BACH: Bridging Adjacency List and CSR Format using LSM-Trees for HGTAP Workloads

Jianfeng Huang
Harbin Institute of Technology
Harbin, China
jfhuang.research@gmail.com

Yihao Cao
Harbin Institute of Technology
Harbin, China
yhcao.research@gmail.com

Shubing Ren
Harbin Institute of Technology
Harbin, China
sbren.research@gmail.com

Baohua Wu
Central South University,
Changsha, China
bhwu.research@gmail.com

Dongjing Miao
Harbin Institute of Technology
Harbin, China
miaodongjing@hit.edu.cn

ABSTRACT

Modern data-intensive applications require databases that support fast analytical processing on massive dynamic graphs in real time, while simultaneously providing transactional guarantees for modifying graph-based objects (*i.e.,* edges, vertices and their properties). Achieving efficient Hybrid Graph Transactional/Analytical Processing (HGTAP) in a database poses significant challenges due to the simultaneous requirements of high operation throughput, high data freshness, and high performance isolation when processing concurrent read/write queries on intricate graph topology. Existing disk-based graph databases fail to meet these requirements at the same time due to their inclined data layout, such as the transactional storage based on adjacency list and the analytical storage based on CSR (compressed sparse row) format.

To address these challenges, we present BACH (Bridging Adjacency List and CSR Format using LSM (Log-Structured Merge)-Trees for HGTAP Workloads) to fill the gaps in HGTAP databases. BACH expands the design space of traditional LSM-Trees to accommodate different graph data layouts in different levels. The compaction process is further extended to seamlessly transform the graph layout from the TP-friendly adjacency list to the AP-friendly CSR format through the data propagation to deeper levels in the LSM-Tree. A novel compaction policy, namely *elastic merge*, is carefully devised to adapt to diverse workloads and the skew vertex degree distribution on graph data. These techniques lead to a *Graph-aware Real-time (GR)-LSM-Tree*, which can provide consistently efficient data access for diverse workloads throughout the entire lifespan of graph objects. Then, a lightweight multi-version scheme is devised for the *GR-LSM-Tree* to accelerate the concurrent read/write processing with the *snap-shot* isolation guarantee. Comprehensive experiments demonstrate that BACH significantly outperforms other disk-based graph database solutions in HGTAP workloads.

PVLDB Reference Format:
Jianfeng Huang, Yihao Cao, Shubing Ren, Baohua Wu, Dongjing Miao. BACH: Bridging Adjacency List and CSR Format using LSM-Trees for HGTAP Workloads. PVLDB, 18(5): 1509 - 1521, 2025.
doi:10.14778/3718057.3718076

Proceedings of the VLDB Endowment, Vol. 18, No. 5 ISSN 2150-8097.
doi:10.14778/3718057.3718076

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/2600254/BACH.

1 INTRODUCTION

Graph data has become increasingly crucial in modern data-driven applications [1, 2], as massive graph processing is behind numerous computing scenarios, such as e-commerce recommendation [3, 4], social media network analysis [5, 6], fraud detection [7, 8] and graph-based deep learning [9, 10]. To accommodate various application scenarios, graph databases are required to support a wide range of graph computations, which can be broadly categorized into two types of workloads: *graph transactional processing* (GTP) and *graph analytical processing* (GAP) [1]. Typically, GTP queries access individual edges, vertices, or the neighborhood of a specific vertex (*i.e.,* the edges, and vertices that are adjacent to it) and require atomicity and isolation during the concurrent insertion, deletion, or modification of multiple vertices and edges. While GAP queries typically involve exploring a large portion or the entire graph (*e.g.,* PageRank and Breadth-First Search) on a static graph snapshot, requiring complicated multi-hop graph traversal and pattern matching processing. And graph pattern matching in contemporary applications often incorporates costly scan-based predicates on properties of graph objects, such as filtering, aggregation, and projection [11], which introduces more challenges for efficient graph analytical processing.

Meanwhile, there is a growing demand for *hybrid graph transactional/analytical processing* (HGTAP) in real-world applications [12, 13]. These applications require fast real-time analysis on dynamic graph with transactional guarantees for precise and prompt decision-making. For instance, in the task of real-time tele-fraud detection, phone signaling transactions are recorded as edge insertions between users. It is imperative to detect abnormal calling behaviors and annotate the phone number of criminals in real-time based on graph analysis for timely user notification prior to answering malicious calls. While in short video applications, real-time user behaviors (*e.g.,* clicks, likes, comments), which are recorded in the form of edge insertions or updates between users and videos vertices

in a graph, are aggregated to optimize real-time recommendation strategy [13]. Other examples include real-time adjustments for marketing planning based on sales transactions in live streaming e-commerce and real-time fraud detection based on trading transactions in finance [8, 14, 15]. It's hard for in-memory storage systems to handle such scale of large graphs in these scenarios (billions of edges), while it is also a waste of resources to store the entire graph in memory as not all graph data are needed for query processing at all times.

Unfortunately, building a disk-based graph database lacking the byte addressing capability to efficiently support HGTAP is not a trivial task since it simultaneously requires high operation throughput, high data freshness, and high performance isolation when processing concurrent read/write queries on intricate graph topology. Existing graph storage structures, despite offering hard trade-offs between update friendliness and static analytic performance, fail to meet these requirements at the same time due to their inclined design. For example, the TP-friendly adjacency list structure represents a graph as a collection of unordered lists, where each list stores the set of neighbors of a particular vertex in the graph. The separating management enable adjacency list-based structures to support efficient graph transactional processing, but it fails to support high throughput graph analytical processing due to the large amounts of random I/Os caused by inconsecutive edge scans [12, 16, 17]. The AP-friendly CSR structure, on the other hand, represents a graph in the form of an adjacency matrix and compresses the sparse matrix into two compact one-dimensional arrays for storage. Due to its ability to ensure purely sequential edge scan, the CSR format has been widely acknowledged as the most compact and efficient structure for most graph analytical processing tasks [14, 17–21], but will incur significant write amplifications and data movements when processing in-place edge deletions and insertions, thus failing to support high-throughput graph transactional processing [12, 16, 18].

In order to provide more balanced query performance and better trade-offs between graph transactional and analytical processing in a database, recent research focuses on transforming graph data between different types of storage structures in batches [14, 22], or devising novel MVCC (multi-version concurrent control) schemes to alleviate concurrent read-write blocking based on a specific graph storage structure, such as the adjacency list-based schemes [12, 16, 23] and the CSR-based schemes [18–20]. Despite offering good performance isolation and operation throughput, the first solution fails to support real-time analysis due to its low data freshness, and significantly increases the difficulty in guaranteeing data consistency and the system maintaining cost due to the existence of two data pipelines. While the second solution provides higher data freshness, it compromises the original benefits of basic graph storage structure and introduces extra overhead to search for the appropriate version of graph-based objects, resulting in worse operation throughput and lower performance isolation. Consequently, contemporary designs of disk-based dynamic graph storage still face significant challenges in supporting efficient HGTAP.

Desiderata. To address these challenges, we outline the following desiderata for disk-based HGTAP databases to meet the simultaneous requirements of **high operation throughput (D1, D2), high data freshness (D3) and high performance isolation (D4)**:

(D1) **The graph transactional processing should prioritize sequential and batched disk I/Os, while minimizing write amplification and data movement.** To achieve high TP throughput, the writes of graph objects should be flushed into disk in batches, rather than incurring a large amount of fragmented, random disk I/O operations or causing severe write amplifications like CSR.

(D2) **The graph analytical processing should achieve comparable query performance and the same computational complexity to static CSR-based purely sequential edge scans.** To achieve high AP throughput closed to static CSR format, the edge scan should be almost purely sequential and causes only a constant times of additional random I/O operations in addition.

(D3) **The data changes committed by graph transactions should be promptly accessible for graph analytical processing.** To achieve high data freshness, ideally, there should not be any time gap and batched transformation between graph transactions committed and their accessibility in graph analytical processing.

(D4) **The concurrent graph transactional and analytical processing should be controlled without interruption or blocking between them, while not introducing extra complexity to edge scans.** To achieve high performance isolation, the concurrent graph transactional and analytical processing should not interrupt or block each other. And the control scheme should not introduce extra random I/O operations to edge scans.

Opportunities. As shown in Table 1, satisfying any two of the four desiderata is relatively simple, but the combination of three or four is significantly challenging, and none of the existing graph storage structures satisfies all four desiderata to our best knowledge. In this paper, we revisit the classic LSM (Log Structured Merged)-Trees structure and discover that it provides a potential framework to achieve the aforementioned desiderata for HGTAP due to the following reasons:

(O1) **LSM-Trees are write-optimized.** The edge writes are initially buffered in memory and only incur batched and sequential disk I/Os within LSM-Trees. And the write amplifications are amortized at file grain and concealed by the backstage compactions.

(O2) **The sequential data structure within LSM-Trees can facilitate sequential edge scans.** The inner sequential data structure in LSM-Trees has the potential to accommodate CSR-like graph storage layout, thereby facilitating sequential edge scans.

(O3) **LSM-Trees can accommodate different graph layout in different levels within one data pipeline, ensuring continuous accessibility throughout the entire lifespan of graph objects.** LSM-Trees store older data in deeper levels and enable seamless graph layout transitions during the data propagation to deeper levels. The committed graph transactions can be immediately accessed by graph analysis, thereby providing high data freshness.

(O4) **LSM-Trees follow the *out-of-place* data ingestion paradigm and naturally support multi-versioning.** The writes to LSM-Trees are first buffered in memory, and then be propagated to immutable read-only disk files in batches, providing high performance isolation. And the scan for searching appropriate version is ensured to be sequential and will not introduce extra random I/Os.

Contributions. We tend to seize these opportunities and take a first step in expanding the design space of LSM-Trees and providing efficient navigation for HGTAP to achieve enhanced operation

throughput, data freshness, and performance isolation. In summary, this paper makes the following contributions:

- We propose the *Graph-aware Real-time (GR)-LSM-Tree*, which extends the traditional LSM-Trees structure for the first time with the ability to accommodate different graph-aware storage layouts in different levels to accelerate HGTAP (§3).
- The compaction process is further extended to enable seamless transitions of graph layouts from TP-friendly adjacency list to AP-friendly CSR format through the data propagation to deeper levels in the *GR-LSM-Tree*, ensuring consistently efficient data access throughout the entire lifespan of graph objects (§4, §5).
- A novel fine-grained compaction policy, namely *Elastic merge*, is carefully devised for *GR-LSM-Tree* to adapt to diverse graph workloads and skew vertex degree distributions, providing better trade-offs between write amplifications and lookups latency (§6).
- A novel lightweight MVCC scheme is devised based on the characteristics of the *GR-LSM-Tree*, incorporating a fine-grained garbage collection mechanism to ensure efficient HGTAP with *snapshot isolation* guarantee (§7).

Table 1: Comparison between different dynamic graph storage systems (*DT* for Desiderata, *ADJ* for adjacency list, ◯ for partial support, failing at the worst case).

DT	**Livegraph** [12] *ADJ-based*	**LLAMA** [19] *CSR-based*	**GraphOne** [22] *Hybrid store*	**Teseo**[1] **[20]** *B^+-Tree based*	**BACH** *LSM-based*
D1	×	✓	✓	×	✓
D2	×	×	×	◯	✓
D3	✓	✓	×	✓	✓
D4	◯	×	✓	◯	✓

2 BACKGROUNDS

Graph Representation. BACH adopts the property graph (PG) model to represent graph data due to its flexibility and popularity [1, 12, 13], where both vertices and edges can have associated labels and properties. Figure 1 (a) shows an example of a property graph used in short video app (*e.g.,* TikTok), which contains vertices labeled *User* and *Video*, and edges labeled *Follows*, *Like* and *Post*. The schema is varied with the labels of vertices/edges (*e.g., User* with {name, age} properties, *Like* with {time} property). Like most graph systems, we adopt the *no-repeated-edges* semantics as foundation [11] (*i.e.,* an edge is uniquely determined by its source and destination vertex).

As mentioned above, the graph topology is typically stored in the TP-friendly adjacency list (AL) or the AP-friendly CSR structure in the graph databases. As shown in Figure 1, the properties in the PG model can be stored correspondingly in an inconsecutive format with the adjacency list (Figure 1 (b)) or in a consecutive columnar format with the CSR structure (Figure 1 (d)). Another graph storage format is edge list (Figure 1 (c)), which is commonly employed when storing a graph in a key-value or relational database, as it represents each edge and its property as a key-value pair. This format facilitates graph transactional processing but is rarely used for graph analytical processing due to its redundant storage and inefficient edge scans.

Given a graph pattern matching query with filters in *Cypher* language: "**MATCH** (a:User)−[e:like]→(b:Video) **WHERE** e.time > d_0 and b.tag = t_0 **RETURN** ∗ ", processing the filter "e.time > d_0" triggers a purely sequential scan on the edge property column in the CSR-based structure (Figure 1 (d)), while introducing four random I/O operations in the AL-based structure (Figure 1 (b)). However, when it comes to updating, inserting, or deleting the edges, it is obvious that the separating management makes AL-based structure more efficient than the CSR-based structure, as the latter typically requires to rebuild the overall compact arrays. Despite the utilization of auxiliary structures (*e.g.,* snapshots, multi-versions, or hybrid structures) can enhance the analytical performance of adjacency lists [12, 16, 22], or improve the update friendliness of the CSR structure [14, 18, 19], these approaches often compromise their original advantages due to poor performance isolation or sacrifice data freshness and transactional supports.

LSM-Trees Basics. The LSM-tree (Log-structured Merge-tree) has been widely adopted in the storage layer of modern NoSQL systems (*e.g.,* key-value storage RocksDB [24]). Different from traditional index structures that apply in-place updates, LSM-Trees defer data file writes and buffer data modifications in a memory-resident segment, while employing a tree structure or skip-list to maintain the order of inserted records. These modifications are then propagated to the immutable disk files through sequential and batched I/Os. The immutable files on the disk are organized into *sorted runs* in different levels. The entries in each *sorted run* are sorted based on the index keys, which can be stored as a single file or alternatively partitioned into several smaller files known as *Sorted String Tables (SSTs)*. A trivial way to store graphs in LSM-trees is using the edge list format. As mentioned above, the edge list format represents each edge and its property in the PG model as a key-value pair, where the key is typically composed of the ID of the source vertex and the destination vertex (Figure 1 (c)). In this way, the LSM-Trees can be employed to improve the write efficiency of graph transactional processing based on the edge list format.

The Compaction Policy. With the continuous writing of changes to edges, the files on disk accumulate over time. And the query performance tends to degrade, as it may require accessing more files with overlapping key range in order to locate a record with a given key. To alleviate this problem, the disk segment is organized into L logical levels of increasing size with a size ratio of T. And the files are merged during the data propagation to deeper levels by a backstage process called *compaction*. As shown in Figure 2, two types of merge policies are typically used in practice, namely *tiering* and *leveling* [25]. The core difference between the two strategies lies in the number of sorted runs permitted in each level within a LSM-Tree. The *leveling* strategy strictly limits a single sorted run for each level. As shown in Figure 2 (a), in *leveling* merge, a SST at level L will be merged multiple times with incoming SSTs at level $L-1$ until it fills up. Then it will be merged with the multiple SSTs at level $L+1$ that share the overlapping key ranges with it. The *tiering* merge, on the other hand, allows multiple sorted runs in a level, and the size of SSTs increases in deeper levels. As shown in Figure 2 (b), when a SST is full, it will be merged with several adjacent SSTs in the same level into a larger sorted run and be directly propagated to the next level. The trade-offs between the two merging policies are well understood: the *leveling* suffers higher write amplifications, but provides better read performance than *tiering*. As the *leveling* policy necessitates multiple merges of a full SST with several SSTs

[1] Although Teseo [20] focuses on in-memory storage for dynamic graphs, the introduced B^+-Tree variant exhibits the promising potential to support disk-based HGTAP. In this analysis, we intactly extend its applicability to the disk storage model.

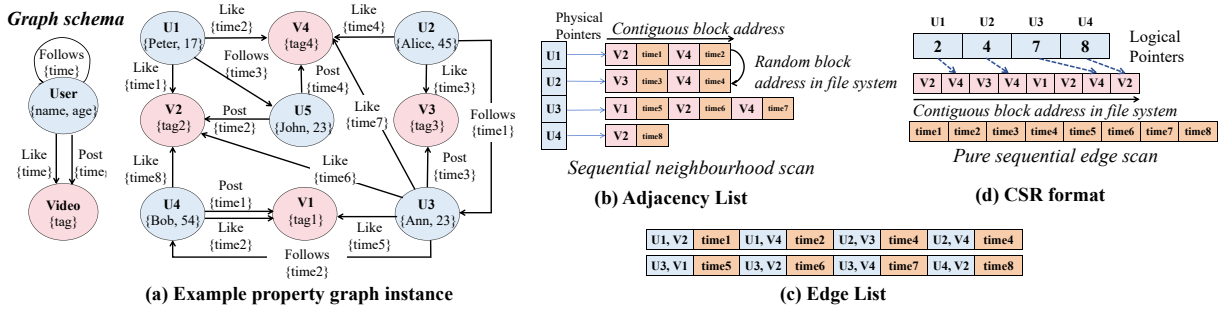


Figure 1: Example property graph instance and various graph storage structures for edge labeled *Like*.

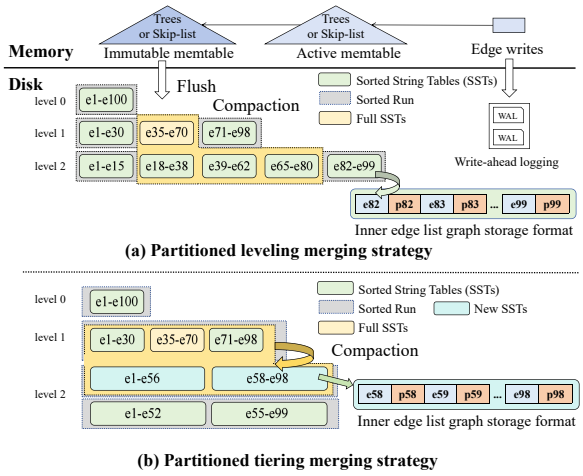


Figure 2: Edge list based LSM-Tree structure

in the subsequent level, but ensuring non-overlapping key ranges among the SSTs within each level after merges. While the *tiering* reduces the merge frequency, it introduces additional complexity in the search of a specific key due to overlapping key ranges among multiple SSTs.

However, although the LSM-Tree based on edge list graph storage format can improve the efficiency of edge writes, it fails to meet all the requirements for efficient HGTAP. This is mainly attributed to the inherent limitations of trivial key-value storage structure within LSM-Trees, which fails to ensure sequential edge scans for efficient graph analytical processing compared to graph-aware structures like the CSR format. In fact, expanding the design space of LSM-Trees to support efficient HGTAP is not straightforward. As the fulfillment of all desiderata for HGTAP mentioned above requires careful and thorough optimization of the embedded data structures, compaction policy, and MVCC scheme within LSM-Trees.

3 DESIGN OVERVIEW OF THE GR-LSM-TREE

The design overview and system architecture of *GR-LSM-Tree* are depicted in Figure 3. The *GR-LSM-Tree* comprises of a *Memory-Resident Component* based on the adjacency list to buffer edge writes and a *On-Disk Component* based on the CSR format to facilitate sequential edge scan. In memory, the active *AL-tables* first buffer the edge writes and organize the graph data in the form of an adjacency list to enable efficient graph transaction processing with high throughput (Figure 3 (a)). Upon reaching its maximum capacity, an active *AL-table* will be frozen and compacted into an immutable *CSR-table* that is ready to be flushed to disk (Figure 3 (b)). On disk, the space is organized into multiple logical levels by the compaction process as well. Once a level reaches its maximum capacity, *CSR-tables* in this level will be merged to a larger one in the next level by the *tiering-like* or *leveling-like* merge policy. The determination of the merge policy is driven by the running workloads and the vertex degree distribution, enabling *GR-LSM-Tree* to possess self-tuning capability and provide consistently superior performance throughout the lifespan of all graph objects (§6). And the *snapshot* isolation is consistently ensured throughout the HGTAP by employing a cooperative in-memory lifetime interval-based and on-disk file snapshot-based multi-version concurrent control scheme.

Overall, the *GR-LSM-Tree* accommodates different graph data layout in different levels, facilitating seamless transitions of graph storage structure from TP-friendly adjacency list to AP-friendly CSR format during data propagation to deeper levels, thereby providing consistently superior performance throughout the entire lifespan of graph objects. The detailed graph-aware structures of *GR-LSM-Tree* and corresponding graph operations are illustrated in the following sections.

4 THE GRAPH-AWARE STRUCTURES

The *GR-LSM-Tree* incorporates two essential graph-aware data structures: the *AL-table* and the *CSR-table*, which respectively organize the graph data in the form of adjacency-list and CSR format.

Memory-Resident AL-table. Edge writes are initially buffered in memory and hashed into multiple memory partitions at $O(1)$ cost based on the source vertex ID. Each of these memory partitions is referred to as an *AL-table*, as it stores the graph data in the form of an adjacency list. For example, in Figure 3 (a), edges sourced from sixty source vertices are hashed into four *AL-tables*. In each *AL-table*, an edge pool is maintained to facilitate efficient memory allocation and sequential edge insertion. In case an edge has multiple versions (*i.e.,* it has been modified multiple times), the stale versions will not be deleted immediately. Instead, the lifetime information of each edge version will be recorded as a time interval [*Creation_time* (T^C), *Deletion_time* (T^D)] to avoid read-write blocks. And a *skip-list* index is maintained for each adjacency list to accelerate the location to the newest versions in the edge pool (ignored in Figure 3). If the stale version has already been flushed to disk, a tombstone will be inserted to the original position to indicate its deletion. Additionally, each version will have a physical pointer that points to the previous version in memory, resulting in a version chain. Upon reaching its

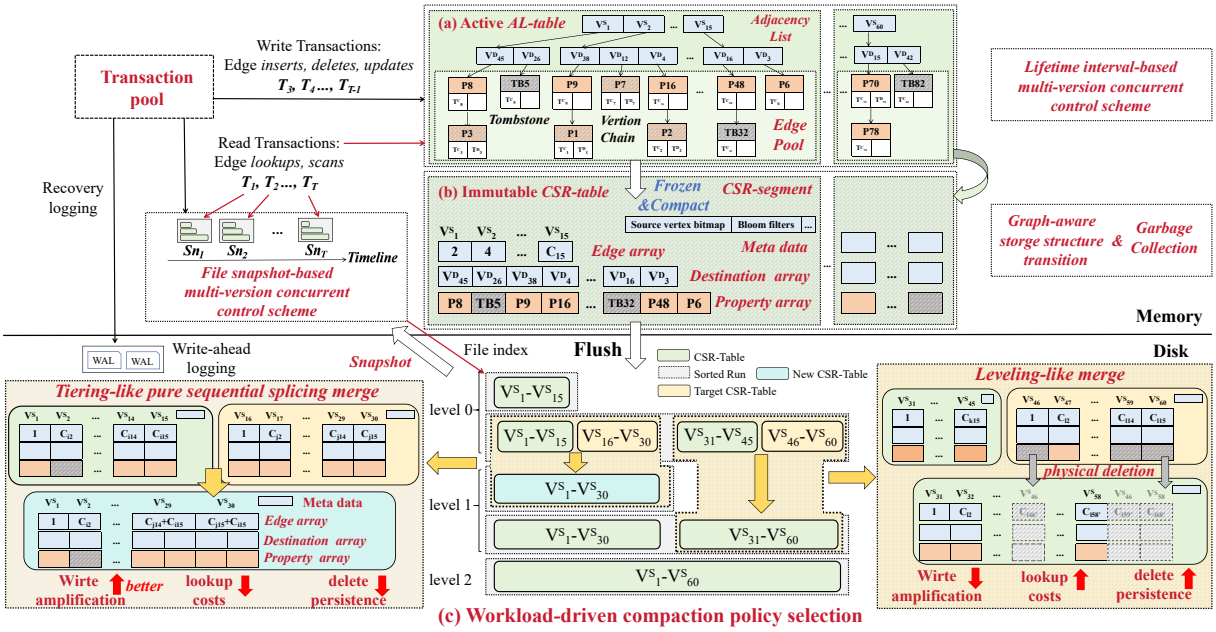


Figure 3: Overview of *Graph-aware Real-time (GR) LSM-Trees*

maximum capacity, the *AL-table* becomes immutable (*i.e., frozen*) and triggers a transition of the graph storage format from adjacency list to CSR format, resulting in an immutable *CSR-table* (Figure 3 (b)). This transition prepares the data to be flushed to disk in a sequential data structure and facilitates efficient graph analytical processing, since the *CSR-table* is more compact and ensures purely sequential edge scans. Meanwhile, since the disk files in *GR-LSM-Tree* are not required to handle in-place data updates, fine-grained lifetime information is no longer needed in *CSR-tables*. And all of the stale versions (*i.e.,* are already deleted before the flush) will be timely deleted before being propagated to disk.

On-disk CSR-table. As shown in Figure 3 (b), there are only three compact arrays stored in a *CSR-table*: the *edge array*, *destination array*, and *property array*. The *edge array* and *destination array* represent the classic CSR graph storage format, while the *property array* consecutively stores the edge properties in blocks, ensuring a purely sequential edge scan. Additionally, each *CSR-table* maintains a bitmap of the source vertices and a vertex-grained edge bloom filter, as well as an overall lifetime information interval of all stored versions in metadata to facilitate a quick determination of whether the appropriate edge versions sourced from a specific vertex are stored in this *CSR-table*. Specifically, BACH replaces the file-grained probabilistic bloom filter (*bf*) in traditional LSM-Tree with a determined and lightweight source vertex bitmap together with a vertex-grained edge bloom filter to amortize the false positive rate and provide better neighborhood locality, while still remaining the $O(1)$ complexity for edge lookups. This is mainly attributes to the source vertex-based fixed partition scheme, which allows us to determine the concrete size of the bitmap in each *CSR-table* during the compactions. The lightweight bitmap index can be further cached in memory to avoid unnecessary I/Os [25, 26]. As files accumulate, the disk space is organized into multiple logical levels by the compaction process. Once a level reaches its maximum capacity, smaller *CSR-tables* in this level will be merged to a larger one in the next level by *tiering-like* merge or *leveling-like* merge.

5 BASIC GRAPH OPERATIONS

Edge Inserts. The edge inserts are initially buffered in the in-memory active *AL-table*. This process involves hashing the newly inserted edges to their respective *AL-tables* and allocating the corresponding list based on the source vertex ID of each edge in $O(1)$ complexity. The newly inserted edge versions will be sequentially written at the tail of the edge pool array. After insertion, the *Creation_time* of this version will be recorded as the timestamp at which this transaction is processed according to the transaction pool management (§7), while the *Deletion_time* is set to -1 (indicating that it is still alive) for subsequent updates or deletions of this edge.

Edge Deletes. Rather than directly execute in-place edge deletes, BACH employs a *lazy deletion* strategy like other LSM-based systems. Specifically, if the target edge has the most recent version stored in the memory-resident active *Al-table*, the deletion process will directly modify the *Deletion_time* of the target edge version to match the timestamp at which the deletion transaction is processed. Otherwise, if the stale versions of the target edge has already been flushed into disk within a immutable *CSR-table*, the deletion process will not require in-place modification of the edge version on disk in case of causing a large amount of random and fragmented disk I/Os. Instead, a tombstone will be inserted to its corresponding position in *AL-table* following the same *insert* process (as depicted in Figure 3 (a)), which records the timestamp of when the edge version is deleted. The tombstone will be subsequently propagated to deeper levels by the compactions and be reclaimed at the bottom, ensuring that all stale versions with the same key remaining in upper levels have been physical deleted.

Edge Updates. The processing of an edge update operation is similar to that of edge insertion. Specifically, if there exist previous

versions of this edge in the memory-resident active *AL-table*, the new version will be linked to the head of the version chain and a physical pointer will be added between the version and its predecessor. Followed by modifying its *Creation_time* and *Deletion_time* of the previous version of this edge. Otherwise, the new version will be directly inserted to its position if the previous version has already been flushed to disk. The simplified update operation will not introduce any chaos, as only the entry with the most recent timestamp can survive during a compaction.

Edge Lookups. During the search for a specific edge, the memory-resident *AL-tables* are accessed with preference. This process involves $O(1)$ adjacency-list location and $O(logD)$ edge lookup to the newest version facilitated by the *skip-list* index. If there is no appropriate version that satisfies the query condition in the version chain within the *AL-table*, the on-disk component is accessed for further search. Specifically, the search on disk will initiate at the top of the *GR-LSM-Tree* (*i.e.,* level 0) and gradually proceed to deeper levels, accessing *CSR-tables* with the key range that contains the edge's source vertex ID. The search process first verifies whether the timestamp of the read query falls within the overall lifetime interval maintained in the metadata and subsequently checks the source vertex bitmap and the edge *bf* to determine whether the target edge is stored in the *CSR-tables*. If either of the two checks returns false, the search process will directly skip the scan of this *CSR-table*. Otherwise, the *Edge array* and the *Destination array* will be further accessed to locate the target edge and obtain the offset of the edge property in the *Property array*. If the search process first encounters the tombstone of this edge, it indicates that the edge version has already been deleted and will return empty results. The search process will end as soon as it finds the appropriate version of the edge in an *AL-table*, or encounters the latest version of the edge or its tombstone in a *CSR-table*, since there is only a correct version living at the time of the read query is processed.

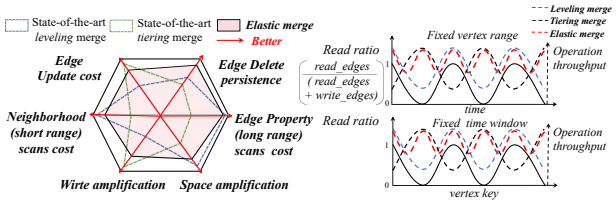
Neighborhood Scans. Neighborhood scan of a specific vertex will first locate all *AL-tables* and *CSR-tables* with the key range containing the vertex ID across all levels within the *GR-LSM tree*. Since a *CSR-table* indexes the source vertex IDs as keys, the neighborhood scan can be accelerated by the memory-cached bitmap to reduce unnecessary file accesses. The file snapshot-based MVCC scheme (§7) ensures that the latest version of a specific edge is just the appropriate version in the accessible snapshot of *GR-LSM-Tree* assigned to a read transaction. Therefore, during the top-to-down search in the *GR-LSM-Tree*, we maintain an intermediate bitmap to indicate whether the latest version of an edge has been processed already and skip the evaluation of staled versions. Within a *CSR-table*, the search process will allocate the target neighborhood based on the *Edge array* in $O(1)$ complexity. And the staled edge versions will be identified and discarded if their corresponding unit in the intermediate bitmap has already been set to 1.

Edge property (Long range) Scans. Graph analytical processing on property graphs (*e.g.,* complex sub-graph pattern matching with filters) requires to scan a large part of the graph. During which, the full scan of edge properties with a specific label may be required. As shown in Figure 1, processing the filter "Like.time > d_0" requires to scan all edge properties labeled *Like*. In *GR-LSM-Tree*, the *CSR-tables* are scanned level by level in a purely sequential way. In each *CSR-table*, the *property array* are scanned to evaluate the filter while the *Edge array* and the *Destination array* are scanned to retrieve the corresponding source and destination vertices of legal edges. Followed by generating a {*source vertex, edge property, destination vertex*} tuple in the intermediate table. The ordered intermediate results obtained from scanning each level will be merged with the result from the previous level to discard staled edge versions, and get a final result upon completion of the search in the last level.

Vertex Operations. Vertex reads first access the vertex index and then the vertex block. The frequency of updates for vertices is comparatively lower than that of edges [1, 12, 13], thus vertex writes create a version chain of the vertex block on disk, and set the vertex index to point to the newest version. In the uncommon case where a read requires a previous version of the vertex, it follows the per-vertex linked list of vertex block versions in backward timestamp order until it finds a version with a timestamp smaller than its read timestamp.

6 WORKLOAD-DRIVEN ELASTIC MERGE

The Problems of Current Designs. Compaction policy greatly influences the performance of LSM-based systems by determining their update cost, delete persistence latency, point and range lookups performance, space amplification, and write amplification [25–27]. However, as depicted in Figure 4 (a), both the *tiering* and *leveling* compaction polices present hard trade-offs between these performance indicators in HGTAP, resulting in an imbalanced and non-hexagonal shape on the performance radar chart. Unbalanced performance severely restricts the effective adaptation of a single compaction policy to diverse graph workloads and vertex degree distributions. As shown in Figure 4 (b), existing compaction policies fail to achieve excellent and stable performance when graph workloads vary across vertex ranges and change over time, which is a common situation in real-world HGTAP scenario (*e.g.,* a sudden burst of likes to a specific video in TikTok). Because neither the *leveling* nor the *tiering* merge can maintain high throughput when varying read ratios over time within a fixed vertex range or varying average read ratios across vertex ranges within a fixed time window. Due to the imbalanced performance, the operation throughput curve of *leveling* roughly varies in accordance with the trend of read ratio and exhibits large fluctuations, while the throughput curve of *tiering* exhibits synchronously opposite fluctuations (as shown by the black and blue dashed line in Figure 4 (b)).



(a) Performance radar chart (b) Example of performance on varying workloads

Figure 4: The trade-offs between different compaction polices

BACH's Solutions. BACH addresses this problem by carefully devising a novel fine-grained merge policy for *GR-LSM-Tree*, named *Elastic merge*, which enables each *CSR-table* to independently choose the merge policy based on the characteristics of running workloads

on it. This is mainly attributed to the optimization of memory-resident *AL-tables*, which facilitates data propagation to the disk within smaller fixed ranges of the source vertices. Specifically, a $k-$ary tree is constructed with these smaller vertex ranges as leaf nodes when the corresponding *CSR-tables* are flushed to disk, where k is the number of *CSR-tables* operated in one compaction. The $k-$ary tree is maintained in memory to track the running status of the entire LSM-tree, where each node stores the counts of each type of running operation and a file list of *CSR-tables* associated with the corresponding vertex range. As shown in Figure 3 (c), k adjacent *CSR-tables* can be either merged into the latest *CSR-table* in their father node following the *leveling-like* policy, or be merged together to a new larger *CSR-table* and be directly added to the file list of their father node following the *tiering-like* policy. The selection of the compaction policy of target files is automatically tuned based on the information of running workloads maintained in each node of the $k-$ary tree. The compaction to the currently last level is strictly controlled to follow the *leveling-like* policy, ensuring that the majority of the inserted graph data are compacted into CSR arrays without overlapping ranges to facilitate sequential edge scans.

Furthermore, as illustrated in Figure 3 (c), BACH optimizes the execution of *tiering-like* and *leveling-like* policy for *CSR-tables* to achieve enhanced performance. Specifically, the *tiering-like* compaction will prioritize merging the *CSR-tables* with non-overlapping key ranges. This strategy enables pure splicing of the compact arrays within *CSR-tables*, thereby fully leveraging the consecutiveness of the CSR format and reducing fragmented and random edge retrievals during compactions. However, this kind of sequential splicing compaction will hinder read performance and delete persistence, since the tombstones will never encounter their owner during the compactions. As for *leveling-like* merge, we optimize the execution by enabling multiple adjacent *CSR-tables* to be merged into the same *CSR-table* in the deeper level, thereby alleviating the average edge write amplifications and the update costs.

Upon a level or an interval reaches its maximum capacity, k adjacent *CSR-tables* in this level that share the same father in the $k-$ary tree will be randomly selected and collectively added to a backstage compaction queue. The compaction policy of the k adjacent *CSR-tables* will be determined by their *Tombstone ratio* and the *Read ratio* maintained in their father node within the $k-$ary tree at execution time. Specifically, as the *Tombstone ratio* and *Read ratio* increase, the compaction execution tends to choose the *leveling-like* strategy because it offers better edge read performance and enables more timely deletion persistence. Conversely, as the two ratios decrease, the *tiering-like* strategy will be chosen with preference to reduce write amplifications and update costs for intensive edge writes. In the current implementation, we employ the classic weighted average method to drive the decision of the merge policy and leave a space for users to tune the compaction policy between the pure *tiering-like* and *leveling-like* execution based on customized requirements. This kind of fine-grained compaction strategy optimization enables BACH with self-tuning ability to efficiently handle a sudden burst of edges writes to a super vertex while having slight impact on the edges scan over other vertices. Furthermore, in case of the intensive reads after the edge writes burst, BACH will trigger a *leveling-like major compaction* to merge all *tiered runs* within the target range into one sorted run in each level.

Complexity Analysis. We make detailed analysis of the operation complexity for different graph storage layouts in Table 2, where D represents the degree of target vertex and $\overline{D}$ represents the average degree of target vertex in each *CSR-table*, $|E|$ represents the total number of edges, B represents the number of entries that fit into a storage block, *FPR* represents the false positive rate of *bf*, T represents the size ratio between levels, L represents the total levels in *GR-LSM-Tree* and M represents the number of levels that exists *tired runs* within the target vertex range made by *Elastic merge*.

The results show that *leveling* and *elastic* merge based *GR-LSM-Tree* reaches the same operation complexity as CSR format in *Edge_lookup* and *Long range_scan*, and achieves a great promotion in sequential *Edge_write* performance, while slightly sacrifices the efficiency of *Neighbor_scan*. This is because the most of edges (*e.g.,* 90% when T is set to ten) are compacted to the overall *CSR-table* at the bottom of *GR-LSM-Tree*, dominating the cost of scanning other levels. While the cost of *Neighbors_scan* and *Edge_lookup* are comparable to CSR format since it incurs at most L times of random I/Os (extra T times for tiered runs) at the worst case. This drawback can be potentially mitigated by utilizing the source vertex bitmap in practice. Regarding edge insert performance, *GR-LSM-Tree* significantly out performances CSR and adjacency list due to the memory-resident buffering structure and the capability to ensure batched and sequential disk I/Os. The analysis verifies the promising advantages of *GR-LSM-Tree* in facilitating efficient HGTAP. As for the comparison between different compaction polices in *GR-LSM-Tree*, *Elastic merge* reveals a more balanced performance and offers better trade-offs between read performance and write amplifications, which can be illustrated by the consistent color shade of signs in Table 2. Therefore, *Elastic merge* enables BACH to provide consistently superior performance and better overall throughput for diverse graph workloads and skew vertex degree distributions in HGTAP.

Table 2: The worst-case disk I/O cost comparison for different graph storage solutions (▼, ♦, ▲ represents worse, the same, or better compared to the classic graph storage structures while the shades of color represent the degrees).

Operations	*ADJ*	*CSR*	*GR-LSM-Tree*		
			Tiering	*Leveling*	*Elastic*
Edge_lookup	$O(1)$	$O(1)$	$O(1+FPR\cdot T)$▼	$O(1)$♦	$O(1)$♦
Neighbors_scan	$O(\frac{D}{B})$	$O(\frac{D}{B})$	$O(\frac{T\cdot L\cdot \overline{D}}{B})$▼	$O(\frac{L\cdot \overline{D}}{B})$▼	$O(\frac{(L+M\cdot(T-1))\cdot \overline{D}}{B})$▼
Long range_scan *(random/sequential)*	$O(\frac{\lvert E\rvert}{B})$ *r.d*	$O(\frac{\lvert E\rvert}{B})$ *s.q.t*	$O(\frac{T\cdot\lvert E\rvert}{B})$▼ *s.q.t*	$O(\frac{\lvert E\rvert}{B})$♦ *s.q.t*	$O(\frac{\lvert E\rvert}{B})$♦ *s.q.t*
Edge_write amplification *(random/sequential)*	$O(\frac{D}{B})$ *r.d*	$O(\frac{\lvert E\rvert}{B})$ *r.d*	$O(\frac{L}{B})$▲ *s.q.t*	$O(\frac{T\cdot L}{B})$▲ *s.q.t*	$O(\frac{M+T\cdot(L-M)/k}{B})$▲ *s.q.t*

7 MVCC SCHEME

To facilitate efficient HGTAP with timely correct results and reduce concurrent read/write blocking, a novel lightweight MVCC scheme is further devised for BACH based on the characteristics of *GR-LSM-Tree*. The *snapshot* isolation is consistently ensured throughout the lifespan of graph data by employing cooperative in-memory lifetime interval-based and the on-disk file snapshot-based multi-version concurrent control scheme.

Concurrent Transaction Processing. As shown in Figure 3, BACH maintains a thread pool for concurrent transaction processing, with each thread responsible for managing a transaction.

All threads share two global epoch counters: GWC for writes and GRC for reads, both of which are initiated to 0 and increase as transactions progress. Each transaction maintains two local parameters: *local write counter* (LWC) and *local read counter* (LRC).

A write transaction goes through three phases: *start, persist* and *commit*. At the *start* phase, a write transaction initiates its LWC and LRC to the current GWC and GRC, respectively. Then, the write transaction will be added to an ongoing write list, and the GWC will increment by one. At the *persist* phase, the write transaction will perform the *Edge operations* (§5) in the target *AL-table* and modify the *Creation_time* or *Deletion_time* of the edge version to its LWC. Where the edge pool is implemented using a lock-free *space-doubling blocked linked list* based on CAS (Compare-and-Swap) operations to facilitate concurrent edge insertions, while the *skip-list* index is also implemented in a lock-free manner. Thereby, the anomalous situation of *dirty writes*, which occurs when a transaction updates a record and another transaction modifies it again before the former commits, is excluded in a purely lock-free way. Then, the modification log entries will be added to a sequential write-ahead log (WAL) and be persisted to stable storage on disk for database recovery. The write transaction is now ready to be committed and moved out from the ongoing write list. Meanwhile, the GRC requires to be updated, as it equals the minimum LWC of transactions in the ongoing write list minus one, thereby making the committed transactions visible to read transactions. However, the GRC may remain unchanged if there are transactions with smaller LWC still being processed. This prevents a transaction from reading a value written by another transaction that has not yet committed (*i.e., dirty reads*), while also ensuring that the LRC of a transaction is always smaller than the LWC of any ongoing transaction.

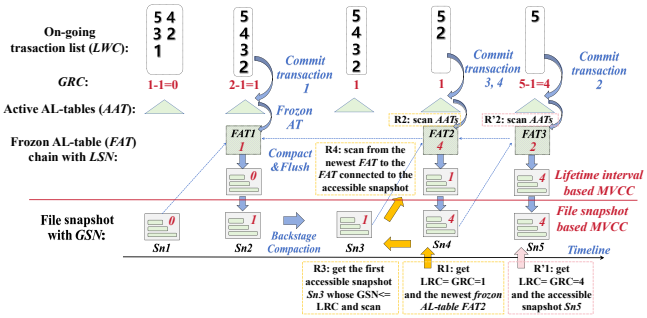


Figure 5: The illustration of co-operative MVCC.

In-memory Lifetime Interval based MVCC. For a read transaction, its LRC will be initiated to the current GRC as it starts. Then it accesses a snapshot of the database determined by its LRC. Specifically, for the search in memory, it only considers the edge versions whose lifetime interval satisfies ((*Creation_time* < LRC) **AND** ((LRC < *Deletion_time*) **OR** (*Deletion_time* < 0))) during the search in target *AL-table*. This prevents BACH from the situation of *read skew* and *phantom reads*, as the modification made by subsequent write transactions is not visible (*i.e.,* the life intervals of edge versions do not satisfy the former condition) to the current read transaction.

On-disk File Snapshot based MVCC. As all files in *GR-LSM-Tree* are immutable and only the compaction process can change the file layout, expensive fine-grained lifetime information is no longer needed. Instead, BACH assigns an accessible file snapshot to each read transaction according to its LRC to ensure the correct retrieval of edge versions. Each file snapshot involves the current state of the in-memory $k-$ary tree, which maintains the physical addresses of each *CSR-table*. Each snapshot also maintains an epoch counter, namely GSN, which is determined when the snapshot is created. Only the compaction (including the flushes of *AL-tables*) can trigger a snapshot evolution. Upon reaching its maximum capability, an *AL-table* will be frozen before being flushed to disk. Each frozen *AL-table* maintains a epoch counter named LSN, which is equal to the largest LWC of the committed transactions that operated on it before the *AL-table* is frozen (*e.g.,* 4 in *FAT2* and 2 in *FAT3* in Figure 5), and will be linked to a chain. When a new snapshot is created by the flush of a frozen *AL-table*, its GSN will be set to *max*{GSN of the last snapshot, LSN of the flushed *AL-table*} (*e.g., 4* in *Sn4* and 4 in *Sn5* shown in Figure 5), and the *frozen AL-table* will be linked to the stale snapshot (*e.g.,* the link between *Sn3* and the *FAT2*). While the new snapshot will directly inherit the GSN of the last snapshot if it is triggered by the backstage compaction (*e.g.,* 1 in *Sn3* shown in Figure 5).

As shown in Figure 5, a read transaction will first get its LRC from the GRC and locate the newest frozen *AL-table*. Then it will begin searching for legal versions in the active *AL-tables* and obtain an accessible file snapshot whose GSN value is less than or equal to its LRC. And the frozen *AL-tables* range from the newest to the one linked to its obtained snapshot are also visible to the read transaction (*e.g., FAT2* in *R4* in Figure 5), where the evaluation of legal versions is based on the lifetime interval as well. Meanwhile, all of edge versions and the tombstones in the assigned accessible snapshot are ensured to have a *Creation_time* less than the LRC of the read transaction. Thus, during the top-to-down search, the first live version (which has not yet encountered its tombstone) of the target edge is ensured to be legal, since its *Deletion_time* is ensured to be larger than the LRC of the read transaction or be less than zero. As for garbage collection, file snapshots (including stale files that do not have snapshot references) and corresponding frozen *AL-tables* will be physically deleted in time at once when there is no ongoing read transaction holding them.

In summary, the *snapshot isolation* is efficiently supported by BACH throughout the lifespan of graph objects since we rule out the anomalous situations of *dirty writes*, *dirty reads*, *read skew* and *phantom reads*. And all the necessary information for controlling the concurrent read/write processing is directly embedded within the inherent data structure in *GR-LSM-Tree* or be maintained by the snapshot mechanism, thereby eliminating the need to build auxiliary data structures and avoiding introducing extra complexity to sequential edge scans.

8 EVALUATION

8.1 Experimental setup

Graph Datasets and Workloads. We employ four simple graph datasets to test the performance of typical graph analytical algorithms including BFS (breadth first search), CDLP (community detection through label propagation), Pagerank, SSSP (weighted shortest paths), WCC (weakly connected components), LCC (local triangle counting) based on the same implementation in the GFE

driver [28]. And two LDBC social network datasets with a scale factor of 30 and 100 are employed to test the performance of complex graph pattern matching with property filtering based on the official driver. Detailed information of different graph datasets is presented in Table 3.

Table 3: Graph dataset information

Dataset	$\|V\|$	$\|E\|$	$Avg.D$	$Max.D$	$Top1\%.D$	Size
Datagen-8_5-fb (DA)	4.6M	332.0M	72	3.8k	783	10.3GB
Graph500-26 (GR)	32.8M	1.1B	32	668.9k	392	18.6GB
Com-friendster (CF)	65.6M	1.8B	28	3.5k	412	32.4GB
UK-2007-05(UK)	105.9M	3.3B	62.8	975.4k	596	58.2GB
LDBC-SF30	88.8M	540.9M	12	920.6k	185	34.4GB
LDBC-SF100	312.0M	1.1B	12	3.0M	194	102.3GB

Competitors. We comprehensively choose four disk-based graph storage engines that support real-time analysis on dynamic graph as competitors: Livegraph [12] (represents the adjacency list based edge logs structure), LLAMA [19] (represents the CSR based snapshots structure), RocksDB [24] (represents the edge list based LSM structure) and Neo4j [29] (represents linked list based adjacency list). The selected competitors encompass the mainstream categories of disk-based dynamic graph storage structures. While Chunk-Graph [30] is employed as a representative of the state-of-the-art static hierarchical CSR format for analytics on simple graphs.

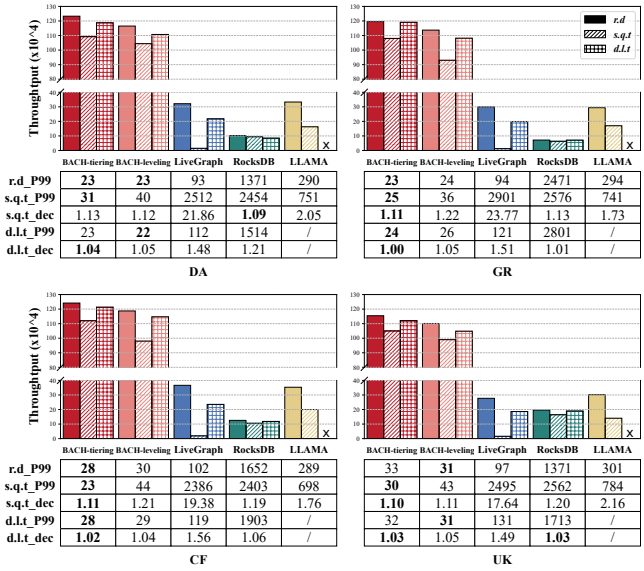
Platform. All experiments are performed on a 64-bit Ubuntu22.04 LTS workstation, equipped with an Intel Xeon E7-8860 @2.2GHz, 384GB DDR4 RAM and 2TB SATA SSD, which can achieve up to 400MBps sequential read/write performance. All codes are programmed in C++ and compiled using GCC v12.3.0 with O3 optimization. The available memory for all systems has been uniformly limited to 16GB in each experiment by using the Linux cgroup tools. The timeout limitation is set to 3.6×10^4 seconds. All systems are allocated 32 threads by default, with 16 threads for edge reads and 16 threads for edge writes. While the LSM-based systems use half of the write threads for backstage compactions.

***GR-LSM-Tree* parameters setting.** The size ratio T is set to 10 by default [24, 26, 27]. $1MB$ limitation is set for each memory-resident *AL-table* to contain edges sourced from up to 2^{16} vertices and the number of operated *CSR-tables* for one compaction operation k is set to 16 at most based on the size of the dataset. The weighted average function for *Elastic merge* is set to $TR^{1/2} + RR$ with 1 as the decision bound to facilitate timely garbage collection and edge lookups, where *TR* represents for *Tombstone ratio* and *RR* represents for *Read ratio*.

8.2 Graph Transactional Processing

Edge Insertion. In the edge insertion experiment, we first evaluate the performance in terms of random access pattern, where the edges are shuffled in a random order and sequentially inserted within each transaction. As shown in Figure 6, BACH significantly outperforms other competitors in terms of both transactional operation throughput and P99 latency for random access. Livegraph[2] [12] exhibits a fine insertion throughput due to its transactional adjacency list-based storage structure. However, the in-place edge insertion will lead to a significant number of random data accesses and reduce

[2] We relax the memory limitation for Livegraph to exclude the influence of its *mmap*-based implementation and focus on the comparison of transactional efficiency of different graph storage structures for the moment.



DA	BACH-tiering	BACH-leveling	LiveGraph	RocksDB	LLAMA
r.d_P99	**23**	**23**	93	1371	290
s.q.t_P99	**31**	40	2512	2454	751
s.q.t_dec	1.13	1.12	21.86	**1.09**	2.05
d.l.t_P99	23	**22**	112	1514	/
d.l.t_dec	**1.04**	1.05	1.48	1.21	/

GR	BACH-tiering	BACH-leveling	LiveGraph	RocksDB	LLAMA
r.d_P99	**23**	24	94	2471	294
s.q.t_P99	**25**	36	2901	2576	741
s.q.t_dec	**1.11**	1.22	23.77	1.13	1.73
d.l.t_P99	**24**	26	121	2801	/
d.l.t_dec	**1.00**	1.05	1.51	1.01	/

CF	BACH-tiering	BACH-leveling	LiveGraph	RocksDB	LLAMA
r.d_P99	**28**	30	102	1652	289
s.q.t_P99	**23**	44	2386	2403	698
s.q.t_dec	**1.11**	1.21	19.38	1.19	1.76
d.l.t_P99	**28**	29	119	1903	/
d.l.t_dec	**1.02**	1.04	1.56	1.06	/

UK	BACH-tiering	BACH-leveling	LiveGraph	RocksDB	LLAMA
r.d_P99	33	**31**	97	1371	301
s.q.t_P99	**30**	43	2495	2562	784
s.q.t_dec	**1.10**	1.11	17.64	1.20	2.16
d.l.t_P99	32	**31**	131	1713	/
d.l.t_dec	**1.03**	1.05	1.49	**1.03**	/

Figure 6: The GTP throughput comparison (*r.d* denotes random pattern, *s.q.t* denotes sequential pattern, *d.l.t* denotes deletion, *P99* denotes the P99 latency in microsecond and *dec* denotes performance decline rate of average throughput)

the overall throughput. While other systems follow the *out-of-place* data ingestion paradigm, which enables batched and sequential I/Os by buffering edge writes in memory first. However, LLAMA [19] processes edge insertions by taking incremental snapshots in CSR format, which is limited to be flushed to disk in serial to maintain the update orders. And the simple key value-based memory structure employed by RocksDB [24] will significantly increase the cost of sorting and allocating inserted edges for a huge part of the graph, while the adopted group commit mechanism will sometimes choke the transactional processing as well. In contrast, Bach optimizes the memory-resident structure by partitioning it into smaller transactional *ADJ-tables* with fixed source vertex range to provide better edge locality and higher insert parallelism. And the *tiering-like* compaction outperforms the *leveling-like* compaction in BACH due to its lower write amplifications and reduced disk flush latency. As a result, *tiering* merge based BACH consistently achieves an average TP throughput of over 1 million across all tests, significantly outperforming all other competitors.

Sequential Access Pattern. Numerous real-world applications require to process intensive edge insertions in terms of sequential access pattern (*e.g.*, a sudden burst of likes to a video). Where the edge insertions within a period of time are highly related to the same group of vertices and exhibit a strong temporal locality [21]. In this experiment, most competitors experienced notable declines in performance due to the obstruction of write transactions. Livegraph encounters the most severe degradation, as it requires a lock of the entire adjacency list when processing edge insertion and necessitates resizing the edge transactional files upon reaching its full capacity. As for LLAMA, the sequential insertions will cause a more significant data movement to modify the CSR-like array within an incremental snapshot, leading to a considerable degradation as well. RocksDB demonstrates moderately low, yet consistent throughput in sequential access pattern, attributed to its absence

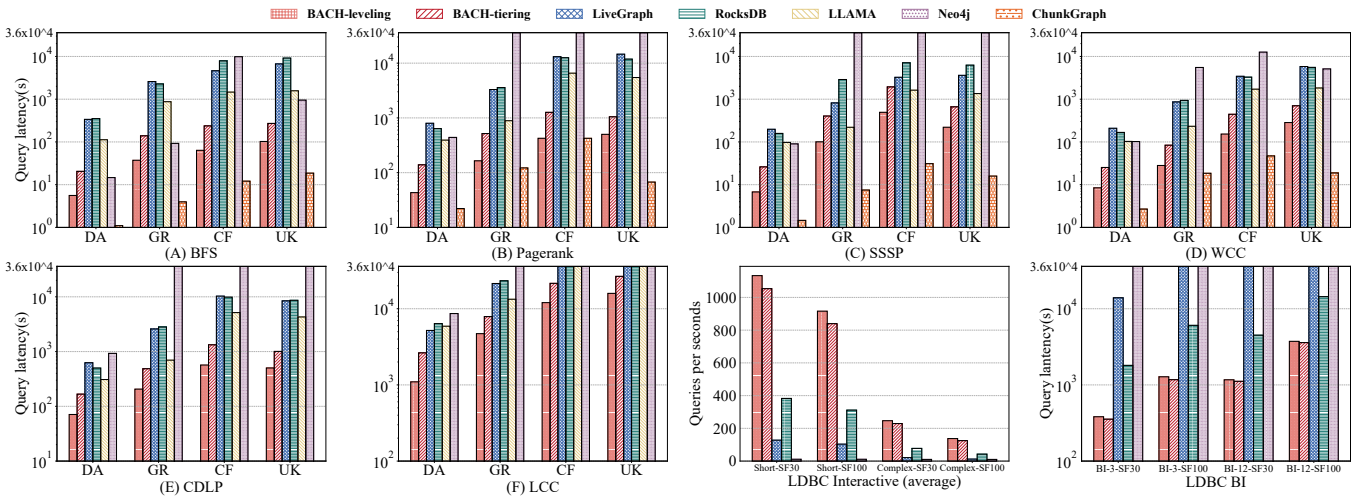


Figure 7: The Graph Analytical Processing Performance comparison

of graph-aware structures in basic key-value storage. BACH variants sustain fairly high transaction throughput due to the lock-free *AL-table* implementation, which substantially reduces the block of edge writes in sequential pattern. However, the *leveling-like* merge experiences a more significant performance decline and higher tail latency compared to the *tiering-like* merge, as evidenced by both throughput and P99 latency. This is attributed to its higher merging costs, which can potentially lead to more frequent write stalls in temporal intensive insertions within certain vertex ranges.

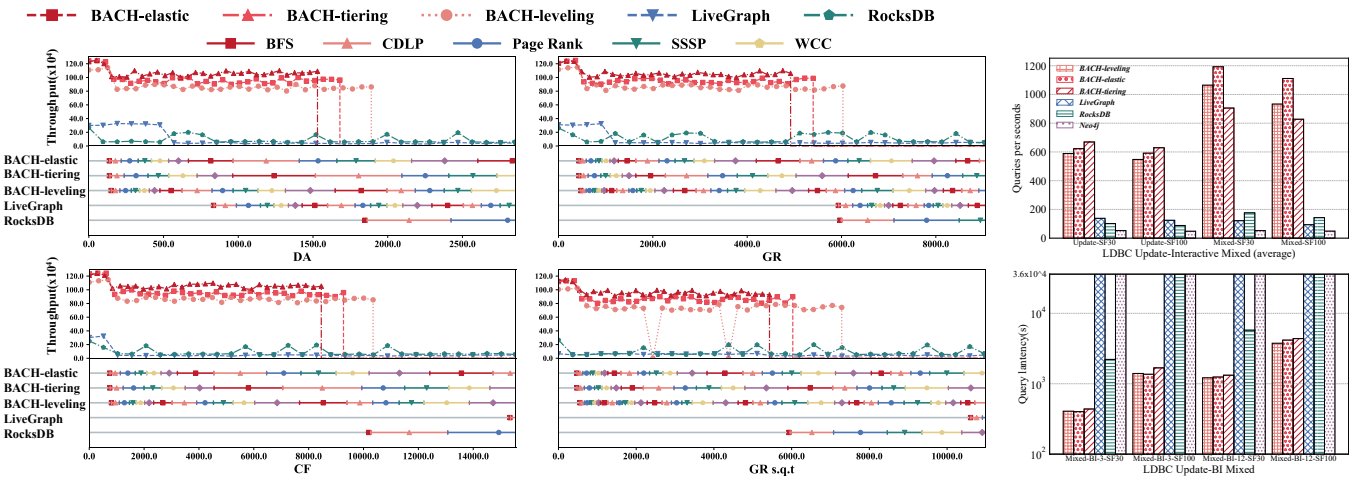
Edge Updates. We further test the GTP performance when incorporating edge updates in random edge insertions. Bach variants remain minimally affected by this issue, as we employ the lifetime interval-based MVCC scheme in conjunction with tombstone-based edge deletions to facilitate concurrent edge writes. This approach enables append-only edge writes and effectively mitigates memory write amplifications. RocksDB remains stable, but relatively low throughput due to its similar tombstone-based deletion mechanism and the naive key value-based memory structure. While Livegraph experiences performance degradation caused by an expensive search for the target edges in the TEL structures during the in-place modifications. LLAMA is excluded from the evaluation due to its inability to accurately process edge deletions on disk [31].

In summary, Bach variants exhibit the most excellent and stable graph transactional throughput in all tests due to its efficient lock-free implementation of the graph-aware memory-resident *AL-table*. And the *tiering-like* merge-based BACH achieves an average 1.08x (up to 1.21x) higher transactional throughput than *leveling-like* merge, 34.12x (up to 85.31x) higher than Livegraph, 5.15x (up to 7.51x, expect edge deletions) higher than LLAMA and 12.73x (up to 18.96x) higher than RocksDB.

8.3 Graph analytical Processing

Graph Analytics. ChunkGraph [30] and Neo4j [29] are incorporated in the static graph analytic tests. In this process, we thoroughly evaluate the average latency of executing the six typical graph algorithms in five runs. These algorithms typically involve intensive sequential/random vertices accesses followed by a sequential scan to their adjacency edges. The results show that *leveling-like* merge-based BACH exhibits performance comparable to static Chunkgraph and significantly outperforms other competitors in all algorithm tests. Compared to snapshot-based LLAMA and link-list-based Neo4j, BACH effectively reduces the significant number of random I/O operations caused by accessing multiple snapshot files or the costly pointer hopping in neighborhood scan. As Bach will only introduce L (height of the tree, which is less than 5 in all tests, extra T runs for *tiering-like* merge) random I/Os at the worst case. And this process can be further accelerated by using the memory-cached vertex bitmap to prune unnecessary file accesses. Although RocksDB theoretically causes the same scale of random I/O operations as BACH, the efficient edge indexing capability of CSR format embedded within the *GR-LSM-Tree* provides a significant advantage over RocksDB when it comes to intensive neighborhood scans. Livegraph, although theoretically capable of ensuring a pure sequential neighborhood scan, experiences significant performance degradation due to frequent memory-disk swapping of *mmap* files when it exceeds the memory restriction. This limitation is endogenous, as the frequent resizing of TEL files and the in-place modifications required by Livegraph will lead to heavy I/O costs and severe write amplifications, which are barely intolerable without the *mmap*-based implementation in real-world scenarios. Overall, the *leveling-like* merge-based BACH achieves an average speedup of 3.25x (up to 3.94x) *tiering merge*, 22.52x (up to 68.94x) over Livegraph, 34.98x (up to 96.16x) over RocksDB, 6.52x (up to 15.50x) over LLAMA, and 78.62x (except timeout) over Neo4j, while average 3.13x (minimum to 1.04x) slower than static ChunkGraph.

Subgraph matching in property graphs. To evaluate the performance of more complex graph analytical processing tasks that incorporate subgraph matching with predicates on edge properties, both interactive and business intelligence (BI) workloads are conducted on the LDBC SF30 and SF100 datasets. All 1-14 complex reads and 1-7 short queries in the interactive LDBC workload are evaluated, along with three BI queries 3, 12 that cover most choke points with respect to complex joins and predicates [32].



(a) Real-time Graph Analytics

(b) HGTAP in property graphs

Figure 8: Hybrid Graph Analytical/Transactional Processing Performance

The efficient neighborhood scan of BACH enables it to consistently outperform all competitors in interactive workloads, showcasing its superior performance. And the advantages of BACH become even more pronounced in BI tests that involve intensive scan-based filters on all edges (*i.e.,* long range scan). This is attributed to the compact *CSR-tables* and the enforced *leveling* compaction in the last level, ensuring a purely sequential scan of edge properties and maintaining the same complexity for long range lookups as static CSR format. While the performance of the adjacency list-based graph storage solutions will significantly degrade as a substantial number of random I/Os are caused by cross-neighborhood accesses.

8.4 Hybrid Analytical/Transactional Processing

Real-time Graph Analytics. As for the hybrid workload, we first evaluate the real-time graph analytics on simple graphs. During the experiment, the first 20% graph transactions continuously insert the edges and the rest 80% transactions involve balanced edge insertions and deletions to keep the scale of graph. All systems will begin to cyclically perform the graph algorithms as soon as they complete the first 10% transactions. The time-varying tendency of transaction throughput and analytic latency is shown in Figure 8. The BACH variants exhibit consistently superior performance in terms of both transaction throughput and analytical latency throughout the entire lifespan of graph objects. While the *mmap*-based Livegraph demonstrates strong GTP throughput and GAP performance at the beginning of the experiment, as all data are processed in memory. However, when memory usage approaches the controlled limitation, the transaction throughput of Livegraph experiences a significant degradation from about four hundred thousand ops to a few thousand ops. And the GAP latency also starts to increase drastically due to the frequent random I/Os caused by the *mmap*-files swapping between memory and disk. While RocksDB demonstrates a relatively stable transaction throughput in the medium position due to its *leveling* merge-based LSM structure and edge list-based key value model. However, it remains a relatively high analytic latency due to the lack of graph-aware storage structures. In addition to BACH's efficient underlying data layout, the excellent performance in HGTAP is also attributed to its lightweight snapshot-based MVCC and timely garbage collection mechanism. As garbage collection is inherently included in the backstage compactions, and the redundant computations of filtering lifetime interval to discard stale edge versions are efficiently excluded by the disk-based snapshot mechanism. While other MVCC-based dynamic graph storage engines typically employ an extra thread to periodically scan the entire graph to collect stale versions, which will disturb the processing of foreground workloads and damage the performance of HGTAP.

HGTAP with sequential insertion pattern. Sequential access patterns also play a crucial role in real-world HGTAP scenarios. It poses more significant challenges for graph storage engines, as it may incur severe chokes in transaction processing and further block the analysis. In this experiment, we replace random edge insertions in the *GR* dataset with sequential access pattern, while keeping the other settings unchanged. As shown in Figure 8 (a), the BACH variants continue to maintain exceptional performance and further widen the gap with other competitors due to the pure lock-free implementation of the memory buffering structure. However, the trade-offs between different merge polices in BACH have become more significant in this test. As shown in Figure 8, the *tiering-like* compaction policy achieves an average 1.73x improvement of GTP throughput, but an average 3.05x degradation of GAP latency than the *leveling-like* compaction. This is because, with limited memory, the more serious write amplification of temporary intensive compactions within a key range will significantly impede the transactional processing. Since it will block the data flushes from memory-resident structure to disk and require page swapping with virtual memory to make room for new edge insertions. The same situation is revealed in the *leveling* merge-based RocksDB. However, *tiering-like* merge will significantly increase the GAP latency due to the existence of files with overlapping key ranges. The *Elastic merge* seamlessly combines the strength of the two compaction policies by enabling each *CSR-table* to flexibly choose between the two compaction policies based on running workloads. As a result, *Elastic merge* shows a competitive GTP throughput (only

1.12x lower in average) to *tiering-like* compaction while remaining a close GAP latency (only 1.76x higher in average) to *leveling-like* compaction, and efficiently avoid temporary write stalls caused by temporary intensive compactions.

HGTAP in property graphs. We next evaluate the performance when incorporating complex subgraph matching with heavy joins and predicates on edge properties in HGTAP using the LDBC 30 and LDBC 100 dataset. The result is shown in Figure 8 (b), where the update-only QPS represents the throughput during edge insertions, and the mixed QPS represents the average throughput incorporating updates with interactive queries (including short queries 1-7 and complex queries 1-14) based on the official shuffling setting. The BI latency that incorporates intensive updates is also measured. The results have reinforced the outstanding HGTAP capabilities of BACH, demonstrating the highest throughput across every test compared to its competitors, with the least performance decline due to heavy concurrent read/write operations. Moreover, its performance has remained consistent on large-scale data, thanks to the embedded CSR-based layout that facilitates purely sequential edge scans. While other adjacency list-based competitors have demonstrated significant performance declines during complex BI tests on larger graphs due to the substantial increase in random I/Os required for cross-neighborhood scans. The superior mixed QPS of *elastic-merge* further confirms its strengths in optimizing the trade-offs between write amplification and read performance, enabling BACH to possess the self-tuning ability to smartly adapt to diverse workloads and skew vertex degree distributions.

9 RELATED WORK

Storage for Dynamic Graphs. Numerous graph storage formats have been developed to enhance the performance of graph transactional and analytical processing. Adjacency list-based [12, 16, 22] structures facilitate rapid and efficient update operations. They are particularly well-suited for maintaining dynamic graphs, as they only require local structural maintenance during transactional operations. LiveGraph [12] devises the Transactional Edge Log (TEL) structure based on the adjacency list to provide snapshot isolation while ensuring sequential neighborhood scans, but failed to maintain the sequentiality of edge scans when processing the long range lookups and is limited by its *mmap*-based implementation. Sortledton [16] employs adjacency list based structures to support graph pattern matching by computing intersections in linear time, but the performance will be gradually degraded as the versions accumulative due to the link list based MVCC. GraphOne [22] transforms the edge list into snapshots of the static adjacency list in batches for fast graph analysis, resulting in a lower data freshness. CSR-based structures [18–20, 33], store graph data in compact arrays to ensure purely sequential edge scans, leading to superior graph analytical performance, but incur great write amplifications when processing graph modifications. LLAMA [19] maintains an incremental CSR-based snapshot for dynamic graphs. VCSR [34] and PCSR [33] use memory packed array variant structures with reserved slots to reduce the update overheads, and Teseo [20] further combines the similar structure within B^+-tree variants to facilitate efficient HGTAP with full transactional support. Spruce [21] employs the tree-like multilevel structure to provide enhanced performance with lower memory footprints. In this paper, we leverages LSM-trees to seamlessly bridge the adjacency lists and CSR formats, harnessing their complementary strengths to enhance the efficiency of disk-based HGTAP workloads.

The Design Space of LSM-Tree. LSM-trees are widely used in various storage systems, such as HBase [35], LevelDB [36], and RocksDB [24], to flush in-memory caches to disk. The compaction policy greatly influences the performance of LSM based systems by determining their write amplification and point and range lookups performance [37]. WriteBuffer Tree [38] organizes the SSTable groups into a B^+-tree-like structure for self-balancing and employs hash partitioning to achieve workload balance. Dayan et al. [26, 39] collaborates the *tiering* and *leveling* policy at level granularity to achieve better performance trade-offs. Real-time LSM-Tree [40] firstly utilizes the compaction process to change the storage layout from row-based to column-oriented for relational HTAP. Sarkar et al. [27] systematically construct the design space of compaction policy for LSM-based systems. While BACH targets at optimizing the compaction policy to adapt to diverse graph workloads and skew vertex degree distributions by introducing *elastic merge* policy, which can provide better trade-offs between edge write amplifications and lookup latency in HGTAP workloads.

Graph Databases. Numerous graph databases [13, 29, 41, 42] have been developed and are widely utilized in industrial applications. Due to the need for high reliability in supporting graph transactions, analytical processing can incur significant overhead. Neo4j [29] uses adjacency lists to store the graph structure, with vertices, edges, and properties stored as pointers on disk, resulting in a significant number of random disk I/Os when processing graph analysis. Some graph databases also employ LSM-Tree-based structure to facilitate concurrent read/write processing, such as NebulaGraph [41], Dgraph [42] and ByteGraph [13]. However, they basically employ the simple edge list based key-value model and the naive implementation of LSM-Tree-based storage engines, resulting in a lack of in-depth optimization towards the characteristics of HGTAP workloads. Instead, BACH constructs the *Graph-aware Real-time* LSM-Tree, which carefully expands the design space of LSM-Trees for HGTAP by integrating the graph-aware sequential structure and devising a customized compaction policy.

10 CONCLUSION

This paper proposes *GR-LSM-Tree*, which expands the design space of LSM-Trees for the first time to bridge the transactional adjacency list and the analytical CSR format for efficient HGTAP. The two graph-aware structures are seamlessly embedded within the inherent framework of LSM-trees and gradually transformed through the data propagation to deeper levels by a novel fine-grained compaction policy. Extensive evaluations confirm the strength and efficiency of *GR-LSM-Tree* in HGTAP workloads.

11 ACKNOWLEDGMENTS

We sincerely thank the anonymous reviewers for their valuable suggestions. This work was partially supported by the National Natural Science Foundation of China grant 62372138, Natural Science Foundation of Heilongjiang Province of China grant HSF20230095 and 2024ZXJ01A04.

REFERENCES

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

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

[3] Jizhe Wang, Pipei Huang, Huan Zhao, Zhibo Zhang, Binqiang Zhao, and Dik Lun Lee. Billion-scale commodity embedding for e-commerce recommendation in alibaba. In *Proceedings of the 24th ACM SIGKDD international conference on knowledge discovery & data mining*, pages 839–848, 2018.

[4] Janneth Chicaiza and Priscila Valdiviezo-Diaz. A comprehensive survey of knowledge graph-based recommender systems: Technologies, development, and contributions. *Information*, 12(6):232, 2021.

[5] Avery Ching, Sergey Edunov, Maja Kabiljo, Dionysios Logothetis, and Sambavi Muthukrishnan. One trillion edges: Graph processing at facebook-scale. *Proceedings of the VLDB Endowment*, 8(12):1804–1815, 2015.

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

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

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

[9] Ziniu Hu, Yuxiao Dong, Kuansan Wang, and Yizhou Sun. Heterogeneous graph transformer. In *Proceedings of the web conference 2020*, pages 2704–2710, 2020.

[10] Will Hamilton, Zhitao Ying, and Jure Leskovec. Inductive representation learning on large graphs. *Advances in neural information processing systems*, 30, 2017.

[11] Renzo Angles, Marcelo Arenas, Pablo Barceló, Aidan Hogan, Juan Reutter, and Domagoj Vrgoč. Foundations of modern query languages for graph databases. *ACM Computing Surveys (CSUR)*, 50(5):1–40, 2017.

[12] Xiaowei Zhu, Guanyu Feng, Marco Serafini, Xiaosong Ma, Jiping Yu, Lei Xie, Ashraf Aboulnaga, and Wenguang Chen. Livegraph: A transactional graph storage system with purely sequential adjacency list scans. *Proceedings of the VLDB Endowment*, 13(7), 2020.

[13] Changji Li, Hongzhi Chen, Shuai Zhang, Yingqian Hu, Chao Chen, Zhenjie Zhang, Meng Li, Xiangchen Li, Dongqing Han, Xiaohui Chen, et al. Bytegraph: a high-performance distributed graph database in bytedance. *Proceedings of the VLDB Endowment*, 15(12):3306–3318, 2022.

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

[15] Guoliang Li and Chao Zhang. Htap databases: What is new and what is next. In *Proceedings of the 2022 International Conference on Management of Data*, pages 2483–2488, 2022.

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

[17] Pranjal Gupta, Amine Mhedhbi, and Semih Salihoglu. Columnar storage and list-based processing for graph database management systems. *Proceedings of the VLDB Endowment*, 14(11):2491–2504, 2021.

[18] Soukaina Firmli, Vasileios Trigonakis, Jean-Pierre Lozi, Iraklis Psaroudakis, Alexander Weld, Dalila Chiadmi, Sungpack Hong, and Hassan Chafi. Csr++: A fast, scalable, update-friendly graph data structure. In *24th International Conference on Principles of Distributed Systems (OPODIS'20)*, 2020.

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

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

[21] Jifan Shi, Biao Wang, and Yun Xu. Spruce: a fast yet space-saving structure for dynamic graph storage. *Proceedings of the ACM on Management of Data*, 2(1):1–26, 2024.

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

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

[24] RocksDB. http://rocksdb.org/, 2023.

[25] Chen Luo and Michael J Carey. Lsm-based storage techniques: a survey. *The VLDB Journal*, 29(1):393–418, 2020.

[26] Niv Dayan and Stratos Idreos. 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*, pages 505–520, 2018.

[27] Subhadeep Sarkar, Dimitris Staratzis, Ziehen Zhu, and Manos Athanassoulis. Constructing and analyzing the lsm compaction design space. *Proceedings of the VLDB Endowment*, 14(11), 2021.

[28] Sortledton. https://github.com/perfuchs/gfe_driver, 2023.

[29] Neo4j. https://neo4j.com/, 2023.

[30] Rui Wang, Weixu Zong, Shuibing He, Xinyu Chen, Zhenxin Li, and Zheng Dang. Efficient large graph processing with {Chunk-Based} graph representation model. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)*, pages 1239–1255, 2024.

[31] LLAMA. https://github.com/goatdb/llama, 2015.

[32] Renzo Angles, János Benjamin Antal, Alex Averbuch, Altan Birler, Peter Boncz, Márton Búr, Orri Erling, Andrey Gubichev, Vlad Haprian, Moritz Kaufmann, et al. The ldbc social network benchmark. *arXiv preprint arXiv:2001.02299*, 2020.

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

[34] Abdullah Al Raqibul Islam, Dong Dai, and Dazhao Cheng. Vcsr: Mutable csr graph format using vertex-centric packed memory array. In *2022 22nd IEEE International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, pages 71–80. IEEE, 2022.

[35] Mehul Nalin Vora. Hadoop-hbase for large-scale data. In *Proceedings of 2011 International Conference on Computer Science and Network Technology*, volume 1, pages 601–605. IEEE, 2011.

[36] LevelDB. https://github.com/google/leveldb, 2023.

[37] Subhadeep Sarkar and Manos Athanassoulis. Dissecting, designing, and optimizing lsm-based data stores. In *Proceedings of the 2022 International Conference on Management of Data*, pages 2489–2497, 2022.

[38] Hrishikesh Amur, David G Andersen, Michael Kaminsky, and Karsten Schwan. Design of a write-optimized data store. 2013.

[39] Niv Dayan and Stratos Idreos. The log-structured merge-bush & the wacky continuum. In *Proceedings of the 2019 International Conference on Management of Data*, pages 449–466, 2019.

[40] Hemant Saxena, Lukasz Golab, Stratos Idreos, and Ihab F. Ilyas. Real-time lsm-trees for htap workloads. *2023 IEEE 39th International Conference on Data Engineering (ICDE)*, pages 1208–1220, 2021.

[41] NebulaGraph. https://www.nebula-graph.com.cn/, 2023.

[42] Dgraph. https://dgraph.io/, 2023.