Revisiting the Design of In-Memory Dynamic Graph Storage

JIXIAN SU, Shanghai Jiao Tong University, China
CHIYU HAO, Shanghai Jiao Tong University, China
SHIXUAN SUN, Shanghai Jiao Tong University, China
HAO ZHANG, Huawei Cloud, China
SEN GAO, National University of Singapore, Singapore
JIAXIN JIANG, National University of Singapore, Singapore
YAO CHEN, National University of Singapore, Singapore
CHENYI ZHANG, Huawei Cloud, China
BINGSHENG HE, National University of Singapore, Singapore
MINYI GUO, Shanghai Jiao Tong University, China

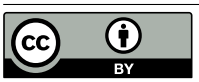
The effectiveness of in-memory dynamic graph storage (DGS) for supporting concurrent graph read and write queries is crucial for real-time graph analytics and updates. Various methods have been proposed, for example, LLAMA, Aspen, LiveGraph, Teseo, and Sortledton. These approaches differ significantly in their support for read and write operations, space overhead, and concurrency control. However, there has been no systematic study to explore the trade-offs among these dimensions. In this paper, we evaluate the effectiveness of individual techniques and identify the performance factors affecting these storage methods by proposing a common abstraction for DGS design and implementing a generic test framework based on this abstraction. Our findings highlight several key insights: 1) Existing DGS methods exhibit substantial space overhead. For example, Aspen consumes 3.3-10.8x more memory than CSR, while the optimal fine-grained methods consume 4.1-8.9x more memory than CSR, indicating a significant memory overhead. 2) Existing methods often overlook memory access impact of modern architectures, leading to performance degradation compared to continuous storage methods. 3) Fine-grained concurrency control methods, in particular, suffer from severe efficiency and space issues due to maintaining versions and performing checks for each neighbor. These methods also experience significant contention on high-degree vertices. Our systematic study reveals these performance bottlenecks and outlines future directions to improve DGS for real-time graph analytics.

CCS Concepts: • **Information systems → Graph-based database models**; *Data structures*; Storage management.

Additional Key Words and Phrases: dynamic graph storage; graph concurrency control; graph neighbor index; benchmark framework.

ACM Reference Format:
Jixian Su, Chiyu Hao, Shixuan Sun, Hao Zhang, Sen Gao, Jiaxin Jiang, Yao Chen, Chenyi Zhang, Bingsheng He, and Minyi Guo. 2025. Revisiting the Design of In-Memory Dynamic Graph Storage. *Proc. ACM Manag. Data* 3, 1 (SIGMOD), Article 70 (February 2025), 27 pages. https://doi.org/10.1145/3709720

Authors' Contact Information: Jixian Su, Shanghai Jiao Tong University, Shanghai, China, sjx13623816973@sjtu.edu.cn; Chiyu Hao, Shanghai Jiao Tong University, Shanghai, China, hcahoi11@sjtu.edu.cn; Shixuan Sun, Shanghai Jiao Tong University, Shanghai, China, sunshixuan@sjtu.edu.cn; Hao Zhang, Huawei Cloud, Beijing, China, zhanghao687@huawei.com; Sen Gao, National University of Singapore, Singapore, sen@u.nus.edu; Jiaxin Jiang, National University of Singapore, Singapore, jxjiang@nus.edu.sg; Yao Chen, National University of Singapore, Singapore, yaochen@nus.edu.sg; Chenyi Zhang, Huawei Cloud, Hangzhou, China, zhangchenyi@huawei.com; Bingsheng He, National University of Singapore, Singapore, hebs@comp.nus.edu.sg; Minyi Guo, Shanghai Jiao Tong University, Shanghai, China, guo-my@cs.sjtu.edu.cn.



ACM 2836-6573/2025/2-ART70
https://doi.org/10.1145/3709720

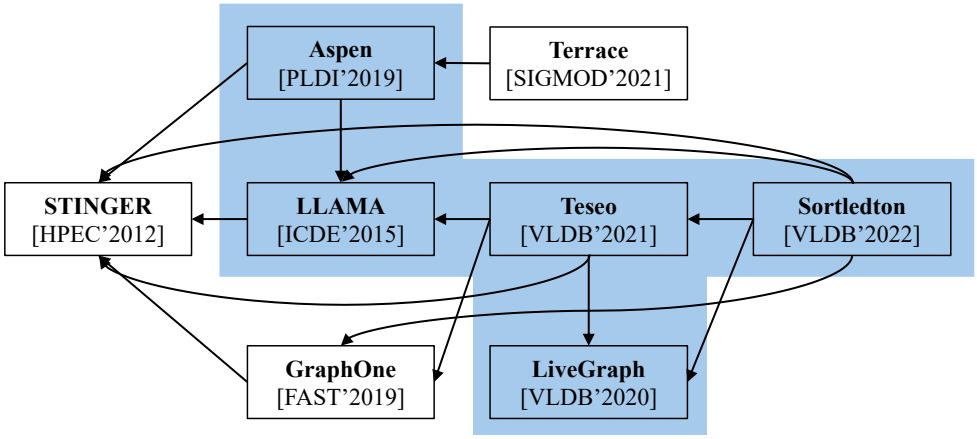


Fig. 1. Comparison of DGS methods from previous experiments. An edge from x to y indicates that x's experiments include y. Shaded methods are transactional approaches.

1 Introduction

Graph storage is the foundation for efficient in-memory graph data processing. As graph data often require frequent updates [45, 46], in-memory dynamic graph storage (DGS) is essential for supporting concurrent graph read and write queries, enabling real-time graph analytics and updates. To address this need, a variety of DGS methods have been proposed [12, 14, 15, 24, 30, 37, 42, 62], as shown in Figure 1. STINGER [15], GraphOne [30], and Terrace [42] do not provide concurrency control, meaning read and write queries can only execute alternately. In contrast, the other approaches support concurrent read and write queries with serializability [44], i.e., the results of executing concurrent queries are equivalent to some serial execution order. LLAMA [37] and Aspen [14] primarily focus on batch updates with a single-writer model, while recent methods have shifted toward optimizing individual updates to support a broader range of applications, which has attracted significant research interest [12, 24, 62]. Despite their varied update strategies, these approaches all enable concurrent read and write queries, thus called *transactional approaches* in this paper.

These approaches differ significantly in their support for read and write operations, space overhead, and concurrency control. First, they employ different graph container designs to support efficient read and write operations. For instance, LLAMA [37] and LiveGraph [62] store a vertex's neighbor set $N(u)$ in a dynamic array, enabling fast scan operations with continuous storage and simple append inserts, but at the cost of expensive search operations. Recent approaches like Aspen [14], Teseo [12], and Sortledton [24] use a segmented strategy, dividing $N(u)$ into small blocks (e.g., $|B| = 256$), storing each block as a sorted array, and linking them with a block index (e.g., a skip list). This design balances insert, scan, and search efficiency. Moreover, these approaches propose additional optimizations, such as adaptive indexing to use different data structures for varying neighbor set sizes to further accelerate graph operations.

Second, although existing methods all use multi-version concurrency control (MVCC) to maximize parallelism among queries [55], they differ in version granularity. *Coarse-grained* methods like LLAMA and Aspen use the typical single-writer-multiple-reader scheme with copy-on-write (CoW) [10]. For each update, they create a new graph snapshot, while read queries work on the snapshot at their start. Recent works such as LiveGraph, Teseo, and Sortledton use a *fine-grained* strategy, maintaining version information for each neighbor and synchronizing read and write

operations with concurrency control protocols like 2PL [44]. This allows multiple writers to update different elements simultaneously.

However, there has been no systematic study exploring the trade-offs among these dimensions. Particularly, we observe several issues in current DGS research. First, there is a lack of a common abstraction to model the problem and capture the design space. Second, although Sortledton [24] covers most existing methods in the experiments, as shown in Figure 1, existing methods compare different approaches as black boxes, while numerous factors can affect performance, making the effectiveness of individual techniques unclear. Additionally, existing experiments focus on evaluating the efficiency of proposed graph containers but lack assessments of concurrency performance. Third, the performance gap between DGS and CSR regarding read efficiency and space consumption is unclear, obscuring the overhead of using DGS for read queries.

Our Work. We propose to revisit the design of in-memory dynamic graph storage to evaluate the effectiveness of individual techniques and identify key performance factors. We first propose a common abstraction for DGS to capture the nature of graph queries and data, highlighting key performance factors and facilitating a systematic study of existing methods. Within this common abstraction, we compare key techniques in existing DGS methods, including graph containers, concurrency control, and specific optimizations. Additionally, we implement a generic test framework based on our abstraction and re-implement these techniques within the framework for fair empirical comparisons.

We evaluate five DGS methods and include two static graph storage methods, CSR and AdjLst, for comparison. We conduct detailed experiments with eight real-world graphs to assess graph container efficiency, the effectiveness of concurrency control techniques, and memory consumption across methods. Our results show that, while fine-grained methods improve write throughput, they also incur significant space and efficiency overhead. Fine-grained strategies face challenges with version maintenance and lock contention, especially when managing high-degree vertices. Although coarse-grained methods alleviate these issues, Aspen still suffers from inefficiencies in graph container design. Consequently, CSR consistently outperforms DGS methods, achieving 2.4–11.0x higher query performance and consuming 3.3–10.8x less memory. We summarize the relative performance of competing methods, key insights, and research opportunities in Section 8. This work is open-sourced on GitHub at https://github.com/SJTU-Liquid/DynamicGraphStorage.git. In summary, this paper makes the following contributions.

- We propose a simple yet effective abstraction for DGS, enabling a systematic study of existing methods.
- Using this abstraction, we compare key techniques in DGS, e.g., graph containers and concurrency control.
- We develop a generic testing framework based on this abstraction to systematically evaluate existing methods.
- Our findings reveal the strengths of current approaches and highlight critical performance factors, providing valuable insights to guide the design of future DGS systems.

2 Preliminaries

We focus on the directed graph $G = (V, E)$, where V is the set of vertices and $E \subseteq V \times V$ is the set of edges. For a vertex $u \in V$, $N(u)$ denotes its neighbors and $d(u)$ is its degree, i.e., $|N(u)|$. An edge $e(u, v)$ is directed from u to v. An undirected graph can be represented by storing edges in both directions, $e(u, v)$ and $e(v, u)$. Each vertex has a unique *vertex ID*. Previous studies [1, 25, 27, 54, 56] have shown that using vertex IDs as integers in $[0, |V|)$ benefits both computation and storage due to compact ID representation. Existing works [12, 14, 24, 37, 62] either require input vertex IDs

Table 1. The time and space complexity of involved data structures in our study.

	Time				Space
	Insert	Search	Scan	Resize	
Dynamic Array(DA)	$O(1)$	$O(n)$	$O(n)$	$\Theta(n)$	$O(n)$
Sorted Dynamic Array(SDA)	$O(n)$	$O(\log n)$	$O(n)$	$\Theta(n)$	$O(n)$
Packed Memory Array(PMA)	$O(\log^2 n)$	$O(\log n)$	$O(n)$	$\Theta(n)$	$O(n)$
Skip List(SL)	$O(\log n)$	$O(\log n)$	$O(n)$	-	$O(n \log n)$
Hash Table(HT)	$O(1)$	$O(1)$	$O(n)$	-	$O(n)$
AVL Tree(AVL)	$O(\log n)$	$O(\log n)$	$O(n)$	-	$O(n)$
Parallel Augmented Map(PAM)	$O(\log n)$	$O(\log n)$	$O(n)$	-	$O(n)$

to belong to $[0, |V|)$ or use dictionary encoding techniques (e.g., concurrent hash maps) to map external vertex IDs to this range. Thus, in our case, each vertex ID $u \in V$ is an integer in $[0, |V|)$.

Graph Storage. A *static graph* can be stored as an *adjacency list* (AdjLst), which uses an array of arrays (or lists), where each array at index u contains all of u's neighbors, sorted by vertex IDs. A common variant, the *compressed sparse row* (CSR) format, stores all vertices' neighbor sets in a single *neighbor array* and uses an *offset array* to record the start position of each vertex's neighbor set in the neighbor array. Given that real-world graphs are often sparse, CSR is the most widely used storage method for static graphs due to its compact memory layout and data access efficiency. A *dynamic graph* $G = (G_0, \Delta\mathcal{G})$ records the evolution of the graph over time. G_0 is the initial graph and $\Delta\mathcal{G} = (\Delta G_1, ..., \Delta G_i, ...)$ is a sequence of updates. $\Delta G_i = (\oplus, v/e)$ contains a set of updates, where $(\oplus = +/-)$ denotes an insertion/deletion of a vertex v or an edge $e(u, v)$. Deleting a vertex also removes all its adjacent edges. ΔV denotes the vertices appeared in ΔG. The graph G_i is obtained by applying the first i updates to G_0, i.e., $G_i = G_0 \oplus \Delta G_1 \oplus ... \oplus \Delta G_i$. If ΔG contains a single operation, it is called a *single update*; otherwise, it is a *batch update*. Static graph formats are inefficient for dynamic graphs because: 1) they cannot coordinate concurrent read and write operations, compromising the correctness of graph queries; and 2) they do not support fast updates.

Data Structures for Vertex Sets. In principle, both $V(G)$ and $N(u)$ (i.e., a subset of $V(G)$) are sets of integers. Various data structures can be used to maintain them. Table 1 presents the time and space complexity of these data structures involved in this paper.

DA is a resizable array with append to perform insert. In contrast, SDA is a resizable sorted array that uses binary search to locate elements and insertion to keep the array sorted. PMA [4, 11] maintains elements in a partially filled array organized into blocks, facilitating efficient insertions while keeping elements sorted. To maintain balance, PMAs periodically rebalance elements within blocks to ensure even distribution and optimal space usage. Both SDA and PMA extend their capacity by allocating a larger block of memory, typically doubling the current size and copying the elements to the new block. The resize cost for n elements is $\Theta(n)$. PAM [50] is a data structure that supports parallel operations on ordered maps. It uses balanced binary trees and join-based algorithms to maintain order and balance efficiently. As SL [43], HT, and AVL are widely used, we omit the details for brevity.

Graph Queries. The graph community [2, 8] categorizes query workloads into three classes: *interactive workloads* (IC) [16], *analytics workloads* (AS) [28], and *business intelligence workloads* (BI) [51]. IC includes queries that involve simple vertex/edge insertion/deletion operations or start from specified vertices and access a small portion of the graph. AS consists of graph traversal algorithms like PageRank, which typically visit the entire graph. BI bridges the gap between IC and AS by introducing pattern queries. IC is referred to as OLTP, while AS and BI are considered OLAP [7, 35].

In the dynamic graph model, these queries are categorized into *read queries* Q, which are strictly read-only, and *write queries* ΔG, which include update operations [24, 35]. Typically, graph queries are read-intensive, and both read and write queries exhibit distinct characteristics:

(1) Read queries can be complex and long-running, with an unpredictable *read set* (i.e., the vertices and edges accessed).

(2) Write queries are lightweight, involving a small, known *write set* of vertex/edge insertion or deletion operations.
(3) As graphs are widely used for connection analysis, scan operations are a common and important graph data access pattern [12, 14, 15, 24, 30, 37, 42, 62].

MVCC. Each query can be abstracted as a transaction with read and write operations on a database. MVCC is currently the most popular transaction management scheme [55]. The basic idea is to maintain multiple physical versions of each logical object to allow operations on the same object in parallel to maximize parallelism of concurrent queries without sacrificing serializability.

Strict two-phase locking (S2PL) [6] is a concurrency control protocol that operates in two phases: 1) the growing phase, where locks can be acquired but not released; and 2) the shrinking phase, where no new locks can be acquired after the first lock is released. It holds exclusive locks until the transaction commits or aborts. To avoid deadlocks, an effective approach called the no-wait policy [5] is to immediately abort a transaction if it cannot acquire a lock [57]. S2PL is a classical approach to achieving serializability.

Unlike pessimistic protocols (e.g., S2PL) that assume queries will conflict and take preventive actions, *optimistic concurrency control* (OCC) [31, 41] assumes conflicts are rare and handles them at transaction commit. OCC typically has three phases: 1) in the read phase, the transaction performs read and write operations on local copies of the data; 2) in the validation phase, upon committing, OCC checks if the data read by the transaction has been modified by other transactions; and 3) in the write phase, if validation passes, the transaction's updates are applied to the database; otherwise, the transaction is aborted and its changes are discarded. To support validation and undo operations, OCC records accessed elements in a log. OCC does not require locking elements during the read and write phases but does track accessed elements to ensure consistency. For details of MVCC, please refer to this study [55].

Lock and Latch. Locks are synchronization mechanisms used to regulate access to shared data objects during concurrent transaction processing. A DBMS employs a lock manager to handle lock requests and maintain the locks acquired by each transaction. There are two types of locks: shared locks, which allow reads from multiple transactions, and exclusive locks, which permit writes exclusively. Latches protect concurrent access to in-memory data structures by threads. A read latch allows multiple threads to access shared items simultaneously, whereas a write latch permits only one thread to access shared items at a time. Existing DGS methods [12, 14, 24, 37, 62] do not use a lock manager but instead rely on the locking mechanisms (e.g., mutex) provided by the OS to synchronize queries. Throughout the paper, we use the term "lock" without further distinction.

3 A Common Abstraction for DGS

In this section, we propose a common abstraction for DGS to facilitate a systematical study of existing methods.

3.1 Graph Query and Data Abstraction

In transaction management [44], a database is a set $\{x\}$ of tuples in tables. A transaction consists of a sequence of read and write operations on $\{x\}$, beginning with a Begin command and ending with either Commit, indicating successful execution, or Abort, indicating failure and reverting modifications. Building on this concept, we propose a simple yet effective multi-level abstraction for graph query and data that aims to: 1) reflect the characteristics of graph queries; 2) indicate the nature of graph data; and 3) capture graph data access patterns. Figure 2 shows our abstraction.

Global Abstraction. Graph queries are categorized into write queries ΔG and read queries Q as discussed in Section 2. The global abstraction models the relationship among these queries by maintaining a global timestamp $t(G)$, initialized to 0 and incremented by 1 only when ΔG is committed. To ensure the serializability of graph queries, DGS requires that each committed write query be uniquely identified by its commit timestamp, denoted as ΔG_{i+1} for the write query committed at $t(G) = i$. This abstraction effectively captures the construction of a dynamic graph $G = (G_0, \Delta\mathcal{G})$, where $\Delta\mathcal{G}$ is the serial execution order of committed write queries. A read query Q starting at $t(G) = i$ has a local timestamp $t(Q)$.

Local Abstraction. In cooperation with the global abstraction, the local abstraction allows us to drill down to each data object and their primitive operations. The graph G consists of vertices and edges. To indicate their

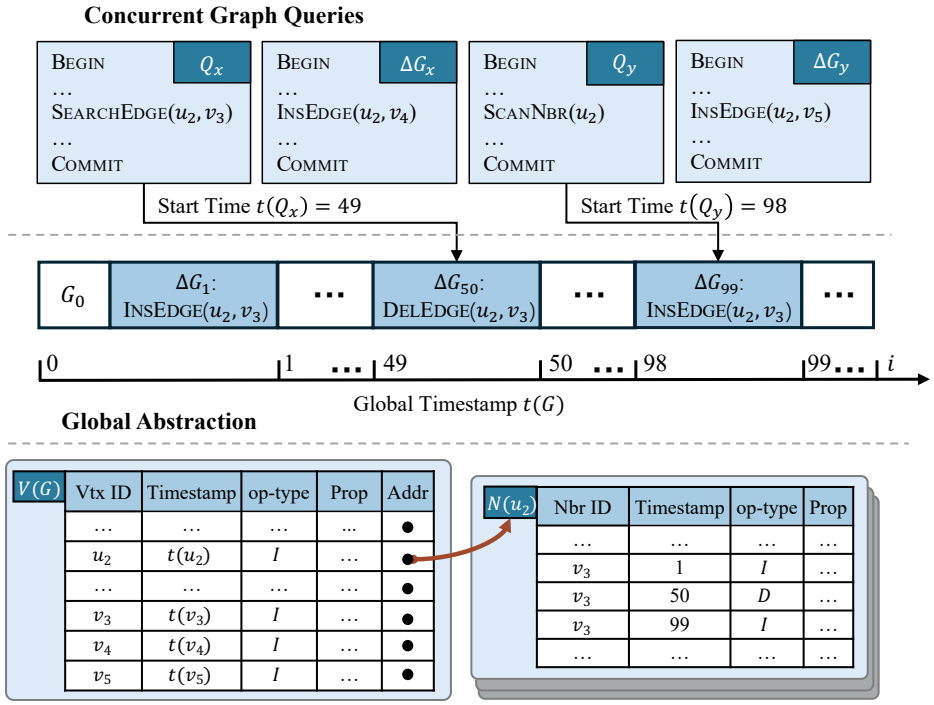


Fig. 2. The abstraction of graph query and data.

differences and interconnections, G is organized into a vertex table containing $V(G)$ and a set of neighbor tables, each corresponding to a neighbor set $N(u)$. Each entry in $V(G)$ contains the vertex ID u, the location of $N(u)$, and its properties, while each entry in $N(u)$ contains the neighbor ID v and the properties associated with edge $e(u, v)$. Given a write query ΔG_i, each operation creates a new version of a vertex or neighbor u with the timestamp $t(u) = i$. Specifically, an insert (resp. delete) operation on u creates a new version with the OP-TYPE as I (resp. D). Updating an element is performed via an insert operation.

Lemma 3.1 can be proven using the dependency graph [19] within the multi-level abstraction. The lemma shows that by maintaining the serializability of write queries, DGS can achieve serializable isolation by ensuring Q has a consistent view of G_i. This is done by allowing Q to access only the latest version of vertices or neighbors u such that $t(u) \leqslant t(Q)$. This approach allows us to coordinate Q's data access based on timestamps without complex concurrency control protocols for read queries. Although existing DGS methods use this optimization, they do not explicitly and formally discuss it.

LEMMA 3.1. *Suppose DGS maintains the serializability of write queries with the serial execution order $\Delta\mathcal{G}$. Given a read query Q starting at timestamp i, ensuring Q has a consistent view of $G_i = G_0 \oplus ... \oplus \Delta G_i$ guarantees global serializable isolation for read and write queries.*

Proof. In the dependency graph, each node Q_x represents a query, and an edge from Q_x to Q_y indicates that an operation $o_x \in Q_x$ conflicts with $o_y \in Q_y$, and o_x precedes o_y in execution. Operations o_x and o_y conflict if they act on the same object and at least one is a write. Queries are conflict serializable *iff* the dependency graph is acyclic. Since DGS maintains the serializability of write queries, the dependency graph formed by them is acyclic. Ensuring Q has a consistent view of G_i guarantees that all operations in Q depend only on operations from preceding write queries. Thus, the node representing Q in the dependency graph has no incoming edges. Therefore, the combined dependency graph of read and write queries remains acyclic, proving the queries are conflict serializable.

3.2 Graph Operations Abstraction

We next analyze graph data access patterns based on the multi-level abstraction. From the perspective of DGS, a write (or read) query consists of a sequence of write (or read) operations on vertices and edges, while the computation logic and state of the query are beyond the scope of DGS. These *graph operations* include:

- InsVtx(u): inserting a vertex u into $V(G)$;
- InsEdge(u, v): inserting an edge $e(u, v)$ into $E(G)$;
- SearchVtx(u): finding a vertex u in $V(G)$;
- SearchEdge(u, v): finding an edge $e(u, v)$ in $E(G)$;
- ScanVtx(G): traversing the vertex set $V(G)$;
- ScanNbr(u): traversing the neighbor set $N(u)$.

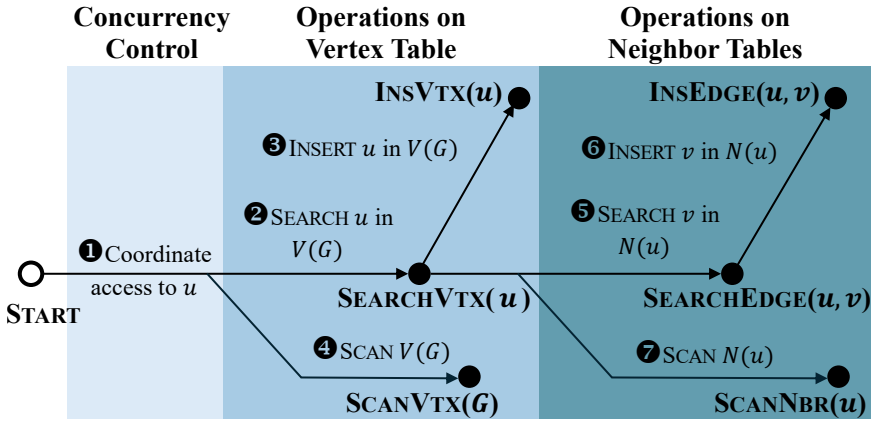


Fig. 3. The abstraction of graph operations.

Figure 3 presents the abstraction of graph operations, where each node represents a graph operation, and each edge denotes a primitive operator on vertex or neighbor tables. The path from the start node to an operation node illustrates the primitive operators required to perform the operation and shows the relationships between different operations (e.g., InsEdge invokes SearchEdge). In summary, this abstraction provides a unified execution routine on graph operations.

Let P_V and P_N denote the paths from the start node to the graph operation node in the vertex table and neighbor table operations, respectively, in Figure 3. Equation 1 defines the cost T of a graph operation, where T_{CC} is the cost of coordinating access to the target vertex, and T_p is the cost of the primitive operator p. Concurrency control requires the cooperation of underlying graph containers, such as checking a creation timestamp for each object. α_p is the overhead amplification ratio of concurrency control on the primitive operator p. An optimal value is 1, indicating no performance degradation due to concurrency control. This equation highlights the factors affecting DGS performance, serving as a tool to systematically study these performance factors rather than predicting the exact cost of a graph operation.

$$T = T_{CC} + \sum_{p \in P_V} \alpha_p T_p + \sum_{p \in P_N} \alpha_p T_p. \tag{1}$$

Remark. We exclude edge update and delete operations as they follow a similar process to insertions. Fine-grained methods like Sortledton, Teseo, and LiveGraph handle deletions by: 1) locating the target vertex, and 2) creating a new version marked as *delete* or updating the end timestamp to indicate the deletion, as discussed above. Space is later reclaimed through garbage collection. Coarse-grained methods like Aspen, use a copy-on-write strategy, where deletion involves removing a vertex from a block rather than adding one, much like with insertions. Graph query workloads generally focus on analyzing connections between vertices, with vertex updates, especially deletions, being rare [62]. Consequently, existing DGS works typically focus on operations on neighbor tables. Therefore, this paper primarily focuses on $N(u)$ as well.

Table 2. A summary of DGS methods under study.

	Graph Concurrency Control		Graph Container		
Method	**Version Management**	**Protocol**	**Vertex Index**	**Neighbor Index**	**Additional Optimization**
LiveGraph	Fine-Grained with Continuous Version	S2PL	Dynamic Array	Dynamic Array	Bloom Filter
Sortledton	Fine-Grained with Version Chain	G2PL	Dynamic Array	Segmented Skip List	Adaptive Indexing
Teseo	Fine-Grained with Version Chain	OCC	Hash Table	PMA	Write-Optimized Segment
Aspen	Coarse-Grained	Single Writer	AVL Tree	Segmented PAM	Vertex Index Flatten & Data Encoding
LLAMA	Coarse-Grained	Single Writer	Dynamic Array	Dynamic Array	-

4 DGS Methods Under Study

Existing DGS methods focus on optimizing two problems: 1) graph concurrency control, which includes version management and concurrency control protocols to coordinate the execution of concurrent graph queries, and minimizing overhead for each graph operation (i.e., T_{CC} and α_p in Equation 1); and 2) graph containers, which include vertex indexes, neighbor indexes, and other optimizations to optimize graph data access T_p for each operation in Equation 1. Table 2 summarizes the key design choices in these methods. In the following, we first briefly introduce these methods and then compare them.

4.1 A Brief Introduction to DGS Methods

4.1.1 ***LiveGraph*** *[62].* **Graph Concurrency Control.** LiveGraph uses a lock for each vertex in $V(G)$ and adapts S2PL to synchronize data access. For a write query ΔG, LiveGraph first obtains all exclusive locks on vertices in ΔV (vertices involved in ΔG), performs the graph operations, and then releases the locks. To handle deadlocks, LiveGraph aborts ΔG if it cannot acquire a lock within a time limit.

For each version of a neighbor or vertex, LiveGraph keeps begin and end timestamps ($begin-ts$ and $end-ts$) to record its lifetime as shown in Figure 4. Given ΔG committed at timestamp i, InsEdge(u, v) searches for the latest version of v in $N(u)$. If found, it sets $end-ts$ to i and creates a new version of v with $begin-ts = i$ and $end-ts = INF$. Otherwise, it directly creates a new version of v. DelEdge(u, v) searches for the latest version of v and sets $end-ts$ to i. For read operations in Q, LiveGraph obtains a shared lock on the vertex and immediately releases it after the operation. For example, ScanNbr(u) acquires the lock on u, accesses neighbors based on timestamps, and releases the lock immediately. Thus, Q never leads to deadlocks because it never holds two locks simultaneously.

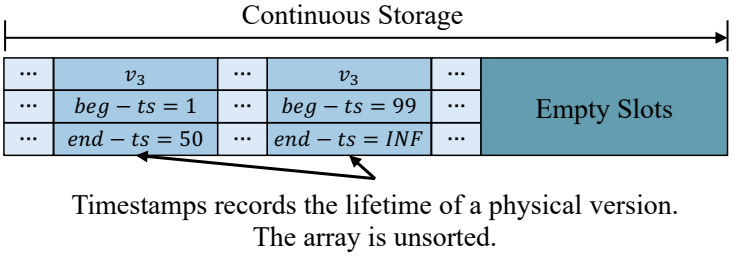


Fig. 4. The neighbor index of $N(u_2)$ in LiveGraph.

Graph Container. Given $u \in V(G)$, LiveGraph uses a dynamic array (DA) as the neighbor index of $N(u)$ where each element corresponds to a physical version of $v \in N(u)$. Graph operations in Figure 3 are based on primitive operators of DA, the time complexity of which is listed in Table 1. As the storage of DA is continuous and no version chain exists, Scan is very fast. Moreover, LiveGraph executes Scan from the end to the beginning of DA since the latest element may be more frequently visited than the stale ones. However,

SEARCH is slow because DA is unsorted and uses SCAN to perform the search. Consequently, INSEDGE is slow because it depends on SEARCHEDGE though adding an element only requires a simple append. To mitigate this issue, LiveGraph maintains a Bloom filter [40] for each $N(u)$ to record whether an element exists in $N(u)$. LiveGraph uses DA as the vertex index of $V(G)$ as well. As the vertex ID is ranged in $[0, |V|)$, the element at the index u is the vertex u. Therefore, the time complexity of SEARCH on the vertex index is $O(1)$. As the implementation is simple, we omit the details.

4.1.2 ***Sortledton*** *[24].* **Graph Concurrency Control.** Sortledton also uses S2PL but optimizes the locking sequence: Sort the vertices in ΔV by ascending vertex IDs and acquire their exclusive locks in that order. This optimization prevents deadlocks by avoiding circular waiting among write queries [49]. Therefore, Sortledton does not need any mechanisms to handle deadlocks. For INSEDGE(u, v) (or DELEDGE(u, v)) in ΔG committed at timestamp i, Sortledton creates a new version of v with timestamp i and OP-TYPE as I (or D) as shown in Figure 5. Sortledton maintains a version chain for different versions of v, where the new version points to the old one. For read queries, Sortledton uses the same concurrency control strategy as LiveGraph.

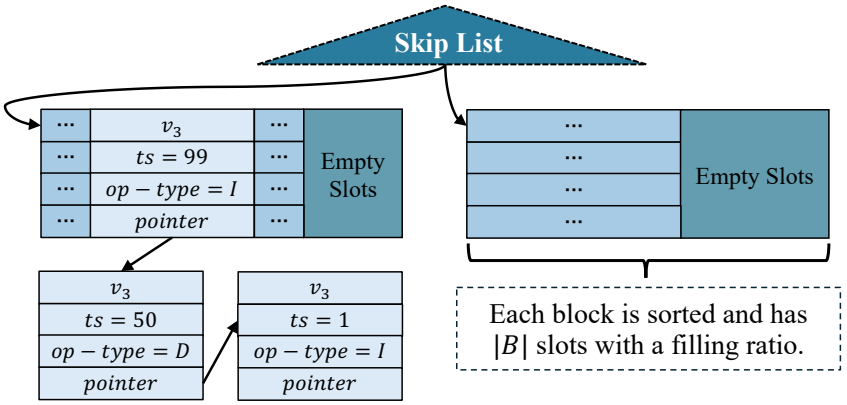


Fig. 5. The neighbor index of $N(u_2)$ in Sortledton.

Graph Container. Like LiveGraph, Sortledton uses a dynamic array as the vertex index. For the neighbor index, Sortledton splits $N(u)$ into blocks B, and uses the skip list as the block index linking them. The fill ratio of each block is maintained between 50% and 100%. When a block is full, Sortledton splits it into two blocks, equally distributing the neighbors. If the fill ratio drops below 50%, Sortledton merges it with adjacent blocks. The first element of each block serves as its key in the skip list. We call this structure the *segmented skip list*. A read operation traverses the version chain to find the target version. Since real-world graphs are sparse, Sortledton proposes the *adaptive neighbor index*: If $|N(u)|$ is below a threshold (e.g., 256), it uses a sorted dynamic array as the neighbor index instead of the segmented skip list.

4.1.3 ***Teseo*** *[12].* **Graph Concurrency Control.** Teseo uses the same version management method as Sortledton but adopts optimistic concurrency control (OCC introduced in Section 2) to coordinate concurrent write queries. Unlike LiveGraph and Sortledton, which keep a lock for each vertex, Teseo maintains a lock for each edge partition to synchronize concurrent data access. Specifically, Teseo logically divides $E(G)$ into equally sized partitions, each with a lock. Before accessing data in a partition, a write query acquires an exclusive lock (or a read query acquires a shared lock) on the partition and releases it immediately after access. **Graph Container.** Teseo employs the packed memory array (PMA) [4, 11] as the neighbor index as shown in Figure 6. But it stores all neighbor tables in a single PMA. If $N(u)$ is large, it spans multiple blocks in the PMA, while multiple small neighbor sets can share a single block. However, the global rebalance overhead is high for a PMA if $E(G)$ is stored together. To address this, Teseo divides the single PMA into multiple large leaves (several megabytes each) and indexes these leaves with an *adaptive radix tree* (ART) [32], calling this data structure FAT. Additionally, Teseo uses a hash table as the vertex index to record the position of each vertex's neighbor index in FAT. By default, blocks in FAT are sorted, known as read-optimized segments. When the insertion rate is high, FAT switches to write-optimized segments (WOS), which handle updates by appending

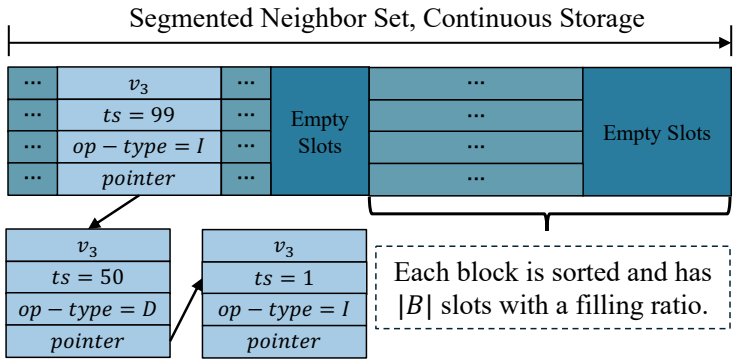


Fig. 6. The neighbor index of $N(u_2)$ in Teseo.

to the update log. For WOS, Teseo loops over the segment to find the target vertex. Teseo is built on top of HyPer [29].

4.1.4 ***Aspen*** *[14].* **Graph Concurrency Control.** Aspen uses a coarse-grained strategy that maintains timestamps for each graph snapshot. Specifically, Aspen uses the *single-writer-multiple-reader* scheme, which executes write queries serially and allows multiple readers to execute concurrently. A write query ΔG_i uses the *copy-on-write* (CoW) method (also called shadow paging) to apply updates on a copy of the graph, creating a new snapshot G_i as shown in Figure 7. Q works on the latest version of the graph snapshot. Therefore, read and write queries never block each other, and multiple read queries can share the same graph snapshot.

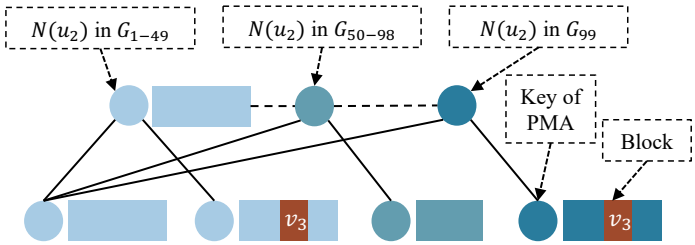


Fig. 7. The neighbor index of $N(u_2)$ in Aspen.

Graph Container. Aspen partitions $N(u)$ into a set of sorted blocks and uses a parallel augmented map (PAM) to index these blocks. These blocks have no empty slots. InsEdge(u, v) copies the block, inserts v, and then copies the path from the block to the root to create a new snapshot of $N(v)$. Given $N(u)$ and block size B, Aspen selects vertices such that $v \mod b = 0$ as heads, i.e., $heads = v \in N(u) \mid v \mod b = 0$. This approach ensures that updates to one block do not affect adjacent blocks. Aspen demonstrates that this segmentation ensures each block has B elements with high probability. Moreover, Aspen uses an AVL tree as the vertex index and copies the path in the tree for each update operation.

Aspen proposes two optimization methods to enhance performance. First, for long-running read queries, Aspen can create an array storing the positions of each neighbor table based on the AVL tree to eliminate the overhead of Search in the AVL tree. Second, Aspen uses a difference encoding scheme to compress data in a block. For a block containing $(v0, v1, v2, \ldots)$, Aspen stores it as $(v0, v1 - v0, v2 - v0, \ldots)$ and compresses it with byte codes to accelerate set intersections [1].

4.1.5 ***LLAMA*** *[37].* LLAMA, proposed in 2015, also employs a coarse-grained graph concurrency control mechanism similar to that in Aspen. Specifically, LLAMA divides the vertex table into partitions, each stored in a data page, and maintains an *indirection table* to store the locations of these pages. Each write query must copy the indirection table to create a new graph snapshot, and this overhead limits update performance and graph scalability.

4.2 Comparison of DGS Methods

Graph Containers. We discuss the time and space cost of graph operations based on Table 1.

Time. Given vertex IDs in the range $[0, |V|)$, using a dynamic array (DA) as vertex indexes allows SearchVtx and InsVtx in $O(1)$ time with simple memory accesses. Due to continuous storage, DA enables fast scan operations. In contrast, hash tables and AVL trees incur more overhead than DA, despite having the same time complexity for some operations.

Continuous storage, used in LiveGraph and LLAMA, stores $N(u)$ in DA, facilitating fast ScanNbr but resulting in slow SearchEdge due to the unsorted nature of the array. Since InsEdge depends on SearchEdge in DGS, its performance is also slow despite Insert in DA taking $O(1)$ time. For the segmented methods, scanning accesses data continuously within a block, while inserting typically moves only a few elements within a block. These blocks are linked by an index (e.g., PAM) to accelerate SearchEdge. Therefore, the cost of these operations includes the block index and the block itself. Increasing block sizes can improve scan performance due to continuous memory access but may degrade insert performance due to data movement within the block. Additionally, Aspen, using CoW, incurs more overhead for insertion than methods performing in-place updates because its insert operation copies the block as well as the root-to-leaf path.

Space. Practical memory consumption is significantly affected by element size. Each element in Sortledton and Teseo consumes $3 \times w$ bytes, where w is the word size: one for the vertex ID, one for the version, and one for the pointer. The op-type can use the high bit in the timestamp. Each element in LiveGraph also takes $3 \times w$ bytes. In contrast, each element in Aspen consumes only w bytes due to its coarse-grained granularity. Additionally, Aspen's neighbor index has no empty slots. Therefore, the coarse-grained method is more memory efficient than the fine-grained method.

Graph Concurrency Control. We first compare fine-grained and coarse-grained strategies, and then discuss fine-grained methods.

Fine-Grained vs. Coarse-Grained. Fine-grained methods require lock operations on each graph operation, which can lead to lock contention and thus expensive T_{CC}. High-degree vertices are particularly prone to frequent access. Fine-grained methods necessitate version checks on each element, resulting in increased data loading from memory and more computation for version comparison, leading to a high α_p value. In contrast, coarse-grained methods avoid these issues. However, fine-grained methods allow multiple writers to update the graph simultaneously and perform in-place updates by simply inserting new elements, enhancing update performance.

Additionally, the fine-grained strategy does not place special requirements on the underlying graph containers, making it more generic. In contrast, the coarse-grained strategy requires support for fast snapshot creation. However, since the coarse-grained strategy does not maintain versions or perform version checks for each element, it can be effectively combined with data compression techniques [14], which is not feasible for the fine-grained approach.

Discussion on Fine-Grained Strategies. First, the continuous version storage in LiveGraph can improve scanning efficiency by avoiding the traversal of a version chain. However, it may increase data access volume by including stale versions, negatively impacting search and insert efficiency. Second, G2PL in Sortledton is generally the optimal fine-grained concurrency control mechanism due to its effective deadlock avoidance optimization. Although OCC in Teseo does not require holding all mutexes of vertices in ΔV, write queries are typically very short because ΔG generally contains a single update. Moreover, executing deadlock detection for write-write conflicts introduces significant overhead and implementation challenges. As such, our study uses G2PL for fine-grained methods because of its advantages.

Empirical Evaluation Targets. Following the above discussion, we will set up test frameworks and empirically evaluate these techniques by addressing the following five questions.

- **Graph Containers:** ***Q1.*** *How effective are existing techniques in graph containers at efficiently performing key operations such as SearchEdge, InsEdge, and ScanNbr, as defined by T_p in Equation 1?* ***Q2.*** *Which neighbor index performs the best, and what is the gap between it and CSR on read queries?*
- **Graph Concurrency Control:** ***Q3.*** *What is the impact of graph concurrency control on graph operations?* ***Q4.*** *How are the scalability and concurrency of competing methods?*
- **Batch Granularity:** ***Q5.*** *How does the batch granularity affect the performance of competing methods?*
- **Memory Consumption:** ***Q6.*** *What is the impact of graph containers and version management in DGS on memory consumption, and what is the gap between DGS and CSR?*

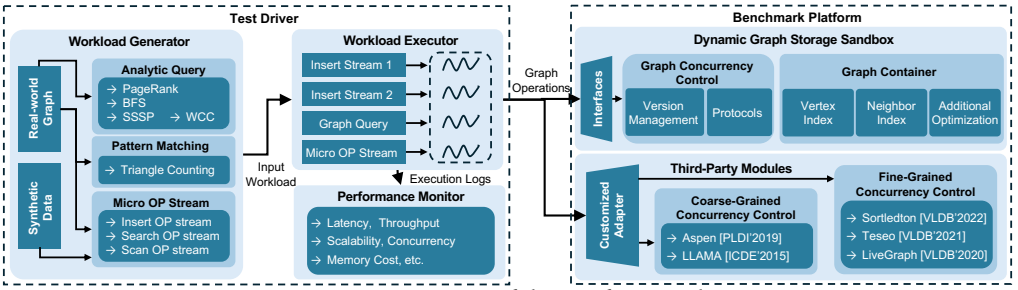


Fig. 8. An overview of the test framework.

5 Test Framework Setup

Figure 8 provides an overview of this framework. It is implemented with around 7800 lines of optimized C++ code. The source code is compiled using g++ 10.5.0 with the -O3 optimization enabled. Experiments are conducted on a Linux server equipped with an AMD EPYC 7543 CPU (32 cores) and 128GB of memory.

5.1 Benchmark Platform

Third-Party Modules. This module integrates the original implementations of LLAMA [36], Aspen [13], LiveGraph [20], Teseo [33], and Sortledton [23] from GitHub. Due to differences in their APIs, we implemented a customized adapter, a wrapper for each method, to standardize the evaluation of graph operations. The overhead of these adapters is negligible because they simply combine the interfaces to provide these functions, if not directly available. Each method is configured with its recommended settings. Specifically, the size of bloom filter in LiveGraph is set as $\frac{1}{16}$ of the block size. Sortledton, Teseo and Aspen set the block size to 256.

DGS Sandbox. In this module, we re-implement key techniques from competing methods within our abstraction for a fair and detailed investigation of individual techniques. These techniques can be composed differently to evaluate graph operations based on the unified execution routines shown in Figure 3. For fine-grained methods, we apply the following optimizations: 1) all methods use G2PL as the concurrency control protocol, and 2) all methods use a dynamic array as the vertex index by default. We implement a simple baseline DGS method by combining the static graph storage AdjLst with G2PL, naming it AdjLst. Specifically, AdjLst is implemented as an array where each element represents a vertex, and each vertex points to an array storing its neighbor set. When inserting a new element, a binary search is performed to find the correct position, after which the subsequent elements are shifted to make space for the new one. If the array is full, pre-allocated space is used, functioning like a dynamic array. For comparison purposes, we also include CSR, the optimal baseline for static graphs.

5.2 Test Driver

Workload Generator. Table 3 presents the statistics of the real-world graphs used in the paper. They span six categories, with $|V|$ ranging from tens of thousands to tens of millions and $|E|$ scaling to hundreds of millions. Both *ldbc* [17] and *nft* [59, 60] originally have timestamps to mark the insertion sequence of edges. *ldbc* simulates actions in a social network. We set the scale factor to 10 to control the graph size. *nft* records NFT transactions on Ethereum from 2017 to 2022. Other graphs are obtained from SNAP [34] and do not include timestamps. These graphs are widely used in DGS research. We also considered larger graphs (e.g., *friendster*) containing billions of edges but omitted them since existing DGS methods frequently run out of memory on these cases.

We generate three types of graph queries. First, the *micro OP stream* contains a sequence of graph operations. For graphs with timestamps, we generate an InsEdge stream using the first 80% of edges as the initial graph and the remaining 20% as the insert edges. For graphs without timestamps, we shuffle the edges and generate the insert stream similarly, following previous works [12, 24, 62]. As these works focus on single updates, each operation corresponds to a write query ΔG. For the SearchEdge stream, we randomly select 20% of edges as the search targets. For the ScanNbr stream, we select 20% of vertices based on their degrees. Each of these operations is a read query. The micro OP stream serves two purposes: 1) Investigating the performance

Table 3. The detailed information of the real-world graphs.

Category	Dataset	Name	$\|V\|$	$\|E\|$	d_{avg}	d_{max}
Social	LiveJournal	*lj*	4.8M	42.8M	8.8	20233
	LDBC	*ldbc*	30.0M	175.9M	5.9	4282595
	Twitter	*tw*	21.3M	265.0M	12.4	698112
Game	DotaLeague	*dl*	0.06M	50.9M	831.6	17004
Web	Wiki	*wk*	14.0M	59.0M	4.2	723404
Citation	Cit	*ct*	3.8M	16.5M	4.4	793
Synthetic	Graph500	*g5*	8.9M	260.4M	29.3	406416
Financial	NFT	*nft*	29.6M	77.5M	2.62	2290853

of competing methods on basic graph operations, and 2) Studying the effectiveness of short queries (i.e., IC workloads).

Second, we integrate four representative *graph analytic* algorithms from GAPBS [3]: PR (PageRank), BFS, SSSP, and WCC, which cover different graph data access patterns. PR sequentially accesses both vertices and neighbors. BFS and WCC visit neighbors sequentially while accessing vertices randomly. SSSP introduces a random access pattern to neighbors when retrieving weights. Third, we implement TC (triangle counting) as the representative *graph pattern matching* query. This query requires DGS to support quick scanning in sorted order for fast set intersections. In summary, these two types of queries evaluate the effectiveness of complex, long-running graph queries (i.e., BI and IS workloads).

Real-world graphs following a power-law degree distribution complicate examining the performance of graph operations on different neighbor set sizes because accessing frequently used sets of elements can improve cache performance and lead to biased results. To address this, we design experiments using synthetic datasets. A *synthetic dataset* consists of sets of elements with uniform sizes. Each element is a vertex with an ID ranging from $[0, 2^{22})$. Assuming each vertex ID is 8 bytes, to evaluate performance on a neighbor set with 512 elements, we generate $x = \frac{8\text{GB}}{512\times 8 \text{ bytes}}$ sets of the same size. These sets are labeled from $[0, x)$. We generate insert, scan, and search OP streams using the same strategy as for real-world graphs, treating set IDs as vertex IDs and the sets as neighbor sets. We default to using 8GB to prevent all sets from residing in the cache, thereby simulating random memory access in large graphs.

Workload Executor. Each thread executes a stream or a graph algorithm. Operations within the same stream execute sequentially by one thread, while multiple threads can execute concurrently. Although the graph algorithms in the test framework can run in parallel, we execute them in a single thread to focus on the DGS's capability to empower concurrent graph query execution and handle different queries.

Performance Monitor. We use *throughput*, the number of edges processed (insert, search, scan) per second, to measure DGS efficiency. We use *latency*, the time elapsed to complete one query, to examine service quality. Recording the latency of each operation incurs non-trivial overhead due to the short duration of single graph operations. Therefore, we record latency every one hundred micro-operations. *Scalability* is measured by the throughput of micro-operations as the number of threads increases, while *concurrency* is measured when multiple micro-operation streams (or graph algorithm queries) are mixed. *Memory cost* tracks the memory consumption of competing storage methods.

6 Experimental Results

For brevity, we report the results for *lj*, a sparse graph with no high-degree vertices, and *g5*, a dense graph with high-degree vertices. Due to space limit, we have included the results on three graph analytics queries: BFS, SSSP, and WCC, graph operation latency and the comparison with their original implementations in the complete version. We omit the results of LLAMA due to its memory consumption issues. Guided by Equation 1, we conduct experiments according to the following roadmap.

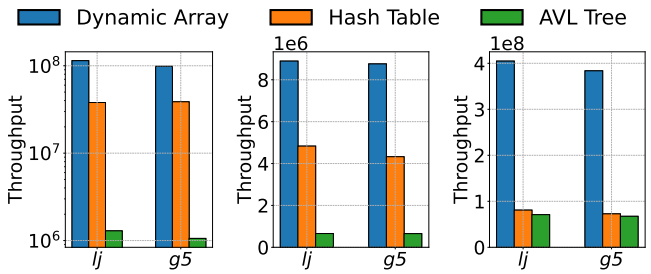
- **Graph Container Efficiency:** We measure the raw performance of graph containers on basic operations such as search, scan, and insert, as well as on graph analytics queries. These experiments are conducted in a single-threaded environment without concurrency control to isolate the baseline performance.

- **Concurrency Control Effectiveness:** We assess the impact of concurrency control on query performance, scalability across varying workloads, and the efficiency of concurrent read and write operations.
- **Batch Granularity:** We examine system performance under batch updates, focusing on behavior with different batch sizes.
- **Memory Usage:** We compare the memory overhead of competing methods against static graph formats.

6.1 Evaluation of Graph Containers

To compare the absolute performance of graph containers, we execute graph queries bypassing concurrency control operations in Figure 3. The elements in both vertex and neighbor indexes contain only vertex IDs without version information.

6.1.1 Efficiency of Vertex Indexes. Figure 9 presents the results for three vertex indexes listed in Table 2. For search efficiency, the dynamic array achieves speedups of over 2.6x compared to the hash table and two orders of magnitude compared to the AVL tree. Since SearchVtx is a step in graph operations on edges, this difference can significantly affect edge operations. Vertices are inserted in vertex ID order since existing methods incrementally assign vertex IDs from 0, as discussed in Section 2. The throughput of the dynamic array is approximately 2x and 8x higher than the hash table and AVL tree, respectively. The AVL tree is very slow because it copies the path for multiple graph snapshots. For scan operations, dynamic arrays are 4x faster than counterparts.



(a) SearchVtx. (b) InsVtx. (c) ScanVtx.

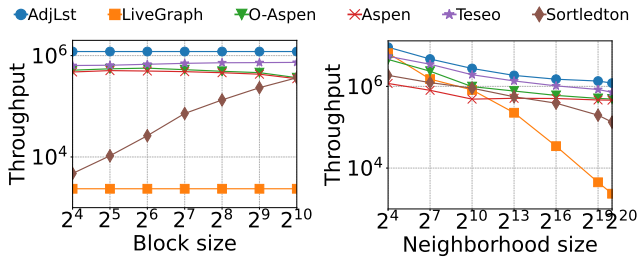
Fig. 9. Efficiency of vertex indexes.

6.1.2 Efficiency of Neighbor Indexes. To provide a performance reference for segmented PAM in Aspen, we implement O-Aspen, which uses a dynamic array as the vertex index and does not create vertex index snapshots for insert operations.

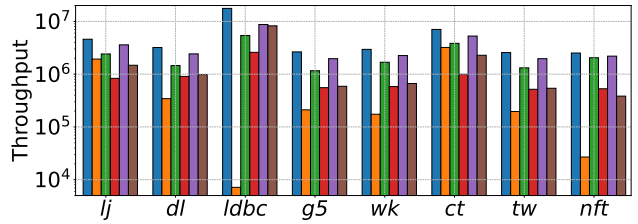
SearchEdge. Figure 10 presents the results on search efficiency. LiveGraph and AdjLst keep $N(u)$ in a continuous array and are therefore unaffected by $|B|$ in Figure 10a. LiveGraph runs slowly due to its unsorted neighbor index, while AdjLst performs the best, benefiting from efficient binary search on the continuous array. All segmented indexes perform better as $|B|$ increases, due to fewer blocks and lower block-index search overhead. O-Aspen, Aspen, and Teseo are much faster than Sortledton due to their efficient block indexing. Overall, AdjLst runs 1.2-5.8x times faster than O-Aspen and Teseo, and they are very close in performance on small neighbor sets in Figure 10b. Additionally, except for LiveGraph, search efficiency slightly decreases as $|N(u)|$ increases.

Figure 10c illustrates the results on real-world graphs. These methods exhibit similar relative performance as shown in Figures 10a and 10b, except for LiveGraph. LiveGraph is competitive with Sortledton on *lj* and *ct* but much slower on the other graphs. This is because most vertices in *lj* and *ct* have small degrees and no vertices with very high degrees.

InsEdge. As shown in Figure 3, insert efficiency is closely related to search efficiency. Figures 11a and 11b show that Teseo performs the best among segmented neighbor indexes. Although O-Aspen is faster than Teseo for search efficiency, it is slower for inserts due to the overhead of CoW. In Figure 11a, the performance of both Teseo and Aspen first increases and then decreases slightly because increasing block size reduces the time to locate the target block but increases the time required to copy data when inserting an element in the block.

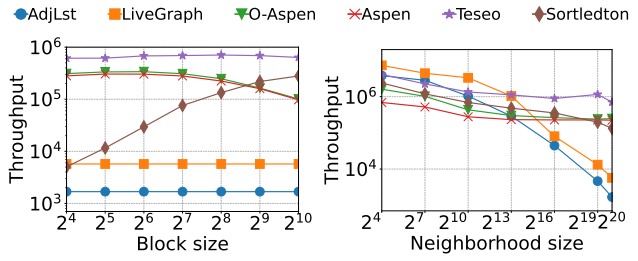


(a) Vary $|B|$ when $|N(u)| = 2^{20}$. (b) Vary $|N(u)|$ when $|B| = 256$.

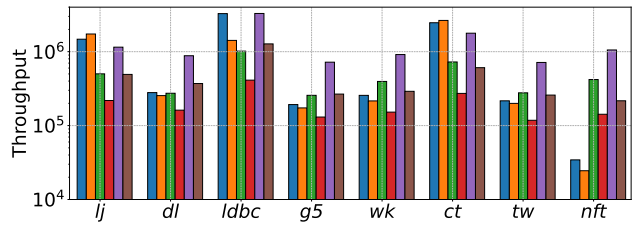


(c) Vary real-world graphs when $|B| = 256$.

Fig. 10. Efficiency of SearchEdge.



(a) Vary $|B|$ when $|N(u)| = 2^{20}$. (b) Vary $|N(u)|$ when $|B| = 256$.

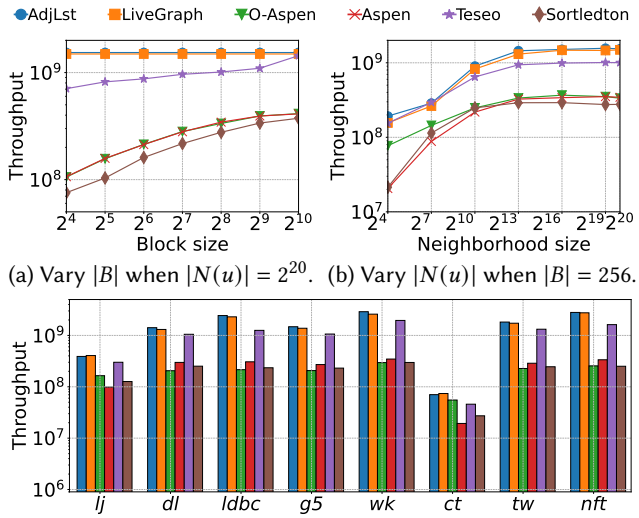


(c) Vary real-world graphs when $|B| = 256$.

Fig. 11. Efficiency of InsEdge.

While AdjLst performs the best for search, it is slow for inserts because it may need to move a large number of elements. Conversely, LiveGraph is slow for inserts because its search is slow. Specifically, LiveGraph checks whether the element to be inserted already exists, requiring a search operation. If the element exists, it updates the timestamp and appends a new version; otherwise, it directly appends the new version. However, since the neighbors are unsorted, LiveGraph must iterate through the entire neighbor set, leading to significant overhead. Although LiveGraph uses a bloom filter to reduce searches for non-existent edges, it struggles with existing edges, and false positives arise due to the filter's limited memory space. Figure 11c presents the results on the real-world graphs. LiveGraph and AdjLst outperform the counterparts on *lj* and *ct* because the two graphs are very sparse and have no vertices with high degree. Teseo generally works very well on all graphs.

ScanNbr. Figure 12 presents the results on scan efficiency. LiveGraph and AdjLst perform the best due to their continuous storage. Among segmented methods, Teseo runs up to 10x faster than its counterparts due to its reduced branch misses and instruction overhead in PMA compared to the skip list and PAM. Notably,



(a) Vary $|B|$ when $|N(u)| = 2^{20}$. (b) Vary $|N(u)|$ when $|B| = 256$.

(c) Vary real-world graphs when $|B| = 256$.

Fig. 12. Efficiency of ScanNbr.

Teseo's performance is close to that of LiveGraph and AdjLst when $|B| = 1024$. Figure 12c illustrates the results on real-world graphs. Consistent with Figures 12a and 12b, AdjLst and LiveGraph perform the best, with Teseo being the best among segmented methods. Specifically, Teseo achieves 52-77% of AdjLst's performance.

Table 4. Hardware metrics on ScanNbr with $|N(v)| = 2^{20}$ and $|B| = 256$. Lg, Ap, Ts, and Sl represent LiveGraph, Aspen, Teseo, and Sortledton, respectively.

	Continuous Neighbor Index		Segmented Neighbor Index		
Metric	**AdjLst**	**Lg**	**Ap**	**Ts**	**Sl**
L1 miss	2.57e8	2.53e8	2.38e8	2.65e8	5.48e8
L2 miss	1.35e8	2.38e8	1.82e8	2.15e8	2.86e8
LLC miss	1.34e8	1.34e8	1.84e8	2.15e8	2.38e8
DTLB miss	1.19e4	1.20e4	1.94e6	4.20e6	4.32e6
Branch miss	1.20e3	2.23e3	7.90e6	1.50e3	6.24e6
Instruction Number	1.07e9	1.61e9	7.57e9	1.14e9	1.18e10

Table 4 presents hardware utilization for competing methods on ScanNbr. We specifically measure cache misses (L1, L2, LLC, and DTLB), branch mispredictions, and number of instructions. Segmented neighbor indexes incur more cache misses than continuous methods due to their non-contiguous memory layout. Particularly, benefiting from continuous storage, AdjLst and LiveGraph have much fewer DTLB misses than counterparts. Additionally, the number of instructions are higher for segmented methods, reflecting the complexity of managing multiple segments during traversal. Branch mispredictions are also more frequent in segmented indexes, leading to performance penalties from pipeline stalls. These factors together result in higher overhead for segmented neighbor indexes during scanning.

6.1.3 Evaluation on Graph Analytical Queries. Table 5 reports the results for PR and TC, while the results for BFS, SSSP, and WCC are included in the full version. PR requires fast scan efficiency, whereas TC relies on efficient set intersection, necessitating both efficient search and scan in sorted order. Consequently, LiveGraph cannot support TC. All segmented methods set $|B|$ to 256. Aspen-w enables the flatten optimization, and Sortledton-w enables adaptive indexing with the threshold set to 256.

First, continuous methods outperform segmented methods. CSR runs faster than AdjLst with all neighbor sets in a single array. Both AdjLst and LiveGraph store neighbor sets in separate arrays, but AdjLst runs 1.0-1.9x faster because LiveGraph scans from the end of the array, preventing automatic vectorization by the compiler using SIMD. Although manual vectorization is possible, it introduces significant engineering

Table 5. Performance of competing methods on graph analytical queries (measured in seconds). Lg, Ap, Ts and Sl denote LiveGraph, Aspen, Teseo and Sortledton, respectively.

	Continuous Neighbor Index			Segmented Neighbor Index				
Dataset	**CSR**	**AdjLst**	**Lg**	**Ap**		**Ts**	**Sl**	
				w	wo		w	wo
PR								
lj	**4.66**	9.00	11.88	13.72	24.61	13.19	**13.06**	28.41
dl	**1.39**	1.43	1.92	5.03	5.22	**2.01**	5.53	5.24
ldbc	**20.57**	28.79	34.19	**44.28**	113.36	90.68	48.09	183.19
g5	**20.98**	29.20	55.73	43.19	80.35	**38.51**	48.30	77.46
wk	**6.64**	10.45	14.67	20.73	53.06	33.11	**19.30**	41.56
ct	**1.88**	6.58	7.25	7.96	15.84	11.02	**7.71**	21.78
tw	**24.87**	39.77	43.89	60.33	105.96	**55.58**	65.24	132.47
nft	**10.60**	19.00	24.52	34.13	103.19	72.09	**34.05**	178.13
TC								
lj	**15.36**	23.57	/	30.30	52.95	**29.95**	27.73	31.79
dl	**452.36**	467.72	/	620.72	620.19	**501.16**	626.52	571.12
ldbc	**89.24**	115.70	/	1404.68	1750.89	**142.19**	172.66	240.49
g5	**1042.68**	1220.60	/	1774.84	1971.00	**1395.10**	1890.74	1605.42
wk	**7.63**	13.58	/	42.43	77.81	**17.10**	17.05	18.91
ct	**3.17**	4.74	/	6.13	16.78	**5.91**	5.31	6.58
tw	**801.14**	860.73	/	1272.99	1467.60	**958.66**	1295.46	1092.51
nft	**218.23**	346.04	/	857.27	1011.24	**382.77**	486.28	482.18

challenges. Second, Teseo generally performs the best among segmented methods without optimizations because it stores blocks of $N(u)$ continuously. Third, enabling adaptive indexing achieves a speedup of 1.0-5.2x. Enabling flatten optimization improves performance by 1.0-3.0x. Nevertheless, CSR achieves a significant speedup, ranging from 1.2-53.7x, over the best segmented methods in each case.

Table 6 presents hardware metrics for PR and TC, including cache misses (L1, L2, LLC, and DTLB), branch mispredictions, and number of instructions. Although segmented neighbor indexes generally share the same big-O time complexity as continuous indexes, they result in higher cache miss rates (L1, L2, LLC), more branch mispredictions, and increased instruction counts, especially with the larger dataset *g5*. Additionally, while both CSR and AdjLst store neighbor sets as continuous arrays, CSR exhibits lower cache miss rates due to its compact memory format. For page rank queries, adaptive indexing mechanisms reduce LLC misses by 57.1% and 31.1% on dataset *lj* and *g5*, respectively, while flatten optimizations lower LLC cache misses by 30.2% and 17.0%. For triangle counting queries, flatten optimizations lower LLC cache misses by 31.3% and 49.2% along with a decrease in DTLB misses by 42.1% and 23.8%, contributing to the performance improvements.

6.1.4 Summary of Findings. **Q1. How effective are existing techniques in graph containers?** The vertex index significantly influences operations on small neighbor sets, as well as search and insert efficiency. This aspect has been overlooked in some previous works, leading to biased conclusions about the efficiency of neighbor indexes. The dynamic array performs best, being orders of magnitude faster than its counterparts. We confirm that segmented strategies can significantly accelerate insert operations on large neighbor sets, while also providing considerable scan efficiency. Both search and scan benefit from large blocks, but there is an optimal block size ($|B| = 256$) for insert efficiency in our experiments. CoW is generally slower than in-place updates. Moreover, we reveal that memory layout and vectorization significantly impact scan and search performance.

For additional optimizations, adaptive indexing can improve performance on real-world graphs because most vertices have a small degree. Second, the bloom filter offers limited help for SearchEdge due to two reasons: 1) we still need to locate the neighbor if it exists, and 2) the bloom filter has a high false positive rate on large neighbor sets. Third, flattening optimization can enhance long-running query performance, but it is challenging to automatically determine when to apply this optimization for a given query.

Table 6. Hardware metrics of competing methods on PR and TC (measured in billions). Lg, Ap, Ts and Sl denote LiveGraph, Aspen, Teseo and Sortledton, respectively.

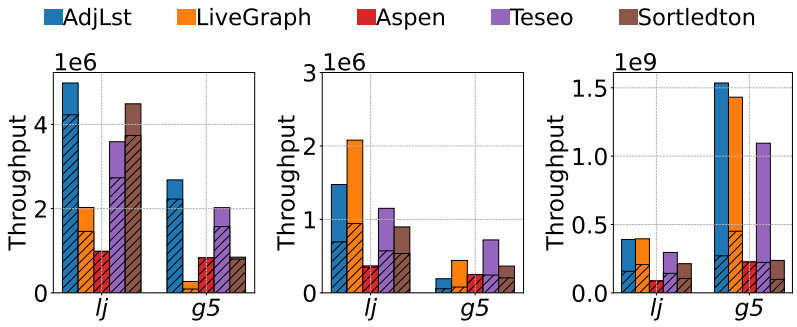
		Continuous Neighbor Index			Segmented Neighbor Index				
Metric	Dataset	CSR	AdjLst	Lg	Ap		Ts	Sl	
					w	wo		w	wo
PR									
L1 miss	*lj*	0.83	0.86	0.97	0.89	0.95	0.92	0.85	0.93
	g5	5.13	5.23	5.47	5.28	5.46	5.37	5.24	5.37
L2 miss	*lj*	0.80	0.96	1.10	0.96	1.07	1.45	0.94	1.35
	g5	4.22	4.59	4.86	4.78	4.97	5.76	4.96	5.68
LLC miss	*lj*	0.16	0.32	0.46	0.30	0.43	0.81	0.30	0.70
	g5	0.93	1.26	1.53	1.27	1.53	2.38	1.57	2.28
DTLB miss	*lj*	8.20	8.74	8.91	8.12	8.29	10.96	8.59	9.89
	g5	35.48	38.34	37.34	35.96	37.57	39.59	36.34	38.15
Branch miss	*lj*	0.05	0.05	0.05	0.05	0.33	0.05	0.05	0.05
	g5	0.09	0.09	0.09	0.12	0.65	0.10	0.09	0.09
Instruction number	*lj*	8.27	9.28	10.09	21.36	52.67	9.82	10.05	14.43
	g5	44.25	46.11	51.22	90.50	149.93	47.36	69.83	77.49
TC									
L1 miss	*lj*	0.07	0.14	/	0.19	0.70	0.23	0.16	0.20
	g5	2.35	3.00	/	24.44	19.93	7.78	8.52	8.61
L2 miss	*lj*	0.93	1.01	/	1.28	1.86	1.37	1.15	1.27
	g5	53.86	56.33	/	93.33	100.53	109.50	111.48	112.92
LLC miss	*lj*	0.65	0.74	/	0.83	1.18	1.11	0.76	1.00
	g5	31.63	31.96	/	62.87	66.15	68.99	69.19	71.77
DTLB miss	*lj*	5.76	10.67	/	13.70	23.67	14.42	12.45	13.60
	g5	97.34	137.09	/	336.05	441.16	211.64	380.34	237.88
Branch miss	*lj*	0.85	0.85	/	0.84	1.27	0.82	0.80	0.79
	g5	67.98	63.01	/	64.59	64.73	65.75	63.33	61.34
Instruction number	*lj*	31.44	31.93	/	60.00	75.00	32.62	44.27	56.05
	g5	6929	6932	/	10778	10900	6976	12020	12075

Q2. Which neighbor index performs the best, and what is the gap between it and CSR on read queries? The simple sorted dynamic array performs best for search, insert, and scan operations on small neighbor sets ($|N(u)| \leqslant 256$). The neighbor indexes of Teseo and Aspen perform much better than Sortledton, and LiveGraph significantly outperforms it on scan efficiency. Sortledton's results are biased due to two factors: 1) using adaptive indexing improves overall performance, and 2) G2PL significantly simplifies concurrency control. Teseo performs the best for both read and write operations on large neighbor sets. Nevertheless, there is still a significant gap between Teseo and CSR on graph analytic queries.

6.2 Evaluation of Graph Concurrency Control

We evaluate graph concurrency control (GCC) from three perspectives: overhead, scalability, and concurrency. All competing methods enable full-fledged GCC. Specifically, AdjLst, LiveGraph, Teseo, and Sortledton use the fine-grained concurrency control strategy with G2PL, while Aspen uses the coarse-grained strategy with CoW. For brevity, we refer to the two strategies as G2PL and CoW.

6.2.1 Overhead Incurred By GCC. Figure 13 presents the experimental results. As discussed in Section 4.2, G2PL causes a 2.0-5.7x slowdown in scan operations, while Aspen's performance remains unaffected. The slowdown on *g5* is greater than on *lj* because *g5*'s vertices have a larger degree, making the cost of looping over neighbor sets dominate scan operations. We also compare the performance of graph operations with and without locking and find that the overhead of locking is negligible. Therefore, the slowdown is primarily caused by the fine-grained version information. In contrast, CoW incurs no additional overhead, as it employs



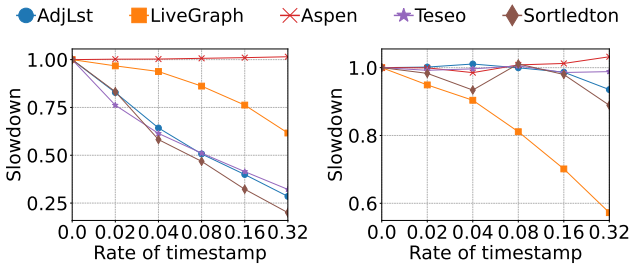
(a) SearchEdge (b) InsertEdge (c) ScanNbr

Fig. 13. Unshadowed denotes throughput without GCC, and shadowed denotes throughput with GCC.

Table 7. Throughput comparison of competing methods using 32 threads (million edges per second).

Datasets	***lj***			***g5***		
Operations	search	scan	insert	search	scan	insert
AdjLst	**64.35**	**3268.96**	2.43	**33.83**	3955.47	0.26
LiveGraph	19.49	1808.15	3.02	0.55	2540.43	1.24
Aspen	24.17	1669.10	0.20	8.37	**4389.51**	0.32
Teseo	48.89	2822.68	3.40	26.37	3021.91	3.17
Sortledton	40.20	2335.66	**3.45**	14.68	2570.57	**3.23**

a single-writer mechanism. As a result, the performance gap between Aspen and these fine-grained methods significantly reduces.

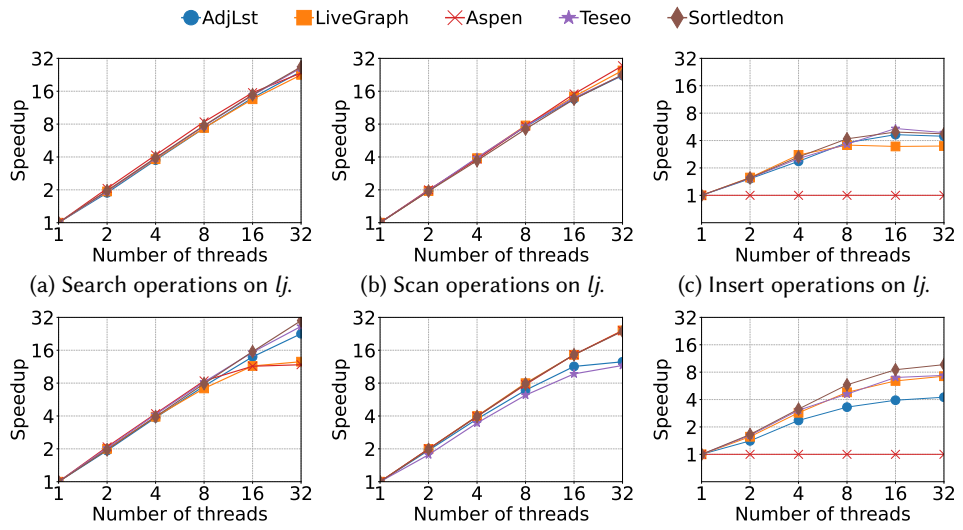


(a) Slowdown of ScanNbr. (b) Slowdown of SearchEdge.

Fig. 14. Impact of varying the percentage of neighbors with more than one versions.

6.2.2 Scalability Evaluation. An element can have multiple versions stored as a version chain or continuous. We examine the impact of multiple versions on search and scan efficiency as follows: given $N(u)$, we randomly pick $x\%$ of vertices $v \in N(u)$ and set three different versions. We vary the ratio from 2% to 32%. Figure 14 presents the results. The scan efficiency of methods using G2PL drops dramatically. The negative impact on AdjLst, Teseo, and Sortledton is greater than on LiveGraph due to the overhead of traversing the list and checking each element's version. In Figure 14b, LiveGraph's search efficiency drops significantly because it has to scan all versions to find the target, whereas AdjLst, Teseo, and Sortledton generally degrades by less than 10%. In contrast, multiple versions do not affect Aspen since it maintains versions in coarse granularity. The trend in insert efficiency mirrors that of search, so we omit the results for InsEdge.

We next evaluate the scalability of competing methods as the number of threads increases to 32, matching the number of physical cores in the testbed. Figure 15 presents the scalability results, and Table 7 shows the throughput with 32 threads. First, for search and scan efficiency on *lj*, competing methods achieve 22.6-29.7x and 22.3-26.7x speedup, respectively. They fail to scale linearly because *lj* is very sparse, and the irregular data accesses by many threads result in cache contention issues. As in the graph container evaluation, AdjLst significantly outperforms its counterparts on *lj*, as shown in Table 7.



(a) Search operations on *lj*. (b) Scan operations on *lj*. (c) Insert operations on *lj*.

(d) Search operations on *g5*. (e) Scan operations on *g5*. (f) Insert operations on *g5*.

Fig. 15. Scalability with the number of threads varying.

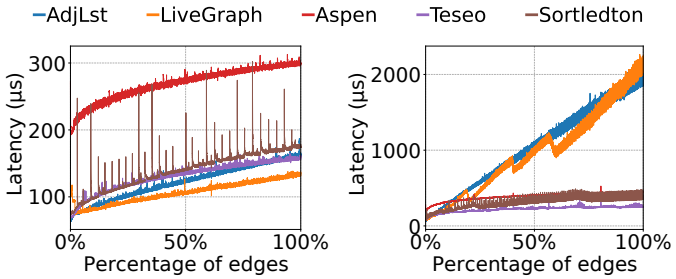
For search and scan efficiency on *g5*, competing methods achieve 12.5-29.8x and 12.5-26.7x speedup, respectively. Specifically, LiveGraph and Aspen only achieve 12.5x and 14.7x speedup on search operations because *g5* is large, and the AVL tree and bloom filter lead to severe cache contention issues. Interestingly, the speedup of AdjLst and Teseo is limited on scan but they achieve good throughput. To address this issue, we examine memory bandwidth utilization in Table 8. At 8 threads, AdjLst and Teseo consume most of the available memory bandwidth, limiting further speedup even as more threads are added. Notably, while AdjLst runs much faster than Aspen in the graph container evaluation, it is slower in Table 7 because AdjLst reads much more data due to fine-grained version management. As shown in Table 8, Aspen uses less bandwidth than competing methods because it avoids reading timestamps for each edge. Second, in Figures 15c and 15f, insert scalability is very limited because write queries require exclusive locks, and high-degree vertices frequently accessed amplify the lock contention issue. Aspen is slower than Teseo and Sortledton because it has a single writer.

Table 8. Memory bandwidth utilization during concurrent scans for competing methods (max bandwidth: 60 GB/sec).

Num of threads	*lj*					*g5*				
	AdjLst	**Lg**	**Ap**	**Ts**	**Sl**	**AdjLst**	**Lg**	**Ap**	**Ts**	**Sl**
1	3.66	3.14	1.35	3.70	2.80	9.54	8.07	2.11	10.19	6.01
2	6.69	5.97	2.10	6.87	5.68	18.37	6.78	5.35	19.79	12.54
4	11.59	10.77	4.50	12.18	10.39	36.15	29.05	10.14	36.49	23.82
8	18.10	17.44	8.10	19.34	17.08	51.03	46.62	19.63	54.41	41.07
16	23.96	24.12	12.68	26.12	24.14	56.00	54.51	35.70	57.33	53.63
32	25.78	26.61	17.70	28.76	26.65	57.76	56.69	49.07	57.81	55.62

We further compare performance variations during the insertion process. Figure 16 shows the insert latency along the edge insertion process. Since *lj* is sparse, AdjLst and LiveGraph generally outperform segmented methods, while they are slower on *g5*, a dense graph. Aspen's latency fluctuation is much smaller than that of other methods because 1) it uses a single writer and has no lock contention, and 2) the overhead of CoW on blocks is steady. The higher fluctuation in other methods is due to the expensive overhead of copying blocks and inserting into the skip list.

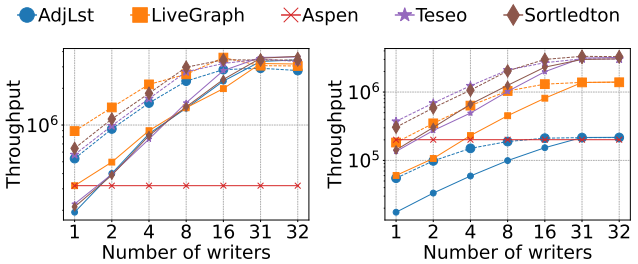
6.2.3 Concurrency Evaluation. We evaluate the concurrency of competing methods by mixing readers and writers with their total count equal to 32, the number of cores. Each writer executes an insertion stream, whereas a reader runs a PR query. Figure 17 presents the experiment results on write efficiency. The dashed



(a) Insert latency on *lj*. (b) Insert latency on *g5*.

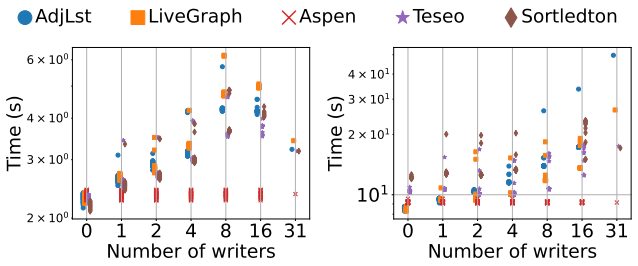
Fig. 16. Insert latency for every 100 operations during the edge insertion process.

line denotes write efficiency without readers executing, while the solid line represents write efficiency with readers. Introducing readers significantly degrades the performance of methods using G2PL due to vertex access contention. With 31 readers and 1 writer, the throughput is only 31.75-46.58% of that without any readers. In contrast, Aspen's throughput is only slightly affected by readers because they access the graph snapshot without locking data.



(a) Insert operations on *lj*. (b) Insert operations on *g5*.

Fig. 17. Write efficiency when mixing readers and writers.

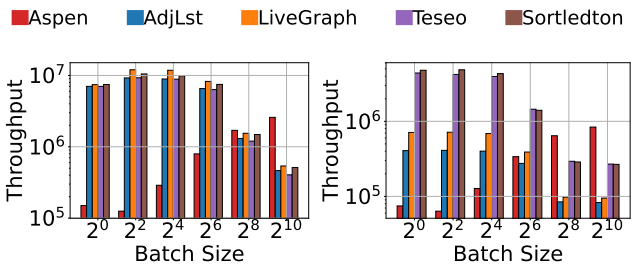


(a) PR on *lj*. (b) PR on *g5*.

Fig. 18. Read efficiency when mixing readers and writers.

Figure 18 presents the experiment results on read efficiency with varying numbers of writers. Each dot represents the average time for a reader to perform one iteration of PR. Methods using G2PL show significant performance decreases as the number of writers increases due to lock contention. Methods with poor insert efficiency (e.g., LiveGraph and AdjLst) are more affected than Teseo and Sortledton because they must hold exclusive locks for longer periods. The performance variance among different readers increases. In contrast, Aspen maintains steady performance. Specifically, Aspen performs worse than its counterparts when there are no writers but outperforms them when there are multiple writers.

6.2.4 Summary of Findings. **Q3. What is the impact of graph concurrency control on graph operations?** For fine-grained methods, maintaining and checking versions for each element incurs significant overhead due to the need to load more data and perform extra computations. Traversing the version list can cause



(a) Batch update on *lj*. (b) Batch update on *g5*.

Fig. 19. Write efficiency with batch size increasing.

more overhead for scans, while continuous version storage can be costlier for searches. In contrast, the coarse-grained method (i.e., Aspen) avoids these issues, resulting in a smaller performance gap.

Q4. How is the scalability and concurrency of competing methods? Overall, all competing methods generally exhibit good scalability in search and scan. Search scalability can be affected by cache contention, while scan scalability can be limited by memory bandwidth. Aspen can outperform the fine-grained methods despite its less efficient graph container. However, existing works overlook the problem of hardware utilization. In contrast, insert operations have very limited scalability due to heavy lock contention from frequent access to high-degree vertices.

Despite using MVCC, in practice, readers and writers of fine-grained methods interfere significantly on graphs because high-degree vertices are frequently accessed. While Aspen's read performance can outperform its counterparts, it suffers from slow insert speeds due to its inefficient CoW strategy. In a word, both fine-grained and coarse-grained methods face severe performance issues in a multi-threaded context.

6.3 Evaluation of Update Granularity

In the above experiments, each write query consisted of a single update. We now evaluate how batch size affects performance. Since our test system has 32 CPU cores, Aspen uses a single writer and 32 threads to process one batch in parallel, while the other methods use 32 writers, each independently processing its own batch.

As shown in Figure 19, Aspen outperforms the other methods when the batch size exceeds 2^8, with throughput steadily increasing as the batch size grows from 2^0 to 2^{10}. This is because Aspen avoids lock contention with a single writer, creates one snapshot for multiple updates, and leverages parallelism across update operations. In contrast, methods using G2PL exhibit a significant performance drop as batch size increases, especially beyond 2^8, due to the need to hold more locks per query and the longer execution time required to process larger batches, which increases lock contention.

Q5. How does the batch granularity affect the performance of competing methods? Coarse-grained methods perform poorly with small batch sizes since parallelism within a write query is limited, and a snapshot must be created for each update. However, their performance improves significantly as batch size increases. In contrast, fine-grained methods excel with small batch sizes but face severe performance degradation due to lock contention among queries as the batch size grows.

6.4 Evaluation of Memory Consumption

We evaluate the memory consumption of competing methods by measuring the *resident set size* as reported by the OS. Table 9 presents the results. *wo* and *w* denote the storage without and with version information, respectively. Aspen uses coarse-grained version management, resulting in negligible memory consumption differences (less than 10MB) between *wo* and *w*. Therefore, we report its *w* implementation for reference.

Q6. What is the impact of graph containers and version management in DGS on memory consumption, and what is the gap between DGS and CSR? Overall, Aspen consumes 3.3-11.0x times more memory than CSR. Among the fine-grained methods, the most optimal consumes 0.9-1.3x and 4.1-8.9x times more memory than Aspen and CSR, respectively. This is primarily because: 1) they maintain version information

for each element, and 2) they leave empty slots in neighbor indexes. These results indicate that fine-grained methods can cause severe memory issues, limiting the scalability of DGS on large graphs.

Table 9. Memory consumption of competing methods (GB).

Dataset	CSR	Aspen	AdjLst		LiveGraph		Teseo		Sortledton	
			wo	w	wo	w	wo	w	wo	w
lj	0.67	3.58	0.98	2.74	1.66	6.24	9.67	30.98	2.62	4.31
dl	0.76	2.52	0.99	2.89	1.21	3.71	1.33	6.87	3.00	3.11
ldbc	2.84	17.96	3.99	10.52	7.61	28.95	61.16	OOM	15.30	20.19
g5	3.95	15.33	5.42	15.41	7.17	24.62	21.57	78.42	15.25	19.55
wk	0.98	8.18	1.41	3.49	3.49	13.09	28.11	86.08	5.38	7.12
ct	0.27	2.06	0.42	1.11	0.84	3.32	7.42	22.81	1.11	2.14
tw	4.11	20.20	5.49	15.09	9.02	31.98	45.89	OOM	17.48	22.04
nft	1.38	14.94	2.06	4.99	5.93	23.01	58.76	OOM	9.07	12.27

7 Related Work

In this paper, we study in-memory dynamic graph storage [12, 14, 24, 37, 62] that supports concurrent read and write queries, particularly single updates. Below, we discuss the related work.

Graph Databases and Benchmarks. Graph databases such as Neo4J, Virtuoso, and K'uzu [22] also support transactions. They typically operate on external storage and focus on supporting labeled property graphs. Some, like Neo4J, use the read-committed isolation level [44] to improve performance and simplify transaction management. Additionally, there are graph databases designed for distributed environments [7, 9, 35, 58], which focus on optimizing communication and distributed storage. For graph benchmarks, LDBC provides a variety of workloads [16, 28, 51] to evaluate the performance of graph databases. These workloads also use labeled property graphs with various operations on labels or properties. Our paper focuses on in-memory DGS, optimizing operations on graph topology. These existing DGS methods do not consider labels and properties, and thus cannot be directly applied. However, as the labeled property graph model is widely used, researching DGS on this model is an interesting direction, for example, combining DGS with well-designed columnar graph storage [39].

Graph Processing Frameworks. Many frameworks have been proposed for parallel analysis of static graphs, such as Ligra [47] and Ligra+[48] for single-machine environments, and Pregel[38], Giraph [26], Gemini [61], and Grapes [18] for distributed environments. These frameworks typically use CSR for in-memory storage and focus on parallelization strategies that exploit inter-query parallelism for graph queries like BFS, SSSP, and PR.

Dynamic Graph Processing Frameworks. Dynamic graph processing frameworks target frequently updated graphs. They focus on designing novel approaches to maintain intermediate results for graph queries, reducing the overhead of re-computation when the graphs are modified. For example, KickStarter [53] minimizes unnecessary computations by obtaining a trimmed approximate subset of the nodes affected by the update. RisGraph [21] uses a variant of adjacency lists and sparse arrays as its storage structure and can switch between vertex parallelism and edge parallelism. To achieve streaming processing, it employs a tree-based classification of safe and unsafe updates. GraphZeppelin [52] uses new linear sketching data structures to solve the streaming connected components problem. These studies are orthogonal to our research.

8 Conclusion

We summarize our findings on the relative performance of computing methods, key technical insights, and opportunities for future research and optimization below.

Relative Performance. Guided by Equation 1, we summarize the efficiency T_p of graph access operations and the concurrency control overhead amplification ratio α_p across competing methods. We focus on four key operators: SearchVtx, SearchEdge, ScanNbr, and InsEdge. The efficiency and overhead amplification ratios are based on experimental results in Sections 6.1.2 and 6.2.2. Relative performance is calculated by first computing the geometric mean of throughput across datasets, then normalizing the results on a scale of 1 to 5, with 5 being the best. Figure 20 visualizes the relative performance of the methods across different dimensions.

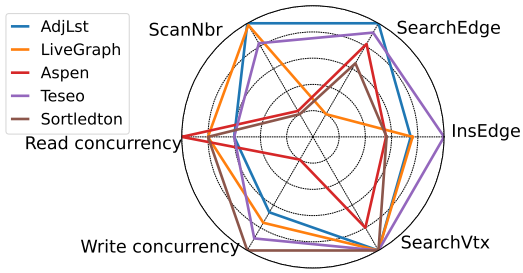


Fig. 20. Relative performance of existing methods for factors in Equation 1.

For read efficiency, including ScanNbr and SearchEdge, AdjLst performs the best. Teseo is slower on ScanNbr and SearchNbr, but offers a balanced read and write performance. For SearchVtx, dynamic array storage for vertices achieves the best results. Aspen excels in read concurrency due to its coarse-grained version management but performs poorly in write concurrency because it only supports a single-writer.

Key Technical Insights. Based on our extensive experimental results, we confirm the following key findings from previous work: 1) segmenting neighbor sets into blocks effectively balances read and write performance; 2) in the single update setting, fine-grained methods improve write throughput and outperform coarse-grained methods; 3) adaptive indexing significantly enhances performance by leveraging the sparsity of real-world graphs, reducing LLC misses; and 4) converting tree indexes to arrays improves the performance of long-running queries in coarse-grained methods.

However, we find that current design choices in concurrency control and graph containers lead to significant performance issues, which diverge from modern trends and present new challenges for the community.

- **The fine-grained strategy, widely adopted in recent works, introduces severe performance issues.** First, performing version checks for each neighbor imposes substantial computation overhead and consumes excessive memory bandwidth. Second, maintaining versions for each neighbor requires a large amount of memory, limiting scalability. Third, readers and writers interfere heavily, and insert operations face limited scalability due to lock contention, especially when accessing high-degree vertices.
- **Existing research often focuses on selecting various block indexes but neglects hardware-level considerations.** This creates a performance gap compared to continuous storage methods. Fragmented memory layouts increase cache misses (L1, L2, LLC, and DTLB). Additionally, the complexity of segmented methods introduces high instruction overhead and frequent branch mispredictions.

Research and Optimization Opportunities. This study identifies key areas for future research and optimization. In graph concurrency control, new version management techniques are needed to reduce the space and efficiency overhead of fine-grained versions, improving scalability. One research opportunity is to manage versions in a more coarse-grained manner, potentially eliminating the overhead of version checks for each edge. Additionally, more efficient concurrency control mechanisms are required to minimize interference among concurrent queries. Given that graph queries are typically read-intensive, another research direction is to explore relaxed locking requirements for read queries, thereby reducing interference from write operations.

For graph containers, segmenting neighbor sets and using adaptive indexing can enhance graph access efficiency. However, further optimization of memory layout and management strategies is necessary to reduce cache misses and instruction overhead, improving hardware utilization. Developing graph containers that balance read and write efficiency while integrating improved concurrency control methods is essential, as a significant read efficiency gap remains between dynamic and static graph storage formats.

Acknowledgments

Jixian Su and Chiyu Hao contributed equally to this work. Shixuan Sun is the corresponding author.

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] Renzo Angles, Peter Boncz, Josep Larriba-Pey, Irini Fundulaki, Thomas Neumann, Orri Erling, Peter Neubauer, Norbert Martinez-Bazan, Venelin Kotsev, and Ioan Toma. 2014. The linked data benchmark council: a graph and RDF industry benchmarking effort. *ACM SIGMOD Record* 43, 1 (2014), 27–31.

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

[4] Michael A Bender and Haodong Hu. 2007. An adaptive packed-memory array. *ACM Transactions on Database Systems (TODS)* 32, 4 (2007), 26–es.

[5] Philip A Bernstein and Nathan Goodman. 1981. Concurrency control in distributed database systems. *ACM Computing Surveys (CSUR)* 13, 2 (1981), 185–221.

[6] Philip A Bernstein, Vassos Hadzilacos, Nathan Goodman, et al. 1987. *Concurrency control and recovery in database systems*. Vol. 370. Addison-wesley Reading.

[7] Maciej Besta, Robert Gerstenberger, Marc Fischer, Michal Podstawski, Nils Blach, Berke Egeli, Georgy Mitenkov, Wojciech Chlapek, Marek Michalewicz, Hubert Niewiadomski, et al. 2023. The Graph Database Interface: Scaling Online Transactional and Analytical Graph Workloads to Hundreds of Thousands of Cores. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–18.

[8] Maciej Besta, Robert Gerstenberger, Emanuel Peter, Marc Fischer, Michał Podstawski, Claude Barthels, Gustavo Alonso, and Torsten Hoefler. 2023. Demystifying graph databases: Analysis and taxonomy of data organization, system designs, and graph queries. *Comput. Surveys* 56, 2 (2023), 1–40.

[9] Andrew Carter, Andrew Rodriguez, Yiming Yang, and Scott Meyer. 2019. Nanosecond indexing of graph data with hash maps and VLists. In *Proceedings of the 2019 International Conference on Management of Data*. 623–635.

[10] Symas Corporation. [n. d.]. Lightning Memory-Mapped Database (LMDB). https://symas.com/lmdb/. Accessed: 2024-06-29.

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

[12] 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.

[13] Laxman Dhulipala. 2019. Aspen. https://github.com/ldhulipala/aspen. Accessed on July 18, 2024..

[14] 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.

[15] 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.

[16] Orri Erling, Alex Averbuch, Josep Larriba-Pey, Hassan Chafi, Andrey Gubichev, Arnau Prat, Minh-Duc Pham, and Peter Boncz. 2015. The LDBC social network benchmark: Interactive workload. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. 619–630.

[17] Orri Erling, A. Averbuch, Josep Larriba-Pey, Hassan Chafi, Andrey Gubichev, A. Prat, Minh-Duc Pham, and Peter Boncz. 2015. The LDBC Social Network Benchmark: Interactive Workload. In *Proceedings of ACM SIGMOD International Conference on Management of Data 2015 (SIGMOD'15)*.

[18] Wenfei Fan, Wenyuan Yu, Jingbo Xu, Jingren Zhou, Xiaojian Luo, Qiang Yin, Ping Lu, Yang Cao, and Ruiqi Xu. 2018. Parallelizing sequential graph computations. *ACM Transactions on Database Systems (TODS)* 43, 4 (2018), 1–39.

[19] Alan Fekete, Dimitrios Liarokapis, Elizabeth O'Neil, Patrick O'Neil, and Dennis Shasha. 2005. Making snapshot isolation serializable. *ACM Transactions on Database Systems (TODS)* 30, 2 (2005), 492–528.

[20] Guanyu Feng. 2020. LiveGraph. https://github.com/thu-pacman/LiveGraph. Accessed on July 18, 2024..

[21] 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.

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

[23] Per Fuchs. 2022. Sortledton. https://gitlab.db.in.tum.de/per.fuchs/sortledton. Accessed on July 18, 2024..

[24] 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.

[25] 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.
[26] Mark Grover, Ted Malaska, Jonathan Seidman, and Gwen Shapira. 2015. *Hadoop application architectures: Designing real-world big data applications.* " O'Reilly Media, Inc.".
[27] 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.
[28] Alexandru Iosup, Tim Hegeman, Wing Lung Ngai, Stijn Heldens, Arnau Prat-Pérez, Thomas Manhardto, Hassan Chafio, Mihai Capotă, Narayanan Sundaram, Michael Anderson, et al. 2016. LDBC Graphalytics: A benchmark for large-scale graph analysis on parallel and distributed platforms. *Proceedings of the VLDB Endowment* 9, 13 (2016), 1317–1328.
[29] Alfons Kemper and Thomas Neumann. 2011. HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots. In *2011 IEEE 27th International Conference on Data Engineering*. IEEE, 195–206.
[30] 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.
[31] 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).
[32] 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.
[33] Dean De Leo. 2022. Teseo. https://github.com/cwida/teseo. Accessed on July 18, 2024..
[34] Jure Leskovec and Andrej Krevl. 2014. SNAP Datasets: Stanford Large Network Dataset Collection. http://snap.stanford.edu/data.
[35] Changji Li, Hongzhi Chen, Shuai Zhang, Yingqian Hu, Chao Chen, Zhenjie Zhang, Meng Li, Xiangchen Li, Dongqing Han, Xiaohui Chen, et al. 2022. ByteGraph: a high-performance distributed graph database in ByteDance. *Proceedings of the VLDB Endowment* 15, 12 (2022), 3306–3318.
[36] Peter Macko. 2017. LLAMA. https://github.com/goatdb/llama. Accessed on July 18, 2024..
[37] 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.
[38] Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. 2010. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. 135–146.
[39] Amine Mhedhbi, Pranjal Gupta, Shahid Khaliq, and Semih Salihoglu. 2021. A+ indexes: Tunable and space-efficient adjacency lists in graph database management systems. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*. IEEE, 1464–1475.
[40] Michael Mitzenmacher. 2001. Compressed bloom filters. In *Proceedings of the twentieth annual ACM symposium on Principles of distributed computing*. 144–150.
[41] 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.
[42] 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.
[43] William Pugh. 1990. Skip lists: a probabilistic alternative to balanced trees. *Commun. ACM* 33, 6 (1990), 668–676.
[44] Raghu Ramakrishnan and Johannes Gehrke. 2002. *Database management systems*. McGraw-Hill, Inc.
[45] 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.
[46] Sherif Sakr, Angela Bonifati, Hannes Voigt, Alexandru Iosup, Khaled Ammar, Renzo Angles, Walid Aref, Marcelo Arenas, Maciej Besta, Peter A Boncz, et al. 2021. The future is big graphs: a community view on graph processing systems. *Commun. ACM* 64, 9 (2021), 62–71.
[47] 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.
[48] 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.
[49] Abraham Silberschatz, James L Peterson, and Peter B Galvin. 1991. *Operating system concepts*. Addison-Wesley Longman Publishing Co., Inc.
[50] 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.
[51] Gábor Szárnyas, Jack Waudby, Benjamin A Steer, Dávid Szakállas, Altan Birler, Mingxi Wu, Yuchen Zhang, and Peter Boncz. 2022. The LDBC social network benchmark: Business intelligence workload. *Proceedings of the VLDB*

Endowment 16, 4 (2022), 877–890.

[52] David Tench, Evan West, Victor Zhang, Michael A Bender, Abiyaz Chowdhury, J Ahmed Dellas, Martin Farach-Colton, Tyler Seip, and Kenny Zhang. 2022. GraphZeppelin: Storage-friendly sketching for connected components on dynamic graph streams. In *Proceedings of the 2022 International Conference on Management of Data*. 325–339.

[53] 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.

[54] 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.

[55] Yingjun Wu, Joy Arulraj, Jiexi Lin, Ran Xian, and Andrew Pavlo. 2017. An empirical evaluation of in-memory multi-version concurrency control. *Proceedings of the VLDB Endowment* 10, 7 (2017), 781–792.

[56] 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.

[57] Xiangyao Yu, George Bezerra, Andrew Pavlo, Srinivas Devadas, and Michael Stonebraker. 2014. Staring into the abyss: An evaluation of concurrency control with one thousand cores. (2014).

[58] Wei Zhang, Cheng Chen, Qiange Wang, Wei Wang, Shijiao Yang, Bingyu Zhou, Huiming Zhu, Chao Chen, Yongjun Zhao, Yingqian Hu, et al. 2024. BG3: A Cost Effective and I/O Efficient Graph Database in Bytedance. In *Companion of the 2024 International Conference on Management of Data*. 360–372.

[59] Zhen Zhang, Bingqiao Luo, Shengliang Lu, and Bingsheng He. [n. d.]. Live Graph Lab: Towards Open, Dynamic and Real Transaction Graphs with NFT. https://livegraphlab.github.io.

[60] Zhen Zhang, Bingqiao Luo, Shengliang Lu, and Bingsheng He. 2023. Live Graph Lab: Towards Open, Dynamic and Real Transaction Graphs with NFT. *ArXiv* abs/2310.11709 (2023). https://api.semanticscholar.org/CorpusID:264289175

[61] Xiaowei Zhu, Wenguang Chen, Weimin Zheng, and Xiaosong Ma. 2016. Gemini: A {Computation-Centric} distributed graph processing system. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. 301–316.

[62] 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).

Received July 2024; revised September 2024; accepted November 2024