LiveGraph: A Transactional Graph Storage System with Purely Sequential Adjacency List Scans

Xiaowei Zhu[1], Guanyu Feng[1], Marco Serafini[2], Xiaosong Ma[3], Jiping Yu[1], Lei Xie[1], Ashraf Aboulnaga[3], and Wenguang Chen* [1,4]

[1]Tsinghua University, [2]University of Massachusetts Amherst, [3]Qatar Computing Research Institute
[4]Beijing National Research Center for Information Science and Technology
[1]{zhuxiaowei,cwg}@tsinghua.edu.cn; {fgy18,yjp19,xie-l18}@mails.tsinghua.edu.cn;
[2]marco@cs.umass.edu; [3]{xma,aaboulnaga}@hbku.edu.qa

ABSTRACT

The specific characteristics of graph workloads make it hard to design a one-size-fits-all graph storage system. Systems that support transactional updates use data structures with poor data locality, which limits the efficiency of analytical workloads or even simple edge scans. Other systems run graph analytics workloads efficiently, but cannot properly support transactions.

This paper presents LiveGraph, a graph storage system that outperforms both the best graph transactional systems *and* the best solutions for real-time graph analytics on fresh data. LiveGraph achieves this by ensuring that adjacency list scans, a key operation in graph workloads, are *purely sequential*: they never require random accesses even in presence of concurrent transactions. Such pure-sequential operations are enabled by combining a novel graph-aware data structure, the *Transactional Edge Log (TEL)*, with a concurrency control mechanism that leverages TEL's data layout. Our evaluation shows that LiveGraph significantly outperforms state-of-the-art (graph) database solutions on both transactional and real-time analytical workloads.

PVLDB Reference Format:
Xiaowei Zhu, Guanyu Feng, Marco Serafini, Xiaosong Ma, Jiping Yu, Lei Xie, Ashraf Aboulnaga, and Wenguang Chen. LiveGraph: A Transactional Graph Storage System with Purely Sequential Adjacency List Scans. *PVLDB*, 13(7): 1020-1034, 2020.
DOI: https://doi.org/10.14778/3384345.3384351

1. INTRODUCTION

Graph data is one of the fastest-growing areas in data management: applications performing graph processing and graph data management are predicted to double annually through 2022 [1]. Applications using graph data are extremely diverse. We can identify two broad classes of graph workloads with different requirements: *transactional graph data management* and *graph analytics*.

Transactional graph data management workloads continuously update and query single vertices, edges, and adjacency lists.[1] Facebook, for example, stores posts, friendship relationships, comments, and other critical data in a graph format [12, 20]. Write transactions incrementally update the graph, while read transactions are localized to edges, vertices, or the neighborhood of single vertices. These applications require a graph storage system to have very low latency and high throughput, be it a key-value store, a relational database management system, or a specialized graph database system. The system must also have classic transactional features: concurrency control to deal with concurrent updates and durability to persist updates.

Graph analytics tasks run on a consistent read-only graph snapshot and their performance highly depends on efficient scans of the neighborhood of a vertex (i.e., the adjacency list of the vertex). A particular class of analytics, *real-time* analytics on *fresh dynamic* graph data, is becoming increasingly important. For example, consider recommendations, where a website needs to find relevant products/connections based on users' properties and most recent interactions, which reflect their interests at the moment.

Other applications in this class include privacy-related data governance (where "expired" data needs to be excluded from analytics for GDPR compliance), finance (where financial institutions establish if groups of people connected through common addresses, telephone numbers, or frequent contacts are issuing fraudulent transactions), or systems security (where monitoring systems detect whether an attacker has performed a sequence of correlated steps to penetrate a system).

It is increasingly attractive to have a graph storage system that simultaneously supports both transactional and (real-time) analytical workloads. Unfortunately, common data structures adopted separately in the two worlds do not fare well when crossing into unfamiliar territory.

Data structures used in state-of-the-art DBMSs and key-value stores do not support well adjacency list scans, a crucial operation in graph analytics and graph database queries.

*Wenguang Chen is the corresponding author.

This work is licensed under the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nc-nd/4.0/. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.
Proceedings of the VLDB Endowment, Vol. 13, No. 7
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3384345.3384351

[1]We call a workload "transactional" if it consists of simple read/write operations that must be interactive and require very low latency, regardless of whether the transactions access only one object or multiple objects atomically.

More specifically, popular structures such as B+ trees and Log-Structured Merge Trees (LSMTs) yield significantly worse performance in graph analytics than graph-aware data structures like Compressed Sparse Rows (CSR) [52]. We performed micro-benchmarks and a micro-architectural evaluation comparing alternative data structures for storing graph data, and in particular adjacency lists. The results show that contiguous in-memory storage of adjacency lists not only improves caching efficiency, but also allows better speculation and prefetching, reducing both memory access costs and the number of instructions executed.

At the other end of the spectrum, analytical graph engines often use sequential memory layouts for adjacency lists like CSR. They feature efficient scans but do not support high-throughput, low-latency concurrent transaction processing. In fact, most existing graph engines do not target mutable graphs at all. Adding concurrency control to deal with concurrent updates is not straightforward. The concurrency control algorithm is on the critical path of every operation and thus directly impacts the performance of adjacency list scans. It should not disrupt otherwise sequential scans with random accesses or a complex execution flow. There has been much recent work on in-memory concurrency control and transactional support for relational data [28, 39, 42, 45, 59, 61], but none of the existing studies has specifically targeted the unique requirements of graph workloads.

This paper is a first step towards filling this gap. It proposes LiveGraph, a graph storage system supporting both transactional and (real-time) analytical workloads. A key design goal of LiveGraph is to ensure that adjacency list scans are *purely sequential*, that is, they never require random accesses even in the presence of concurrent transactions. To this end, we *co-design* the system's graph-aware data structure and its concurrency control algorithm. LiveGraph stores adjacency lists in a new data structure called the *Transactional Edge Log* (TEL). The TEL combines multi-versioning with a sequential memory layout. The concurrency control of LiveGraph leverages the cache-aligned timestamps and counters of the TEL to preserve the sequential nature of scans even in the presence of concurrent transactions. It is an efficient yet simple algorithm whose regular execution flow enables speculation and prefetching.

Our evaluation compares LiveGraph with several state-of-the-art systems, including specialized graph databases, graph database solutions built on key-value stores or traditional RDBMSes, and graph engines. Results demonstrate that LiveGraph outperforms the current leaders at their specialty, in particular outperforming Facebook's RocksDB [2] by up to 7.45× using Facebook's social graph benchmark [12]. In addition, LiveGraph dramatically outperforms (up to 36.4× better than the runner-up) all systems that have implemented the LDBC SNB interactive workload [27], ingesting updates and performing real-time analytics queries concurrently. We further perform micro-benchmarking and extensive profiling to understand the performance differences. Finally, LiveGraph allows lower end-to-end processing time by conducting in-situ iterative graph analytics (like PageRank) on its latest snapshot, as the expensive ETL cost can now be eliminated. More details can be found in [66].

2. PURELY SEQUENTIAL SCANS

A key design choice of LiveGraph is ensuring purely sequential adjacency list scans: scans should never entail random accesses. Before introducing the details of LiveGraph, we motivate why purely sequential adjacency list scans are important. We use single-threaded micro-benchmarks and micro-architectural analysis to compare different commonly used data structures and quantify the advantage of a sequential memory layout. Then, we discuss how concurrency control algorithms introduce additional complexity in the form of random accesses and branching.

Table 1: Adjacency list scan properties of different data structures. N is the size of the tree.

Cost	Seek		Scan (per edge)
B+ Tree	$O(\log N)$	random	sequential w. random
LSMT	$O(\log N)$	random	sequential w. random
Linked List	$O(1)$	random	random
CSR	$O(1)$	random	sequential
TEL	$O(1)$	random	sequential

2.1 The Benefits of Sequential Edge Storage

Adjacency lists contain the key topological information in a graph. Full or partial scans of these lists are fundamental operations in graph workloads, from simple queries to full-graph analytics. Graph storage must balance fast scans with efficient edge insertions, which are frequent in graph writes [20, 12]. In the following, we compare the scan performance of different data structures used for graph storage.

Graph data representations. Graph data consists of two types of objects: vertices and edges.

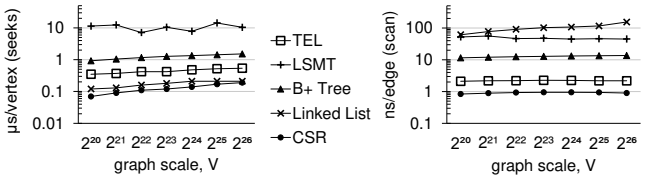
The CSR representation consists of two arrays, the first storing the adjacency lists of all vertices as sequences of destination vertex IDs, while the second storing pointers to the first array, indexed by source vertex ID. CSR is very compact, leading to a small storage footprint, reduced memory traffic, and high cache efficiency. Also, unlike most other data structures, it enables pure sequential adjacency list scans. These properties make it a top choice for graph engines [46, 31, 65], which target read-only analytical workloads. On the flip side, it is immutable, making it unsuitable for dynamic graphs or transactional workloads.

Linked list is an intuitive choice for adjacency lists and is used by Neo4j [3], a popular transactional graph database. It easily supports edge insertions but suffers from random accesses during scans when traversing through pointers.

Other state-of-the-art graph stores adopt general-purpose data structures such as the B+ tree and the LSMT (Log-Structured Merge-Tree). The adjacency list is represented as a single sorted collection of edges, whose unique key is a ⟨`src`,`dest`⟩ vertex ID pair.

In this work, we propose a new data structure, the Transactional Edge Log (TEL), which simultaneously allows sequential adjacency list scans and fast edge insertion. Unlike existing structures used in graph systems, it features purely sequential, yet mutable, edge storage. For this discussion, it suffices to say that edges in the same adjacency list are stored sequentially in contiguous blocks with empty slots at the tail. Edge insertions and updates are appended at the tail of the block until it fills up, at which point the TEL is upgraded to a larger block. Like CSR, TEL has purely sequential adjacency list scans.

Table 1 compares major operation complexities of the aforementioned data structures. Each adjacency list scan consists of a one-time *seek* operation, which locates the first



(a) Seek latency **(b)** Edge scan latency

Figure 1: Adjacency list scan micro-benchmark

edge of the adjacency list, followed by an *edge scan*, a sequence of edge accesses. Note that the initial seek cost often cannot be amortized, especially considering that most real-world graphs exhibit power-law degree distributions so most vertices have few edges. For a more detailed comparison, we refer to a comprehensive survey of data structures used in graph databases [18].

Micro-benchmark results. To see the impact of data structure choices on actual graph systems, we use a micro-benchmark that performs 10^8 adjacency list scans, where each start vertex is selected randomly under a power-law distribution. Graphs are generated using the Kronecker generator [41] with sizes ranging from 2^{20} to 2^{26} vertices, and an average degree of 4 (similar to those generated by Facebook's LinkBench [12] benchmark). All graphs fit in the memory of a single socket. For accurate cache monitoring, we perform single-thread experiments, with the workload driver running on a separate socket.

We evaluate LMDB [4] and RocksDB [2], embedded storage systems that adopt B+ trees and LSMTs respectively. To fairly isolate the impact of data structure choices, we disable compression in RocksDB, and also implement an efficient in-memory linked list prototype in C++ rather than running Neo4j on a managed language (Java). For reference, we also include CSR, which is widely used by state-of-the-art graph engines, though CSR is read-only.

We consider two metrics: seek latency and edge scan latency. Seek latency is the time needed to locate the adjacency list of a vertex. Edge scan latency is the time required to access the next edge in the adjacency list. Figure 1 shows that using different data structures results in orders of magnitude gaps in these metrics.

The sequential data layout of TEL is clearly superior to other pointer-based data structures. To locate the first edge of the range (seek), B+ trees have a logarithmic number of random accesses. RocksDB's implementation of LSMTs uses a skip list as memtable, which has a similar behavior for seeks. However, LSMTs also require reading the (on-disk) SST tables for seeks, since only the first part of the edge key is known (the source vertex ID) while the second part is unknown (the destination ID). This explains the bad performance of LSMT in Figure 1a. For linked-lists and TELs we consider one data structure instance per adjacency list, as done in Neo4j and LiveGraph, respectively. For CSR, the beginning of an adjacency list is stored in the offset array. Reading it requires only a constant-time index lookup.

Systems using B+ trees and LSMTs store edges in a single sorted collection, which corresponds to a single edge table. An adjacency list scan becomes a range query where only the first component of an edge key, the source vertex ID, is given. To iterate over all edges of a vertex (scans), a B+ tree sequentially scans elements of the same node but it needs random accesses whenever an adjacency list spans multiple nodes. LSMTs require scanning SST tables also for scans because, similar to seeks, only the first component of the edge key is known. Skip Lists and Linked List require random accesses for each edge scan. By contrast, scans in TEL and CSR are always sequential. Figure 1b shows that TEL has a scan speedup larger than 29× over linked list, 20× over LSMT, and 6× over B+ tree. CSR's latency for scans is 43% that of TEL. The gap is mainly from TEL's larger memory footprint and the overheads of checking edge visibility to support transactions (our double timestamps design, which we will discuss later).

We performed a more detailed micro-architectural analysis on the 2^{26} scale graph to further understand the behavior of data structures. B+ trees and LSMTs trigger 7.09× and 11.18× more last-level cache misses than TEL. Linked Lists with mostly random memory accesses are the worst, incurring 63.54× more LLC-misses than TEL. TEL has a simpler sequential execution flow, leveraging CPU pipelining+prefetching and reducing branch mispredictions. This profiling confirms the huge gap between pointer-based data structures and a sequential data structure like TEL. Compared with CSR, TEL triggers 2.42× more LLC-misses due to our memory footprints: a single edge in TEL takes 2× memory than CSR in our micro-benchmark.

In total adjacency list scan latency, TEL yields, on average among different graph scales, a 22× performance improvement over LSMT, 46× over linked list, 5.6× over B+ tree, and 40% higher than CSR. For seeks, there is a significant gap between tree-based (logarithmic) and graph-aware (constant) data structures. For scans, TEL performs much better as its accesses are purely sequential, while others involve random accesses and additional branching.

These results show that existing dynamic data structures used by transactional systems leave a lot of performance on the table for graph workloads, which can be harvested by using a graph-aware storage layout like TEL.

2.2 Transactions with Sequential Access

The previous experiments show the importance of eliminating random accesses during adjacency list scans through graph-aware data structures. However, they consider a single-threaded setting without concurrency. In transactional workloads, it is necessary to preserve the benefits of sequential scans in the presence of concurrent transactions.

Real-time graph analytics feature read-only transactions that need to access a consistent snapshot of the graph. These transactions may be long-running and access a large fraction of the graph, but they should not hinder the progress of concurrent transactions. Multi-versioning is a common approach to ensure this property and many efficient concurrency control algorithms have been proposed for relational databases [28, 39, 42, 45, 61]. However, they target data with flat layouts (i.e., indexed records). Representing each adjacency list as a whole record with multiple versions would make edge updates prohibitively expensive because every time the entire list would have to be written again.

Enabling multi-versioned sequential accesses to adjacency lists is to our knowledge an open topic encompassing multiple challenges: (1) different versions of edges belonging to the same adjacency list must be stored in contiguous memory locations; (2) finding the right version of an edge should

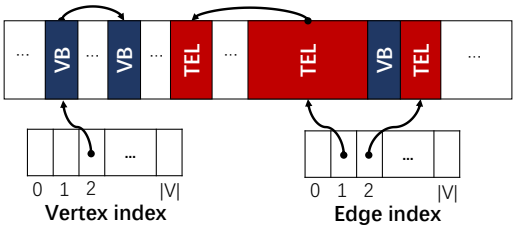


Figure 2: LiveGraph data layout. For simplicity, here we omit label index blocks and the vertex lock array.

not require auxiliary data structures that, in turn, require random access to be visited; (3) the concurrency control algorithm should not require random access during scans.

LiveGraph is the first system that guarantees these properties, achieved by co-designing a graph-aware data structure (Section 3) and the concurrency control algorithm (Sections 4 and 5) to ensure purely sequential scans even in the presence of concurrent transactions.

3. LIVEGRAPH DATA LAYOUT

LiveGraph implements both in-memory and out-of-core graph storage on a single server. It adopts the *property graph model* [51], where each object (vertex or edge) can have associated properties (i.e., arbitrary key-value pairs).

Edges have a special type of property called *label*. Each edge can have only one label. Edges that are incident to the same vertex are grouped into one adjacency list per label. Labels can be used to separate edges that are scanned separately, e.g., "is-friend-with" and "has-posted" edges in a social graph. For simplicity, our discussion depicts the case where all edges have the same label.

Edge storage is particularly critical since (1) usually graphs have more edges than vertices and edge operations are more frequent [20], and (2) efficient edge scan is crucial, as shown earlier. Therefore, LiveGraph adopts a 2-D approach: vertices are stored individually whereas all the edges incident on the same vertex are grouped in a single Transactional Edge Log (TEL).

Figure 2 shows the data structures of LiveGraph, which mainly consist of *vertex blocks* (VB) and *TELs*. These are stored in a single large memory-mapped file managed by LiveGraph's memory allocator. The blocks are accessed via two index arrays, a *vertex index* and an *edge index*, storing pointers to appropriate blocks by vertex ID. Though not depicted in Figure 2, there is an additional level of indirection between the edge index and TELs, called *label index blocks*, used to separate the storage of per-vertex edges with different labels. Since vertex IDs grow contiguously, we use extendable arrays for these indices. We have not found this to be a limiting factor in any of our experiments.

Vertices. LiveGraph stores each vertex separately into the vertex block. Updates to vertices are relatively infrequent and transactions typically access the latest version. Therefore, for vertices we use a standard copy-on-write approach, where the newest version of the vertex can be found through the vertex index, and each version points to its previous version in the vertex block.

Adjacency lists. A TEL is a fixed-size memory block with free space that is resized when filled. Different versions of a TEL are linked with "previous" pointers like vertex blocks. This organization combines efficient sequential scans of read-optimized formats for analytics (like CSR) with support for updates of dynamic arrays. Instead of just storing edges constituting the current adjacency list, a TEL represents all edge insertions, deletions, and updates as log entries appended at the tail of the log. Note that while our discussion focuses on using TEL for adjacency list storage, ideas proposed here can be used to implement a general key-value set data structure with sequential snapshot scans and amortized constant-time inserts.

The layout of a TEL block is depicted in Figure 3. Edge log entries are appended backwards, from right to left, and scanned forwards, from left to right. This is because many scan operations benefit from time locality, as in Facebook's production workload [12], where more recently added elements are read first. Edge log entries have fixed size with cache-aligned fields. This is required by our transaction processing protocol, as to be discussed later. Each entry has two timestamps, a creation timestamp and an invalidation timestamp, indicating its lifecycle. Edge properties have variable lengths and are appended from the beginning of the block forwards. These property entries are stored separately from the edge log entries to preserve the data layout alignment of the latter, again as required by transaction processing. Their content is opaque to LiveGraph.

For a new vertex, its adjacency list starts small, with 64-byte blocks that accommodate a single edge in our implementation. When a block is full, LiveGraph copies the log to an empty block of twice its current size. Similar to dynamic arrays, appends to the log have a constant amortized cost. The worst-case cost is linear, but copying contiguous blocks of memory is fast and does not result in high tail latency, as our evaluation shows.

This design is particularly suited to power-law degree distributions that are common in real-world graphs. The majority of vertices, being low-degree and less active, will grow slowly in degree, with limited space waste. The high-degree vertices are likely to grow faster and incur higher data copying costs when "upgraded", but such relocation happens at decreasing frequency with exponentially growing block sizes. The power-law degree distribution also implies that only a small fraction of vertices occupy large log blocks. We describe the details of LiveGraph's memory management and log relocation in Section 6.

4. SINGLE-THREADED OPERATIONS

We first describe LiveGraph's operations in the absence of concurrency. LiveGraph uses a multi-versioned data store and each transaction is associated with a read timestamp, which determines the version of the data it operates on. This timestamp is determined by the transaction processing mechanism (to described in Section 5).

Vertex operations. Vertex reads access the vertex index and then the vertex block; writes create a new version of the vertex block in the block storage, including a pointer to the current version, and set the vertex index to point to the new version. In the uncommon case where a read requires a previous version of the vertex, it follows the per-vertex linked list of vertex block versions in backward timestamp order until it finds a version with a timestamp smaller than its read timestamp.

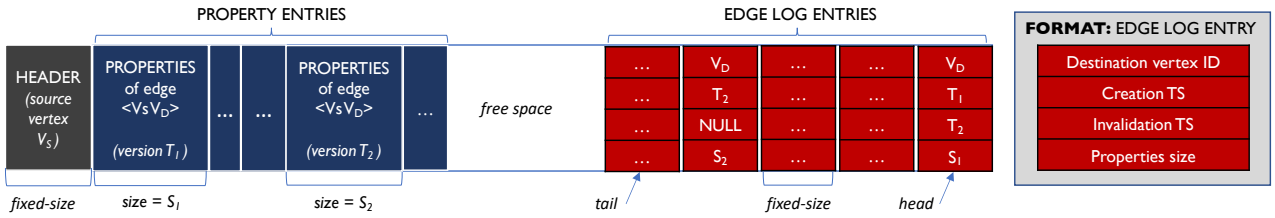


Figure 3: A simplified view of a Transactional Edge Log block. A TEL stores the adjacency list of a specific source vertex, indicated in the TEL header. Edge log entry fields are cache-aligned, as required by the transaction processing algorithm. In this example, the adjacency list is for vertex V_S and it contains five entries. An edge to vertex V_D is created at time T_1 (head entry). The edge is later updated at time T_2. The previous entry is invalidated and a new property block is appended.

Adding a new vertex first uses an atomic `fetch-and-add` operation to get the vertex ID, fills in empty pointers in the corresponding locations of vertex and edge indices, and sets the lock status. If the vertex needs to store properties or add edges, it asks the storage manager to allocate a block according to the size it needs, whose details will be shown in Section 6. Garbage collection to reuse the IDs of deleted vertices can be achieved by using techniques described later in Section 6. Since vertex deletions are rare, we leave the implementation of this mechanism to future work.

Sequential adjacency list scans. Scanning adjacency lists efficiently is a key requirement of analytics workloads. LiveGraph achieves purely sequential adjacency list scans by combining log-structured storage of the adjacency list and double timestamps.

A log structure is a more convenient multi-versioned representation of the adjacency list compared to alternative approaches. For example, a coarse-grained copy-on-write approach to multi-versioning would create a new copy of the entire adjacency list *every* time an edge is updated. This is the approach used by Grace [49]. However, it makes updates very expensive, especially for high-degree vertices.

Storing multiple versions of the adjacency list in contiguous memory locations as a log is key to achieving purely sequential adjacency list scans, but it is not sufficient. The same edge can now correspond to multiple entries in the log, as shown in Figure 3. When a thread executing an adjacency list scan reads an edge entry, it cannot tell whether the entry is still valid or if it has been deleted or modified at a later position in the log. The thread could keep a hash map to progressively update the latest version of each edge during the scan. But accessing the map would again require random accesses, which we strive to avoid.

To this end, LiveGraph stores a double timestamp for each edge. A read operation with timestamp T considers only edge entries such that T is within the entry's creation and invalidation timestamps. These two timestamps determine if the entry is valid for the read timestamp of the transaction (see Figure 3). Such a design makes TEL scans sequential: a transaction can check the validity of an edge entry simply by checking the embedded timestamp information. Scans that access edge properties require two sequential scans, one forwards from "tail" (as shown in Figure 3) to the right end, for edge log entries, and one backwards from the end of the property entries, for properties.

Edge updates and constant-time insertions. To support fast ingestion of new edges, LiveGraph inserts edges in amortized constant time. Insertions in a regular log-based data structure are simple appends, which can be done in constant time. TELs sometimes require resizing in order to append new entries, but the amortized cost of appends is still constant, like in dynamic arrays (see Section 3). The dual timestamp scheme of LiveGraph, while useful for sequential scans, makes updating the adjacency list more complex. Appending log entries is not always sufficient any longer. If an operation updates or deletes an existing edge, it must also update the invalidation timestamp of the previous entry for that edge, which entails scanning the log. However, if a new edge is inserted, the scan is not necessary and a constant-time append is still sufficient.

LiveGraph includes a Bloom filter in the TEL header to determine whether an edge operation is a simple insert or a more expensive update. Inserting a new edge appends an entry at the end of a TEL and updates the Bloom filter as well as the adjacency list block size. Edge deletions/updates first append a new entry to the TEL and check, using the Bloom filter, if a previous version of the edge is present in the TEL. If so, its invalidation timestamp needs to be updated. The Bloom filter is also handy to support fast "upsert" semantics (such as in the Facebook LinkBench workload [12]), where a lookup is needed to check whether the entity to be inserted already exists. It allows distinguishing which of them are "true insertions" that add new edges, such as "likes" in social networks or new purchases in online stores. Inserts often compose the majority of write operations and LiveGraph processes them in amortized constant time.

Upon a "possibly yes" answer from the Bloom filter, however, finding the edge itself involves a tail-to-head TEL scan, traversing the entire adjacency list in the worst case. In practice though, edge updates and deletions have high time locality: edges appended most recently are most likely to be accessed. They are the closest to the tail of the TEL, making average cost fairly low for edge updates/deletions.

Each Bloom filter is fixed-sized: 1/16 of the TEL for each block larger than 256 bytes. A blocked implementation [50] is adopted for cache efficiency considerations.

Reading a single edge. Reading a single edge involves checking if the edge is present using the Bloom filter. If so, the edge is located with a scan. As with adjacency list scans, the scan skips any entries having a creation-invalidation interval inconsistent with the read timestamp.

5. TRANSACTION PROCESSING

Next, we discuss concurrent execution of transactions, each consisting of one or more of the *basic read/write operations* described in Section 4. The major innovation of Live-

Graph here lies in its integrated design: unlike traditional MVCC solutions [61], which adopt auxiliary data structures to implement concurrency control, LiveGraph embeds the information required for transactional adjacency list scans within its main TEL data structure. This leads to significant performance advantages because it enables sequential access even in the presence of concurrent transactions.

We first present this integrated data layout, followed by the corresponding transaction management algorithms.

Data layout and coordination. LiveGraph stores adjacency lists as multi-versioned logs to efficiently support *snapshot isolation* [16], which allows read operations to proceed without interfering with each other and with write operations (proof of isolation guarantee in [66]). Snapshot isolation is stronger than *read committed*, the default isolation level of Neo4j and many other commercial DBMSs [15].

In LiveGraph, basic read operations on edges do not acquire locks. Coordination with basic write operations on edges occurs only through *cache-aligned 64-bit word timestamps*, written and read atomically. Cache-alignment is done by separating the variable-size edge properties and fixed-size edge log entries, which grow from the opposite ends of TEL blocks (Figure 3). Thus all timestamps in the log entries are cache-aligned. Basic read operations on vertices access different versions as described in Section 4.

Two additional cache-aligned per-TEL variables are used for coordination: the log commit timestamp `CT` and the log size `LS`. They are both stored in a TEL's fixed-size header.

Write-write conflicts are detected using per-vertex locks, implemented with a `futex` array of fixed-size entries (with a very large size pre-allocated via `mmap`). We also explored other choices, such as concurrent hashtables or spinlock arrays, but found the futex array method most scalable. For write-intensive scenarios when many concurrent writers compete for a common lock, spinning becomes a significant bottleneck while futex-based implementations utilize CPU cycles better by putting waiters to sleep. The space consumption for locks is acceptable: under 0.5% of the overall dataset size in our evaluation.

Transactions. LiveGraph keeps a pool of transaction-serving threads (henceforth referred to as "workers"), plus one transaction manager thread. All threads share two global epoch counters, `GRE` for reads and `GWE` for writes, both initially set to 0. They also share a reading epoch table to establish a safe timestamp for compaction.

A transaction has two local variables: a transaction-local read epoch counter `TRE`, which the worker initializes to `GRE`, and a transaction-local write epoch counter `TWE`, which is determined by the transaction manager at commit time. The worker assigns to the transaction a unique transaction identifier `TID`, by concatenating its unique thread ID and a worker-local logical transaction count.

Basic read operations access a snapshot determined by their `TRE`. When they read a vertex, transactions visit the linked list of vertex versions until they find the right version. In practice, a transaction needs to visit the list only in the rare case in which there are concurrent vertex writes. Reads need not acquire locks.

Adjacency list scans are strictly sequential. When they scan a TEL, read operations only consider edge log entries such that either `(0 <= CreationTS <= TRE) AND ((TRE < InvalidationTS) OR (InvalidationTS < 0))` (entry is valid at time `TRE`), or `(CreationTS == -TID) AND (InvalidationTS != -TID)` (a transaction sees its own earlier writes, more details below). The scan starts from the end of the log, using the `LS` size variable within the TEL header, and proceeds backwards towards the beginning.

Basic write operations initially store their updates in a transaction-private working version, in the same TEL as committed updates, to keep adjacency list scans sequential. Transaction-private updates are made visible to other transactions only after commit.

A write transaction goes through three phases: *work*, *persist*, and *apply*. In the work phase, it starts by computing its unique transaction identifier `TID` and acquiring write locks. To avoid deadlocks, a simple timeout mechanism is used: a timed-out transaction has to rollback and restart. The write transaction then executes write operations as discussed in Section 4, using the timestamp `-TID` to make the updates private. This phase concludes by calling the transaction manager to persist the changes.

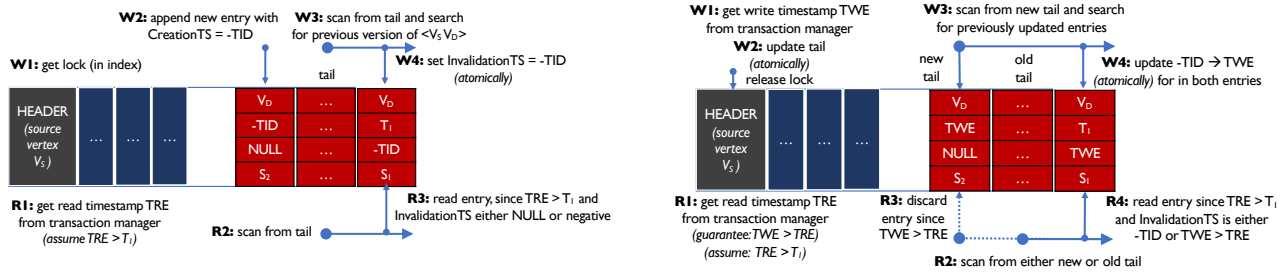
In the persist phase, LiveGraph adopts standard group commit processing [17] to enhance throughput. The transaction manager first advances the `GWE` counter by 1, then appends a batch of log entries to a sequential write-ahead log (WAL) and uses `fsync` to persist it to stable storage. Next, it notifies the involved transactions by assigning them a write timestamp `TWE`, set as `GWE`.

Now the write transaction enters the final apply phase, by updating the commit timestamp of the TEL, which is set to `TWE`, and the log size `LS` in the header. For vertex blocks, it also updates the version pointers and sets the vertex index pointer to the latest version. Next, it releases all its locks before starting the potentially lengthy process of making its updates visible by converting their timestamps from `-TID` to `TWE`. After all transactions in the commit group make their updates visible, the transaction manager advances the global read timestamp `GRE`, exposing the new updates to upcoming transactions. This also guarantees that the read timestamp of a transaction is always smaller than the write timestamp of any ongoing transaction.

Figure 4 shows an example of a write operation executed concurrently with reads in two of its phases (the persist phase is done by the transaction manager/logger and omitted due to space limitations). It shows that (1) read operations do not need to use locks and (2) multi-version transaction management is done as part of the sequential TEL scan. Both aspects significantly contribute to LiveGraph's processing speed and throughput.

To ensure consistent versioning, it is necessary to guarantee the invariant that each entry's invalidation timestamp is always larger than its creation timestamp. Write operations having a timestamp T must abort if they try to invalidate an entry with creation timestamp larger than T. LiveGraph stores the latest timestamp that has modified a TEL in a commit timestamp variable `CT` in the TEL header. This way, write operations can simply compare their timestamp against `CT` instead of paying the cost of scanning the TEL only to later find that they must abort.

Whenever a transaction aborts, it reverts the updated invalidation timestamps from `-TID` to `NULL`, releases all acquired locks, and returns any newly allocated blocks to the memory manager. An aborted transaction never modifies the log size variable `LS` so its new entries will be ignored by future reads and overwritten by future writes.



(a) Work phase **(b)** Apply phase

Figure 4: Example of concurrent access to the same TEL. A transaction is executing a basic write operation (see notes above the TEL) while another concurrent transaction executes a basic read operation (notes below) on the same TEL. Each operation is illustrated in steps. Concurrency can happen in the work/apply phase of the writing transaction. Accesses to cache-aligned timestamps are guaranteed to be atomic.

6. STORAGE MANAGEMENT

Currently, LiveGraph uses a memory-mapped file to store graph data, comprising vertex blocks, which store single vertices, and TEL blocks, each storing a TEL (see Figure 2), as well as per-vertex label index blocks storing pointers to TELs with different labels of each vertex. This allows LiveGraph to store graphs that do not fit in memory and rely on the operating system to decide how data is cached in memory and evicted. We plan to replace `mmap` with a managed page cache [40] to enable more robust performance on very large datasets backed by high-speed I/O devices.

Compaction. The TEL is not just an adjacency list but also, implicitly, a multi-version log record of the adjacency list with entries sorted by creation timestamps. Invalidated entries are useful for retrieving and analyzing historical snapshots, but their accumulation will eventually bloat the TEL size and impede in-memory processing. Therefore, LiveGraph performs periodic compaction.

LiveGraph provides the capability of a user-specified level of historical data storage, trading off disk space and checkpointing overhead, to allow full or partial historical snapshot analysis. In the prototype evaluated, we performed rather aggressive garbage collection (GC), without saving invalidated copies to disk. LiveGraph periodically (every 65536 transactions in our default setting) launches a compaction task. Each worker thread in LiveGraph maintains a *dirty vertex set*, marking vertices whose corresponding blocks have been updated since the last compaction executed within this thread. When doing compaction, a thread scans through its local dirty set and compacts or garbage-collects blocks based on *version visibility*. Each worker stores the transaction-local read epoch counter (the `TRE`) used by its *ongoing* transaction (if any) in a table with one entry per worker. *Future* transactions will get a `TRE` that is greater or equal to the current global read epoch counter (the `GRE`). A thread doing compaction accesses all these epoch counters to determine version visibility for all transactions.

The compaction processes one block at a time, asynchronously and independently, with only minimal interference with the regular workload. If a TEL block is not visible any longer to any ongoing or future transaction, it is garbage-collected. Otherwise, the thread removes all entries that will not be visible to future transactions. A thread first scans the block to compute the new capacity (sometimes the block could shrink after many edges being deleted). A new block is then allocated and only the entries visible to future transactions are copied (sequentially) to this new block. Like in a regular write transaction, the corresponding vertex lock of the TEL is held to temporarily prevent concurrent writes to that specific block. Writes to the new blocks are also committed like with write transactions. Once the copy to the new TEL block is committed, the compacting thread releases the lock and moves to the next vertex in the dirty set. New transactions that start after the commit access the new version. Ongoing transactions continue having read-only access to the old versions, which are kept until they are no longer visible by any transaction. At that point, in a future compation cycle, the thread garbage-collects them. Compaction only occurs in a lightweight vertex-wise fashion: unlike LSMT, LiveGraph never needs to compact multiple on-disk files through merging.

Compactions for vertex blocks are similar to existing MVCC implementations. Invalidated blocks that will never be visible to any existing or future transactions are simply garbage collected (to be reused later) and any related "previous pointers" are cleared simultaneously.

Space overhead for timestamps. Using two timestamps (which are not explicitly used by graph analytics algorithms themselves) dilutes TEL's storage density and lowers its bandwidth/caching efficiency compared to compact data structures such as CSR. This results in a performance gap between running analytics on top of LiveGraph compared to state-of-the-art engines for static graphs such as Ligra [54] or Gemini [65]. However, analytics on top of LiveGraph do not need to perform expensive ETL (Extract-Transform-Load) operations to load the graph into a dedicated tool. Compared to systems that support transactional graph updates and use pointer-based data structures, LiveGraph uses sequential memory regions and thus saves the storage cost of keeping many pointers, in addition to supporting multi-versioning. Overall, our evaluation shows that LiveGraph has a similar memory overhead as these systems.

Memory management. In selecting the adjacency list block size, there is a trade-off between the cost of repeated data relocation and space utilization. In making this decision, we seize a rare opportunity offered by the power-law degree distribution found in many real-world graphs [23, 29, 55]. Inspired by the buddy system [37], LiveGraph fits each TEL into a log block of the closest power-of-2 size.

LiveGraph has TELs starting from a size of 64 bytes (a 36-byte header plus a 28-byte log entry, whose contents were de-

scribed earlier). This minimal configuration accommodates one edge and occupies one cache line in common processors today. An array of lists `L` is used for keeping track of the free blocks in the block store, where `L[i]` ($i = 0, 1, ..., 57$) contains the positions of blocks with size equal to $2^i \times 64$ bytes. When a block of a certain size is needed, LiveGraph first checks the corresponding free list, allocating new blocks from the tail of the block store only when that list is empty. Vacated blocks or those that do not contain any valid data, meanwhile, are recycled into the proper free lists.

Again considering the power-law degree distribution, we accelerate the allocation process by differentiating the free list management of smaller and larger blocks. We define a tunable threshold m, with each thread maintaining its private free list array {`S[0], ..., S[m]`} and sharing a global free list array {`S[m+1], ..., S[57]`}. This significantly reduces the contention over the free lists for allocating highly popular small blocks, while mitigating waste by centralized large block management. Measurements on our 24-core (48 hardware threads) test platform show that block allocation is not a performance bottleneck (with m set at 14).

Recovery. The recovery mechanism of LiveGraph is similar to write-ahead-logging systems. A checkpointer (which can be configured to use any number of threads) periodically persists the latest consistent snapshot (using a read-only transaction) and prunes the WAL entries written before the epoch of the snapshot. When a failure happens, LiveGraph first loads the latest checkpoint and then replays the WAL to apply committed updates.

7. EVALUATION

7.1 Experimental Setup

Platform. We set up experiments on a dual-socket server, whose specification is given in Table 2. The persistence features are enabled for all the systems, except when specified otherwise. We evaluate two alternative SSD devices for persistence, with the Intel Optane as the default.

Table 2: Testbed specification

Processor	2-socket Intel Xeon Gold 6126 CPU 12 cores per CPU
Memory	192 GB DDR4 RAM
Storage	Intel Optane P4800X 750 GB SSD Dell Express PM1725a 1.6 TB SSD

Workloads. For lightweight graph accesses, we use LinkBench [12], a Facebook benchmark based on its social graph interactive workloads. Besides its default configuration (*DFLT*) with 69% read and 31% write operations, we add a "read-mostly" workload (*TAO*) with 99.8% of reads, with parameters set according to the Facebook TAO paper [20]. All experiments start on a 32M-vertex, 140M-edge base graph generated by the LinkBench driver, each client sends 500K query requests to the server.

For real-time graph analytics, we use the interactive workload in LDBC Social Network Benchmark (SNB) [27], which simulates the users' activities in a social network for a period of time. Its schema has 11 entities connected by 20 relations, with attributes of different types and values, providing a rich benchmark dataset. The SNB data generator is designed to produce directed labeled graphs that mimic the characteristics of real-world social graphs. We set 10 as the Scale Factor, with 30M vertices and 177M edges in the generated initial graph. Its requests are classified into three categories: short reads (similar to LinkBench operations), transactional updates (possibly involving multiple objects), and complex reads (multi-hop traversals, shortest paths, and analytical processing such as filters, aggregations, and joins). Finally, we run two popular iterative analytical algorithms on top of the generated graph, PageRank and Connected Components (ConnComp). PageRank runs for 20 iterations, while ConnComp runs till convergence.

Choice and rationale of baselines. For LinkBench, we first tested MySQL (v5.7.25) and MyRocks (v5.6.35) using their official adaptor, but found that inter-process communication between the server and client (benchmark driver) amplifies latencies. Thus we compare LiveGraph with three *embedded* implementations,[2] LMDB (v0.9.22), RocksDB (v5.10.3), and Neo4j (v3.5.4), as representatives for using B+ tree, LSMT, and linked list respectively. This way we focus on comparing the impact of data structure choices. For Neo4j, we use its Core API rather than Cypher, to eliminate potential query language overhead.

For SNB, besides graph databases including Neo4j [3], DBMS S, and DBMS T (anonymized due to licensing restrictions), we also compared with PostgreSQL (v10.7) [5] and Virtuoso (v7)[3] [6], two relational databases. DBMS S is based on the RDF model and uses a copy-on-write B+ tree similar to LMDB as the storage backend; DBMS T is a commercial graph database that provides the highest performance among graph databases according to benchmark reports; PostgreSQL is among the most popular relational databases for OLTP; Virtuoso is a multi-modal database that has published its SNB results (and offers state-of-the-art SNB results, based on our survey and experiments).

The implementations for these systems are included in the official SNB repository [7], except for DBMS T, whose implementation is from its own repository, and currently only implements read-only queries and the driver is limited to running one type of query each time rather than the mix of concurrent queries spawned by the official driver. Therefore, its throughput is estimated by a weighted sum according to each query's frequency (given by the official driver) and measured average latency. Neo4j and DBMS S's results are omitted as they are several orders of magnitude slower, echoing the findings in existing literature [47].

PostgreSQL and Virtuoso implement the requests with SQL plus stored procedures, and interact with the benchmark driver through JDBC. DBMS T calls RESTful APIs to serve queries by pre-installed stored procedures. LiveGraph's implementation uses a fixed query plan for each type of request. We build a server with the Apache Thrift RPC framework to communicate with the benchmark driver.

We enable transactional support on all systems. LiveGraph guarantees snapshot isolation. LMDB and DBMS T do not support concurrent write transactions, so they provide serializable isolation. We configure the other systems to use

[2]We record traces collected from MySQL runs and replay them for each system. Thinking times (i.e., the time to generate each request) are also recorded and reproduced.

[3]The feature/analytics branch from `https://github.com/v7fasttrack/virtuoso-opensource.git`, which is about 10× faster than v7.2.5 from master branch.

Table 3: Latency w. LinkBench TAO in memory (ms)

Storage	Optane SSD			NAND SSD		
System	LiveGraph	RocksDB	LMDB	LiveGraph	RocksDB	LMDB
mean	**0.0044**	0.0328	**0.0109**	**0.0051**	0.0309	**0.0098**
P99	**0.0053**	0.0553	**0.0162**	**0.0058**	0.0526	**0.0161**
P999	**1.0846**	4.8716	**2.0703**	**1.1224**	4.1968	**1.5769**

Table 4: Latency w. LinkBench DFLT in memory (ms)

Storage	Optane SSD			NAND SSD		
System	LiveGraph	RocksDB	LMDB	LiveGraph	RocksDB	LMDB
mean	**0.0449**	**0.1278**	1.6030	**0.0588**	**0.1459**	1.6743
P99	**0.2598**	**0.6423**	9.3293	**0.2838**	**0.8119**	9.8334
P999	**0.9800**	**3.5190**	12.275	**1.4642**	**4.8753**	13.365

Table 5: Latency w. LinkBench TAO out of core (ms)

Storage	Optane SSD			NAND SSD		
System	LiveGraph	RocksDB	LMDB	LiveGraph	RocksDB	LMDB
mean	**0.0166**	0.0420	**0.0364**	**0.0725**	**0.1065**	0.1322
P99	**0.1856**	0.1135	**0.3701**	**0.4830**	**0.2535**	0.8372
P999	**1.9561**	4.9366	**3.3600**	**2.5112**	**4.6701**	4.9119

Table 6: Latency w. LinkBench DFLT out of core (ms)

Storage	Optane SSD			NAND SSD		
System	LiveGraph	RocksDB	LMDB	LiveGraph	RocksDB	LMDB
mean	**0.0735**	**0.1312**	2.4099	**0.2184**	**0.2526**	2.2824
P99	**0.7923**	**0.6364**	13.799	**1.6543**	**2.2387**	12.557
P999	**3.0133**	**3.5250**	17.794	**5.0363**	**5.4436**	16.698

either snapshot isolation like LiveGraph, if available, or a default (usually weaker) isolation level. More precisely we use read committed for Neo4j, MyRocks, and PostgreSQL, and repeatable read for MySQL and Virtuoso.

7.2 Transactional Workloads

In-memory latency. First we evaluate transaction processing. Tables 3 and 4 give the average latency measured from LMDB, RocksDB, and LiveGraph in memory, with the LinkBench TAO and DFLT workloads respectively, using 24 client threads for request generation and Optane/NAND SSD for transactional durability. The average latencies for MySQL/MyRocks/Neo4j using Optane SSD as the backing storage are 0.187/0.214/0.236 ms for TAO, and 0.708/0.280/1.231 ms for DFLT. As these systems are clearly slower than LiveGraph, RocksDB, and LMDB, their detailed results and analysis are omitted.

The results demonstrate LiveGraph's significant performance advantage for both workloads. For the almost read-only TAO, LiveGraph improves the average latency by 2.47× from the runner-up (LMDB). TEL's major advantage here comes from storing edges by time order, facilitating fast-backward partial scans returning *latest edges*. This type of query is common not only in social network workloads [20] but also in other transactional graph applications (such as traffic maps and financial records). Not only does this accelerate memory accesses, it also improves both spatial and temporal locality, achieving more effective prefetching. In addition, compared to B+ trees and LSMTs, TEL has lower complexity for most operations, and avoids pointer chasing through the use of sequential data structures.

For DFLT, which contains 31% writes, LiveGraph remains a consistent winner across all categories. Due to its write-friendly sequential storage, its margin of advantage is even higher, beating the runner-ups in average, p99, and p999 latency by 2.67×, 3.06× and 4.99× respectively. Here LMDB suffers due to B+ tree's higher insert complexity and its single-threaded writes, while both LiveGraph and RocksDB benefit from their write-friendly log-structured design. However, as DFLT's majority of transactions are still reads, RocksDB's overall performance is severely dragged down by its inferior read performance in-memory.

LiveGraph has linear complexity (in terms of adjacency list size) when searching for a single edge, as opposed to the logarithmic cost of B+ trees or LSMTs. However, these operations (i.e., read/update/delete a specific edge of a high-degree vertex) are rare in the two LinkBench workloads. In particular, insertions can usually (in more than 99.9% of the cases, as found by our profiling) skip such searches, thanks to early rejection enabled by its embedded Bloom filters. Therefore, these operations do not impact tail latency much. LiveGraph's use of compaction also does not result in significantly higher tail latency than other systems. This is because its compaction only scans a small subset of blocks: the dirty vertex set maintained by each thread.

Out-of-core latency Tables 5 and 6 list the out-of-core (OOC) results, enabled by limiting memory (using Linux cgroup tools) to 4GB, which is about 16% of LiveGraph's, 9% of LMDB's and 28% of RocksDB's memory usage. This cap is the minimal memory for RocksDB to run with 128 client threads while delivering its peak throughput.

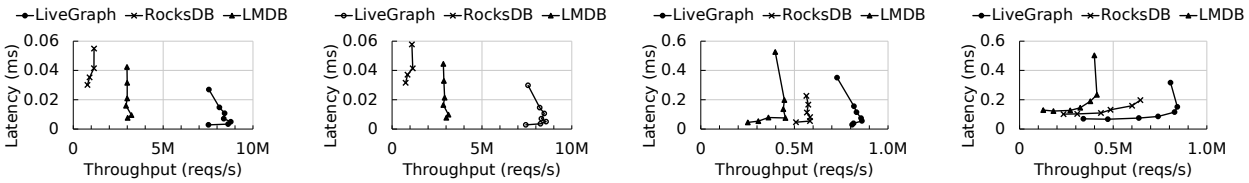
RocksDB is optimized for OOC writes, by dumping sorted blocks of data sequentially and performing compression for better I/O bandwidth usage. LiveGraph's design prioritizes reads instead. It performs sequential writes within an adjacency list but it does not ensure sequential storage of multiple dirty adjacency lists. It also issues smaller I/O requests by doing page write-back, with write size starting at 4KB, as opposed to the several MBs of RocksDB's LSMT. Fortunately, low-latency SSDs like our Optane device or byte-addressable NVM alleviate such problems.

Test results confirm the rationale above. With the more read-heavy TAO, LiveGraph wins across the board, cutting average latency by 2.19× from the runner-up LMDB on Optane. On NAND SSD, RocksDB beats LMDB, being more bandwidth-efficient with its compression. Still, its average latency is 1.46× higher than LiveGraph. For DFLT, LiveGraph outperforms RocksDB by 1.15× on NAND SSD, and by 1.79× on the faster Optane SSD.

Across the latency tests, by checking the top 2 finishers, it becomes apparent that among existing solutions, the B+-tree-based LMDB and LSMT-based RocksDB offer good performance under mostly-read and mixed workloads, respectively. However, when placed under unfavorable conditions, they switch places, with a 2× to 10× performance gap in between for each latency category. LiveGraph, in contrast, provides clearly better performance both in memory and out of core with Optane SSD or with the TAO workload.

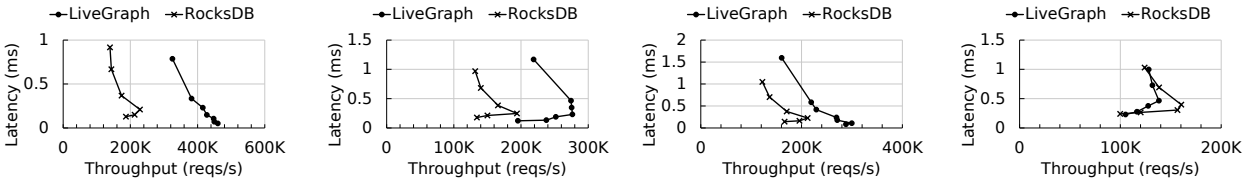
Scalability and throughput. We examine the multi-core scalability of LiveGraph by running LinkBench with an increasing number of clients. Figure 7a gives the result. We can see that LiveGraph's throughput scales smoothly with more cores being used until 24 clients when all the physical cores are occupied. For TAO, the scalability curve is very close to the ideal one. For DFLT, the write-ahead-logging bottleneck makes it hard for LiveGraph to achieve perfect scalability. We expect the emergence of NVM devices and related logging protocols [13, 14] would resolve this issue.

We then saturate the systems to measure throughput under the two workloads, removing the think time between re-



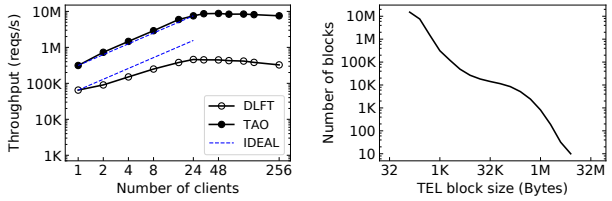
(a) Optane in memory **(b)** SSD in memory **(c)** Optane out of core **(d)** SSD out of core

Figure 5: TAO throughput and latency trends, with different number of clients and memory hardware



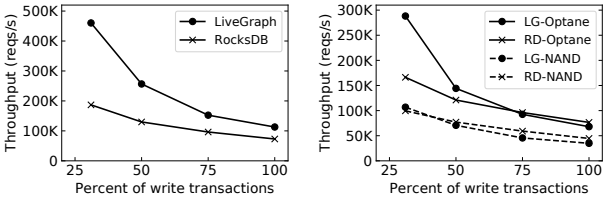
(a) Optane in memory **(b)** SSD in memory **(c)** Optane out of core **(d)** SSD out of core

Figure 6: DFLT throughput and latency trends, with different number of clients and memory hardware



(a) LiveGraph scalability **(b)** Block size distribution

Figure 7: LiveGraph scalability and block size distribution



(a) In memory **(b)** Out of core

Figure 8: LinkBench throughput with varying writing ratio

quests. Figures 5 and 6 show latency and throughput when increasing the number of clients from 24 (the number of cores in our server) until the peak throughput is reached, which required up to 256 clients.

For TAO (Figure 5), when in-memory on top of Optane (for durability), LMDB saturates at 32 clients (i.e., near to the number of physical cores) with a throughput of 3.24M requests/s, after which the contention on the single mutex intensifies. LiveGraph's throughput peaks at 8.77M requests/s with 48 clients, strained by context switch overhead afterwards. We get similar results with NAND.

Out of core, running TAO on top of Optane, RocksDB beats LMDB and reaches the peak point at 48 clients with a throughput of 584K requests/s, LiveGraph reaches 860K requests/s at 64 clients. With NAND, LiveGraph still improves the throughput by 1.31× from RocksDB.

For DFLT (Figure 6), RocksDB reaches 228K requests/s in memory and saturates at 48 clients, when compaction starts to pause writes frequently and write/read amplification becomes more significant. By contrast, LiveGraph is able to push beyond 460K at 24 clients, as TEL does not have such limits. NAND SSD results are similar, showing LiveGraph 4.83× and 1.43× faster than the runners-up, respectively. Out of core with Optane, LiveGraph peaks at 300K requests/s with 32 client threads and RocksDB saturates at 212K with 48 clients. With NAND, LiveGraph reaches 96.1% of RocksDB's peak throughput.

Memory consumption. Using our default compaction frequency every 65536 transactions, the DFLT workload, 24 client threads, and 12M transactions total, LiveGraph consumed 24.9GB in total, against 44.8 GB and 14.4 GB for LMDB and RocksDB, respectively. For LiveGraph, 706 MB space is recycled but not yet used at the end of the DFLT run. Of the allocated space, the aggregate over-provisioned space is about 4.6GB, leading to 81.2% final occupancy.

Figure 7b gives the TEL block count distribution at different sizes for this workload, which matches the power-law degree distribution among vertices [29], validating TEL's "buddy-system" design.

Effectiveness of compaction. When compaction is completely turned off, LiveGraph's footprint sees a 33.7% increase, requiring 33.3GB space. Compaction only scans a small dirty set so its time overhead is fairly small: varying the compaction frequency brings insignificant changes in performance (<5%).

Write-intensive workloads. To test more write-intensive workloads, we scale the write ratio, starting from LinkBench DFLT's 31% to 100%, and use 24 clients.

Figure 8a plots the in-memory throughput of LiveGraph and RocksDB (winners of DFLT according to Tables 3-6), with Optane SSD. It shows that LiveGraph's advantage weakens as the write ratio grows, but still significantly outperforms RocksDB (113K over 73K requests/s, at 1.54×) even with a *write-only* workload.

Figure 8b gives the out-of-core results on both SSDs. Here due to the write amplification and random write patterns on disk when out of core, LiveGraph is overtaken by RocksDB at write ratios of 75% (Optane SSD) and 50% (NAND SSD). Considering that LiveGraph's design favors read-intensive workloads, which are common in real-world graph database

Table 7: Throughput w. SNB in memory (reqs/s). (*) For DBMS T we extrapolated an optimistic estimate.

System	LiveGraph	Virtuoso	PostgreSQL	DBMS T
Complex-Only	9,106	292	3.79	59.3(*)
Overall	9,420	259	52.4	–

applications [20, 12], it fares reasonably well with an uncommon, extremely write-intensive workload. At 100%-write, LiveGraph reaches 88.8% of RocksDB throughput with Optane (78.3% with NAND).

We also profile disk output volume to measure write amplification and compare with RocksDB, which compresses SSTs to reduce disk I/O. LiveGraph writes 3.02× as much data as RocksDB: while the latter flushes only sorted updates, the former performs random writes of entire dirty pages and writes larger updates. However, the bandwidth of both our SSD devices is larger than the maximum bandwidth used by LiveGraph (522 MB/s), nearly removing the performance impact of write amplification. Given the common low write-ratio in commercial workloads, increasing SSD capacities, and falling storage price per GB, we argue that LiveGraph provides much improved latency and throughput at a small cost for average use cases.

Long-running transactions and checkpoints. Our design supports low overhead read-only transactions with snapshot isolation. In the following test, we let a checkpointer continuously scan the whole graph and dump the latest snapshot to the NAND SSD, while simultaneously running LinkBench DFLT (in-memory with Optane).

Dumping a snapshot with a single thread to shared-memory takes 16.0s, without concurrent LinkBench queries. With concurrent queries with 24 clients (one per core), it takes 20.6s (22.5% slower). Meanwhile, LinkBench throughput only slows down by 6.5% when running concurrent, continuous checkpointing. We get similar results on NAND SSD, with 10.9% and 3.6% slowdown in checkpointing and LinkBench processing, respectively. Users may choose to configure LiveGraph to use more threads for faster checkpointing, at the cost of higher interference. E.g., with 24 checkpointing threads, LiveGraph completes a 12GB checkpoint in 6.5s, fully utilizing the NAND SSD write bandwidth (2GB/s), while still delivering 86.4% of the full LinkBench throughput (the one obtained without checkpointing).

7.3 Real-Time Analytics Workloads

Real-time analytics. Table 7 gives the throughput measured from LiveGraph, Virtuoso, PostgreSQL, and DBMS T in memory, with only complex reads (referred to as Complex-Only) and all three categories of requests (referred to as Overall), using 48 client and 48 worker threads and Optane SSD for persistence. DBMS T has its own SNB driver that only runs requests sequentially. We extrapolate throughput by running it with one worker and then optimistically assume that the system scales linearly to 48 workers. The Overall workload uses SNB's official mix: 7.26% complex queries, 63.82% short queries, and 28.91% updates. RocksDB and LMDB are skipped as (1) these two K-V stores do not have official SNB implementations, and (2) our earlier micro-benchmark and LinkBench results of their basic graph database operations (upon which more complex queries are built) significantly lags behind LiveGraph. We only report DBMS T's Complex-Only results as its implementation for SNB update requests is still absent. DBMS T is single-writer so it is unlikely that it would outperform the other baselines on update requests. LiveGraph outperforms the runner-up, Virtuoso, by 31.19× and 36.43×, producing gains far larger than those observed in microbenchmarks or LinkBench. Meanwhile, Virtuoso beats DBMS T by 4.92× and beats PostgreSQL by 77.05× and 4.94×.

Table 8: Throughput w. SNB out of core (reqs/s)

System	LiveGraph	Virtuoso
Complex-Only	31.0	2.91
Overall	350	14.7

Table 9: Average latency of queries in SNB (ms)

System	LiveGraph	Virtuoso	PostgreSQL	DBMS T
Complex read 1	7.00	23,101	371	717
Complex read 13	0.53	2.47	10,419	19.4
Short read 2	0.22	3.11	3.31	3.22
Updates	0.37	0.93	2.19	–

We found fast edge scans are even more critical with complex analytics workloads. This also explains why PostgreSQL underperforms, since it does not support clustered indexes [8]. Second, MVCC is crucial for fast analytics when these complex queries are mixed with write transactions. Compared to Complex-Only, Overall has more short queries/updates, so LiveGraph and PostgreSQL both produce higher throughput (the reason for LiveGraph's small Complex-Only vs. Overall difference is that Overall is more write-intensive and throughput is limited by persisting the WAL). Virtuoso, on the other hand, performs worse by spending over 60% of its CPU time on locks. Of course, MVCC comes with its space overhead: for this workload, LiveGraph consumes about 30GB, PostgreSQL 19GB, and Virtuoso only 8.3GB. Compared to PostgreSQL, which also performs MVCC, LiveGraph's space overhead comes from its longer timestamps, plus its overprovisioning in adjacency lists (a key design choice that helps in query performance).

One may argue that Virtuoso's smaller memory footprint helps with out-of-core execution. Table 8 (with 3GB DRAM cap, persisting on Optane) shows a heavy performance hit for both LiveGraph and Virtuoso when going out-of-core, and the gap between them does shrink. However, LiveGraph is still an order of magnitude better, and for the Overall mix, beats Virtuoso's *in-memory* performance by 1.35×.

Again, though results are not listed, Neo4j and DBMS S lose to all three systems by another 2-3 orders of magnitude. This is in part due to their use of Java, which is not ideal for data-intensive workloads, and in part because of their choice of data structures: Neo4j uses linked list and DBMS S builds on copy-on-write B+ trees similar to LMDB. Our examination reveals that multi-hop graph queries dramatically stress Java's garbage collection.

Query case study. Due to space limits, we present brief case studies on selected queries in the SNB mixes, with their average latencies listed in Table 9.

"Complex read 1" accesses many vertices (3-hop neighbors). Here the benefit of MVCC stands out, with Virtuoso suffering lower throughput caused by locks.

"Complex read 13" performs pairwise shortest path (PSP) computation. Virtuoso's implementation uses its custom SQL extension, with well-optimized PSP primitive, but still loses to LiveGraph by 4.68×. PostgreSQL is limited to only recursive SQL for PSP and loses to other approaches by orders of magnitude. This demonstrates the benefit of having a graph specific query language or extensions.

Table 10: ETL and execution times (ms) for analytics

System	LiveGraph	Gemini
ETL	-	1520
PageRank	266	156
ConnComp	254	62.6

"Short read 2" is a 1-hop query with many short neighborhood operations. Here latency relates to seek performance, demonstrated to be LiveGraph's forte in the microbenchmark results.

"Updates" shows average latencies of all requests containing updates. Though LiveGraph is designed mainly toward fast reads, it performs well in writes too, leading Virtuoso by 2.51× and PostgreSQL by 5.92×.

7.4 Iterative Long-Running Analytics

Finally, we evaluate LiveGraph's capability of performing iterative long-running analytics directly within its primary graph store. Table 10 presents its performance in running two common graph algorithms, PageRank and Connected Components (ConnComp), on a subgraph (with 3.88M edges) of the SNB SF10 dataset, involving all the Person nodes and their relationships.

LiveGraph performs iterative analyses directly on the latest snapshot, finishing both tasks under 300 milliseconds, using 24 threads. For comparison, we give the results of Gemini [65], a state-of-the-art graph engine dedicated to such static graph analytics. Without the compact yet immutable CSR-based graph storage used by Gemini, LiveGraph manages to reach 58.6% and 24.6% of Gemini's performance for PageRank and ConnComp, respectively. In addition, to enjoy Gemini's superior performance, one has to export to its data format, then load the graph into its memory. We measured this ETL overhead (converting from TEL to CSR) for this specific graph to be 1520ms, greatly exceeding the PageRank/ConnComp execution time, not to mention the difference between LiveGraph and Gemini.

8. RELATED WORK

Transactional Systems. Graph databases can be grouped into two categories. Native graph databases [3, 9, 10, 11] are designed from scratch for storing/querying graphs. Non-native graph databases store graphs using general-purpose stores, sometimes using a layer on top of an existing RDBMS or key-value store [20, 56, 6, 33, 26].

All these systems support transactional graph workloads and some of them support analytical graph workloads [18]. As discussed in the motivation, these systems adopt tree-based [9, 10, 20, 56, 6, 33, 36, 35, 26] or pointer-based [3] data structures and, therefore, suffer a major limitation of requiring pointer chasing for adjacency list scans. Even state-of-the-art systems designed for low latency do not optimize for purely sequential scans of adjacency lists [21].

There are efforts on improving the analytical performance of existing transactional systems [36, 30, 35, 64]. These systems generally compare with and offer comparable performance to GraphLab, while reducing loading time. However, GraphLab is not a state-of-the-art baseline, as shown by existing literature [54, 46, 65].

Analytics on Dynamic Graphs. Several graph engines support graph analytics over an evolving graph, to study graph evolution or to update computation results incrementally. Kineograph [24] supports incremental graph analysis by periodically applying updates and generating snapshots. Chronos [34] and ImmortalGraph [44] are designed to analyze graph evolution, processing a sequence of predefined, read-only graph snapshots. LLAMA [43] applies incoming updates in batches and creates copy-on-write delta snapshots dynamically for temporal graph analysis. Grace [49] supports transactional updates but uses an expensive copy-on-write technique: every time an adjacency list is modified, the entire list is copied to the tail of the edge log. This makes scans purely sequential but it also makes updates very expensive, especially for high-degree vertices. GraphOne [38] serializes edge updates by appending them onto a single edge log. It does not support transactions or durability.

These systems focus primarily on graph analysis. LiveGraph supports real-time transactional workloads with better performance than existing graph databases, while supporting whole-graph analytics on the same primary graph store. Many incremental analytics techniques above can readily be incorporated, leveraging LiveGraph's multi-versioning.

Graph engines. Graph engines perform analytical processing, such as graph processing [31, 32, 65, 54, 46, 57, 62] or graph mining [60, 58, 22]. Their design assumes an immutable graph topology, hence widely adopting read-optimized CSR/CSC representations. As discussed earlier, this delivers superior analytics performance, but does not handle updates/insertions. Hence existing graph engines have been limited to processing static, stale snapshots dumped from the data source. In contrast, LiveGraph supports analytics on dynamic graphs without costly ETL.

9. CONCLUSION AND FUTURE WORK

Our work shows that it is possible to design graph data management systems that are fast at both transactional and analytical workloads. The key is using data structures that are tailored to the operations of graph workloads, as well as associated algorithms for transactional support. Our evaluation confirms the strength of LiveGraph as a potential all-around choice across multiple graph workloads.

The next steps of LiveGraph include adapting relational concurrency control and storage techniques optimized for modern hardware [45, 14, 42, 40]. LiveGraph's design is amenable to scaling out, leveraging techniques in distributed graph query processing [53, 63, 26] and distributed transaction management [48, 25, 19, 26]. In addition, LiveGraph can be extended in another direction: the multi-versioning nature of TELs makes it natural to support temporal graph processing [34, 44], with modifications to the compaction algorithm to efficiently store and index older graph versions.

10. ACKNOWLEDGMENTS

We sincerely thank all the reviewers for their insightful comments and suggestions. We appreciate suggestions from colleagues Saravanan Thirumuruganathan, Nan Tang, and Mourad Ouzzani during our revision. This work is supported in part by NSFC Grant 61525202 for Distinguished Young Scholars.

11. REFERENCES

[1] https://www.gartner.com/en/newsroom/press-releases/2019-02-18-gartner-identifies-top-10-data-and-analytics-technolo.

[2] http://rocksdb.org/.

[3] https://neo4j.com/.

[4] https://symas.com/lmdb/.

[5] https://www.postgresql.org/.

[6] https://virtuoso.openlinksw.com/.

[7] https://github.com/ldbc/ldbc_snb_implementations.

[8] https://en.wikipedia.org/wiki/Database_index#Clustered.

[9] http://titan.thinkaurelius.com/.

[10] http://orientdb.com/.

[11] https://www.arangodb.com/.

[12] T. G. Armstrong, V. Ponnekanti, D. Borthakur, and M. Callaghan. LinkBench: a database benchmark based on the Facebook social graph. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pages 1185–1196. ACM, 2013.

[13] J. Arulraj, A. Pavlo, and S. R. Dulloor. Let's talk about storage & recovery methods for non-volatile memory database systems. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 707–722. ACM, 2015.

[14] J. Arulraj, M. Perron, and A. Pavlo. Write-behind logging. *PVLDB*, 10(4):337–348, 2016.

[15] P. Bailis, A. Davidson, A. Fekete, A. Ghodsi, J. M. Hellerstein, and I. Stoica. Highly available transactions: Virtues and limitations. *PVLDB*, 7(3):181–192, 2013.

[16] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil. A critique of ANSI SQL isolation levels. In *ACM SIGMOD Record*, volume 24, pages 1–10. ACM, 1995.

[17] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[18] M. Besta, E. Peter, R. Gerstenberger, M. Fischer, M. Podstawski, C. Barthels, G. Alonso, and T. Hoefler. Demystifying graph databases: Analysis and taxonomy of data organization, system designs, and graph queries. *arXiv preprint arXiv:1910.09017*, 2019.

[19] C. Binnig, S. Hildenbrand, F. Färber, D. Kossmann, J. Lee, and N. May. Distributed snapshot isolation: global transactions pay globally, local transactions pay locally. *The VLDB Journal*, 23(6):987–1011, 2014.

[20] N. Bronson, Z. Amsden, G. Cabrera, P. Chakka, P. Dimov, H. Ding, J. Ferris, A. Giardullo, S. Kulkarni, H. C. Li, et al. TAO: Facebook's distributed data store for the social graph. In *Proceedings of the 2013 USENIX Annual Technical Conference*, pages 49–60, 2013.

[21] A. Carter, A. Rodriguez, Y. Yang, and S. Meyer. Nanosecond indexing of graph data with hash maps and VLists. In *Proceedings of the 2019 ACM SIGMOD International Conference on Management of Data*, pages 623–635. ACM, 2019.

[22] H. Chen, M. Liu, Y. Zhao, X. Yan, D. Yan, and J. Cheng. G-miner: an efficient task-oriented graph mining system. In *Proceedings of the 2018 EuroSys Conference*, page 32. ACM, 2018.

[23] R. Chen, J. Shi, Y. Chen, and H. Chen. PowerLyra: Differentiated graph computation and partitioning on skewed graphs. In *Proceedings of the 2015 EuroSys Conference*, page 1. ACM, 2015.

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

[25] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, et al. Spanner: Google's globally distributed database. *ACM Transactions on Computer Systems (TOCS)*, 31(3):8, 2013.

[26] A. Dubey, G. D. Hill, R. Escriva, and E. G. Sirer. Weaver: a high-performance, transactional graph database based on refinable timestamps. *PVLDB*, 9(11):852–863, 2016.

[27] O. Erling, A. Averbuch, J. Larriba-Pey, H. Chafi, A. Gubichev, A. Prat, M.-D. Pham, and P. Boncz. The LDBC social network benchmark: Interactive workload. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 619–630. ACM, 2015.

[28] J. M. Faleiro and D. J. Abadi. Rethinking serializable multiversion concurrency control. *PVLDB*, 8(11):1190–1201, 2015.

[29] M. Faloutsos, P. Faloutsos, and C. Faloutsos. On power-law relationships of the internet topology. In *ACM SIGCOMM Computer Communication Review*, volume 29, pages 251–262. ACM, 1999.

[30] J. Fan, A. G. S. Raj, and J. Patel. The case against specialized graph analytics engines. In *Proceedings of the 2015 Biennial Conference on Innovative Data Systems Research (CIDR)*, 2015.

[31] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. PowerGraph: Distributed graph-parallel computation on natural graphs. In *Proceedings of the 2012 USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2012.

[32] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica. GraphX: Graph processing in a distributed dataflow framework. In *Proceedings of the 2014 USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 599–613, 2014.

[33] S. Gurajada, S. Seufert, I. Miliaraki, and M. Theobald. TriAD: a distributed shared-nothing RDF engine based on asynchronous message passing. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 289–300. ACM, 2014.

[34] 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 *Proceedings of the 2014 EuroSys Conference*. ACM, 2014.

[35] A. Jindal, S. Madden, M. Castellanos, and M. Hsu. Graph analytics using Vertica relational database. In *Proceedings of the 2015 IEEE International*

Conference on Big Data, pages 1191–1200. IEEE, 2015.

[36] H. Kimura, A. Simitsis, and K. Wilkinson. Janus: Transactional processing of navigational and analytical graph queries on many-core servers. In *Proceedings of the 2017 Biennial Conference on Innovative Data Systems Research (CIDR)*, 2017.

[37] K. C. Knowlton. A fast storage allocator. *Communications of the ACM*, 8(10):623–624, 1965.

[38] P. Kumar and H. H. Huang. Graphone: A data store for real-time analytics on evolving graphs. In *Proceedings of the 2019 USENIX Conference on File and Storage Technologies (FAST)*, pages 249–263, 2019.

[39] P.-Å. Larson, S. Blanas, C. Diaconu, C. Freedman, J. M. Patel, and M. Zwilling. High-performance concurrency control mechanisms for main-memory databases. *PVLDB*, 5(4):298–309, 2011.

[40] V. Leis, M. Haubenschild, A. Kemper, and T. Neumann. Leanstore: In-memory data management beyond main memory. In *Proceedings of the 2018 IEEE International Conference on Data Engineering (ICDE)*, pages 185–196. IEEE, 2018.

[41] J. Leskovec, D. Chakrabarti, J. Kleinberg, C. Faloutsos, and Z. Ghahramani. Kronecker graphs: An approach to modeling networks. *Journal of Machine Learning Research*, 11(Feb):985–1042, 2010.

[42] H. Lim, M. Kaminsky, and D. G. Andersen. Cicada: Dependably fast multi-core in-memory transactions. In *Proceedings of the 2017 ACM SIGMOD International Conference on Management of Data*, pages 21–35. ACM, 2017.

[43] P. Macko, V. J. Marathe, D. W. Margo, and M. I. Seltzer. LLAMA: Efficient graph analytics using large multiversioned arrays. In *Proceedings of the 2015 IEEE International Conference on Data Engineering (ICDE)*, pages 363–374. IEEE, 2015.

[44] Y. Miao, W. Han, K. Li, M. Wu, F. Yang, L. Zhou, V. Prabhakaran, E. Chen, and W. Chen. ImmortalGraph: A system for storage and analysis of temporal graphs. *ACM Transactions on Storage (TOS)*, 11(3):14, 2015.

[45] T. Neumann, T. Mühlbauer, and A. Kemper. Fast serializable multi-version concurrency control for main-memory database systems. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 677–689. ACM, 2015.

[46] D. Nguyen, A. Lenharth, and K. Pingali. A lightweight infrastructure for graph analytics. In *Proceedings of the 2013 ACM Symposium on Operating Systems Principles (SOSP)*, pages 456–471, 2013.

[47] A. Pacaci, A. Zhou, J. Lin, and M. T. Özsu. Do we need specialized graph databases?: Benchmarking real-time social networking applications. In *Proceedings of the 2017 International Workshop on Graph Data-management Experiences & Systems*, page 12. ACM, 2017.

[48] D. Peng and F. Dabek. Large-scale incremental processing using distributed transactions and notifications. In *Proceedings of the 2014 USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 251–264, 2010.

[49] V. Prabhakaran, M. Wu, X. Weng, F. McSherry, L. Zhou, and M. Haradasan. Managing large graphs on multi-cores with graph awareness. In *Proceedings of the 2012 USENIX Annual Technical Conference*, pages 41–52, 2012.

[50] F. Putze, P. Sanders, and J. Singler. Cache-, hash-and space-efficient Bloom filters. In *Proceedings of the International Workshop on Experimental and Efficient Algorithms*, pages 108–121. Springer, 2007.

[51] I. Robinson, J. Webber, and E. Eifrem. *Graph databases: New opportunities for connected data*. O'Reilly Media, 2015.

[52] Y. Saad. *Iterative methods for sparse linear systems*, volume 82. SIAM, 2003.

[53] J. Shi, Y. Yao, R. Chen, H. Chen, and F. Li. Fast and concurrent RDF queries with RDMA-based distributed graph exploration. In *Proceedings of the 2016 USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 317–332, 2016.

[54] J. Shun and G. E. Blelloch. Ligra: a lightweight graph processing framework for shared memory. In *ACM SIGPLAN Notices*, volume 48, pages 135–146. ACM, 2013.

[55] K. Soramäki, M. L. Bech, J. Arnold, R. J. Glass, and W. E. Beyeler. The topology of interbank payment flows. *Physica A: Statistical Mechanics and its Applications*, 379(1):317–333, 2007.

[56] W. Sun, A. Fokoue, K. Srinivas, A. Kementsietsidis, G. Hu, and G. Xie. SQLGraph: an efficient relational-based property graph store. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 1887–1901. ACM, 2015.

[57] N. Sundaram, N. Satish, M. M. A. Patwary, S. R. Dulloor, M. J. Anderson, S. G. Vadlamudi, D. Das, and P. Dubey. Graphmat: High performance graph analytics made productive. *PVLDB*, 8(11):1214–1225, 2015.

[58] C. H. Teixeira, A. J. Fonseca, M. Serafini, G. Siganos, M. J. Zaki, and A. Aboulnaga. Arabesque: a system for distributed graph mining. In *Proceedings of the 2015 ACM Symposium on Operating Systems Principles (SOSP)*, pages 425–440. ACM, 2015.

[59] S. Tu, W. Zheng, E. Kohler, B. Liskov, and S. Madden. Speedy transactions in multicore in-memory databases. In *Proceedings of the 2013 ACM Symposium on Operating Systems Principles (SOSP)*, pages 18–32. ACM, 2013.

[60] K. Wang, G. Xu, Z. Su, and Y. D. Liu. Graphq: graph query processing with abstraction refinement: scalable and programmable analytics over very large graphs on a single pc. In *Proceedings of the 2015 USENIX Annual Technical Conference*, pages 387–401. USENIX Association, 2015.

[61] Y. Wu, J. Arulraj, J. Lin, R. Xian, and A. Pavlo. An empirical evaluation of in-memory multi-version concurrency control. *PVLDB*, 10(7):781–792, 2017.

[62] W. Xie, G. Wang, D. Bindel, A. Demers, and J. Gehrke. Fast iterative graph computation with block updates. *PVLDB*, 6(14):2014–2025, 2013.

[63] Y. Zhang, R. Chen, and H. Chen. Sub-millisecond

stateful stream querying over fast-evolving linked data. In *Proceedings of the 2017 ACM Symposium on Operating Systems Principles (SOSP)*, pages 614–630. ACM, 2017.

[64] K. Zhao and J. X. Yu. All-in-One: Graph processing in RDBMSs revisited. In *Proceedings of the 2017 ACM SIGMOD International Conference on Management of Data*, pages 1165–1180. ACM, 2017.

[65] X. Zhu, W. Chen, W. Zheng, and X. Ma. Gemini: A computation-centric distributed graph processing system. In *Proceedings of the 2016 USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 301–316, 2016.

[66] X. Zhu, G. Feng, M. Serafini, X. Ma, J. Yu, L. Xie, A. Aboulnaga, and W. Chen. LiveGraph: A transactional graph storage system with purely sequential adjacency list scans. *CoRR*, abs/1910.05773, 2020.