GRF: A Global Range Filter for LSM-Trees with Shape Encoding

HENGRUI WANG, Institute for Interdisciplinary Information Science, Tsinghua University, China
TE GUO, Department of Computer Science, Purdue University, USA
JUNZHAO YANG, Institute for Interdisciplinary Information Science, Tsinghua University, China
HUANCHEN ZHANG*, Institute for Interdisciplinary Information Science, Tsinghua University, China

Log-structured merge-trees (LSM-trees) are widely used in key-value stores because of its excellent write performance. To reduce LSM-tree's read amplification due to overlapping sorted runs, each file (i.e., SSTable) in an LSM-tree is typically associated with a point or range filter to reduce unnecessary I/Os to the runs that do not contain the target key (range). However, as modern SSDs get faster, probing multiple in-memory filters per query often makes the system CPU bottlenecked, thus compromising the system's throughput. In this paper, we developed the Global Range Filter (GRF) for RocksDB that reduces the number of filter probes per query to one. We follow the pioneering Chucky's approach by storing the sorted run IDs within the filter. However, we identify two practical challenges in building a global range filter: correctness in multi-version concurrency control and efficiency in frequent updates. We solve both challenges by the novel Shape Encoding algorithm. With further optimizations, GRF achieves a dominating performance over the state-of-the-art filters under different workloads when integrated into RocksDB.

CCS Concepts: • **Information systems** → **Data Structures**; • **Storage systems**;

Additional Key Words and Phrases: Range Filters, LSM-Trees, MVCC

ACM Reference Format:
Hengrui Wang, Te Guo, Junzhao Yang, and Huanchen Zhang. 2024. GRF: A Global Range Filter for LSM-Trees with Shape Encoding. *Proc. ACM Manag. Data* 2, 3 (SIGMOD), Article 141 (June 2024), 27 pages. https://doi.org/10.1145/3654944

1 INTRODUCTION

The Log-structured merge-tree (LSM-tree) is the basis of many modern key-value stores [1, 2, 4] that powers a wide range of real-world applications [15, 23, 32, 35]. An LSM-tree buffers inserted keys in memory and then flushes them to disk in batches as sorted runs. Sorted runs are organized into levels, and smaller sorted runs in upper levels are periodically compacted to lower levels according to different compaction policies [49]. Each sorted run can contain multiple files, called SSTables. Because the key ranges of these sorted runs overlap, a point query must probe each sorted run to find a particular key-value pair, causing significant read amplification.

*Huanchen Zhang is also affiliated with Shanghai Qi Zhi Institute.

Authors' addresses: Hengrui Wang, wang-hr21@mails.tsinghua.edu.cn, Institute for Interdisciplinary Information Science, Tsinghua University, China; Te Guo, guo777@purdue.edu, Department of Computer Science, Purdue University, USA; Junzhao Yang, yang-jz20@mails.tsinghua.edu.cn, Institute for Interdisciplinary Information Science, Tsinghua University, China; Huanchen Zhang, huanchen@tsinghua.edu.cn, Institute for Interdisciplinary Information Science, Tsinghua University, China.

ACM 2836-6573/2024/6-ART141
https://doi.org/10.1145/3654944

	Build Locally	Build Globally
Point	Bloom[6] Ribbon[21] XOR[31]	Chucky[18]
Range	SuRF[54] SNARF[51] Rosetta[41] REncoder[52]	Remix(?)[55]

Table 1. State-of-the-art filters for LSM-trees.

To reduce such read amplification, LSM-trees typically equip each SSTable with a filter data structure (i.e., point filter [6, 13, 21, 26, 27, 31, 47, 56] or range filter[34, 41, 51, 52, 54]) to reduce unnecessary I/Os to the sorted runs that do not have the target key or key range. Under this construction, a key-value lookup must probe one filter (pinned in memory) per sorted run. Such a CPU cost is no longer negligible with much faster modern SSDs. Recent studies [10, 18, 55] have shown that the time taken by probing the Bloom filters constitutes a significant portion of the query latency on Optane SSDs, especially when using the state-of-the-art lazier compaction policies [11, 16, 17]. We further found that filter probing would make the system CPU bottlenecked even with a standard NVMe SSD with asynchronous I/O enabled [3].

Therefore, building one global filter for the entire LSM-tree becomes an attractive solution. Chucky [18] is a pioneer work that constructs a global Cuckoo filter on top of RocksDB [4]. The key idea of Chucky is to efficiently store the sorted-run ID for each key along with its fingerprint in the Cuckoo filter so that a key-value lookup can be directed to the sorted run containing the key after probing the global filter. Essentially, Chucky functions as a memory-efficient approximate (i.e., with false positives) hash index, but it cannot handle range queries. Remix [55] attempts to build a space-efficient global range index for the LSM-tree. However, it sacrifices the ability to filter unnecessary I/Os for negative queries (details in Section 2).

Table 1 summarizes the existing filters for LSM-trees. The purpose of this paper is to fill in the missing piece of global range filters. A basic idea is to follow Chucky's approach by embedding each key's run ID into an existing range filter. However, we identify two practical challenges regarding its correctness and efficiency.

First, storing a single run ID for each key, as done in Chucky and Remix, fail to support the Multi-Version Concurrency Control (MVCC) typically adopted by modern LSM-tree systems such as RocksDB [4]. These systems rely on background threads to perform compaction with concurrent read operations. Therefore, each query is confined to a snapshot, and different snapshots may exist simultaneously at run time. As a result, a key may appear temporarily in multiple physical positions in different snapshots, and the single run ID in the global filter (no matter up-to-date or not) may lead to incorrect query results (details in Section 3.1).

Second, unlike local filters that are rebuilt only at compaction, a global filter must handle individual inserts and updates efficiently. To achieve this, Chucky requires pre-allocating space for insertion before the next last-level compaction, resulting in up to 2× memory overhead, while Remix needs to read all keys in the system for rebuilding at each memory table flush. Moreover, each LSM-tree compaction would trigger run-ID updates for a considerable portion of keys in the global filter. The corresponding CPU overhead is significant because those run IDs are scattered in the global filter, causing cache misses.

Term	Definition
L	total levels in LSM-tree (Level $L-1$ is the last level.)
T	LSM-tree size ratio
M	bitset length in SNARF and GRF
K	bits per key in sparse bitset
l	key's physical level
r	key's physical sorted run in its level
s	number of keys per segment in Remix
C	Learned CDF Model in SNARF and GRF

Table 2. Terms in this paper.

In this paper, we address both challenges by introducing the Global Range Filter (**GRF**) for LSM-trees. GRF uses the state-of-the-art range filter SNARF as the base data structure. The key idea in GRF is to store the current shape (i.e., number of sorted runs per level) of the LSM-tree for each key in the global filter when it is flushed to disk. We refer to this approach as **Shape Encoding**(SE). The beauty of SE is that it represents a key's run-ID trace during its lifetime, that is, the key's entire future positions are determined at its insertion time. By storing the SE for each key in GRF, it no longer requires the costly run-ID updates at every compaction because the SE already contains all the future run IDs. Moreover, we can use the SE for each snapshot as its identifier and determine the correct run ID for the querying key by intersecting the snapshot identifier with the key's SE in GRF. To reduce the space overhead of storing the keys' SEs, we developed a garbage collection mechanism to achieve a memory overhead similar to Chucky's Huffman Encoding.

We evaluated the performance of GRF in RocksDB. We compared GRF to the following baselines: Chucky[18], Monkey[13], SuRF[54], SNARF[51], REncoder[52] and Remix[55]. The following comparisons are under the condition of the same filter sizes (i.e., bits per key). For negative point queries, GRF is on par with Chucky while outperforming locally-built filters by an order of magnitude. For negative range queries where Chucky is unable to accelerate, GRF exhibits a similar ($\approx 10\times$) performance advantage over the locally-built filters. For positive queries, GRF outperforms the baselines by 33% and 2× on NVMe SSD and Optane SSD, respectively. When asynchronous I/O is enabled, the speedup improves to $4-6\times$ over the baseline filters.

We make the following contributions in this paper. First, we identify the two challenges in building a global (range) filter for LSM-trees: correctness in concurrency control and efficiency in frequent updates. Second, we propose the novel Shape Encoding for storing the sorted run IDs in the global filter that solves both challenges with a minimal space overhead. Finally, we build the Global Range Filter (GRF) and integrate it into RocksDB. Experiments show that GRF dominates the other solutions in improving the performance of RocksDB under different workloads. We hope our work fills in the last missing piece in the research of LSM-tree filters, as indicated in Table 1.

2 BACKGROUND AND RELATED WORK

LSM Tree Overview. The LSM tree maintains multiple sorted runs to store data. These sorted runs exist in multiple levels with exponentially increasing capacities. The LSM tree also maintains in-memory tables as write buffers. As shown in Table 2, we assume the LSM tree has L sorted runs, and the size ratio is T. When an in-memory table becomes full, a background thread flushes it to level 0 in the storage. Each level will be compacted to its next level when it reaches its capacity. The compaction process can be triggered by different conditions and has different compaction granularity. According to [49], different compaction methods have different trade-offs. The LSM tree has numerous variants that optimize its various aspects [5, 8, 9, 16, 17, 33, 40, 45, 50].

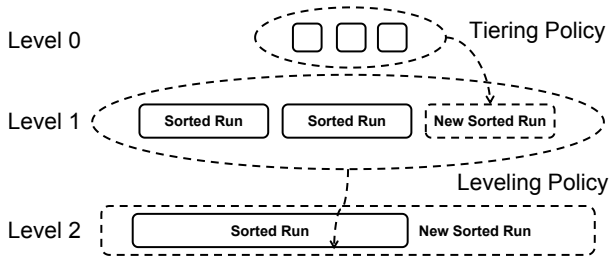


Fig. 1. An overview of two LSM tree compaction policies with full merge.

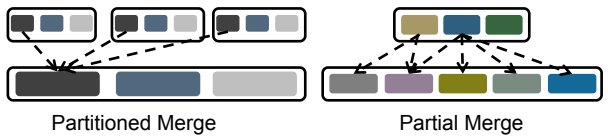


Fig. 2. Different colors represent SSTables with distinct ranges.

Compaction Policies. When a level reaches its compaction conditions, the LSM tree uses background threads to compact the data in that level into the next level. This process is repeated until a level is found that can hold all key-value pairs from upper levels. There are mainly two types of compaction policies for the LSM tree. The first is the leveling policy, which greedily sorts all data in a level. The second is the tiering policy, which allows each level to have up to $T - 1$ sorted runs. As shown in Figure 1, level 0 and level 1 use the tiering policy, and level 2 uses the leveling policy. With the tiering policy, the compaction process does not need to read and rewrite data in the next level, resulting in better write performance. However, it leads to more sorted runs, which can decrease read performance. On the other hand, the leveling policy provides better read performance. Modern LSM trees always using a hybrid policy that using tiering policy in upper levels and leveling policy in lower levels. In RocksDB [4], the first level uses the tiering policy, while the other levels use the leveling policy. Dostoevsky [16] uses the leveling policy for the last level and the tiering policy for others, which is known as **Lazy Leveling**. With the help of filters [13], Dostoevsky can achieve similar read performance as RocksDB while greatly improving write performance.

Compaction Granularity. During compaction, adjacent levels can merge either at the level granularity (full merge) or at the SSTable granularity (partial merge). A full merge compacts the entire level to the first non-full level at once, while a partial merge selects only one SSTable from the current level and merges it with overlapping SSTables in the next level. Spooky [19] introduced the concept of partitioned merge. This approach partitions SSTables in non-last levels based on the boundaries of the last-level SSTables and selects one group of perfectly overlapping files for compaction, as illustrated in Figure 2. Full merge results in lower write amplification but requires temporary space amplification. In contrast, partial merge leads to higher write amplification because an SSTable in the target level may overlap with multiple SSTables in the upper level, resulting in multiple rewrites, as demonstrated by the pink SSTable in Figure 2. Partitioned merge combines the advantages of both full merge and partial merge. In the case of Spooky, the LSM tree employs partitioned merge for compaction between level $L - 2$ and level $L - 1$, while using full merge for all the upper levels. Upper levels are not allowed to be compacted into level $L - 2$ when level $L - 2$ is in the middle of a partitioned merge to level $L - 1$.

Bloom Filters. LSM trees leverage filters to reduce I/O operations for queries. Various filter data structures [6, 21, 26, 27, 31, 34, 37, 41, 51, 54] are widely employed. Filters can quickly determine the presence of keys within a dataset but with a false positive rate (FPR). In an LSM tree, each SSTable builds a bloom filter to avoid unnecessary I/O. Monkey [13] proposed an optimal memory allocation principle that assigns more bits to upper-level keys, resulting in lower FPR with same memory. There are also some learned filters [7, 12, 20, 25, 34, 38, 39, 43, 44, 48, 53]. Their high-level idea is to leverage query workload knowledge for better performance.

Range Filters. To further optimize range queries, range filters have been proposed [34, 41, 51, 51, 52, 54]. The high-level design involves maintaining distribution information for keys in a compact manner. Some of them [51, 54] accurately record keys' prefixes using different methods but completely ignore suffixes. False positives occur when the queried key range shares a common prefix with existing keys. These types of range filters experience significant performance degradation with correlated workloads. REncoder and Rosetta [41, 52] use a set of bloom filters to approximately record the full key information. In this case, FPR significantly increases for longer ranges. These types of range filters are primarily used for short-range queries or correlated workloads.

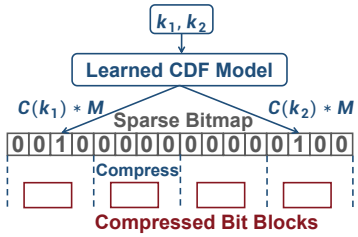
SNARF. SNARF [51] introduces a novel learning-based range filter inspired by learned indexes [22, 28, 29, 36]. SNARF is the state-of-the-art range filter with the lowest FPR and supports incremental updates. SNARF utilizes learned models to fit the cumulative distribution function (CDF) of the entire dataset, enabling it to efficiently store prefixes of keys compared to SuRF. After training the model, SNARF maps each key to a location in a large sparse bitmap with length M and sets the corresponding bit to 1. To query a key range $[lk, uk]$, SNARF simply tests the sparse bitmap from $C(lk) * M$ to $C(uk) * M$ to see if any bits are set. For space efficiency, the set bits positions in the sparse bitmap is then compressed using Golomb coding [30]. The storage of these set bits positions are similar to fingerprints in a fingerprint based filter, such as cuckoo filter. We refer to the set bit position of a key as its **learned fingerprint** in the rest of our paper. Assuming there are N set bits in the sparse bitmap, Golomb coding could compress the sparse bitmap to N learned fingerprint with length $2 + log(\frac{M}{N})$. Figure 3a is an overview of SNARF. To improve query and insertion speed, SNARF divides the bitmap into multiple blocks and compresses them individually.

Golomb Coding. The primary concept behind Golomb coding is to represent both the quotient and remainder of each fingerprint with a fixed number K. In Figure 3b, we illustrate the compression of numbers 3, 6, 11, 18 using $K = 4$. These numbers have respective quotients of 0, 1, 2, 4 and remainders of 3, 2, 3, 2. We directly store the remainders in the remainder part with normal binary encoding. In the quotient part, quotients are encoded in unary. Each set bit corresponds to a learned fingerprint (we refer the set bit as the quotient bit for the learned fingerprint), and the count of preceding unset bits signifies the quotient (denoted as quotient number) for that particular fingerprint . In Figure 3b, the first set bit represents the first fingerprint, which is '3', and its quotient is 0. The second set bit represents the quotient for '6', with one preceding unset bit, indicating the quotient is 1.

Decoding Normal Golomb Coding. In regular Golomb coding, we begin by sequentially reading the bits of the quotient part. Two variables, $quotient$ and $rank$, are initialized to 0. When we encounter an unset bit, we increment $quotient$ by one. In the case of a set bit, it signifies a learned fingerprint, which we decode as $quotient \cdot K + remainder[rank]$. Following this, we increment $rank$ by 1 and proceed to read the next bits for further decoding.

Updatability. For incremental updates, SNARF only needs to decode and rebuild a small compressed bit block. This is important for building a global range filter. We choose SNARF as the building block of GRF because of its efficient updates and robust FPR.

Chucky. Chucky [18] replaces Bloom filters with a single Cuckoo filter. The Cuckoo filter stores fingerprints and compressed run IDs for each key. For point queries, we only need to probe a single



(a) SNARF Structure.

(b) An example of Golomb Coding with $K = 4$.

Fig. 3. Overview of SNARF.

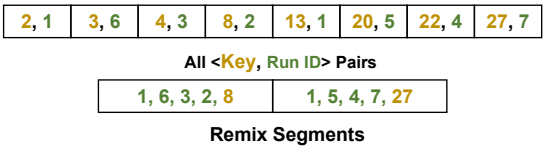


Fig. 4. Remix Overview with $s = 4$.

Cuckoo filter. Since most keys are located in the last level, run IDs are highly compressible. Chucky employs Huffman encoding to compress combinations of run IDs within the Cuckoo filter's buckets. Such compressed run IDs require less than one bit per key to store. As an approximate hash index, Chucky does not provide any benefits for range queries.

Remix. In Remix [55], all keys are logically sorted. Remix partitions consecutive keys into segments. Only the maximum key in each segment is recorded, as illustrated in Figure 4. Within each segment, only run IDs are stored to indicate keys' physical positions. These run IDs are sorted according to their corresponding keys. When serving queries, Remix employs a binary search to locate the correct segments. Within each segment, another binary search helps find the target key and all reported sorted runs must be searched. Assuming there are s keys in a segment, the expected number of I/O operations is approximately $O(log(s))$. This is a significant number for negative queries or point queries.

2.1 FPR of Global and Local Filters

In the case of locally constructed filters, a filter needs to be queried for each sorted run during a query operation. Consequently, the overall FPR increases linearly with the number of sorted runs. This problem is addressed by Monkey, as explained in [13], which maintains a consistent overall FPR for bloom filters by allocating different bpk values for different sorted runs.

With the global filter Chucky, there's no longer a need to query numerous filters. However, we do face the task of allocating bits for run IDs. Binary encoding of run IDs results in additional space overhead. By employing compressed run IDs, Chucky achieves a FPR similar to that of Monkey. In fact, the core principles of compressing run IDs and Monkey's allocation method are fundamentally the same. Similarly, binary encoding run IDs and using uniform bpk bloom filters are also identical.

When it comes to local range filters, Monkey's approach may not be an ideal fit, as range filters are influenced by data distribution. Nevertheless, employing a lightweight encoding of run IDs, similar to Chucky, consistently reduces space overhead for a global range filter. Intuitively, constructing a global range filter with compressed run IDs can still yield a significantly lower FPR when compared to local range filters with uniform bpk values. A theoretical analysis will be presented in Section 4.3. *In summary, building a global range filter can benefit both CPU overhead and LSM tree's overall FPR.*

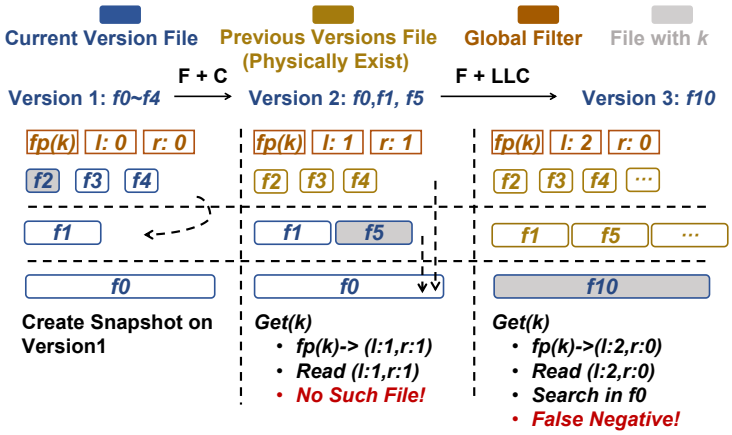


Fig. 5. MVCC problem of global Filter. F stands for *flush*. C stands for *compaction*. LLC stands for *last level compaction*. We assume each sorted run has only one SST file in this figure.

3 PROBLEMS OF EXISTING GLOBAL FILTERS

3.1 The Concurrency Problem

Multi-Versions in LSM Tree. Concurrency control is essential as LSM trees use background threads for flushing and compaction. Most LSM tree implementations employ multi-version concurrency control methods [24]. For instance, in RocksDB, a new version is created after each compaction or flush operation. A version contains a list of files. To ensure consistency, queries are confined to snapshots created within specific versions. In LSM trees, multiple versions always coexist simultaneously. For the same key, queries on different versions may need to read different files. In RocksDB's implementation, snapshots are automatically created from the current version when invoking the *Get* API or creating new iterators. Subsequently created files are invisible in these snapshots. In Figure 5, Version 1 encompasses a list of files: f_4, f_3, f_2, f_1, f_0. A compaction operation may lead to the creation of Version 2. During this process, key-value pairs from f_4, f_3, f_2 are merged into a single file, denoted as f_5 at level 1. A snapshot constructed based on Version 1 cannot access f_5, requiring queries based on this snapshot to search files f_4, f_3, f_2 for the requested data. Therefore, these files cannot be immediately deleted after the creation of f_5. They will continue to exist until all snapshots built on Version 1 have been released. Filters of these files will also coexist in memory. As each filter is small, keeping multiple versions of these files' filters does not incur a huge additional memory overhead.

Lack of MVCC Support. In Chucky and Remix, only the current run ID for each key was recorded. Background threads would update each key's stored run ID during flush and compaction to keep it up-to-date. However, with multiple versions, storing only the current run ID for each key is not sufficient because the key's current file may not be visible in an old snapshot. As a result, a query might need to search through all sorted runs. In the worst-case scenario, a query may even experience false negatives. In LSM trees, it's possible to store filters of old files for snapshots when using small local filters. However, it's not practical to store multiple global filters for different snapshots because global filters are too large.

Example. In Figure 5, Snapshot 1 is created based on Version 1. In Snapshot 1, we may want to query a key k, which is stored in sorted run 0 at level 0 ($f2$). However, background threads continue to perform flushes and compactions simultaneously. This could lead to a change in the

stored run ID of key k. If the current version is Version 2, querying for key k in the global filter may incorrectly report that it is in sorted run 1 at level 1. However, we cannot find any corresponding files in the snapshot for this case. Consequently, we may need to search through all sorted runs one by one. In the event of a last-level compaction, all previous key-value pairs will be stored in the last level, and Version 3 will be created. If Version 3 is the current version, the global filter will report that key k is in the last level. But when we search the last level in Snapshot 1, we cannot find key k. This situation results in a false negative.

3.2 Maintenance Overhead

Key Insert Overhead. Traditionally, during a flush operation, we allocate new space and build a filter for the new SSTable. However, in the case of Chucky, it is necessary to pre-allocate sufficient space because the size of Chucky (number of slots) remains fixed until the next last-level compaction. In the worst case, Chucky can only achieve a 50% load factor, resulting in significant space wastage. The recent work InfiniFilter [14] proposed an elegant approach for dynamically resizing fingerprint-based filters. It assigns longer fingerprints to recent keys. However, applying this idea to Chucky is challenging because Chucky requires longer fingerprints for old keys and shorter fingerprints for recent keys. Maintaining Remix is even more challenging. When a key is inserted into a segment, determining where to place its run ID within the segment is crucial. However, we lack information about the original keys in this segment, necessitating a complete rebuild of Remix. To amortize the overhead of such rebuilding, Remix utilizes a 4GB write buffer. This is an unusual setting.

Run ID Update and Filter Rebuilding Overhead. When keys are compacted to the next level, local filters need to be rebuilt from scratch. The filter rebuilding overhead has a complexity of $O(L + h \cdot T)$, where h represents the number of levels adopting a leveling policy. On the other hand, Chucky must update the stored physical positions (run IDs) of all the keys in the upper level in each compaction. It also requires a complete rebuild during the last-level compaction. The combined overhead has a complexity of $O(L + T)$. Changing run IDs contributes the $O(L)$ term while rebuilding Chucky during last-level compactions contributes the $O(T)$ term. Although Chucky has a lower maintenance complexity, it has poor cache locality because the keys involved in a compaction are scattered in the global filter. According to our evaluation, an LSM tree with Chucky exhibits similar insertion speed compared to an LSM tree with local bloom filters. Directly combining Chucky with SNARF leads to a much-compromised insertion speed with a maintenance complexity of $O(L \cdot \log N + L \cdot R + T)$, where $O(\log N)$ is the overhead of finding a single run ID, and $O(R)$ is the overhead of rebuilding a compressed bit block for updating a run ID.

4 GLOBAL RANGE FILTER

In this section, we introduce GRF, a practical global range filter for LSM trees. We begin by presenting an overview of **GRF**. Subsequently, we delve into how GRF addresses the aforementioned challenges. For ease of discussion, we assume that the LSM tree uses Lazy Leveling with full merge. We will relax this assumption and discuss how to generalize our designs in the next section.

4.1 GRF Overview

GRF is constructed on top of the entire LSM tree to facilitate query processing. It can directly tell where to find target keys and key ranges. In GRF, run ID(s) are stored within a range filter. The central component of GRF is SNARF. As shown in Figure 6, each key's run ID, referred to as its Shape Encoding, is stored after its quotient in the compressed bit block. GRF is built using keys in the last level during the last level compaction, and all keys' learned fingerprints are stored in the same GRF.

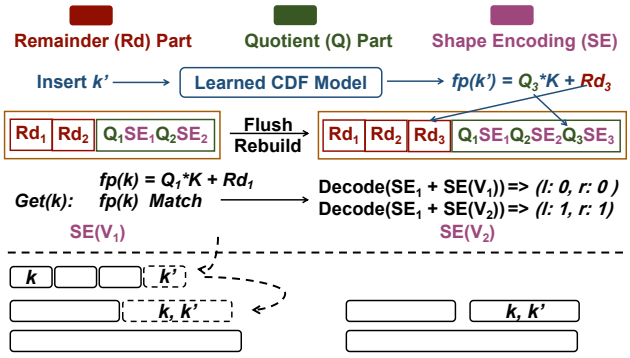


Fig. 6. GRF Overview. We only show a compressed bit block instead of whole global range filter.

During the flush operation, new keys need to be added to GRF. To insert a new key k', we start by calculating its learned fingerprint using the CDF model. Then, we locate the corresponding compressed bit block and modify it to include k''s learned fingerprint and SE. Since we're only rebuilding a small compressed bit block for the insertion, there's no need to allocate memory in advance. This keeps our GRF consistently compact. The model is retrained only during the last-level compaction, and until then, we continue to use the previously learned model. We will explore how incremental insertion might impact GRF's FPR in Section 4.3.

For both point and range queries, we initially follow the query process outlined in SNARF and check for a matching learned fingerprint. If a match is found, we then interact the stored Shape Encoding with the Shape Encoding of the version to determine the accurate physical location. To support MVCC, a key's SE should report different physical positions for different versions. As depicted in Figure 6, when querying k in version 1, SE_1 indicates that k is in sorted run 0 at level 0. In version 2, SE_1 points to sorted run 1 at level 1. Further details are presented below.

4.2 Shape Encoding

LSM Tree's Shape. Our Shape Encoding is based on the shape of the LSM Tree. We define the shape of an LSM tree with a sequence of L numbers. Each number in the sequence represents the number of sorted runs in a level. As illustrated in the first part of Figure 7, the LSM tree has 2 sorted runs at level 0 and level 1, and 1 sorted run at level 2 and level 3. We denote the current shape of the LSM tree as 2211. As flushes and compaction operations proceed, the shape of the LSM tree will change. In the second part of Figure 7, a new sorted run is flushed into level 0, altering the LSM tree's shape to 3211. When another sorted run is flushed, there will be 4 sorted runs at level 0, triggering a compaction from level 0 to level 1. After the compaction, a new sorted run will be created at level 1, and the LSM tree's shape in the third part of Figure 7 is 0311. Finally, after many rounds of flushes and compactions, the LSM tree will compact all sorted runs into a single sorted run at level 3 since level 3 is a leveling level. At this point, the LSM tree's shape is 0001.

Full Shape Encoding (SE) and Future Run ID Trace. We define the LSM tree's shape at a key's flush time as the key's Full SE. As depicted in the left part of Figure 8, when the key k is flushed, the LSM tree's shape is 1211, which is also the Full SE of k. *A key's Full SE is also the key's future run-ID trace.* Specifically, if there is only one sorted run (denoted as sorted run 0) at level 0 when flushing k, k will be in sorted run 1 at level 0. This corresponds to the first number in its Full SE (1211). Similarly, a compaction from level 0 to level 1 will create the third sorted run at level 1 (denoted as sorted run 2). As a result, k will be in sorted run 2 at level 1, as shown in the right part of Figure

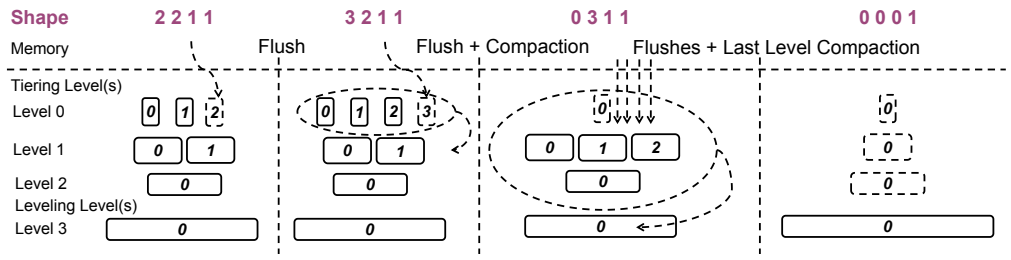


Fig. 7. Examples of Shape Encoding, the rectangle outlined with solid lines represents an existing sorted run. The size ratio T in this case is 4.

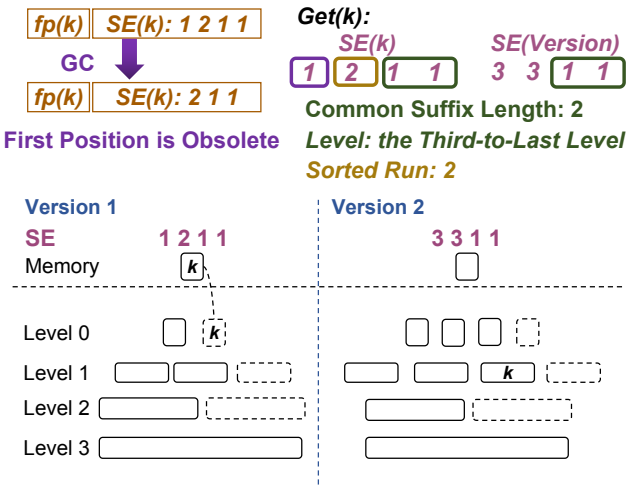


Fig. 8. Search keys in a snapshot with Shape Encoding.

8. This corresponds to the second number at its Full SE. The reason behind this is that the LSM tree rewrites all keys from level 0 to level $i-1$ to a new sorted run at level i during compaction, except for the last level. A key's future positions (denoted as sorted run index) at any level can be determined at its flush time based on the LSM tree's shape at that time. A key will only be stored in those L positions in its Full SE at any time.

Supporting MVCC and Efficient Updates with Full SE. As a key's Full SE can be determined at flush time, we only need to insert its Full SE into GRF once. No further operations are required during compaction, which allows for efficient updates. As discussed earlier, a key will only be stored in those L sorted runs in its Full SE at any time. When querying the key in a snapshot, we first check if its learned fingerprint exists. After finding a matching learned fingerprint, we can obtain the correct query result by searching among those L positions in the corresponding Full SE. This enables MVCC support. However, searching through L sorted runs for each query is slow. We can further reduce the number of target sorted runs from L to 1 (ignoring FPR here) by leveraging Snapshot's SE.

Shape Encoding for Versions and Snapshots. After a flush or compaction alters the shape of the LSM tree, a new version is created. We define the Shape Encoding of a version and the snapshots built on it as the LSM tree's shape at that version. A version's Shape Encoding is exactly the Full

Shape Encoding of the flushed keys at that version. As illustrated in Figure 8, the Shape Encoding of version 1 is also 1211, which is also the Full Shape Encoding of key k.

Summary of Terminologies. We denote a key's SE as KSE and the SE of a version/snapshot of the LSM tree as VSE. We refer to the number for level i in an SE as a position, denoted by $KSE[i]/VSE[i]$. We say an SE is larger than another SE if it has more sorted runs in larger levels. For example, 1211 is larger than 3301 because $1 > 0$ at level 2. Besides, we define the time between two consecutive last-level compaction as a *compaction round*. A snapshot's lifetime may span across a last-level compaction, we define such snapshot as a *cross-round snapshot*. We use m to denote the maximum common suffix length between a KSE and a VSE.

Reducing Number of Target Sorted Runs. To avoid searching through L sorted runs, we must accurately determine the level at which the key is located. Then, we can utilize the level's corresponding number in the KSE to identify the correct sorted run at that level. The correct level can be identified in a snapshot by extracting the maximum common suffix between the KSE and the snapshot's VSE. If the KSE and VSE share a common suffix with a length of m (i.e., $KSE[L-1-i] = VSE[L-1-i]$ for $i = 0, 1, \ldots, m-1$ and $KSE[L-m-1] \neq VSE[L-m-1]$), it indicates that the last m levels have remained unchanged from the key's flush time to the snapshot's time. This implies that the key's level must be in the range from level 0 to level $L-m-1$, including both endpoints. $KSE[L-m-1] \neq VSE[L-m-1]$ indicates that keys in upper levels have been compacted into level $L-m-1$ at least once. In such cases, the key's level must be in the range from level $L-m-1$ to level $L-1$. As a result, we can conclude that the key resides at level $L-m-1$ in this snapshot. This approach enables an SE to consistently report only one correct sorted run in any snapshot.

Example. As depicted in Figure 8, k's KSE is 1211, indicating that when compacted to level 0, k will be in sorted run 1, and when compacted to level 1, it will be in sorted run 2. We store this sequence in the GRF when inserting key k. Assuming the current version is version 2 and its VSE is 3311, we want to find the position of k in version 2. k's KSE is 1211 and the version's VSE 3311 share a common suffix of length 2 (11), which means that the number of sorted runs at level 1 has changed while level 2 remains the same. From this, we deduce that k must be at level 1 in version 2. We can then determine k's position at level 1 by referring to its stored sequence, which indicates that it belongs to sorted run 2 at level 1.

Garbage Collection. Storing L positions for each key can be expensive. We observed that only the maximum common suffix and the first position before the maximum common suffix are relevant when querying a key in a snapshot. Consequently, if a key's stored KSE has a maximum common suffix with the oldest snapshot's VSE of length m, we can store only the last $m+1$ positions in KSE ($KSE[i]$ for $i = L-m-1, \ldots, L-1$) and discard other positions. In Figure 8, if the oldest snapshot was built on version 2, we can clear key k's KSE from 1211 to 211 because $m = 2$ in this case (as shown in Figure 8). To handle variable-length KSEs, we use unary encoding. We use a 1-bit indicator for each position in the KSE to mark the end of an KSE. For example, for KSE 231, where $2(010_2)$ and $3(011_2)$ are not the last positions, their indicators will be set to 0. These positions will be stored as **0**010 and **0**011, respectively. On the other hand, position $1(001_2)$ is in the last level, so its indicator will be set to 1, and it will be recorded as **1**001. Additionally, as all keys' positions in the last level are the same due to the leveling policy of the last level, 001 can be omitted, and only the indicator is necessary. As a result, KSE 211 will be represented as (**0**010)(**0**001)(**1**) and will be stored as **001**(010)(001) with the indicators grouped together. Garbage collections are performed lazily when new entries are inserted into a compressed bit block (i.e., at flush time). In cases of insertion skew, obsolete positions in certain blocks may not get GCed promptly. To address this, we monitor the size of each compressed bit block and trigger GC periodically on the blocks larger than a predefined threshold.

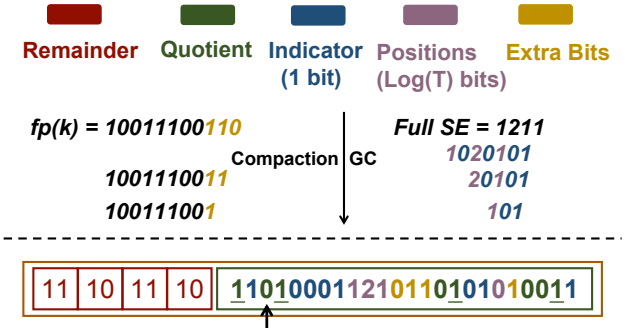


Fig. 9. Varying-length Learned Fingerprint and compressed bit block's layout. The upper half and the lower half of this figure are independent.

Dealing with Cross-Round Snapshots. After a last-level compaction, the LSM tree's shape transitions to 0001, and a new compaction round begins. For ease of management, we set all *KSE*s in the new GRF to a single set bit since all keys are in the last level. However, the new last level is invisible to cross-round snapshots. We, therefore, must keep the original GRF around until the old cross-round snapshots are no longer referred. Because the *KSE*s of the last-level keys (i.e., = **1**) in the original GRF stay the same in the new GRF, we only need to maintain the non-last-level keys in the original GRF to save memory: the size of the shrunk original GRF is only $\approx \frac{1}{T}$ of that of the new GRF. Consequently, for a query referring to a cross-round snapshot, we search both the shrunk original GRF for the previously non-last-level keys and the new GRF for the rest.

Dealing with Duplicate GRF Entries. There might be identical learned fingerprints in GRF caused either by fingerprint collision of nearby keys or by updates or deletes of the same key[1]. GRF allows identical fingerprints if they belong to different sorted runs (i.e., with different *KSE*s). If they end up in the same sorted run, one of the duplicates will be remained in GRF and others will be removed during garbage collection. If one of the duplicated learned fingerprints corresponds to a tombstone entry, both duplicates can be removed. However, during garbage collection, we are unable to determine whether a learned fingerprint corresponds to a tombstone entry or not. We, therefore, must keep both duplicates until the next GRF rebuild at last-level compaction, trading memory space for compaction speed. Alternatively, we can eagerly handle tombstones by deleting the corresponding GRF entry during compaction when the tombstone has been merged with the corresponding key, trading compaction speed for memory efficiency.

4.3 Maintaining Robust FPR

The GRF exhibits the lowest FPR (denoted as p) at the start of a compaction round, when it has just been rebuilt. To understand the impact of inserting new keys into the GRF on its FPR. We differentiate between two types of updates: in-distribution updates that follow the distribution of existing data, and out-of-distribution updates that do not.

4.3.1 FPR is Robust for In-distribution Updates. With in-distribution updates, the positions of set bits in the sparse bitmap are uniformly distributed. For a sparse bitmap of size M with N set bits[2]

[1]LSM trees use out-of-place deletions and updates. Initially, a new version (or tombstone) of a key is inserted into the mem-table. The actual deletions and updates take place when different versions of the same key are compacted into the same sorted run.

[2]Following SNARF's paper, we also assume that the number of set bits is equal to the number of keys.

its FPR is approximately $\frac{1}{K} = \frac{N}{M}$. M remains unchanged in a compaction round. For N, suppose inserting every B keys to the LSM tree will trigger a last-level compaction, and the last-level compaction has already happened z times, then there will be a total of $z \cdot B$ keys in the last level at the moment. Thus, N ($\approx$ the total number of keys in the LSM tree) will increase by $\frac{1}{z}$ for the next compaction round. The average FPR in this compaction round is $\frac{1}{2}(p + (1 + \frac{1}{z}) \cdot p) = p + \frac{p}{2 \cdot z}$, where p denotes algorithm's FPR at the start of a compaction round. As the value of z varies from 1 to $T - 1$, the average FPR over time is $(1 + \frac{\frac{1}{2 \cdot 1} + \frac{1}{2 \cdot 2} + \ldots + \frac{1}{2 \cdot (T-1)}}{T-1}) \cdot p = (1 + \frac{\ln(T-1)}{2 \cdot (T-1)})p$. When $T = 10$, GRF has an average FPR of $1.12 \cdot p$.

4.3.2 Optimizing Robustness for Out-of-Distribution Updates. In GRF, out-of-distribution keys can lead to a higher FPR because the learned CDF model does not fit these keys. A natural idea is to use a larger sparse bitmap and longer learned fingerprints for these keys to compensate for the loss of accuracy in the learned model. An insight can be incorporated into such a design: more recent keys should have longer learned fingerprints. Because the learned model is trained/constructed using keys in the last level, more recent keys, such as those in level 0 and level 1, are more likely to be out-of-distribution.

Our Method. As discussed above, a key's learned fingerprint should be gradually shortened as it is compacted downward. During insertion, we generate a long fingerprint for the key. When the key is compacted to the next level, the least significant bit is removed from its fingerprint. This process of shortening the fingerprint is conducted during garbage collection. A key at level l should have $L - l - 1$ extra bits[3] for its fingerprint. In the original SNARF, a key's learned fingerprint is calculated as $\lceil C(k) \cdot M \rceil$, where M is determined by the bpk budget. In our GRF, a larger number, denoted as $M' = M \cdot 2^{L-1}$ is used, where $L - 1$ represents the number of non-last levels. After being compressed by Golomb Coding, the generated learned fingerprints of newly inserted keys at level 0 have extra $L - 1$ bits of suffixes. When the key has moved through $L - 1$ levels to the last level, these extra bits have all been cleared, and the length of the fingerprint is equal to that of the original SNARF. This approach ensures that the stored learned fingerprint for each key aligns with our insight. In this paper, prefix comparison is adopted to compare two fingerprints. For instance, to compare 10011101_2 and 10010111111_2, it can be determined that the first one is greater by comparing 10011_2 and 10010_2. Through prefix comparison, learned fingerprints of varying lengths can maintain the correct order of their corresponding keys.

Example. If $C(k) = 0.612$, $M = 256$, and $L = 4$, then $M' = 256 \times 2^3 = 2048$. The full fingerprint for k is $\lceil 0.612 \times 2048 \rceil = 10011100110_2$. As a result, the stored learned fingerprint of k will be 10011100110_2 if it is at level 0, and 10011100_2 if it is at level 3. Suppose we are querying k' and $C(k') = 0.613$. With $M = 256$, the initially learned fingerprints for both k and k' are 10011101_2, resulting in a false positive. With M', the learned fingerprint of k' becomes $\lceil 0.613 \times 2048 \rceil = 10011101000_2$. k and k' can be distinguished from each other even if k is in the last level. As illustrated in Figure 9, the KSE of k is 1211. When k has been compacted to level 1, the positions in its SE corresponding to level 0 can be deleted (causing the SE to change from 1211 to 211). At this point, the least significant bit of k's suffix, which is 0, will also be removed. This process will be repeated every time k is compacted to the next level.

Storage and compressed bit block's Layout. For a last-level key, we denote the length of its fingerprint as len. The len-bit prefixes of all fingerprints are compressed into compressed bit blocks through normal Golomb Coding. For keys in non-last levels, their fingerprints still have suffixes, which are stored together with their SEs. As discussed above, if a key is at level l, it has $L - 1 - l$

[3]Starting from the last level, for each level we move up, we compensate the fingerprint with one more extra bit. We could choose to compensate with more, but our current setting already effectively addresses the out-of-distribution problem.

positions in its SE and the same number of bits for the suffix. We need to know the key's level to determine the number of bits to read for the suffix and the number of positions to read for *KSE*. Therefore, we store the indicators of the key right after its quotient bit, followed by its *KSE* and the suffix of its fingerprint. As shown in Figure 9, each underlined 1 represents the quotient bit of a learned fingerprint, followed by the indicators (blue), positions in *KSE* (purple), and suffix (yellow) of this fingerprint. When we first read a set bit in the quotient part, we know it is the quotient bit. We continue reading the following bits until encountering another set bit, which is the indicator bit for the last level. Suppose there are a unset bits between the quotient bit and the indicator bit, then we know that there are a non-last levels and we thus read $a \cdot \log(T)$ bit positions and a extra bits for the suffix.

Example. In Figure 9, assuming $K = 4$, the first bit is a set bit, so it is a quotient bit. It does not have any preceding unset bit, so its quotient number is 0. It is followed by another set bit, so we could know that the corresponding key of this quotient bit is a last-level key. After reading the remainder part with index 0, we know the learned fingerprint is $4 \times 0 + 3 = 3$ (0011_2). Then, we start to decode the next learned fingerprint. When we read another set bit in the quotient part, we know it is a quotient bit. We will read the remainder part with index 1 and get a remainder of 2. As the second set bit has only 1 preceding unset bit (as pointed in Figure 9), the learned fingerprint's prefix is $4 \times 1 + 2 = 6$ (0110_2). Then we decode the indicators. We find 3 unset bits before the next set bit. Consequently, we read the following 3 positions (121) and 3 extra bits (011) as the learned fingerprint's suffix. As a result, the key has a learned fingerprint 0110011 and its *KSE* is 1211. The last 1 in its *KSE* serves as the indicator for the last level.

4.4 Space Overhead

We provide an analysis of the static space overhead of GRF in this section. Because of the interference of snapshots, it is hard to compute the space overhead in a specific scenario. In this following analysis, we assume that no snapshots are built on previous versions and that all obsolete positions in *KSE*s have been cleared. Also, we provides an upper bound for the worst-case space overhead when using periodical GC.

Shape Encoding and Extra Bits Space Overhead. The keys in the last m_{th} level require m bits for indicators ($m - 1$ indicators are 0, one indicator is 1), $m - 1$ positions ($\log(T)$ bits for each non-last level), and $m - 1$ extra bits for learned fingerprints. The number of keys in the last m_{th} level accounts for $\frac{1}{T^{m-1}}$ of all keys. As a result, the average bits for a *KSE* can be calculated as: $\sum_{m=1}^{L} \frac{(1+(m-1)\cdot(2+\log(T)))}{T^{(m-1)}} = \frac{T}{T-1} + \frac{T(2+\log(T))}{(T-1)^2}$. When $T = 10$, it is approximately 1.6 bits. Assuming that GC is periodically conducted when compaction happens at level $L - 2$. All keys at level $L - 1$ and level $L - 2$ have been garbage-collected. Meanwhile, assuming that all keys in other levels have $L - 1$ positions in their *KSE*s (no GC has been conducted on them at all), then the additional bpk is bounded by $\frac{1+(L-1)\cdot(2+\log(T))}{T^2}$ (less than 0.2 bit when $T = 10$).

GRF Overhead. SNARF can achieve an FPR of $\frac{1}{2^{\text{BPK}-3}}$ [51]. According to the analysis above, GRF requires less than 2 extra bpk for Shape Encoding and extra bits in the worst case when $T = 10$. Therefore, in our implementation, GRF can achieve an FPR of $p = \frac{1}{2^{\text{BPK}-5}}$ at the beginning of a compaction round and an average FPR of $1.12 \cdot p$ throughout the round. Compared to the use of local SNARFs, our GRF can achieve a better FPR with the same BPK when there are more than 5 sorted runs. This is a common case in modern LSM trees, as level 0 is always a tiering level.

5 GENERALIZATIONS AND LIMITATIONS

In this section, we discuss how to generalize Shape Encoding to different compaction policies and compaction granularity. We first propose the basic requirements of Shape Encoding for LSM

trees, followed by a discussion of its limitations. Then, we demonstrate how to generalize Shape Encoding to example common LSM tree designs that meet the basic requirements.

Basic Requirements. Shape Encoding requires the LSM tree to have a deterministic compaction scheduling. *For any key, we must know its future run-ID trace at insertion time. Also, we must be able to locate the key's current run-ID in the trace (i.e., to compute the number of compactions/rewrites the key has experienced) according to the key's KSE and the LSM tree's VSE.* Shape Encoding applies to any compaction design that meets the above requirements.

Limitations. The partial merge strategy, which is typically associated with the leveling policy, violates the above requirements because a key can get rewritten an arbitrary number of times at each level. However, [19] has shown that a partitioned merge, which satisfies our basic requirements, can achieve a better performance than a partial merge. In the case of a large number of delete and update entries, a non-last leveling level may not reach its capacity threshold after T compactions. We must force the level to trigger a compaction to the next level to stick to a deterministic compaction schedule. This may lead to a worse write amplification.

Generalization 1: Concurrent Flush and Compaction. If the LSM tree allows flushing the mem-table in concurrent with the level compaction, level 0 could temporarily have more than T SSTables. For example, if the current VSE is 3311 ($T = 4$), and level 0 and level 1 are being compacted to level 2, the keys flushed during this compaction would record their future run-ID traces as $x021$ rather than the current $VSE = 3311$, where x is the number of SSTables flushed from the beginning of this compaction. If these keys are queried before the compaction completes, however, GRF may incorrectly report that they are at level 2. We, therefore, add the following check: if a KSE is larger than the current VSE, and a compaction is taking place, we know that the key was inserted during the compaction and it should currently be in sorted run $T + x$ in level 0.

A similar problem occurs when mem-table flush is allowed during a last-level compaction, where the GRF is under reconstruction. For example, if the current VSE is 3331, and a last-level compaction is on-going, newly flushed keys should record their KSEs as $x001$. The above approach does not work when dealing with queries to these keys during the last-level compaction because the "wrapped-around" KSE is no longer larger than the VSE. We, therefore, introduce a *waiting buffer* in memory to hold these to-be-flushed keys (along with their KSEs) temporarily until the last-level compaction is completed. If a query key is found in the waiting buffer, we can infer that it is currently in sorted run $T + x$ in level 0. The waiting buffer is merged to the GRF when it is rebuilt after the compaction. The size of the waiting buffer is typically small because most LSM trees limit the maximum number of SSTables allowed in level 0.

Generalization 2: Partitioned Merge and Spooky. If the entire LSM tree uses partitioned merge, it can be regarded as multiple small LSM trees with full merge, and each small LSM tree will have a disjoint GRF and an independent VSE. These disjoint GRFs can be retrained individually. All other operations remain the same. For Spooky, when performing a partitioned merge from level $L - 2$ to level $L - 1$, concurrent flushes and compactions could happen in the upper levels. We again use the temporary waiting buffer to handle queries during this period. Note that at most $\frac{1}{T^2}$ of the total keys will be inserted into the waiting buffer. Assuming $T = 10$ and the keys are 64-bit integers, the waiting buffer will cause an additional 0.8 bpk in the worst case.

Generalization 3: Leveling Policy. The Shape Encoding for the leveling policy records the number of compactions (up to T) that already happened at each level rather than the shape of the LSM tree, as in the tiering policy. We can still use the SE to determine a key's current sorted run. If the workload contains a large number of deletes and updates, we will estimate a $T' > T$ to delay the scheduled compaction so that each level can approach its capacity before compaction. The parameter in Shape Encoding is thus set to T' in this scenario.

Generalization 4: Different Size Ratios. For LSM trees that adopt different size ratios for different levels (i.e., different levels have different maximum numbers of sorted runs) [17, 33, 46], Shape Encoding still applies but with a variable-length position for each level. This does not affect SE decoding in GRF because the indicators are parsed first to determine the number of positions (as well as the corresponding number of bits) to read next.

Generalization 5 Adaptive Compaction Policies. There is also research on adjusting the compaction policy adaptively according to the workload [46]. GRF requires the compaction policy to remain unchanged within the same compaction round. Upon starting a new compaction round, the LSM tree is free to change its compaction policy as long as it follows the basic requirements for using GRF, as described before.

6 EVALUATION

6.1 Setup, Datasets and Workloads

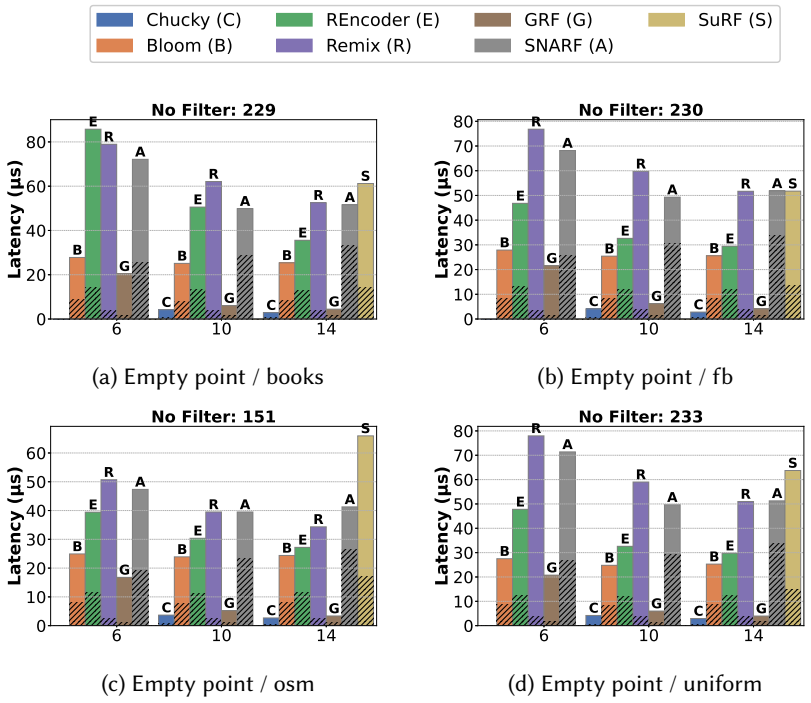
Setup: Our experiments were conducted on a machine with the following specifications: an AMD EPYC 7742 64-Core Processor, 512MB of L3 cache, and two SSDs (800GB Intel Optane SSD DC P5800X and 3TB D7-P5620 NVMe SSD) connected to it. Without further statement, we used the Optane SSD for our experiments. The machine ran Ubuntu 20.04. Throughout the experiments, we used a RocksDB setup with an LSM-tree of size ratio 10 and a lazy leveling compaction policy. We also use the state-of-the-art compaction granularity Spooky during compaction. The memory write buffer was set to 8MB. Fence pointers were maintained for each disk page, and filters were constructed for every file with a default memory allocation of 10 bpk (for SuRF, we used 14 bpk by default). An 8MB block cache was allocated for data blocks. We used 64-bit integer keys and 256 bytes of value for each key. We enabled direct I/Os for both read and write operations. The number of background threads was set to 4. We started with an empty LSM tree and inserted keys until the LSM tree was just prior to the last-level compaction, which occurred when the system contained the highest number of runs. Then we evaluated different types of queries on it to emphasize each filter's worst-case performance.

Dataset: For our experiments, we use datasets from SOSD [42].

- **Uniform Random:** Keys are generated uniformly at random in the range $\left[0, 2^{50}\right]$.
- **Normal:** Keys are generated from normal distribution ($N(\mu = 100, \sigma = 20)$) and are linearly scaled to range $\left[0, 2^{50}\right]$.
- **osm:** cell IDs from Open Street Map representing a location.
- **book:** book sale popularity data at Amazon.
- **fb:** unique Facebook user IDs.

Algorithms: We conducted experiments to show the performance of following algorithms.

- **GRF:** Our implementation with Shape Encoding.
- **GRF_H:** Another implementation with only one Huffman Encoded run ID for each key, like Chucky. This version of GRF highlights the power of Shape Encoding in improving the insertion speed of the LSM tree.
- **Bloom filter:** This is the default filter policy in RocksDB. We follow the optimal configuration of bloom filters provided in Dostoevsky[16] and Monkey[13].
- **Chucky:** We have implemented Chucky based on the cuckoo filter. Each key has only one run ID, encoded by Huffman encoding. We compactly build Chucky with 95% load factor, but we should notice that such a load factor may not always be easily achievable in real cases.
- **Remix:** To avoid the need for complete reconstruction during each flush, our Remix implementation is only **semi-sorted**. Specifically, we do not enforce the sorting of run IDs



(a) Empty point / books

(b) Empty point / fb

(c) Empty point / osm

(d) Empty point / uniform

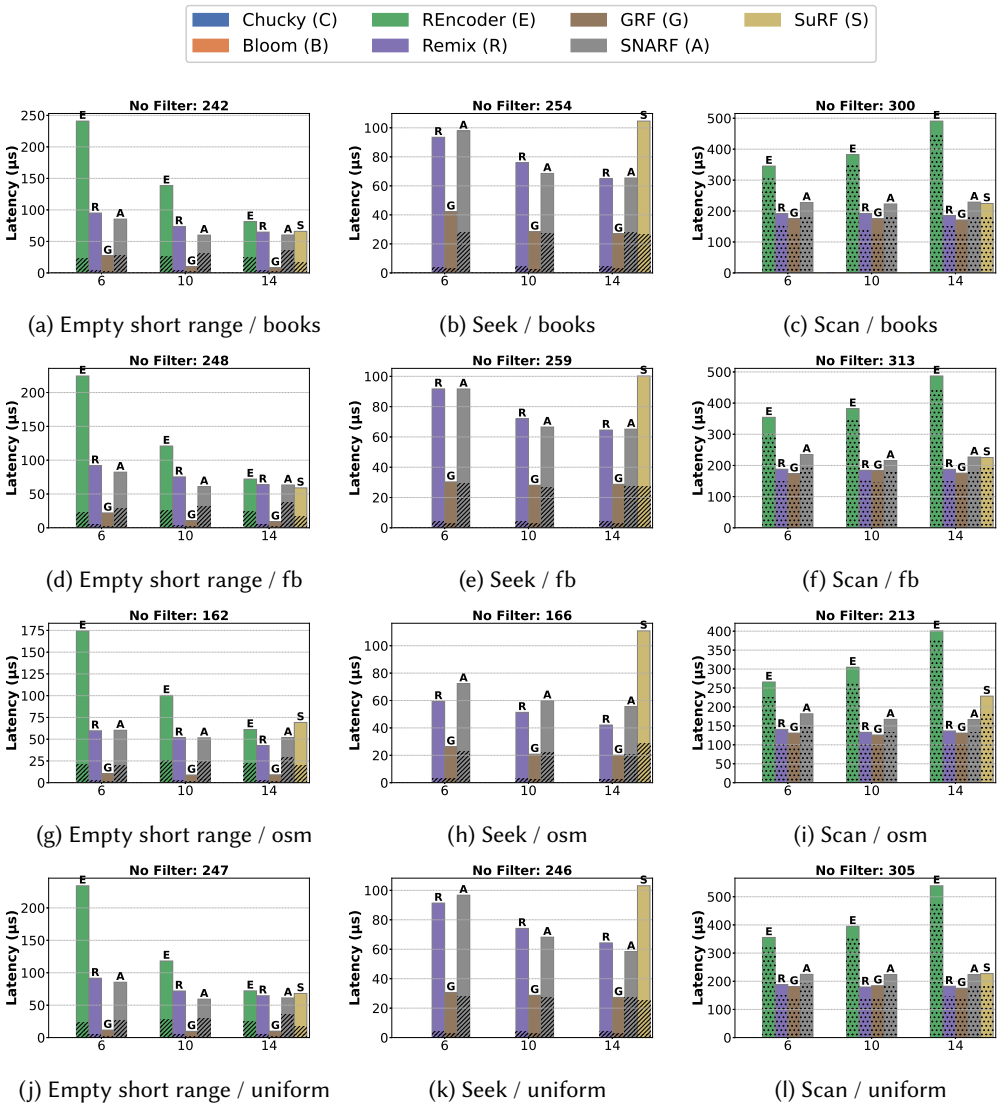
Fig. 10. Point Query Performance on Optane SSD. Shadow part is Filter Probing Time.

within a segment based on their corresponding keys. Instead, we utilize a bitset within each segment to indicate whether a sorted run needs to be searched when querying a key or key range within that segment. While this implementation may result in slightly inferior query performance compared to the original Remix, we believe it is more practical in real-world scenarios due to its ease of maintenance.

- **SuRF:** SuRF stores unique prefixes (or longer) for each key with a compact trie structure. We utilize the open-source implementation of SuRF. Real suffixes are recorded for each experiment.
- **REncoder:** Rosetta [41] encodes range information into multiple layers of Bloom filters for range queries. REncoder [52] further optimized Rosetta's query speed by leveraging locality. These two range filters store full keys approximately. Both Rosetta and REncoder are not suitable for searching long ranges. We utilize the open-source implementation of REncoder. We disable REncoder's query workload sampling. Since REncoder outperforms Rosetta in all aspects of performance, we do not include Rosetta as one of our baselines.
- **SNARF:** We utilize the open-source implementation of SNARF.

6.2 Influence of Bits per Key

In this section, we demonstrate the influence of memory budget on query performance. We vary the number of bits per key for each filter from 6 to 14. We provide both end-to-end performance and filter probing performance. In this section, all queries, whether point or range, are negative queries. Short-range queries involve ranges less than 64. We use the latency of both *Seek* and *Seek*+*Next* to



(a) Empty short range / books (b) Seek / books (c) Scan / books

(d) Empty short range / fb (e) Seek / fb (f) Scan / fb

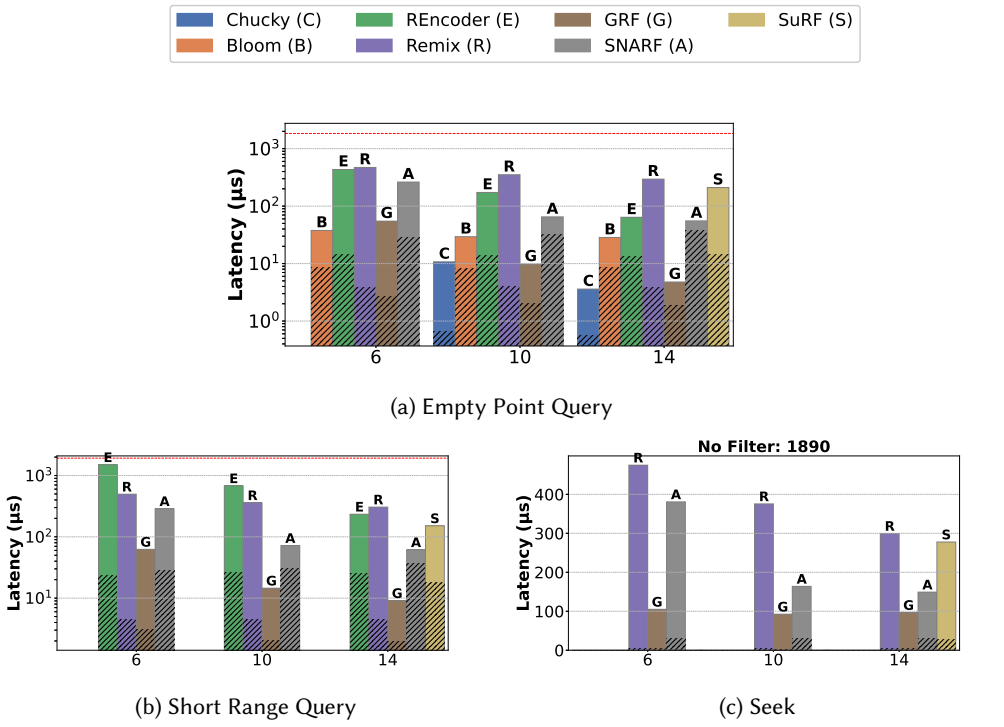
(g) Empty short range / osm (h) Seek / osm (i) Scan / osm

(j) Empty short range / uniform (k) Seek / uniform (l) Scan / uniform

Fig. 11. Range Query Performance on Optane SSD. Shadow part is Filter Probing Time for Seek and Short Empty Range Query Workload. Shadow part is Seek + 20 Next Time for Seek + 50 Next Workload.

illustrate the performance of different data structures for positive long range queries. Throughout figures from Figure 10 to Figure 12, the x-axis represents bpk.

Negative Query Performance. As shown in the figures from Figure 10a to Figure 11j, our GRF exhibits the lowest latency among all algorithms, with only Chucky showing a similar performance on empty point query. For empty queries, filter probing time significantly contributes to the overall latency. Additionally, Chucky and GRF also achieve better FPR compared to other algorithms. These two factors both contribute to the excellent performance of Chucky and GRF. While REncoder is specifically designed for short negative range queries, it does not perform satisfactorily when bpk



(a) Empty Point Query

(b) Short Range Query

(c) Seek

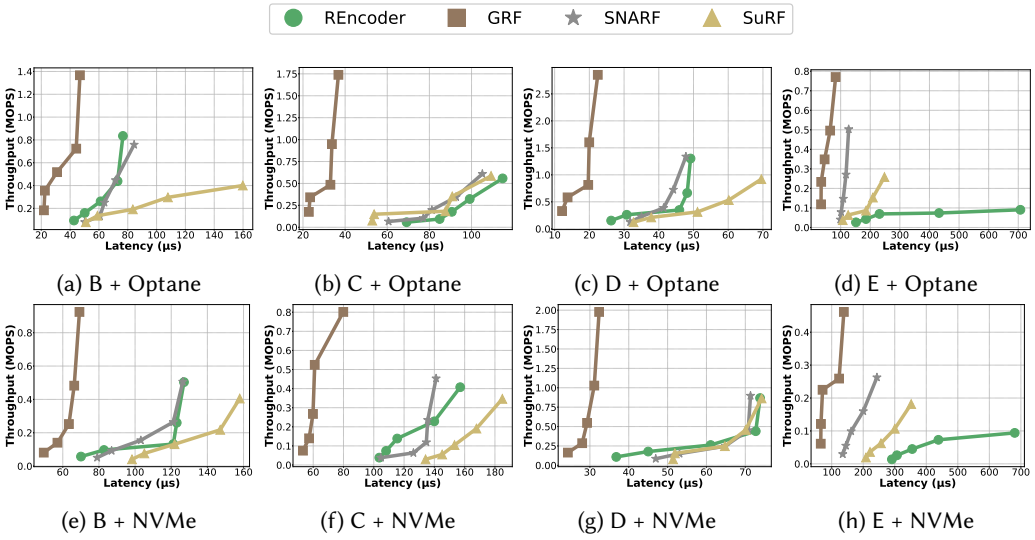
Fig. 12. Experiments on NVMe SSD (books dataset). The red line represents latency without any filters.

is below 14. For SuRF, SNARF and Remix, they have different CPU and I/O overhead but resulting in similarly overall performance for negative queries.

Positive Range Query Performance. As depicted in Figure 11 our GRF achieves the lowest latency for both *Seek* and $Seek + 50Next$ operations. For standalone seek operations, GRF is twice as fast as Remix and SNARF. Remix is only slightly slower than SNARF due to lower CPU overhead. Although *Seek*'s latency is bounded by 1 I/O, the filter probing overhead is also significant, which is why SNARF performs worse than GRF. However, for the $Seek + 50Next$ (scan) operation, Remix, GRF, SNARF, and SuRF all exhibit similar performance, with Remix and GRF slightly ahead. In addition, the scan performance only minor improvements as the bpk increases. This is because necessary I/O operations dominate the overall performance. We demonstrate that further optimization can only be achieved through a more greedy compaction policy. Notably, REncoder can degrade the performance of scan compared to a plain LSM tree because it cannot reduce I/O for such long ranges[4]. Instead, it incurs heavy filter probing overhead.

Performance on NVMe SSD. We further tested each design's performance on NVMe SSD, where filter probing latency contributes little to the overall latency. We examined the latency of empty point queries, empty short-range queries, and seeks. As shown in Figure 12, for empty queries, GRF continues to exhibit significantly better performance compared to other baselines due to advantanges on both CPU and I/O overhead. For seek, GRF demonstrates similar but slightly better performance compared to Snarf. The CPU overhead for Snarf is less pronounced on NVMe SSDs.

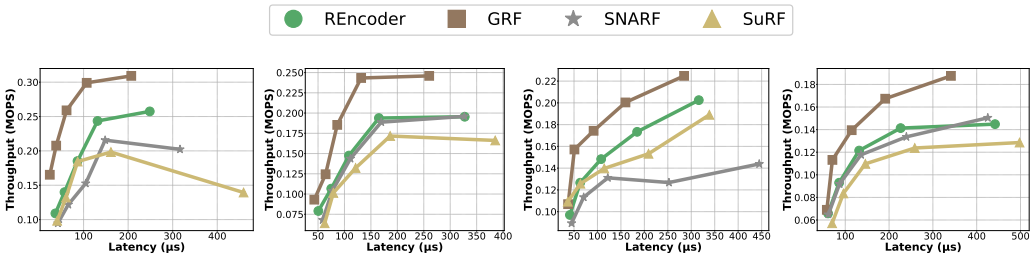
[4]As REncoder does not support Seek and Next operations in the filter, we query the existence of the range from the seeked key to the 50th key after it in REncoder.



(a) B + Optane (b) C + Optane (c) D + Optane (d) E + Optane

(e) B + NVMe (f) C + NVMe (g) D + NVMe (h) E + NVMe

Fig. 13. Experiments on YCSB workloads.



(a) Optane + Lazy Leveling (b) NVMe + Lazy Leveling (c) Optane + Leveling (d) NVMe + Leveling

Fig. 14. Update Performance (YCSB A Workload).

However, Remix and SuRF, while having similar performance to Snarf on Optane SSD, performs significantly worse than Snarf on NVMe SSD because of higher FPR.

6.3 YCSB Performance

YCSB consists a load phase for insertions and five distinct run phases (A through E), each with varying query and update ratios: A) 50% queries, 50% updates. B) 95% queries, 5% updates. C) All queries. D) 95% queries on recent keys, 5% insertions. E) 95% short-range queries, 5% insertions. We used different number of threads (4, 8, 16, 32, 64) to execute the workloads. Since all queries follow a Zipfian distribution, we employed a 1GB block cache to highlight each algorithm's performance on zipf workload. In YCSB C, with the skewed queries and 1GB block cache, the block cache hit rate is about 50%. In each workload, we simulated performance for the worst-case scenario. After completing both the load phase and the run phase, the LSM tree is positioned just before the last-level compaction. In YCSB C, the LSM tree has the most number of sorted runs compared with other workloads. We present the throughput-latency curve for Optane and NVMe SSDs in Figure

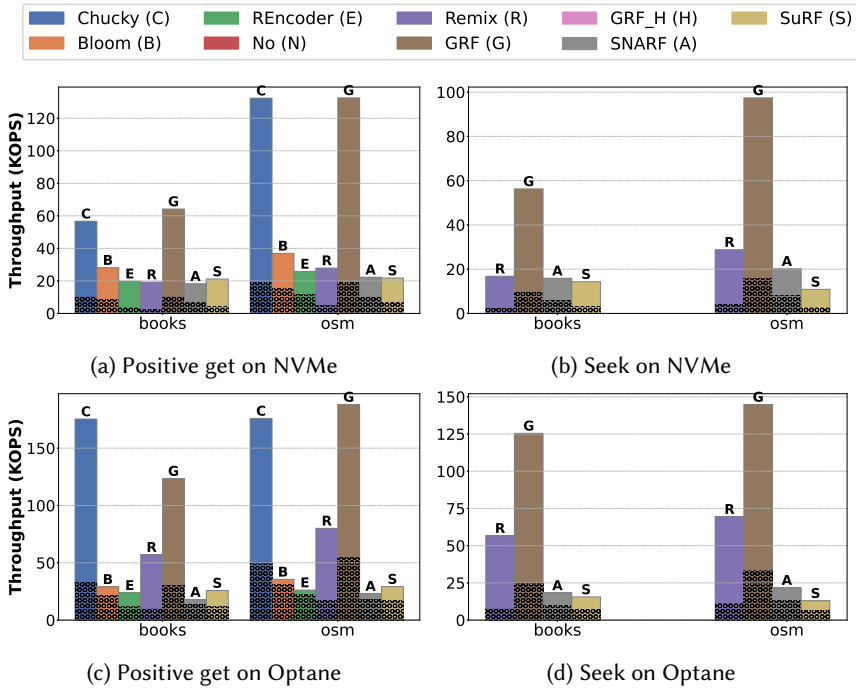


Fig. 15. Experiments with Asynchronous I/O. Shadow part is the throughput with synchronous I/O.

13. In each curve, a higher latency data point indicates more threads. Our GRF outperforms other baselines in all workloads with both SSDs due to its lower FPR and lighter CPU overhead. The curves of our GRF are consistently Pareto optimal, achieving higher throughput with lower latency and fewer threads.

6.4 Update Performance

Our GRF may experience higher write amplification if the workload contains a significant number of update operations and a leveling policy is applied to a non-last level. In this experiment, we employ RocksDB's default level policy (level 0 is a tiering level, and the other levels are leveling levels) to test the performance of each algorithm on YCSB A. For GRF, each level will be compacted to the next level after T compactions. As for the other algorithms, compaction is triggered when a level reaches its capacity. All other configurations remain consistent with previous YCSB experiments. As shown in Figure 14, GRF could outperform other baselines, even though it has higher write amplification. Faster insertion speed and better I/O performance for query make the additional write amplification caused by Shape Encoding's requirements less serious.

6.5 Performance with Asynchronous I/O

Asynchronous I/O is a common approach to improve throughput by harnessing the large bandwidth of modern SSDs. After locating a target data block and dispatching an I/O request to the SSD, the CPU does not need to await the completion of the I/O operation; instead, it can proceed with processing the next query. This approach may lead to an increase in query latency for each query request, but there are situations where throughput takes precedence. As shown in Figure 15, Chucky

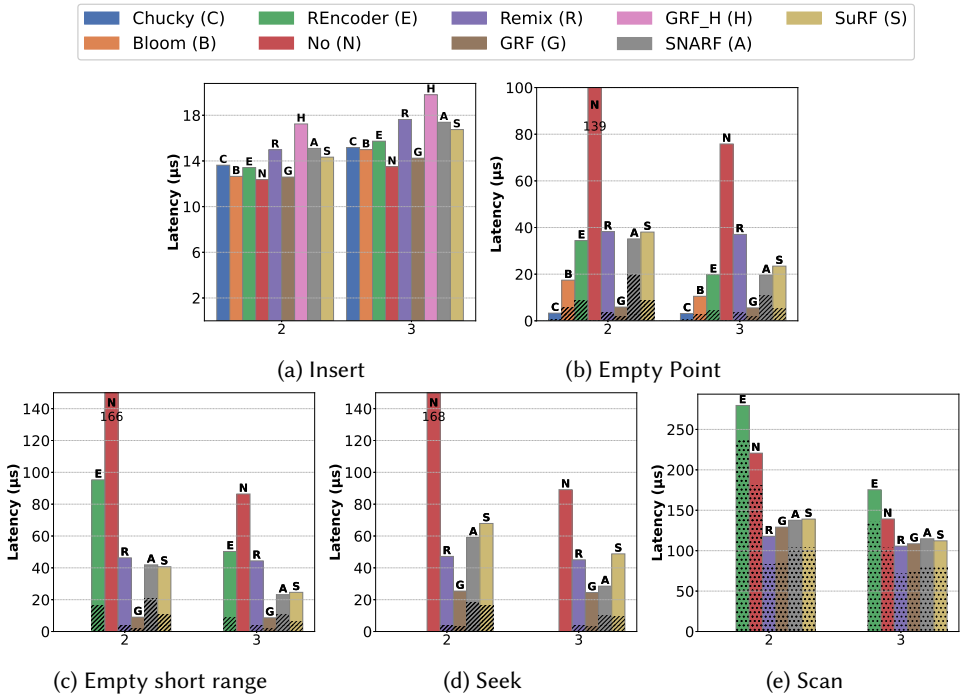


Fig. 16. Experiments on compaction policy . The $x-axis$ stands for the number of leveling levels.

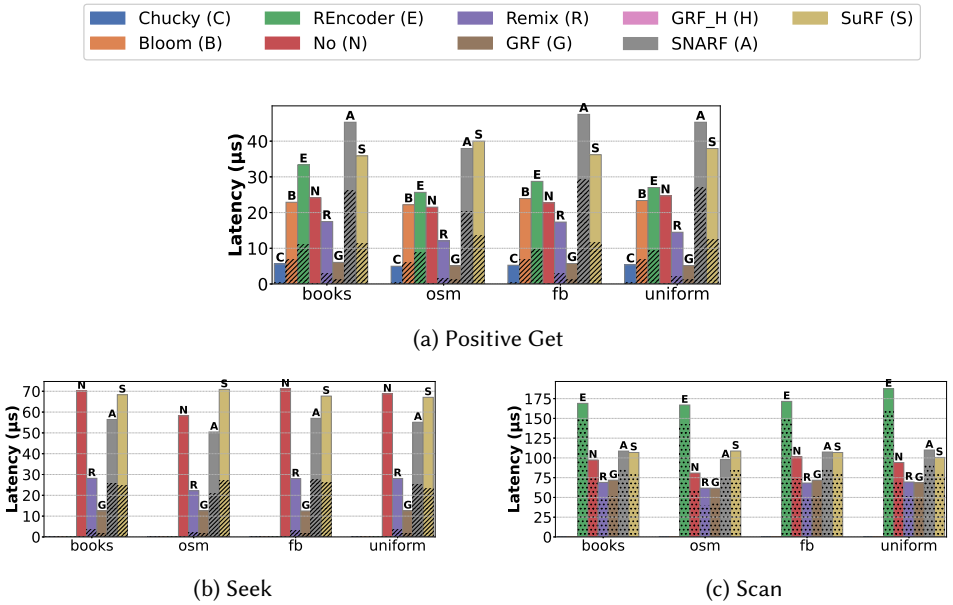
and GRF significantly outperform all other algorithms with asynchronous I/O. Global filters did not exhibit advantages for positive queries in previous experiments on NVMe SSD. However, with asynchronous I/O, the throughput of global filters increases significantly, while the throughput of other algorithms improves only slightly. This is because global filters have lower CPU overhead and can parallelize more I/O operations. These experiments further highlight the growing need for a global filter in LSM trees.

6.6 Influence of Compaction Policy

We present the influence of different compaction policies. Our LSM tree has a total of 4 levels. Previous experiments are conducted with 1 leveling levels. We vary the number of leveling levels. 3 leveling levels represent the default setting in RocksDB. In Figure16, we observe that our GRF achieves the best query performance for all compaction policies on all datasets. Regarding insertion, we find that without workload shifts, our GRF has similar or even lower write latency compared to other algorithms and the plain LSM tree. To highlight the effect of Shape Encoding, we also use Huffman encoding as Chucky. Our results show that a global range filter with Huffman Encoding has significantly higher write latency because the compaction processes frequently change the run IDs, which is expensive for GRF_H. Another important observation is that other algorithms with 3 leveling levels can hardly outperform GRF with 1 leveling level (except for $Seek + Next$). This suggests that GRF can utilize a lazier compaction policy in real-world scenarios.

6.7 Influence of workload

We conduct experiments using a hybrid dataset to simulate a workload shift. We use one dataset to build the model in the last level and insert keys from another dataset into the upper levels. We also



(a) Positive Get

(b) Seek

(c) Scan

Fig. 17. Experiments on highly skewed workload.

conduct experiments with highly skewed query workloads, where most of the related data blocks are cached in memory.

Performance on Highly Skewed Workload. As illustrated in Figure 17, our GRF significantly improves query speed, ranging from 2x to 10x compared to other algorithms in the case of skewed query workloads. Filters build locally can hardly help and may even do harm to the query performance. For skewed query workloads, most of the relevant data blocks are cached in memory. Probing a local filter can only avoid binary searching a small in-memory data block, which is inefficient. Probing a SNARF may need longer time than binary searching the cached data block. Similarly, in the case of *Seek* operations only Remix and GRF exhibit significantly faster performance. Additionally, GRF is 2x faster compared to Remix.

Performance under Workload Shift As depicted in Figure 18, our GRF exhibits higher insert latency but maintains the lowest latency for all types of queries. In cases where keys in non-last levels follow different distributions or come from different real datasets compared to the last level, GRF can lead to skewed access patterns to compressed blocks, causing some compressed blocks to containing more and more entries, which results in longer insertion times. In addition, the insertion skew also triggers periodic garbage collections, contributing to an average write latency of approximately 1 microsecond. To solve the problem, we can split those large compressed blocks, with additional 64-bits boundary for each block. This is a trade-off between space and insertion speed. We found that by allocating more bits to upper-level keys, our GRF can still outperform all other algorithms for all types of queries.

7 CONCLUSION

In this paper, we introduced GRF, the first practical global range filter for LSM trees. We recognized the growing demand for reducing filter probing overhead in LSM trees and identified issues with existing global filter design approaches. Our proposed Shape Encoding not only enhances insertion speed but also makes the global filter MVCC-friendly. Our experiments demonstrate that GRF

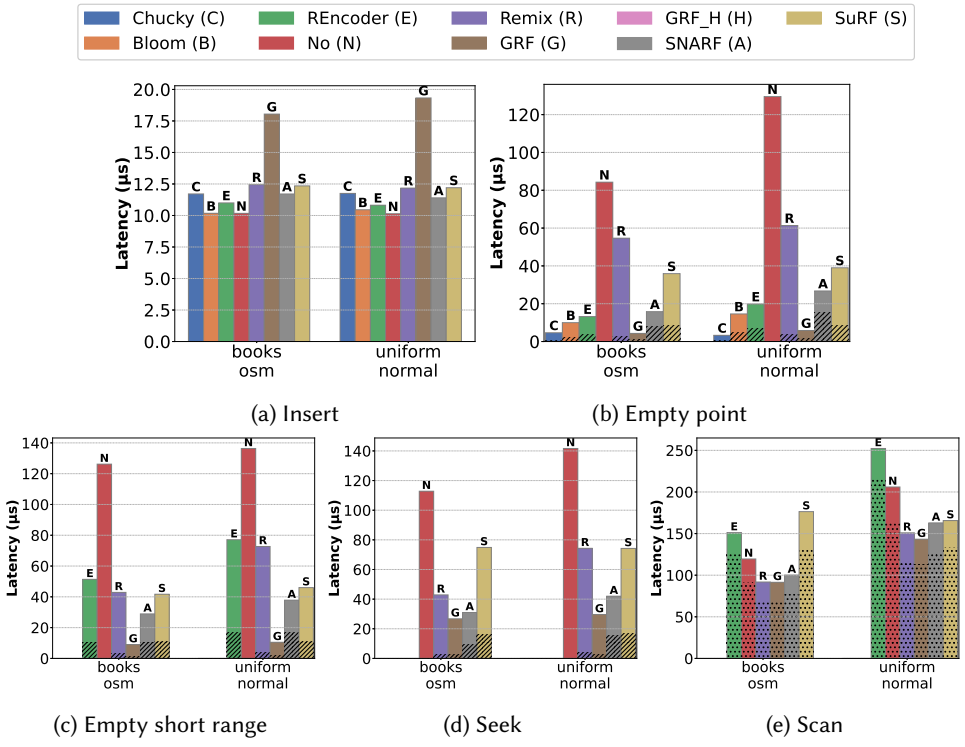


Fig. 18. Experiments on hybrid dataset.

significantly outperforms other algorithms across various query types and even slightly improves insertion speed in typical scenarios.

ACKNOWLEDGEMENT

We thank the anonymous reviewers for their constructive feedback and Niv Dayan for sharing the Dostoevsky and Spooky code. This work was supported by Shanghai Qi Zhi Institute.

REFERENCES

[1] [n. d.]. Apache. Cassandra. http://cassandra.apache.org.

[2] [n. d.]. Apache. HBase. http://hbase.apache.org/.

[3] [n. d.]. Asynchronous IO in RocksDB. https://rocksdb.org/blog/2022/10/07/asynchronous-io-in-rocksdb.html.

[4] [n. d.]. Facebook. RocksDB. https://github.com/facebook/rocksdb..

[5] Oana Balmau, Diego Didona, Rachid Guerraoui, Willy Zwaenepoel, Huapeng Yuan, Aashray Arora, Karan Gupta, and Pavan Konka. 2017. TRIAD: Creating Synergies Between Memory, Disk and Log in Log Structured Key-Value Stores. In *USENIX Annual Technical Conference*. https://api.semanticscholar.org/CorpusID:22824631

[6] Burton H. Bloom. 1970. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM* 13 (1970), 422–426.

[7] Laura Carrea, Alexei Vernitski, and Martin J. Reed. 2016. Yes-no Bloom filter: A way of representing sets with fewer false positives. *ArXiv* abs/1603.01060 (2016). https://api.semanticscholar.org/CorpusID:16308190

[8] Helen H. W. Chan, Yongkun Li, Patrick P. C. Lee, and Yinlong Xu. 2018. HashKV: Enabling Efficient Updates in KV Storage via Hashing. In *USENIX Annual Technical Conference*. https://api.semanticscholar.org/CorpusID:260548458

[9] Subarna Chatterjee, Meena Jagadeesan, Wilson Qin, and Stratos Idreos. 2021. Cosine: A Cloud-Cost Optimized Self-Designing Key-Value Storage Engine. *Proc. VLDB Endow.* 15 (2021), 112–126. https://api.semanticscholar.org/CorpusID:245811523

[10] Alex Conway, Martín Farach-Colton, and Rob Johnson. 2023. SplinterDB and Maplets: Improving the Tradeoffs in Key-Value Store Compaction Policy. *Proc. ACM Manag. Data* 1, 1, Article 46 (may 2023), 27 pages. https://doi.org/10.1145/3588726

[11] Alexander Conway, Abhishek Gupta, Vijay Chidambaram, Martin Farach-Colton, Richard Spillane, Amy Tai, and Rob Johnson. 2020. SplinterDB: Closing the Bandwidth Gap for NVMe Key-Value Stores. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, 49–63. https://www.usenix.org/conference/atc20/presentation/conway

[12] Zhenwei Dai and Anshumali Shrivastava. 2019. Adaptive Learned Bloom Filter (Ada-BF): Efficient Utilization of the Classifier. *ArXiv* abs/1910.09131 (2019). https://api.semanticscholar.org/CorpusID:204800314

[13] 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* (Chicago, Illinois, USA) *(SIGMOD '17)*. Association for Computing Machinery, New York, NY, USA, 79–94. https://doi.org/10.1145/3035918.3064054

[14] Niv Dayan, Ioana Orianna Bercea, Pedro Reviriego, and R. Pagh. 2023. InfiniFilter: Expanding Filters to Infinity and Beyond. *Proceedings of the ACM on Management of Data* 1 (2023), 1 – 27. https://api.semanticscholar.org/CorpusID:259107003

[15] Niv Dayan, Philippe Bonnet, and Stratos Idreos. 2016. GeckoFTL: Scalable Flash Translation Techniques For Very Large Flash Devices. *Proceedings of the 2016 International Conference on Management of Data* (2016).

[16] 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. *Proceedings of the 2018 International Conference on Management of Data* (2018).

[17] Niv Dayan and Stratos Idreos. 2019. The Log-Structured Merge-Bush & the Wacky Continuum. *Proceedings of the 2019 International Conference on Management of Data* (2019).

[18] 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* (Virtual Event, China) *(SIGMOD '21)*. Association for Computing Machinery, New York, NY, USA, 365–378. https://doi.org/10.1145/3448016.3457273

[19] Niv Dayan, Tamar Weiss, Shmuel Dashevsky, Michael Pan, Edward Bortnikov, and Moshe Twitto. 2022. Spooky: Granulating LSM-Tree Compactions Correctly. *Proc. VLDB Endow.* 15 (2022), 3071–3084.

[20] Kyle Deeds, Brian Hentschel, and Stratos Idreos. 2020. Stacked Filters: Learning to Filter by Structure. *Proc. VLDB Endow.* 14 (2020), 600–612. https://api.semanticscholar.org/CorpusID:230087744

[21] Peter C. Dillinger and Stefan Walzer. 2021. Ribbon filter: practically smaller than Bloom and Xor. *ArXiv* abs/2103.02515 (2021).

[22] Jialin Ding, Umar Farooq Minhas, Hantian Zhang, Yinan Li, Chi Wang, Badrish Chandramouli, Johannes Gehrke, Donald Kossmann, and David B. Lomet. 2019. ALEX: An Updatable Adaptive Learned Index. *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (2019).

[23] Tien Tuan Anh Dinh, Ji Wang, Gang Chen, Rui Liu, Beng Chin Ooi, and Kian-Lee Tan. 2017. BLOCKBENCH: A Framework for Analyzing Private Blockchains. *Proceedings of the 2017 ACM International Conference on Management of Data* (2017).

[24] Siying Dong, Andrew Kryczka, Yanqin Jin, and Michael Stumm. 2021. Evolution of Development Priorities in Key-value Stores Serving Large-scale Applications: The RocksDB Experience. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*. USENIX Association, 33–49. https://www.usenix.org/conference/fast21/presentation/dong

[25] Benoit Donnet, Bruno Baynat, and Timur Friedman. 2006. Retouched bloom filters: allowing networked applications to trade off selected false positives against false negatives. In *Conference on Emerging Network Experiment and Technology*. https://api.semanticscholar.org/CorpusID:672670

[26] Tomer Even, Guy Even, and Adam Morrison. 2022. Prefix Filter: Practically and Theoretically Better Than Bloom. *ArXiv* abs/2203.17139 (2022).

[27] Bin Fan, David G. Andersen, Michael Kaminsky, and Michael Mitzenmacher. 2014. Cuckoo Filter: Practically Better Than Bloom. *Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies* (2014).

[28] Paolo Ferragina and Giorgio Vinciguerra. 2019. The PGM-index. *Proceedings of the VLDB Endowment* 13 (2019), 1162 – 1175.

[29] Alex Galakatos, Michael Markovitch, Carsten Binnig, Rodrigo Fonseca, and Tim Kraska. 2018. FITing-Tree: A Data-aware Index Structure. *Proceedings of the 2019 International Conference on Management of Data* (2018).

[30] R. Gallager and D. van Voorhis. 1975. Optimal source codes for geometrically distributed integer alphabets (Corresp.). *IEEE Transactions on Information Theory* 21, 2 (1975), 228–230. https://doi.org/10.1109/TIT.1975.1055357

[31] Thomas Mueller Graf and Daniel Lemire. 2019. Xor Filters: Faster and Smaller Than Bloom and Cuckoo Filters. *arXiv: Data Structures and Algorithms* (2019).

[32] Gui Huang, Xuntao Cheng, Jianying Wang, Yujie Wang, Dengcheng He, Tieying Zhang, Feifei Li, Sheng Wang, Wei Cao, and Qiang Li. 2019. X-Engine: An Optimized Storage Engine for Large-scale E-commerce Transaction Processing. *Proceedings of the 2019 International Conference on Management of Data* (2019).

[33] Andrew Huynh, Harshal A. Chaudhari, Evimaria Terzi, and Manos Athanassoulis. 2021. Endure: A Robust Tuning Paradigm for LSM Trees Under Workload Uncertainty. *Proc. VLDB Endow.* 15 (2021), 1605–1618.

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

[35] Haridimos Kondylakis, Niv Dayan, Konstantinos Zoumpatianos, and Themis Palpanas. 2018. Coconut: A Scalable Bottom-Up Approach for Building Data Series Indexes. *ArXiv* abs/2006.13713 (2018).

[36] Tim Kraska, Alex Beutel, Ed H. Chi, Jeffrey Dean, and Neoklis Polyzotis. 2017. The Case for Learned Index Structures. *Proceedings of the 2018 International Conference on Management of Data* (2017).

[37] Harald Lang, Thomas Neumann, Alfons Kemper, and Peter A. Boncz. 2019. Performance-Optimal Filtering: Bloom overtakes Cuckoo at High-Throughput. *Proc. VLDB Endow.* 12 (2019), 502–515. https://api.semanticscholar.org/CorpusID:85529414

[38] 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. *Proceedings of the ACM Web Conference 2022* (2022). https://api.semanticscholar.org/CorpusID:248367432

[39] Hyesook Lim, Jungwon Lee, and Changhoon Yim. 2015. Complement Bloom Filter for Identifying True Positiveness of a Bloom Filter. *IEEE Communications Letters* 19 (2015), 1905–1908. https://api.semanticscholar.org/CorpusID:26468122

[40] Lanyue Lu, Thanumalayan Sankaranarayana Pillai, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2016. WiscKey: Separating Keys from Values in SSD-conscious Storage. In *USENIX Conference on File and Storage Technologies*. https://api.semanticscholar.org/CorpusID:11367463

[41] 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* (Portland, OR, USA) *(SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 2071–2086. https://doi.org/10.1145/3318464.3389731

[42] Ryan Marcus, Andreas Kipf, Alexander van Renen, Mihail Stoian, Sanchit Misra, Alfons Kemper, Thomas Neumann, and Tim Kraska. 2020. Benchmarking learned indexes. *Proceedings of the VLDB Endowment* 14, 1 (sep 2020), 1–13. https://doi.org/10.14778/3421424.3421425

[43] Michael Mitzenmacher. 2018. A Model for Learned Bloom Filters and Optimizing by Sandwiching. In *Neural Information Processing Systems*. https://api.semanticscholar.org/CorpusID:54173703

[44] Michael Mitzenmacher, Salvatore Pontarelli, and Pedro Reviriego. 2017. Adaptive Cuckoo Filters. *Journal of Experimental Algorithmics (JEA)* 25 (2017), 1 – 20. https://api.semanticscholar.org/CorpusID:9085650

[45] 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* abs/2308.07013 (2023). https://api.semanticscholar.org/CorpusID:260886977

[46] 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. *Proc. ACM Manag. Data* 1, 3, Article 213 (nov 2023), 25 pages. https://doi.org/10.1145/3617333

[47] Prashant Pandey, Alex Conway, Joe Durie, Michael A. Bender, Martín Farach-Colton, and Rob Johnson. 2021. Vector Quotient Filters: Overcoming the Time/Space Trade-Off in Filter Design. *Proceedings of the 2021 International Conference on Management of Data* (2021). https://api.semanticscholar.org/CorpusID:233238457

[48] Jack W. Rae, Sergey Bartunov, and Timothy P. Lillicrap. 2019. Meta-Learning Neural Bloom Filters. *ArXiv* abs/1906.04304 (2019). https://api.semanticscholar.org/CorpusID:86477253

[49] Subhadeep Sarkar, Dimitris Staratzis, Zichen Zhu, and Manos Athanassoulis. 2021. Constructing and Analyzing the LSM Compaction Design Space. *Proc. VLDB Endow.* 14 (2021), 2216–2229.

[50] Risi Thonangi and Jun Yang. 2017. On Log-Structured Merge for Solid-State Drives. *2017 IEEE 33rd International Conference on Data Engineering (ICDE)* (2017), 683–694. https://api.semanticscholar.org/CorpusID:852089

[51] Kapil Vaidya, Subarna Chatterjee, Eric Knorr, Michael Mitzenmacher, Stratos Idreos, and Tim Kraska. 2022. SNARF: A Learning-Enhanced Range Filter. *Proc. VLDB Endow.* 15, 8 (jun 2022), 1632–1644. https://doi.org/10.14778/3529337.3529347

[52] 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)*. 2036–2049. https://doi.org/10.1109/ICDE55515.2023.00158

[53] Rongbiao Xie, Meng Li, Zheyu Miao, Rong Gu, He Huang, Haipeng Dai, and Guihai Chen. 2021. Hash Adaptive Bloom Filter. *2021 IEEE 37th International Conference on Data Engineering (ICDE)* (2021), 636–647. https://api.semanticscholar.org/CorpusID:235421991

[54] 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* (Houston, TX, USA) *(SIGMOD '18)*. Association for Computing Machinery, New York, NY, USA, 323–336. https://doi.org/10.1145/3183713.3196931

[55] Wenshao Zhong, Chen Chen, Xingbo Wu, and Song Jiang. 2021. REMIX: Efficient Range Query for LSM-trees. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*. USENIX Association, 51–64. https://www.usenix.org/conference/fast21/presentation/zhong

[56] Zichen Zhu. 2023. SHaMBa: Reducing Bloom Filter Overhead in LSM Trees. In *PhD@VLDB*. https://api.semanticscholar.org/CorpusID:259848936

Received October 2023; revised December 2023; accepted February 2024