LSMGraph: A High-Performance Dynamic Graph Storage System with Multi-Level CSR

SONG YU and SHUFENG GONG, Northeastern University, China
QIAN TAO and SIJIE SHEN, Alibaba Group, China
YANFENG ZHANG*, Northeastern University, China
WENYUAN YU, Alibaba Group, China
PENGXI LIU, ZHIXIN ZHANG, and HONGFU LI, Northeastern University, China
XIAOJIAN LUO, Alibaba Group, China
GE YU, Northeastern University, China
JINGREN ZHOU, Alibaba Group, China

The growing volume of graph data may exhaust the main memory. It is crucial to design a disk-based graph storage system to ingest updates and analyze graphs efficiently. However, existing dynamic graph storage systems suffer from read or write amplification and face the challenge of optimizing both read and write performance simultaneously. To address this challenge, we propose LSMGraph, a novel dynamic graph storage system that combines the write-friendly LSM-tree and the read-friendly CSR. It leverages the multi-level structure of LSM-trees to optimize write performance while utilizing the compact CSR structures embedded in the LSM-trees to boost read performance. LSMGraph uses a new memory structure, MemGraph, to efficiently cache graph updates and uses a multi-level index to speed up reads within the multi-level structure. Furthermore, LSMGraph incorporates a vertex-grained version control mechanism to mitigate the impact of LSM-tree compaction on read performance and ensure the correctness of concurrent read and write operations. Our evaluation shows that LSMGraph significantly outperforms state-of-the-art (graph) storage systems on both graph update and graph analytical workloads.

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

Additional Key Words and Phrases: Dynamic Graph Structure, CSR, LSM-tree

ACM Reference Format:
Song Yu, Shufeng Gong, Qian Tao, Sijie Shen, Yanfeng Zhang, Wenyuan Yu, Pengxi Liu, Zhixin Zhang, Hongfu Li, Xiaojian Luo, Ge Yu, and Jingren Zhou. 2024. LSMGraph: A High-Performance Dynamic Graph Storage System with Multi-Level CSR. *Proc. ACM Manag. Data* 2, 6 (SIGMOD), Article 243 (December 2024), 28 pages. https://doi.org/10.1145/3698818

*Yanfeng Zhang is the corresponding author.

Authors' Contact Information: Song Yu, yusong@stumail.neu.edu.cn; Shufeng Gong, gongsf@mail.neu.edu.cn, Northeastern University, China; Qian Tao, qian.tao@alibaba-inc.com; Sijie Shen, shensijie.ssj@alibaba-inc.com, Alibaba Group, China; Yanfeng Zhang, Northeastern University, China, zhangyf@mail.neu.edu.cn; Wenyuan Yu, Alibaba Group, China, wenyuan.ywy@alibaba-inc.com; Pengxi Liu, liupengxi@stumail.neu.edu.cn; Zhixin Zhang, yinshi@stumail.neu.edu.cn; Hongfu Li, lihongfu@stumail.neu.edu.cn, Northeastern University, China; Xiaojian Luo, Alibaba Group, China, lxj193371@alibaba-inc.com; Ge Yu, Northeastern University, China, yuge@mail.neu.edu.cn; Jingren Zhou, Alibaba Group, China, jingren.zhou@alibaba-inc.com.

ACM 2836-6573/2024/12-ART243
https://doi.org/10.1145/3698818

1 Introduction

Real-time analysis of large-scale dynamic graph data has become a key requirement in various fields, extensively applied in recommendation systems [39, 50, 76, 79], fraud detection [34, 78, 84], community management [65, 82], network monitoring [15, 43, 44] and so on. Designing an efficient dynamic graph storage system capable of rapid data storage and effective real-time graph analysis is increasingly crucial and challenging. Such a system must not only support fast real-time graph analysis, but also efficiently ingest updates, especially in scenarios where graphs are updated frequently, such as social networks, e-commerce, and smart city management [44, 72, 86, 89].

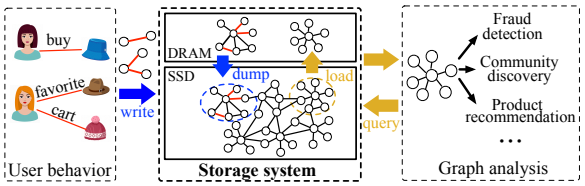

Fig. 1. An example of a graph storage system working in an e-commerce platform, where users or items represent vertices and the interactions between users and items are regarded as edges of the graph.

Real-world Example. Consider a graph storage system that stores the user-item graphs on a large-scale e-commerce platform, such as Alibaba Taobao [13]. Fig. 1 shows a toy example of the graph storage system, where users or items represent vertices, and the interactions between users and items, such as favorite and buy, are regarded as edges of the graph. The system should maintain graph updates and support graph analysis tasks such as product recommendation and fraud detection. However, the massive number of users and their frequent behaviors bring huge challenges to the graph storage system. First, Taobao has approximately 400 million daily active users [49], each generating an average of 10 behavioral data records per day [87]. This means that Taobao generates approximately 46,000 behavioral data records per second. *Therefore, the storage system must support fast data ingestion.* Second, the average size of each behavioral data is approximately 31 bytes [87]. Given the high data generation rate, a 1 TB RAM will be exhausted in less than 9 days, *i.e.,* 1 TB/(4×10^8 users $\times$ 10 bh./day $\times$ 31 bytes) $\approx$ 8.9 days, where bh. refers to customer behaviors such as buying and favoriting. *Therefore, the storage system needs to efficiently digest updates using limited memory and persist them on disks.* In this paper, we default to using SSDs as persistent storage, as they provide a good balance between storage capacity, price, and performance. Third, graph data needs to be analyzed using graph algorithms, such as using PageRank for product recommendations [39, 50] and Label Propagation for fraud detection [78, 84]. *Therefore, the storage system needs to support fast queries for these graph analysis algorithms.*

Existing Disk-based Dynamic Graph Storages. A naive method is to extend the existing memory-based dynamic graph storage systems [36, 89] to disk-based ones by the utilization of automatic memory and disk mapping tools like *mmap* [27]. When updates occur on the graph data stored on disk, these systems automatically load the graph data into memory using *mmap* to perform in-place graph updates. However, this strategy leads to significant read/write amplification, although it makes data operations easier. This is because even if only one edge is updated, *mmap* must read the entire data page from the disk into memory and then write back the entire page after the updates.

On the other hand, there are some dynamic graph storage systems specifically designed for disks, *e.g.,* LLAMA [54] and GraphSSD [55], which use Compressed Sparse Rows (CSR) [70] to improve read performance since CSR is friendly to both random and sequential access to graphs due to its efficient offset index and compactness. However, the compactness of CSR leads to a large amount

of data movement when inserting new data, which decreases graph update performance. Although LLAMA overcomes data movement by generating new CSRs in batches, a large number of CSR fragments are generated as updates continuously arrive. To access all neighbors of a vertex, it is required to read data from multiple CSRs, which results in a lot of random I/Os and decreases graph query performance.

Finally, some graph databases, such as NebulaGraph [10] and Dgraph [7], leverage Log-Structured Merge-tree (LSM-tree) [60] designed primarily for key-value stores to improve write performance. NebulaGraph [10] takes each vertex or edge as a key, and the properties of the vertex or edge as the value. However, when Nebula accesses all the edges of a vertex, the non-contiguous storage of these edges results in a large number of random reads, which decreases read performance. Dgraph [7] takes each vertex as a key, and the associated edges as the value. However, when Dgraph updates an edge of a vertex, it requires updating the whole edge block (as a value), which results in severe write amplification. This deficiency is more pronounced for vertices with more edges. In summary, these works suffer from problems in reading or writing as they ignore the properties of the graphs.

Dilemma. It is challenging to design a graph storage system for persistent storage since there is a dilemma in optimizing read and write performance. In terms of optimizing write performance, a log structure is preferred with only sequential writes (*e.g.,* LSM-tree [60]). However, due to the random arrival order of updated edges, the edges of a vertex can be dispersed throughout the whole log-structured storage, which decreases the read performance when accessing all the edges of a vertex. In terms of optimizing read performance, a compact and sorted structure (*e.g.,* CSR [70]) is preferred with efficient indexing and sequential reads. However, it requires significant overhead due to the data movement to maintain compactness when edges are inserted or deleted.

Insight. LSM-tree [61] is a log structure known for its excellent write performance, implemented by caching recent writes in memory and then performing sequential writes to persistent storage. On the contrary, CSR [70] is a widely used graph storage structure with excellent graph query performance due to its continuous storage of edges for each vertex and efficient vertex/edge index. It is worthwhile to design a novel graph storage structure by combining LSM-tree and CSR so that the structure can simultaneously leverage high performance in the writing of LSM-tree and high performance in the reading of CSR.

Our solution. Based on the above insight, we propose `LSMGraph`, a novel dynamic graph storage system that takes advantage of both LSM-tree and CSR. From a memory perspective, we design an efficient memory cache structure for graph data, *MemGraph*, which can not only rapidly ingest graph updates but also effectively persist LSM-style data structures to disk. From a disk perspective, we propose organizing the graph data into multiple levels similar to the LSM-tree, and each level maintains a part of the graph data in CSR format. When the storage of the i-th level is full, a *compaction* is performed in the background, asynchronously merging the CSR of the i-th level into the next level. In this way, the online continuous data movement overhead of CSR is replaced by the offline periodical compaction overhead of LSM-tree. It is noteworthy that although the edges of a vertex are stored continuously in each CSR, they may span multiple CSRs across different levels, which results in the need to locate their position in multiple files and decreases read performance. To enhance read performance, we design a *multi-level index* that records the positions of each vertex's edges on multiple levels to avoid many random searches. Additionally, we design a *vertex-grained version control mechanism* to mitigate the compaction overhead and allow concurrent read/write operation during compaction.

To sum up, our contributions are summarized as follows.

- A novel dynamic graph storage system LSMGraph that leverages the write performance of LSM-tree and the read performance of CSR (Section 3).
- A memory cache structure MemGraph that manages the cached graph updates efficiently before flushing them to disk (Section 4.1).
- A multi-level index that supports fast reading of a vertex's edges from multiple CSRs on different levels (Section 4.2).
- A vertex-grained version control mechanism that maximizes read and write performance during CSR compaction, while simultaneously guaranteeing the correctness of reading and writing (Section 4.3).
- A high-performance implementation and a comprehensive evaluation to verify the efficiency of our LSMGraph (Section 5). Experiments show that for graph update, LSMGraph achieves an average speedup of 36.12× over LiveGraph, 2.85× over LLAMA, and 8.07× over RocksDB. For graph analysis, LSMGraph achieves an average speedup of 24.4× over LiveGraph, 3.1× over LLAMA, 30.8× over RocksDB, and 6.6× over MBFGraph.

2 Background

In this section, we first introduce graph workloads and their requirements for graph storage and then review the design of CSR and LSM-trees.

2.1 Graph and Graph Workloads

Graph. Given a graph $G = (V, E)$, where V is a finite set of vertices and $E \subseteq V \times V$ is a set of edges. The property of each edge $(u, v) \in E$ is denoted by $p_{u,v}$, which indicates properties such as weight and label, and can be empty.

Graph Update. Graph update is the operation of modifying or updating graph data. It includes adding and deleting vertices/edges and modifying properties of vertices/edges. The graph storage system should support the rapid storage of updates and adjust the structure of the graph according to the updates.

Graph Analysis. Graph analysis algorithms, such as PageRank, SSSP, Label Propagation, etc., require discovering relationships between entities in graph data and some valuable insights. The most basic operation in the graph analysis algorithm is to scan all edges of a vertex [38, 89]. Therefore, to achieve efficient graph analysis, the graph storage structure should support fast neighbor scanning. Besides, algorithms such as triangle counting for graph pattern matching require scanning of ordering neighbors for fast intersections [14, 37, 57]. Therefore, the graph storage system should support both ordered and unordered scanning neighbors.

2.2 CSR and LSM-tree

We next introduce two storage structures, CSR and LSM-tree, and analyze their read and write performance in terms of storing graphs.

CSR. Compressed Sparse Row (CSR) [70] is a type of compressed sparse matrix, which stores the non-zero elements of the matrix in multiple dense arrays. As shown in Fig. 2, CSR utilizes three arrays to store a graph: offset array, edge array, and property array.

- The offset array stores the offset of the first edge of each vertex in the edge array.
- The edge array stores the associated edges of each vertex, *i.e.,* destination vertices, in a continuous manner.

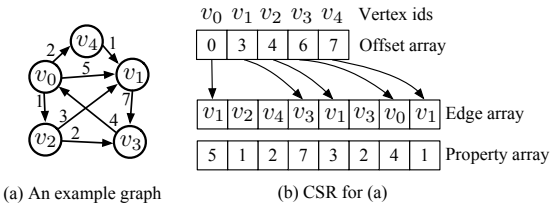

Fig. 2. An example graph and its CSR.

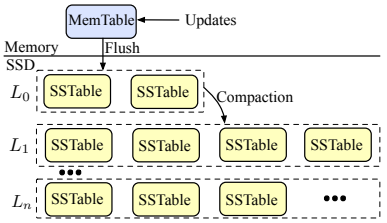

Fig. 3. A classic implementation of LSM-tree.

- The property array stores the properties corresponding to each edge in the edge array, such as weights.

It stores sparse graphs in a compact and contiguous format, providing excellent spatial locality and space efficiency, making it widely adopted in many graph systems [35, 45, 59, 75, 80, 88].

Read Performance Analysis. In CSR, edges are stored in the edge array and indexed by an offset array. When reading a vertex's edges, the first edge's offset in the edge array can be obtained from the offset array using the vertex ID, requiring only one memory I/O, *i.e.,* $O(1)$ memory I/O, as the offset array is usually cached in memory in the existing graph systems. Then, the edge data is loaded from the disk with one disk I/O according to the offset, *i.e.,* $O(1)$ disk I/O. Additionally, to obtain edge properties, an extra disk I/O is required. The primary reason for storing edges and properties separately in CSR is to accommodate different types of graph analysis algorithms, as some algorithms only need the graph's topology without edge properties, *e.g.,* BFS.

Write Performance Analysis. Due to the compactness of CSR, the cost of inserting and deleting edges in CSR is expensive because these operations may result in a significant data movement to maintain compactness. In the worst-case workload, inserting an edge into a CSR requires moving all edges/properties in the edge/property array and modifying all offsets in the offset array. To mitigate the overhead, we partition the edge array by the block size B, ensuring that each I/O operation can move a maximum of B edges. Here, B donates the size of a block, *i.e.,* the number of edges in one block. The amortized disk I/O times for data movement is $O(\frac{|E|}{B})$, and the memory I/O times for modifying the offset is $O(|V|)$.

LSM-tree. The Log-Structured Merge-tree (LSM-tree) [61] is designed to improve the performance of writing ordered data to disk. Fig. 3 shows a classic implementation of LSM-tree, which has been widely adopted in numerous key-value stores [9, 12, 23, 40, 46, 56]. It consists of an in-memory *MemTable* and disk-based *Sorted String Tables (SSTable)* organized on multiple levels. The Memtable is typically implemented with a *skip list* that supports fast insertions and queries. The SSTable is a *fixed-size immutable* structure stored on disk in an ordered manner, where the data is sorted based on the keys.

In an LSM-tree, data is typically organized using key-value pairs. When storing graph data, it is better to take the edge $e(src, dst)$ as the key and the property of the edge as the value [89], where the src and dst represent the IDs of the edge's source and destination vertex.

Write Performance Analysis. LSM-tree first caches graph updates in Memtable, *i.e.,* a skip list. Inserting an edge incurs $O(\log(P))$ memory I/Os, since a binary search is performed, where P is the maximum number of edges that can be cached in MemTable. When the MemTable is full, it is flushed to the disk in SSTable format. LSM-tree employs a multi-level structure to store data on disk. The SSTable flushed from memory is stored at L_0. When the number of SSTables at L_i reaches a threshold, the LSM-tree compacts these SSTables from L_i to L_{i+1}. The capacity of each

Table 1. Comparison of Memory and Disk I/O Complexities for Different Structures.

Structure	Read		Write	
	Memory I/O	Disk I/O	Memory I/O	Disk I/O
CSR	$O(1)$	$O(1)$	$O(\|V\|)$	$O(\frac{\|E\|}{B}))$
LSM-tree	$O(\log(\|E\|))$	$O(L)$	$O(\log(P))$	$O\left(\frac{L \cdot T}{B}\right)$
LSMGraph	$O(1)$	$O(L)$	$O(\log(d))$	$O\left(\frac{L \cdot T}{B}\right)$

level grows exponentially by a factor of T, *i.e.,* the capacity of L_i is $P \cdot T^i$ edges. To facilitate query performance, we only consider the leveling compaction strategy [28, 29], in which compaction can be regarded as a multi-way merge that sorts data by key so that the edges of each vertex are stored contiguously at the current level. Notably, the compaction process always occurs asynchronously in the background, thus hiding the overhead of data movement from writing tasks. The I/O cost of updating an edge is amortized through the subsequent merge operations in which the updated edge participates. In a worst-case workload, all updates reach the largest level $L = \left\lceil \log_T \left(\frac{|E|}{P} \cdot \frac{T-1}{T} \right) \right\rceil$. This means that each edge will be merged across all levels without being discarded by a more recent edge at a smaller level [29]. Before reaching capacity, the edges at each level undergo an average of $O(T)$ merge operations, and across $O(L)$ levels, there are a total of $O(T \cdot L)$ merge operations. We divide the total number of merge operations by the block size B since every disk I/O during a merge operation copies B edges from the original SSTables to the new SSTable. Thus, the amortized disk I/O times for one update is $O(\frac{L \cdot T}{B})$ I/O.

From the above analysis, we can see that LSM-tree improves writing performance in three aspects. 1) It caches and reorders random arrival edges and writes them to the disk in an append-only manner. 2) The expensive online data movement is replaced by the offline compaction periodically. 3) It employs a multi-level structure to organize disk data to mitigate write amplification of compaction.

Read Performance Analysis. In LSM-tree, the edges of a vertex may be distributed across both the MemTable in memory and the SSTables on disk. For edges in MemTables, query operations require $O(\log(P))$ times memory I/O. For edges in SSTables, given that each level's SSTables and their internal data are ordered, enable binary search queries to locate the relevant data block that contains the queried vertex in $O(\log(|E|))$ memory I/Os. Then a Bloom filter [21, 28, 30] is used to detect whether the found data block needs to be loaded from the disk. For example, RocksDB's blocked Bloom filters require only one memory I/O per query [30]. Since P is usually much smaller than $|E|$, the total memory I/O complexity is $O(\log(|E|))$. If the Bloom filter query returns true, one disk I/O is needed to load the data block. In the worst-case workload, each level contains edges for the vertex, which requires loading multiple data blocks. Therefore, the total disk I/O is $O(L)$.

Compared to graph-aware structures like CSR, LSM-trees have poorer read performance in two aspects. 1) Reading a vertex's edges involves a large amount of memory I/O, and in the worst case, requires disk I/Os equal to the maximum number of levels. 2) LSM-tree's SSTables store data as key-value pairs and organize them in fixed quantities without considering the semantics of graphs. In contrast, CSR indexes edges by source vertex ID and stores all edges compactly by source vertex. It was reported that LSM-tree is on average 55 times slower than CSR in reading edges of vertices [89].

Some graph databases [7, 10, 48] use LSM-tree based storage systems to store graph data, such as NebulaGraph [10], Dgraph [7] and ByteGraph [48]. In NebulaGraph [10], keys store vertices or edges, while values store the properties of vertices or edges. Dgraph [7] stores vertices as keys and edges of a vertex as the value. Additionally, ByteGraph [48] stores the edges of each vertex

as a tree structure in memory, with each node of the tree stored as a key-value pair. It should be noted that these systems use existing LSM-tree based systems as their underlying storage, such as RocksDB [12] for NebulaGraph and ByteGraph, and Badger [6] for Dgraph. This approach, which treats LSM-tree based system as a black box, does not incorporate the structural properties and workload characteristics of graphs, and thus still suffers from the same read performance issues analyzed above for LSM-trees.

3 Opportunities and Overview

3.1 Combination Opportunities

As we analyzed in Section 2.2, CSR has been widely adopted in static graph systems [35, 45, 59, 75, 80, 88], as it optimizes read performance through compact storage and efficient indexing. In contrast, LSM-tree, with its multi-level structure that replaces random write with periodical sequential write, has become the preferred storage structure for databases that prioritize write performance [9, 12, 23, 40, 46, 56].

Given the advantages and disadvantages of CSR and LSM-tree, neither structure is ideally suited for supporting both high read and write performance on its own. Fortunately, we now identify two opportunities to combine their advantages for this work.

Opportunity #1: Utilize LSM-tree to hide the movement overhead of CSR. As analyzed in Section 2.2, when inserting or deleting edges, CSR may move enormous data to keep compactness, which is unacceptable as it results in significant I/O overhead on disk. The design of the LSM-tree that turns random writes into sequential writes and online data movement into offline periodic compactions, which is the reason why the LSM-tree has good write performance, can be used to manage CSR to avoid the overhead of online data movement.

Opportunity #2: Utilize CSR to accelerate graph reading. LSM-tree is non-graph-aware. For some frequent query operations, such as reading all edges of a vertex or finding k-hop neighbors of a vertex, will result in a large number of random reads. Suppose the graph data at each level in the LSM-tree is stored in CSR format. In that case, the efficient index and graph-aware storage of CSR, in which the edges of each vertex are stored continuously and ordered on the disk, will be exploited when reading the graph data to improve read performance.

Although there are opportunities to combine their advantages to build new structures that provide high read and write performance, it is non-trivial to efficiently embed CSR into the LSM-tree, as several challenges need to be addressed.

Write. Like LSM-tree, to provide high write performance, it first caches updates in memory, then flushes them to disk. To efficiently flush the cached updates to disk, the edges of each vertex should be stored continuously. A simple method is to reserve a fixed contiguous space for each vertex in memory to store its edges. However, the number of edges of each vertex varies, typically following a power-law distribution. This means that when a high-degree vertex's space is full, it is necessary to expand the capacity and move data, while vertices with few edges will waste memory resources. Therefore, designing a cache structure that can be efficiently updated and flushed to disk presents a challenge.

Read. While the multi-level structure of LSM-tree provides efficient write performance, it complicates the query operations. Since the edges of a vertex may be distributed across multiple levels, extensive lookups are required at each level to locate the data in existing LSM-tree designs, resulting in poor read performance. Designing an index to accelerate the location of a vertex's edges in a multi-level structure is challenging.

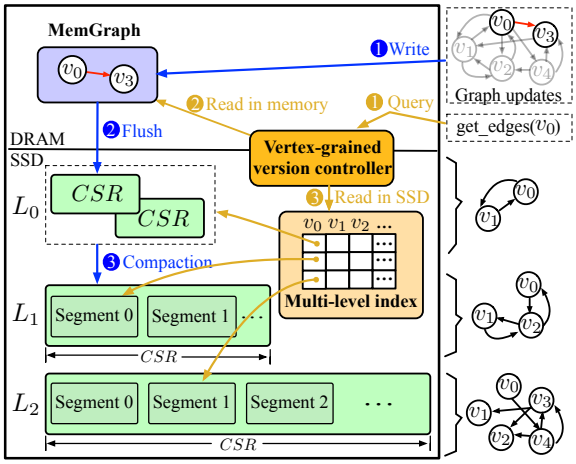

Fig. 4. Overall architecture of LSMGraph.

Write-Read Concurrency. In an LSM-tree architecture, the cached data in memory is continuously flushed, disk files are dynamically compacted and new files are generated. If a query operation encounters deleted or duplicate data, it will lead to incorrect results. It is a challenge to design an efficient version management strategy that ensures each vertex reads the correct version of the data while maintaining high parallelism.

Based on the above two opportunities, we design an efficient disk-based dynamic graph storage system LSMGraph and overcome the mentioned challenges.

3.2 Overview

The overall architecture of LSMGraph is shown in Fig. 4.

Architecture. As shown in Fig. 4, the system consists of four components, *i.e.,* MemGraph, multi-level CSR, multi-level index, and vertex-grained version controller. MemGraph is an efficient in-memory structure to cache recently arrived graph updates efficiently. CSRs organized on different levels are stored on disk and the multi-level index is built to read the edges of vertices from CSRs of different levels efficiently. Vertex-grained version controller is used to support concurrent read/write operations and mitigate the impact of compaction on read/write performance.

Workflow. The read and write workflow of LSMGraph is as follows.

Write. In LSMGraph, when the graph update arrives, *e.g.,* the added edges or deleted edges, they are firstly inserted into the MemGraph (Section 4.1). As graph updates continue to be inserted, once the MemGraph reaches its capacity limit, it is flushed to L_0 on disk with the CSR format. Note that, to improve write performance, the MemGraph is directly written to L_0 without compacting with CSRs at L_0. When the number of CSR files at L_0 reaches its limit, compaction is triggered to merge multiple CSR files at L_0 and L_1 into a new CSR, which is then written to L_1 (Section 4.2.1). There is only one CSR on each level, except for L_0, and it can be divided into several segments. Similarly to LSM-tree, when L_i reaches its capacity limit, compaction is triggered to merge CSR segments and write them to L_{i+1}. The multi-level index is also updated after compaction (Section 4.2.2).

Read. When performing graph analysis algorithms, there are different query requests, such as reading a vertex/edge, reading all edges of a vertex, etc. We take reading all edges of vertex v_0 as an example to demonstrate the query process, as it covers the most common query flow. When the query operation is triggered to read the edges of vertex v_0, LSMGraph utilizes vertex-grained version

Table 2. Degree distribution of cached data in LSM-tree memory cache structure.

Degree	[1,2]	[3,4]	[5,8]	[9,16]	>16
IT-2004 [1]	96.35%	2.85%	0.64%	0.15%	0.01%
UK-2007 [22]	93.53%	3.55%	1.59%	0.74%	0.58%
Twitter [67]	89.45%	5.66%	2.94%	1.19%	0.76%
Friendster [2]	94.57%	4.58%	0.81%	0.04%	0.00%

controller to obtain a set of accessible data, denoted as *version*, which includes MemGraph and the index of v_0 in the multi-level index. LSMGraph first searches and reads data from the MemGraph recorded in the version. Then, according to the index in the version, LSMGraph can obtain the position information of v_1's edges on each level, and read the corresponding data based on this information (Section 4.3).

4 System Design

In this section, we present the details of each core component in LSMGraph.

4.1 Memory Cache Structure

In LSM-tree, the MemTable is used to cache random writes and enhance write performance by converting these into sequential disk writes, making it a crucial component of the system. However, MemTable is primarily designed for key-value data and does not consider the characteristics of graph data and workloads, rendering it inadequate for graph storage requirements. Therefore, designing a memory cache structure that supports fast graph updates and queries is critical for improving both the read and write performance of LSMGraph.

We present the following two important observations before describing the detailed design of LSMGraph's memory cache structure.

Observation 1. For LSM-tree based works such as RocksDB[12], the MemTable is typically implemented as a skip list. A popular implementation [10, 89] for storing graphs using these works takes an edge as a key-value pair, where the vertex ID pair (*i.e.,* <src, dst>) of the edge serves as the unique key, and the edge property as the value. The edges are usually sorted by source vertex ID (*i.e.,* src) in ascending order and destination vertex ID (*i.e.,* dst) in second ascending order. For any inserted edge, the MemTable needs to search for the correct position in the skip list based on the key, and then insert it to maintain the overall key-sorted order of the skip list. Since the MemTable stores all edges of all vertices in the same skip list, inserting and querying edges for a vertex require a complexity of $O(\log n)$, where n is the total number of stored edges. Furthermore, as introduced in Section 2.1, a common operation in graph analysis tasks is scanning all the edges of a vertex. However, scanning all edges of a vertex in the MemTable is not efficient due to the discontinuous memory storage of the skip list. Since MemTable is not optimized for the traits of graph data and workloads, simply applying MemTable's design would lead to poor insertion and query performance.

Observation 2. In most scenarios, *e.g.,* shopping networks, social networks, and web networks, graphs often follow a power-law distribution [18, 32, 83]. Table 2 illustrates the degree distribution of vertices in the memory cache structure of LSM-tree (with a size of 64MB default in RocksDB) when storing the real-world graph datasets (see Table 3 for details). It can be observed that approximately 95% of the vertices have no more than 4 edges, while less than 1% of the vertices have more than 16 edges. This inspires us to consider the characteristics of data distribution when designing memory cache structures.

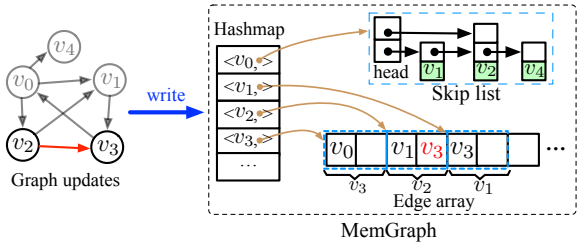

Fig. 5. An example of MemGraph.

MemGraph. Based on the above observations, we design a new memory cache structure for graph data, Fig. 5 illustrates the structure of MemGraph. The MemGraph utilizes a hashmap to store vertex IDs and the address of their first edge. On one hand, the hashmap provides approximate array-like query performance. On the other hand, since MemGraph stores only a part of the graph data at a time, the vertices are sparse, and using a hashmap as an index can save more memory than using an array. The MemGraph employs an edge array and a skip list to store the neighbors of low-degree and high-degree vertices respectively. This design choice is based on the consideration of the power-law nature of graphs, as we mentioned in Observation 2. Storing low-degree vertices in an array since the array provides good spatial locality, which enables fast scanning and quicker flushing to disk. However, storing high-degree vertices in the array may cause a large amount of data movement when inserting edges. Thus, for high-degree vertices, we store their edges in a skip list, which has a better insertion performance and logarithmic query complexity.

Specifically, for vertices with low degrees, we store their edges in the edge array. Different from the edge array in CSR, the edge array in MemGraph is divided into multiple equally sized segments, and each segment only stores the edges of one vertex. Note that, to prevent data movement, the assignment of each segment on the edge array does not follow the order of the source vertices. Instead, segments are assigned based on the order in which the edges arrive, *e.g.*, the first segment is allocated to the source vertex of the first arriving edge. In Fig. 5, with the segment size set to 2, the unique outgoing edge of vertex v_3 is stored in the first segment of the edge array, as $e(v_3, v_0)$ is the first arriving edge. For vertices with edges larger than the segment size, MemGraph inserts their edges into a skip list. As shown in Fig. 5, vertex v_0 has three edges that exceed the size of the array segment, so its edges are stored in a skip list.

4.2 Multi-level CSR

In this section, we first introduce the structure of multi-level CSR, which incorporates vertex-aware compaction. We then introduce the multi-level index to facilitate efficient reading of graph data.

4.2.1 Compact Multi-level CSR. As we mentioned in Section 2.2, the multi-level structure of the LSM-tree effectively mitigates the write amplification of compaction, thus, we also employ a multi-level structure to organize CSRs on disk. As shown in Fig. 4, unlike the LSM-tree, where there are multiple fixed-size SSTables, we ensure that there is only one CSR at L_i ($i > 0$), while multiple CSRs are allowed at L_0 to flush MemGraph more quickly. For each level, the capacity, similar to an LSM-tree, grows exponentially by a factor of T, with $T = 10$ by default. We divide the CSR at L_i ($i > 0$) into multiple segments, each stored in a separate file. When determining the size of each CSR segment, we try to balance the size of each segment as much as possible while ensuring that the edges of each vertex are assigned to the same segment. This approach ensures load balancing during compaction and avoids accessing multiple segments when reading the edges of a vertex within a level. However, if a vertex has many edges, the segment containing these edges may become very large, leading to more severe read and write amplification during compaction. To address this, we

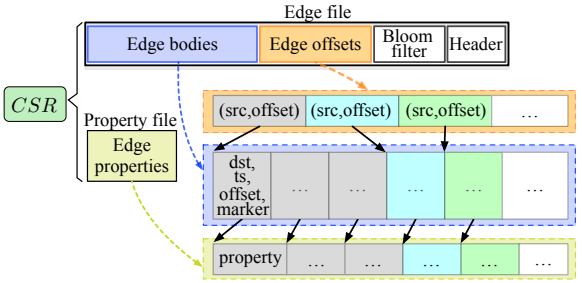

Fig. 6. Details of CSR (segment) file on disk.

allow each vertex with many edges to be stored in a separate segment. By segmentation, when the CSR's size at L_i ($i > 0$) exceeds the capacity of L_i, only one of the segments (denoted as S_j) is compacted to L_{i+1} without affecting the other data of the CSR at L_i. Meanwhile, only a few segments at L_{i+1} are involved in compaction, *i.e.,* segments at L_{i+1} whose edges have the same source vertex with edges in S_j. Regarding L_0, its multiple CSRs originate from the flushing of MemGraph, and their vertex ranges almost always overlap. If each CSR is compacted to the next level individually, it would cause many identical files at the next level to be compacted multiple times. Therefore, we require all overlapping CSRs of L_0 to be compacted to the next level in a single compaction to reduce write amplification.

Compared to the simple graph data stored in Fig. 2, the CSR file requires more information to be persisted on disk, such as metadata to support data recovery and lookup, as well as timestamps for snapshot isolation. Additionally, in each CSR segment, there is only a part of vertices' edges. Employing the original CSR storage format would result in space wastage, which requires allocating arrays with the size of the number of vertices. These differences prompt us to design a new CSR storage format for CSR (segment) on disk tailored to the requirements of LSMGraph.

CSR Storage Format. Similar to the original CSR, we divide a CSR into two separate files, the edge file and the property file, as shown in Fig. 6. For the *edge file*, it comprises four parts, header, Bloom filter, edge offsets and edge bodies. The header stores the metadata of the edge file, including the number of edge bodies, the number of edge offsets, the maximum and minimum source vertex IDs of the edges within the file, and the creation timestamp of the file. The Bloom filter, a highly compact probabilistic representation of a set, is used to roughly filter whether a particular edge is in the file, which could speed up edge queries. Specifically, the two vertex IDs (64 bits each) of an edge are hashed into 32-bit integers and then concatenated into a 64-bit integer to serve as the key for the Bloom filter. Edge offsets record a set of pairs $\langle src, offset \rangle$, where src represents a vertex ID in the current file, and $offset$ records the offset of the first edge of vertex src. The role of $offset$ is similar to the *offset array* in the CSR shown in Fig. 2. Since each CSR segment only stores edges of a part of vertices, we use edge offsets instead of the *offset array* in CSR (Fig. 2(b)) to reduce index size. Edge bodies store the data of each edge that comprises $\langle dst, ts, offset, marker \rangle$. Here, dst represents the destination vertex ID of the edge, ts denotes the timestamp when the edge was inserted, serving for fine-grained snapshot isolation, $offset$ stores the offset of the property of this edge in the property file, as depicted in Fig. 6, and $marker$ indicates whether the edge is deleted.

Compaction is a key operation to maintain the multi-level structure of LSMGraph, and it determines the data movement between different levels. We further introduce the details of the compaction in the following, which combines the characteristics of graph data.

Vertex-aware Compaction. For the CSR segments that will be compacted, we first select the edges with the smallest source vertex ID from all the CSR segments. Then, we sort them in ascending

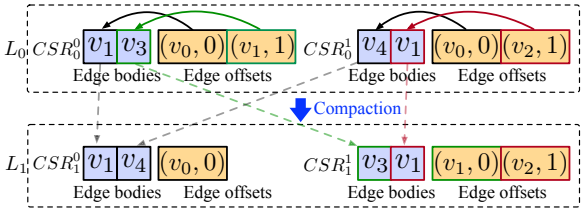

Fig. 7. An example of compacting two CSR segments.

order based on the destination vertex ID, and write them into the same CSR segment sequentially. If the CSR segment has become very large, we write them into a new CSR segment to keep each CSR segment in a reasonable size. After that, we compact the next vertex's edges similarly. In this way, the edges of each vertex are compacted into the same segment, and the edges are sorted in ascending order based on the destination vertex ID.

Example 1: Fig. 7 shows an example of two CSR segments being compacted from level L_0 to level L_1, where level L_1 is initially empty. CSR_i^j denotes the j-th CSR or CSR segment at L_i. First, consider edge offset (v_0, 0) of CSR_0^0 indicating the first edge as edge (v_0, v_1) and edge offsets (v_0, 0) of CSR_0^1 indicating the first edge as edge (v_0, v_4). Obviously, through the offset information, we can determine that they belong to the same source vertex (v_0), and the edge (v_0, v_1) of CSR_0^0 has a smaller destination vertex ID, so it needs to be written to the CSR file (CSR_1^0) of L_1. Then, consider the next edge offset (v_1, 1) of CSR_0^0 indicating $e(v_1, v_3)$ of a new source vertex (v_1). The edge waiting for compaction in CSR_0^1 is still $e(v_0, v_4)$ of the source vertex v_0. Therefore, by comparing the source vertices, the source vertex v_0 of CSR_0^1 is the unique minimum, and $e(v_0, v_4)$ that will be written to CSR_1^0. Repeat the above process. When the third edge $e(v_1, v_3)$ needs to be written, CSR_1^0 reaches the size limit (if each CSR file is limited to 2 edges), and since this edge is the new source vertex (v_1), so this edge is written into the new file CSR_1^1. The final compaction result is shown in Fig. 7. □

4.2.2 Efficient Multi-level Index. Although multi-level CSR combines the excellent write performance of LSM-trees and the efficient read performance of CSR, the multi-level structure of LSM-trees makes reading complex. Since the edges of a vertex may be distributed across multiple levels, existing LSM-tree designs typically use binary search to find the blocks containing the data and then use Bloom filters to determine whether to load those blocks. As analyzed in Section 2.2, this requires numerous random lookups in memory, leading to poor read performance. Therefore, it is essential to design an index to accelerate the location of a vertex's edges in the multi-level structure.

To efficiently select the CSR segments that contain the edges of the queried vertex, we design a multi-level index that directly locates the CSR segments without numerous random lookups. Since the edges are stored continuously within each CSR segment, it is sufficient for our index to locate the first edge of each vertex. In addition, at each level except L_0, the edges of each vertex are stored in only one CSR segment. Therefore, for each level, we can store one position information for each vertex in the index, *i.e.,* the file ID of the CSR segment and the offset of the first edge of each vertex. A simple method to store the position information of each vertex is to employ an array, which is similar to the offset array in CSR (as shown in Fig. 2(b)). For multi-level CSR, we need to create an array for each level. However, we found that not every vertex's edges are present at each level, resulting in the arrays always being sparse. To compress these sparse arrays and save space, we use the structure shown in Fig. 8 to store the position information of vertices.

Multi-level Index. As shown in Fig. 8, there is an array to store the position information of each vertex, where each entity in the array contains three data items. The first item stores a file ID,

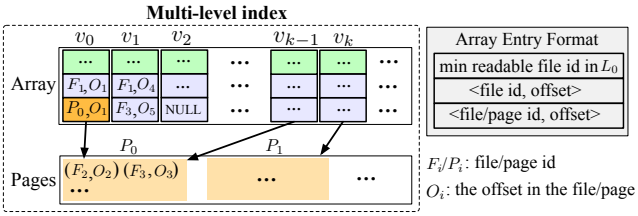

Fig. 8. The structure of multi-level index.

which points to the first CSR file at L_0 that contains the edges of the vertex. Because the edges of one vertex at L_0 may spread over multiple CSRs, recording the first CSR file that contains edges can help filter some invalid random reads. The second and third data items in the array are used to store the position information of the vertex's edges at different levels, *i.e.,* the file ID of the CSR (segment) containing the edge and offset of the first edge in the file. We only record two location information in the array based on the following observation. When each level is full, the last two levels at the bottom of multi-level CSR will hold 99% of the edges because the capacity of each level is tenfold the previous level. When the number of edge positions for a vertex exceeds 2, *i.e.,* the edges of the vertex appear on more than 2 levels, we only use the second data item to store one of the positions, while storing the other position information in a page in the page set (*i.e.,* Pages in Fig. 8). The third data item records the page ID and offset of the position within the page. Each page has a size of 4K and is evenly allocated to a continuous interval of size k vertices (default is 1024) during system startup to maintain the locality of index data access. Each page records position information, as well as the current storage size and remaining space (not shown in Fig. 8). If a page cannot accommodate the indexes of the current interval, the vertex interval is split in half, and the data on the page is written to two new pages, while modifying the recorded page ID and offset in the array according to the vertex interval. When the data stored in a page becomes too small, it will be considered to be merged with pages of adjacent vertices ranges. In addition, the multi-level index is synchronously updated during compaction. Since the position information (*i.e.,* file ID and offset) of all vertices mirrors the edge offset portion of the CSR file (as shown in Fig. 6), obtaining them is straightforward and does not require additional complex calculations.

Note that the multi-level index is updated in place based on the following three intuitions. First, the multi-level index is frequently used in query operations and is usually cached in memory, meaning that most update operations can be completed in memory. Second, the updated multi-level index does not need to be periodically synchronized to disk as long as the memory is enough. Even if it is lost due to a system crash, we can still reconstruct it by the edge offset information from the CSR files on disk. Third, the index updates exhibit good spatial locality because each compaction is limited to a range of vertices. Thus, even if the indexes are on disk, loading them can still take advantage of the amortized I/O.

By utilizing the multi-level index, LSMGraph can quickly obtain the distribution information of all edges of a vertex at each level by using the vertex ID, avoiding many random lookups.

Example 2: Fig. 8 presents a simple example of the multi-level index. The edges of vertex v_0 are stored in three files (*i.e.,* F_1, F_2 and F_3) at different levels. Therefore, in the multi-level index, only the position information of v_0 in the first file is recorded in the array, while the remaining position information is stored on a page with ID P_0, and the page ID and offset are recorded in the array. For v_1, its edge is only stored in files F_1 and F_3, so its position information is recorded in the array. □

I/O Analysis. In terms of write performance, LSMGraph achieves similar write performance to LSM-trees due to its multi-level structure and significantly outperforms CSR. Specifically, inserting

an update into the MemGraph entails O(log(d)) memory I/Os, where d is the average degree of vertices. Disk I/O from compaction is similar to LSM-trees, the amortized disk I/O cost for one update is $O(\frac{L \cdot T}{B})$ I/O. In addition, during the compaction, one memory I/O is required to update the index of the vertex in the multi-level index. Notably, if the multi-level index cannot be cached in memory, it will incur at most one disk I/O because compaction usually targets vertices within a contiguous range, providing good locality and allowing multiple vertices to amortize the I/O. In terms of read performance, LSMGraph uses a multi-level index and CSR format, resulting in better read performance than LSM-trees. However, due to the multi-level structure, its performance is inferior to that of the original CSR. Specifically, locating edges of a vertex in MemGraph requires only $O(1)$ memory I/O. Subsequently, through the multi-level index, the position of a vertex's edges in the multi-level CSR can be obtained with 1 or 2 memory I/Os. Note that if the index is on disk, the I/O becomes disk I/O. In the worst-case workload, where edges are distributed across all levels, loading the edges requires $O(L)$ disk I/O. If properties of edges are also needed, an additional $O(L)$ I/O is required. Notably, when executing graph analysis algorithms, LSMGraph not only reduces lookups in memory but also benefits from the graph-aware CSR storage format. In comparison, LSM-trees need to parse key-value pairs one by one. We also list the times of memory and disk I/O of LSMGraph in terms of write and read in Table 1.

4.3 Vertex-grained Version Control

Consider a concurrent scenario, when reading the graph data in the LSM-tree, the data at L_i is compacting with the data on the L_{i+1}, or the MemGraph is being flushed to L_0. After flushing or compacting, the original data, *i.e.,* MemGraph or CSR (segment) files at L_i will be deleted. However, if the original data is deleted before the data is read, it will cause a read error. Therefore, it is critical to design a version management strategy for the system's files to ensure that read and write tasks run correctly.

Existing LSM-tree based systems [12, 24] maintain a version chain to manage different versions of files. Whenever a new file is generated or deleted, a new version is created. Each version in the version chain records a readable data set in the system. The version in the version chain is a static view of all data, which requires that the recorded data cannot change. However, in LSMGraph, a query task needs to read the multi-level index that needs to be updated during compaction. If an immutable copy is generated for the multi-level index and added to the version each time, the cost is intolerable due to frequent compaction. In LSMGraph, we propose the vertex-grained version control to solve the above problem.

Vertex-grained Version Control. LSMGraph manages versions in two parts based on the location of the data distribution. For the data on MemGraph and L_0, there is no index to record specific location information (*e.g.,* offsets) for each vertex, so the granularity of the version is the entire MemGraph and CSR at L_0. For the data at L_1 and subsequent levels, we maintain the index in the multi-level index for each vertex, which specifies the data range that each vertex can read, providing version control at vertex granularity.

Version Control on MemGraph and L_0. LSMGraph maintains a version chain in memory, with a current version pointer (*i.e., curr_version*) pointing to the most recent version. Each version records MemGraph in memory (including the MemGraph being written and the MemGraph that has reached its capacity, *i.e.,* MemG and MemG. full) and the file IDs (*i.e.,* fid) of the CSR files at L_0. As shown in Fig. 9, at time t_0, the most recent version is version 0 (*i.e.,* vs_0) and it records a MemGraph (*i.e.,* MemG.), a full MemGraph (*i.e.,* MemG. full), and a CSR file ID (*i.e.,* fid 4) at L_0. When a new MemGraph is generated, LSMGraph creates a new version by copying the contents of *curr_version* and adding the new MemGraph. When a MemGraph reaches its capacity, as shown in MemG. full

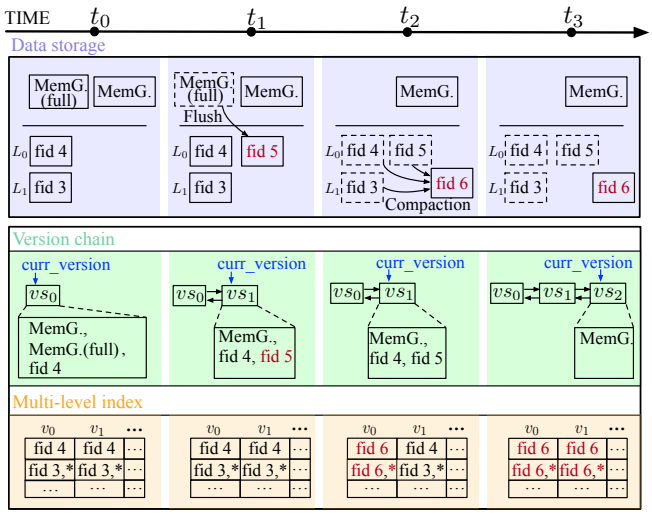

Fig. 9. An example of a version chain and multi-level index that are updated at different times in response to changes in data storage.

in Fig. 9, it needs to be flushed to L_0. Once the flush of the MemGraph is completed, LSMGraph also generates a new version by copying *curr_version*, removing the MemGraph from it, and adding the file ID obtained from the flush. As shown in Fig. 9, at time t_1, the MemG. full is flushed to disk and written to the file (*i.e.,* fid 5). At this point, a new version vs_1 is generated by copying vs_0, removing the MemG. full, and adding the corresponding file ID (*i.e.,* fid 5). Each time a new version is generated, it will be inserted into the version chain as a chain head and become a new *curr_version*. The old version may still exist in the version chain, and once no read operation accesses the version, it is automatically removed from the version chain.

Version Control at L_1 and Subsequent Levels. Each compaction causes the data of the old file to be merged and written into the new file. As shown in Fig. 9, at time t_2, we assume that the vertex ranges of files fid 4 and fid 5 overlap and are selected with file fid 3 at L_1 for compaction. They are then written into a new file, fid 6. For simplicity in this example, segmentation of fid 6 is not considered. Since data on disk is accessed through the multi-level index, we can ensure that the multi-level index points to either the new or the old CSR files, thereby avoiding the reading of duplicate or deleted data. Specifically, if the compaction occurs at L_0, LSMGraph needs to record a minimum readable file ID for each vertex involved in the compaction in the multi-level index, *i.e.,* $fid + 1$, where fid is the maximum file ID at L_0 involved in the compaction. The minimum readable file ID for v_i indicates that query operations for vertex v_i can only access files at L_0 with IDs greater than or equal to this file ID. According to the compaction rules, files at L_0 with IDs less than this file ID do not contain edges of v_i; otherwise, they would have been compacted to the next level. In addition, after each compaction is completed, the position information at L_1 or subsequent levels recorded in the multi-level index is updated based on the new file ID and the edge offset of each vertex in the new file. As shown in Fig. 9, we illustrate two time points (*i.e.,* t_2 and t_3) during the multi-level index update process. At time t_2, only the index of v_0 has been updated, while at time t_3, the index update of the remaining vertices is completed. Specifically, at time t_2, the index (*i.e.,* minimum readable file ID) of v_0 at L_0 is updated to fid 6 (*i.e.,* fid 5 + 1). It means that v_0 can only read files with file ID greater than or equal to fid 6 at L_0. At the same time, the index of v_0 at L_1 is updated to fid 6 and the new offset (denoted as * in Fig. 9) in the file. The updates to the vertex

indices are managed using vertex-grained read-write locks to handle read and write conflicts with the reading threads.

Overall, for data on MemGraph and L_0, LSMGraph uses version chain and *curr_version* for version management. The version chain only maintains the data on MemGraph and L_0 in LSMGraph, unlike traditional version chains (*e.g.,* in RocksDB [12] and LevelDB [24]), which avoids the overhead of maintaining other levels. For data at L_1 and subsequent levels, LSMGraph uses the multi-level index for version management. Note that there is some overlap between these two types of data, as data at L_0 is written to L_1 during compaction. Therefore, LSMGraph ensures global version consistency by cleverly recording the minimum readable file ID in the multi-level index, preventing read operations from accessing duplicate data.

Example 3: We use times t_2 and t_3 in Fig. 9 to explain how to ensure that each vertex reads the correct data during the multi-level index update process. At time t_2, the multi-level index is being updated. For example, information for v_0 has already been modified, while v_1, although needing an update, has not yet started. Using *curr_version* and the multi-level index, v_0 can only read MemGraph and fid 6. Note that files fid 4 and fid 5 will not be read for v_0, even though they are in *curr_version* at L_0, because v_0 can only read files with IDs greater than or equal to its minimum ID (*i.e.,* fid 6) recorded in the multi-level index. For v_1, the data retrieved includes MemGraph, fid 4, fid 5, and fid 3. Although v_0 and v_1 read different files, the compaction process merely merges the data, ensuring that the data they access is equivalent; specifically, the data in file fid 6 is identical to that in files fid 4, fid 5, and fid 3. At t_3, compaction is complete, information for v_1 has already been updated in the multi-level index, and vs_2 is created by deleting fid 4 and fid 5 from vs_1. □

Vertex-grained version control ensures each vertex can access a unique data view during flush and compaction. However, concurrent read-write scenarios still require concurrency control strategies to provide correct data access among multiple read and write threads.

Concurrent Read and Write. There are two kinds of conflicts in our LSMGraph to handle, write-write and read-write conflicts.

Write-write Conflict. As shown in Fig. 4, a write-write conflict only occurs in MemGraph because updates are always written to MemGraph first in LSMGraph. In MemGraph, each vertex's edge is written to its independent array position or skip list, so there are no conflicts between different vertices. When multiple write threads attempt to insert edges for the same vertex simultaneously, LSMGraph uses vertex-grained write locks to ensure safety. They must acquire the write lock for the vertex before performing the write operation. This is a common approach that many existing dynamic graph storage systems have adopted [38, 54, 63, 73, 89].

Read-write Conflict. It occurs when a read operation accesses a vertex's index from the multi-level index while a compaction thread needs to modify the index. LSMGraph ensures read-write safety through vertex-grained read-write locks. A vertex's read-write lock allows multiple read threads to simultaneously acquire the read lock to access the vertex's index, while only one write thread can acquire the write lock to modify the index. This ensures both read-write safety and high performance. Then, based on the obtained index and *curr_version*, we can further access the specific data. When the data being read is in MemGraph, the read operation does not need to acquire any locks. This is because the read operation acquires a timestamp τ before reading, and it only reads edge with timestamps less than or equal to τ. Thus, even if new data is inserted during the reading process, it will not be read because its timestamp must be greater than τ. When the data being read is in the CSR or CSR segment files, there are no read-write conflicts because these files are not modified.

Table 3. Graph datasets used in the experiments.

Dataset	$\|V\|$	$\|E\|$	Avg. Deg.	Size	Type
IT-2004 [1]	41,291,594	1,150,725,436	28	18GB	Web Graph
UK-2007 [22]	105,153,953	3,301,876,564	31	50GB	
Twitter [67]	41,652,230	1,468,365,182	35	22GB	Social network
Friendster [2]	68,349,467	2,586,147,869	38	39GB	

In concurrent read-write scenarios, it is critical to ensure that read tasks access consistent graphs at different times. For example, to achieve correct results in graph analysis, it is necessary to perform each iteration on a consistent graph, *i.e.,* on the same snapshot. The following paragraph introduces how LSMGraph implements this.

Read Graph with Vertex-grained Version Control. LSMGraph provides a consistent graph for a long-running query (*e.g.,* graph analysis) by using timestamps recorded on the edges as snapshots. Specifically, as shown in Fig. 6, each edge records a timestamp (*i.e., ts*) when it is inserted, representing a snapshot. When a graph analysis task is executed, it first acquires the current latest snapshot number (*i.e.,* timestamp), denoted as τ. Subsequently, each time LSMGraph reads the edges of a vertex, it first uses vertex-grained version control to retrieve readable data range (including MemGraph and CSRs on disk) from the multi-level index and the latest version. LSMGraph reads data within the range and only reads edges with timestamps less than or equal to τ. Additionally, any edge with a deletion marker is discarded. If new edges are continuously inserted during the graph analysis process, these new edges will have timestamps greater than τ and will not be read by the current graph analysis task, ensuring that the analysis is always performed on the same snapshot (τ).

5 Evaluation

5.1 Evaluation Setup

Implementation. LSMGraph is implemented in approximately 13,000 lines of C++ code, including the support for various analytical algorithms and the validation for the proposed functionalities. It supports both plain graphs and property graphs with 4-byte or 8-byte vertex sizes (8 bytes in our default setting). The capacity of MemGraph is limited to 64MB, and the capacity of each level on the disk grows by a factor of 10, with a maximum of 5 levels. Additionally, two MemGraph are allowed to alternately accept updates in memory. For various types of properties (*e.g.,* weights), we uniformly store them as string types on the edges. LSMGraph primarily focuses on the design and optimization of edge storage, and we adopt a similar approach to LiveGraph [89] for storing vertices by appending them to a vertex file. Additionally, we use a vertex ID recycling strategy similar to that in LiveGraph [89] and RisGraph [36] to manage deleted vertices, *i.e.,* the IDs of deleted vertices are reused by recycling the IDs of the deleted vertices and assigning them to the newly inserted vertices.

Evaluation Platform. Experiments are conducted on AliCloud ecs.i2.4xlarge instance, which contains 16 hyperthread vCPU cores, 128GB memory (33MB L3 Cache), and 2 SSDs of 1.7TB, which can achieve up to 500MBps read/write sequential performance. All experiments limit their memory to 16GB using the Linux cgroup tool [89]. The instance runs Ubuntu 18.04 with Linux kernel version 4.15.0-173-generic. All codes are compiled using GCC v11.4.0.

Graph Algorithms. We consider four typical graph analysis algorithms in our experiments, including Single Source Shortest Path (SSSP), Breadth-First Search (BFS), Connected Component

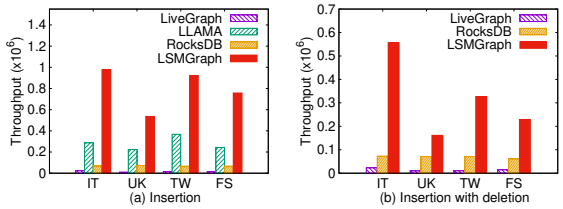

Fig. 10. Throughput comparison of graph updates.

(CC), and SCAN. SCAN refers to traversing all one-hop neighbors of each vertex, which is a fundamental operation in many graph algorithms, such as PageRank, PHP, and GNN. We use the same implementations of all graph algorithms to ensure a fair comparison among various storage systems (except for MBFGraph, we use its own test code because it employs the edge-centric computation model).

Graph Datasets. We adopt four real-world graphs in our experiments, as shown in Table 3, including two web graphs, IT-2004 (IT) [1] and UK-2007 (UK) [22], and two social networks, Twitter (TW) [67] and Friendster (FS) [2]. These datasets are provided as binary files with 8-byte vertex IDs. We randomly inserted edges from the datasets into each system in all experiments. When evaluating the insertion performance, we first insert 80% of the data to form a baseline and then use the remaining data to evaluate the performance [26, 41]. For mixed updates, we default to a ratio of 20:1 for insertions and deletions [17].

Competitors. We compare `LSMGraph` with four state-of-the-art dynamic (graph) storage systems that support updates and analysis on disk, LiveGraph [89], LLAMA [54], MBFGraph [52] and RocksDB [12]. LiveGraph [89] is an advanced dynamic storage system mainly designed for memory scenarios. It also uses *mmap* to support scenarios where the data volume exceeds memory capacity. It stores each vertex's neighbors contiguously to improve the performance of reading the edges of a vertex. LLAMA [54] stores graphs as a time series of snapshots, where each snapshot is similar to a CSR. An excessive number of snapshots can lead to performance degradation. LLAMA creates a new snapshot every 10 seconds as recommended by the author. However, under this setting, LLAMA does not work on the dataset we used. Therefore, we adjusted the interval to 40 seconds. MBFGraph [52] is a state-of-the-art SSD-based analytics system for evolving graphs. It achieves ultimate update performance by directly appending updates to the end of the file. However, this design makes MBFGraph only suitable for the edge-centric computation model. RocksDB [12] is a well-known and widely used open-source key-value store based on LSM-tree. In RocksDB, the adjacency list is represented as a single sorted collection of edges, whose unique key is a vertex ID pair (*i.e.,* ⟨src, dest⟩) [89]. To be fair, we turn off WAL (Write-Ahead Log) in Rocksdb because other systems do not support it. In addition, for other parameters, we adopt RocksDB's default configuration. All systems use 16 threads by default, with 8 threads allocated for reading and 8 threads for writing in mixed read-write workloads. For write tasks, RocksDB and LSMGraph use half of the threads for background compaction.

5.2 Graph Update Performance

We first evaluate the performance of different systems for ingesting graph updates on four real-world datasets, and the results are shown in Fig. 10. Fig. 10(a) shows the throughput (edges per second) of all systems when performing only edge insertions. Fig. 10(b) shows the throughput of insertions with random delete operations (4.76% of all the operators). As mentioned in the documentation

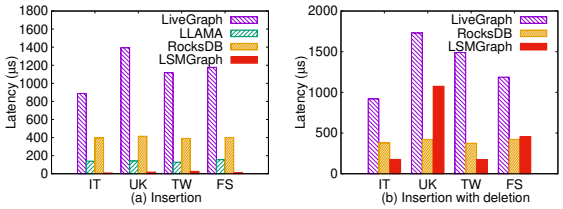

Fig. 11. P99 latency comparison of graph updates.

of the LLAMA code repository [1], LLAMA cannot handle deletion operations correctly. Therefore, the results for LLAMA are not reported in Fig. 10(b). In addition, MBFGraph provides a reference implementation that utilizes the Linux *cat* command and output redirection to append the update file to the original edge file. Using this method, we test the throughput of MBFGraph with different datasets, which obtains the average throughput 3×10^7, surpassing other systems 30-2400×. However, we find that the significant improvement primarily arises from the direct file operations performed by MBFGraph, rather than individual edge insertions. Since this paper focuses on the real-time data ingestion capability of the systems, we mainly discuss graph update performance differences between LSMGraph and other competitors.

As shown in Fig. 10(a), LSMGraph achieves significantly higher throughput than other systems across all datasets. Specifically, LSMGraph achieves an average speedup of 50.56× (up to 58.71×) over LiveGraph. The main benefit of LSMGraph comes from its adoption of an LSM-tree structure, where updates are first cached in memory and then sequentially written to disk. In contrast, LiveGraph performs in-place updates for inserted data and constantly adjusts the size of its edge vector to ensure continuous storage of neighbors, resulting in the lowest throughput compared to all other systems. Compared to LLAMA, LSMGraph achieves an average speedup of 2.85× (up to 3.40×) over LLAMA. After MemGraph reaches its capacity limit, LSMGraph allows for asynchronous flushing to disk and a new MemGraph can continuously accept new updates. In contrast, LLAMA can only insert memory data and write it to disk sequentially to maintain and update links between different versions. In addition, since LSMGraph employs an LSM-tree structure, the data cached in memory at one time can be much smaller than in LLAMA, and the conversion to the CSR file format on disk is faster each time it is flushed. Besides, LSMGraph achieves an average speedup of 11.60× (up to 14.15×) over RocksDB. These benefits primarily come from the efficient memory cache structure, MemGraph, which is specifically designed for graph data in LSMGraph. In addition, RocksDB employs a group commit approach before inserting data into the MemTable in memory, where only the group leader can perform the insertion operation, impacting its insertion performance.

Fig. 10(b) demonstrate similar performance trends as Fig. 10(a), and LSMGraph continues to exhibit outstanding performance advantages. It is worth noting that graph systems experience some performance degradation in deletion scenarios. This is because deletion operations require verifying the existence of the edges to be deleted, resulting in additional query operations compared to pure insertion operations.

We also evaluate the p99 latency of different systems during the above ingestion update process. Fig. 11(a) shows the p99 latency of each system with only insert operations; LSMGraph exhibits significantly lower latency than other storage systems, thanks to LSMGraph's multi-level write-optimized structure and graph-aware cache structure in memory (*i.e.,* MemGraph), which enable fast updates. Fig. 11(b) shows the p99 latency with inserts accompanied by random delete operations; LSMGraph still demonstrates lower latency than other systems in most cases. However, these latencies are higher compared to scenarios with only insert operations. This is because, like other

[1]https://github.com/goatdb/llama

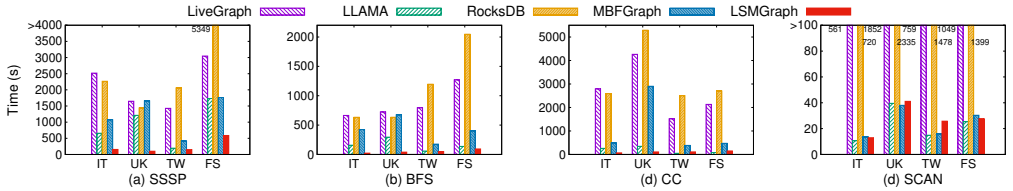

Fig. 12. Running time comparison of graph analysis.

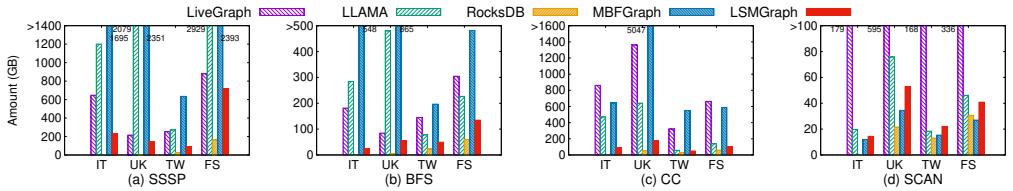

Fig. 13. I/O amount comparison of graph analysis.

graph systems [38, 54, 89], LSMGraph needs to look up an edge before deleting it, causing random disk I/O. In contrast, RocksDB treats deletions the same as insertions without performing lookup operations. Although LiveGraph also performs lookup operations during deletions, its in-place updates cause disk I/O even for insertion operations. Therefore, its latency is consistently high whether deletions are included or not, exhibiting little variation.

5.3 Graph Analysis Performance

We next evaluate the performance of all systems in executing graph analysis on different graph datasets. We also measure the I/O amount during the execution of the algorithms (default representation in the experiment indicates disk I/O). The results are shown in Fig. 12 and Fig. 13, respectively. Overall, the results indicate that LSMGraph outperforms other systems in most cases. Specifically, LSMGraph achieves an average speedup of 24.4× (up to 45.1×) over LiveGraph, 3.1× (up to 12.1×) over LLAMA, 30.8× (up to 57.4×) over RocksDB, and 6.6× (up to 27.2×) over MBFGraph.

For SSSP, BFS, and CC, LSMGraph demonstrates significant performance advantages in graph analytics. Compared to LiveGraph, LLAMA, and MBFGraph, LSMGraph exhibits a lower I/O amount and shorter algorithm execution time. For LiveGraph, the edges of different vertices are scattered and distributed across the disk, resulting in severe read amplification and ultimately leading to degraded performance. LLAMA generates new snapshots continuously to handle updates, requiring the reading of multiple snapshots when accessing all edges of a vertex. However, due to the use of a CSR-like representation for each snapshot, LLAMA achieves better locality between different vertices, resulting in superior algorithm performance compared to other competitors. RocksDB, designed for key-value pairs, has the lowest storage size (analysis in Section 5.4), which makes it have the lowest I/O amount. However, compared to all other graph systems, it has the worst graph analysis performance due to its complex lookup process and inefficient traversal performance of SSTables. By employing MemGraph, multi-level index, and CSR, LSMGraph achieves better lookup and traversal performance than RocksDB. MBFGraph has almost the highest I/O amount because it adopts an edge-centric computation model. For these algorithms, even if only a small number of vertices need to be processed, it still needs to traverse the entire edge file, leading to poor algorithm performance. It is noteworthy that the trend of changes in the amount of I/O and running time are not always completely consistent. For example, in Fig. 12/13(a), the I/O amount for LLAMA and MBFGraph is higher than that for LiveGraph, yet the running time is shorter. This is mainly

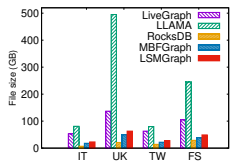

Fig. 14. Space cost comparison.

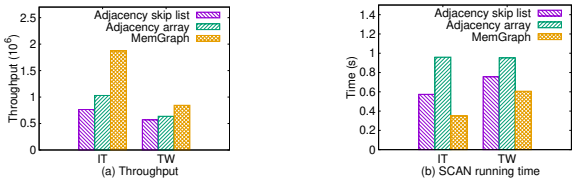

Fig. 15. Performance comparison of different memory cache structures.

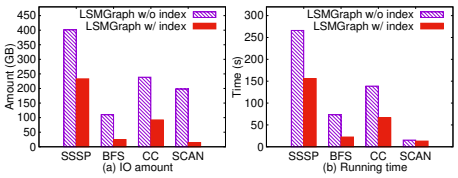

Fig. 16. The effectiveness of multi-level index.

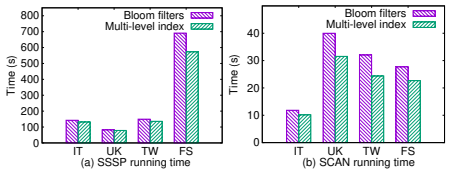

Fig. 17. Comparison of Bloom filters and Multi-level index.

due to LiveGraph's scattered and disordered storage structure, which leads to more random I/Os when retrieving edges for different vertices. In contrast, LLAMA and MBFGraph benefit from more sequential I/O operations due to their compact CSR-like storage structure and edge-centric computation model, respectively. Particularly, MBFGraph requires sequential scanning of the entire edge file in almost every iteration. Although the volume of data read is large, the sequential I/O access pattern better leverages the high data transfer rates of SSDs.

For SCAN, it traversals all one-hop neighbors of each vertex. In this sequential traversal mode, LLAMA, MBFGraph, and `LSMGraph` demonstrate comparable performance and outperform the other two systems. Among them, LLAMA and `LSMGraph` utilize their CSR structures to achieve the greatest spatial locality in sequential traversal mode. Compared with LLAMA, `LSMGraph` achieves comparable performance while storing more information in each edge to provide fine-grained snapshot isolation support for real-time graph analysis, which is a feature that LLAMA's coarse-grained snapshots do not possess. In addition, for MBFGraph, regardless of how much data in the graph is involved in the computation, almost the entire edge file needs to be loaded into memory. Since SCAN requires the participation of all graph data, MBFGraph performs better in running SCAN compared to the first three algorithms.

5.4 Space Cost

We also evaluate the disk space occupied by `LSMGraph` and other systems after inserting four real-world datasets. The results are shown in Fig. 14. Compared to LiveGraph and LLAMA, `LSMGraph` has minimal storage overhead. Specifically, the average storage overhead of `LSMGraph` is only 45% of LiveGraph and 24% of LLAMA. LiveGraph's larger storage size primarily results from the significant replication of edge blocks and its inefficient recycling mechanism. LLAMA's larger storage size stems from the substantial data redundancy between multiple snapshots. Compared to MBFGraph, `LSMGraph` consumes 27% more space on average. This is because MBFGraph only stores the topology and properties of the graph, while `LSMGraph`, as well as LiveGraph, stores timestamps additionally to achieve snapshot isolation. It is worth noting that RocksDB has the lowest space overhead compared to all native graph systems. That's because RocksDB stores vertex IDs as strings, which can save a significant amount of space when vertex IDs are small, and it uses compression algorithms to further compress space, which is orthogonal to storage structure design

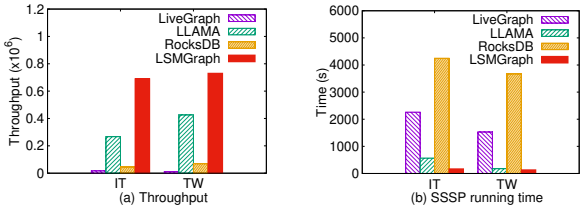

Fig. 18. Performance comparison under update-analysis mixed workload.

and can also be used in other graph storage systems, including LSMGraph. However, these methods lead to higher parsing and comparison costs during the lookup and traversal processes.

5.5 Effectiveness of Memory Cache Structure

To evaluate the design of MemGraph, we compare the throughput of graph updates and the scan time when using only the adjacency array and only the adjacency skip list, as shown in Fig. 15. The skip list implementation is sourced from RocksDB [12], which is a lock-free and efficient implementation that supports concurrent read and write operations. The adjacency array is our implementation, which ensures concurrent read and write operations using read-write locks. LSMGraph outperforms the comparative structures in terms of throughput and scan time on both datasets. For read performance, LSMGraph improves spatial locality by storing edges of low-degree vertices in the single large edge array, thereby avoiding the need to scan multiple small arrays or multiple skip lists, which enhances read performance. For write performance, the high flexibility of skip lists is hard to show with low data volumes and can disrupt the spatial locality of the data. Therefore, when inserting edges of low-degree vertices, LSMGraph uses a common array of edges to avoid issues with frequent small memory allocations and expansions. For high-degree vertices, LSMGraph employs skip lists to increase flexibility, avoiding data migration when inserting edges into the array.

5.6 Effectiveness of Multi-level Index

To verify the effectiveness of the multi-level index on system performance, we compare the performance of LSMGraph with the multi-level index (LSMGraph w/ index) and LSMGraph without the multi-level index (LSMGraph w/o index) on different graph analysis algorithms. When LSMGraph does not use the multi-level index and instead adopts a naive index approach, it utilizes range indexing on the file level and offsets information within the CSR file for lookup. This method is similar to the indexing approach used in LSM-tree implementations like RocksDB. We measure the running time and I/O amount generated by the two methods on IT, and the results are shown in Fig. 16. It can be observed that adopting the multi-level index effectively reduces the I/O amount and the running time of various graph analysis algorithms. These benefits mainly stem from the fact that the multi-level index almost only requires a single random I/O to obtain the distribution information of a vertex's edges on each level. It can also quickly determine whether edges exist on a specific level, avoiding unnecessary read operations. Therefore, the reduction of the I/O amount here is the key to performance gain.

We also compare the efficiency of Bloom filters in RocksDB and LSMGraph's multi-level index on graph analysis performance. Fig. 17 shows the running time of SSSP and SCAN on four datasets using the two different indexing schemes. It can be seen that the multi-level index is more efficient than Bloom filters in these two classical graph workloads. Specifically, the multi-level index outperforms the Bloom filter by an average of about 1.05-1.32×. This gain comes from the fact that multi-level indexes always reduce random memory accesses.

5.7 Performance under Update-analysis Mixed Workload

We analyze the performance of LSMGraph in executing concurrent read-write tasks to evaluate its vertex-grained version control strategy. We evaluate the insertion throughput and execution time of SSSP on different systems in IT and TW. MBFGraph is not included in the evaluation as it does not support the update-analysis mixed workload. Initially, we insert 80% of the edge data as a baseline, followed by continuous edge data insertion while concurrently running the SSSP algorithm. The results are shown in Fig. 18. LSMGraph outperforms all other systems in the concurrent read-write scenario. This is mainly due to the vertex-grained version control of LSMGraph, which is different from the version chain implementation in LSM-trees like RocksDB. It is more flexible, allowing query requests to access the newly merged data earlier. Compared to read-only and write-only scenarios, in the concurrent scenario, LSMGraph achieves a 3× increase in write throughput and a 1.6× improvement in SSSP performance compared to RocksDB.

6 Related Work

Dynamic Graph Storage Systems in Memory. There are numerous in-memory dynamic graph storage systems [31, 36, 38, 42, 44, 47, 63, 64, 74, 81, 89]. Sortleton [38], GraphOne [44], and others [44, 63, 64, 81, 89] achieve fast graph data insertion by employing structures with reserved space or linked-list-like structures. Teseo [47], VCSR [42], and PCSR [81] adopt a structure similar to CSR and utilize reserved slots to alleviate data migration overhead. However, they still suffer from data migration and space expansion when data exceeds the reserved space. Spruce [74] is an advanced dynamic in-memory graph structure that allocates a buffer block and a sorted edge block for each vertex to support edge insertions and updates for that vertex. However, applying this vertex-grained edge block allocation scheme to disk would suffer from poor locality and write amplification issues, which is similar to LiveGraph [89].

Dynamic Graph Storage Systems on Disk. Several disk-based dynamic graph storage systems have been developed to handle scenarios where the graph data exceeds available memory [45, 52, 54, 55, 68, 89]. LiveGraph [89] supports both in-memory and disk-based scenarios. However, LiveGraph's [89] disk performance is poor due to in-place updates and scattered distribution of edge blocks. GraphSSD [55] and LLAMA [54] aim to improve disk-based read performance by utilizing structures similar to CSR. However, they face challenges in updating their CSRs. For instance, LLAMA generates multiple snapshots in CSR format to receive updates; however, read performance degrades when there are too many snapshots. MBFGraph [52] and X-Stream [68] leverage log append approaches to achieve fast write performance and efficient storage of streaming data. However, this approach negatively impacts graph analysis performance, as it requires scanning almost all edges of the graph. Although MBFGraph uses Bloom filters to reduce the number of edges read, it still suffers from severe I/O amplification. Unlike these systems, LSMGraph designs a new graph store to combine CSR and LSM-tree to leverage their complementary advantages and simultaneously optimize read and write performance.

Graph Databases. Numerous graph databases have been developed [3–5, 7, 8, 10, 11, 48, 51, 85]. The main distinction between graph storage systems and graph databases lies in the emphasis of graph databases on transactional support and more complex graph data management. Due to the need for transaction support, even simple graph analysis queries like single-source shortest path (SSSP) and PageRank can incur significant overhead in graph databases, resulting in poor performance when running graph analysis algorithms within the database. Additionally, these systems support exporting data into various graph data formats (*e.g.,* CSR) and running graph analysis algorithms externally. However, the cost of the export process alone can exceed the cost of

running the algorithms and the freshness of the data is low. Similar to other graph storage systems, the goal of LSMGraph is to facilitate the fast storage of large-scale graph data and enable graph analysis on the stored data to provide more real-time graph analysis services.

Transforming from Static Structure to Dynamic Structure. The Bentley–Saxe transformation [20, 71] is able to transform static data structures into dynamic structures. It has been used in various works [16, 58, 69], such as the dynamic extension of static indexes [69]. Theoretically, this can be applied to CSR for dynamic updates. However, simply transforming CSR into a dynamic structure without considering the disk context, the nature of graph data (*e.g.,* power-law distribution), and the characteristics of graph workloads (*e.g.,* retrieving all neighbors of a specific vertex) can lead to poor performance in graph updates and analysis. LSMGraph is designed to provide high read and write performance by leveraging the characteristics of graph data and the common access patterns of graph workloads. Additionally, LSMGraph utilizes LSM-tree to optimize disk I/O overhead caused by random writes.

Filter in LSM-tree. To support better read performance of LSM-tree, numerous LSM-trees implementations use filters to test whether a key is contained in a block, thereby reducing unnecessary disk I/O [21, 28, 33, 53, 77]. However, the filter of each block needs to be queried, which results in a large number of random memory accesses. Some novel filters are designed to reduce the number of memory accesses, such as SlimDB [66], Chucky [30], and Mapped SplinterDB [25]. SlimDB [66] uses a multi-level cuckoo filter [33] to map each key to its level number, and Chucky [30] maintains a filter for the entire LSM-tree, mapping each key to a sub-level number. Mapped SplinterDB [25] uses quotient maplets [19] to map keys to potential SSTables. However, these methods are not suitable for graph workloads that require to access consecutive vertex IDs, *e.g.,* PageRank [62] algorithm. For these workloads, they need to test the filters frequently. GRF [77] is a recent global filter that uses full shape encoding for MVCC but relies on a fixed compaction strategy, which is not suitable for LSMGraph, as LSMGraph uses a partial merge leveling strategy to balance read and write performance.

7 Conclusion

This paper presents LSMGraph, a novel dynamic graph storage system that combines the write-friendly LSM-tree and the read-friendly CSR. We leverage three key designs to enhance system performance and ensure the correctness of read and write tasks. Firstly, we design an efficient in-memory structure, MemGraph, to enable cache graph updates in memory and persist them to disk efficiently. Secondly, we design a multi-level CSR equipped with a multi-level index, where the multi-level CSR utilizes the compact CSR structure embedded in the LSM-tree, which can mitigate write amplification and reduce random reads. The multi-level index is used to locate the edges of the vertex to reduce the random lookups. Lastly, we design a vertex-grained version control mechanism to support concurrent read/write operations and mitigate the impact of compaction on read/write performance. Our evaluations confirm the efficacy and efficiency of LSMGraph.

Acknowledgments

We thank the anonymous reviewers for their constructive comments and suggestions. The work is supported by the National Key R&D Program of China (2023YFB4503601), the National Natural Science Foundation of China (U2241212, 62072082, and 62202088), the 111 Project (B16009), the Joint Funds of Natural Science Foundation of Liaoning Province (2023-MSBA-078), and the Fundamental Research Funds for the Central Universities (N2416011).

References

[1] 2004. IT-2004. https://law.di.unimi.it/webdata/it-2004/.
[2] 2011. Friendster. https://archive.org/details/friendster-dataset-201107/.
[3] 2024. AgensGraph. https://bitnine.net/agensgraph/.
[4] 2024. Alibaba GDB. https://www.aliyun.com/product/gdb/.
[5] 2024. ArangoDB. https://www.arangodb.com/.
[6] 2024. Badger. https://github.com/dgraph-io/badger.
[7] 2024. Dgraph. https://github.com/dgraph-io/dgraph.
[8] 2024. JanusGraph. https://janusgraph.org/.
[9] 2024. LevelDB. https://github.com/google/leveldb.
[10] 2024. NebulaGraph. https://www.nebula-graph.com.cn/.
[11] 2024. Neo4j. https://github.com/neo4j/neo4j.
[12] 2024. RocksDB. https://github.com/facebook/rocksdb.
[13] 2024. Taobao. https://www.taobao.com/.
[14] Christopher R. Aberger, Susan Tu, Kunle Olukotun, and Christopher Ré. 2016. EmptyHeaded: A Relational Engine for Graph Processing. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*. ACM, 431–446.
[15] Leman Akoglu, Hanghang Tong, and Danai Koutra. 2015. Graph based anomaly detection and description: a survey. *Data Min. Knowl. Discov.* 29, 3 (2015), 626–688.
[16] Fatemeh Almodaresi, Jamshed Khan, Sergey Madaminov, Michael Ferdman, Rob Johnson, Prashant Pandey, and Rob Patro. 2022. An incrementally updatable and scalable system for large-scale sequence search using the Bentley-Saxe transformation. *Bioinform.* 38, 12 (2022), 3155–3163.
[17] Renzo Angles, János Benjamin Antal, Alex Averbuch, Peter A. Boncz, Orri Erling, Andrey Gubichev, Vlad Haprian, Moritz Kaufmann, Josep Lluís Larriba-Pey, Norbert Martínez-Bazan, József Marton, Marcus Paradies, Minh-Duc Pham, Arnau Prat-Pérez, Mirko Spasic, Benjamin A. Steer, Gábor Szárnyas, and Jack Waudby. 2020. The LDBC Social Network Benchmark. *CoRR* abs/2001.02299 (2020).
[18] Albert-László Barabási and Réka Albert. 1999. Emergence of scaling in random networks. *Science* 286, 5439 (1999), 509–512.
[19] Michael A. Bender, Martin Farach-Colton, Rob Johnson, Russell Kraner, Bradley C. Kuszmaul, Dzejla Medjedovic, Pablo Montes, Pradeep Shetty, Richard P. Spillane, and Erez Zadok. 2012. Don't Thrash: How to Cache Your Hash on Flash. *Proc. VLDB Endow.* 5, 11 (2012), 1627–1637.
[20] Jon Louis Bentley and James B. Saxe. 1980. Decomposable Searching Problems I: Static-to-Dynamic Transformation. *J. Algorithms* 1, 4 (1980), 301–358.
[21] Burton H. Bloom. 1970. Space/Time Trade-offs in Hash Coding with Allowable Errors. *Commun. ACM* 13, 7 (1970), 422–426.
[22] Paolo Boldi, Bruno Codenotti, Massimo Santini, and Sebastiano Vigna. 2004. UbiCrawler: a scalable fully distributed Web crawler. *Softw. Pract. Exp.* 34, 8 (2004), 711–726.
[23] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Michael Burrows, Tushar Chandra, Andrew Fikes, and Robert Gruber. 2006. Bigtable: A Distributed Storage System for Structured Data (Awarded Best Paper!). In *7th Symposium on Operating Systems Design and Implementation (OSDI '06), November 6-8, Seattle, WA, USA*. USENIX Association, 205–218.
[24] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Michael Burrows, Tushar Chandra, Andrew Fikes, and Robert Gruber. 2006. Bigtable: A Distributed Storage System for Structured Data (Awarded Best Paper!). In *7th Symposium on Operating Systems Design and Implementation (OSDI '06), November 6-8, Seattle, WA, USA*. USENIX Association, 205–218.
[25] Alex Conway, Martin Farach-Colton, and Rob Johnson. 2023. SplinterDB and Maplets: Improving the Tradeoffs in Key-Value Store Compaction Policy. *Proc. ACM Manag. Data* 1, 1 (2023), 46:1–46:27.
[26] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing, SoCC 2010, Indianapolis, Indiana, USA, June 10-11, 2010*. ACM, 143–154.
[27] Andrew Crotty, Viktor Leis, and Andrew Pavlo. 2022. Are You Sure You Want to Use MMAP in Your Database Management System?. In *12th Conference on Innovative Data Systems Research, CIDR 2022, Chaminade, CA, USA, January 9-12, 2022*. www.cidrdb.org.
[28] 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, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*. ACM, 79–94.

[29] 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, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*. ACM, 505–520.

[30] Niv Dayan and Moshe Twitto. 2021. Chucky: A Succinct Cuckoo Filter for LSM-Tree. In *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*. ACM, 365–378.

[31] Laxman Dhulipala, Guy E. Blelloch, and Julian Shun. 2019. Low-latency graph streaming using compressed purely-functional trees. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*. ACM, 918–934.

[32] Michalis Faloutsos, Petros Faloutsos, and Christos Faloutsos. 1999. On power-law relationships of the internet topology. *ACM SIGCOMM computer communication review* 29, 4 (1999), 251–262.

[33] Bin Fan, David G. Andersen, Michael Kaminsky, and Michael Mitzenmacher. 2014. Cuckoo Filter: Practically Better Than Bloom. In *Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies, CoNEXT 2014, Sydney, Australia, December 2-5, 2014*. ACM, 75–88.

[34] Wenfei Fan, Tao He, Longbin Lai, Xue Li, Yong Li, Zhao Li, Zhengping Qian, Chao Tian, Lei Wang, Jingbo Xu, Youyang Yao, Qiang Yin, Wenyuan Yu, Kai Zeng, Kun Zhao, Jingren Zhou, Diwen Zhu, and Rong Zhu. 2021. GraphScope: A Unified Engine For Big Graph Processing. *Proc. VLDB Endow.* 14, 12 (2021), 2879–2892.

[35] Wenfei Fan, Jingbo Xu, Yinghui Wu, Wenyuan Yu, and Jiaxin Jiang. 2017. GRAPE: Parallelizing Sequential Graph Computations. *Proc. VLDB Endow.* 10, 12 (2017), 1889–1892.

[36] Guanyu Feng, Zixuan Ma, Daixuan Li, Shengqi Chen, Xiaowei Zhu, Wentao Han, and Wenguang Chen. 2021. RisGraph: A Real-Time Streaming System for Evolving Graphs to Support Sub-millisecond Per-update Analysis at Millions Ops/s. In *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*. ACM, 513–527.

[37] Per Fuchs, Peter A. Boncz, and Bogdan Ghit. 2020. EdgeFrame: Worst-Case Optimal Joins for Graph-Pattern Matching in Spark. In *GRADES-NDA'20: Proceedings of the 3rd Joint International Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA), Portland, OR, USA, June 14, 2020*. ACM, 4:1–4:11.

[38] Per Fuchs, Jana Giceva, and Domagoj Margan. 2022. Sortledton: a universal, transactional graph data structure. *Proc. VLDB Endow.* 15, 6 (2022), 1173–1186.

[39] Wentian Guo, Yuchen Li, Mo Sha, and Kian-Lee Tan. 2017. Parallel Personalized Pagerank on Dynamic Graphs. *Proc. VLDB Endow.* 11, 1 (2017), 93–106.

[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, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*. ACM, 651–665.

[41] Abdullah Al Raqibul Islam and Dong Dai. 2023. DGAP: Efficient Dynamic Graph Analysis on Persistent Memory. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2023, Denver, CO, USA, November 12-17, 2023*. ACM, 93:1–93:13.

[42] Abdullah Al Raqibul Islam, Dong Dai, and Dazhao Cheng. 2022. VCSR: Mutable CSR Graph Format Using Vertex-Centric Packed Memory Array. In *22nd IEEE International Symposium on Cluster, Cloud and Internet Computing, CCGrid 2022, Taormina, Italy, May 16-19, 2022*. IEEE, 71–80.

[43] Issa Khalil, Ting Yu, and Bei Guan. 2016. Discovering Malicious Domains through Passive DNS Data Graph Analysis. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security, AsiaCCS 2016, Xi'an, China, May 30 - June 3, 2016*. ACM, 663–674.

[44] Pradeep Kumar and H. Howie Huang. 2019. GraphOne: A Data Store for Real-time Analytics on Evolving Graphs. In *17th USENIX Conference on File and Storage Technologies, FAST 2019, Boston, MA, February 25-28, 2019*. USENIX Association, 249–263.

[45] Aapo Kyrola, Guy E. 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 2012, Hollywood, CA, USA, October 8-10, 2012*. USENIX Association, 31–46.

[46] Avinash Lakshman and Prashant Malik. 2010. Cassandra: a decentralized structured storage system. *ACM SIGOPS Oper. Syst. Rev.* 44, 2 (2010), 35–40.

[47] Dean De Leo and Peter A. Boncz. 2021. Teseo and the Analysis of Structural Dynamic Graphs. *Proc. VLDB Endow.* 14, 6 (2021), 1053–1066.

[48] Changji Li, Hongzhi Chen, Shuai Zhang, Yingqian Hu, Chao Chen, Zhenjie Zhang, Meng Li, Xiangchen Li, Dongqing Han, Xiaohui Chen, Xudong Wang, Huiming Zhu, Xuwei Fu, Tingwei Wu, Hongfei Tan, Hengtian Ding, Mengjin Liu, Kangcheng Wang, Ting Ye, Lei Li, Xin Li, Yu Wang, Chenguang Zheng, Hao Yang, and James Cheng. 2022. ByteGraph: A High-Performance Distributed Graph Database in ByteDance. *Proc. VLDB Endow.* 15, 12 (2022), 3306–3318.

[49] Hongfu Li. 2023. GastCoCo: Graph Storage and Coroutine-Based Prefetch Co-Design for Dynamic Graph Processing. *CoRR* abs/2312.14396 (2023).

[50] Zihao Li, Dongqi Fu, and Jingrui He. 2023. Everything Evolves in Personalized PageRank. In *Proceedings of the ACM Web Conference 2023, WWW 2023, Austin, TX, USA, 30 April 2023 - 4 May 2023*. ACM, 3342–3352.

[51] Heng Lin, Zhiyong Wang, Shipeng Qi, Xiaowei Zhu, Chuntao Hong, Wenguang Chen, and Yingwei Luo. 2023. Building a High-Performance Graph Storage on Top of Tree-Structured Key-Value Stores. *Big Data Mining and Analytics* 7, 1 (2023), 156–170.

[52] Chun-Yi Liu, Wonil Choi, Soheil Khadirsharbiyani, and Mahmut T. Kandemir. 2023. MBFGraph: An SSD-based External Graph System for Evolving Graphs. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2023, Denver, CO, USA, November 12-17, 2023*. ACM, 24:1–24:13.

[53] Siqiang Luo, Subarna Chatterjee, Rafael Ketsetsidis, Niv Dayan, Wilson Qin, and Stratos Idreos. 2020. Rosetta: A Robust Space-Time Optimized Range Filter for Key-Value Stores. In *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*. ACM, 2071–2086.

[54] Peter Macko, Virendra J. Marathe, Daniel W. Margo, and Margo I. Seltzer. 2015. LLAMA: Efficient graph analytics using Large Multiversioned Arrays. In *31st IEEE International Conference on Data Engineering, ICDE 2015, Seoul, South Korea, April 13-17, 2015*. IEEE Computer Society, 363–374.

[55] Kiran Kumar Matam, Gunjae Koo, Haipeng Zha, Hung-Wei Tseng, and Murali Annavaram. 2019. GraphSSD: graph semantics aware SSD. In *Proceedings of the 46th International Symposium on Computer Architecture, ISCA 2019, Phoenix, AZ, USA, June 22-26, 2019*. ACM, 116–128.

[56] Yoshinori Matsunobu, Siying Dong, and Herman Lee. 2020. MyRocks: LSM-Tree Database Storage Engine Serving Facebook's Social Graph. *Proc. VLDB Endow.* 13, 12 (2020), 3217–3230.

[57] Amine Mhedhbi and Semih Salihoglu. 2019. Optimizing Subgraph Queries by Combining Binary and Worst-Case Optimal Joins. *Proc. VLDB Endow.* 12, 11 (2019), 1692–1704.

[58] Bilegsaikhan Naidan and Magnus Lie Hetland. 2014. Static-to-dynamic transformation for metric indexing structures (extended version). *Inf. Syst.* 45 (2014), 48–60.

[59] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. 2013. A lightweight infrastructure for graph analytics. In *ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP '13, Farmington, PA, USA, November 3-6, 2013*. ACM, 456–471.

[60] Patrick E. O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth J. O'Neil. 1996. The Log-Structured Merge-Tree (LSM-Tree). *Acta Informatica* 33, 4 (1996), 351–385.

[61] Patrick E. O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth J. O'Neil. 1996. The Log-Structured Merge-Tree (LSM-Tree). *Acta Informatica* 33, 4 (1996), 351–385.

[62] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. 1999. *The PageRank citation ranking: Bringing order to the web.* Technical Report. Stanford infolab.

[63] Prashant Pandey, Brian Wheatman, Helen Xu, and Aydin Buluç. 2021. Terrace: A Hierarchical Graph Container for Skewed Dynamic Graphs. In *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*. ACM, 1372–1385.

[64] Hao Qi, Yiyang Wu, Ligang He, Yu Zhang, Kang Luo, Minzhi Cai, Hai Jin, Zhan Zhang, and Jin Zhao. 2024. LSGraph: A Locality-centric High-performance Streaming Graph Engine. In *Proceedings of the Nineteenth European Conference on Computer Systems, EuroSys 2024, Athens, Greece, April 22-25, 2024*. ACM, 33–49.

[65] Usha Nandini Raghavan, Réka Albert, and Soundar Kumara. 2007. Near linear time algorithm to detect community structures in large-scale networks. *Physical review E* 76, 3 (2007), 036106.

[66] Kai Ren, Qing Zheng, Joy Arulraj, and Garth Gibson. 2017. SlimDB: A Space-Efficient Key-Value Storage Engine For Semi-Sorted Data. *Proc. VLDB Endow.* 10, 13 (2017), 2037–2048.

[67] Ryan A. Rossi and Nesreen K. Ahmed. 2015. The Network Data Repository with Interactive Graph Analytics and Visualization. In *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence, January 25-30, 2015, Austin, Texas, USA*. AAAI Press, 4292–4293.

[68] Amitabha Roy, Ivo Mihailovic, and Willy Zwaenepoel. 2013. X-Stream: edge-centric graph processing using streaming partitions. In *ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP '13, Farmington, PA, USA, November 3-6, 2013*. ACM, 472–488.

[69] Douglas B. Rumbaugh and Dong Xie. 2023. Practical Dynamic Extension for Sampling Indexes. *Proc. ACM Manag. Data* 1, 4 (2023), 254:1–254:26.

[70] Yousef Saad. 2003. *Iterative methods for sparse linear systems*. SIAM.

[71] James B. Saxe and Jon Louis Bentley. 1979. Transforming Static Data Structures to Dynamic Structures (Abridged Version). In *20th Annual Symposium on Foundations of Computer Science, San Juan, Puerto Rico, 29-31 October 1979*. IEEE Computer Society, 148–168.

[72] Jiwon Seo, Jongsoo Park, Jaeho Shin, and Monica S. Lam. 2013. Distributed SociaLite: A Datalog-Based Language for Large-Scale Graph Analysis. *Proc. VLDB Endow.* 6, 14 (2013), 1906–1917.

[73] Sijie Shen, Zihang Yao, Lin Shi, Lei Wang, Longbin Lai, Qian Tao, Li Su, Rong Chen, Wenyuan Yu, Haibo Chen, Binyu Zang, and Jingren Zhou. 2023. Bridging the Gap between Relational OLTP and Graph-based OLAP. In *2023 USENIX Annual Technical Conference, USENIX ATC 2023, Boston, MA, USA, July 10-12, 2023*. USENIX Association, 181–196.

[74] Jifan Shi, Biao Wang, and Yun Xu. 2024. Spruce: a Fast yet Space-saving Structure for Dynamic Graph Storage. *Proc. ACM Manag. Data* 2, 1 (2024), 27:1–27:26.

[75] Julian Shun and Guy E. Blelloch. 2013. Ligra: a lightweight graph processing framework for shared memory. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '13, Shenzhen, China, February 23-27, 2013*. ACM, 135–146.

[76] Qiaoyu Tan, Ninghao Liu, Xing Zhao, Hongxia Yang, Jingren Zhou, and Xia Hu. 2020. Learning to Hash with Graph Neural Networks for Recommender Systems. In *WWW '20: The Web Conference 2020, Taipei, Taiwan, April 20-24, 2020*. ACM / IW3C2, 1988–1998.

[77] Hengrui Wang, Te Guo, Junzhao Yang, and Huanchen Zhang. 2024. GRF: A Global Range Filter for LSM-Trees with Shape Encoding. *Proc. ACM Manag. Data* 2, 3 (2024), 141.

[78] Haobo Wang, Zhao Li, Jiaming Huang, Pengrui Hui, Weiwei Liu, Tianlei Hu, and Gang Chen. 2020. Collaboration Based Multi-Label Propagation for Fraud Detection. In *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence, IJCAI 2020*. ijcai.org, 2477–2483.

[79] Jizhe Wang, Pipei Huang, Huan Zhao, Zhibo Zhang, Binqiang Zhao, and Dik Lun Lee. 2018. Billion-scale Commodity Embedding for E-commerce Recommendation in Alibaba. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, KDD 2018, London, UK, August 19-23, 2018*. ACM, 839–848.

[80] Rui Wang, Yongkun Li, Hong Xie, Yinlong Xu, and John C. S. Lui. 2020. GraphWalker: An I/O-Efficient and Resource-Friendly Graph Analytic System for Fast and Scalable Random Walks. In *2020 USENIX Annual Technical Conference, USENIX ATC 2020, July 15-17, 2020*. USENIX Association, 559–571.

[81] Brian Wheatman and Helen Xu. 2018. Packed Compressed Sparse Row: A Dynamic Graph Representation. In *2018 IEEE High Performance Extreme Computing Conference, HPEC 2018, Waltham, MA, USA, September 25-27, 2018*. IEEE, 1–7.

[82] Jierui Xie, Mingming Chen, and Boleslaw K. Szymanski. 2013. LabelRankT: incremental community detection in dynamic networks via label propagation. In *Proceedings of the Workshop on Dynamic Networks Management and Mining, DyNetMM 2013, New York, New York, USA, June 22-27, 2013*. ACM, 25–32.

[83] Feng Yao, Qian Tao, Wenyuan Yu, Yanfeng Zhang, Shufeng Gong, Qiange Wang, Ge Yu, and Jingren Zhou. 2023. RAGraph: A Region-Aware Framework for Geo-Distributed Graph Processing. *Proc. VLDB Endow.* 17, 3 (2023), 264–277.

[84] Chang Ye, Yuchen Li, Bingsheng He, Zhao Li, and Jianling Sun. 2021. GPU-Accelerated Graph Label Propagation for Real-Time Fraud Detection. In *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*. ACM, 2348–2356.

[85] Wei Zhang, Cheng Chen, Qiange Wang, Wei Wang, Shijiao Yang, Bingyu Zhou, Huiming Zhu, Chao Chen, Yongjun Zhao, Yingqian Hu, Miaomiao Cheng, Meng Li, Hongfei Tan, Mengjin Liu, Hexiang Lin, Shuai Zhang, and Lei Zhang. 2024. BG3: A Cost Effective and I/O Efficient Graph Database in Bytedance. In *Companion of the 2024 International Conference on Management of Data, SIGMOD/PODS 2024, Santiago AA, Chile, June 9-15, 2024*. ACM, 360–372.

[86] Yunhao Zhang, Rong Chen, and Haibo Chen. 2017. Sub-millisecond Stateful Stream Querying over Fast-evolving Linked Data. In *Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai, China, October 28-31, 2017*. ACM, 614–630.

[87] Han Zhu, Xiang Li, Pengye Zhang, Guozheng Li, Jie He, Han Li, and Kun Gai. 2018. Learning Tree-based Deep Model for Recommender Systems. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, KDD 2018, London, UK, August 19-23, 2018*. ACM, 1079–1088.

[88] Xiaowei Zhu, Wenguang Chen, Weimin Zheng, and Xiaosong Ma. 2016. Gemini: A Computation-Centric Distributed Graph Processing System. In *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016*. USENIX Association, 301–316.

[89] Xiaowei Zhu, Marco Serafini, Xiaosong Ma, Ashraf Aboulnaga, Wenguang Chen, and Guanyu Feng. 2020. LiveGraph: A Transactional Graph Storage System with Purely Sequential Adjacency List Scans. *Proc. VLDB Endow.* 13, 7 (2020), 1020–1034.

Received April 2024; revised July 2024; accepted August 2024