Aster: Enhancing LSM-structures for Scalable Graph Database

DINGHENG MO, Nanyang Technological University, Singapore
JUNFENG LIU, Nanyang Technological University, Singapore
FAN WANG, Nanyang Technological University, Singapore
SIQIANG LUO*, Nanyang Technological University, Singapore

There is a proliferation of applications requiring the management of large-scale, evolving graphs under workloads with intensive graph updates and lookups. Driven by this challenge, we introduce Poly-LSM, a high-performance key-value storage engine for graphs with the following novel techniques: (1) Poly-LSM is embedded with a new design of graph-oriented LSM-tree structure that features a hybrid storage model for concisely and effectively storing graph data. (2) Poly-LSM utilizes an adaptive mechanism to handle edge insertions and deletions on graphs with optimized I/O efficiency. (3) Poly-LSM exploits the skewness of graph data to encode the key-value entries. Building upon this foundation, we further implement Aster, a robust and versatile graph database that supports Gremlin query language facilitating various graph applications. In our experiments, we compared Aster against several mainstream real-world graph databases. The results demonstrate that Aster outperforms all baseline graph databases, especially on large-scale graphs. Notably, on the billion-scale Twitter graph dataset, Aster achieves up to 17x throughput improvement compared to the best-performing baseline graph system.

CCS Concepts: • **Information systems → Graph-based database models**; **Data management systems**; **Information storage systems**; • **Theory of computation → Data structures design and analysis**.

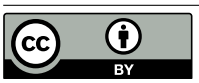
ACM Reference Format:
Dingheng Mo, Junfeng Liu, Fan Wang, and Siqiang Luo. 2025. Aster: Enhancing LSM-structures for Scalable Graph Database. *Proc. ACM Manag. Data* 3, 1 (SIGMOD), Article 12 (February 2025), 26 pages. https://doi.org/10.1145/3709662

1 Introduction

Graph database systems provide robust platforms for storing, processing, and analyzing graph data, supporting a variety of applications including social networks, recommendation systems, and biological networks. These platforms necessitate rapid responses for online graph navigations where only small fractions of the graph are explored, including intensive data updates (e.g., adding a new edge) and massive data queries (e.g., navigating edges of a vertex) [22, 33, 57]. For instance, social media users often interact with new connections, while the system must regularly recommend content tailored to their latest activities. In such scenarios, both lookup and update efficiencies are critical. Therefore, developing efficient graph database systems for evolving graphs is indispensable for modern applications.

*Corresponding Author

Authors' Contact Information: Dingheng Mo, Nanyang Technological University, Singapore, dingheng001@e.ntu.edu.sg; Junfeng Liu, Nanyang Technological University, Singapore, junfeng001@e.ntu.edu.sg; Fan Wang, Nanyang Technological University, Singapore, fan008@e.ntu.edu.sg; Siqiang Luo, Nanyang Technological University, Singapore, siqiang.luo@ntu.edu.sg.



This work is licensed under a Creative Commons Attribution International 4.0 License.

© 2025 Copyright held by the owner/author(s).
ACM 2836-6573/2025/2-ART12
https://doi.org/10.1145/3709662

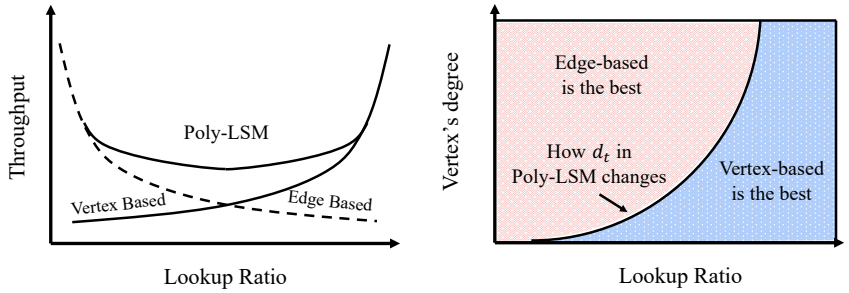


Fig. 1. The left figure illustrates the sub-optimal trade-off between lookup and update in existing LSM-based structures. The right figure demonstrates our insight that the superiority of vertex-based updates and edge-based updates are affected by lookup ratio and vertex degree.

Existing graph database systems can be *in-memory* [4, 27, 37, 59, 83] or *disk-resident* [2, 6, 7, 9, 28]. In-memory graph database systems consider that the whole graph data resides in main memory, focusing on designing efficient in-memory data structures. Disk-resident graph database systems, in contrast, operate under the assumption that the graph size may exceed the capacity of the main memory, or only a portion of the main memory is available for managing graph data. As a result, these databases primarily store graph data on disk, presenting unique challenges in designing efficient disk-based graph operations. Regarding scalability and data persistence, disk-resident graph database systems such as Neo4j [6], OritentDB [7], and DGraph [28] occupy a large share of the market in the industry. Therefore, this paper aims to develop a comprehensive disk-resident graph database capable of efficiently managing large, evolving graph structures.

Challenges and limitations of existing graph database systems. Despite the recent advancements, existing graph database systems still struggle to simultaneously provide strong performance for fundamental graph operators such as querying incident edges of a vertex or updating an edge. These core operations are essential for supporting other important graph queries. To facilitate the discussion, let us consider three evaluation criteria: (1) Update, which measures the ability to handle graph evolution, such as edge inserts or deletes; (2) Lookup, which assesses the ability to perform graph traversal along the edges; and (3) Scalability, which evaluates the extent of performance degradation as the graph size increases.[1]

There are three types of mainstream storage engines supporting current (disk-resident) graph database systems, which are *linked-list-based*, *relational-table-based* and *LSM-structure-based*. Neo4j [6] and OrientDB [7] are representatives that employ linked list-based storage, where the adjacency list of a vertex is maintained as a linked list. This approach allows efficient updates, as new edges can be appended directly to the end of the edge file. However, it also leads to inferior lookup performance and unsatisfactory scalability due to incurring substantial random access. In the worst case, a total of d (the degree of a vertex) I/Os is required. Relational-table-based databases, such as SQLG [9], store edges and vertices in separate tables, facilitating efficient updates by appending new entries directly to the tables. However, graph traversal requires costly table joins, limiting neighbor retrieval performance and scalability. Log-Structured-Merge (LSM)-tree-based databases such as DGraph [28] and Sparksee [8] can store either an adjacency list (vertex-based) or an edge (edge-based) as a key-value entry (See Section 3.1). Edge-based LSM-trees offer exceptional update performance and moderate traversal performance by buffering and sequentially flushing new edges

[1]Note that this term does not pertain to the ability of the graph database to scale across multiple machines.

Structures	Linked List	Relational Table	LSM-tree (vertex-based)	LSM-tree (edge-based)
Update	★★★	★★★	★★	★★★★★
Lookup	★★	★	★★★★	★★★
Scalability	★★	★★	★★★★	★★★★

Table 1. Summary of different disk-based storage structures.

that lead to reduced I/Os. Vertex-based LSM-trees, while having strong traversal performance, exhibit lower update performance due to the read-and-modify scheme. The analysis of these structures is concluded in Table 1.

The Problem: Is there a data structure supporting graph storage with even stronger performance on lookup and update, while achieving high scalability? From Table 1, LSM-trees are close to this goal but fall short in either lookup or update performance. A deeper investigation shows that vertex degree plays a crucial factor in determining the benefits of employing vertex-based or edge-based LSM-trees for operations related to a vertex (See Section 3.3). Inspired by this observation, we propose a vertex-wise adaptive update mechanism to harness the strength of both vertex-based and edge-based LSM-trees for graph storage, forming a novel graph storage engine named Poly-LSM. Poly-LSM achieves efficient update and lookup operations concurrently for large-scale evolving graphs, by wisely adapting its LSM-tree update approach to each individual operation. Poly-LSM supports large-scale evolving graphs with efficient update and lookup performance by wisely adapting its LSM update approach to each individual operation. Our experiments show that Poly-LSM can achieve up to 17x throughput improvement compared to the state-of-the-art graph database systems. Our novel designs are summarized below.

Design 1: Multi-layout graph storage to accelerate update and lookup operations concurrently. Poly-LSM employs an innovative LSM-tree-based structure that allows the co-existence of both vertex-based and edge-based graph storage. This design significantly expands the concept of key-value entries, allowing each entry to represent a vertex, an edge, or an adjacency list. Moreover, Poly-LSM fully exploits the dynamic nature of the LSM-tree. It initially represents graph updates in an edge-based form, ensuring decent update efficiency and subsequently utilizes the LSM-tree's dynamic merging process to gradually consolidate entries of the same vertex on disk. Therefore, entries at larger LSM-tree levels more closely resemble vertex-based representation, which further enhances the efficiency of lookups. In this manner, Poly-LSM achieves update efficiency comparable to edge-based LSM-trees and lookup efficiency akin to vertex-based LSM-trees simultaneously.

Design 2: Adaptive edge update strategy to facilitate evolving graph. The hybrid storage layout of Poly-LSM enables Poly-LSM to deal with edge updates through either an edge-based or vertex-based approach flexibly. As depicted in Figure 1 (Right), we discover that the overhead of updating edges of the two methods varies from the degree of the updated vertex. In vertex-based storage approaches, updating an edge for a high-degree vertex necessitates the retrieval and rewriting of a large entry, thereby incurring significant overhead. Hence, the edge-based layout is preferred. Conversely, for low-degree vertices, this additional cost is relatively low, thus opting for an edge-based storage layout is suboptimal as it incurs higher overhead for later lookups. Hence, we carefully analyzed the costs associated with two distinct update approaches with different vertex degrees. This analysis yields corresponding selection criteria that can be further bolstered by a degree sketch method [67]. This enables Poly-LSM to adjust the storage layout of the edge update adaptively thus consistently delivering desirable overall performance, as shown in Figure 1 (Left).

Design 3: Comprehensive graph database to support intricate graph applications. We developed Aster, a comprehensive graph database empowered by Poly-LSM as its storage engine,

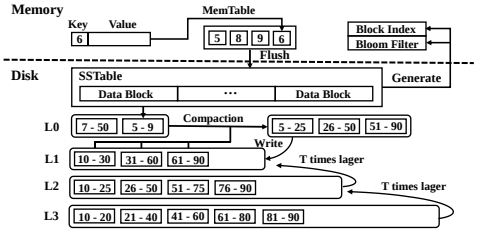


Fig. 2. Diagram of the LSM-tree structure.

to effectively manage and navigate graph structures for diverse graph applications ranging from fundamental operations to sophisticated queries. Aster provides a graph query interface utilizing Gremlin [77], a powerful data-flow query language. Moreover, it is able to efficiently parse graph queries into fundamental operations within Poly-LSM which are subsequently assembled into an execution schedule for supporting complex graph queries.

Contributions. In summary, we make the following contributions.

- This paper is at the forefront of efforts to optimize LSM-tree-based graph storage. We introduce Poly-LSM, a fast storage engine for graphs embedded with a graph-oriented LSM-tree-based structure, with the following techniques: (1) We design an effective and flexible LSM-based storage layout that maps diverse graph structures into polymorphic key-value entries in the LSM-tree and supports both vertex-based and edge-based updates. (2) We scrutinize the impacts of employing various update methods and proposing an adaptive update mechanism that achieves better overall performance. Our analysis indicates that Poly-LSM achieves comparatively optimal processing overhead for a series of graph updates under a uniform workload. (3) We exploit the intrinsic characteristics of graph data within LSM-trees to further improve space efficiency through appropriate encoding of entries.
- We develop and implement a complete graph database, Aster, on top of Poly-LSM and evaluate its performance against multiple mainstream graph database systems widely used in the industry. The result demonstrates that Aster outperforms all of them over a wide spectrum of workloads.

2 Background

Key-value Paradigm. Key-value store is a non-relational database paradigm that uses an associative mapping to store data. In this paradigm, data is organized as a collection of key-value entries, where each key serves as a unique identifier and is paired with its corresponding information as the value. Due to its simplicity, the key-value store has the potential to offer better scalability, usability, and efficiency compared to the traditional relational database paradigm, leading to its increasing popularity in the industry.

LSM-tree Structure. The LSM-tree is a durable, multi-tiered indexing structure for key-value pairs, which achieves efficient write performance by transforming random writes to the disk into sequential writes. As illustrated in Figure 2, all updates, insertions, and deletions are initially sorted in a main memory buffer, termed MemTable, as key-value entries. When the MemTable reaches its capacity limit, it is converted into a persistent SSTable file stored on the disk. The SSTables are organized into several levels, with each level having a capacity T times larger than the previous one, and keys are sequential and non-overlapping in SSTables within each level. When the total data size within a level exceeds its capacity, a compaction is triggered. This process involves fetching

Notation	Description
$G(V, E)$	The input graph with vertex set V and edge set E.
n	Total number of vertices in the graph.
m	Total number of edges in the graph.
$\overline{d}$	Average degree of the graph.
$d(u)$	Number of neighbors of vertex u.
T	Capacity ratio between adjacent levels in LSM-tree.
L	Number of levels in LSM-tree.
B	Size of a disk block (in bytes).
I	Size of a vertex ID (in bytes).

Table 2. Frequently used notations.

one or more SSTables from that level and all overlapping SSTables from the next level into memory to perform a merge sort, and then placing the resulting new SSTable into the next level.

Lookup in LSM-tree. The LSM-tree assists queries by indexing the key range and offset of all data blocks in memory and generating Bloom filters for all keys in each SSTable as we depict in Figure 2. When querying a key, it locates the SSTable containing the key at each level by checking the key range, then uses the Bloom filter to verify the presence of the key. If the Bloom filter returns a positive result, it finds the data block containing the key through the block index and retrieves it from the disk, if it exists. Thus, each lookup incurs at most one block I/O per level, and is very likely to avoid incurring I/O when the key is absent in the level.

Existing Graph Database Systems. Over the past decades, there are extensive studies on graph database systems that developed many distinctive storage engines. Specifically, Neo4j [6], one of the most renowned graph database systems, uses linked storage blocks to store vertices and edges respectively. To illustrate, each vertex is formatted as a fixed size entry in the vertex block, where a physical pointer (i.e., position offset in some edge block) is recorded, by which Neo4j can access the edges of a given vertex by doing traversal on the linked list. Similarly, OrientDB [7] stores vertices and edges separately and maintains pointers to connect them. However, the pointers in OrientDB are not physical position indicators but rather logical ones, which are mapped to the physical positions in an append-only data structure. Likewise, ArangoDB [2], a document-based graph database, serializes each vertex, edge, and the connection of them into a JSON format document. Moreover, some traditional relational databases like PostgreSQL [66] and MySQL [5] also have been adapted, known as SQLG [9] and MyRocks [31], which utilize relational tables to store vertices and edges separately and join them up when doing graph traversal. In addition, JanusGraph [4] stores graph in adjacency list format by using key-value storage backends (i.e., BerkeleyDB by default), where each entry holds the vertex id as the key and the collection of edges of this vertex is the value. Meanwhile, NebulaGraph [88] serializes each edge or vertex into a key-value entry and stores it in an LSM-tree.

3 Poly-LSM: Graph-Oriented LSM-based Storage Engine

This section presents the storage engine Poly-LSM, which is built on an innovative graph-oriented LSM-based design.

3.1 Vertex-based and Edge-based Mechanisms

We first introduce two typical LSM-tree layouts for storing graphs, which we term as vertex-based LSM-tree and edge-based LSM-tree.

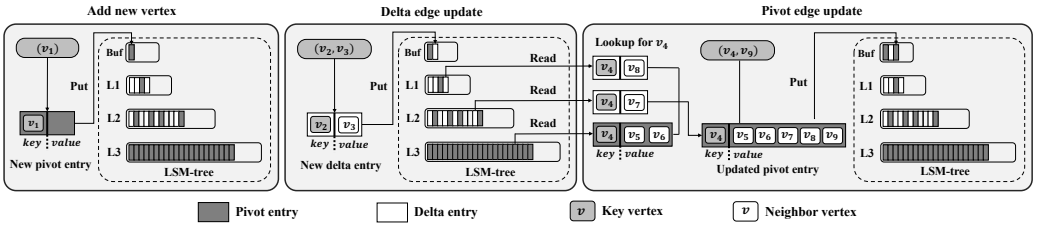


Fig. 3. An overview of the Poly-LSM demonstrating the workflow of several basic operations including *add new vertex*, *delta edge update*, and *pivot edge update*. In this figure, *Buf* represents the memory buffer of the LSM-tree.

Edge-based LSM-trees. Each graph edge corresponds to a key-value entry in the LSM-tree. A notable example of graph database systems employing this layout is NebulaGraph, which creates a tuple for each edge and serializes it as a key-value entry for storage. Hence edge updates simply require inserting a new entry into the LSM-tree. Meanwhile, querying a vertex requires identifying all edges related to that vertex which might access multiple entries at different positions across each LSM level. This is much slower than point lookups even accelerated through range scans.

Vertex-based LSM-trees. Each key-value entry represents vertex's adjacency list, where the key represents the vertex and the value contains all its neighbors. Therefore, querying a vertex u and its neighbors requires just one fetch of the latest entry keyed by u. Nevertheless, inserting new edge (u, v) is more complex, which reads the existing entry containing u's adjacency list to update the value, and then reinserted into the LSM-tree. This necessitates an additional lookup for each edge update. In addition, storing all neighbors of a vertex typically leads to large entry size which could introduce significant write cost during the compaction process.

3.2 Poly-LSM Layout and Basic Operators

We introduce Poly-LSM, a new LSM-based structure incorporating the merits of both vertex-based and edge-based schemes, to achieve efficient large-scale graph storage on disk. Poly-LSM supports polymorphic key-value entries and adaptively selects proper update schemes on the fly to enhance system performance.

Multiple Entry Types. Different from vertex-based LSM-trees, each vertex in Poly-LSM has multiple entries to store its neighbor list, including one unique *pivot entry* and possibly multiple *delta entries*. The pivot entry is usually located at the largest level, with its value containing the majority of the neighbors of that vertex. For a vertex, each edge update generates a delta entry containing the updated neighbor list. This delta entry would be merged into the corresponding pivot entry during the subsequent compaction process. In cases where two or multiple delta entries are merged, they would produce another delta entry rather than a pivot entry.

Querying Poly-LSM. Most complex graph queries and algorithms, such as finding multi-hop contexts and PageRank, can be broken down into a series of adjacency queries of vertices. These fundamental neighbor retrieval operations in Poly-LSM are executed by *lookup*. Specifically, to get the neighbors of vertex u, Poly-LSM initiates a search from the memory buffer and iteratively searches through subsequent levels for entries with u's ID as the key, and stops until it encounters u's pivot entry. The values of all relevant entries found are then aggregated to create a comprehensive list of u neighboring edges. For simpler graph queries, such as inquiries a single edge (u, v), Poly-LSM follows the same processes while returning immediately upon finding the relevant result.

Updating Poly-LSM. Poly-LSM supports both vertex update and edge update. Adding a new vertex can be simply realized by inserting a pivot entry with an empty value, and vertex deletion is

achieved with the out-of-place update mechanism in LSM-trees which inserts a special mark of delete. Poly-LSM allows two edge updates simultaneously, which benefit different scenarios.

Delta Edge Update. For adding an edge (u, v) in the graph, Poly-LSM simply needs to insert a delta entry into the LSM-tree with the key as u's ID and the value as v's ID. [2] In addition, an edge deletion is also performed as the insertion of an entry, where a specific value is designated to signify the deletion of that edge.

Pivot Edge Update. To add an edge (u, v) with pivot update, Poly-LSM first conducts a lookup for vertex u to retrieve its current neighboring edges. It then forms a new pivot entry for u, appending the ID of vertex v to the fetched list of neighboring edges. This updated pivot entry is subsequently inserted into the LSM-tree. For instance, as shown in Figure 3, to add edge (v_4, v_9), Poly-LSM firstly lookup the two delta entries in Level 1 and Level 2, and the pivot entry in Level 3, with values of $\{v_8\}$, $\{v_7\}$, and $\{v_5, v_6\}$ respectively. These, along with the new neighbor v_9, are merged to form the updated pivot entry value $\{v_5, v_6, v_7, v_8, v_9\}$, which is then appended to the memory buffer. Edge deletions follow a similar process, with the distinction that the edge is to be removed from the lookup results before forming the updated pivot entry.

The delta update increases the number of valid entries corresponding to a vertex, resulting in higher read amplification when querying that vertex. Therefore, it is more suitable for scenarios with a lower lookup ratio in the workload. On the other hand, the pivot update requires reading all the neighbors of the vertex being updated first, making it more suitable for vertices with a low degree, as the total data size that needs to be read is smaller.

Hybrid Layout in Poly-LSM. In practice, for each edge update, Poly-LSM would adaptively select the most appropriate approach from these two update methods. This approach is more flexible than the default merge behavior in RocksDB, which relies solely on a fixed delta update method. As a result, it offers greater potential for enhancing system performance. Typically, an edge update is performed as delta updates, while pivot updates are used for vertexes with low degree or under workloads with fewer lookups. We will present detailed reasoning by analyzing the distinct trade-off in these two schemes in Section 3.3. In this manner, a newly updated edge (u, v) in Poly-LSM likely first materializes in the edge-based layout and subsequently moves to deeper levels through compaction, eventually merging into the vertex-based pivot entry corresponding to vertex u. This hybrid storage layout enables Poly-LSM to be more versatile in managing large, evolving graphs.

Practical Implementation in RocksDB. After discussing the high-level layouts of Poly-LSM, we will proceed to describe how we extended RocksDB to implement it in practice.

Default RocksDB. By default, RocksDB uses *Put* interface to insert data. However, the inserted entries will overwrite previous entries with the same key during compaction. Algorithmically, this means that each inserted entry acts as a pivot entry.

Merge Operator. To support delta entries, we leverage the *Merge Operator* [14, 30, 90] API provided by RocksDB. This is a user-defined interface that allows specifying custom behaviors for handling multiple values associated with the same key. When entries are inserted using the *Merge* operation, they are combined with other entries of the same key according to the behavior defined by the merge operator, rather than discarding obsolete values like *Put*.

Our Implementation. We override the *Merge Operator* in RocksDB to realize delta entries in Poly-LSM's hybrid layout. The basic *Merge Operator* [30] in RocksDB appends the value of a new entry

[2] Note that in undirected graphs, an edge update (u, v) is always accompanied by another edge update (v, u), and the processing of the latter is identical to the former. Therefore, in our subsequent discussion, we will only mention the edge update (u, v).

to those with the same key. Whereas, we significantly enrich the semantics of this interface to support graph storage with following key features. (1). Our implementation ensures no duplicate edges within an adjacent list. For example, suppose we have a pivot entry with the key as node u and the value v, w, and a delta entry with the value v. The value produced by RocksDB's default merge operator is v, w, v, while our implementation gives v, w with duplicate edges filtered out. (2). Our custom *Merge Operator* ensures edges in the value being ascending sorted by node ID which is not guaranteed by RocksDB's basic appending method. This facilitates subsequent merging, querying, and value encoding. (3). The *Merge Operator* interface only supports value accumulation, making it unable to handle edge deletions independently. We address this limitation by adding a label to the delta entry's value which triggers the edge removal during the merge process.

With this implementation, hybrid updates can be supported by invoking different RocksDB APIs. Specifically, pivot updates are handled by *Get* and *Put* interfaces to read existing entries and insert updated version. While delta updates solely rely on *Merge* interface to append the edge entry. Poly-LSM employs a carefully designed adaptive algorithm to determine the optimal timing for applying pivot updates versus delta updates, as we will detail in next section.

3.3 Adaptive Updates in Poly-LSM: Toward Optimal I/O Efficiency

Poly-LSM would adaptively employ the most appropriate method for each incoming edge update (u, v) based on the current workload and the degree of u, which effectively decreases the expected I/O cost. The rationale is that the pivot update incurs additional costs at insertion, as it requires reading and rewriting the existing neighbors of the edge's vertices. Moreover, the updated pivot entry is usually larger than a delta entry, which leads to more I/Os in subsequent compactions. On the other hand, the delta update leads to additional costs during subsequent related queries. This additional cost arises because, until the new edge is merged with the pivot entry of the corresponding vertex during compaction, any query targeting the vertex requires an independent I/O operation to access this newly updated edge. Next, we will analyze the I/O costs associated with the delta and pivot update methods in detail. Specifically, in line with prior works [24, 45, 74], we assume a static workload with fixed proportions of different operations and a uniform key-value distribution. The LSM-tree is assumed to have a balanced structure, where the size ratio between consecutive levels remains T, including the largest level. The frequently used notations are listed in Table 2.

Cost of Delta Update. Equation 1 represents the expected cost of delta update C_D of the edge (u, v). The first term accounts for the cost of writing new delta entry into the LSM-tree. The second estimates the additional prospective read cost incurred for lookups of vertex u before the delta entry merges into u's pivot entry.

$$C_D = \underbrace{\frac{2I \cdot T \cdot L}{B}}_{\text{Write I/O cost}} + \underbrace{N_L \cdot P_u}_{\substack{\text{Prospective} \\ \text{read I/O cost}}} \tag{1}$$

Write I/O Cost. In the first term, $\frac{2I}{B}$ represents the number of block I/Os generated each time the new delta entry is written to disk, where I is the size of the vertex IDs in entries in bytes, and B is the size of a disk block in bytes. To determine the actual write I/O cost, this value must be multiplied by the LSM-tree's inherent write amplification, which is the product of the adjacency ratio T and the total number of levels L.

Prospective Read I/O Cost. In the second term, N_L indicates the expected number of lookups in the workload that occur from the insertion of the new delta entry until it merges with u's pivot entry and P_u is the probability that a lookup targets vertex u. The expression $N_L \cdot P_u$ represents the

number of lookups for vertex u during the existence of the delta entry corresponding to the new edge (u, v). Each of these lookups incurs an additional block I/O due to the need to fetch this delta entry. We model the expected number of operations that occurred during the existence of edge (u, v)'s delta entry in Equation 2, where m is the total number of edges in the graph, θ_L and θ_U denotes the ratio of lookups and updates in the workload, respectively.

$$W = \frac{m \cdot \theta_L}{(T-1) \cdot \theta_U} \tag{2}$$

The rationale is that from the moment an entry is newly written to the LSM-tree until it reaches the final level, on average, a number of edges equivalent to the sum of the capacities from the first to the penultimate level are written into the LSM-tree. Since each level is T times larger than its precedent level, this amount roughly constitutes $\frac{1}{T-1}$ of the total size of the LSM-tree, which contains m edges in total. As for P_u, because there are n vertices on the graph, and the lookup might target at any vertex, we have $P_u = 1/n$. Finally, we can derive the following equation, where $\overline{d} = \frac{m}{n}$ denotes the average degree of the graph.

$$C_D = \frac{2I \cdot T \cdot L}{B} + \frac{\theta_L \cdot \overline{d}}{\theta_U \cdot (T-1)} \tag{3}$$

Cost of Pivot Update. The expected cost of adding an edge (u, v) with the pivot update can be written as Equation 4, which also consists of two terms: (1). The read I/O cost of conducting a query to fetch neighboring edges of vertex u from disk. (2). The cost of rewriting the updated pivot entry into the LSM-tree.

$$C_P = \underbrace{2 + \frac{(d(u)+1) \cdot I}{B}}_{\text{Lookup cost for } u} + \underbrace{\frac{(d(u)+2) \cdot I \cdot T \cdot L}{B}}_{\text{Rewrite I/O cost}} \tag{4}$$

Rewrite I/O Cost. In the second part of Equation 4, $(d(u)+2) \cdot I$ represents the size of the updated pivot entry, which includes one vertex ID in the key and $(d(u)+1)$ IDs in the value. We multiply this by the LSM-tree's write amplification TL then divide by disk block size B to derive block I/O incurred by writing this entry.

Lookup Cost for u. The cost of performing a lookup for vertex u can be further divided into the I/Os that may occur when finding delta entries of u in a level, and the necessary I/Os when reading out u's pivot entry upon its encounter. [3]

We estimate the expected overhead for retrieving delta entries of vertex u as approximately one I/O considering the significantly reduced size of smaller levels. From previous analysis, the last level holds roughly $\frac{(T-1) \cdot m}{T}$ edges, leading to the $(L-i)$-th level containing $\frac{(T-1) \cdot m}{T^{1+i}}$ edges. For uniform key distribution, the probability that the target delta entry is found at the $(L-i)$-th level is $P_L^i = 1 - \left(1 - \frac{1}{n}\right)^{\frac{(T-1) \cdot m}{T^{1+i}}}$, where $\frac{1}{n}$ denotes the probability of an update operation involving the target vertex. Since $m = n \cdot \overline{d}$, we can reformulate P_L^i as $\left(1 - \left(\frac{n-1}{n}\right)^{n \cdot \frac{(T-1) \cdot \overline{d}}{T^{1+i}}}\right)$. In our context, the total vertex number n is typically large, indicating $\left(\frac{n-1}{n}\right)^n$ approaches $\frac{1}{e}$, which yields the following equation.

$$P_L^i \approx 1 - e^{-\frac{(T-1) \cdot \overline{d}}{T^{1+i}}} \tag{5}$$

It is clear that as i increases, P_L^i declines sharply, with its value affected by $\overline{d}$ and T as well. For real-world graphs, whose average degrees typically range from 10 to 100, Equation 5 indicates that

[3] During the lookup for u, false positives from Bloom filters might occasionally result in disk I/Os, but this occurrence is negligible. In Poly-LSM, the Bloom filter is configured with 10 bits per key, a standard setting in LSM-based key-value databases. This configuration keeps the false positive rate below 1%, making it highly unlikely.

the probability P_L^1 of accessing delta entries at the $(L-1)$-th level is close to 1 when RocksDB's default size ratio 10 is employed. In contrast, for smaller levels, P_L^2 and P_L^3 decrease drastically even approaching zero. This allows us to approximate the expected overhead for retrieving delta entries for vertex u as roughly 1 I/O, due to the minimal contributions from levels smaller than the $(L-1)$-th level. Besides, the precise expected overhead can also be calculated by collecting the expected retrieval count across all levels:

$$C_R = \sum_{i=1}^{L-1} P_L^i. \tag{6}$$

More specifically, consider the Wikipedia dataset [92], whose average degree is 37.11, the probability at the (L-1)-th level is $P_L^1 = 0.964$. While at the (L-2)-th level, this probability reduces to $P_L^2 = 0.284$, and at the (L-3) level, it diminishes further to $P_L^3 = 0.033$.

The expected cost of retrieving the pivot entry equals $1 + \frac{(d(u)+1)\cdot I}{B}$. In Poly-LSM, entry sizes are irregular, which can lead entries to span multiple disk pages and consequently induce multiple block I/Os during retrieval. Assuming the starting positions of entries on the disk relative to the start of a block are uniformly random, the expected number of blocks an entry spans is its length $(d(u)+1)\cdot I$ divided by the disk block size B, plus one.

Combining these factors, the total I/O cost of the lookup for u is thus calculated to be $2+\frac{(d(u)+1)\cdot I}{B}$.

Adaptive Selection. The delta update incurs lower costs compared with the pivot update when $C_P > C_D$, which can be written as

$$2 + \frac{(d(u)+1)\cdot I}{B} + \frac{d(u)\cdot I\cdot T\cdot L}{B} > \frac{\theta_L \cdot \overline{d}}{\theta_U \cdot (T-1)} \tag{7}$$

Equation 7 gives a threshold d_t expressed in Equation 8. When $d(u) \geq d_t$, Poly-LSM always adopts the delta update for edge (u, v). Otherwise, Poly-LSM employs the pivot update.

$$d_t = \max\{\lceil \frac{\theta_L \cdot \overline{d} \cdot B}{\theta_U \cdot I \cdot (T-1)\cdot(T\cdot L+1)} - \frac{2B}{I\cdot(T\cdot L+1)} - \frac{1}{T\cdot L+1} \rceil, 0\} \tag{8}$$

Here we give a practical running example. We set the vertices and edges as 64-bit integers, each occupying 8 bytes, which gives $I = 8$ Bytes. We let the disk block size B be 4KB, the LSM-tree's adjacency ratio T be 10, the number of levels in LSM-tree L be 4, and the average degree of input graph $\overline{d}$ be 32, which all reflect typical real-world configurations. We assume the workload comprises equal proportions of lookups and updates, resulting in $\theta_L = 0.5$, and $\theta_U = 0.5$. Then, as specified by Equation 8, we have $d_t = 21$. This means that in this scenario Poly-LSM opts for the delta update when the degree of the source vertex is 21 or higher. For lower degrees, it employs the pivot update.

According to Equation 8, when the workload is update-intensive, the parameter d_t becomes very small. This causes the majority of vertex updates to be handled as delta updates, which enhances write performance. Conversely, when the workload is lookup-intensive, d_t becomes very large, leading most updates to be processed as pivot updates, thereby improving read performance. This adaptability enables Poly-LSM to provide solid throughput in any scenario, offering better robustness compared to other graph database systems.

Is the Adaptive Update Scheme Optimal? For an evolving graph $G(V, E)$, consider a sequence of N edge updates, where each update can be chosen between a delta update or a pivot update. Essentially, there are 2^N possible update sequences, and we denote $S^* = \{v_1^*, v_2^* ... v_N^*\}$ as the optimal sequence which incurs the smallest overhead among all possible combinations. Further let $S = \{v_1, v_2 ... v_N\}$ represent the sequence chosen by Poly-LSM. *We can show that the cost of S is very*

close to that of the optimum S^*. In particular, when the workload is uniform across graph vertices, S is exactly the optimal sequence S^*. In more complicated situations, when the workload is skewed over graph vertices, the cost of S is at most a $\log m$ factor of S^*, as Lemma 3.1 indicates.[4]

Lemma 3.1. *When the workload is uniformly distributed, the total cost of Poly-LSM is $O(1)$-competitive. When the workload is skewed, Poly-LSM is $O(\log m)$-competitive.*

Extending to 1-Leveling LSM-tree Designs. In our analysis, we adopt the commonly used leveling structure [23, 26, 45, 61] by default, which maintains a single sorted run at each level. However, our approach can be easily extended to other structures through adjustments to the cost model. For example, in practice, RocksDB typically employs a variation known as 1-leveling, where each buffer flush is stored as a separate run in the first level and does not merge with existing data. In this case, the cost of a delta update is given by $\frac{2I\cdot(T(L-1)+1)}{B} + \frac{\theta_L\cdot\overline{d}}{\theta_U\cdot(T-1)}$, where the first term represents the modified write I/O cost for a delta entry. The rationale is that each entry incurs only one time write amplification at the first level and T times at the remaining levels under the 1-leveling design. Correspondingly, the cost of a pivot update is $2 + \frac{(d(u)+1)\cdot I}{B} + \frac{(d(u)+2)\cdot I\cdot(TL-T+1)}{B}$. According to Equation 7, a delta update outperforms a pivot update when

$$2 + \frac{(d(u)+1)\cdot I}{B} + \frac{d(u)\cdot I\cdot(T\cdot L-T+1)}{B} > \frac{\theta_L\cdot\overline{d}}{\theta_U\cdot(T-1)} \tag{9}$$

This leads to the corresponding threshold d_t' as presented

$$d_t' = \max\{\lceil \frac{B}{I\cdot(T\cdot L-T+2)}\cdot(\frac{\theta_L\cdot\overline{d}}{\theta_U\cdot(T-1)} - 2) - \frac{1}{T\cdot L-T+2}\rceil, 0\} \tag{10}$$

Notably, the threshold with 1-leveling is slightly higher than that of the leveling structure, which suggests an increased likelihood of using pivot updates.

The Degree Sketch. When adding an edge (u, v), Poly-LSM's adaptive mechanism uses u's degree to determine the appropriate edge update method, based on the threshold outlined in Equation 6. Therefore, it is necessary to index the degree information of vertices in memory to avoid I/O overhead during this decision-making process. However, directly storing degree information in memory for very large graphs consumes significant space. On the other hand, compressing degree information is not viable, as it cannot accommodate changes in degrees caused by intensive edge updates in evolving graphs. Poly-LSM employs an approximate sketch based on Morris Counter [67] to index the degree of each vertex in memory with minimal space overhead, which also supports dynamic updates to the degree information.

The key idea of the degree sketch involves accounting for degree increments based on a probability that correlates with the current degree size. As Figure 4 illustrates, for each vertex, Poly-LSM allocates 8 bits to index its degree in memory, divided into two parts: the first 4 bits termed as the exponent, and the last 4 bits termed as the mantissa, each corresponding to an integer between

[4]The full proof of Lemma 3.1 is included in our technical report [1].

Algorithm 1 Update Degree Sketch

Input: a new edge (u, v)

1: $E_u \leftarrow sketch[u] >> 4$;
2: $r \leftarrow rand(0, 1)$;
3: **if** $r < 2^{-E_u}$
4: $\quad sketch[u] \leftarrow sketch[u] + 1$;

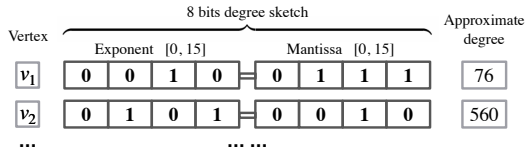


Fig. 4. The degree sketch allows Poly-LSM to index the degree of each vertex in memory with very few bits.

0 and 15. The update process of the degree sketch is shown in Algorithm 1. Let E_u denote the exponent of vertex u in the degree sketch, and M_u denote its mantissa. Every time the degree of vertex u increments, M_u has a probability of $\frac{1}{2^{E_u}}$ to increment. Otherwise, M_u remains unchanged. If M_u is incremented upon reaching its maximum value of 15, it will be reset to 0, and then E_u is increased by 1. We use the equation below to calculate the approximate value of u's degree.

$$\hat{d}(u) = (2^{E_u} - 1) \cdot 2^4 + 2^{E_u} \cdot M_u \tag{11}$$

For example, in Figure 4, for vertex v_1 we have $E_u = 2$ and $M_u = 7$, thus the approximate degree that it sketches is $\hat{d}(v_1) = (2^2 - 1) \times 2^4 + 2^2 \times 7 = 76$. Similarly, the approximate degree for v_2 is $\hat{d}(v_2) = (2^5 - 1) \times 2^4 + 2^5 \times 2 = 560$. It is obvious that for any vertex u, $\hat{d}(u)$ is an unbiased approximation of u's actual degree, which satisfies $\mathbb{E}[\hat{d}(u)] = d(u)$. Furthermore, our degree sketch provides a steady approximation for any degree scale, as the following lemma summarizes.

Lemma 3.2. *When a graph $G(V, E)$ is loaded into Poly-LSM, for any $u \in V$, the approximation $\hat{d}(u)$ returned by the degree sketch of Poly-LSM satisfies*

$$\Pr[|\hat{d}(u) - d(u)| \geq \epsilon d(u)] \leq \frac{1}{6\epsilon^2} \tag{12}$$

Proof Sketch. According to Corollary 3 in [20], we have

$$\frac{\mathrm{Var}[\hat{d}(u)]}{\mathbb{E}[\hat{d}(u)]} \leq \sqrt{\frac{3}{8 \cdot 2^4 - 3}} \leq \frac{1}{6} \tag{13}$$

Applying Chebyshev's inequality to Eqaution 13, we can derive the conclusion in Equation 12. □

In practice, the relative error of our degree sketch's approximation is around 10% for any degree. This level of precision is adequate for applying adaptive edge updates, as Poly-LSM selects the update method based on an adaptive degree threshold. For vertices with degrees that vary by no more than 10% from this threshold, the cost of applying either update method is comparable. Furthermore, the memory consumption of the degree sketch is generally notably lower than that of the Bloom filters. This is because it allocates only 8 bits per vertex, whereas the Bloom filters allocates 10 bits for each key in LSM-tree, which can include duplicate vertices. Since both the exponent and mantissa are between 0 and 15, the maximum value that the degree sketch can represent is $\hat{d}_{max} = (2^{15} - 1) \cdot 2^4 + 2^{15} \cdot 15 = 1015792$, which is sufficiently large. In instances where a vertex's degree exceeds the maximum limit, Poly-LSM would consistently select the edge-based update for this vertex to avoid the extreme cost of retrieving all its neighbors.

3.4 Improve Space Efficiency with Partitioned Elias-Fano Encoding

The space efficiency of Poly-LSM could be further enhanced by customizing the encoding method to the specific characteristics of graph storage. As outlined in Section 3.2, each entry in Poly-LSM consists of a key, representing the ID of a vertex u, and a value, which is a sorted list of vertex IDs detailing u's neighboring edge. The vertex IDs are integers constrained by n, the total number of

vertices in the graph, making the inverted index compression techniques [91] particularly suitable. Additionally, edge distribution for a vertex is often skewed in real-world applications. Therefore, we propose encoding the entry value using the partitioned Elias-Fano method [69], which provides desirable compression rates and encoding/decoding efficiency by considering edge locality.

In our implementation, the partitioned Elias-Fano algorithm divides the vertex ID list $\{k_i\}$ in an entry into t segments and encodes the local information using a two-level structure. Assume each segment contains s consecutive vertex IDs within a sub-universe N_j, where j denotes the segment ID. The first level comprises a list of the starting ID of each segment and the terminating ID of the last segment to identify them. In the second level, each segment is encoded using the Elias-Fano algorithm. Specifically, the sub-universe N_j is further divided into equal-length sub-segments represented by different prefixes. Within each sub-segment, each ID is divided into a prefix and remaining bits. The Elias-Fano algorithm then unary encodes the ID counts concatenated with the remaining bits of each ID, requiring $2 + \log_2 \frac{N_i}{t}$ bits per element. Additionally, the ID list in the first level can be encoded similarly, consuming approximately $2 + \log_2 t$ bits per segment. It is worth mentioning that the compression rate can be adjusted by tuning the number of segments or the prefix length in the Elias-Fano encoding, depending on the distribution of edges.

3.5 Adapt for Various Graph Format

We will now describe how these previously discussed techniques are applied to directed graphs in our practical implementation.

Firstly, at the data model level, it is necessary to distinguish between out-edges and in-edges. Each entry's value is separated into two sorted lists, representing the ID of out-neighbors and in-neighbors respectively. For instance, processing with the edge-based update entails inserting two entries: one with a key of u's ID and a value of v's ID in the out-neighbor list, and another with a key of v's ID and a value of u's ID in the in-neighbor list. For these two updates, Poly-LSM will adaptively select the optimal update method for each one, based on the threshold defined in Equation 7. Specifically, $d(u)$ in Equation 4 and 7 represents the total count of out-neighbors and in-neighbors of vertex u in directed graphs. The degree sketch now also represents this total count, meaning that whenever vertex u gains a new out-edge or in-edge, its degree sketch will increment by 1. The encoding method introduced in Section 3.4 can be conveniently extended to the directed graphs as well. Since the out-edge set and in-edge set are separately stored in the value of a certain entry while the vertex IDs are sorted inside each set. We can encode the out-edge set and in-edge set with the partitioned Elias-Fano workflow, respectively, and combine the results, maintaining the original layout of the value.

The primary focus of Poly-LSM is to facilitate the efficient storing and processing of graph structure information (i.e., vertices and the edges adjacent to each vertex), which is the foundation of graph storage systems. At the same time, the adaptable nature of the key-value paradigm allows for the seamless incorporation of property graphs. Specifically, in our implementation, we introduce an additional column family in RocksDB to store the mapping from vertices and edges to properties. The ID of the vertex or edge associated with the property key forms the key of the entry, which enables a fast property lookup by utilizing the prefix range lookup in the LSM-tree. User-defined secondary indexes can also be employed to speed up the query on vertices/edges with specific properties.

4 Aster: Poly-LSM Empowered Database

This section introduces Aster, a comprehensive graph database that facilitates extensive graph storage applications, empowering efficient fundamental operations to intricate graph tasks. It

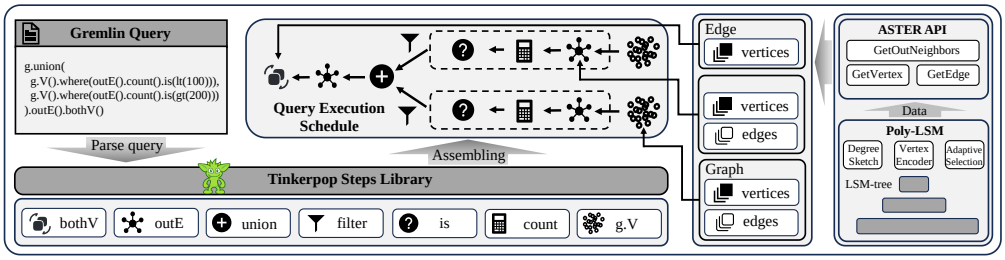


Fig. 5. The architecture of Aster.

enables users to perform queries through Gremlin with Poly-LSM as the storage backend. We will also introduce our distinctive graph traversal optimization for Poly-LSM and implementations of some necessary functions in graph database systems, such as transactions and multi-version concurrent control (MVCC).

Overview. The overall architecture of Aster is depicted in Figure 5 which mainly comprises three fundamental components including graph query interface, query executor, and storage engine. The graph query interface allows the users to precisely formulate various graph tasks on the graph data with straightforward statements. Additionally, with the integration of Gremlin language, this interface facilitates the convenient execution of these operations via the submission of Gremlin queries. Upon reception of a Gremlin query, the query executor converts it into a viable execution schedule to execute through the APIs provided by the storage engine. To this end, Aster leverages the prominent graph computation framework, TinkerPop [12], to systematically break down the query statements into individual operations using a detailed step library. These operations are subsequently assembled to construct a query execution schedule as Figure 5 presents. Then the schedule is executed sequentially with graph data accessed via the storage engine, the results of which are aggregated and returned to the user.

Poly-LSM Boosted Query Execution. As shown in Figure 5, a streaming computation graph is constructed when a Gremlin query is executed, which can be decomposed into a sequence of operations on graph data defined in the Tinkerpop step library. The execution of these operations depends on the interaction between the query executor and the underlying storage engine that manages the graph data. In this setup, Aster leverages Poly-LSM as its storage engine to ensure robust and efficient performance. Poly-LSM offers various interfaces, such as *AddVertex* and *GetOutNeighbors*, to facilitate the execution of these operations, thus enabling flexible manipulation of graph data and supporting various query execution plans for Aster. Furthermore, we tailor the Tinkerpop step library to our storage engine by implementing the *Vertex* and *Edge* interfaces. These interfaces access graph data through Poly-LSM, allowing the vertices and edges stored in Poly-LSM to be transformed into elements that can be processed by the computation graph, thereby completing the query execution process.

Query Optimization for Poly-LSM. In addition to the specialized Aster interfaces, there are opportunities to further optimize query execution by leveraging the characteristics of the LSM-based storage engine. Specifically, Aster can utilize efficient range scans in the LSM-tree to avoid expensive random disk reads during a full graph scan (i.e., *g.V()*). Furthermore, the retrieval of properties and neighbors for a vertex or edge is deferred until necessary. Until then, Poly-LSM only verifies the existence of the vertex or edge and creates a placeholder with the appropriate ID.

Transaction and MVCC. Poly-LSM is built on top of RocksDB [32], which supports both pessimistic and optimistic transaction control. Given that write conflicts on the same key are rare in graph database systems, we employ the optimistic transaction policy by default. In this approach,

conflict detection occurs when a transaction is committed, and the transaction will fail if there are conflicts with other writes. Read-write conflicts can also be escalated to write-write conflicts using the *GetForUpdate* interface. To ensure repeatable reads, Poly-LSM assigns a unique timestamp to each transaction and maintains it inline for each update of a graph element. Users can specify a snapshot with a timestamp using the *GetSnapshot* interface to obtain elements that are not newer than the specified timestamp. Poly-LSM ensures that elements visible to this snapshot are not reclaimed by compaction, allowing multi-version elements to coexist in the same LSM-tree.

Running Example. When a Gremlin query is issued to print vertices with out-edge counts less than 100 or greater than 200, Aster parses it into a bunch of elements, including graph components as vertexes and operations like edge lookups, using the Tinkerpop framework. These components are organized into a two-branch execution graph, where each branch retrieves vertices from Poly-LSM using the *GetOutNeighbors* interface and identifies vertices that meet the respective conditions. The results from both branches are then merged into a single set. Finally, the two vertices connected by the element's outgoing edges are returned using the *GetEdge* and *GetVertex* interfaces.

5 Related Works

Graph Database Systems. Over the past decade, extensive works have emerged on graph database research. Feng *et al.* designed Kùzu [35] to optimize the join efficiency of graph database systems, which is further enhanced by Arroyuelo *et al.* [13]. TED [42], VisualNeo [43], and VINCENT [44] improve the subgraph enumeration and subgraph mining efficacy. Luo *et al.* [62] improved the performance of vertex-centric graph systems for large-scale distributed graph processing. CG-graph [21] and MiniGraph [98] accelerate graph processing with CPU-GPU cooperative scheme and multi-core parallelism. These approaches focus on graph processing tasks, specifically targeting the execution management and scheduling of complex graph algorithms [37, 85], instead of enhancing graph storage, thus being orthogonal to our work. Additionally, since we focus on the generic graph database, works focusing on specific tasks are not closely related to ours, such as vector data storage [87], logic bugs identification [99], graph databased outsource [15], scalable second-order random walk [55] and temporal applications [39].

The research on on-disk graph database for large-scale graphs [36, 50, 51, 54, 80] are more relevant to our work, thus the main-memory graph database [17, 27, 83] are not examined. GraphChi-DB [51] is a disk-based system that leverages LSM-trees for efficient computation on large-scale graphs. GraphChi-DB adopts the partitioned adjacency list data structure from its well-known predecessor, GraphChi [50], which divides the complete adjacency list of a graph into several partitions on disk to reduce write amplification. However, we determined not to include GraphChi-DB in our experiential evaluation for 3 reasons: (1) GraphChi-DB is more akin to a storage engine rather than a complete graph database. It lacks support for running graph algorithms through a graph query language and does not incur the overhead of query parsing and execution. Therefore, directly comparing it with graph database systems like Neo4j would be unfair. (2) It was developed using Java 7, a very outdated version of Java, and is not compatible with later Java versions. (3) Its index structure has high read amplification, which limits its practical efficiency. The reason is that a vertex has its corresponding slice in each partition, it incurs at least one I/O at each partition to lookup any vertex's neighbors.

Moreover, some graph processing systems tailor their designs to specific hardware characteristics and have presented impressive efficacy. For example, CAVE [70] leverages the parallelism of SSD-based storage to achieve exceptional concurrency performance, while other studies [38, 78, 97] address the challenge of slow random disk access, delivering sound results on traditional hard disk drives (HDDs). In contrast, our work focuses on a different task, graph storage optimization, with a

more general solution that applies to any secondary storage device, thus placing it in a distinct scope.

LSM-tree Optimization. LSM-tree is widely adopted as the storage backend for real-world key-value stores [19, 32, 40, 52, 53, 71]. Therefore, in recent years, there have been abundant studies that focused on optimizing LSM-trees in different aspects such as cost modeling [23, 24, 58, 65], novel hardwares [16, 41, 60, 94], and space efficiency [10, 26, 64, 76, 81]. To improve query efficiency, new filters [25, 56], data distribution conversion [95, 96], and spatial partitioning [29, 86] are introduced to reduce unnecessary data access. For update performance enhancement, various index structures [74, 82, 89] and memory allocation strategies [47, 61, 93] are proposed. Furthermore, due to the high adaptability and efficiency of the key-value paradigm, there is a trend toward applying LSM-trees in various other fields [11, 18, 48, 72, 75, 84]. LSM-RUM-tree [84] proposed an effective structure that combines LSM-tree and R-tree to index spatial data. ChainKV [18] leverage LSM-tree to store the blockchain of Ethereum systems. Almodaresi *et al.* [11] implemented an LSM-based system for answering sequence-level biological enquiries. Our work focuses on optimizing LSM-tree under the scenario of graph storage. To the best of our knowledge, this is a novel problem that has yet to be scrutinized in existing works. Additionally, some LSM-based storage systems, such as RocksDB, support the *Merge Operator* interface, which, as noted in [14, 88, 90], accumulates the values of two entries with the same key rather than obsoleting the older entry. This behavior is similar to how Poly-LSM handles delta entries and is utilized in our practical implementation.

6 Evaluation

This section presents experimental evaluations of Aster against mainstream graph database systems under various workloads and tasks. The results demonstrate that Aster exhibits exceptional competitiveness relative to the baselines, excelling across a spectrum of fundamental graph queries and complex graph algorithms.

6.1 Experimental Setup

Environment. Our experiments are conducted on a server with a 13th Gen Intel(R) Core(TM) i9-13900K CPU @ 4.0GHz processor, 128GB DDR4 main memory, and 2TB NVMe SSD, running 64-bit Ubuntu 20.04.4 LTS on an ext4 file system.

Implementation. We implement Poly-LSM based on RocksDB [32], a widely adopted LSM-based storage engine. We adhere to the default settings of RocksDB across most configurations, where $T = 10$ and $B = 4096$ bytes. The IDs of vertices are 64-bit integers. Furthermore, we set the bits-per-key of Bloom filters as 10 and the bits-per-vertex of degree sketch as 8. We implement Aster in JAVA, integrating the TinkerPop [12] framework and connecting to Poly-LSM using Java Native Interface (JNI).

Baselines. We compare Poly-LSM against existing LSM-tree structures for graph storage, specifically edge-based LSM-tree and vertex-based LSM-tree, designated as Edge-LSM and Vertex-LSM. In addition, we also include Poly-LSM without adaptive update mechanism, which trivially adopts either the delta update mechanism or the pivot update mechanism for all updates, designated as Delta-Poly and Pivot-Poly. These baselines are also built on top of RocksDB, similar to Poly-LSM. In addition, most entries in Pivot-Poly exist in the form of edge-based adjacency lists. As a result, the performance of Vertex-LSM and Pivot-Poly is nearly identical. Therefore, we only present the results for Vertex-LSM.

We perform a comprehensive comparison of Aster against various mainstream graph database systems that include Neo4j (v3.2), ArangoDB (v2.8), OrientDB (v2.2), SQLG (PostgreSQL v9.6), JanusGraph (v1.0), and NebulaGraph (v3.8). Neo4j and OrientDB employ traditional linked list

Scale	Graph	n	m	$\bar{d}$	Size
Moderate	DBLP [92]	317,080	1,049,866	3.31	131MB
	Twitch [79]	168,114	6,797,557	40.43	213MB
	LDBC Datagen* [57]	184,329	767,894	4.17	22MB
	Wiki-Talk	2,394,385	5,021,410	2.10	140MB
	Cit-Patents	3,774,768	16,518,947	4.38	351MB
Large	Wikipedia [57]	3,333,397	123,709,902	37.11	1.8GB
	Orkut [92]	3,072,441	234,370,166	76.28	1.3GB
	Freebase Large* [57]	28,408,172	31,475,362	1.11	1.4GB
Massive	Twitter [49]	41,652,230	1,202,513,046	57.74	24GB

Table 3. Datasets. n is the total number of vertices; m is the number of edges; $\bar{d}$ represents the average degree. The datasets with * are property graphs.

storage model while SQLG utilizes PostgreSQL [66] as its storage backend and represents a graph with relational models. JanusGraph utilizes a NoSQL storage engine BerkeleyDB [3] as its backend by default, and maintains an adjacent list for each vertex. ArangoDB represents graph elements as documents and links them by indexes. NebulaGraph employs an edge-based LSM-tree that stores each edge as an entry. These baselines all support the TinkerPop framework. To ensure fairness in evaluation, we use Gremlin as the graph query language for both Aster and all baseline databases, adhering to established practices in graph database benchmarking [57].

Graph Datasets. We employ five real graph datasets that are commonly used in graph processing and benchmarking to assess the effectiveness of our proposed methods in performing updates and lookups on graph structures, as detailed in Table 3. These datasets span across various scales. The moderate-scale dataset includes DBLP and Twitch which have hundreds of thousands of vertices and Twitch contains significantly more edges. Wikipedia and Orkut are large-scale datasets featuring millions of vertices and Orkut has higher average degree. Twitter is a massive-scale dataset with over a billion edges. In addition, LDBC Datagen and Freebase Large are property graphs from Lissandra's microbenchmark [57], used to evaluate the performance of property queries. Furthermore, we also evaluate the performance of graph algorithms in LDBC Graphalytics [46] with two LDBC-recommended datasets, Wiki-Talk and Cit-Patents. We load these graphs into a graph database by sequentially inserting all edges in our experiments.

Experiment Design. We evaluate the update and lookup performance of our proposed graph database, Aster by subjecting it to varied workloads encompassing different proportions of typical update and lookup operations, for comparison with the baseline graph database systems. Additionally, we assess the scalability of Aster across different-sized graph datasets like DBLP, Wikipedia, Orkut, and Twitter. For clarity, we focus on two representative operations: adding edges and retrieving neighbors. These operations represent common tasks in online graph database services. Furthermore, we generalize the workloads by evaluating various graph queries, including graph structure updates, graph traversal, and property queries. Lastly, while analytical workloads like full graph scans are not our primary focus, we also demonstrate Aster's potential in handling substantial graph scans. Moreover, we also compare our core design, Poly-LSM, against a variety of LSM-tree-based baselines across diverse workloads to scrutinize its efficacy. We further validate the effectiveness of our cost model by comparing actual performance with our theoretical predictions.

6.2 System Performance

Aster outperforms mainstream graph database systems in diverse workloads and demonstrates better scalability. In Figure 6, we compare the throughput of Aster against real-world graph database systems across a spectrum of workloads, ranging from write-heavy tasks (10%

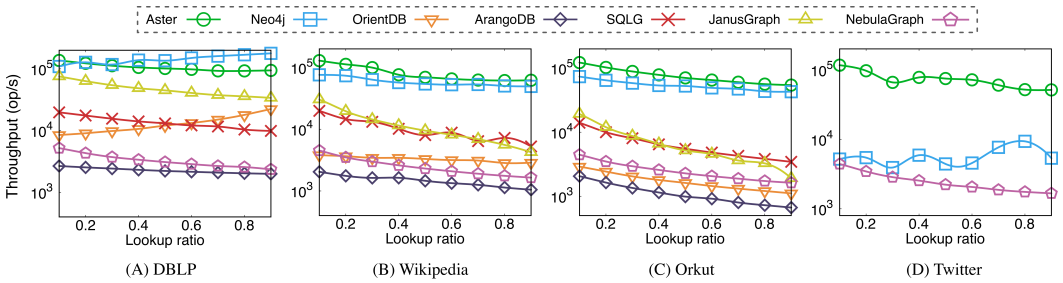


Fig. 6. Aster demonstrates strong efficiency and scalability across various workloads and dominates all baselines on large-scale and massive-scale graph datasets.

lookups and 90% updates) to read-heavy tasks (90% lookups and 10% updates), on various graph datasets from moderate-scale to massive-scale. On DBLP, Neo4j and Aster demonstrates strong performance, surpassing other graph database systems, while Aster shows slightly slower performance with heavy lookups. The slowdown of Aster is due to the relatively small scale of the DBLP dataset, causing the LSM-tree in Poly-LSM to have fewer levels. Consequently, the immutable table in Level 0 would markedly lower the overall read performance. In contrast, Neo4j uses link lists as its backbone, whereas querying neighbors in a smaller data scale involves fewer random jumps, resulting in superior throughput capabilities. However, as the graph size increases, the limitations of the linked list become evident. Specifically, both Neo4j and OrientDB use an adjacent linked list storage model, where each vertex maintains a head pointer to its linked list of edges. Since the linked list is not stored contiguously on disk, retrieving all edges for a vertex requires multiple I/O operations. This issue becomes more significant for high-degree vertices, leading to performance degradation on dense datasets such as Wikipedia and Orkut, as shown in Figure 6(B) to (C).

The relational-table-based baseline, SQLG, is slower than linked-list-based baselines due to the significant overhead involved in traversal using join operations, which scans through the vertex table and edge table incurring substantial I/O cost. This overhead results in performance that is roughly 10 to 100 times slower than Neo4j. JanusGraph's neighbor retrieval is inefficient because BerkeleyDB stores pointers to key-value entries in its leaf nodes, leading to significant random disk access when retrieving multiple edges. Consequently, JanusGraph shares the same drawbacks as linked-list-based storage and experiences performance degradation as the graph scale increases, as shown in Figure 6(B) and (C). ArangoDB does not perform well in this experiment because it uses traditional document mapping storage, which leads to significant write amplification. Moreover, the necessity for frequent random accesses to vertices and edges documents also incurs substantial I/O costs.

Benefiting from the use of the LSM-tree structure, NebulaGraph exhibits substantial scalability, maintaining robust performance on the Twitter dataset without the significant degradation seen in many other baselines. Nevertheless, its performance on moderate-scale datasets is relatively mediocre. This is because Nebula utilizes an edge-based storage model, enabling lightweight updates but incurring significant overhead in neighbor retrieval, as discussed in Section 3.1. In addition, it is important to note that Nebula primarily optimizes for horizontal distribution, therefore its performance is less competitive on a single machine.

In contrast, Aster maintains satisfactory performance and outperforms all other databases on moderate and large-scale graphs. This is because the LSM-tree reduces write amplification by buffering incoming updates and flushing them sequentially. Additionally, the adaptive selection scheme ensures performance stability as the workload shifts. Moreover, the performance does not degrade with increasing graph scale, as shown in Figure 6 (B) to (C). This stability is attributed to the

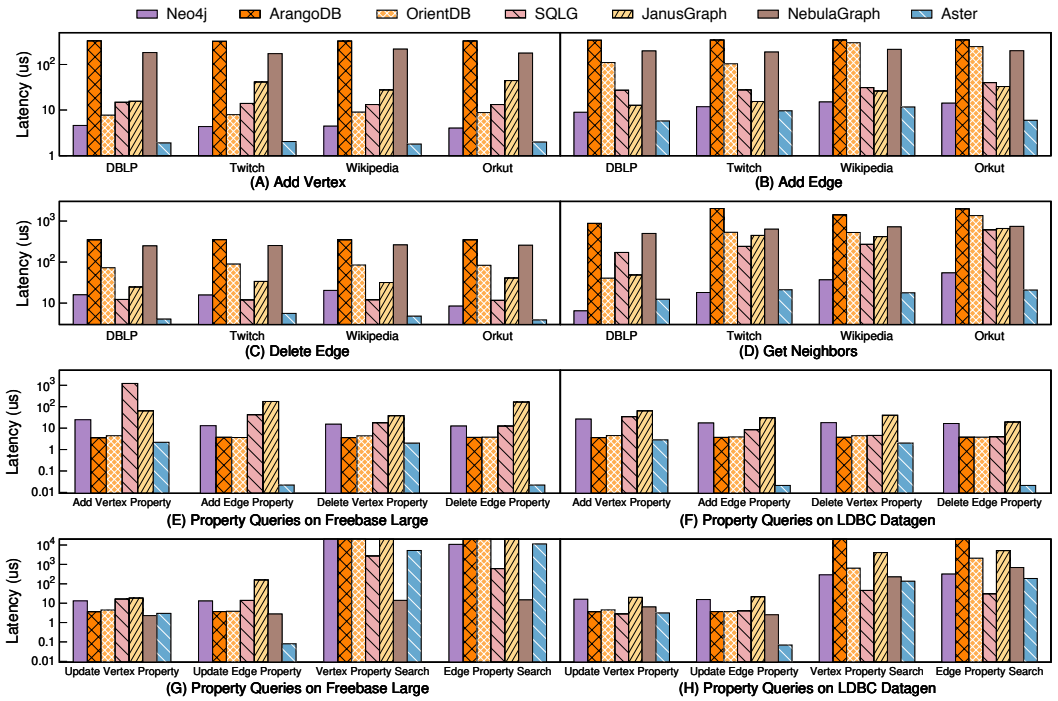


Fig. 7. Time required for processing graph structure update, traversal, and property queries.

System	Add vertex	Add edge	Delete edge	Get neighbor
Neo4j	4.21us	171.47us	10.97us	1302.51us
ASTER	1.79us	6.03us	3.87us	20.67us
NebulaGraph	238.14us	218.85us	220.04us	844.17us

Table 4. Latency of various graph queries on Twitter.

fact that the overhead of edge retrieval and updates is not linearly proportional to the graph scale, resulting in a less significant increase in I/O as the graph scale grows, as analyzed in Section 3.3.

To further compare the scalability of these methods, we evaluate them on Twitter, a massive billion-scale dataset. With the data loading time limited to two days, all baselines except for Neo4j and ASTER failed to complete the data loading process. In this experiment, as shown in Figure 6 (D), the linked-list-based baseline, Neo4j, experienced continuous deterioration, with its throughput dropping nearly tenfold. In contrast, ASTER maintained the same magnitude of throughput as observed in Figure 6 (A) to (D).

ASTER consistently delivers efficient performance encompassing various graph operations. In Figure 7, we provide a comparative study of ASTER and all baseline models by reporting the average latency of executing a series of graph queries including fundamental graph traversal queries, graph structure manipulations and property-related graph queries across all datasets. Our experiments encompass the most commonly used operations in graph database systems, including adding vertex, adding edge, removing edge, getting neighbors, adding property, removing property, updating property, and getting edges/vertices according to given properties. We report results from all moderate-scale and large-scale datasets in Figure 7. All baselines apart from Neo4j and

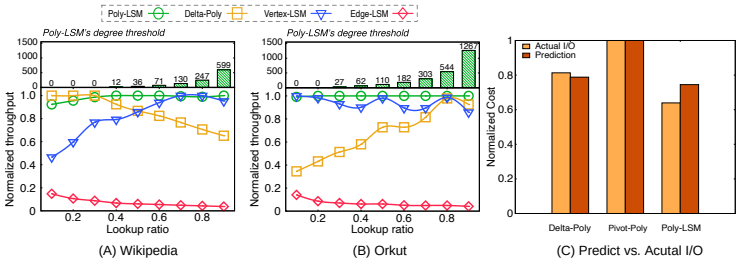


Table 5. The statistics of Aster on Wikipedia under 50% lookup ratio workload.

Statistics of Poly-LSM	
CPU Time of Adaptive Mechanism	0.723 us / op
Morris Counter Read	0.107 us / op
Read Block Latency	10345.114 ms
Write Block Latency	572.054 ms
Number of Vertex-based Updates	879286
Number of Edge-based Updates	1120714
Avg. Degree of Vertex-based Update	3.58
Avg. Degree of Edge-based Update	118.36

Fig. 8. (A) and (B) shows Poly-LSM exhibits strong robustness as the workload changes and outperforms other LSM-tree-based baselines. (C) indicates the cost model of Poly-LSM accurately predicts the I/O overhead.

NebulaGraph time out while loading the massive-scale Twitter dataset. Therefore, we report the results for Twitter separately in Table 4.

As illustrated in Figure 7 (A) ~ (C), Aster consistently achieves the lowest latency for all kinds of graph structure updates across various datasets, including the addition and removal of vertices and edges, benefiting from the efficient update of Poly-LSM. For the get neighbor operation in Figure 7 (D), Aster's performance slightly trails Neo4j on moderate-scale datasets. In contrast, on large-scale datasets, Aster outperforms all baselines on these queries, demonstrating superior scalability for lookup-related queries. The reason is that, as the data scale expands, the random access required by the linked-list-based structure increases significantly. In contrast, the number of read I/O operations for Aster grows more gradually and can be bounded by the number of levels in Poly-LSM.

Figure 7 (E) ~ (H) showcases the evaluation of graph property queries on our property graph datasets Freebase Large and LDBC Datagen. [5] The results indicate that our method excels in updates, such as adding, updating, and removing properties, outperforming all other baselines. Additionally, we demonstrate acceptable performance in searching vertices and edges by property, which is slightly slower than SQLG and Nebula. This is because both SQLG and Nebula stores edges or vertices in a continuous manner and filter them using a simple "where" clause, which avoids the need for join or random I/O operations. By employing appropriate indexes, their structures retrieve results relatively quickly.

Poly-LSM excels over other LSM-based graph storage designs. Figure 8 (A) and (B) compared Poly-LSM against various LSM-tree-based baselines including edge-based LSM-tree (Edge-LSM), vertex-based LSM-tree (Vertex-LSM), and Poly-LSM using only delta updates (Delta-Poly). We present the normalized throughput on our two large-scale graph datasets, Wikipedia and Orkut, under workloads consistent with the precedent experiment in Figure 6. In addition, the top subfigures display the degree thresholds for Poly-LSM's adaptive update mechanism under each corresponding workload, as specified in Equation 8.

Vertex-LSM performs well for read-heavy scenarios but struggles under intensive graph updates since the read-and-modify scheme increases the cost of update while reducing the cost of lookup. In contrast, the performance of Delta-Poly and Edge-LSM noticeably declines as the lookup ratio increases. Despite both employing edge-based update methods, the hybrid layout of Poly-LSM enables Delta-Poly to dominate Edge-LSM across all workloads, because the lookup performance of Edge-LSM is significantly hindered by its edge-based storage layout. Meanwhile, Poly-LSM outperforms all baselines that maintain optimal or near-optimal throughput across all workloads

[5]NebulaGraph requires a fixed schema for each vertex and edge, which prevents it from dynamically adding or deleting individual properties for a vertex or edge. Instead, it only supports updating all properties of an instance as a whole. As a result, NebulaGraph was not included in the results for Figure 7 (E) and (F).

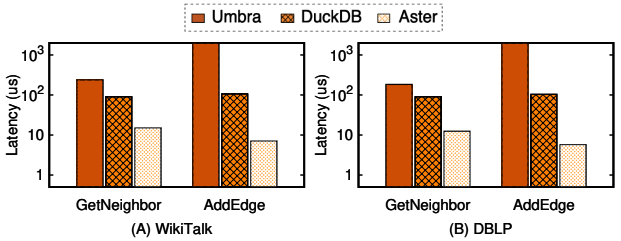


Fig. 9. In-memory comparison with memory-centric systems.

by adaptively selecting the best policy for each updated vertex, which has been proven to be theoretically global optimal in Section 3.3.

For instance, in the experiment on Wikipedia under a 50% lookup ratio, Poly-LSM outperforms both Delta-Poly and Pivot-Poly by intelligently selecting either of them for each updated vertex based on its specific degree. The detailed statistics and the predictions of our model are shown in Figure 8 (C) and Table 5. The predicted cost in Figure 8 (C) closely matches the actual cost, indicating that our model accurately captures the cost of different update methods. Additionally, we provide a detailed latency breakdown in Table 5 to support a more comprehensive evaluation. As shown, the edge count estimation and adaptive mechanism determination take just 0.1 µs and 0.7 µs, respectively, which contributes a small portion of the overall overhead. Hence we can achieve obviously enhanced system performance with minimal and manageable additional CPU overhead which further validates the effectiveness of our method. Moreover, the selection of pivot or delta updates follows our expectations, with pivot updates being applied to vertices with higher degrees and delta updates to those with lower degrees.

Competitive performance on small dataset. To demonstrate the robustness of Aster on small datasets that fit into memory, we conducted a case study evaluating performance on two datasets, WikiTalk and DBLP, against two memory-centric storage systems, DuckDB [73] and Umbra [68]. The results in Figure 9 indicate that Aster outperforms both baselines in the basic graph traversal operation, *GetNeighbors*, and the graph update operation, *AddEdge*. The main reason is that DuckDB and Umbra are not graph-structure-aware systems designed specifically for graph storage, like Aster. For example, DuckDB processes analytical SQL queries efficiently by employing a relational data model, yet this model is less effective for graph traversal. Additionally, Umbra's use of sophisticated SQL compilation techniques, Just-In-Time (JIT) compilation, which convert queries into lightweight Intermediate Representation (IR) and then compile them to executable binary code, provides significant performance benefits for complex queries on large datasets but introduces overhead in simpler queries and smaller datasets, leading to weaker performance in this experiment.

Poly-LSM achieves desirable performance on LDBC Graphalytics benchmark. In Table 6, we evaluate Aster and other graph database baselines using the LDBC Graphalytics benchmark [46]. Although substantial analytical algorithms are typically not the workload for online graph database systems and are usually conducted on OLAP graph processing systems such as GraphScope [34], we still aim to demonstrate the potential of Poly-LSM to adapt to analytical workloads. We test five algorithms specified in the LDBC benchmark on two LDBC datasets, including PageRank, Community Detection using Label Propagation (CDLP), Weakly Connected Component (WCC), Single-Source Shortest Path (SSSP), and Breadth-First Search (BFS). In addition, we also compare these graph databases with two representative graph processing systems, GridGraph [97] and Mosaic [63], to demonstrate the gap in processing analytical workloads for graph databases. The time limit for conducting these algorithms is set to 2 hours. As shown in Table 6, Aster and

Dataset	Cit-Patents (**n**=3,774,768, **m**=16,518,947)					Wiki-Talk (**n**=2,394,385, **m**=5,021,410)				
Algorithm	PageRank	CDLP	WCC	SSSP	BFS	PageRank	CDLP	WCC	SSSP	BFS
Aster	491.886	**334.991**	1187.583	**0.013**	**0.019**	287.229	**115.912**	267.540	**0.118**	11.638
Neo4j	**294.540**	479.899	**684.763**	0.038	0.046	**127.642**	176.790	**117.614**	0.300	**9.726**
OrientDB	1604.027	3211.459	5246.498	0.053	0.051	441.791	910.428	718.166	37.536	36.372
SQLG	2626.018	5964.497	timeout	1.290	1.424	1198.875	3651.953	1862.763	3.244	5676.086
JanusGraph	1593.294	1912.659	3029.494	0.302	0.419	563.823	657.349	480.889	0.687	timeout
ArangoDB	timeout	timeout	timeout	2.257	2.272	timeout	timeout	timeout	1052.341	1039.346
GridGraph	0.570	7.430	0.410	0.380	0.360	0.300	3.210	0.130	0.040	0.250
Mosaic	19.40	127.68	3.720	1.690	1.590	3.227	32.10	2.019	3.994	1.656

Table 6. This table presents the latency (in seconds) of Aster compared to various baselines in the LDBC Graphalytics Benchmark. The baselines are categorized into two groups: graph databases (top) and graph processing systems (bottom). While Aster is slower than the graph processing systems, it demonstrates solid performance relative to other graph databases.

Neo4j outperform other baselines remarkably. Generally, Neo4j performs better on algorithms requiring full graph scans, such as PageRank and WCC, while Aster shows strengths in algorithms that require local graph traversal, such as BFS and SSSP. Notably, both graph processing systems outperform all graph databases by a factor of 10 to 100, particularly in PageRank, CDLP[6], and WCC, which require extensive full graph traversal. However, for BFS and SSSP, certain graph databases like Aster and Neo4j outperform Mosaic and GridGraph on the Cit-Patents dataset. This is likely due to the latter's reliance on substantial preprocessing and concurrent scheduling, which can be time-consuming and less effective for simpler queries. This result highlights the need for graph databases to improve their support for full graph traversal, suggesting a promising direction for future development.

7 Conclusion

We introduce Poly-LSM, a high-performance LSM-tree-based storage engine tailored for large-scale graphs. It achieves commendable performance in both update and lookup operations, leveraging a hybrid graph storage model, adaptive edge update mechanism, and refined entry encoding strategy. Based on Poly-LSM, we propose a robust and versatile graph database, Aster, that outperforms mainstream graph databases in terms of graph structure storage.

Acknowledgments

This research is supported by NTU-NAP startup grant (022029-00001). We thank the anonymous reviewers for their valuable suggestions.

References

[1] -. Aster Technical Report. https://sites.google.com/view/aster-technical-report.
[2] 2024. ArangoDB. https://www.arangodb.com/.
[3] 2024. Berkeley DB Java Edition Architecture. https://www.oracle.com/technetwork/database/berkeleydb/learnmore/bdb-je-architecture-whitepaper-366830.pdf.
[4] 2024. JanusGraph. https://janusgraph.org/.
[5] 2024. MySQL. https://www.mysql.com/.
[6] 2024. Neo4j. https://neo4j.com/.
[7] 2024. OrientDB. https://github.com/orientechnologies/orientdb.
[8] 2024. Sparsity Technologies, Sparksee. http://www.sparsity-technologies.com/.
[9] 2024. SQLG. http://www.sqlg.org/.

[6]Note that CDLP requires tracking the number of labels for each vertex's neighbors, which is not well-suited to the edge-centric co-processing model used in Mosaic and GridGraph, resulting in relatively slower performance.

[10] Wail Y Alkowaileet, Sattam Alsubaiee, and Michael J Carey. 2019. An LSM-based Tuple Compaction Framework for Apache AsterixDB (Extended Version). *arXiv preprint arXiv:1910.08185* (2019).

[11] Fatemeh Almodaresi, Jamshed Khan, Sergey Madaminov, Prashant Pandey, Michael Ferdman, Rob Johnson, and Rob Patro. 2021. An incrementally updatable and scalable system for large-scale sequence search using LSM trees. *BioRxiv* (2021), 2021–02.

[12] Apache. 2024. TinkerPop. https://tinkerpop.apache.org/.

[13] Diego Arroyuelo, Benjamin Bustos, Adrián Gómez-Brandón, Aidan Hogan, Gonzalo Navarro, and Juan Reutter. 2024. Worst-Case-Optimal Similarity Joins on Graph Databases. *Proceedings of the ACM on Management of Data* 2, 1 (2024), 1–26.

[14] Zhichao Cao, Siying Dong, Sagar Vemuri, and David HC Du. 2020. Characterizing, Modeling, and Benchmarking RocksDB Key-Value Workloads at Facebook. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*. 209–223.

[15] Javad Ghareh Chamani, Ioannis Demertzis, Dimitrios Papadopoulos, Charalampos Papamanthou, and Rasool Jalili. 2023. GraphOS: Towards Oblivious Graph Processing. *Proceedings of the VLDB Endowment* 16, 13 (2023), 4324–4338.

[16] Helen HW Chan, Chieh-Jan Mike Liang, Yongkun Li, Wenjia He, Patrick PC Lee, Lianjie Zhu, Yaozu Dong, Yinlong Xu, Yu Xu, Jin Jiang, et al. 2018. HashKV: Enabling Efficient Updates in KV Storage via Hashing. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. 1007–1019.

[17] Hongzhi Chen, Changji Li, Chenguang Zheng, Chenghuan Huang, Juncheng Fang, James Cheng, and Jian Zhang. 2022. G-tran: a high performance distributed graph database with a decentralized architecture. *Proceedings of the VLDB Endowment* 15, 11 (2022), 2545–2558.

[18] Zehao Chen, Bingzhe Li, Xiaojun Cai, Zhiping Jia, Lei Ju, Zili Shao, and Zhaoyan Shen. 2023. ChainKV: A Semantics-Aware Key-Value Store for Ethereum System. *Proceedings of the ACM on Management of Data* 1, 4 (2023), 1–23.

[19] Source Code. 2024. WiredTiger. https://github.com/wiredtiger/wiredtiger.

[20] Miklós Csűrös. 2010. Approximate counting with a floating-point counter. In *International Computing and Combinatorics Conference*. Springer, 358–367.

[21] Pengjie Cui, Haotian Liu, Bo Tang, and Ye Yuan. 2024. CGgraph: An Ultra-fast Graph Processing System on Modern Commodity CPU-GPU Co-processor. *Proc. VLDB Endow.* 17, 6 (2024), 1405–1417. https://www.vldb.org/pvldb/vol17/p1405-yuan.pdf

[22] Ali Davoudian, Liu Chen, and Mengchi Liu. 2018. A survey on NoSQL stores. *ACM Computing Surveys (CSUR)* 51, 2 (2018), 1–43.

[23] Niv Dayan, Manos Athanassoulis, and Stratos Idreos. 2017. Monkey: Optimal navigable key-value store. In *Proceedings of the 2017 ACM International Conference on Management of Data*. 79–94.

[24] Niv Dayan and Stratos Idreos. 2018. Dostoevsky: Better Space-Time Trade-Offs for LSM-Tree Based Key-Value Stores via Adaptive Removal of Superfluous Merging. In *Proceedings of the 2018 International Conference on Management of Data* (Houston, TX, USA) *(SIGMOD '18)*. Association for Computing Machinery, New York, NY, USA, 505–520. https://doi.org/10.1145/3183713.3196927

[25] Niv Dayan and Moshe Twitto. 2021. Chucky: A Succinct Cuckoo Filter for LSM-Tree. In *Proceedings of the 2021 International Conference on Management of Data*. 365–378.

[26] Niv Dayan, Tamar Weiss, Shmuel Dashevsky, Michael Pan, Edward Bortnikov, and Moshe Twitto. 2022. Spooky: granulating LSM-tree compactions correctly. *Proceedings of the VLDB Endowment* 15, 11 (2022), 3071–3084.

[27] 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.

[28] DGraph. 2024. DGraph. https://dgraph.io/.

[29] Ahmed Eldawy, Vagelis Hristidis, Saheli Ghosh, Majid Saeedan, Akil Sevim, AB Siddique, Samriddhi Singla, Ganesh Sivaram, Tin Vu, and Yaming Zhang. 2021. Beast: Scalable exploratory analytics on spatio-temporal data. In *Proceedings of the 30th ACM International Conference on Information & Knowledge Management*. 3796–3807.

[30] Facebook. 2024. Merge Operator. https://github.com/facebook/rocksdb/wiki/Merge-Operator.

[31] Facebook. 2024. MyRocks. https://myrocks.io/.

[32] Facebook. 2024. RocksDB. https://github.com/facebook/rocksdb.

[33] Wenfei Fan. 2022. Big graphs: challenges and opportunities. *Proceedings of the VLDB Endowment* 15, 12 (2022), 3782–3797.

[34] Wenfei Fan, Tao He, Longbin Lai, Xue Li, Yong Li, Zhao Li, Zhengping Qian, Chao Tian, Lei Wang, Jingbo Xu, et al. 2021. GraphScope: a unified engine for big graph processing. *Proceedings of the VLDB Endowment* 14, 12 (2021), 2879–2892.

[35] Xiyang Feng, Guodong Jin, Ziyi Chen, Chang Liu, and Semih Salihoğlu. 2023. KÙZU graph database management system. In *The Conference on Innovative Data Systems Research*.

[36] Scott Freitas, Yuxiao Dong, Joshua Neil, and Duen Horng Chau. 2020. A large-scale database for graph representation learning. *arXiv preprint arXiv:2011.07682* (2020).
[37] 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.
[38] Wook-Shin Han, Sangyeon Lee, Kyungyeol Park, Jeong-Hoon Lee, Min-Soo Kim, Jinha Kim, and Hwanjo Yu. 2013. TurboGraph: a fast parallel graph engine handling billion-scale graphs in a single PC. In *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining*. 77–85.
[39] Jiamin Hou, Zhanhao Zhao, Zhouyu Wang, Wei Lu, Guodong Jin, Dong Wen, and Xiaoyong Du. 2024. AeonG: An Efficient Built-in Temporal Support in Graph Databases. arXiv:2304.12212 [cs.DB]
[40] Gui Huang, Xuntao Cheng, Jianying Wang, Yujie Wang, Dengcheng He, Tieying Zhang, Feifei Li, Sheng Wang, Wei Cao, and Qiang Li. 2019. X-Engine: An optimized storage engine for large-scale E-commerce transaction processing. In *Proceedings of the 2019 International Conference on Management of Data*. 651–665.
[41] Haoyu Huang and Shahram Ghandeharizadeh. 2021. Nova-LSM: a distributed, component-based LSM-tree key-value store. In *Proceedings of the 2021 International Conference on Management of Data*. 749–763.
[42] Kai Huang, Haibo Hu, Qingqing Ye, Kai Tian, Bolong Zheng, and Xiaofang Zhou. 2023. TED: Towards Discovering Top-k Edge-Diversified Patterns in a Graph Database. *Proceedings of the ACM on Management of Data* 1, 1 (2023), 1–26.
[43] Kai Huang, Houdong Liang, Chongchong Yao, Xi Zhao, Yue Cui, Yao Tian, Ruiyuan Zhang, and Xiaofang Zhou. 2023. VisualNeo: Bridging the Gap between Visual Query Interfaces and Graph Query Engines. *Proceedings of the VLDB Endowment* 16, 12 (2023), 4010–4013.
[44] Kai Huang, Qingqing Ye, Jing Zhao, Xi Zhao, Haibo Hu, and Xiaofang Zhou. 2022. VINCENT: towards efficient exploratory subgraph search in graph databases. *Proceedings of the VLDB Endowment* 15, 12 (2022), 3634–3637.
[45] Andy Huynh, Harshal Chaudhari, Evimaria Terzi, and Manos Athanassoulis. 2021. Endure: A Robust Tuning Paradigm for LSM Trees Under Workload Uncertainty. *arXiv preprint arXiv:2110.13801* (2021).
[46] Alexandru Iosup, Tim Hegeman, Wing Lung Ngai, Stijn Heldens, Arnau Prat-Pérez, Thomas Manhardto, Hassan Chafio, Mihai Capotă, Narayanan Sundaram, Michael Anderson, Ilie Gabriel Tănase, Yinglong Xia, Lifeng Nai, and Peter Boncz. 2016. LDBC graphalytics: a benchmark for large-scale graph analysis on parallel and distributed platforms. *Proc. VLDB Endow.* 9, 13 (sep 2016), 1317–1328. https://doi.org/10.14778/3007263.3007270
[47] Taewoo Kim, Alexander Behm, Michael Blow, Vinayak Borkar, Yingyi Bu, Michael J Carey, Murtadha Hubail, Shiva Jahangiri, Jianfeng Jia, Chen Li, et al. 2020. Robust and efficient memory management in Apache AsterixDB. *Software: Practice and Experience* 50, 7 (2020), 1114–1151.
[48] Young-Seok Kim, Taewoo Kim, Michael J Carey, and Chen Li. 2017. A comparative study of log-structured merge-tree-based spatial indexes for big data. In *2017 IEEE 33rd International Conference on Data Engineering (ICDE)*. IEEE, 147–150.
[49] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. 2010. What is Twitter, a social network or a news media?. In *Proceedings of the 19th international conference on World wide web*. 591–600.
[50] Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. 2012. GraphChi:Large-Scale graph computation on just a PC. In *10th USENIX symposium on operating systems design and implementation (OSDI 12)*. 31–46.
[51] Aapo Kyrola and Carlos Guestrin. 2014. GraphChi-DB: Simple design for a scalable graph database system–on just a PC. *arXiv preprint arXiv:1403.0701* (2014).
[52] Cockroach Labs. 2024. CockroachDB. https://github.com/cockroachdb/cockroach.
[53] Avinash Lakshman and Prashant Malik. 2010. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review* 44, 2 (2010), 35–40.
[54] 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.
[55] Hongzheng Li, Yingxia Shao, Junping Du, Bin Cui, and Lei Chen. 2022. An I/O-efficient disk-based graph system for scalable second-order random walk of large graphs. *Proceedings of the VLDB Endowment* 15, 8 (2022), 1619–1631.
[56] Meng Li, Deyi Chen, Haipeng Dai, Rongbiao Xie, Siqiang Luo, Rong Gu, Tong Yang, and Guihai Chen. 2022. Seesaw Counting Filter: An Efficient Guardian for Vulnerable Negative Keys During Dynamic Filtering. In *Proceedings of the ACM Web Conference 2022*. 2759–2767.
[57] Matteo Lissandrini, Martin Brugnara, and Yannis Velegrakis. 2018. Beyond macrobenchmarks: microbenchmark-based graph database evaluation. *Proc. VLDB Endow.* 12, 4 (dec 2018), 390–403. https://doi.org/10.14778/3297753.3297759
[58] Junfeng Liu, Fan Wang, Dingheng Mo, and Siqiang Luo. 2024. Structural Designs Meet Optimality: Exploring Optimized LSM-tree Structures in A Colossal Configuration Space. *Proceedings of the ACM on Management of Data* 2, 3 (2024), 1–26.
[59] Weifeng Liu and Brian Vinter. 2015. CSR5: An Efficient Storage Format for Cross-Platform Sparse Matrix-Vector Multiplication. In *Proceedings of the 29th ACM on International Conference on Supercomputing* (Newport Beach, California, USA)

(ICS '15). Association for Computing Machinery, New York, NY, USA, 339–350. https://doi.org/10.1145/2751205.2751209
[60] Lanyue Lu, Thanumalayan Sankaranarayana Pillai, Hariharan Gopalakrishnan, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. 2017. Wisckey: Separating keys from values in ssd-conscious storage. *ACM Transactions on Storage (TOS)* 13, 1 (2017), 1–28.
[61] Chen Luo and Michael J Carey. 2020. Breaking down memory walls: adaptive memory management in LSM-based storage systems. *Proceedings of the VLDB Endowment* 14, 3 (2020), 241–254.
[62] Siqiang Luo, Zichen Zhu, Xiaokui Xiao, Yin Yang, Chunbo Li, and Ben Kao. 2023. Multi-Task Processing in Vertex-Centric Graph Systems: Evaluations and Insights.. In *EDBT*. 247–259.
[63] Steffen Maass, Changwoo Min, Sanidhya Kashyap, Woonhak Kang, Mohan Kumar, and Taesoo Kim. 2017. Mosaic: Processing a Trillion-Edge Graph on a Single Machine. In *Proceedings of the Twelfth European Conference on Computer Systems* (Belgrade, Serbia) *(EuroSys '17)*. Association for Computing Machinery, New York, NY, USA, 527–543. https://doi.org/10.1145/3064176.3064191
[64] Qizhong Mao, Mohiuddin Abdul Qader, and Vagelis Hristidis. 2020. Comprehensive comparison of LSM architectures for spatial data. In *2020 IEEE International Conference on Big Data (Big Data)*. IEEE, 455–460.
[65] Dingheng Mo, Fanchao Chen, Siqiang Luo, and Caihua Shan. 2023. Learning to Optimize LSM-trees: Towards A Reinforcement Learning based Key-Value Store for Dynamic Workloads. *Proceedings of the ACM on Management of Data* 1, 3 (2023), 1–25.
[66] Bruce Momjian. 2001. *PostgreSQL: introduction and concepts*. Vol. 192. Addison-Wesley New York.
[67] Robert Morris. 1978. Counting large numbers of events in small registers. *Commun. ACM* 21, 10 (1978), 840–842.
[68] Thomas Neumann and Michael J Freitag. 2020. Umbra: A Disk-Based System with In-Memory Performance.. In *CIDR*, Vol. 20. 29.
[69] Giuseppe Ottaviano and Rossano Venturini. 2014. Partitioned elias-fano indexes. In *Proceedings of the 37th international ACM SIGIR conference on Research & development in information retrieval*. 273–282.
[70] Tarikul Islam Papon, Taishan Chen, Shuo Zhang, and Manos Athanassoulis. 2024. CAVE: Concurrency-Aware Graph Processing on SSDs. *Proceedings of the ACM on Management of Data* 2, 3 (2024), 1–26.
[71] PingCAP. 2024. TiDB. https://github.com/pingcap/tidb.
[72] Mohiuddin Abdul Qader, Shiwen Cheng, and Vagelis Hristidis. 2018. A comparative study of secondary indexing techniques in LSM-based NoSQL databases. In *Proceedings of the 2018 International Conference on Management of Data*. 551–566.
[73] Mark Raasveldt and Hannes Mühleisen. 2019. Duckdb: an embeddable analytical database. In *Proceedings of the 2019 International Conference on Management of Data*. 1981–1984.
[74] Pandian Raju, Rohan Kadekodi, Vijay Chidambaram, and Ittai Abraham. 2017. Pebblesdb: Building key-value stores using fragmented log-structured merge trees. In *Proceedings of the 26th Symposium on Operating Systems Principles*. 497–514.
[75] Pandian Raju, Soujanya Ponnapalli, Evan Kaminsky, Gilad Oved, Zachary Keener, Vijay Chidambaram, and Ittai Abraham. 2018. mLSM: Making authenticated storage faster in ethereum. In *10th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 18)*.
[76] Kai Ren, Qing Zheng, Joy Arulraj, and Garth Gibson. 2017. SlimDB: A space-efficient key-value storage engine for semi-sorted data. *Proceedings of the VLDB Endowment* 10, 13 (2017), 2037–2048.
[77] Marko A Rodriguez. 2015. The gremlin graph traversal machine and language (invited talk). In *Proceedings of the 15th Symposium on Database Programming Languages*. 1–10.
[78] Amitabha Roy, Ivo Mihailovic, and Willy Zwaenepoel. 2013. X-stream: Edge-centric graph processing using streaming partitions. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. 472–488.
[79] Benedek Rozemberczki and Rik Sarkar. 2021. Twitch Gamers: a Dataset for Evaluating Proximity Preserving and Structural Role-based Node Embeddings. arXiv:2101.03091 [cs.SI]
[80] Sherif Sakr, Faisal Moeen Orakzai, Ibrahim Abdelaziz, and Zuhair Khayyat. 2016. *Large-scale graph processing using Apache Giraph*. Springer.
[81] Subhadeep Sarkar, Tarikul Islam Papon, Dimitris Staratzis, and Manos Athanassoulis. 2020. Lethe: A tunable delete-aware LSM engine. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 893–908.
[82] Russell Sears and Raghu Ramakrishnan. 2012. bLSM: a general purpose log structured merge tree. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*. 217–228.
[83] Jifan Shi, Biao Wang, and Yun Xu. 2024. Spruce: a Fast yet Space-saving Structure for Dynamic Graph Storage. *Proceedings of the ACM on Management of Data* 2, 1 (2024), 1–26.
[84] Jaewoo Shin, Jianguo Wang, and Walid G Aref. 2021. The LSM RUM-tree: a log structured merge R-tree for update-intensive spatial workloads. In *2021 IEEE 37th international conference on data engineering (ICDE)*. IEEE, 2285–2290.
[85] Li Su, Xiaoming Qin, Zichao Zhang, Rui Yang, Le Xu, Indranil Gupta, Wenyuan Yu, Kai Zeng, and Jingren Zhou. 2022. Banyan: a scoped dataflow engine for graph query service. *Proceedings of the VLDB Endowment* 15, 10 (2022),

2045–2057.

[86] Tin Vu, Ahmed Eldawy, Vagelis Hristidis, and Vassilis Tsotras. 2021. Incremental partitioning for efficient spatial data analytics. *Proceedings of the VLDB Endowment* 15, 3 (2021), 713–726.

[87] Mengzhao Wang, Weizhi Xu, Xiaomeng Yi, Songlin Wu, Zhangyang Peng, Xiangyu Ke, Yunjun Gao, Xiaoliang Xu, Rentong Guo, and Charles Xie. 2024. Starling: An I/O-Efficient Disk-Resident Graph Index Framework for High-Dimensional Vector Similarity Search on Data Segment. *Proceedings of the ACM on Management of Data* 2, 1 (2024), 1–27.

[88] Min Wu, Xinglu Yi, Hui Yu, Yu Liu, and Yujue Wang. 2022. Nebula Graph: An open source distributed graph database. *arXiv preprint arXiv:2206.07278* (2022).

[89] Xingbo Wu, Yuehai Xu, Zili Shao, and Song Jiang. 2015. LSM-trie: An LSM-tree-based Ultra-Large Key-Value Store for Small Data Items. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*. 71–82.

[90] Yi Xu, Henry Zhu, Prashant Pandey, Alex Conway, Rob Johnson, Aishwarya Ganesan, and Ramnatthan Alagappan. 2024. IONIA:High-Performance Replication for Modern Disk-based KV Stores. In *22nd USENIX Conference on File and Storage Technologies (FAST 24)*. 225–241.

[91] Hao Yan, Shuai Ding, and Torsten Suel. 2009. Inverted index compression and query processing with optimized document ordering. In *Proceedings of the 18th international conference on World wide web*. 401–410.

[92] Jaewon Yang and Jure Leskovec. 2012. Defining and evaluating network communities based on ground-truth. In *Proceedings of the ACM SIGKDD Workshop on Mining Data Semantics*. 1–8.

[93] Weiping Yu, Siqiang Luo, Zihao Yu, and Gao Cong. 2024. CAMAL: Optimizing LSM-trees via Active Learning. *Proceedings of the ACM on Management of Data* 2, 4 (2024), 1–26.

[94] Yu, Geoffrey X and Markakis, Markos and Kipf, Andreas and Larson, Per-Åke and Minhas, Umar Farooq and Kraska, Tim. 2022. TreeLine: an update-in-place key-value store for modern storage. *Proceedings of the VLDB Endowment* 16, 1 (2022), 99–112.

[95] Teng Zhang, Jian Tan, Xin Cai, Jianying Wang, Feifei Li, and Jianling Sun. 2022. SA-LSM: optimize data layout for LSM-tree based storage using survival analysis. *Proceedings of the VLDB Endowment* 15, 10 (2022), 2161–2174.

[96] Xin Zhang, Qizhong Mao, Ahmed Eldawy, Vagelis Hristidis, and Yihan Sun. 2022. Bi-directional Log-Structured Merge Tree. In *Proceedings of the 34th International Conference on Scientific and Statistical Database Management*. 1–4.

[97] Xiaowei Zhu, Wentao Han, and Wenguang Chen. 2015. GridGraph:Large-Scale graph processing on a single machine using 2-level hierarchical partitioning. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*. 375–386.

[98] Xiaoke Zhu, Yang Liu, Shuhao Liu, and Wenfei Fan. 2023. MiniGraph: Querying Big Graphs with a Single Machine. *Proceedings of the VLDB Endowment* 16, 9 (2023), 2172–2185.

[99] Zeyang Zhuang, Penghui Li, Pingchuan Ma, Wei Meng, and Shuai Wang. 2023. Testing Graph Database Systems via Graph-Aware Metamorphic Relations. *Proceedings of the VLDB Endowment* 17, 4 (2023), 836–848.

Received July 2024; revised September 2024; accepted November 2024