Don't Forget Range Delete! Enhancing LSM-based Key-Value Stores with More Compatible Lookups and Deletes

Fan Wang
fan008@e.ntu.edu.sg
Nanyang Technological University
Singapore

Dingheng Mo
dingheng001@e.ntu.edu.sg
Nanyang Technological University
Singapore

Siqiang Luo
siqiang.luo@ntu.edu.sg
Nanyang Technological University
Singapore

ABSTRACT

LSM-trees are featured by *out-of-place* updates, where key deletion is handled by inserting a tombstone to mark its staleness instead of removing it in place. This defers actual removal to compactions with greatly reduced overhead. However, this classic strategy struggles with another fundamental operator–range deletes–which removes all keys within a specified range, requiring the system to insert numerous tombstones and causing severe performance issues.

To address this, modern LSM-based systems introduce range tombstones that record the start and end keys to avoid per-key tombstones. Although this strategy achieves impressive range delete efficiency, we show that such de facto industrial range delete solution is *incompatible* with lookup operators. In particular, our experiments show that point lookup latency can increase by 30% even with just 1% range deletions in the workload. Further to our surprise is that this issue has not been raised before, although the range tombstone solution has been employed for more than five years.

To address this critical system performance issue, we propose GLORAN, an efficient range delete method that can be integrated into modern LSM-based systems and offers desirable range deletion performance without compromising point lookup efficiency. It introduces a global index to efficiently manage deleted ranges. This index allows point lookups to quickly locate the relevant deleted ranges without retrieving many irrelevant elements, reducing the I/O complexity from $O(\frac{N}{\lambda})$ to either $O(\log^2 \frac{N}{\lambda F})$ or $O(\varphi \cdot \log \frac{N}{F})$, where $1/\lambda$ is the ratio of range deletes, and φ is the FPR of Bloom filters in LSM-trees. Furthermore, we design an entry validity estimator to further enhance expected I/O cost to $O(\varepsilon \cdot \log^2 \frac{N}{\lambda F})$ for looking up existing keys. Extensive evaluations demonstrate that GLORAN consistently outperforms baseline approaches, while achieving up to 10.6× faster point lookups and 2.7× higher overall throughput compared to the state-of-the-art method.

PVLDB Reference Format:
Fan Wang, Dingheng Mo, and Siqiang Luo. Don't Forget Range Delete! Enhancing LSM-based Key-Value Stores with More Compatible Lookups and Deletes. PVLDB, 14(1): XXX-XXX, 2020.
doi:XX.XX/XXX.XX

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://anonymous.4open.science/r/GLORAN-56FF.

Proceedings of the VLDB Endowment, Vol. 14, No. 1 ISSN 2150-8097.
doi:XX.XX/XXX.XX

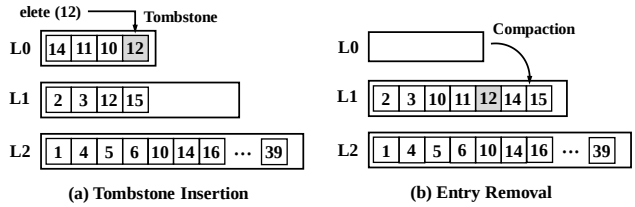

Figure 1: The LSM-tree employs out-of-place deletion with tombstones, and removes the entry during compaction.

1 INTRODUCTION

The Log-Structured Merge-tree (LSM-tree) has been widely employed as the backbone of many renowned key-value stores such as RocksDB [18], Cassandra [31], AsterixDB [4], ScyllaDB [52], CockroachDB [30], and InfluxDB [26]. It organizes data as key–value pairs, or *entries*, across multiple levels with exponentially increasing capacities. The smallest level, L_0, resides in memory, while all other levels are stored on disk, with entries sorted by key. New entries are first inserted into L_0 and progressively migrated to larger levels through compaction processes. The LSM-tree supports efficient key deletion through an *out-of-place* approach. Instead of finding the key and removing it in place, it inserts a special entry, a *tombstone*, to conceptually mark the key as invalid. The obsolete entries remain until purged by the tombstone during subsequent compactions as Figure 1 shows. This efficiently handles single key deletions, while, still falling short in supporting range deletes.

A range delete removes all keys within a certain range from the LSM-tree. This operation is naturally utilized in modern applications and has been incorporated by many renowned LSM-based key-value stores like RocksDB and Cassandra. For example, in e-commerce platforms, time-limited promotional activities may assign a special prefix to product codes, grouping all promotional items within a specific key range. Once the activity ends, removing the associated data requires efficient range delete. Similar needs arise in other applications, such as purging time-bound data in time series databases [27] or discarding outdated dataset versions in machine learning pipelines.

Surprising Impact of Range Delete. While LSM-tree optimizations have been extensively studied, the impact of range delete remains underexplored. While one may not expect a heavy portion of range deletes in key-value store workloads, to our surprise, our benchmark shows that even a tiny range delete portion (e.g., 1%) can significantly degrade the performance of *other operations* (e.g., *point lookup* that retrieves the entry for a specific key). This calls for rethinking the system design regarding range deletes. Naively, implementing range delete in an LSM-tree requires inserting a

tombstone for each key within the target range. This incurs substantial insertion overhead and additional costs for the subsequent compactions, especially for long ranges. To mitigate this, the state-of-the-art method introduces *range tombstone*. It represents an entire key range with a single entry by encoding both the start and end keys. With this design, only one entry is inserted per range delete, thus significantly reducing the I/O cost. As a result, this method is widely adopted in modern LSM-based key-value stores such as RocksDB and CockroachDB. While this approach delivers outstanding performance for range deletes, it may significantly drag down the point lookup performance. In our experiments, *in a workload without range deletes, replacing just 1% of operations with range deletes increased point lookup latency by more than 30% in RocksDB. The point lookup performance can even drop by 2.3 times if the range delete workload occupies 5%*. Despite such significant impact, the issue has not been explicitly studied. In this paper, we conduct a thorough analysis of this issue and propose a novel approach that effectively improves point lookup efficiency while delivering desirable range delete performance.

The state-of-the-art design introduces a dedicated block at each LSM-tree level to accommodate the range tombstones. This local storage layout leads to substantial and often unnecessary overhead for point lookups. In order to find a specific key, the system searches the LSM-tree levels from top to bottom. At each level, the corresponding range tombstone block must also be checked to determine whether the key has been deleted. Many levels, however, do not contain the target key, leading to unnecessary accesses of the range tombstone blocks. Furthermore, within a certain block, point lookups would retrieve numerous unrelated range tombstones, incurring high I/O costs. Although range tombstones are sorted by their start key, their variable lengths prevent the system from knowing the exact key range coverage before retrieving them. As a result, every tombstone with a start key smaller than the target key must be examined, which greatly increases lookup cost.

To address these inefficiencies effectively, we propose managing the deleted ranges within a global index. To enhance clarity, we refer to the deleted range issued by a range delete operation as its range record. The global index captures the coverage information of range records and organizes them accordingly. Unlike the state-of-the-art method that indexes ranges solely by their start key, our approach enables efficient locating of the relevant ranges for a target key while skipping large amounts of irrelevant ones. Moreover, the global index can be bypassed when the target key is absent from the LSM-tree, thereby reducing the overhead. Hence, this design leads to obviously improved point lookup efficiency while retaining desirable range delete support.

Constructing such a global index is by no means straightforward. If range tombstones are stored naively, their temporal information is lost, which may cause incorrect deletion of keys that arrive after the range deletion. To solve this problem, we introduce a two-dimensional representation for range records. Each record explicitly encodes both the key boundaries and the time boundaries of a deletion. We then present an efficient spatial structure, LSM-DRtree, as the global structure for managing range records. By properly exploiting the intrinsic spatial relationships among range records, this structure effectively eliminates overlaps within the index, thus achieving impressive point lookup performance. Our theoretical analysis shows that, compared with the state-of-the-art design, the point lookup cost can be reduced from $O(\frac{N}{\lambda})$ to $O(\log^2 \frac{N}{\lambda F})$, where $\frac{N}{\lambda}$ is the number of range deletes. At the same time, the LSM-style structure ensures its update efficiency that satisfies the performance of range deletes. It also enables the index to operate in minimal memory overhead, with the buffer size limited to 4 MB in our experiments (and one can foresee performance upgrade when more buffer memory is allowed). Beyond the global index, we enhance the design with an entry validity estimator to further speed up point lookups. We summarize our contributions as follows.

Term	Definition	Unit
F	Capacity of memory buffer of an LSM-tree	entries
B	Size of data blocks	bytes
k	Size of key in an LSM-tree	bytes
e	Size of entry in an LSM-tree	bytes
φ	False positive rate of Bloom filter	
ϵ	False positive rate of extendable validity filter	
N	Number of entries in an LSM-tree	
Q	Number of range deletes	
ℓ	Average length of deleted range	
L	Number of levels in an LSM-tree	
T	Size ratio of an LSM-tree	

Table 1: List of terms used throughout the paper.

- To our knowledge, the impact of range delete has not been comprehensively reviewed in the literature. We provide a first comprehensive quantitative analysis of the range delete overhead and its impact on other operations under various existing methods.
- We propose GLORAN, a novel range delete strategy that organizes range records globally, achieving both efficient range delete and high-performance lookups.
- We design an efficient global index structure, LSM-DRtree, which combines fast range deletions with low-latency point lookups. Together with a lightweight entry validity estimator, it reaches further enhanced lookup performance.
- We conduct extensive evaluations under diverse workloads, where GLORAN consistently delivers competitive performance, achieving up to 2.7× higher throughput than state-of-the-art methods.

2 BACKGROUND

This section introduces the basic operations of LSM-trees and analyzes their associated costs under workloads without range deletes. This prepares readers for the next section, where we discuss how the range deletes impact the workflow and cost of these operations. Following prior works [13, 23, 34], the operational costs are evaluated as the number of I/Os, with terms used throughout the paper summarized in Table 1 for ease of reference.

Update the LSM-tree. The LSM-tree consists of multiple levels, among which L_0 resides in the main memory serving as a write buffer with the capability of accommodating F entries. For the on-disk levels, their capacity is T times larger than their upper level, where T is the size ratio. In other words, the capacity of the i-th level is $F \cdot T^i$. Therefore, to store N entries, $L = \log_T(\frac{N}{F})$ levels should be included. To update the LSM-tree, a new entry is inserted into the write buffer that can eventually be compacted to the bottommost

Operation	LRR (SOTA)	GLORAN (ours)
RangeDelete	$O\left(\frac{k}{B} \cdot T \cdot \log \frac{N}{F}\right)$	$O\left(\frac{k}{B} \cdot T \cdot \log\left(\frac{1}{\lambda} \cdot \frac{N}{F}\right)\right)$
Lookup (V)	$O\left(\frac{Nk}{\lambda B} + \lceil\varphi\rceil \log \frac{N}{F} + 1\right)$	$O\left(\varepsilon \log^2 \frac{N}{\lambda F} + \lceil\varphi \log \frac{N}{F}\rceil\right)$
Lookup (N)	$O\left(\frac{Nk}{\lambda B} + \lceil\varphi\rceil \log \frac{N}{F}\right)$	$O\left(\varphi \log \frac{N}{F}\right)$
Lookup (O)	$O\left(\frac{Nk}{\lambda B} + \lceil\varphi\rceil \log \frac{N}{F}\right)$	$O\left(\log^2 \frac{N}{\lambda F} + \lceil\varphi \log \frac{N}{F}\rceil\right)$

Table 2: Operational cost of different range delete methods, where V, N, and O represent lookups on keys that are valid, non-existent in the LSM-tree, and obsoleted by range deletes while not removed from LSM-tree. In this table, all logarithmic terms are taken with base T. GLORAN delivers better lookup performance than the SOTA method across diverse cases that can reduce the cost from $O\left(\frac{N}{\lambda}\right)$ to $O\left(\log^2\left(\frac{N}{\lambda F}\right)\right)$. It also enables enhanced range delete, which matches the performance of LLR with only $F/16$ memory assigned to the global index in our experiments.

level. Therefore, an entry can be rewritten up to $O(T \cdot L)$ times. The entries are processed by sequential read/write that handles data in the granularity of disk data blocks with size of B. Hence, the update cost is $O(T \cdot L \cdot \frac{e}{B})$ I/Os, where e is the size of an entry. Here, we adopt a typical LSM design, leveling configuration, in our analysis for clarity and readability. We also provide associated analysis on other configurations in our technical report [1].

Point Lookup. This operation retrieves the entry with a target key from the LSM-tree through a level-by-level search. To accelerate this process, modern LSM-trees introduce a fence pointer for each data block that records the associated key range. Since the entries are sorted in the disk levels, this facilitates identifying the data block that may contain the target key in each level and retrieving it with a single I/O. In addition, the Bloom filters further reduce I/O cost by predicting the presence of a matched key in a certain level without false negative results. Hence, we can safely skip the levels when the Bloom filter returns false to save I/O. We denote the false positive rate of a Bloom filter as φ. Then, if there is no matched key in the LSM-tree, the operation should look through all levels with a cost of $O(\varphi \cdot L)$. If the matched key does exist, it requires an additional I/O to retrieve the entry, counting $O(1 + \varphi \cdot L) = O(\lceil\varphi \cdot L\rceil)$.

Point Delete. This operation removes a specific key from the LSM-tree using an out-of-place strategy. Instead of directly locating and immediately deleting obsolete entries, the system inserts a *tombstone* into the memory buffer to mark the target key as invalid. A tombstone is a special entry that carries only the key without an associated value. During flush or compaction, it is processed like a normal entry and used to discard obsolete entries containing the same key. Moreover, since a key can have multiple outdated versions across levels, the tombstone must be retained until its expiration when reaching the bottommost level, which incurs an I/O cost of $O(T \cdot L \cdot \frac{k}{B})$. Besides, point lookups can stop at tombstones as they confirm the non-existence of target keys.

3 THE IMPACT OF RANGE DELETES

While the previous analyses give hints on the performance of point-lookup/delete and update operations, these analyses become inaccurate with the consideration of range deletes. Hence, this section further presents an analysis of the associated costs *in the presence of range deletes*, which have not been formally studied before. To this end, we detail the workflow of relevant operations and quantitatively evaluates their cost as summarized in Table 2. These findings not only offer a deeper understanding of the range delete operation but also inspire the design of our global range delete method.

Modern key–value stores introduce a specialized entry, range tombstone, to represent the range records. For each deleted range, the range tombstone records the start key as the entry key, while storing the end key in its value. This design allows a compact representation of range records, which remains consistent across various range delete lengths. For instance, as the key size is k, a range tombstone only takes up $2k$ in size. To store these range tombstones, a dedicated block is allocated in each LSM level. Within these blocks, the range tombstones are sorted by their start keys to improve query efficiency. Since these tombstones are locally stored at individual levels, we refer to this approach as the local range record (LRR) method.

Range Delete Operation. The range delete can be performed by simply inserting a range tombstone into the range tombstone block of the memory buffer. These blocks would be consulted during flushes or compactions to remove obsolete entries. Similar to point deletions, range tombstones must be propagated to the bottommost level to ensure that all obsolete entries are eventually cleared. To achieve this, range tombstone blocks are also merged during flushes and compactions, with the results written into the target level. This process requires reading and rewriting the range tombstones, incurring an I/O overhead of $O(\frac{k}{B})$ per entity. As analyzed in Section 2, a range tombstone typically undergoes $O(T \cdot L)$ compactions before expiration, leading to a total range delete cost of $O(T \cdot L \cdot \frac{k}{B})$. This result highlights the notable efficiency of the LRR approach, as its cost is even lower than that of a single update operation.

The Impact on Point Lookups. Since range tombstone blocks are locally attached to each LSM-tree level, point lookups still need to search the tree from the memory buffer down to the bottommost level. Whereas, at each level, in addition to checking the data blocks for the target entry, the corresponding range tombstone block must also be probed to determine whether the key has been deleted. Then, the lookup can terminate if the key is found in the data blocks and not invalidated by any tombstones, or if it is determined to be deleted by a range tombstone. Therefore, apart from the lookup cost analyzed in Section 2, probing range tombstone blocks brings extra overhead, which is evaluated in detail in the following part.

Within a certain range tombstone block, a target key v can be covered by any range tombstone whose key is smaller than v due to the various range lengths. For instance, key 500 can be deleted by either range delete $[1, 1000)$ or $[490, 510)$. To estimate the number of such tombstones to check, let $\frac{N_i}{\lambda}$ denote the number of range tombstones in the i-th level, where N_i and $1/\lambda$ represent the number of data entries and the ratio of range deletes, respectively. The key of the range tombstones is uniformly distributed over the key universe $[0, U)$. In this case, the number of candidates q_i follows a binomial distribution with the expected value of $\frac{N_i}{2\lambda}$. In order to retrieve them, one I/O is required to load the first page of the tombstone block, followed by sequential reads since tombstones are sorted by start key. Thus, the probing cost is $O(1 + \frac{N_i}{\lambda} \cdot \frac{k}{B})$, together with φ I/Os

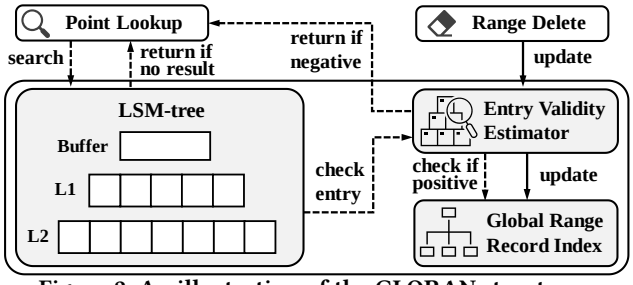

Figure 2: An illustration of the GLORAN structure.

to search the data blocks. If v does not exist or is invalidated by a tombstone, all L levels must be checked with cost presented in Equation 1. If a valid v exists, one more I/O is needed to retrieve it.

$$Z = O\left(\sum_{i=1}^{L}\left(\frac{N_i}{\lambda}\cdot\frac{k}{B}+\varphi+1\right)\right) = O\left(\frac{N}{\lambda}\cdot\frac{k}{B}+L\cdot\varphi+L\right) \quad (1)$$

Discussion. As indicated in Equation 1, the range tombstone blocks incur at least L additional I/Os during point lookups. This overhead is considerable compared with the cost of data block access, since the false positive rate of Bloom filter φ is usually much smaller than 1. The key reason is that range tombstone blocks are stored locally, hence requiring at least one I/O at every level to retrieve them. Consequently, the overhead persists even in workloads with only a few range deletes. In addition, the lack of an efficient index forces point lookups to examine $O(\frac{N}{\lambda})$ tombstones. Within a specific level, all range tombstones whose start key is smaller than the target are retrieved. Whereas many of them are irrelevant to the target key. For instance, when searching the key 100, the range tombstone $(5, 25)$ should be accessed, though it does not overlap with 100, which leads to unnecessary I/Os. These problems can be effectively mitigated by introducing an efficient global index for range records, which decouples the probing cost from the number of LSM levels. Furthermore, organizing records by their ranges allows the system to skip many irrelevant ones, thus achieving lower I/O cost. Based on these analyses, we propose a novel method that can significantly reduce point lookup cost while maintaining efficient range deletes.

4 GLORAN

We introduce a global range delete method, GLORAN, as illustrated in Figure 2. It incorporates a global index to organize the range records instead of storing them in LSM levels. Under this new configuration, the conventional range tombstone is no longer suitable. To tackle this, we design a novel representation approach to properly profile the range records (Section 4.1). Besides, the associated index structure, LSM-DRtree, is proposed to manage them efficiently. This structure delivers impressive query performance thus providing strong efficiency guarantees for point lookups while preserving desirable range delete performance (Section 4.2). Moreover, GLORAN employs a lightweight entry validity estimator that offers a shortcut in point lookup workflow, which further reduces expected lookup overhead (Section 4.3).

4.1 Effective Area of Range Records

The global range delete method necessitates effective approach to represent the range records for efficient management. This, however, is nontrivial. Due to the out-of-place deletion strategy of LSM-tree, simply recording the key range of a range delete is insufficient because each record implicitly carries temporal information. For

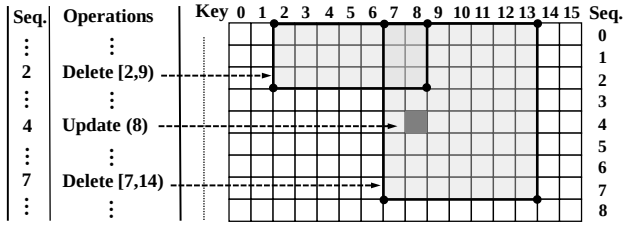

Figure 3: The rectangular effective area of a range delete within working space invalidates all entries within it.

example, suppose a range delete inserts a range $[5, 15)$ in the global index, followed by an update on key 8. Then, the global index would still mark this key as invalid, thus leading to incorrect point lookup results and mistaken removal of valid entries during compaction. Consequently, an effective representation of range records must capture both the key range and the temporal information. This can be naturally achieved when range records are stored locally in each LSM level within range tombstone blocks. As a special entry type, range tombstones are automatically assigned sequence numbers and managed by the multi-version control mechanism of the LSM-tree. However, in the global index, range records are stored outside the LSM-tree, making this mechanism unachievable. Therefore, a novel representation is required to explicitly encode both the key range and the sequence number range of each record.

To this end, we introduce a two-dimensional working space spanning the key domain and the sequence number domain. As shown in Figure 3, a range delete on keys $[7, 14)$ is issued with sequence number 8. This operation invalidates all entries whose keys fall within the specified range and sequence numbers are smaller than 8 in the LSM-tree. These conditions naturally enclose a rectangle in the working space, which we define as its **effective area**. Then, any entry mapped inside this area should be obsoleted by this operation, such as the entry with key 8 and sequence number 5 in Figure 3. Based on this abstraction, the validity of entries can be determined according to Lemma 4.1.

Lemma 4.1. *For a sequence of range deletes with effective areas $\mathscr{A} = \{\alpha_1, \alpha_2, \cdots, \alpha_n\}$, any entry invalidated by a range delete must be covered by a $\alpha_i \in \mathscr{A}$ in the working space.*

Accordingly, the global index represents each range record by its effective area, a rectangle defined by the two vertices. For example, in Figure 3, a range delete on keys $[7, 14)$ with sequence number 8 is represented by two vertices: $(7, 0)$, denoting the smallest key and sequence number, and $(14, 8)$, their largest counterparts. Note that the smallest sequence of a range record specifies the minimum boundary before which the record expires, which differs across range records. Figure 3 shows it as zero for simplicity. In general, the effective area can precisely capture the boundaries of each range record to avoid unnecessary lookups for irrelevant entries. Observant readers may notice that an appropriate index over these rectangles is also crucial for the system performance. We thus next introduce the design of our global index structure.

4.2 Global Range Record Index: LSM-DRtree

The global index is utilized to store range records and determine whether a key has been invalidated by them, which is key for achieving desirable range deletes and point lookups. Considering the rectangular effective areas of the range records, the index must

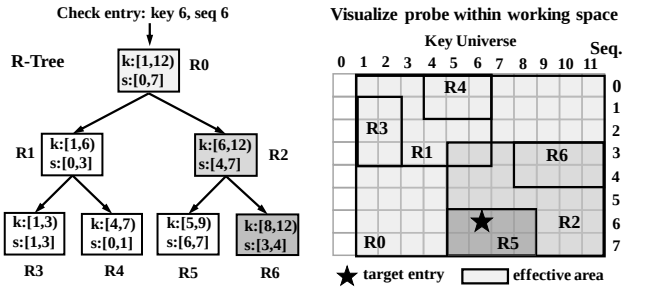

Figure 4: The R-tree structure is able to locate an entry efficiently within the working space during point lookups.

efficiently store a set of rectangles and determine whether a given point is covered by any of them.

In this context, the performant spatial R-tree serves as a natural and attractive choice. As illustrated in Figure 4, an R-tree organizes range records in a hierarchical manner: leaf nodes (R3–R6) store the actual range records, while internal nodes (R1, R2) maintain the minimum bounding rectangles (MBRs) that enclose all their child nodes. Higher-level internal nodes recursively index lower ones until a single root node remains. To check whether an entry is covered by any range record, the R-tree is probed from the root downward. For instance, in Figure 4, checking an entry with key 6 and sequence number 6 sequentially visits nodes R0, R2, and R6. The effective areas of these nodes are also presented for clarity. During this descent, the search region narrows progressively, thus effectively skipping irrelevant elements. When inserting a new range record, the R-tree attaches it to the internal node whose MBR incurs the smallest expansion, which naturally clusters related records and minimizes MBR overlap, thus improving query efficiency.

Despite its practical effectiveness, the R-tree suffers from theoretical inefficiency due to its lack of sound worst-case query complexity guarantees. When range records are highly skewed in the key universe, their effective areas become densely clustered, resulting in significant MBR overlap among internal nodes. Consequently, certain queries may need to access substantial, even nearly all, internal nodes in each level with excessive overhead. For example, if the queried entry in Figure 4 has a sequence number of 3, R1 cannot be skipped even though none of its range records actually cover the entry. This property results in undesirable tail latency and unstable performance. Moreover, the R-tree incurs heavy update overheads. As the number of range records grows, a node may exceed its capacity, which necessitates a split and adjustments across the parent nodes. These operations are complex and costly. In particular, the global index must be serialized on disk to accommodate large volumes of range records. Hence, direct updates to on-disk R-tree nodes lead to substantial I/O overhead that significantly degrades range delete performance of the system.

To address these issues, LSM-DRtree is proposed to enhance the theoretical query complexity of R-tree using the disjoint property of effective areas, as well as improve update performance with an LSM-style structure, as detailed in the following.

Effective Areas Disjointization As introduced above, the overlaps among effective areas are the main reason for the unsatisfying query performance of R-tree. Therefore, eliminating overlaps among them is effective to enhance the query performance guarantee. However, this is infeasible for conventional R-trees since

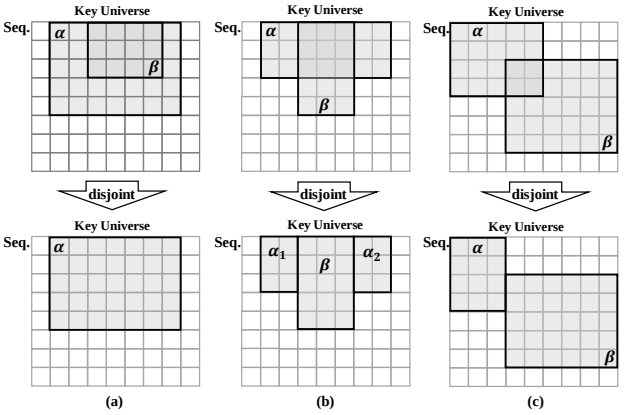

Figure 5: Disjointization between two effective areas.

the stored rectangles are unchangeable. In contrast, the disjoint property of effective areas makes it achievable in our context.

In LSM-trees, two effective areas overlap when their associated range records invalidate the same key range. Since the latter range record carries more recent information, its efficacy dominates the earlier one within the overlapped key range. Thus, we can reorganize them to preserve only the most recent range record in each key interval to eliminate overlaps. This process we call *disjointization*.

Figure 5 illustrates the three possible cases between two overlapping effective areas, denoted as α and β. (a) When both the key and sequence ranges of β are fully contained in those of α, β is completely dominated and can be directly replaced by α. (b) When β's key range is contained in α but its sequence range only partially overlaps, β updates part of α's key range while the remaining region is still dominated by α. In this case, α is split by the key range of β in the working space. Notably, when both start and end keys differ for α and β, a new effective area would be introduced after the disjointization. (c) When both the key and sequence ranges partially overlap, as the figure presents, the overlapped region is dominated by β due to its higher sequence number. Hence, α is trimmed to disjoint β in the key domain. A potential concern is that trimming α might expose obsolete entries not covered by β. This does not occur because an effective area inherently defines the key and sequence ranges where its range delete is valid. When such an area is created, no matched entries exist in the LSM-tree that were issued before its starting sequence number. Therefore, trimming this area does not compromise the correctness of the resulting effective areas.

Lemma 4.2. *For a set of effective areas $\mathscr{A} = \{\alpha_1, \alpha_2, \cdots, \alpha_n\}$, disjointization generates a new set $\mathscr{B} = \{\beta_1, \beta_2, \cdots, \beta_m\}$, where an entry can be covered by at most one $\beta_i \in \mathscr{B}$ in the working space.*

The above discussion examines the overlap scenarios between two effective areas. When multiple effective areas overlap, their disjointization can always be decomposed into these fundamental cases as stated in Lemma 4.2. Though new elements would be generated after disjointization, the total number is no more than twice of the original set. For instance, Figure 6 illustrates disjointization with six effective areas that form a sequence of disjoint effective areas sorted by key order. These disjoint areas can then be recursively merged to construct higher-level parent nodes, leading to a tree structure termed the **Disjoint R-tree (DR-tree)**.

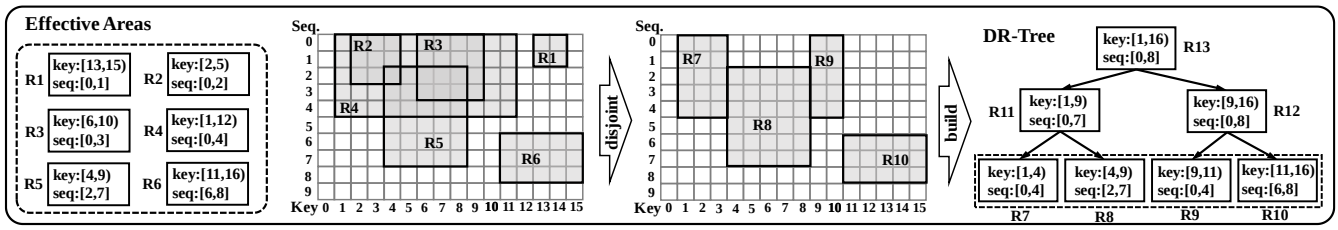

Figure 6: An illustration of DR-tree construction from a set of effective areas.

Remark. Owing to the disjoint property, each key is covered by at most one leaf node (Lemma 4.2), and consequently, only a single internal node needs to be accessed per level during query processing. This property provides a strong theoretical bound on the worst-case query complexity and substantially improves lookup efficiency compared with traditional R-trees.

Build DR-tree from Effective Areas. Obviously, deriving disjoint effective areas and ordering them by key is essential for DR-tree construction. While pairwise disjointization across multiple overlaps is computationally expensive, we provide a more efficient approach.

As shown in Figure 6, the disjointization results occupy separate key intervals and are outlined by the most recent effective area covering that range. Hence, we can scan the effective areas along the key universe to determine the boundaries of these key intervals and record their dominant effective area. To this end, we use two min-heaps to store the start and end keys of all effective areas and scan them by repeatedly extracting their smallest key. Meanwhile, a max-heap, *curr*, is employed to maintain currently active areas, whose start keys have been processed but end keys remain pending. These effective areas are ordered by sequence numbers within this max-heap, ensuring that its top always tracks the dominant effective area for the preceding key interval.

A new interval begins during scanning whenever a start key has a larger sequence number than the current dominant area, as illustrated by R4, R5, and R6 in the figure. As a more recent element arrives, the top of *curr* is replaced and temporarily terminates its efficacy at this point, which generates a new record in the result, as R7 does. It is worth mentioning that the replaced effective area is still kept in *curr* since the end key has not been reached. Hence, it is possible to become the dominant effective area again, like R9. Then, when the end key of the current dominant area is encountered, it indicates the termination of this area that should be recorded with the preceding key interval. Besides, if *curr* remains non-empty, the endpoint simultaneously marks the beginning of the next key interval (e.g., R9). By iterating through all start and end keys in this manner, we obtain a complete set of disjoint effective areas naturally ordered by key.

Whereas updating an RD-tree on disk is still costly since the inserted effective areas would change the layout of leaf nodes. To address this issue, we further design **LSM-DRtree** that maintains the outstanding query efficiency of the RD-tree while achieving more favorable update performance.

LSM-DRtree Structure. The LSM-DRtree introduces an LSM-style design for the DR-tree to significantly enhance its update performance. A similar idea was also explored in [51] that introduces LSM-Rtree to enhance R-tree's update efficiency. However, our approach targets a fundamentally different structure, DR-tree, which requires distinct configurations, flushing method, and compaction mechanisms. LSM-DRtree maintains an in-memory R-tree as the write buffer and organizes multiple on-disk levels, each storing a DR-tree. Similar to the LSM-tree, the capacity of each level grows by a fixed size ratio. Under this design, once a DR-tree is written to disk, it remains immutable until the next compaction, thus effectively avoiding the costly in-place updates required when directly storing an entire RD-tree on disk. Moreover, although updates to the write buffer are performed in memory and do not incur extra I/O cost, frequently updating a DR-tree would still impact the system performance. To mitigate this, we employ an R-tree as the write buffer to further accelerate updates. We next detail the flush and compaction workflows for the index construction.

Construction of LSM-DRtree. Upon receiving a range record, its effective area is inserted into the in-memory R-tree. When the R-tree reaches its maximum capacity, a flush process is triggered. During this process, all effective areas stored in the R-tree's leaf nodes are extracted to construct a DR-tree as introduced earlier. The resulting DR-tree is serialized level by level, ensuring that nodes within the same level are physically clustered and sorted by key range. Since flushes occur less frequently and the write buffer is limited in size, this process incurs moderate overhead.

When a disk level becomes full, a compaction process is triggered to merge its DR-tree with that of the next level. Benefiting from the fact that effective areas in each DR-tree are already disjoint and sorted by key order, the merging can be performed entirely through sequential I/O, avoiding random disk access. The compaction proceeds as a simple two-way merge: an iterator is built for each DR-tree, and at each step, the system retrieves the effective areas with the smallest start keys from both iterators, performs disjointization between them if they overlap, and proceeds iteratively until all effective areas have been processed.

Remark. The compaction workflow of LSM-DRtree only involves pairwise disjointization between two effective areas at a time, avoiding the more intricate process for building DR-tree from arbitrary effective areas. Moreover, its streaming nature enables on-the-fly execution without introducing additional I/O overhead, thereby ensuring high update efficiency. Compared with LSM-Rtree, LSM-DRtree offers a much simpler merging process that avoids complex spatial alignment among rectangles, which achieves approximately 11% faster construction latency in our experiments. We next describe how LSM-DRtree supports range deletes and point lookups in LSM-based key-value stores and quantitatively analyze the operational overhead.

Range Delete with Global Index. Each range delete inserts a range record into the LSM-DRtree. The record is represented as a rectangular effective area that is stored by two ends of the key range and two sequence numbers. Meanwhile, the size of the sequence number is typically much smaller than the keys. Hence, the size of each range record can be formulated as $2k$. With the LSM structure,

the range record would be rewritten during flush or compaction processes, which would incur subsequent overhead.

Lemma 4.3. *Updating a global index with $\frac{N}{\lambda}$ range records leads to $O(\frac{k}{B} \cdot T \cdot \log_T(\frac{N}{\lambda F}))$ amortized I/O cost.*

Proof Sketch. The $\frac{N}{\lambda}$ range records produce at most $\frac{2N}{\lambda}$ effective areas after disjointization. To store them, the LSM-DRtree should introduce $L' = O(\log_T(\frac{N}{\lambda F'}))$ levels, where F' and T are the write buffer size and size ratio, respectively. Though the write buffer of the global range record index is typically smaller than that of the LSM-tree, $F' = O(F)$ still holds. Within an LSM-DRtree, the effective areas can eventually be stored in the bottommost level, after experiencing $O(TL')$ times compactions. Therefore, the overall construction I/O cost is $O(\frac{2N}{\lambda} \cdot \frac{T \cdot 2k}{B} \cdot \log_T(\frac{N}{\lambda F}))$, leading to the range delete cost for each operation as $O(\frac{k}{B} \cdot T \cdot \log_T(\frac{N}{\lambda F}))$ I/Os. □

Remark. By the above lemma, the update cost is lower than that incurred by the local range delete method, particularly for typical cases where λ is large.

Point Lookup with Global Index. The point lookup begins with searching the target key in an LSM-tree. If no matching key is found, the query terminates immediately without accessing the LSM-Rtree. As previously discussed, this leads to the I/O cost of $\lceil \varphi \log \frac{N}{F} \rceil$. Otherwise, if a matching entry is found, the global index should be checked to verify whether it has been invalidated by subsequent range deletions.

Lemma 4.4. *After inserting $\frac{N}{\lambda}$ range records, global index checks validity of an entry with $O(\log_T^2(\frac{N}{\lambda F}))$ I/O cost in the worst case.*

Proof. Following the design philosophy of LSM structures, checking the global index proceeds the LSM-DRtree level by level. In order to figure out the overall query cost, we first analyze the associated costs at the i-th level that contains Q_i range records. As discussed, these range records result in up to $2Q_i$ effective areas after disjoinitzation that are stored in the leaf nodes of a DR-tree. Following Lemma 4.2, only one leaf node may cover the target key, which requires one I/O to retrieve. Moreover, storing these effective areas requires a DR-tree with $O(\log_D(Q_i))$ levels, and one node would be accessed in each level during the searching process due to the property of DR-tree. Moreover, all L' levels of the LSM-DRtree would be searched. Note that $Q_i = F' \cdot (L')^i$, we immediately have the cumulative cost:

$$Z_0 = \sum_{i}^{L'} \left(\log_D Q_i + 1\right) = O\left(\frac{L' \cdot (L'+1)}{2} \cdot \log_D T + L' \cdot \log_D F'\right) \quad (2)$$

Also, we have $D \geq T$ and $F' = O(F)$. The query cost can then be simplified to $O((L')^2) = O(\log_T^2(\frac{N}{\lambda F}))$. □

Remark. After issuing N/λ range deletes, the local range delete method incurs I/Os linear to N/λ for point lookups. The above lemma shows that GLORAN significantly improves this cost to poly-logarithmic scale regarding N/λ.

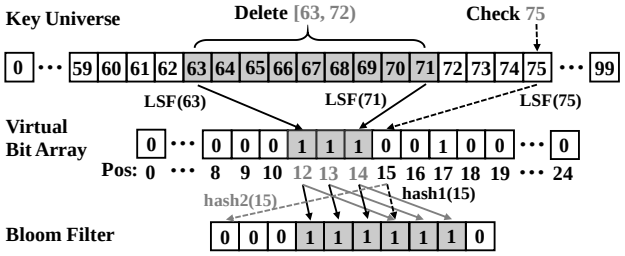

Figure 7: The range-aware estimator (RAE) uses the virtual bit array to encode key ranges in a Bloom filter.

4.3 Predictive Shortcut for Point Lookup

The global range record index can efficiently determine the validity of entries during point lookup. However, it is noticed that, according to the introduced point lookup workflow, all entries would be checked, including the valid ones that are not included in the global index. This observation motivates a predictive shortcut: if valid entries can be quickly identified before entering the global index with a certain probability, the results can be directly returned without incurring global index related overhead, thereby, further reducing the amortized point lookup cost.

This requires a dedicated component to store range record information and determine the coverage of individual keys without false negative estimation, ensuring that obsolete entries are not mistakenly returned during point lookups. Furthermore, this component is expected to be efficient and lightweight to avoid excessive overhead that otherwise degrade system performance, as formally defined in Problem 1. In addition, since range deletes are continuously applied to the database, this structure must deliver desirable dynamic performance. Hence, we propose **entry validity estimator (EVE)**, a practical design that well supports above requirements with its efficacy evaluated in our experiments.

Problem 1. *Given a set S of n key ranges within an integer universe $U = \{0, \ldots, U-1\}$, build a space-efficient in-memory data structure that answers emptiness queries of the form $q \cap S \neq \emptyset$? for any q in U. The structure is allowed to return "not empty" when actually $q \cap S = \emptyset$ (i.e. false-positive error) with certain probability.*

A naive solution is to employ a Bloom filter to store every key within a range record. However, each range record corresponds to a large number of individual keys and inserting all of them into the Bloom filter leads to a significantly degraded false positive rate (FPR). As a result, many valid entries are likely to be incorrectly identified as obsolete, thus still requiring to consult global index. Moreover, processing every key within a range record further incurs significant computational costs, making this approach impractical.

Therefore, it is crucial to leverage range information inherent in range deletes. For this purpose, we propose a **range-aware estimator (RAE)** that integrates a range encoding strategy with the Bloom filter, as presented in Figure 7. For RAE, a linear scaling function is employed to map the key universe into a bit array, where each bit represents a key range rather than an individual key. Consequently, a deleted key range can be represented by only a few bits. Then, the positions of these bits are recorded in the Bloom filter. In this process, the bit array used for range encoding is only a conceptual aid to derive bit positions that does not need

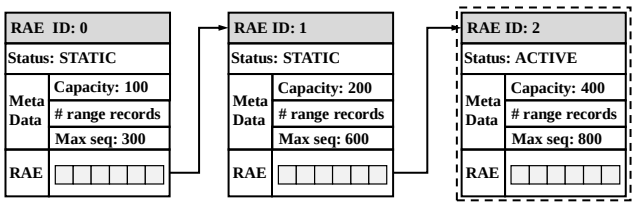

Figure 8: The structure of the entry validity estimator (EVE).

to be physically stored. Hence, we refer to it as a *virtual bit array*. This approach substantially reduces the number of insertions and greatly improves both estimation accuracy and efficiency.

In Figure 7, when a range delete is issued, the two boundaries of its key range are mapped through the linear scaling function to identify the occupied virtual bit segment [12,14]. The positions within this segment are then inserted into the Bloom filter. For checking key 75 during point lookups, it is similarly mapped onto the virtual bit array to check the position in the Bloom filter. Since the Bloom filter returns a negative result, this key is definitely not covered by any range delete, and thus the corresponding entry can be directly returned without probing the global range record index.

Meanwhile, with more range records inserted, the estimation accuracy of a single RAE can gradually degrades, and eventually becoming ineffective. This necessitates an extendable structure that can scale with the growing number of range deletes to support the application of key-value stores. Therefore, we adopt a chained structure for the RAE, as illustrated in Figure 8, which completes the design of our entry validity estimator. To construct a EVE, an empty RAE is initialized before workload execution. This RAE accepts incoming range deletes and maintains the smallest and largest sequence numbers among them. When the current RAE reaches its capacity, a new active estimator with doubled capacity is created, while the previous one is marked as static. During a point lookup, once the target entry is found in the LSM-tree, the EVE is queried. The active RAE is checked first. If it returns a positive result, the entry may have been deleted by a range delete and must be verified against the global range record index. Otherwise, the query proceeds recursively to the most recent RAE until either a positive result is obtained or the entry's sequence number exceeds that of the RAE. If all subfilters return negative, the entry is guaranteed to be valid and can be returned without further cost.

Discussion. Therefore, if the target entry is valid, EVE can reduce the point lookup cost from $O(\log_T^2(N/\lambda F))$ to $O(\varepsilon \cdot (\log_T^2(N/\lambda F)))$, where ε is the false positive rate of EVE. We acknowledge that similar functionality is also achievable for certain dynamic range filters. However, these filters are typically designed to store individual keys hence struggle to offer sound performance in this case, particularly under limited memory budgets. Moreover, their efficacy is confined to the key space, whereas EVE can leverage sequence number with its chained design. Nevertheless, we compare EVE with several competitive range filters in Section 6, which achieves the highest accuracy with over 20% lower false positive rate.

4.4 Space Amplification of GLORAN

The space amplification caused by obsolete entries under GLORAN is comparable to that of other methods, since these entries can be properly reclaimed during compaction. In addition, range deletes introduce extra space overhead for storing deletion metadata. For this part, GLORAN exhibits satisfying space efficiency, as each range record occupies a compact size, unlike the approaches that insert a tombstone for every key. We further compare the size of our global index with the range tombstone blocks used in LRR, another range record method. At the i-th level of the LSM-DRtree, which contains Q_i range records, a DR-tree maintains at most $2Q_i$ leaf nodes after effective area disjointization. The number of nodes in higher levels decreases by a factor of D from bottom to top. Denoting the total number of levels as L_i, the total number of nodes R_i in this DR-tree can be expressed as:

$$R_i = \frac{D}{D-1} \cdot \frac{D^{L_i}-1}{D^{L_i}} \cdot 2Q_i < \frac{D}{D-1} \cdot 2Q_i \tag{3}$$

Aggregating the DR-trees across all levels of the LSM-DRtree, the overall size of the global range record index is bounded by $O(\frac{D}{D-1} \cdot Q \cdot k)$. Since D is the DR-tree fanout (an integer greater than 2), $\frac{D}{D-1}$ is smaller than 2. Therefore, the space overhead of GLORAN is $O(Q \cdot k)$, which is asymptotically the same as that of LRR. This result indicates that the space amplification of GLORAN remains modest and well bounded.

In addition, GLORAN incorporates a garbage collection (GC) strategy to clear obsolete range records from the global index to ensure space overhead and query efficiency. A range record becomes obsolete in two cases. The first occurs when its effective area is fully covered by a subsequent record, which is automatically handled by the LSM-DRtree compaction through effective area disjointization. The second arises when all entries it matches have been removed from the LSM-tree. To identify such cases, we attach an event listener to detect compactions on the bottommost level and record the largest sequence number of the resulting data. After such compaction, GC is triggered to purge elements whose sequence numbers are below this threshold. Since outdated records are typically concentrated at the bottommost level of LSM-DRtree, GC is confined to this level to reduce overhead. Moreover, the same threshold is also used to drop outdated RAEs in the entry validity estimator, for further improving its accuracy and query efficiency.

5 RELATED WORK

Key-value Stores Enhancement. LSM-based key–value stores are applicable to a wide range of use cases across diverse application domains [8, 9, 42, 47, 57, 62]. Research on LSM-based key–value stores mostly targets enhancing the system performance, including overall throughput [38] and space efficiency [39] via improving different operations like point lookups, range queries, and updates. Point lookup improvements refine or replace Bloom filters [12, 15, 32, 67, 68], adapt data distribution [65, 66], and exploit spatial locality [17, 55]. Range queries benefit from succinct tries, hierarchical or hybrid range filters, and learned or adversarial-resistant models [7, 10, 29, 37, 44, 54, 58, 64]. Update throughput boosts via compaction scheme variants [2, 25, 41, 46, 48, 53, 59, 61], key–value separation [6, 35], learned indexes [11], and parallelism [22, 63]. While memory allocation tuning [28, 36] and finer-grained compaction cut space amplification [3, 16, 40].

Range Deletion Optimization. There are several prior works [5, 21, 33] researching on the bulk deletion in relational databases. However, this typically follows the direction for read enhancement, which pursues quick location and entry clustering through sorting or hashing, which is different from the range deletion in LSM-trees.

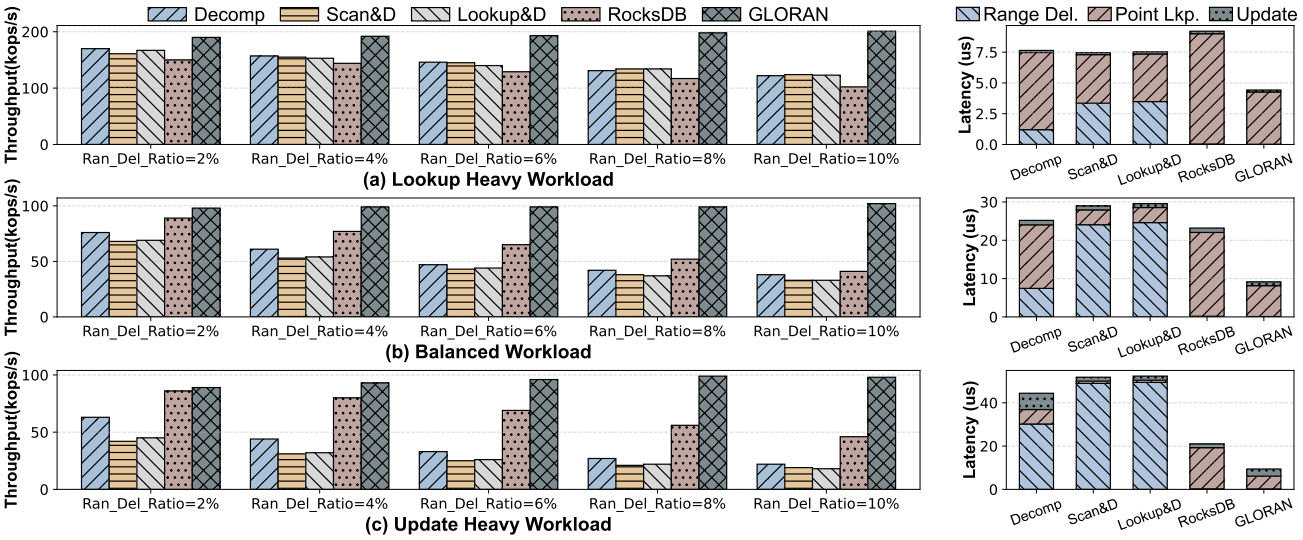

Figure 9: We evaluate various range delete methods across diverse workloads. The results show that GLORAN consistently delivers the best performance across all test cases while enabling fast range deletes and efficient point lookups.

Nie *et al.* [45] tailored deletion for specified device like ZNS SSD. Sarkar *et al.* [50] proposed Lethe to reduce the lifetime of tombstones with FADE, a novel compaction strategy. Whereas, these works mainly focus on the optimization for point deletion, which requires different techniques compared to our target operation range delete. Moreover, Lethe also discussed range deletes on a secondary delete key, where the range deletion is not applied to the primary key in the LSM-tree. This requires optimization concerning the data layout which falls within another field.

6 EVALUATION

This section presents evaluation results of different methods across diverse test cases. All experiments are conducted on a server equipped with a 13th Gen Intel(R) Core i9-13900K CPU @ 5.80GHz, 128GB DDR4 RAM, and a 2TB NVMe SSD, running 64-bit Ubuntu 22.04.5 LTS on an ext4 file system.

Implementation. We implement GLORAN on top of RocksDB [18], a widely adopted key-value store used in many prior studies [13, 14, 23, 34, 50, 56]. Our implementation offers various tuning factors to support flexible customization of GLORAN structure, such as the memory buffer and size ratio of the global LSM-DRtree, as well as the fanout of DRtrees. For the entry validity estimator, the capacity and memory of each RAE are also adjustable.

Baselines and Implementation. We include all existing range delete methods for LSM-based key-value stores and build them on RocksDB [18]. The decomposition method (abbr. *Decomp*) deletes all keys within the target range individually using the *Delete* API. The lookup-and-delete (abbr. Lookup&D) performs a point lookup for each key with *Get* API and only *Delete* the existing ones. The scan-and-delete method (abbr. Scan&D) substitutes the point lookups with a range scan employing an *iterator* to identify the existing keys for deletion. The local range record method (abbr. RocksDB) is natively supported in RocksDB, so we use the built-in *DeleteRange* API directly. Moreover, Lethe [50] also discusses deletions in LSM-based key-value stores. Whereas it focuses on a fundamentally different operation, secondary range delete. Nevertheless, we also evaluate it in a case study based on the code provided by the authors [49].

Experimental Setting. In our experiments, we retain most parameters as RocksDB's default settings, which are developer-optimized for general SSD-based applications [20]. In this setting, the memory buffer is configured to 64 MB. To enrich the evaluation, we also vary this value from 8 MB to 256 MB (Figure 12 (a)). Besides, the Bloom filter is assigned 10 bits per entry. For the global range record index in GLORAN, we explore different memory buffer sizes (Figure 11 (b)) and size ratios (Figure 11 (c)), with defaults of 4 MB and 10, respectively. For EVE, the first RAE is configured to hold 0.8 million range records with 10 bits per record, which we vary in Figure 13 (c). We evaluate these methods under workloads with different ratios of updates and point lookups. To access the impact of range deletes, we replace varying percentages of updates with range deletes (Figure 9) and adjust their lengths to measure the impact on throughput, space amplification, and memory usage (Figure 10). Each test runs 100 million randomly generated operations on an empty database, with direct I/O and block cache enabled following prior work [14, 24, 34, 41, 50, 56]. Each entry consists of a 256-byte key and an 768-byte value, which are varied in Figure 11 (a) and (b), respectively. To ensure comprehensive evaluation, we also assess the influence of diverse block cache configurations (Figure 11 (d)), data scales (Figure 11 (c)), the efficacy of LSM-DR tree structure and EVE (13), as well as the impact of range lookups (Table 3). In addition, we further include two practical benchmarks that are widely adopted in previous works [23, 41, 43], RocksDB micro benchmark, db_bench [19] (Table 5), and YCSB [60] (Table 4) with different key distributions to assess more aspects of these range delete methods.

Overall Comparison. We evaluate different range delete methods under three representative workloads: lookup heavy (90% point lookups 10% updates), balanced (50% point lookups 50% updates), and update heavy (10% point lookups 90% updates). To assess these methods with different amounts of range deletes, we vary the range delete ratio from 0% to 10%. As Figure 9 presented, GLORAN consistently delivers the highest throughput across diverse workloads. Notably, in the balanced workload, it outperforms the second-best

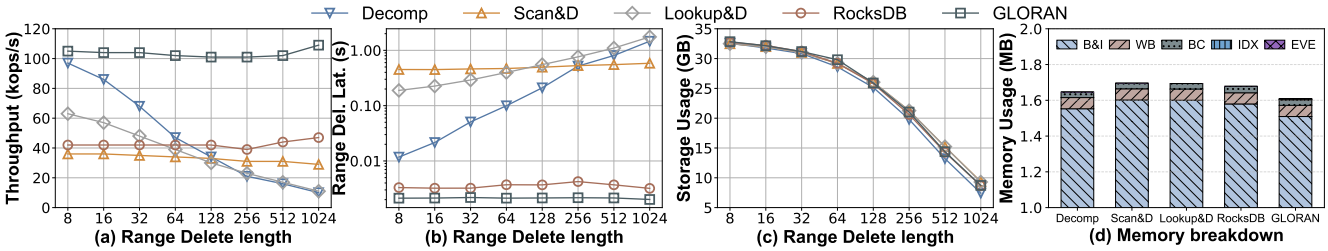

Figure 10: The performance of different range delete methdos under various range delete lengths.

method, *RocksDB*, by up to 2.4×. In this workload, both lookup and range delete efficiency critically impact overall performance. This demonstrates GLORAN's ability to support efficient range deletes without sacrificing point lookup performance. By contrast, point-based methods incur significant overhead from range deletes, while *RocksDB* suffers slower point lookups. Even in workloads, where baselines may partially mask their weaknesses, GLORAN maintains superior performance, achieving up to 1.6× higher throughput than *Scan&D* in lookup heavy workloads and 2.1× higher than *RocksDB* in update heavy workloads.

We further observe that baseline performance degrades noticeably as the range delete ratio increases. In the balanced workload, *RocksDB*'s throughput drops by over 30% when the range delete ratio increases from 2% to 10%, as more range tombstones are being checked during point lookups. Other methods also suffer from increased range deletes due to more tombstone insertions. In contrast, the efficient global index and effective EVE bring higher robustness for GLORAN towards varying range delete numbers.

These experiments indicate the outstanding range delete and point lookup performance of GLORAN at a high level. We next provide a finer performance breakdown to validate this conclusion.

Exp 1: Point Lookup and Range Deletes. We profile the performance of each method when the range delete ratio is 10% and decompose the latency into point lookup, update, and range delete counterparts, as depicted in Figure 9. Notably, the methods using point deletion suffer from significant range delete overhead due to the large number of tombstone insertions or costly additional validation processes, particularly in update heavy workloads. Moreover, these tombstones could also introduce higher update overheads as illustrated in Figures 9 (b). *RocksDB* exhibits much better range delete performance, while it still spends substantial time on lookups which is 3.4 times slower than GLORAN. *In contrast, GLORAN achieves outstanding range delete latency and competitive lookup performance simultaneously.*

Exp 2: Performance v.s. Range Length. We vary the range delete length to evaluate the overall throughput, range delete latency, space amplification, and memory usage of different range delete methods under balanced workload. Figures 10 (a) and (b) demonstrates that GLORAN consistently delivers the most satisfying overall throughput and range delete latency across various cases, particularly for long ranges. Additionally, the point deletion based methods show explicit degradation as the range length grows, especially for *Decomp* that necessitates obvious more tombstone insertions with higher overhead. In contrast, *RocksDB* and GLORAN exhibit more robust performance against the increase in range length due to their range records based strategies. Whereas, GLORAN delivers better overall throughput than *RocksDB* by up to 2.7× when the range delete length is 1024. *Hence, GLORAN maintains desirable and robust performance across varying range delete lengths.*

Different range delete lengths lead to varied disk size which is reported in Figure 10 (c). Generally, longer deletes remove more entries from the LSM-tree, resulting in reduced data volume. This applies to all methods. Meanwhile, their disk usage is comparable because all of them can effectively reclaim space occupied by obsolete entries via compaction. Although *RocksDB* and GLORAN maintain extra structures for range records, the associated space overhead is slight due to their compact size. *This indicates GLORAN achieves sound space amplification and memory efficiency.*

In addition, Figure 10 presents the memory breakdown with delete length of 128. Clearly, Bloom filters and indexes (B&I) dominate memory usage across all methods. While LSM-tree buffer (WB) and block cache (BC) occupy smaller and fixed amounts for each method. In comparison, GLORAN uses extra memory for global index write buffer (IDX) and entry validity estimator (EVE). Whereas their sizes are minimal compared to other parts.

Exp 3: Performance v.s. Entry Sizes. We vary the key size and value size under balanced workload and show the results in Figure 11 (a) and (b), respectively. In Figure 11 (a), the key size is changed with entry size fixed at 1024 bytes. With larger key sizes is applied, the throughput of *Decomp* decreases since larger tombstones are inserted. The performance of *RocksDB* also drops for the more costly point lookups resulted by enlarged range tombstones. By contrast, the LSM-DRtree and EVE facilitate GLORAN with efficient point lookups hence achieves stable performance that is up to 2.2× improvement over *RocksDB*. Figure 11 (b) varies value size with key size fixed to 64 bytes. For all the methods, their throughput decreases as value size increases due to the increased compaction frequency and the associated overhead. Moreover, the enlarged value incurs deeper LSM-tree that requires higher point lookup cost. This eventually dominates the overall overhead and progressively mitigates the performance difference among diverse methods. Nevertheless, GLORAN still maintains 20% higher throughput than the best baseline at 2048 bytes. Interestingly, the performance of *RocksDB* is noticeably lower than others with small value size. In such cases, the point lookups cost on LSM-tree is relatively low, thus amplifying the impact incurred by checking range tombstones. As the results show, GLORAN achieves 10.6× faster point lookups than *RocksDB* when value size is 256 bytes. This also explains why *RocksDB* is outperformed by other baselines, though they delivers considerably less overall throughput than GLORAN.

Exp 4: Performance v.s. Data Scales. We incorporate more operations in the balanced workloads to test the range delete methods under diverse data scales, which varies from 2^{27} bytes to 2^{37} bytes

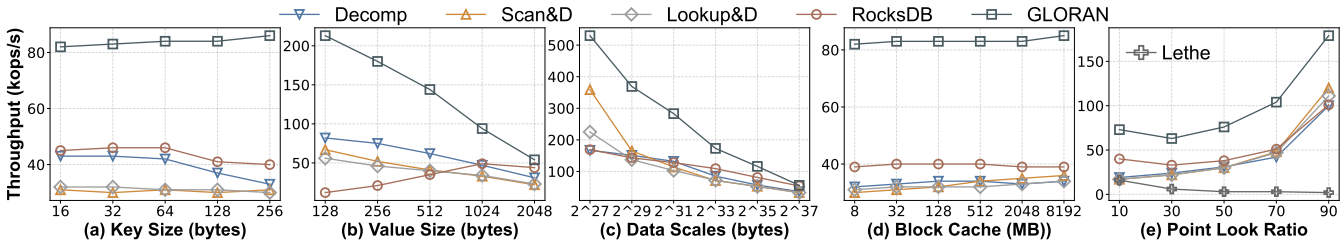

Figure 11: Throughput across varying key sizes (a), value sizes (b), data scales (c), block cache sizes (d), and with Lethe (e).

Table 3: Normalize throughput with range lookup.

	Range Lookup Ratio (%)				
	2	4	6	8	10
Decomp	1.00×	1.00×	1.00×	1.00×	1.00×
Scan&D	1.02×	1.11×	1.03×	1.04×	1.01×
Look&D	1.05×	1.13×	1.00×	1.06×	1.00×
RocksDB	1.35×	1.37×	1.23×	1.23×	1.17×
GLORAN	2.20×	2.04×	1.72×	1.62×	1.45×

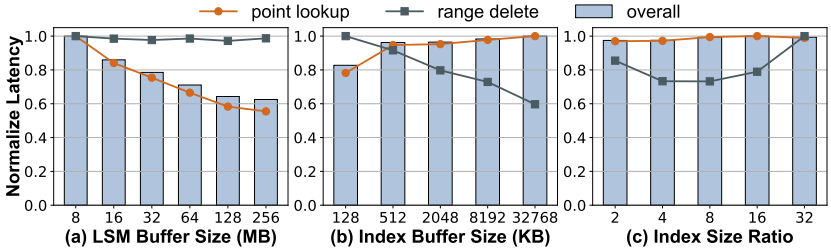

Figure 12: The impact of range lookups (Table 3), LSM buffer size (a), as well as global index buffer size (b) and size ratio (c).

in Figure 11 (c). With increased data volume, the LSM-tree develops more levels that brings higher update and lookup costs. Hence generally impact the throughput of all methods. Meanwhile, this incurs more overhead for tomsbtons insertion that impact the point deleteion based methods. The increased operation number also issues more range records. Hence the point lookup performance of *RocksDB* is drastically affected according to Table 2. While GLORAN presents obviously improved point lookup complexity that makes the deterioration resulted by scalded data volume more moderate. As a result, though the increases data scale poses challenge for GLORAN, it still presents impressive performance and adaptability.

Exp 5: Performance v.s. LSM Parameters. We vary the block cache size from 8 MB to 8192 MB to show the normalized throughput of different methods in Figure 11(d). The results indicate that enlarging the block cache typically cannot obviously enhance the performance since it does not impact the workflow of range delete relevant operations. Although *Lookup&D* and *Scan&D* gain minor improvements from faster lookups, the benefits are limited, and GLORAN still achieves superior performance. Figure 12(a) reports GLORAN's performance under different LSM memory buffer sizes. For clarity, we present both normalized overall latency and the breakdown for point lookups and range deletes. In this figure, larger buffers mainly improve point lookup efficiency by reducing the number of LSM levels, thus lowering overall latency. Meanwhile, the overheads of other operations remain relatively stable.

Exp 6: Performance v.s. Global Index Parameters. We evaluate various configurations of the global range record index by tuning the LSM-DRtree parameters. As shown in Figures 12(b), larger memory buffers reduce range delete overhead by lowering the LSM-DRtree depth and compaction frequency. However, they also increase the DR-tree depth at each level, resulting in higher point lookup costs. This contrasts with the LSM-tree in Figures 12(a), where sorted entries and fence pointers make lookup cost less sensitive to level size. Similarly, the range deletes also benefit from an increased size ratio in Figure 12(c). While the performance degrades when the ratio exceeds 8 due to the high cost of merging enlarged levels. These findings highlight the need to carefully tune LSM-DRtree parameters for different workloads and suggest opportunities for further optimization.

Exp 7: Range Deletes and Range Lookups. The range lookup operation retrieves all entries within a specified key range. To accomplish this, the system constructs iterators across all LSM levels and sequentially retrieves the matching keys. In *RocksDB*, iterators should also be created for range tombstone blocks to determine key validity. Similarly, in GLORAN, iterators are built for each DR-tree in the global index, as the effective areas are sorted and sequentially stored on disk. In our experiments, we replace a portion of point lookups with randomly generated range lookups of length 100 and vary the proportion (range lookup ratio) from 0% to 10%. For ease of comparison, we normalize the throughput of all methods and report the results in Table 3. Clearly, GLORAN consistently outperforms all baselines, achieving more than 45% higher throughput than the basic baseline. This demonstrates its sound capability in handling range lookup operations efficiently.

Exp 8: Performance of Lethe. Lethe [50] also discusses range deletion in LSM-tree. Whereas it targets other operation, *secondary range delete*, which is applied to secondary key. This is fundamentally different from our target. Nevertheless, we also evaluate its performance and compare with other methods. To this end, we adopt the code released by author [49] and implement range deletes via the *deleterange* API. As Figure 11 (e) illustrates, Lethe delivers relatively limited throughput that is consistently deteriorated as more point lookups are applied. This is not surprising because of the different design targets and techniques. To accelerate deletion on the secondary key, Lethe introduces KiWi, a new data layout tailored for its application. However, this simultaneously complicates the workflow of other operations like point lookup, especially when range deletes are considered.

Exp 9: Micro Benchmark db_bench. We further evaluate the range delete methods using the db_bench microbenchmark provided in RocksDB [18] to generate test cases with different ratios of updates and point lookups. We randomly replace 10% of updates with range deletes, while varying the percentage of point lookups from 10% to 90%, and show the results in Table 5. Across all cases,

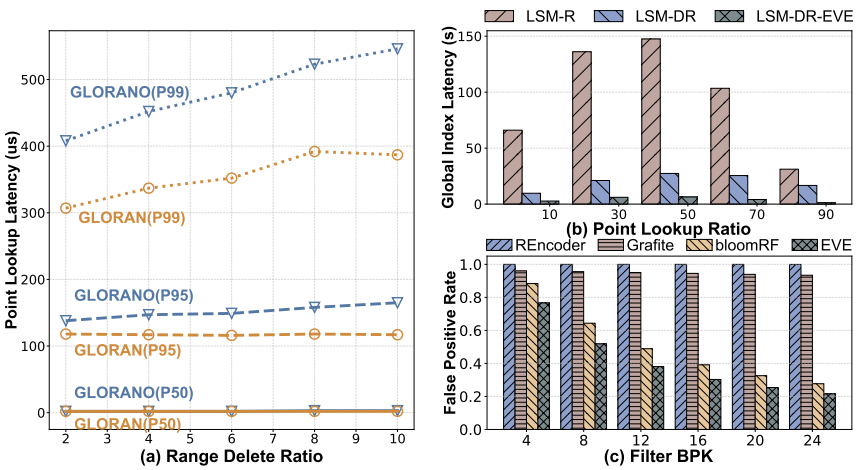

Table 4: Normalize throughput under YCSB.

	Workload			
	Point-L	Balance	Update	Range-L
Decomp	1.01×	1.00×	1.29×	1.00×
Scan&D	1.02×	1.16×	1.03×	1.17×
Look&D	1.04×	1.10×	1.00×	1.08×
RocksDB	1.00×	1.11×	1.63×	1.08×
GLORAN	1.29×	2.11×	3.87×	1.54×

Table 5: Normalize throughput under db_bench.

	Point Lookup Ratio (%)				
	10	30	50	70	90
Decomp	1.04×	1.06×	1.04×	1.00×	1.12×
Scan&D	1.01×	1.00×	1.00×	1.14×	1.22×
Look&D	1.00×	1.02×	1.00×	1.16×	1.26×
RocksDB	3.81×	2.56×	1.99×	1.66×	1.00×
GLORAN	4.51×	3.08×	2.63×	2.54×	1.92×

Figure 13: Results on additional benchmarks YCSB (Table 4) and db_bench (Table 5). We further present point lookup latency (a) and global index query latency (b) to evaluate LSM-DRtree's efficacy, as well as compare EVE with other range filters (c).

GLORAN achieves the best overall performance, reaching up to 1.5 times the throughput of the strongest baseline. Similar to Figure 9, the range record-based methods like *RocksDB* and GLORAN are more competitive in update-heavy workloads. Nevertheless, GLORAN still achieves 52% higher throughput than the most performant point delete based method, *Lookup&D*, under lookup-heavy workload that only contains 10% updates, indicating GLORAN's impressive performance of both range deletes and point lookups.

Exp 10: YCSB Benchmark and Zipfian Distribution. We introduce YCSB [60], another widely used benchmark to provide a more comprehensive assessment. We utilize four typical workloads under Zipfian distribution, *Point-L* contains 90% point lookups and 10% updates, *Balance* contains 50% point lookups and 50% updates, *Update* contains 10% point lookups and 90% updates, *Range-L* contains 20% range lookups and 80% updates. Since YCSB does not include range deletes, we randomly replace 10% of updates with range deletes. According to Table 4, GLORAN delivers the most competitive performance across all four workloads. In particular, it achieves 3.9× improvements over the baseline in update-heavy workloads. It is observed that the enhancement in lookup-heavy is less significant. In Zipfian distribution, a certain part of keys are updated and queried more frequently. In this case, most queried keys are valid, thus mitigating the efficacy of EVE. Even though GLORAN still apparently higher throughput than baseline by 1.3 and 1.5 times under Point-L and Range-L workloads, respectively. These results confirm GLORAN's capability of handling various benchmarks and diverse access patterns.

Exp 11: Efficacy of LSM-DRtree and Entry Validity Estimator. Figure 13 (a) compares the point lookup performance of GLORAN with GLORANO, which utilizes LSM-Rtree as the global range record index. We prepare the database with 100M updates and replace different ratios of operations with range deletes. Then, we conduct 50M point lookups to collect the latencies and illustrate the median (P50) and tail (P95 and P99) values in the figure. Obviously, GLORAN achieves lower tail latency under both percentiles. Specifically, its P99 latency is smaller than GLORANO by almost 1.5× when 10M range deletes are included in the workload. Moreover, the performance of GLORAN is more constant as the range delete ratio increases, while GLORANO presents more apparent rises in latency. Besides, both methods achieve a satisfying median performance. This indicates that GLORAN can not only effectively mitigate point lookup overhead but also deliver more predictable and robust performance under diverse workloads. This owes to the two core designs of GLORAN, the LSM-DRtree-based global index and range encoding filter, which are further assessed.

Figure 13 (b) exhibits the latency for querying the global index under three designs, global index using LSM-R and LSM-DR, and LSM-DRtree integrated with EVE (LSM-DR-REF). We include 100M operations to vary the point lookup ratio while keeping range delete ratio as 10%. The results show that LSM-DR significantly outperforms LSM-R, which delivers up to 6.2× quicker latency, by effectively reducing redundant range record access. With the integration of EVE, it further reduces global index latency by more than 3.5× across all cases. This effectively evidences the efficacy the LSM-DRtree and entry validity estimator.

We further compare EVE with three competitive range filters that support dynamic updates: Grafite [10], REncoder [58], and bloomRF [44]. Figure 13 (c) presents the their FPR when assigning diverse bits per range delete(BPK). Specifically, we insert 14M random ranges with a length of 100, followed by 1M random queries. The capacity of the first RAE is set to 2M for EVE. The results indicate that EVE consistently achieves the lowest false positive rate across all BPKs, especially under higher BPKs. This is under expectation since our estimator actually addresses the different problem of range filters, which highlights the capability of key range encoding instead of individual key points. Meanwhile, BloomRF also considers key ranges and coarsely encodes them with prefixes, thus performing better than other filters. However, our EVE still delivers more than 20% better FPR across all memory budgets.

7 CONCLUSION

This paper proposes a novel range delete method, GLORAN, that efficiently manages the range records with a global index, which offers competitive and robust range delete performance as well as satisfying lookup performance. We further integrate it with an entry validity estimator to enhance it for better performance.

REFERENCES

[1] 2025. Technical Report for GLORAN. https://anonymous.4open.science/r/GLORAN-56FF/GLORAN_Technial_Report.pdf.

[2] Muhammad Yousuf Ahmad and Bettina Kemme. 2015. Compaction management in distributed key-value datastores. *Proceedings of the VLDB Endowment* 8, 8 (2015), 850–861.

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

[4] Sattam Alsubaiee, Yasser Altowim, Hotham Altwaijry, Alexander Behm, Vinayak Borkar, Yingyi Bu, Michael Carey, Inci Cetindil, Madhusudan Cheelangi, Khurram Faraaz, et al. 2014. AsterixDB: A scalable, open source BDMS. *arXiv preprint arXiv:1407.0454* (2014).

[5] Bishwaranjan Bhattacharjee, Timothy Malkemus, Sherman Lau, Sean Mckeough, Jo-Anne Kirton, Robin Von Boeschoten, and John Kennedy. 2007. Efficient Bulk Deletes for Multi Dimensionally Clustered Tables in DB2.. In *VLDB*. 1197–1206.

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

[7] Guanduo Chen, Zhenying He, Meng Li, and Siqiang Luo. 2024. Oasis: An Optimal Disjoint Segmented Learned Range Filter. *Proceedings of the VLDB Endowment* (2024).

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

[9] Zehao Chen, Bingzhe Li, Xiaojun Cai, Zhiping Jia, Zhaoyan Shen, Yi Wang, and Zili Shao. 2021. Block-lsm: An ether-aware block-ordered lsm-tree based key-value storage engine. In *2021 IEEE 39th International Conference on Computer Design (ICCD)*. IEEE, 25–32.

[10] Marco Costa, Paolo Ferragina, and Giorgio Vinciguerra. 2024. Grafite: Taming Adversarial Queries with Optimal Range Filters. *Proceedings of the ACM on Management of Data* 2, 1 (2024), 1–23.

[11] Yifan Dai, Yien Xu, Aishwarya Ganesan, Ramnatthan Alagappan, Brian Kroth, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. 2020. From WiscKey to Bourbon: A Learned Index for Log-Structured Merge Trees. In *14th USENIX Symp. on Operating Systems Design and Implementation (OSDI 20)*. 155–171.

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

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

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

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

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

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

[18] Facebook. 2025. RocksDB. https://github.com/facebook/rocksdb.

[19] Facebook. 2025. RocksDB Benchmark. https://github.com/facebook/rocksdb/wiki/Benchmarking-tools.

[20] Facebook. 2025. RocksDB tuning guide. https://github.com/facebook/rocksdb/wiki/RocksDB-Tuning-Guide.

[21] A Gartner, Alfons Kemper, Donald Kossmann, and Bernhard Zeller. 2001. Efficient bulk deletes in relational databases. In *Proceedings 17th International Conference on Data Engineering*. IEEE, 183–192.

[22] Haoyu Huang and Shahram Ghandeharizadeh. 2021. Nova-LSM: a distributed, component-based LSM-tree key-value store. In *Proceedings of the 2021 International Conference on Management of Data*. 749–763.

[23] Andy Huynh, Harshal Chaudhari, Evimaria Terzi, and Manos Athanassoulis. 2021. Endure: A Robust Tuning Paradigm for LSM Trees Under Workload Uncertainty. *arXiv preprint arXiv:2110.13801* (2021).

[24] Andy Huynh, Harshal A Chaudhari, Evimaria Terzi, and Manos Athanassoulis. 2024. Towards flexibility and robustness of LSM trees. *The VLDB Journal* 33, 4 (2024), 1105–1128.

[25] Stratos Idreos, Niv Dayan, Wilson Qin, Mali Akmanalp, Sophie Hilgard, Andrew Slavin Ross, James Lennon, Varun Jain, Harshita Gupta, David Li, and Zichen Zhu. 2019. Design Continuums and the Path Toward Self-Designing Key-Value Stores that Know and Learn. In *Proceedings of the Biennial Conference on Innovative Data Systems Research (CIDR)*.

[26] Influxdata. 2025. InfluxDB. https://https://www.influxdata.com/.

[27] Influxdata. 2025. InfluxDB Delete Data. https://docs.influxdata.com/influxdb/v2/write-data/delete-data/.

[28] Taewoo Kim, Alexander Behm, Michael Blow, Vinayak Borkar, Yingyi Bu, Michael J Carey, Murtadha Hubail, Shiva Jahangiri, Jianfeng Jia, Chen Li, et al. 2020. Robust and efficient memory management in Apache AsterixDB. *Software: Practice and Experience* 50, 7 (2020), 1114–1151.

[29] Eric R Knorr, Baptiste Lemaire, Andrew Lim, Siqiang Luo, Huanchen Zhang, Stratos Idreos, and Michael Mitzenmacher. 2022. Proteus: A Self-Designing Range Filter. In *Proceedings of the 2022 International Conference on Management of Data*. 1670–1684.

[30] Cockroach Labs. 2025. CockroachDB. https://github.com/cockroachdb/cockroach.

[31] Avinash Lakshman and Prashant Malik. 2010. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review* 44, 2 (2010), 35–40.

[32] Meng Li, Deyi Chen, Haipeng Dai, Rongbiao Xie, Siqiang Luo, Rong Gu, Tong Yang, and Guihai Chen. 2022. Seesaw Counting Filter: An Efficient Guardian for Vulnerable Negative Keys During Dynamic Filtering. In *Proceedings of the ACM Web Conference 2022*. 2759–2767.

[33] Timo Lilja, Riku Saikkonen, Seppo Sippu, and Eljas Soisalon-Soininen. 2006. Online bulk deletion. In *2007 IEEE 23rd International Conference on Data Engineering*. IEEE, 956–965.

[34] Junfeng Liu, Fan Wang, Dingheng Mo, and Siqiang Luo. 2024. Structural Designs Meet Optimality: Exploring Optimized LSM-tree Structures in A Colossal Configuration Space. *PACMMOD* 2, 3 (2024), 1–26.

[35] Lanyue Lu, Thanumalayan Sankaranarayana Pillai, Hariharan Gopalakrishnan, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. 2017. Wisckey: Separating keys from values in ssd-conscious storage. *ACM Transactions on Storage (TOS)* 13, 1 (2017), 1–28.

[36] Chen Luo and Michael J Carey. 2020. Breaking down memory walls: adaptive memory management in LSM-based storage systems. *Proceedings of the VLDB Endowment* 14, 3 (2020), 241–254.

[37] 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 ACM SIGMOD International Conference on Management of Data*. 2071–2086.

[38] Siqiang Luo, Ben Kao, Guoliang Li, Jiafeng Hu, Reynold Cheng, and Yudian Zheng. 2018. Toain: a throughput optimizing adaptive index for answering dynamic k nn queries on road networks. *Proceedings of the VLDB Endowment* 11, 5 (2018), 594–606.

[39] Yina Lv, Qiao Li, Quanqing Xu, Congming Gao, Chuanhui Yang, Xiaoli Wang, and Chun Jason Xue. 2025. Rethinking LSM-tree based Key-Value Stores: A Survey. *arXiv preprint arXiv:2507.09642* (2025).

[40] Qizhong Mao, Mohiuddin Abdul Qader, and Vagelis Hristidis. 2020. Comprehensive comparison of LSM architectures for spatial data. In *2020 IEEE International Conference on Big Data (Big Data)*. IEEE, 455–460.

[41] Dingheng Mo, Fanchao Chen, Siqiang Luo, and Caihua Shan. 2023. Learning to Optimize LSM-trees: Towards A Reinforcement Learning based Key-Value Store for Dynamic Workloads. *arXiv preprint arXiv:2308.07013* (2023).

[42] Dingheng Mo, Junfeng Liu, Fan Wang, and Siqiang Luo. 2025. Aster: Enhancing LSM-structures for Scalable Graph Database. *Proceedings of the ACM on Management of Data* 3, 1 (2025), 1–26.

[43] Dingheng Mo, Siqiang Luo, and Stratos Idreos. 2025. How to Grow an LSM-tree? Towards Bridging the Gap Between Theory and Practice. *Proceedings of the ACM on Management of Data* 3, 3 (2025), 1–25.

[44] Bernhard Mößner, Christian Riegger, Arthur Bernhardt, and Ilia Petrov. 2023. bloomRF: On performing range-queries in Bloom-Filters with piecewise-monotone hash functions and prefix hashing. In *Advances in database technology: Proceedings of the 26th International Conference on Extending database Technology (EDBT), 28th March-31st March 2023, Ioannina, Greece*, Vol. 26. Open Proceedings. org, Univ. of Konstanz, 131–143.

[45] Shiqiang Nie, Tong Lei, Menghan Li, Jie Niu, Song Liu, and Weiguo Wu. 2024. Zone-Aware Persistent Deletion for Key-Value Store Engine. In *2024 13th Non-Volatile Memory Systems and Applications Symposium (NVMSA)*. IEEE, 1–6.

[46] Fengfeng Pan, Yinliang Yue, and Jin Xiong. 2017. dCompaction: Delayed compaction for the LSM-tree. *International Journal of Parallel Programming* 45, 6 (2017), 1310–1325.

[47] Zhu Pang, Qingda Lu, Shuo Chen, Rui Wang, Yikang Xu, and Jiesheng Wu. 2021. ArkDB: a key-value engine for scalable cloud storage services. In *Proceedings of the 2021 International Conference on Management of Data*. 2570–2583.

[48] Pandian Raju, Rohan Kadekodi, Vijay Chidambaram, and Ittai Abraham. 2017. Pebblesdb: Building key-value stores using fragmented log-structured merge trees. In *Proceedings of the 26th Symp. on Operating Systems Principles*. 497–514.

[49] Sarkar. 2021. Lethe. https://github.com/BU-DiSC/lethe-codebase/tree/main.

[50] Subhadeep Sarkar, Tarikul Islam Papon, Dimitris Staratzis, and Manos Athanassoulis. 2020. Lethe: A tunable delete-aware LSM engine. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 893–908.

[51] Jaewoo Shin, Jianguo Wang, and Walid G Aref. 2021. The LSM rum-tree: A log structured merge r-tree for update-intensive spatial workloads. In *2021 IEEE 37th international conference on data engineering (ICDE)*. IEEE, 2285–2290.

[52] Cloudius Systems. 2025. ScyllaDB. https://www.scylladb.com/.

[53] Risi Thonangi and Jun Yang. 2017. On log-structured merge for solid-state drives. In *2017 IEEE 33rd International Conference on Data Engineering (ICDE)*. IEEE, 683–694.

[54] Kapil Vaidya, Subarna Chatterjee, Eric Knorr, Michael Mitzenmacher, Stratos Idreos, and Tim Kraska. 2022. SNARF: a learning-enhanced range filter. *Proceedings of the VLDB Endowment* 15, 8 (2022), 1632–1644.

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

[56] Hengrui Wang, Te Guo, Junzhao Yang, and Huanchen Zhang. 2024. GRF: A Global Range Filter for LSM-Trees with Shape Encoding. *PACMMOD* 2, 3 (2024), 1–27.

[57] Zhiqi Wang and Zili Shao. 2023. MirrorKV: An Efficient Key-Value Store on Hybrid Cloud Storage with Balanced Performance of Compaction and Querying. *Proceedings of the ACM on Management of Data* 1, 4 (2023), 1–27.

[58] Ziwei Wang, Zheng Zhong, Jiarui Guo, Yuhan Wu, Haoyu Li, Tong Yang, Yaofeng Tu, Huanchen Zhang, and Bin Cui. 2023. Rencoder: A space-time efficient range filter with local encoder. In *2023 IEEE 39th International Conference on Data Engineering (ICDE)*. IEEE, 2036–2049.

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

[60] Yahoo! 2010. Yahoo! Cloud Serving Benchmark(YCSB). https://github.com/brianfrankcooper/YCSB.

[61] Ting Yao, Jiguang Wan, Ping Huang, Xubin He, Qingxin Gui, Fei Wu, and Changsheng Xie. 2017. A light-weight compaction tree to reduce I/O amplification toward efficient key-value stores. In *Proc. 33rd Int. Conf. Massive Storage Syst. Technol.(MSST)*. 1–13.

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

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

[64] Huanchen Zhang, Hyeontaek Lim, Viktor Leis, David G Andersen, Michael Kaminsky, Kimberly Keeton, and Andrew Pavlo. 2018. Surf: Practical range query filtering with fast succinct tries. In *Proceedings of the 2018 International Conference on Management of Data*. 323–336.

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

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

[67] Yueming Zhang, Yongkun Li, Fan Guo, Cheng Li, and Yinlong Xu. 2018. ElasticBF: Fine-grained and Elastic Bloom Filter Towards Efficient Read for LSM-tree-based KV Stores. In *10th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 18)*.

[68] Zichen Zhu, Ju Hyoung Mun, Aneesh Raman, and Manos Athanassoulis. 2021. Reducing bloom filter cpu overhead in lsm-trees on modern storage devices. In *Proceedings of the 17th International Workshop on Data Management on New Hardware (DaMoN 2021)*. 1–10.