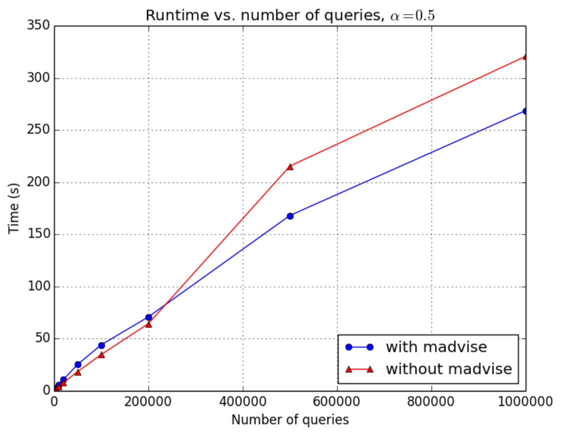

Fig. 2: Elapsed time for edge queries vs the number of queries. We set $\alpha = 0.5$. Once again, in the warmup period of 200K nodes, the non-prefetching strategy performs better. However, after this period, our prefetching strategy outperforms its non-prefetching counterpart by 35% at times.

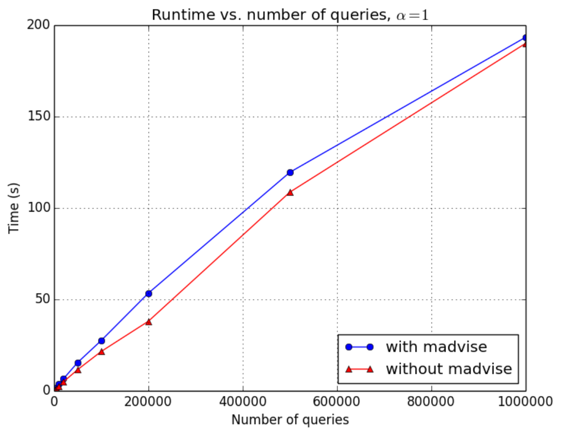

Fig. 3: Elapsed time for edge queries vs the number of queries. We set $\alpha = 1.0$, meaning that queries are uniformly random. The non-prefetching strategy outperforms the prefetching strategy, as expected.

strategy takes longer to run than the *without madvise* strategy, even though the runtimes do begin to converge after the initial cold-cache period of 200K nodes. This is what we expect since for a completely random sequence of nodes, there should be no performance gain from prefetching friends.

Note that the absolute time it takes to query sequences with $\alpha = 1$ is lower than the absolute time it takes for those with $\alpha = 0.5$ and $\alpha = 0.1$. This seemingly counterintuitive result is due to the fact that for low-

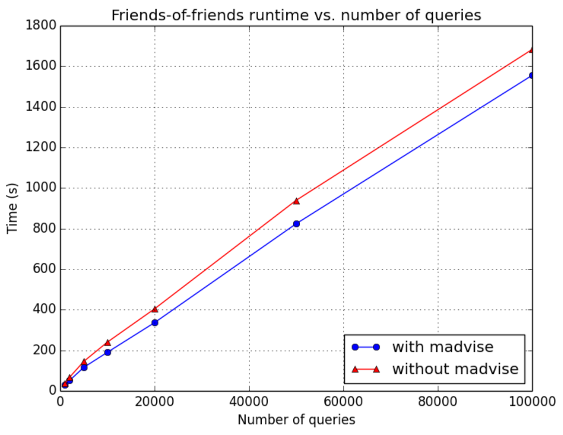

Fig. 4: Elapsed time for friends-of-friends vs the number of queries. The prefetching strategy outperforms the non-prefetching strategy for all numbers of queries.

α query sequences, neighbors of recently queried nodes are accessed with high probability, so nodes with lots of neighbors — i.e., high degree nodes — are queried with higher frequency. For instance, when $\alpha = 0.1$, the average degree is around 17000, when $\alpha = 0.5$, the average degree is around 10000, and when $\alpha = 1$, the average degree is around 50. Since it takes longer to go through all nodes adjacent to higher-degree nodes, completely random sequences ($\alpha = 1$) take less time to process than those that are more correlated.

B. Friends of Friends

We ran the friends-of-friends benchmark varying the number of vertices N for which we find their friends-of-friends. The parameter N takes values in the set {1K, 2K, 5K, 10K, 20K, 50K}. We ran 5 cold-cache trials for each N and used average runtimes and memory usage. For each trial, we used our query creator to generate a sequence of N random nodes, i.e., using $\alpha = 1$.

We see from runtime data in Figure 4 that the prefetching strategy outperforms the non-prefetching strategy for all values of N. For friends-of-friends, it turned out that the cold-cache effect occurred until 20K nodes. Interestingly enough, our data showed that there was about a 20% speedup for $N \leq 20K$ from prefetching while there was only 10% speedup for all nodes processed after this initial cold-cache period, i.e., the next 30K nodes for $N = 50K$ and the next 80K nodes for $N = 100K$.

C. Shortest Path

The shortest path benchmark involved choosing 5 pairs of random nodes and calculating the shortest path

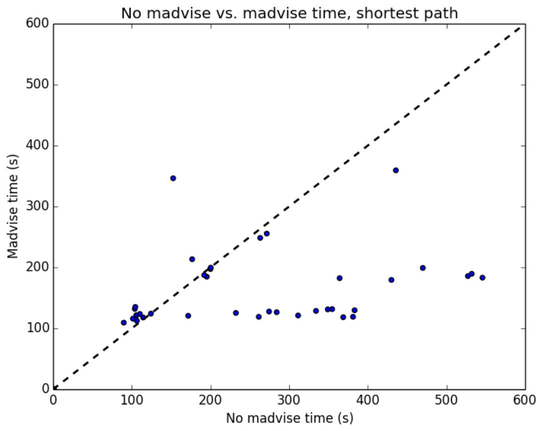

Fig. 5: Scatterplot of shortest-path running times, with *madvise* running time plotted against *no madvise* running time for each sequence. Three outliers have been removed. We see that most of the data points are below the superimposed line of slope 1, as desired.

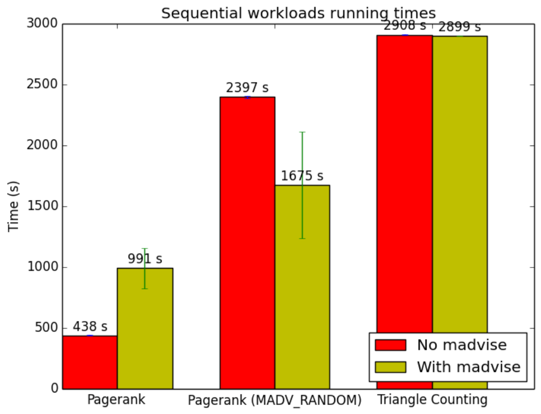

Fig. 6: Running times for various sequential workloads, with and without madvise. PageRank (MADV_RANDOM) refers to PageRank when we advise MADV_RANDOM to the kernel before running the workload.

between each pair. We ran 70 trials, with the same sequences of random nodes used for each trial using the *madvise* and *without madvise* strategies, respectively. In 30 of the 70 trials, the benchmark took less than 0.5 seconds to run, i.e., the nodes chosen randomly happen to be quite close together. In these cases, the `madvise` and non-`madvise` approaches have approximately the same performance. In 37 of the 70 trials, the benchmark takes between 1 minute and 10 minutes to run. Finally, in 3 of the 70 trials, the benchmark takes over 10 minutes to run.

The scatterplot in Figure 5 shows that for the majority of trials, the prefetching strategy is faster or the same speed as the non-prefetching strategy. Indeed, for the 37 trials that took between 1 minute and 10 minutes to run, the prefetching strategy was on average 21.7% faster than its non-prefetching counterpart, with some trials showing up to a 75% speedup. Indeed, we notice a horizontal cluster of points in Figure 5, suggesting that the performance of shortest path with madvise deteriorates at a slower rate with respect to query difficulty (i.e., degree of encountered nodes and distance of source and destination).

Interestingly, for the 3 very long jobs that we disregarded, i.e., when the nodes chosen happened to be far apart, the `madvise` strategy was about 2 or 3 times slower than the corresponding "no `madvise`" strategy. Though we have not concluded exactly why this slowdown occurs and only for long runs, we hypothesize that this discrepancy was due to the system state at the time, as another individual was using the machine for benchmarking during a few of the trials we were running.

D. Sequential Workloads

Figure 6 shows the performance of our prefetching iterator on PageRank and triangle counting, two sequential workloads considered in the original LLAMA paper [7]. We ran two slightly different versions of PageRank. In one version of PageRank, we issue an `madvise` call for the entire edge table with parameter `MADV_RANDOM`. This advises the kernel that access to the edge table is non-sequential (which is not actually the case), causing the kernel to disable the default sequential prefetching. As expected, we see the performance of PageRank drops dramatically as a result, showing both the effectiveness of the kernel's sequential prefetcher and the kernel's responsiveness to our `madvise` call.

We found that when we did not interfere with the kernel's sequential prefetcher, `madvise` did not improve performance; on the contrary, for PageRank, `madvise` imposed significant overhead, and for triangle counting, `madvise` improved performance by only 8 seconds (0.2%; this improvement is consistent). We experimented with various different ways of sequential `madvise`ing, but we were unable to significantly improve performance. We hypothesize that this is because the kernel's sequential prefetcher is already effective enough, so `madvise`ing could only add overhead. Indeed, we note that when we disable the kernel's sequential prefetcher by issuing the `MADV_RANDOM` call, `madvise` did significantly improve performance. This finding is in

	No Spinning Thread	Spinning Thread
Wallclock Time	12.020 s	9.996 s
User Time	3.836 s	3.098 s
System Time	3.445 s	2.405 s
CPU Time	7.281 s	5.503 s
Task-Clock	7611.8	6300.383
Context switches	37829	37811
CPU migrations	10	15
Page faults	39555	39549
Cycles	16,915,626,185	18,862,757,086
Stalled cycles (frontend)	15,245,113,422	16,846,710,648
Stalled cycles (backend)	11,026,710,293	11,025,285,331
Instructions	3,137,249,542	3,703,927,496
Branches	676,770,359	785,759,171
Branch misses	6,064,110	5,105,940

TABLE I: Performance measurements for benchmark, with and without idle spinning thread. We ran edge queries with 20000 queries on Linux. The first four statistics are reported by the benchmark; user and system time are obtained from get_rusage. All remaining numbers are obtained from the Linux perf stat tool.

contrast to the findings in Yacoby et al's paper [15], who found (surprisingly) that sequential `madvise`ing can improve upon the kernel's prefetcher due to the fact that the kernel prefetcher was not optimized for SSD.

One problem we encountered was significant variance in the running times for PageRank when we issue `madvise` calls. We found that sometimes, issuing `madvise` calls did not improve upon performance at all, even when we issued `MADV_RANDOM` before hand, and other times, we achieved a 50% improvement on performance. We did not encounter this issue with triangle counting. One hypothesis is optimization-related bugs with our madvising thread: we found a bug in which we forgot to declare various variables to be `volatile`; adding this declaration removed a significant amount of variance for many of our workloads.

E. Anomaly with Spinning Threads

While running our benchmarks, we noticed that starting a spinning thread in the background can reduce the wall clock time, user time, and system time of our benchmark on both Mac OS and Linux. We first witnessed this effect when trying to measure the effect of memory pressure on our buffer management system. We wrote a memory hog that would `malloc` a user-specified amount of memory and cycle through it repeatedly to make sure none of the memory could be evicted. Quite unexpectedly, we found that all benchmarks take statistically significantly less time (as measured both by wall clock time and by `get_rusage`) to complete. This effect is most significant for Macs (almost 20% time reduction), but is also present on Linux (at worst, about 16%).

To eliminate confounding variables such as other processes, we conducted tests on our MacBook using both single-user mode and safe mode, but the effect is still present in the same magnitude. We also conducted tests in single-user mode on a Linux machine (different from the one on which we did our benchmarks), which did not change any results. One specific hypothesis we tested on the Linux machine was that the presence of an idle spinning thread causes our computation thread to be pinned to one CPU, thereby reducing thread migration costs (unfortunately, Macs do not provide a mechanism to pin a thread to a CPU core, preventing us from testing this hypothesis on Macs). However, when we used the `perf stat` tool on the Linux machines to measure system statistics while running our benchmark, we found no differences in CPU migrations, page faults, or other important metrics, yet benchmarks consistently ran 10% faster (we also attempted to collect cache miss information, but we found that collection of this metric significantly altered performance). Sample performance measurements for our edge-queries workload showing this effect are presented in Table I. As of December 2014, this issue has not been completely resolved.

VI. Conclusion and Future Work

We have found that introducing a buffer management strategy can improve performance for LLAMA on non-sequential workloads. The specific mechanism we considered was an asynchronous `madvise`ing thread that issues `madvise` calls according to some policy. We found that by issuing `madvise` calls for neighbors of recently queried nodes, we can improve the performance of workloads like edge queries, friend-of-friends, and shortest-path queries. On sequential workloads, the results were more mixed, as we were not able to improve on the kernel's default sequential prefetcher.

In the future, we want to more fully understand our results. For each of the benchmarks, we want to see how our prefetching algorithms compare to an omniscient prefetcher that knows exactly what queries will come in the future. Furthermore, we want to profile the warm-cache vs. cold-cache effects of our prefetching advisors by examining the latencies at different points in the execution. We had attempted to do this, but it seemed to interfere with the timing of the benchmarks we had been running, and we did not have sufficient time to analyze why these discrepancies occurred. In the future, we definitely want to separate out these speedup effects.

Furthermore, we want to see how parallelism in the computation affects the prefetching thread and the performance improvement it can create. Finally, we wish to investigate if `madvise`ing `DONT_NEED`s can improve runtime and memory usage of these benchmarks. Currently, we have only been advising the kernel on prefetching and not at all on eviction. However, the problem of eviction is coupled with the problem of prefetching: when too many items are prefetched, the kernel must evict something, and a poor eviction strategy can ruin the performance brought by prefetching. A key part of buffer management is to be able to limit the amount of buffer cache used, and perhaps, this broader approach is a way to accomplish this goal.

Appendix A
Evaluation on MacBook Pro

We ran all the following workloads on a MacBook Pro with a 2GHz Intel Core i7 processor, which contains 4 hyperthreaded cores (2 threads per core), 8 GB of 1600 MHz DDR3 memory, and an SSD. We performed cold-cache experiments by using the `purge` command.

A. *Edge queries*

We note that for $\alpha = 0.1$, $\alpha = 0.5$, and $\alpha = 1$ (in Figures 7, 8, and 9), there is almost a 30% improvement in runtime on this machine, which is quite a bit bigger than the overall runtime improvement on the Linux machine. Some of this performance gain might come from simplying having an idle thread, as discussed in Section V-E; however, we found the speedup effect to be larger than the speedup due to a pure spinning thread, suggesting that madvise could be effective.

B. *Friends of friends*

Similarly, for our friends-of-friends, we have huge performance improvements, sometimes as large as 60% using `madvise`. We can see the results for various numbers of vertices in Figure 10.

Acknowledgment

We would like to thank Peter Macko for his help with the LLAMA code base and his advise on prefetching strategies; Daniel Margo for giving us access to his Linux machine for us to run our tests; and Margo Seltzer for her guidance throughout the project.

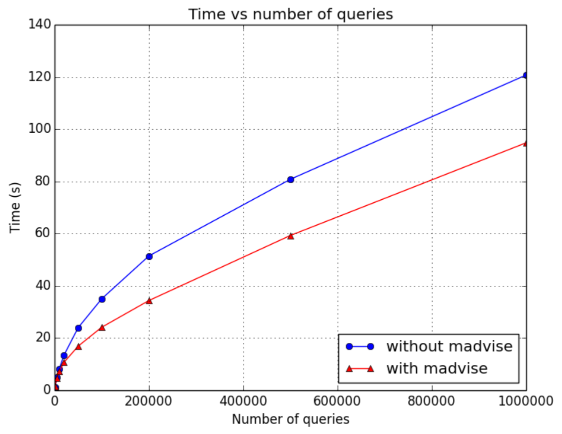

Fig. 7: Elapsed time for edge queries vs the number of queries. with $\alpha = 0.1$.

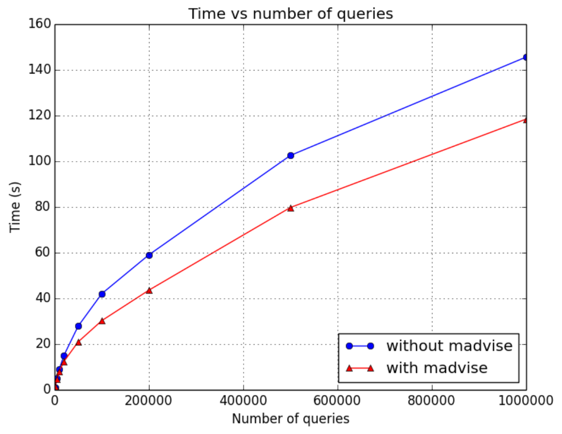

Fig. 8: Elapsed time for edge queries vs the number of queries. with $\alpha = 0.5$.

References

[1] J.E. Gonzalez, Y. Low, H. Gu, D. Bickson, C. Guestrin. *PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs*. OSDI, 2012.

[2] 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*. ACM, 2013.

[3] H. Kwak, C. Lee, H. Park, and S. Moon. *What is Twitter, a Social Network or a News Media?*. Proceedings of the 19th International World Wide Web (WWW) Conference, 2010.

[4] A. Kyrola, G. Blelloch, and C. Guestrin. *GraphChi: large-scale graph computation on just a PC*. OSDI, 2012.

[5] A. Kyrola and C. Guestrin. *GraphChi-DB: Simple Design for a Scalable Graph System – on Just a PC*. Preprint. arXiv:1403.0701

[6] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J.M. Hellerstein. *Distributed GraphLab: A Framework for Machine Learning and Data Mining in the Cloud*. CoRR, 2012.

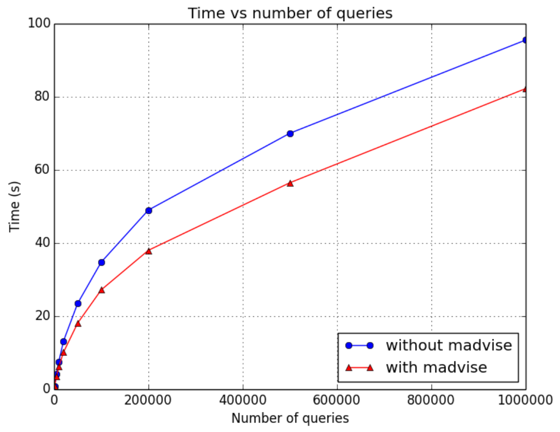

Fig. 9: Elapsed time for edge queries vs the number of queries. We set $\alpha = 1.0$.

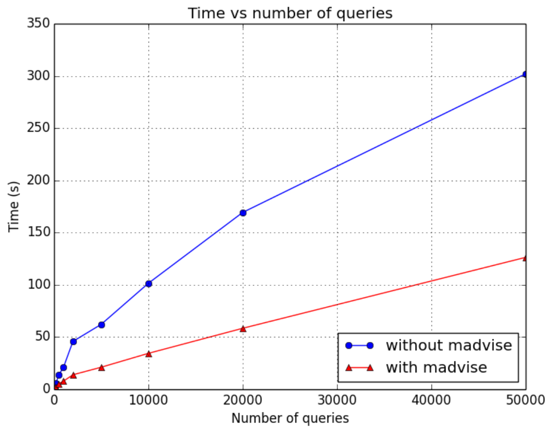

Fig. 10: Elapsed time for friend-of-friend queries.

[7] P. Macko, V.J. Marathe, D.W. Margo and M.I. Seltzer. *LLAMA: Efficient Graph Analytics Using Large Multiversioned Arrays*. Manuscript, 2014.

[8] 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*. SIGMOD, 2010.

[9] N. Martínez-Bazan, V. Muntés-Mulero, S. Gómez-Villamor, J. Nin, M.M.A. Sánchez-Martínez, J. Larriba-Pey. *DEX: High-Performance Exploration on Large Graphs for Information Retrieval*. CIKM, 2007.

[10] Neo4j - The World's Leading Graph Database. http://neo4j.com.

[11] R. Steinhaus, D. Olteanu, T. Furche. *G-Store: A Storage Manager for Graph Data*. Ph.D. dissertation, Citeseer, 2010.

[12] B. Shao, H. Wang, Y. Li. *Trinity: A Distributed Graph Engine on a Memory Cloud*. SIGMOD, 2013.

[13] J. Shun and G.E. Blelloch. *Ligra: a lightweight graph processing frameowrk for shared memory*. POPP, 2013.

[14] L. Torvalds. Madvise source code. https://github.com/torvalds/linux/.

[15] Y. Yacoby. *Madvise: Let Your Inner Oracle Shine*. Manuscript, 2014.