RapidStore: An Efficient Dynamic Graph Storage System for Concurrent Queries

Chiyu Hao
Shanghai Jiao Tong University
hcahoi11@sjtu.edu.cn

Jixian Su
Shanghai Jiao Tong University
sjx13623816973@sjtu.edu.cn

Shixuan Sun
Shanghai Jiao Tong University
sunshixuan@sjtu.edu.cn

Hao Zhang
Huawei
zhanghao687@huawei.com

Sen Gao
Shanghai Jiao Tong University
Gawssin@gmail.com

Jianwen Zhao
Huawei
zhaojianwen3@huawei.com

Chenyi Zhang
Huawei
zhangchenyi2@huawei.com

Jieru Zhao
Shanghai Jiao Tong University
zhao-jieru@sjtu.edu.cn

Chen Chen
Shanghai Jiao Tong University
chen-chen@sjtu.edu.cn

Minyi Guo
Shanghai Jiao Tong University
guo-my@cs.sjtu.edu.cn

ABSTRACT

Dynamic graph storage systems are essential for real-time applications such as social networks and recommendation, where the graph continuously evolves. However, they face significant challenges in efficiently handling concurrent read and write operations. We find that existing methods suffer from write queries interfering with read efficiency, substantial time and space overhead due to per-edge versioning, and an inability to balance performance, such as slow searches. To address these issues, we propose RapidStore, a holistic approach for efficient in-memory dynamic graph storage designed for read-intensive workloads. Our key idea is to exploit the characteristics of graph queries through a decoupled system design that separates the management of read and write queries and decouples version data from graph data. Besides, we design an efficient dynamic graph store to cooperate with the graph concurrency control mechanism. Experiments show that RapidStore enables fast and scalable concurrent graph queries, effectively balancing the performance of inserts, searches, and scans, and significantly improving efficiency in dynamic graph storage systems.

1 INTRODUCTION

As graph data frequently evolves, in-memory dynamic graph storage systems play a crucial role in applications such as social networks [16, 34, 39], transaction networks [30], and transportation networks [31]. These systems aim to efficiently support concurrent read and write queries on graphs, enabling real-time online graph data processing [33]. Given the importance of dynamic graph storage, several systems have been proposed recently such as Sortledton [14], Teseo [7], and LiveGraph [49] to support transactional graph queries that ensure serializability of concurrent operations. Specifically, since the *Compressed Sparse Row* (CSR) graph format cannot efficiently handle updates, these works focus on optimizing graph storage formats to facilitate both read and write operations.

To ensure isolation among concurrent queries, these methods adopt concurrency control mechanisms designed for relational database management systems (RDBMS) in the graph context [21, 28]. Specifically, they use *Multi-Version Concurrency Control* (MVCC) to maximize parallelism. As edges are the basic units in graphs, these systems employ a *per-edge versioning* strategy, maintaining multiple versions for each edge so that different queries can simultaneously access the same edge. Furthermore, since graph queries typically follow a "vertex-neighbor" access pattern—first locating a vertex u and then operating on its neighbor set $N(u)$—these systems acquire locks on vertices rather than edges to coordinate access.

Despite these advancements, we observe significant performance issues due to their concurrency control methods. First, both read and write operations must acquire a lock on a vertex before accessing it, leading to lock contention and performance degradation for read queries. Second, graph queries are generally read-intensive and heavily rely on scan operations that traverse neighbor sets, such as in PageRank and breadth-first search. Consequently, the overhead of version checks on each edge access is substantial, and maintaining versions for each edge results in significant memory consumption. Third, while existing systems focus on optimizing scan performance for traversal-based queries, they overlook search

efficiency, failing to support complex queries like pattern matching. These challenges highlight the need for improved concurrency control, version management, and graph data structure design.

To address these challenges, we propose **RapidStore**. This holistic solution, which is designed for read-intensive applications, combines a novel graph concurrency control mechanism with an optimized graph data store. Specifically, we propose a *subgraph-centric graph concurrency control* mechanism, which maintains versions at the subgraph level. This coarse-grained version management separates version information from the graph data, eliminating the cost of version checks. We coordinate write queries using the MV2PL protocol, while read queries operate on graph snapshots without any locks, mitigating the impact of writes on read efficiency. To enhance the concurrency control mechanism and enable fast search operations, we design the *Compressed Adaptive Radix Tree* (C-ART) to store graphs. Extensive experiments demonstrate that RapidStore achieves up to 3.46x speedup over the latest systems on graph analytic queries while saving 56.34% of memory. RapidStore also exhibits high concurrency: with 4 writers and 28 readers running concurrently, its read query completion time increases by at most 13.36%, compared to up to 41.04% for other systems. In summary, this paper makes the following contributions:

- Introduction of a novel graph concurrency control mechanism that minimizes read-write interference.
- Development of an optimized graph data store that balances performance across various operations.
- Implementation of a decoupled system design, enhancing overall system responsiveness.

Compared to existing approaches, RapidStore aims to achieve a high concurrent performance while achieving high read performance and comparable write performance, thus fulfilling the unmet needs of current dynamic graph storage systems.

2 PRELIMINARY

We focus on directed graphs, denoted as $G = (V, E)$, where V represents the set of vertices and $E \subseteq V \times V$ denotes the set of directed edges. An edge $e(u, v) \in E$ indicates a directed connection from u to v. For any vertex $u \in V$, we define $N(u)$ as the neighborhood of u, which includes all vertices v such that $e(u, v) \in E$. The out-degree of vertex u, denoted as $d(u)$, is given by $|N(u)|$. In contrast, an undirected graph can be represented by storing edges in both directions, i.e., $e(u, v)$ and $e(v, u)$. Each vertex is assigned a unique integer ID within the range $[0, |V|)$. Previous studies [1, 15, 17, 43, 45] have shown that using integer vertex IDs within this range offers significant computational and storage advantages due to their compact representation. Therefore, existing works [7, 9, 14, 25, 49] either require vertex IDs to fall within $[0, |V|)$ or utilize dictionary encoding techniques to map external vertex IDs to this range. In this paper, we assume that each vertex ID $u \in V$ is an integer in $[0, |V|)$. An ID is an 4-bytes integer in the implementation.

A *dynamic graph* $G = (G_0, \Delta G)$ represents the evolution of a graph over time, where G_0 is the initial graph and ΔG is a sequence of updates. Each update $\Delta G_i = (\oplus, e)$ modifies the graph by either inserting or deleting $e(u, v)$, with $\oplus = +/-$ indicating the operation. By applying updates from ΔG_0 to ΔG_i, we obtain the graph G_i. Dynamic graph storage systems are designed to efficiently support

Figure 1: An example of ART.

both read and write operations: *Search(u, v)*, which finds vertex v in $N(u)$, *Scan(u)*, which traverses $N(u)$, and *Insert(u, v)* and *Delete(u, v)*, which add or remove a neighbor from $N(u)$ (i.e., adding or removing an edge $e(u, v)$). Since graph applications are typically read-intensive, this paper focuses on scenarios with small updates and heavy read operations, consistent with prior works [7, 14, 49]. Nevertheless, RapidStore also efficiently supports batch updates involving tens of thousands of edges.

Adaptive Radix Tree. The *adaptive radix tree* (ART) [22] is a high-performance data structure optimized for fast and memory-efficient key storage and retrieval. It builds on the radix tree by indexing keys through byte sequences. In a typical radix tree, each node may hold up to 256 child pointers (one for each byte value), which enables rapid lookups but can waste memory when many pointers are unused. ART overcomes this inefficiency by dynamically adapting the size of its nodes to the number of children. It defines four node types: *N4*, *N16*, *N48*, and *N256*, supporting up to 4, 16, 48, and 256 child pointers, respectively. Nodes automatically grow or shrink as keys are inserted or deleted, optimizing memory usage. Furthermore, ART employs *path compression* to reduce tree depth by compressing common prefixes shared by multiple keys into a single edge or node. *Lazy expansion* further collapses nodes by removing the path to single leaf. These two techniques reduce the number of nodes, speeding up searches. For n elements in ART, each with a byte length of w, the time complexity for *Search*, *Insert*, and *Delete* operations is $O(w)$, while *Scan* takes $O(n)$. The space complexity is $O(n)$.

Example 2.1. Figure 1 shows an ART storing a set of vertices represented by 4-byte integers (8 hexadecimal digits). Each leaf node holds a vertex. *N4* nodes store up to four key-pointer pairs, while *N256* nodes store pointers only, with the array index implicitly indicating the key. *Depth D* tracks which byte in the sequence is used as the key for the node, while *Prefix* records the common prefix from the root to that node. The byte sequence M of a vertex is indexed from 0. To search vertex $M = 0x010200FF$, we first examine $M[0] = 01$ in *Node1* since *Node1*'s depth is 0, and follow
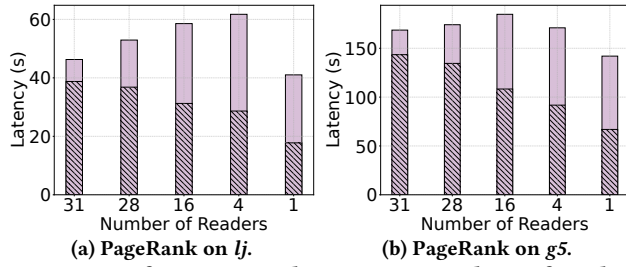
(a) PageRank on *lj*. (b) PageRank on *g5*.

Figure 2: Performance under varying numbers of readers and writers (total threads fixed at 32). The shadowed bars represent the latency of readers in the absence of writers.

the pointer to *Node2*. Due to path compression, *Node2*'s depth is 2, so we skip to $M[2] = 00$ and proceed to *Node3*, which has a depth of 3. We match $M[3] = FF$ in *Node3* and retrieve the target vertex. *Insert* and *Delete* operations follow similar steps to add or remove elements. *Scan* performs a depth-first search across all values.

Graph Concurrency Control. We use Sortledton [14] as a representative to describe existing graph concurrency control mechanisms. Sortledton employs MVCC to manage concurrent operations, maintaining multiple version logs for each edge in a linked list, called the version chain, ordered from newest to oldest. Each log entry records the operation type and its timestamp. To ensure data consistency during concurrent reads and writes, Sortledton adapt MV2PL: Writers acquire locks on all vertices they will modify at the beginning of the transaction and release them after completing the updates, while readers lock the accessed vertices and release them immediately after use. This approach leverages the characteristic that the set of vertices affected by a write query is known at the start of the transaction.

3 MOTIVATION

Although several dynamic graph storage methods have been proposed recently, we observe that they have severe performance issues, leading to unique challenges of efficiently processing concurrent read and write queries. In the following, we demonstrate these issues using Sortledton [14], the state-of-the-art method. For brevity, we present the experiment results on *livejournal* (*lj*) and *graph500* (*g5*). The server has a CPU equipped with 32 physical cores. For detailed experimental settings, please refer to Section 7.

Issue 1: Interference between Concurrent Reads and Writes. In existing approaches, both read and write operations must acquire a lock on a vertex before accessing its neighbor set or properties to ensure serializability among queries. This locking strategy leads to contention when multiple queries attempt to access the same vertex concurrently, causing performance degradation for both reads and writes. This issue is particularly severe for high-degree vertices, accessed more frequently due to their numerous connections.

To illustrate this issue, Figure 2 shows the latency for 10 iterations of PageRank. Each reader independently executes a PageRank query. When no writers are active, the average latency of readers decreases as the number of readers reduces. This reduction occurs because fewer readers alleviate memory bandwidth pressure during the frequent graph scans in PageRank. Conversely, when writers are performing updates, read performance significantly deteriorates due to lock contention between read and write operations. Figure 3
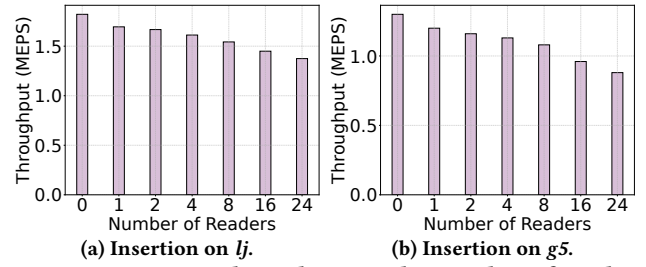

(a) Insertion on *lj*. (b) Insertion on *g5*.

Figure 3: Insertion throughput as the number of readers varies, with the number of writers fixed at 8.

examines insertion throughput as the number of readers increases while keeping the number of writers fixed at 8. The results demonstrate a drop in insertion throughput as more readers are added. This decline stems from increased lock contention, which impairs the writers' ability to execute updates efficiently. These findings highlight a significant challenge:

Challenge 1: How can we mitigate the interference between concurrent read and write queries to enhance performance and scalability in graph storage systems?

Issue 2: Time and Space Overheads Due to Per-Edge Versioning. To support concurrent read and write queries, existing methods maintain versions for each edge. This strategy requires queries to perform a version check on each edge access to ensure they retrieve the correct data version. Additionally, storing version chains for each edge increases memory requirements.

Table 1: Performance of *Search, Scan* and PageRank (PR) without and with versioning.

Dataset	Versioned?	Search (TEPS)	Scan (TEPS)	PR (s)
lj	No	**2270.85**	**40371.23**	**16.23**
	Yes	1898.17	19033.40	31.16
g5	No	**672.89**	**89055.10**	**44.15**
	Yes	635.12	38885.92	94.26

Table 1 compares the performance of Sortedlton with and without versioning. Scan iterates over the neighbor set of a vertex, while Search identifies a target edge. Both are measured in thousand edges processed per second (TEPS). Versioning reduces *Scan* throughput by approximately 53% for *lj* and 56% for *g5*, reflecting the overhead of version checks during frequent edge scans. For *Search*, the throughput drops by about 16% for *lj* and 6% for *g5*, showing a smaller but noticeable impact. PageRank experiences a significant performance degradation, with latency nearly doubling for both datasets. This issue highlights the second challenge.

Challenge 2: How can we reduce the time and space inefficiencies caused by per-edge versioning to enhance the speed and scalability of graph analytics?

Issue 3: Inefficient Support for Fast Search Operations. Traversal-based algorithms like PageRank and breadth-first search rely heavily on fast scan operations. Consequently, existing methods prioritize optimizing scan performance while often neglecting the efficiency of search operations, which are critical for pattern matching tasks such as triangle counting (TC). For instance, when intersecting two sets of vastly different sizes, iterating over the smaller set and performing searches in the larger set is more efficient than using merge-based set intersections.
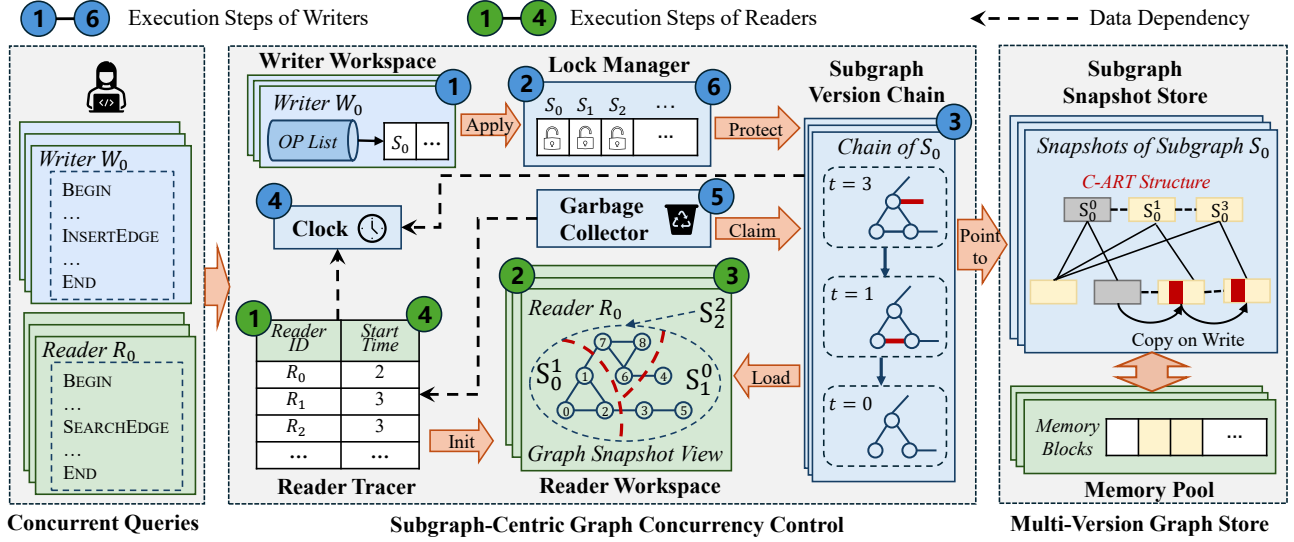
Figure 4: An overview of RapidStore.

Table 2: Performance of *Search* and Triangle Counting (TC).

Dataset	Method	Search (TEPS)			TC (s)
		General	Low Deg.	High Deg.	
lj	CSR	**7116.73**	**8567.41**	**6745.4**	**47.21**
	Sortledton	1898.65	2139.71	2414.32	153.30
g5	CSR	**3342.92**	**7437.11**	**3031.4**	5268.23
	Sortledton	635.33	1830.93	546.09	18786.13

Table 2 presents results comparing the performance of search operations and TC between the CSR format and Sortedlton. The results indicate that Sortedlton is approximately 3.26x slower than CSR for search operations and 3.12x slower for TC. This significant performance gap highlights the following challenge:

Challenge 3: How can we design an efficient graph data store that enables fast search operations while maintaining a balance across scan and insert operations?

4 AN OVERVIEW OF RAPIDSTORE

The significant issues and challenges faced by existing methods severely limit the ability to perform fast and scalable concurrent graph queries. To address these problems, we propose **RapidStore**, a holistic solution that combines a novel graph concurrency control mechanism with an optimized graph data store.

Execution Flow. Figure 4 provides an overview of RapidStore. The system uses a logical clock, initialized to 0, to coordinate query execution and versioning. Since write queries update the graph with a priori known write sets, RapidStore coordinates them using the classical MV2PL protocol to ensure their serializability.

Specifically, RapidStore executes a write query in six steps: ① Identify the set of subgraphs S to be modified based on the write set. ② Obtain locks on these subgraphs to ensure exclusive access during updates among write queries. ③ Create new versions of the subgraphs using the copy-on-write strategy, assigning each a version number equal to the current clock value plus 1. ④ Commit the changes and increment the clock by 1 to represent the new state of the graph. ⑤ Remove outdated versions based on the read

queries in execution to free up resources. ⑥ Release the locks held by the write query, allowing other queries to access the subgraphs.

In contrast, RapidStore executes a read query as follows: ① Register the query with the start time obtained from the current clock value. ② Build the graph snapshot by selecting the appropriate versions of subgraphs based on the start time. ③ Perform graph data access operations on this snapshot to complete the query. ④ Unregister the query after execution is complete. The initial graph G_0 is associated with version 0, indicated by the initial value of the logical clock. As a result, the reader execution flow remains consistent regardless of whether updates have occurred, i.e., whether the clock is 0 (no updates) or greater than 0 (some updates committed).

System Design. To support this concurrency control method, our key idea is to exploit the characteristics of graph queries through a decoupled system design that separates the management of read and write queries, maintains coarse-grained versions at the subgraph level, and decouples version data from graph data. Specifically, a write query creates a new snapshot for the subgraphs it modifies using the copy-on-write strategy instead of creating a version at the edge level, while read queries construct a graph snapshot by combining snapshots of these subgraphs. Additionally, RapidStore employs a memory pool to support the copy-on-write strategy, reducing the overhead of frequent memory allocation and deallocation by the operating system.

Advantages and Novelty. The novel decoupled design offers significant advantages over traditional edge-versioning approaches:

Non-Blocking Reads through Decoupled Query Management. While existing approaches like Sortledton and Teseo also manage to write and read queries differently, they require writers to acquire exclusive locks and readers to acquire shared locks on the neighbor sets of vertices to be accessed, leading to blocking between them. In contrast, RapidStore's separation of read and write queries ensures that read queries do not require any lock operations after registration. This novel approach allows read queries to execute efficiently without interference from concurrent writes, and vice versa.

Efficient Snapshot Retrieval with Coarse-Grained Subgraph Versioning. Read queries need to work on a graph snapshot. Traditional edge-versioning strategies are inadequate for fast snapshot retrieval because they require scanning the entire graph to find and store the correct version for each edge, incurring prohibitive time and space costs. By maintaining versions at the subgraph level rather than per-edge, RapidStore enables fast graph snapshot retrieval by combining snapshots of subgraphs without scanning and storing each edge individually. This strategy significantly reduces the time and space costs associated with version maintenance.

Elimination of Version Checks via Decoupled Version and Graph Data. In traditional approaches, version and graph data are stored together, requiring both to be loaded from memory to the CPU and frequent version checks during graph operations, which adds significant overhead. By decoupling version data from graph data and utilizing coarse-grained versioning, RapidStore eliminates the need for both read and write queries to check versions during running time, reducing operational overhead and improving performance.

Additionally, RapidStore's optimized graph data store supports the above graph data management methodology and effectively balances scan, search, and insert operations, providing consistent efficiency across different types of graph workloads. These innovations collectively allow RapidStore to overcome the limitations of traditional edge-versioning approaches, offering a more efficient and scalable solution for dynamic graph storage systems. In the following sections, we will detail the subgraph-centric concurrency control and the multi-version graph store.

5 SUBGRAPH-CENTRIC CONCURRENCY CONTROL

In this section, we propose the subgraph-centric graph concurrency control mechanism.

5.1 Coarse-Grained Version Management

Given a read query R starting at time t, R needs to operate on the graph snapshot G_t to ensure correctness. A straightforward method to eliminate version checks is to materialize G_t at the beginning of R and then execute R on this materialized snapshot. However, with edge-versioning strategies, materializing G_t requires scanning every edge to retrieve and store the correct versions, incurring prohibitive time and space costs.

To address this problem, we propose a coarse-grained version management approach that maintains versions at the subgraph level. Specifically, RapidStore partitions the graph G by dividing its vertex set $V(G)$ into a set of equal-sized partitions P. Each subgraph S consists of the vertex set P and the edges adjacent to vertices in P, formally defined as $V(S) = P$ and $E(S) = \{e(u,v) \in E(G) \mid u \in P\}$. RapidStore currently uses a simple graph partitioning strategy, which assigns continuous blocks of $|P|$ vertices to each partition based on their IDs. Using the graph in Figure 4 as an example, we set the partition size $|P|$ to 3 and partition the nine-vertex graph into three subgraphs: vertices 0–2 and their adjacent edges form S_0, 3–5 form S_1, and 6–8 form S_2.

RapidStore maintains a version chain for each subgraph, where each new version points to its predecessor as illustrated in Figure

4. When S is modified, RapidStore creates a new version of S, leveraging the underlying multi-version graph store (see Section 6). The version chains are stored separately from the graph data, with each version keeping a pointer to its corresponding subgraph snapshot, effectively decoupling version information from graph data.

5.2 Concurrency Control Protocol

In existing graph storage systems, both read and write queries must acquire locks on the vertices they access, leading to interference between readers and writers and reducing overall concurrency. RapidStore decouples the management of read and write queries to alleviate this issue.

5.2.1 Concurrency Control for Write Queries. To synchronize write queries, we adopt MV2PL. Specifically, given a write query W_0 that intends to update a set of vertices ΔV, we first identify the subgraphs ΔS containing these vertices and thus require modification. Inspired by the locking strategy in Sortledton, we acquire locks on the subgraphs in ΔS in ascending order of their subgraph IDs. This sorted locking order prevents deadlocks by ensuring a consistent lock acquisition sequence across concurrent write queries. After obtaining all the necessary locks, we update each subgraph in ΔS by creating new snapshots based on the latest snapshots that reflect the changes introduced by W_0 (see Section 6 for details).

To coordinate read and write queries and maintain consistency, RapidStore introduces two timestamps: the global write timestamp (t_w), which tracks the order of write query commits, and the global read timestamp (t_r), which indicates the latest consistent snapshot available to read queries. The commit phase for W_0 involves incrementing t_w by 1 atomically and assigning the new value to a local commit timestamp t, representing the commit time of W_0. We then assign the commit timestamp t to the new versions of the modified subgraphs, linking them to the head of their respective version chains to make them the most recent versions. Next, we poll the current value of t_r; if $t_r = t - 1$, we atomically increment t_r by 1 to advance the global read timestamp. MV2PL ensures the serializability of write queries. The polling and conditional increment of t_r guarantee that write queries commit in a serial order determined by their commit timestamps. It also ensures consistent snapshots for read queries, as read queries operating with timestamp t_r access a consistent snapshot. After the commit phase, W_0 performs garbage collection on the modified subgraphs to remove obsolete versions (as detailed in Section 5.3) and then releases all acquired locks, allowing other queries to proceed.

Take W_0 in Figure 4 as an example. Suppose it inserts edge $e(1,6)$ into the graph when $t_w = 2$. Based on the vertex IDs, the affected subgraphs are S_0 and S_2. To avoid deadlock, W_0 acquires locks on S_0 and S_2 in subgraph ID order. It then creates new snapshots of both subgraphs using the copy-on-write strategy and links them to their respective version chains. Given $t_w = 2$, the new version is set to 3 by atomically incrementing t_w. After committing, t_r is updated to 3, making the change visible to readers. Finally, W_0 performs GC to remove obsolete versions and releases the locks.

5.2.2 Concurrency Control for Read Queries. In RapidStore, read queries do not require any locks on subgraph snapshots. Consequently, they neither block write queries nor are they blocked by

them, allowing for high concurrency and performance. To keep track of active read queries, RapidStore employs a mechanism called the *reader tracer*. The reader tracer is an array where each element is an 8-byte integer. The highest bit of each integer (referred to as the *status* bit) indicates whether the slot is in use (1) or free (0). The remaining bits store the start timestamp of a read query. By default, the size k of the array is set to the number of CPU cores in the machine, but the user can configure it to suit different workloads.

When a read query R begins execution, it registers itself in the reader tracer through the following steps: 1) Loop over the reader tracer to locate an empty slot based on the status; and 2) Set the status to 1 and its start timestamp t as the current read timestamp t_r. The operation can be executed with atomic compare-and-swap (CAS) instructions without locks, ensuring that the registration process is both efficient and thread-safe.

After registration, R constructs its graph snapshot view by iterating over the version chains of all subgraphs and selecting the appropriate subgraph snapshots (the latest version of each subgraph with timestamp $t \leqslant t_r$). Pointers to these snapshots are copied into R's reader workspace. When accessing the version chain of a subgraph, R does not require any locks because:

(1) The start timestamp t ensures that only committed subgraph snapshots with versions less than or equal to t are visible to R.
(2) Writers create new subgraph snapshots using the copy-on-write strategy, which does not affect existing snapshots.

During its execution, R accesses graph data according to its constructed snapshot view, ensuring consistency without impeding concurrent write operations. Figure 4 illustrates the snapshot construction process. Reader R_0 obtains a start timestamp $t_r = 2$. For subgraph S_0, it traverses the version chain and selects the latest version with timestamp $t \leqslant 2$, which is $t = 1$, and copies the pointer to snapshot S_0^1 into its workspace. The same procedure is applied to subgraphs S_1 and S_2, completing the snapshot view. After R completes its execution, it unregisters itself by resetting the status bit in its slot in the reader tracer back to 0 and setting the start timestamp to a maximum value (e.g., the largest representable integer). This action marks the slot as available for future read queries.

5.3 Garbage Collection

Each update to a subgraph creates a new snapshot, leading to an ever-growing version chain for that subgraph. Similar to garbage collection (GC) approaches in existing works [10, 24, 41, 49], RapidStore reclaims stale versions based on the timestamps. Specifically, a subgraph version can be reclaimed if it is not the latest in the version chain and is not being used by any active readers. After committing its updates, a writer W performs GC as follows:

(1) W scans the reader tracer to collect the start timestamps of all active readers. To ensure it does not block other queries, W retrieves each value in the reader tracer using atomic operations.
(2) Based on the collected start timestamps, W loops over the version chain of the modified subgraphs to determine which versions can be reclaimed.

RapidStore leverages writers to perform GC because frequently modified subgraphs are more likely to require GC, and performing GC during write operations benefits from spatial locality without the need for additional background service threads. The garbage

collection of subgraph snapshots is supported by the underlying multi-version graph store, which will be introduced in Section 6.

5.4 Analysis of Graph Concurrency Control

Proposition 5.1 guarantees the correctness of the proposed concurrency control mechanism. We provide a proof sketch here and include the full details in the technical report [18].

PROPOSITION 5.1. *The subgraph-centric concurrency control mechanism guarantees the serializability of both write and read queries.*

PROOF. MV2PL and the enforcement of commit order based on t_w and t_r ensure that write queries are serialized according to their commit timestamps. Meanwhile, read queries access a consistent snapshot of the graph at their start time, proceeding without interfering with ongoing writes. Since the start timestamp depends on t_r, which advances based on the commit timestamps of write queries, both write and read queries are serialized in commit timestamp order. Thus, the proposition is proven. □

Time and Space Cost. We analyze the time and space overhead introduced by the graph concurrency control mechanism. Let $p = \lceil \frac{|V|}{|P|} \rceil$ denote the number of subgraphs in G, and let k be the size of the reader tracer array corresponding to the maximum number of concurrent read queries. We have the following proposition, the proof of which is included in the technical report.

PROPOSITION 5.2. *For any subgraph S, the length of its version chain is at most $k + 1$.*

For a read query R, the reader workspace requires $O(p)$ space to store pointers to subgraph snapshots. The concurrency control for R incurs $O(p \times k)$ time overhead: 1) Registering R takes $O(k)$ time, as it searches the reader tracer; 2) Constructing the snapshot view takes $O(p \times k)$ time since R traverses the version chain for each subgraph; and 3) Unregistering R takes $O(1)$ time. This process is efficient because new versions are at the head of the version chain, and R obtains the latest value of t_r.

A write query W can be blocked by other write queries during lock acquisition. The overhead of lock contention depends on the concurrency of write queries; here, we analyze the overhead without lock contention. Let ΔS be the set of subgraphs modified by W, and let $s = |\Delta S|$. The workspace cost of W is $O(s)$. Since RapidStore sorts ΔS to obtain locks, the time cost of acquiring locks is $O(s \log s)$. Committing updates takes $O(s)$ time, as W sets the version value for each subgraph in ΔS. Finding versions to be reclaimed takes $O(k \log k + sk)$ time: W acquires start timestamps of active readers, sorts them, and traverses the version chain to determine which versions to reclaim. Since W typically updates only a few subgraphs, s remains small, and the version chain length k is bounded as per Proposition 5.2. This ensures that both commit and garbage collection operations are highly efficient.

Impact of Partition Size. The partition size (i.e., subgraph size) $|P|$ directly influences the number k of subgraphs. At the minimum, $|P|$ can be set to 1, where each subgraph contains only a single vertex. This setup can improve write performance by reducing lock contention among concurrent write queries. However, as discussed earlier, smaller subgraph sizes increase the space and time overhead for read queries. Additionally, minimizing subgraph size limits
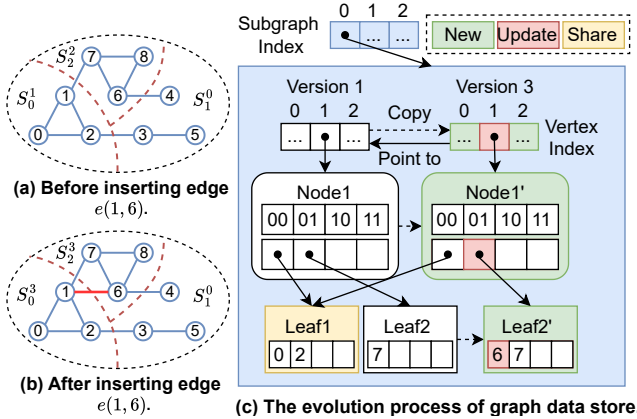
(a) Before inserting edge $e(1, 6)$.

(b) After inserting edge $e(1, 6)$.

(c) The evolution process of graph data store.

Figure 5: Overview of the multi-version graph store design with the copy-on-write strategy.

opportunities to optimize the storage of small neighbor sets within a subgraph, thereby decreasing read efficiency. To balance read and write performance, we set $|P|$ to 64 by default, which achieves good empirical performance. We currently use a static partitioning strategy, randomly dividing the graph into equal-sized partitions at initialization, and dynamically selecting data structures for neighbor sets based on vertex degrees. A promising research direction is to design adaptive partitioning strategies that adjust to operation skewness, such as recent write frequencies, to further reduce write conflicts and improve write performance.

6 MULTI-VERSION GRAPH STORE

We present the multi-version graph store that efficiently maintains graph data in this section.

6.1 Design of Graph Store

To efficiently handle concurrent read and write queries, a graph store must meet two key requirements. First, it must quickly generate a snapshot of a subgraph S upon updates to support subgraph-centric concurrency control and ensure the correctness of concurrent queries. Second, since graph queries commonly follow a "vertex-neighbor" access pattern, the store must efficiently retrieve the neighbor set $N(u)$ for any vertex $u \in V(S)$ and support fast search, insertion, and scan operations on $N(u)$. To achieve these goals, we adopt a copy-on-write strategy for updating S, ensuring that read queries remain unaffected by write queries.

Figure 5 illustrates the design of the store. The example assumes 4-bit vertex IDs, using 2 bits per level in a classical radix tree. When W_0 inserts edge $e(1, 6)$ at logical time $t_w = 2$, it locates the affected subgraph S_0 via the subgraph index, which maintains pointers to version chains. Each subgraph version contains a vertex index pointing to radix trees that store neighbor sets. Vertices with common prefixes share the same leaf node (e.g., vertex 1's neighbors 0 and 2 in Leaf 1). The insertion triggers a copy-on-write: the affected radix tree path (from leaf to root) is duplicated and updated, creating a new version S_0^3 (with t_w incremented to 3). Subgraph S_2 is updated similarly, as $e(1, 6)$ also affects it. Its version is also set to 3, since both updates are committed in the same transaction.

The vertex index is an array where each entry corresponds to a vertex $u \in V(S)$ and stores a pointer to its neighbor set $N(u)$. As

Table 3: Comparison of filling ratios of ART and C-ART.

Dataset	lj	ot	ldbc	g5	tw	fr
ART	2.17%	3.13%	3.44%	2.40%	2.24%	2.94%
C-ART	66.17%	67.12%	64.27%	67.80%	67.22%	64.70%

described in Section 5, $V(S)$ is a contiguous subset of $V(G)$ in the range $[0, |V(G)|)$, enabling $O(1)$ lookup of $N(u)$ by vertex ID. Since vertex indices are small, copying them for new versions is fast and lightweight. To handle degree skewness in real-world graphs, $N(u)$ is stored differently based on vertex degree: high-degree vertices use a tree structure, while low-degree vertices use small arrays. These small arrays are further grouped into a tree to optimize memory usage. Next, we first introduce the *compressed adaptive radix tree* (C-ART) in Section 6.2, which stores the neighbor set for high-degree vertices. Then, in Section 6.3, we present the *clustered index*, which centrally stores low-degree vertices within a subgraph to fully exploit locality.

6.2 Compressed Adaptive Radix Tree

As discussed in Section 2, ART is memory-efficient and supports fast retrieval and insertion operations. Its hierarchical structure, which limits each node to a maximum of 256 entries, enables an efficient copy-on-write mechanism by duplicating the root-to-leaf path. This makes ART a suitable choice for storing $N(u)$. However, storing vertices individually in each leaf can degrade scan performance due to frequent node traversal. A straightforward approach to improve scan performance is to organize the leaves into contiguous segments of size $B = 256$, each storing vertices from $N(u)$. However, the distribution of $N(u)$ is often skewed and sparse across the range $[0, |V(G)|)$, leading to low filling ratios (the proportion of occupied entries within each segment). As shown in Table 3, filling ratios for various graphs are below 4%, resulting in poor scan performance and significant memory waste. To address this issue, we propose the *compressed adaptive radix tree* (C-ART), which compresses leaves to significantly improve the filling ratio. This compression enhances memory locality and traversal efficiency, making C-ART a more effective solution for graph storage.

Compressing Leaves. ART employs path compression by vertically merging paths and nodes within the tree, reducing memory usage and search costs. In contrast, C-ART, introduced in this paper, applies horizontal compression to leaves, enhancing scan performance in graph-related operations. Specifically, unlike ART, where each leaf stores a single vertex, C-ART leaves can store up to $B = 256$ vertices. Each C-ART leaf is defined by a *longest common prefix* (LCP) shared by all vertices within the leaf, with the *depth* indicating the LCP's length in bytes. This design allows multiple keys in a C-ART node to point to the same leaf, effectively compressing the leaves of multiple keys into a single leaf. This improves the filling ratio (as shown in Table 3), enhancing scan performance and reducing memory consumption.

Figure 6 provides an example of C-ART. Each leaf contains up to 256 vertices, with the LCP and its length recorded. Multiple keys within a node can point to the same leaf (e.g., two keys in Node2 point to Leaf3). Compared to ART in Figure 1, C-ART stores vertices much more compactly. Next, we describe the graph operations on a neighbor set $N(u)$ of vertex u stored in C-ART. The insert operation illustrates the construction of a C-ART.
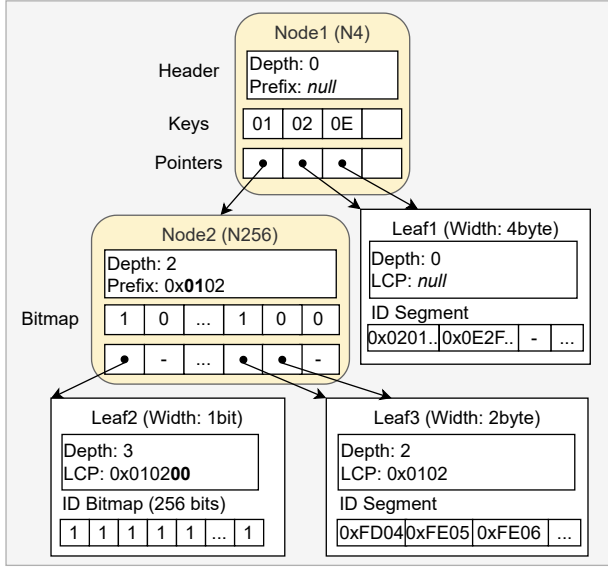
Figure 6: An example of C-ART storing the same elements as ART shown in Figure 1.

1) *Search(u, v)* locates vertex v in $N(u)$ by traversing the C-ART nodes based on the byte sequence of v, similar to the process in ART. Upon reaching a leaf, C-ART performs a binary search to locate v within the leaf. This design ensures fast search performance due to the limited number of vertices per leaf.

2) *Scan(u)* traverses the C-ART structure in a depth-first-search (DFS) order, enabling sequential memory access when processing leaves. As multiple vertices are stored continuously in leaves, this approach significantly improves performance for graph analytic queries compared with ART.

3) *Insert(u, v)* adds vertex v to $N(u)$ stored in a C-ART. Since C-ART inherits the node management strategy of ART, the focus is on inserting v into a leaf. Initially, C-ART is created with a root node and a leaf segment containing $B = 256$ entries. To insert v, the position in the leaf is determined using *Search(u, v)*. Let the target leaf currently contain b vertices. As illustrated in Figure 7, the insertion process involves three possible cases based on the state of the leaf:

- **Case 1:** $b < B$. Insert v directly into the leaf and update the affected pointers in the parent node to reflect the change.
- **Case 2:** $b = B$ **and the leaf is shared by multiple keys.** Identify the first key with a pointer offset of at least $\frac{B}{2}$. Split the leaf at this offset, then insert v following the procedure in Case 1.
- **Case 3:** $b = B$ **and the leaf is associated with a single key.** Compute the LCP of all vertices in the leaf, create a new internal node using this LCP as its prefix, split the leaf as in Case 2, and insert v accordingly.

4) *Delete(u, v)* removes vertex v from $N(u)$. The operation begins by locating v using *Search(u, v)* and then removing it from the leaf. If the number of vertices in the leaf drops below $\frac{B}{2}$ after deletion, the leaf is checked for potential merging with adjacent sibling leaves to maintain a high filling ratio.

5) When two neighbor sets are stored in C-ARTs, set intersections can be performed efficiently based on the structures of the two trees.
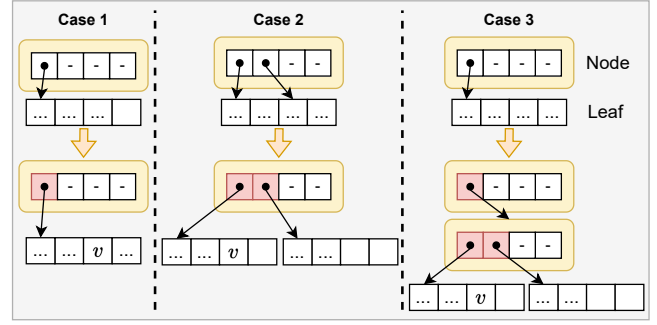

Figure 7: Insertion of vertex v into $N(u)$ stored in a C-ART. Red highlights the updated pointers.

Note that both insert and delete operations are executed on the copied root-to-leaf paths to ensure consistency and isolation.

Optimization. First, vertex IDs are compressed by removing the LCP and storing only the unique suffixes. During computation, the full vertex ID is reconstructed by concatenating the LCP with the stored suffix. This approach reduces memory usage and minimizes data movement between memory and the CPU, accelerating computation. If the vertex IDs in a leaf differ only in their last byte, they are stored using a 256-bit bitmap. Second, to address the overhead of iterating through up to 256 child pointers per node, many of which may be empty, a bitmap is maintained in each node to record the presence of non-empty pointers. By utilizing AVX2 instructions, we efficiently identify the indices of set bits in the bitmap, bypassing empty pointers and significantly improving performance.

6.3 Clustered Index

The overhead of storing neighbor sets for low-degree vertices in C-ART can diminish its benefits, as the tree's depth depends on the vertex ID length. Therefore, we introduce the *clustered index* to store neighbor sets of low-degree vertices. The clustered index is implemented as a B+ tree where the keys are edge pairs $e(u, v)$ representing source and destination vertices. Neighbor sets of low-degree vertices in $V(S)$ are stored sequentially within the clustered index according to the (u, v) order, eliminating the random memory access overhead caused by traversing different neighbor sets. The vertex array records the position of each neighbor set within the clustered index to accelerate the visit of $N(u)$. Since $|V(S)|$ is small (64 by default) and the clustered index exclusively stores neighbor sets of low-degree vertices, the depth of the clustered index remains low, typically two levels. Consequently, graph access operations and updates are executed efficiently.

6.4 Garbage Collection of Graph Store

RapidStore employs a classical reference counting method for GC. Specifically, each node and leaf maintains a reference count that records the number of parents from different snapshots referring to it. The reference count is incremented by one during the copy-on-write operation along the root-to-leaf path. Conversely, when graph concurrency control claims a snapshot version, RapidStore traverses from top to bottom, decrementing the reference count. If a node's reference count drops to zero, it is reclaimed.

6.5 Analysis of Multi-Version Graph Data Store

The clustered index follows the classical B+ tree's time and space complexities. Here, we analyze the time and space costs of C-ART. Let $|P|$ denote the partition size, B the leaf segment size, $d = |N(u)|$, and w the length of a vertex ID in bytes.

Time Cost. Locating $N(u)$ for a vertex u takes $O(1)$ time using the vertex index, which can be omitted. Compressing leaves does not increase the number of nodes compared to ART. Consequently, *Scan(u)* operates in $O(d)$ time, the same as ART, while *Search(u, v)* requires $O(w + \log B)$ time due to the binary search within the leaf. For *Insert(u, v)*, two operations are involved: 1) Copying the root-to-leaf path, which takes $O(wB)$ time since the depth of C-ART is bounded by w. 2) Inserting v, which requires $O(|P| + w + \log B)$ time to find the insertion position and potentially split the leaf. Because of the copy-on-write strategy, delete has the same cost as insert. For set intersections between two neighbor sets $N(u)$ and $N(v)$ with degrees d_1 and d_2 (assume $d_1 \leqslant d_2$), two strategies are employed: If $\frac{d_2}{d_1} < 10$, a merge-based set intersection is performed, taking $O(d_1 + d_2)$ time. Otherwise, for each neighbor in $N(u)$, existence in $N(v)$ is checked, taking $O(d_1 \times (w + \log B))$ time.

For vertex operations, *VertexDelete(u)* removes vertex u by first deleting all its incident edges $e(u, v)$ for each $v \in N(u)$. This involves acquiring locks on the subgraphs containing the neighbors, following MV2PL rules. After edge deletion, a flag bit is unset to mark the vertex as removed, and its ID is added to a queue for potential reuse. Vertex deletions are generally rare. *VertexInsert(u)* adds vertex u by first checking the queue for reusable IDs. If available, one is reused; otherwise, N is atomically incremented to assign a new ID. The vertex is then added to the appropriate subgraph by setting a flag bit, requiring a lock only on that subgraph. The time complexity of *VertexDelete(u)* is proportional to the number of adjacent edges to be deleted. After edge removal, deleting the vertex from the subgraph incurs an additional $O(|P|)$ cost due to snapshot creation. *VertexInsert(u)* has the same $O(|P|)$ cost, as it involves no additional operations.

In summary, RapidStore can incur higher write overhead than per-edge versioning approaches due to its copy-on-write strategy. This design trade-off is intentional to optimize read performance, as graph workloads are typically read-intensive. Nevertheless, write operations remain efficient in practice, benefiting from the small values of w and B, and experimental results confirm that RapidStore achieves good write performance.

Space Cost. Given S, the vertex index requires $O(|P|)$ space. RapidStore uses $O(d)$ space to store $N(u)$ since each leaf contains at least one vertex. Each entry in the clustered index corresponds to a single neighbor. Thus, the overall space complexity for storing the graph is $O(|V(G)| + |E(G)|)$. The additional overhead from multi-versioning is minimal. The number of subgraph versions is bounded by the number of concurrent read queries (as detailed in Section 5), and each snapshot only duplicates a root-to-leaf path, which incurs negligible space overhead due to its small size.

Hyperparameters. RapidStore has two hyperparameters: the partition size $|P|$ and the segment size B. As analyzed above, increasing these values can improve read efficiency since $N(u)$ are stored in larger chunks. However, larger values decrease write performance.

Table 4: The detailed information of the graph datasets.

Dataset	Abbr.	$|V|$	$|E|$	Avg. Deg.	Max Deg.	Size(GB)
LiveJournal	*lj*	4M	43M	17.4	14815	0.67
Orkut	*ot*	3M	117M	76.3	33313	1.7
LDBC	*ldbc*	30M	176M	11.8	4282595	2.84
Graph500	*g5*	9M	260M	58.7	406416	4.16
Twitter	*tw*	21M	265M	24.8	698112	4.11
Friendster	*fr*	65M	2B	55.1	5214	30.1

We empirically set $|P|$ and B to 64 and 512 to balance read and write performance, and this value cooperates with AVX2 instructions.

7 EXPERIMENTS

Experiment Setup. We conduct our experiments on a machine equipped with Intel Xeon Gold 6430 @ 3.40GHz processors. The machine features 256GB of DDR5 memory and a maximum bandwidth of 61.2GB/s. The CPU has a 60MB L3 Cache and 32 cores. We compile the source code using GCC v10.5.0 with O3 optimization. Each experiment is executed five times, and we report the median.

Graph Datasets. We utilize a diverse set of graph datasets to evaluate the systems' time and space performance, as depicted in Table 4. These graphs vary in size and structure and are widely used in previous graph research, enhancing the comprehensiveness of our results. Table 4 provides a brief description of each dataset.

Graph Analytic Workload. For graph analytic workloads, we select five algorithms from GAPBS [2]: PageRank (PR), breadth-first search (BFS), single-source shortest path (SSSP), weakly connected components (WCC), and Triangle Counting (TC). These workloads cover a range of graph data access patterns and represent common graph analytics tasks. The parameters for each algorithm were set according to standard practices, such as 10 iterations for PageRank.

Systems Under Study. We compare RapidStore with several of the latest graph systems: **Sortledton** [14] is a library using a two-level array to store vertices. It stores small neighbors directly in arrays, while large neighbors are stored in unrolled skip lists. **Teseo** [7] is a library that stores vertices and edges together in a Packed Memory Array (PMA) [6] indexed by ART. **Aspen** [9] applies copy-on-write and versions the global graph, focusing on read performance. It uses PAM tree [40] to store edges. **LiveGraph** [49] stores the vertex's neighbors as logs in timestamp order, improving insertion and *scan* but limiting its functionality. **Spruce** [36] is a library that uses an ART-like structure to index vertices and buffer block and sorted arrays to store neighbors independently. However, Spruce does not provide full isolation support and faces OOM issues on the three larger datasets. Therefore, we did not include Spruce. We also consider **LSMGraph** [47], leveraging an LSM tree to optimize disk performance. However, the source code is currently unavailable. All systems are implemented in C++. Additionally, we include CSR as a baseline to demonstrate optimal static performance.

Supplement Experiments. Due to space constraints, we present additional results including partition size evaluation, memory consumption, basic read operations, multicore write scalability, batch updates, real insert traces on *ldbc*, and insertions with varying neighbor sizes in the full version of the paper. [18]

Table 5: Performance of graph analytics. For the CSR, we report the latency time. For the systems, we report the slowdown over CSR. The best performance is in bold. "-" means not supported. "OOT" means it cannot be completed within 5 hours.

	lj					*ot*					*ldbc*				
	BFS	**PR**	**SSSP**	**WCC**	**TC**	**BFS**	**PR**	**SSSP**	**WCC**	**TC**	**BFS**	**PR**	**SSSP**	**WCC**	**TC**
CSR	0.56s	4.66s	1.42s	1.95s	48.21s	0.64s	7.91s	1.64s	1.73s	232.80s	2.68s	20.57s	6.36s	5.73s	442.10s
Sortledton	7.23	6.69	2.45	3.11	3.18	5.97	4.77	2.46	3.07	3.07	8.35	8.49	3.04	4.20	4.94
Teseo	15.25	4.32	3.53	2.52	3.99	3.83	2.74	3.50	4.29	4.34	6.28	4.01	3.84	3.23	7.31
Aspen	7.11	7.10	4.81	3.97	6.24	3.67	3.34	2.71	2.80	10.15	8.90	9.07	5.97	4.58	9.87
LiveGraph	21.77	17.12	4.76	7.01	-	15.77	10.49	7.84	6.86	-	11.46	13.91	4.99	6.41	-
RapidStore	**2.11**	**1.53**	**1.61**	**1.18**	**1.39**	**1.59**	**1.35**	**1.28**	**1.22**	**1.51**	**1.82**	**1.65**	**1.79**	**1.36**	**1.49**
	g5					*tw*					*fr*				
	BFS	**PR**	**SSSP**	**WCC**	**TC**	**BFS**	**PR**	**SSSP**	**WCC**	**TC**	**BFS**	**PR**	**SSSP**	**WCC**	**TC**
CSR	2.00s	20.98s	4.25s	4.88s	5268.23s	2.92s	24.87s	6.71s	10.48s	4631.08s	31.80s	323.60s	68.20s	65.39s	1974.24s
Sortledton	4.42	4.49	2.42	3.06	3.57	6.70	6.80	2.60	3.45	3.47	3.18	3.79	1.92	3.58	9.62
Teseo	4.65	2.37	3.02	2.70	OOT	4.61	3.36	3.87	2.48	OOT	1.91	2.10	2.13	2.04	10.35
Aspen	4.35	4.21	3.45	2.93	OOT	7.61	7.03	5.23	3.89	OOT	2.41	1.80	2.81	2.32	OOT
LiveGraph	8.90	9.64	4.07	5.74	-	12.39	12.51	4.46	6.29	-	OOM	OOM	OOM	OOM	-
RapidStore	**1.60**	**1.61**	**1.60**	**1.24**	**1.89**	**1.71**	**1.68**	**1.80**	**1.33**	**2.26**	**0.92**	**1.04**	**1.17**	**1.16**	**3.67**

(a) Performance of edge insertion.
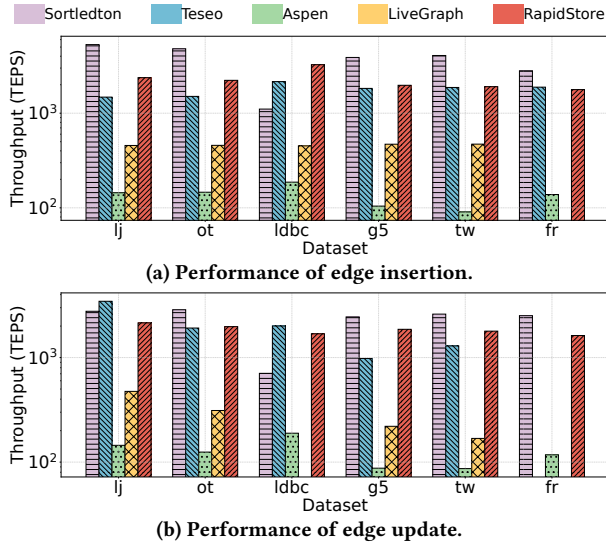

(b) Performance of edge update.

Figure 8: Performance of write operations.

7.1 Evaluation on Read Performance

Table 5 shows the performance of graph analytics workloads. **PR** and **WCC**, which involve sequential vertex and neighborhood access, demonstrate significant improvements with RapidStore. It reduces latency by 31.86-64.50% for PR and 43.30-57.91% for WCC compared to the best-performing system, showcasing its superior scan performance. Teseo ranks second, benefiting from its PMA's locality advantages. **BFS**, characterized by random vertex access and sequential neighborhood access, sees a latency reduction of 51.71-71.08% with RapidStore. For **SSSP**, which requires random neighborhood access, RapidStore reduces latency by 30.81-47.89%. Sortledton ranks second on most datasets, with other systems showing similar performance trends. These results highlight RapidStore's capability to efficiently handle random access patterns, a critical requirement in dynamic graph applications.

TC requires efficient *intersect* operations, which depend on fast *search* and *scan*. RapidStore achieves latency reductions of 34.85-69.84%, demonstrating robust performance in these tasks. Notably,

RapidStore exhibits balanced performance across workloads, occasionally showing smaller slowdowns on sequential *Scan*-based tasks like PR. This indicates its adaptability to diverse access patterns, making it highly versatile for graph analytics workloads. In summary, RapidStore achieves significantly better read performance than existing systems.

7.2 Evaluation on Write Performance

In this section, we evaluate the system's edge insertion and deletion performance. While RapidStore supports property updates, this was excluded from the evaluation because Sortledton crashes during updates, and Teseo and Aspen do not support this functionality.

Insert. We evaluate insertion performance by inserting edges in random order using 32 writer threads. For comparison, Aspen, designed for single-writer execution, is also tested with 32 threads. The results are shown in Figure 8a. Sortledton achieves the highest throughput, outperforming all other systems. RapidStore ranks second, with slightly lower throughput due to the overhead of copy-on-write compared to direct edge insertion. However, it benefits from optimized *search* and ID compression, which accelerate insertion preparation and reduce copy cost, resulting in only a 1.9–2.2x slowdown relative to Sortledton. On the *ldbc* dataset, RapidStore achieves the best performance, as Sortledton suffers from severe lock contention caused by high skewness. Teseo ranks third, with throughput occasionally interrupted by PMA rebalancing. LiveGraph shows the lowest throughput, limited by inefficient *search* operations. Overall, RapidStore effectively reduces insertion overhead through constant-time *search* and compact ID encoding, particularly for high-degree vertices.

Update. This experiment evaluates update performance by repeatedly deleting and re-inserting 20% of edges over five rounds, generating version chains for the modified elements. Figure 8b shows the results. Sortledton's throughput drops by 34.01% compared to pure insertion, due to the overhead of managing version chains. Teseo performs well on small datasets but degrades on larger ones, where frequent updates trigger costly background garbage collection. In contrast, RapidStore and the other two systems show minimal performance change. RapidStore's throughput drops by only
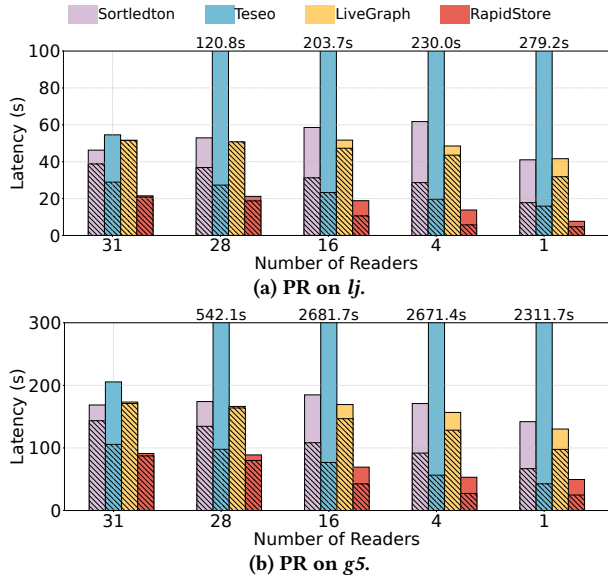
Figure 9: Read performance under varying numbers of readers and writers (total threads fixed at 32). The shadowed bars represent the latency of readers in the absence of writers.

14.67%, benefiting from constant-time *search*. Overall, RapidStore delivers stable and competitive write performance, slightly behind Sortledton except on *ldbc*, where Sortledton suffers from severe lock contention. These results highlight the sensitivity of Sortledton and Teseo to GC and locking, whereas RapidStore maintains consistent performance despite copy-on-write overhead.

7.3 Evaluation on Concurrent Read and Write

This section evaluates concurrent read and write performance, analyzing the impact of writers on readers and vice versa. The goal is to assess the concurrency capabilities of the systems under study. To reduce the influence of memory bandwidth limitations, property storage is disabled for all systems. Aspen is excluded from this evaluation as it supports only a single writer.

Read with Concurrent Writers. We first load the graph and then run a mixed workload with 32 threads, combining PR readers and edge update writers (delete + re-insert) to keep the graph size stable. As shown in Figure 9, RapidStore consistently achieves the lowest latency and best performance. While Teseo and Sortledton perform well under isolated workloads, their read performance degrades significantly under concurrent access due to severe lock contention between readers and writers. In contrast, RapidStore experiences minimal interference, showing negligible performance drop even with 31 or 28 writers.

Although RapidStore's decoupled design avoids reader-writer blocking, its read performance declines when the number of writers increases to 28 or 31 (i.e., only 4 or 1 readers remain). This is due to high scan throughput saturating memory bandwidth. As shown in Figure 11, adding writers introduces bandwidth contention, making RapidStore sensitive to memory bandwidth pressure. In summary, Teseo and Sortledton are limited by lock contention, while RapidStore is constrained by hardware bandwidth, demonstrating its ability to fully utilize system resources. It's worth noting that such
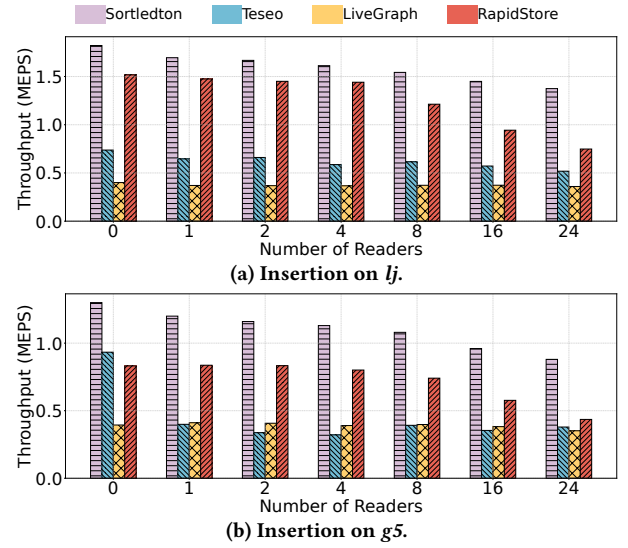

Figure 10: Insertion performance under varying numbers of readers. Each reader executes PR independently. The number of writers is fixed at 8.
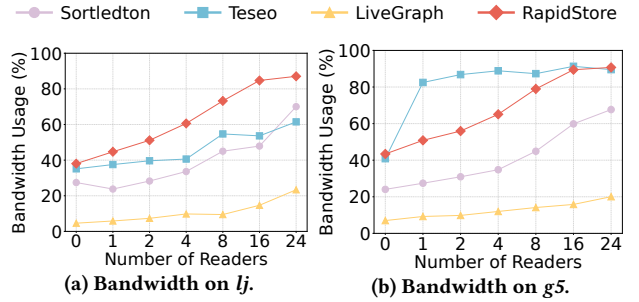

Figure 11: Corresponding avg. memory bandwidth usage rate in concurrent read and write experiments.

extreme write-heavy workloads are rare in practice, as: 1) graph applications are typically read-intensive, and 2) adding more writers yields diminishing returns due to increased contention (see evaluation on Multicore Scalability in the full version).

Interestingly, LiveGraph shows little impact from concurrent writers. This is because its slow scan and insert operations lead to low bandwidth usage. Additionally, it stores neighbor sets in append order, allowing readers to record set lengths without locking. However, as shown in prior experiments, LiveGraph suffers from poor overall performance due to its unsorted neighbor sets.

Write with Concurrent Readers. We initially load 80% of the edges and launch 8 writer threads. The number of readers varies from 0 to 24. Writers insert the remaining edges, while readers execute PageRank. Figures 10 and 11 report insertion throughput and memory bandwidth utilization, respectively.

As the number of readers increases from 0 to 4, RapidStore's insertion throughput drops by no more than 5.06%, demonstrating strong resilience to reader interference. When #readers exceeds 8, throughput declines further as RapidStore approaches the memory bandwidth limit due to its high scan performance. In contrast, Sortledton's throughput drops by 13.29% when #readers increases

Table 6: Ablation study results. Insertion throughput is measured in thousands of edges per second (TEPS), and graph analytics latency is measured in seconds.

Dataset	Method	Insert (TEPS)	Analytics Query (s)			
			BFS	PR	SSSP	WCC
	ART	**2445.59**	10.92	59.84	11.01	17.04
	ART + SC	2119.22	7.31	38.97	5.42	11.55
lj	C-ART + SC	1991.25	7.30	37.77	5.25	10.81
	C-ART + SC + VEC	1710.56	2.42	11.25	2.73	2.59
	C-ART + SC + CI	2373.05	**1.05**	**5.90**	**2.31**	**1.97**
	ART	1592.24	25.22	246.28	32.21	45.98
	ART + SC	1562.43	22.85	243.93	27.46	41.86
g5	C-ART + SC	1797.23	9.73	98.81	13.20	18.87
	C-ART + SC + VEC	1711.60	4.18	47.12	8.23	9.06
	C-ART + SC + CI	**1973.13**	**2.71**	**29.28**	**6.47**	**5.81**

from 0 to 4, despite low memory bandwidth usage. Teseo degrades more severely, with throughput reductions of 29.72% and 57.22% on the two datasets, primarily due to MVCC overhead. These results highlight the effectiveness of RapidStore's decoupled design, which avoids reader-writer lock contention and fully utilizes available memory bandwidth. Its performance degradation is not due to contention, but rather due to saturating hardware limits.

Summary. RapidStore achieves high performance and efficient hardware utilization under concurrent workloads. It delivers stable read and write performance in typical read-intensive configurations, such as 28 readers with 4 writers and 24 readers with 8 writers.

7.4 Ablation Study

We conduct an ablation study to evaluate the performance impact of the three key techniques proposed in this paper: subgraph-centric concurrency control (denoted as SC), C-ART for the graph store, and the clustered index (CI) for managing low-degree vertices. As ART is a well-studied data structure and per-edge versioning is widely adopted in dynamic graph systems, we use ART combined with per-edge versioning as the baseline. In addition, since Sortledton optimizes performance by storing neighbor sets of low-degree vertices in separate vectors (VEC), we incorporate this technique into RapidStore for comparison with our clustered index design.

Table 6 presents the results. Enabling subgraph-centric concurrency control (SC) improves read performance by eliminating blocking on read queries caused by concurrent writes and per-edge versioning, while slightly affecting write performance due to coarser-grained versioning. This confirms both our analysis and the effectiveness of SC. Replacing ART with C-ART significantly improves analytics performance on *g5*, while the improvement on *lj* is limited. This is because *lj* is a sparse graph where most vertices have small degrees, whereas *g5* contains many high-degree vertices that pose greater performance challenges. Incorporating VEC improves read performance, demonstrating the benefit of using specialized structures for low-degree vertices. However, VEC degrades write performance due to the overhead of managing separated vectors. In contrast, our clustered index (CI) not only improves read performance significantly but also enhances write performance. These results validate the effectiveness of each individual technique.

8 RELATED WORK

Tree-like Structures. Tree-like structures are crucial for designing dynamic graph systems. Terrace [29] employs a *Structure Hierarchy*

and uses B+ trees to store large neighborhoods efficiently. Additionally, binary search trees have been utilized for graph storage [8, 44]. Various radix tree variants have also been explored in graph contexts [32]. These structures enhance the performance of dynamic graph operations by optimizing data retrieval and storage.

Graph Systems. The pioneering works Ligra [37] and Ligra+[38] set the stage for high static read performance in graph processing. Other systems, such as Stinger [11] and NXGraph [4], implement *Segmentation* techniques to improve update efficiency. However, these early systems face challenges in efficiently handling concurrent read and write operations. In contrast, LLAMA [27] introduces delta storage to reduce update costs while ensuring snapshot isolation. Pensieve [46] further enhances performance by applying delta storage to high-degree vertices, effectively balancing read and write operations. Similarly, GraphOne [20] adopts a logging approach for updates, transforming them into a compact structure. LSMGraph [47] is a disk-based system that combines LSM Tree and CSR, providing efficient read and write. Its in-memory part uses CSR and skip lists to store low- and high-degree vertices. Some systems such as KickStarter [42], GraphBolt [26], and RisGraph [12] are designed for continuous graph processing. These systems enhance continuous graph algorithms [23, 48] by storing intermediate results during computation, reducing re-computation costs.

Graph Databases. Several graph databases, including Neo4J, Virtuoso, GraphFlow [19], and Kùzu [13], adopt specialized internal data structures (e.g., linked lists) to store and traverse graph data efficiently. An alternative approach augments relational databases with additional indexes [3, 5, 35], offering flexibility for diverse workloads but often at the cost of lower performance on graph-specific operations. Our work enhances in-memory dynamic graph storage through system-level optimizations, including subgraph-centric MVCC and C-ART, which reduce contention and improve memory access locality. These techniques could be integrated into existing graph systems to improve both throughput and concurrency. Furthermore, RapidStore's decoupled architecture and memory-efficient indexing offer promising directions for building high-performance, general-purpose graph storage engines.

9 CONCLUSION

In this paper, we present RapidStore, an efficient in-memory dynamic graph storage system optimized for concurrent queries. RapidStore introduces a decoupled design that separates read and write query management while isolating version data from graph data. This is achieved through a subgraph-centric concurrency control and an efficient multi-version graph store. RapidStore significantly reduces query execution times compared to existing solutions, maintains high concurrency performance with very high hardware utilization, and offers competitive write performance with lower memory consumption. These features establish RapidStore as a robust and effective dynamic graph storage solution. Besides, extending it to support persistence on emerging storage media, such as NVMe SSDs and cloud storage, is a promising direction for future research.

REFERENCES

[1] Christopher R Aberger, Andrew Lamb, Susan Tu, Andres Nötzli, Kunle Olukotun, and Christopher Ré. 2017. Emptyheaded: A relational engine for graph processing. *ACM Transactions on Database Systems (TODS)* 42, 4 (2017), 1–44.

[2] Scott Beamer, Krste Asanović, and David Patterson. 2015. The GAP benchmark suite. *arXiv preprint arXiv:1508.03619* (2015).

[3] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry Li, et al. 2013. {TAO}:{Facebook's} distributed data store for the social graph. In *2013 USENIX Annual Technical Conference (USENIX ATC 13)*. 49–60.

[4] Yuze Chi, Guohao Dai, Yu Wang, Guangyu Sun, Guoliang Li, and Huazhong Yang. 2016. Nxgraph: An efficient graph processing system on a single machine. In *2016 IEEE 32nd International Conference on Data Engineering (ICDE)*. IEEE, 409–420.

[5] James C Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, Jeffrey John Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, et al. 2013. Spanner: Google's globally distributed database. *ACM Transactions on Computer Systems (TOCS)* 31, 3 (2013), 1–22.

[6] Dean De Leo and Peter Boncz. 2019. Packed memory arrays-rewired. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. IEEE, 830–841.

[7] Dean De Leo and Peter Boncz. 2021. Teseo and the analysis of structural dynamic graphs. *Proceedings of the VLDB Endowment* 14, 6 (2021), 1053–1066.

[8] Laxman Dhulipala, Guy E Blelloch, Yan Gu, and Yihan Sun. 2022. Pac-trees: Supporting parallel and compressed purely-functional collections. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 108–121.

[9] Laxman Dhulipala, Guy E Blelloch, and Julian Shun. 2019. Low-latency graph streaming using compressed purely-functional trees. In *Proceedings of the 40th ACM SIGPLAN conference on programming language design and implementation*. 918–934.

[10] Cristian Diaconu, Craig Freedman, Erik Ismert, Per-Ake Larson, Pravin Mittal, Ryan Stonecipher, Nitin Verma, and Mike Zwilling. 2013. Hekaton: SQL server's memory-optimized OLTP engine. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. 1243–1254.

[11] David Ediger, Rob McColl, Jason Riedy, and David A Bader. 2012. Stinger: High performance data structure for streaming graphs. In *2012 IEEE Conference on High Performance Extreme Computing*. IEEE, 1–5.

[12] Guanyu Feng, Zixuan Ma, Daixuan Li, Shengqi Chen, Xiaowei Zhu, Wentao Han, and Wenguang Chen. 2021. Risgraph: A real-time streaming system for evolving graphs to support sub-millisecond per-update analysis at millions ops/s. In *Proceedings of the 2021 International Conference on Management of Data*. 513–527.

[13] Xiyang Feng, Guodong Jin, Ziyi Chen, Chang Liu, and Semih Salihoğlu. 2023. Kùzu Graph Database Management System. In *CIDR*.

[14] Per Fuchs, Domagoj Margan, and Jana Giceva. 2022. Sortledton: a universal, transactional graph data structure. *Proceedings of the VLDB Endowment* 15, 6 (2022), 1173–1186.

[15] Joseph E Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. 2012. PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs. In *10th USENIX symposium on operating systems design and implementation (OSDI 12)*. 17–30.

[16] Pankaj Gupta, Venu Satuluri, Ajeet Grewal, Siva Gurumurthy, Volodymyr Zhabiuk, Quannan Li, and Jimmy Lin. 2014. Real-time twitter recommendation: Online motif detection in large dynamic graphs. *Proceedings of the VLDB Endowment* 7, 13 (2014), 1379–1380.

[17] Shuo Han, Lei Zou, and Jeffrey Xu Yu. 2018. Speeding up set intersections in graph algorithms using simd instructions. In *Proceedings of the 2018 International Conference on Management of Data*. 1587–1602.

[18] Chiyu Hao, Jixian Su, Shixuan Sun, Hao Zhang, Sen Gao, Jianwen Zhao, Chenyi Zhang, Jieru Zhao, Chen Chen, and Minyi Guo. 2025. RapidStore: An Efficient Dynamic Graph Storage System for Concurrent Queries. arXiv:2507.00839 [cs.DB] https://arxiv.org/abs/2507.00839

[19] Chathura Kankanamge, Siddhartha Sahu, Amine Mhedhbi, Jeremy Chen, and Semih Salihoglu. 2017. Graphflow: An active graph database. In *Proceedings of the 2017 ACM International Conference on Management of Data*. 1695–1698.

[20] Pradeep Kumar and H Howie Huang. 2020. Graphone: A data store for real-time analytics on evolving graphs. *ACM Transactions on Storage (TOS)* 15, 4 (2020), 1–40.

[21] Per-Åke Larson, Spyros Blanas, Cristian Diaconu, Craig Freedman, Jignesh M Patel, and Mike Zwilling. 2011. High-Performance Concurrency Control Mechanisms for Main-Memory Databases. *Proceedings of the VLDB Endowment* 5, 4 (2011).

[22] Viktor Leis, Alfons Kemper, and Thomas Neumann. 2013. The adaptive radix tree: ARTful indexing for main-memory databases. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*. IEEE, 38–49.

[23] Youhuan Li, Lei Zou, M Tamer Özsu, and Dongyan Zhao. 2020. Space-Efficient Subgraph Search Over Streaming Graph With Timing Order Constraint. *IEEE Transactions on Knowledge and Data Engineering* 34, 9 (2020), 4453–4467.

[24] Hyeontaek Lim, Michael Kaminsky, and David G Andersen. 2017. Cicada: Dependably fast multi-core in-memory transactions. In *Proceedings of the 2017 ACM International Conference on Management of Data*. 21–35.

[25] Peter Macko, Virendra J Marathe, Daniel W Margo, and Margo I Seltzer. 2015. Llama: Efficient graph analytics using large multiversioned arrays. In *2015 IEEE 31st International Conference on Data Engineering*. IEEE, 363–374.

[26] Mugilan Mariappan and Keval Vora. 2019. Graphbolt: Dependency-driven synchronous processing of streaming graphs. In *Proceedings of the Fourteenth EuroSys Conference 2019*. 1–16.

[27] Robert Campbell McColl, David Ediger, Jason Poovey, Dan Campbell, and David A Bader. 2014. A performance evaluation of open source graph databases. In *Proceedings of the first workshop on Parallel programming for analytics applications*. 11–18.

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

[29] Prashant Pandey, Brian Wheatman, Helen Xu, and Aydin Buluc. 2021. Terrace: A hierarchical graph container for skewed dynamic graphs. In *Proceedings of the 2021 International Conference on Management of Data*. 1372–1385.

[30] Xiafei Qiu, Wubin Cen, Zhengping Qian, You Peng, Ying Zhang, Xuemin Lin, and Jingren Zhou. 2018. Real-time constrained cycle detection in large dynamic graphs. *Proceedings of the VLDB Endowment* 11, 12 (2018), 1876–1888.

[31] M Mazhar Rathore, Awais Ahmad, Anand Paul, and Gwanggil Jeon. 2015. Efficient graph-oriented smart transportation using internet of things generated big data. In *2015 11th International Conference on Signal-Image Technology & Internet-Based Systems (SITIS)*. IEEE, 512–519.

[32] Pedro Ribeiro and Fernando Silva. 2014. G-tries: a data structure for storing and finding subgraphs. *Data Mining and Knowledge Discovery* 28 (2014), 337–377.

[33] Siddhartha Sahu, Amine Mhedhbi, Semih Salihoglu, Jimmy Lin, and M Tamer Özsu. 2017. The ubiquity of large graphs and surprising challenges of graph processing. *Proceedings of the VLDB Endowment* 11, 4 (2017), 420–431.

[34] Aneesh Sharma, Jerry Jiang, Praveen Bommannavar, Brian Larson, and Jimmy Lin. 2016. GraphJet: Real-time content recommendations at Twitter. *Proceedings of the VLDB Endowment* 9, 13 (2016), 1281–1292.

[35] Sijie Shen, Zihang Yao, Lin Shi, Lei Wang, Longbin Lai, Qian Tao, Li Su, Rong Chen, Wenyuan Yu, Haibo Chen, et al. 2023. Bridging the Gap between Relational {OLTP} and Graph-based {OLAP}. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*. 181–196.

[36] Jifan Shi, Biao Wang, and Yun Xu. 2024. Spruce: a Fast yet Space-saving Structure for Dynamic Graph Storage. *Proceedings of the ACM on Management of Data* 2, 1 (2024), 1–26.

[37] Julian Shun and Guy E Blelloch. 2013. Ligra: a lightweight graph processing framework for shared memory. In *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming*. 135–146.

[38] Julian Shun, Laxman Dhulipala, and Guy E Blelloch. 2015. Smaller and faster: Parallel processing of compressed graphs with Ligra+. In *2015 Data Compression Conference*. IEEE, 403–412.

[39] Weiping Song, Zhiping Xiao, Yifan Wang, Laurent Charlin, Ming Zhang, and Jian Tang. 2019. Session-based social recommendation via dynamic graph attention networks. In *Proceedings of the Twelfth ACM international conference on web search and data mining*. 555–563.

[40] Yihan Sun, Daniel Ferizovic, and Guy E Belloch. 2018. PAM: parallel augmented maps. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 290–304.

[41] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. 2013. Speedy transactions in multicore in-memory databases. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. 18–32.

[42] Keval Vora, Rajiv Gupta, and Guoqing Xu. 2017. Kickstarter: Fast and accurate computations on streaming graphs via trimmed approximations. In *Proceedings of the twenty-second international conference on architectural support for programming languages and operating systems*. 237–251.

[43] Jianguo Wang, Chunbin Lin, Yannis Papakonstantinou, and Steven Swanson. 2017. An experimental study of bitmap compression vs. inverted list compression. In *Proceedings of the 2017 ACM International Conference on Management of Data*. 993–1008.

[44] Brian Wheatman and Helen Xu. 2018. Packed compressed sparse row: A dynamic graph representation. In *2018 IEEE High Performance extreme Computing Conference (HPEC)*. IEEE, 1–7.

[45] Boyu Yang, Weiguo Zheng, Xiang Lian, Yuzheng Cai, and X Sean Wang. 2024. HERO: A Hierarchical Set Partitioning and Join Framework for Speeding up the Set Intersection Over Graphs. *Proceedings of the ACM on Management of Data* 2, 1 (2024), 1–25.

[46] Tangwei Ying, Hanhua Chen, and Hai Jin. 2020. Pensieve: Skewness-aware version switching for efficient graph processing. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 699–713.

[47] Song Yu, Shufeng Gong, Qian Tao, Sijie Shen, Yanfeng Zhang, Wenyuan Yu, Pengxi Liu, Zhixin Zhang, Hongfu Li, Xiaojian Luo, et al. 2024. LSMGraph: A High-Performance Dynamic Graph Storage System with Multi-Level CSR. *Proceedings of the ACM on Management of Data* 2, 6 (2024), 1–28.

[48] Siyi Zhang, Xiaoxi Cui, Yurong Cheng, Ye Yuan, and Guoren Wang. 2022. Online Global Query Planning for Dynamic Road Networks. In *2022 IEEE 8th International Conference on Cloud Computing and Intelligent Systems (CCIS)*. IEEE, 666–670.

[49] Xiaowei Zhu, Guanyu Feng, Marco Serafini, Xiaosong Ma, Jiping Yu, Lei Xie, Ashraf Aboulnaga, and Wenguang Chen. 2019. Livegraph: A transactional graph storage system with purely sequential adjacency list scans. *arXiv preprint arXiv:1910.05773* (2019).