Mnemosyne: Dynamic Workload-Aware BF Tuning via Accurate Statistics in LSM trees

ZICHEN ZHU, Boston University, USA
YANPENG WEI, Tsinghua University, China
JU HYOUNG MUN, Brandeis University, USA
MANOS ATHANASSOULIS, Boston University, USA

Log-structured merge (LSM) trees typically employ Bloom Filters (BFs) to prevent unnecessary disk accesses for point queries. The size of BFs can be tuned to navigate a memory vs. performance tradeoff. State-of-the-art memory allocation strategies use a worst-case model for point lookup cost to derive a closed-form solution. However, existing approaches have three limitations: (1) the number of key-value pairs to be ingested must be known *a priori*, (2) the closed-form solution only works for a *perfectly shaped* LSM tree, and (3) the model assumes a *uniform query distribution*. Due to these limitations, the available memory budget for BFs is sub-optimally utilized, especially when the system is under memory pressure (i.e., less than 7 bits per key).

In this paper, we design Mnemosyne, a BF reallocation framework for *evolving* LSM trees that does not require prior workload knowledge. We use a more general query cost model that considers the access pattern *per file*, and we find that no system accurately maintains access statistics per file, and that simply maintaining a counter per file significantly deviates from the ground truth for evolving LSM trees. To address this, we propose Merlin, a dynamic sliding-window-based tracking mechanism that accurately captures these statistics. The upgraded Mnemosyne$^+$ combines Merlin with our new cost model. In our evaluation, Mnemosyne reduces query latency by up to 20% compared to RocksDB under memory pressure, and Mnemosyne$^+$ further improves throughput by another 10% when workloads exhibit higher skew.

CCS Concepts: • **Information systems** → **Point lookups**.

Additional Key Words and Phrases: LSM-Trees, Bloom Filter, Optimization

ACM Reference Format:
Zichen Zhu, Yanpeng Wei, Ju Hyoung Mun, and Manos Athanassoulis. 2025. Mnemosyne: Dynamic Workload-Aware BF Tuning via Accurate Statistics in LSM trees. *Proc. ACM Manag. Data* 3, 3 (SIGMOD), Article 190 (June 2025), 28 pages. https://doi.org/10.1145/3725327

1 Introduction

LSM-Tree-Based Key-Value Stores. Log-Structured Merge-trees (LSM trees) [50] have emerged as the core data structure in most modern key-value stores [1, 3–5, 11, 12, 22, 25, 26, 28, 29, 34, 57, 60, 65]. LSM trees are widely adopted because they achieve high ingestion throughput via an *out-of-place* update strategy. With this paradigm, data ingestion operations (including inserts, deletes, and updates) are buffered in memory and eventually flushed to disk as a *sorted immutable run* whenever the buffer fills up. While LSM trees use *compaction* to sort-merge several runs to form fewer but larger runs, there could still be multiple runs to probe when answering a point query before all data is compacted into a single run. To facilitate point lookups, commercial LSM-tree-based key-value

Authors' Contact Information: Zichen Zhu, Boston University, USA, zczhu@bu.edu; Yanpeng Wei, Tsinghua University, China, weiyp21@mails.tsinghua.edu.cn; Ju Hyoung Mun, Brandeis University, USA, jmun@brandeis.edu; Manos Athanassoulis, Boston University, USA, mathan@bu.edu.

This work is licensed under a Creative Commons Attribution 4.0 International License.
© 2025 Copyright held by the owner/author(s).
ACM 2836-6573/2025/6-ART190
https://doi.org/10.1145/3725327

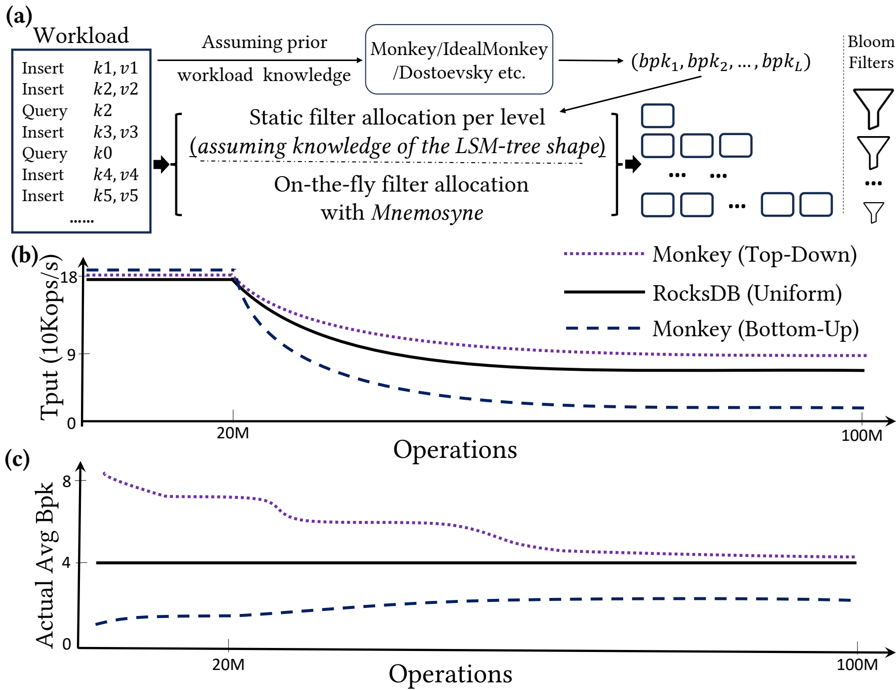


Fig. 1. (a) State-of-the-art models (e.g., Monkey[15]) require prior workload knowledge to compute the bits-per-key per SST and then apply it during LSM tree construction. Instead, Mnemosyne does not require prior workload knowledge and is easy to integrate with existing systems. (b) For a mixed read-write workloads starting with $20M$ inserts, followed by $40M$ queries, $20M$ inserts and $20M$ updates that are mixed in an interleaved manner, building LSM trees with 4 bits-per-key using static Monkey allocation may not have higher throughput than uniform allocation for bottom-up LSM-tree construction. (c) The actual average bits-per-key can be either over utilized or under utilized due to inaccurate LSM-tree shape estimation.

stores typically maintain metadata such as fence pointers and Bloom filters to reduce the number of storage accesses [45].

Bloom Filters in LSM trees. Traditionally, a single *Bloom Filter* (BF) is constructed per level or sorted run. In practical LSM tree implementations like LevelDB [29] and RocksDB [26], a sorted run can span multiple *Sorted-String Table* (SST) files, and each SST file has its own BF. A BF is used to identify whether the desired key belongs to a given file/run with a *False Positive Rate* (FPR). Typically, smaller BFs that encode more entries have a higher FPR, exhibiting a space-accuracy trade-off, controlled by the *bits-per-key* parameter that specifies the ratio between the overall size of BFs and the total number of entries in LSM trees. Since a BF has no false negatives, storage access to the raw data of a file/run can be avoided if the BF returns a negative answer. Traditionally, all BFs are pre-fetched in a read buffer with fixed capacity (termed *block cache* in LSM-based systems) to be readily available during a point query before accessing slow storage. However, when the filters do not fit in memory, we spend a significant number of I/Os fetching them from disk, adversely impacting query performance [49].

The Impact of Memory Pressure on BFs. While both memory and storage prices drop, their respective rates differ. Over the last few years, the price drop for memory has been slower than for storage – since 2007, the unit price per MB for DRAM and SSD has decreased by 90% and 98.3%, respectively [47]. In other words, in 2007, memory was 1.67× more expensive than storage,

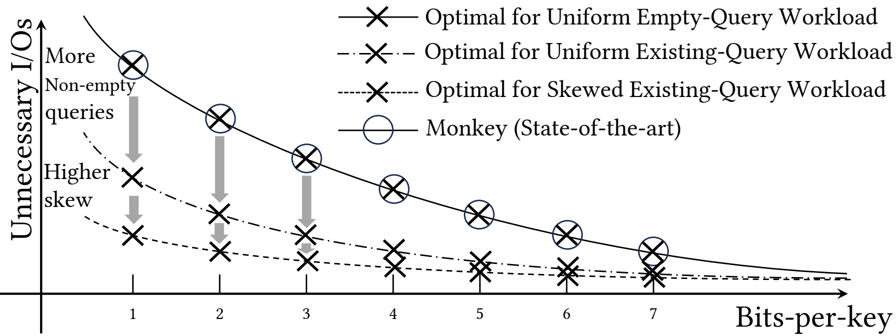


Fig. 2. A conceptual graph that compares the state-of-the-art bits-per-key allocation solution from Monkey and the solution from our workload-aware model.

and in 2018 it has been more than 10× more expensive [6, 49]. In the context of LSM-based systems, the ratio between bits-per-key and the average size of key-value pairs provides a minimum memory-to-storage requirement to ensure all BFs fit in the block cache. As a concrete example, both storage-optimized and compute-optimized EC2 instances in Amazon have less than 3% memory-to-storage size ratio [2], while a BF with 8 bits-per-key needs at least 5% memory-to-storage ratio for 20-byte key-value pairs [7]. As a result, the bits-per-key for BFs has to be small enough to ensure most BFs fit in the block cache. BFs also compete for memory resources with fence pointers, the write buffer, and temporary buffer for compactions. In production systems, when some other applications are running on the same server and consuming the memory space, the memory-to-storage ratio can be as low as 1% [40], forcing no larger than 8 bits-per-key to ensure filters can fit in memory [13]. As BFs in LSM tree are often configured with small bits-per-key due to memory pressure, reallocating BF size across different files with maintaining the overall memory budget unchanged becomes more essential to enhance the overall query throughput.

Challenges for Existing BF Reallocation in LSM trees. Existing BF reallocation models [15, 18, 20] cannot fully exploit the available memory when *(A) the LSM tree is heavily updated and thus evolves rapidly*, *(B) the LSM tree has an imperfect shape* (e.g., due to iterative compactions), or *(C) the workload has a high access skew*. To illustrate these limitations, Figure 1 highlights the first two challenges using a mixed workload comprising 40*M* inserts, 40*M* empty point queries, and 20*M* updates. In addition, Figure 2 demonstrates the conceptual optimization headroom for BF reallocation when more workload characteristics are considered. We explain in detail why existing approaches are suboptimal in Section 2.

Inaccurate Access Statistics. One of the key reasons that prior work could not offer a true workload-aware memory allocation model is that such a model would require *complete and precise-enough* knowledge of the access statistics for each BF within the LSM tree. While it is fair to assume that past workload is a good predictor for the future, capturing statistics in evolving LSM trees is still hard. Maintaining query counters and propagating statistics at compaction time by averaging [41], overlooks the evolving access patterns during flushes and compactions (§5). This leads to a significant discrepancy between the tracked accesses and the ground truth. The challenge is to estimate the accesses of a newly generated file after a compaction. Although such a file has never been physically accessed, to allocate memory for it properly, we need an accurate estimate of the accesses it would have received if it had been generated earlier. Such a mechanism can facilitate other workload-specific tuning decisions in LSM trees (e.g., prefetching files with high estimated access frequency into block cache), in addition to what we propose in this paper.

Mnemosyne: Dynamic BF Memory Allocation for Evolving LSM trees. To address the aforementioned challenges, we build Mnemosyne (named after the ancient Greek goddess of

memory), a BF reallocation framework for *evolving* LSM trees, which can dynamically determine the bits-per-key for newly generated files without any prior workload knowledge. Furthermore, to support workload-aware tuning, we use a more faithful cost model for BFs with constrained memory, and we also develop a statistics tracking mechanism called Merlin (named after a wizard who could foresee the outcome of battles in Celtic mythology), using which we build Mnemosyne$^+$, a system that offers workload-aware memory allocation.

Merlin captures the historical workload characteristics with a sliding window and estimates the access frequency for each file with high accuracy. The estimated statistics are used to instantiate our cost model with the current LSM tree structure to decide the bits-per-key for newly generated files during each flush and compaction. This approach captures the exact tree structure (whether it is leveling, tiering, or hybrid), including how full each level is, thus, Mnemosyne$^+$ does not need to make any assumptions regarding the LSM tree shape. Mnemosyne$^+$ enforces the user-provided memory budget (average bits-per-key) without assuming a perfectly shaped tree or a predefined tree structure. Overall, Mnemosyne$^+$ is a practical approach for BF reallocation in evolving LSM trees to reduce unnecessary I/Os compared to the state of the art.

Contributions. In summary, our contributions are as follows:

- We build a generalized cost model for bits-per-key allocation, and propose an efficient solver for it. To address Challenges (B) and (C), our model considers workload skew and empty queries per file without assuming a perfectly shaped LSM tree (§3).
- We design a dynamic BF reallocation framework, Mnemosyne, for evolving LSM trees. To address Challenge (A), our framework does not rely on any prior knowledge of the workload and reallocates memory during flushes and compactions (§4).
- We find that averaging the access counters at compaction time to estimate access frequency leads to a large discrepancy between the estimated frequency and the ground truth (§5.1).
- We design Merlin, a new access frequency estimation mechanism that considers the impact on the access frequency from continuous flushes and compactions in LSM trees. Compared to naïve tracking, Merlin drastically reduces the estimation error, thus also addressing the challenge of statistics estimation (§5.2).
- We build Mnemosyne$^+$ on top of RocksDB by combining our cost model, Merlin and Mnemosyne, and show that Mnemosyne$^+$ outperforms RocksDB by up to 2× with small bits-per-key (§6).
- We make our code available for exploration and reproducibility[1].

2 Background

In this section, we review the LSM tree background [45, 50]. We summarize the most commonly used notations in Table 1.

Log-Structured Merge tree. The LSM tree is a classical *out-of-place* key-value data structure. To support fast writes, LSM trees buffer all inserts (including updates and deletes) into a memory buffer with a predefined capacity. When the buffer fills up, all the entries in the write buffer are flushed to secondary storage as an immutable sorted *run*. As more runs accumulate, a *compaction* is triggered, which essentially sort-merges smaller runs to form a larger sorted run. Since runs may have overlapping key ranges, a compaction can effectively restrict the number of runs that a query searches, and it also discards obsolete entries during this process. Specifically, all sorted runs are organized in a tree-like structure where each level has exponentially larger capacity with a predefined size ratio T. A compaction is triggered for a level whenever its accumulated data size reaches the predefined capacity. Classically, LSM trees are constructed in a *top-down* manner, i.e., Level 0 files compacted into Level 1, Level 1 into Level 2, and so forth. In contrast, RocksDB

[1]https://github.com/BU-DiSC/Mnemosyne

Table 1. Summary of our notation.

Notation	Definitions (Explanations)
T	the size ratio in an LSM tree
L	the number of levels for an LSM tree
M	the total memory budget in bits for all the Bloom Filters
F	the total number of files in an LSM tree
$\epsilon_i(\epsilon_j)$	the false positive rate for File i (Level j)[2]
$n_i(n_j)$	the number of entries in File i (Level j)
$\text{bpk}_i(\text{bpk}_j)$	bits-per-key for File i (Level j)
z_i	the number of zero-result (empty) queries for File i
x_i	the number of existing (non-empty) queries for File i
q_i	the number of queries for the ith file ($q_i = z_i + x_i$)
m_i	the Bloom Filter size of File i ($m_i = n_i \cdot \text{bpk}_i$)
I_j	the set of file IDs in level j in a compaction
$\mathcal{Q}$	a workload that only contains point queries
$\mathcal{I}$	a workload that only contains ingestion
Z	the proportion of zero-result queries in $\mathcal{Q}$

supports a *bottom-up* construction approach [24] to reduce space amplification, allowing files at Level 0 to be compacted directly into the last level (logical Level 1 in this case).

Compaction Policy. The compaction policy in an LSM tree specifies when the compaction process is triggered and how it is executed. There have been extensive studies focusing on tuning a compaction policy to leverage various trade-offs among the costs associated with reads, writes, and space [18, 19, 21, 53, 56, 64, 67]. One common tuning guideline is that the *leveling* compaction policy is optimized for reads, while the *tiering* policy is optimized for writes. There are also hybrid strategies [19, 56] that allow different levels to have different compaction policies. In a leveling architecture, there could be a significant latency spike when a compaction involves two entire levels, especially for deeper (larger) levels. To amortize the compaction cost, real systems like LevelDB, RocksDB, and Pebble typically employ *partial compactions*, where multiple immutable *Sorted-String Table (SST)* files form a single sorted run, and one or only a few files are selected for a compaction to the next level whenever it is triggered.

Point Queries in LSM trees. A point lookup begins by querying the write buffer and then traverses the LSM tree from the shallowest level to the deepest level until it finds the first match. After finding the first match, the point lookup does not need to access deeper levels because entries in older levels (runs) are superseded. As such, a *zero-result* query (i.e., when the target key does not exist in the database, also called *empty* query) may result in large #I/Os, as it examines all levels (runs). To reduce the lookup cost, LSM trees maintain the key range of every SST file in memory (also called *file-wise fence pointers*). These fence pointers ensure that at most one file is accessed when querying a sorted run. Similarly, since entries in a file are sorted and stored by contiguous data blocks, *block-wise fence pointers* (i.e., min-max keys per page, stored in dedicated index blocks) are created for each SST file to ensure that at most one data block is accessed when querying the file (assuming all index blocks are prefetched into the *block cache*).

Bloom Filters. LSM trees use Bloom Filters (BFs) [8, 58] to accelerate point lookups. Similar to the index block, each SST file has a filter block that stores the associated BF (filter blocks are often prefetched in the block cache). The BF is queried before accessing any data blocks to determine if they can be skipped. For positive BF results, the search proceeds to access the index block and

[2]We use i to index the file, and j to index the level throughout this paper.

then the data block. However, the BF is a probabilistic membership test data structure that can yield false positives. For each key-value pair in an SST file, the BF encodes the key into k indexes (using k independent hash functions) in an m-bit vector and sets the corresponding bits. When the number of key-value pairs n is large, and the vector size m is small, hash collisions may frequently occur, leading to a high false positive rate (FPR, also noted by ϵ). Formally, by selecting the optimal $k_{opt} = \lceil m/n \cdot \ln 2 \rceil$, ϵ can be obtained as follows:

$$\epsilon = e^{-(\ln 2)^2 \cdot \frac{m}{n}}, \tag{1}$$

where m/n is also known as bits-per-key, short as bpk.

State-of-the-art BF Reallocation Models for Empty Point Queries Assumes a Static LSM tree. Monkey [15] builds a cost model for empty point lookups, noted by $Cost(\{\epsilon_j\})$, to obtain the bpk assignment per level j. In the leveling case of Monkey, the expected number of data blocks accessed by an empty query is the sum of the FPRs at each level, and thus $Cost(\{\epsilon_j\}) = \sum_{j=1}^{L} \epsilon_j$, where L is the number of levels and ϵ_j is the false positive rate in level j. Taking into account Eq. (1) and the memory budget M, Monkey restricts $-\sum_{j=1}^{L} n_j \cdot \ln \epsilon_j \leq (\ln 2)^2 \cdot M$, where n_j represents the number of entries in level j, and M represents the total number of bits for all filters. The number of entries for each level, $\{n_j\}$, is assumed to be fixed, and thus, Monkey works for a *static* LSM tree. When the average bits-per-key for the entire LSM tree is configured as bpk, we have $M = \text{bpk} \cdot \sum_{j=1}^{L} n_j$. Since Monkey assumes the same number of sorted runs per level (either leveling or tiering), we can only apply the Monkey model to Level 1 and onward in RocksDB since files in Level 0 are unsorted and can accumulate even beyond the maximum number of files in Level 0. As such, all files in Level 0 have the user-defined bpk and files in deeper level have very small bpk, which explains why the Monkey model does not have any advantage over the default RocksDB in Figure 1b. In contrast to Monkey, Dostoevsky [18] allows a different number of sorted runs per level, and the overall cost is now $Cost(\{\epsilon_j\}) = \sum_{j=0}^{L} r_j \cdot \epsilon_j$ where r_j is the number of predefined sorted runs in level j. However, Dostoevsky also assumes a perfectly shaped LSM tree, where the number of key-value pairs per level increases by size ratio T. An optimized version of Monkey, termed in this paper IdealMonkey [16], allows the user to specify an arbitrary number of key-value pairs per level, thus supporting imperfect shapes. Since we no longer have closed-form solutions for arbitrary $\{n_j\}$, IdealMonkey uses a gradient decent algorithm with complexity $O(L^2 \cdot \log M)$ to obtain the optimal $\{\epsilon_j\}$. As IdealMonkey does not restrict the LSM tree shape, it can also act as a more general version of Dostoevsky since we can treat each sorted run as a separate level, and re-apply the IdealMonkey model.

Limitations of State of the Art. We now elaborate on the limitations of Monkey and IdealMonkey.

- (L1) Monkey assumes that the number of levels L is given in advance, or L can be estimated using the given number of key-value pairs to be inserted
- (L2) IdealMonkey relies on the exact number of entries per level.

Note that here L refers to the number of non-empty levels that depends on the input workload, instead of the maximum number of levels which can be estimated by storage capacity. Both are *impractical* when deploying a real-world key-value store engine, since we generally do not have any prior knowledge of the workload. Further, even if we know the approximate workload scale in advance and we can estimate L for Monkey, the overall throughput of Monkey could be worse than a standard RocksDB due to inaccurate LSM-tree shape estimation. Figure 1b shows how the observed throughput evolves as we run a mixed read-write workload for uniform bits-per-key allocation and Monkey approaches (one for top-down LSM tree construction and the other for bottom-up construction). Figure 1c shows how the memory allocated for BFs evolves during execution. We observe that Monkey with top-down LSM-tree construction outperforms RocksDB, which, however,

comes at the cost of overusing the user-defined bits-per-key since data first reside in the shallow levels using higher bits-per-key. On the other hand, when Monkey respects the memory budget in the bottom-up LSM-tree construction, this comes with a significant performance cost because files in Level 0 are compacted first into the last level which has the smallest bits-per-key in Monkey. Note that, although RocksDB strictly aligns with the memory budget, its throughput is not ideal, so, overall, there is room for improvement, both in terms of performance and memory allocation.

(L3) Monkey uses a closed-form solution that assumes that a perfect LSM tree shape, i.e., the number of keys per level grows *exactly* exponentially with a constant size ratio T.

The *perfect shape* assumption does not always hold for evolving LSM trees. Classical tiering and leveling compact the whole level into the next one, which means that every compaction leaves an empty level (a *hole* in the tree). Even for partial compaction [56, 63], multiple files can be selected to be compacted together into the next level, akin to the expansion mechanism in RocksDB [54], and partition-based compactions for deeper levels in Spooky [21]. When more than one file is picked, the size ratio between two adjacent levels changes drastically. Note that compactions are not necessarily triggered by level capacity [56], instead, they can also be triggered by tombstone-related metadata [34, 55]. Finally, the key-value entry size may vary across levels, thus preventing the number of entries from growing exponentially even with a constant size ratio. Note that the size ratio in most LSM tree implementations specifies the ratio of the *data size* between two adjacent levels instead of the ratio of the *number of entries* that idealistic models typically assume.

(L4) Both Monkey and IdealMonkey assume that point queries are perfectly uniform in the key domain, and thus, files in the same level should have the *same* bpk.

Real-world workloads exhibit skewed access patterns [7, 10, 14], in which case the best bpk assignment could also differ among files within a level (or a sorted run). Further, memory allocation in prior work considers non-empty queries opportunistically [16], while a more accurate cost model has been recently proposed for holistic tuning [35]. In practice, if we know that the queries targeting a file will be exclusively non-empty, no BF is needed for this file since no queries can be skipped (i.e., bits-per-key should be 0), and any hashing for the BF will only negatively impact query latency due to its CPU cost [70]. Recent work on optimizing collections of BFs in the presence of query skew [41, 48, 59] focuses on the total number of queries and also assumes static data. Overall, the challenge to exploit skew, focusing on empty queries in evolving LSM trees, remains a key challenge. Figure 2 shows that there is a large improvement headroom if we can ideally allocate bits-per-key for each file at compaction time.

3 Workload-Aware bpk Allocation for A Static LSM Tree

We now study the optimal workload-aware bpk allocation model for a *static* LSM tree. If point queries are skewed, then the optimal design should have files at the same level with different bpk allocation, with the exact numbers depending on the number of empty queries that access each file.

In this section, we assume that we know all read statistics in advance, which include the number of queries per file $\{q_i\}$, the number of zero-result queries per file $\{z_i\}$, and the number of entries per file $\{n_i\}$. Note that, in practice, it is **unrealistic** to know the **exact** query statistics when building filters during flushes or compactions, since the associated files are newly created and have not received any queries. Thus, this section provides a theoretically optimal bpk assignment that provides a **headroom** of improvement for our practical algorithm.

3.1 Problem Definition

We aim to minimize the number of data blocks that are unnecessarily accessed, which is equivalent to minimizing the total number of data blocks accessed, since the number of necessary data blocks

accessed is constant. Note that all unnecessary accesses to a file i are triggered by the corresponding false positives (ϵ_i) for empty queries (z_i). Thus, the cost function is:

$$Cost(\{\epsilon_i\}) = \sum_{i=1}^{F} z_i \cdot \epsilon_i, \tag{2}$$

where F is the number of total BFs (files). When a file does not receive empty queries ($z_i = 0$), it does not affect the objective function and would also not benefit from a BF. In this case, we manually assign $\text{bpk}_i = 0$. For the remainder of this section, we assume that $z_i > 0$ for every file. In addition, the memory constraint specifies that the total BF size should be no more than M, where $M = \text{bpk} \cdot \sum_{i=1}^{F} n_i$ and bpk is the user-defined average bits-per-key. Further, since bpk is defined for all the entries and we do not create BFs for files with $z_i = 0$, we effectively reallocate the BF budget for those files to files with $z_i > 0$. We formalize the problem statement as follows:

$$\begin{aligned} \min_{\epsilon_1,\epsilon_2,\ldots,\epsilon_F} \quad & Cost(\{\epsilon_i\}) \\ \text{subject to} \quad & -\frac{1}{(\ln 2)^2} \cdot \sum_{i=1}^{F} n_i \cdot \ln \epsilon_i \leq M \\ & 0 < \epsilon_i \leq 1, \forall i \in [F] \end{aligned} \tag{3}$$

Note that we now want to find F false positive rates (one per file), as opposed to L false positive rates in Monkey/IdealMonkey (one per level or one per sorted run). The new problem space is significantly larger since an LSM tree typically has less than seven levels but thousands of files. We will revisit this when we discuss the efficiency of solving the optimization problem. Similarly to previous work [15, 16, 59], we use a *Lagrangian* multiplier for the memory constraint to obtain:

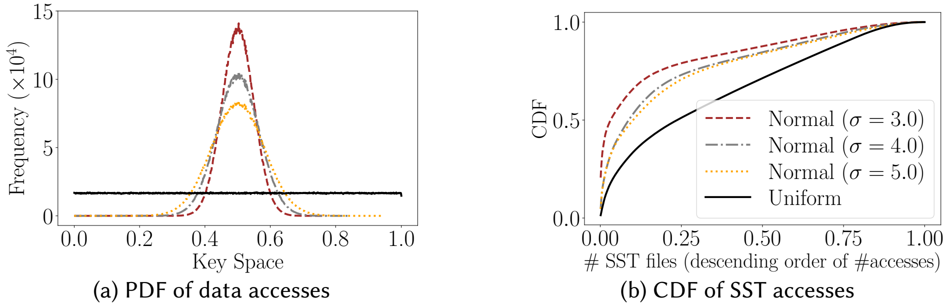
$$\epsilon_i = \frac{n_i}{z_i} \cdot e^C \text{ where } C = -\frac{M \cdot (\ln 2)^2 + \sum_{i=1}^{F} n_i \cdot \ln \frac{n_i}{z_i}}{\sum_{i=1}^{F} n_i} \tag{4}$$

As the constraint $\epsilon_i \leq 1, \forall i \in [F]$ is not enforced in Eq. (4), we may get files with $\epsilon_i > 1$. Since the false positive of a BF can only be between 0 and 1, we need to handle these cases specially. This may happen when the overall cost does not benefit from a specific file having BF despite having some empty queries ($z_i > 0$). In this case, we assign $\epsilon_i = 1$ and reallocate any memory of this file to other files. We then recalculate Eq. (4) to obtain the new bpk assignment. We may execute this process many times until we have no $\epsilon_i > 1$ after applying Eq. (4). To do this efficiently, we develop an algorithm that identifies which files have $\epsilon_i \geq 1$ in the optimal solution, assign to them $\text{bpk}_i = 0$, and use Eq. (4) for the remaining files. We discuss how to develop this algorithm in the next section.

3.2 Headroom for Improvement

To investigate the headroom for improvement, we conduct a micro-benchmark that compares the number of unnecessarily accessed data blocks among the default BF allocation strategy in RocksDB (the same bpk for all BFs), IdealMonkey (that allows imperfect LSM shapes), and the optimal one obtained by our model in Eq. (3).

Experimental Methodology. We populate an LSM tree with $4M$ 512-byte entries (2GB) using the default BF allocation strategy in RocksDB (all the BFs have the same bpk). We set the size ratio $T = 4$, the data block size as 16KB, and the target file size as 16MB, which produces a 3-level LSM tree. Then we allocate a 64MB block cache, which ensures all the filters and indexes fit in the cache except the data blocks, and we execute a read workload of $15M$ point queries. We run our experiments with varying the proportion of empty queries (noted by Z) and varying the query distribution. In Figure 3, we plot different query distributions in the key domain and we also show



(a) PDF of data accesses

(b) CDF of SST accesses

Fig. 3. Different distributions for 15M point queries. Note that, even with uniform access patterns in the data domain, the access frequency of the SST files is not uniform.

how the read access patterns on the domain are translated into access requests on various SST files of the LSM tree. In the following experiment with Normal distribution, we choose $\sigma = 5.0$ by default. For a fixed Z and a query distribution, we can run the query workload and obtain the statistics for each SST file (including the number of entries n_i, the number of point reads q_i, and the number of existing point reads x_i). Next we rebuild each BF in LSM tree with specifying different bpk_i per file, where we obtain the bpk_i with the gradient decent algorithm in IdealMonkey and with solving Eq. (3) for our model. To compare different bpk allocation strategies, we execute the same query workload again and measure and the number of accessed data blocks. Finally, we report the average number of unnecessarily accessed data blocks (which is the difference between the number of accessed data blocks and the number of existing point queries). We also repeat the above procedure by varying overall bpk from 4 to 8. Note that we do not use the number of overall false positives as a measure here, although this metric is closely related to unnecessarily accessed data blocks. False positives occur when BFs incorrectly indicate the presence of a key, resulting in unnecessary data block accesses. However, in scenarios where IdealMonkey assigns no BFs to certain files in lower levels for a very small user-specified bpk, accessing these files bypasses BFs entirely. Consequently, while there are no false positives in such cases, empty point queries may still lead to unnecessarily accessed data blocks, which means that our metric, the number of unnecessarily accessed data blocks, can more accurately capture all wasted I/Os.

Observations. As shown in Figure 4, when we fix the distribution and vary the proportion of empty queries Z, our model significantly reduces the number of unnecessary block accesses compared to IdealMonkey. The benefit is especially pronounced when there are fewer empty queries (i.e., as Z approaches 0), as indicated by the larger gap between the blue and red bars in Figure 4. In addition, when we fix bpk and compare the impact of query distribution, we also observe that our model can further reduce unnecessary block accesses for a more skewed distribution (normal vs. uniform). For example, the optimal (red bar) when $Z = 0.0$ in Figure 4a is two order of magnitude higher than the one in Figure 4b while RocksDB and IdealMonkey (black and blue bars) remain unchanged regardless of the distribution. These observations confirm our expectations from Figure 2. We conclude that a file-based skew-aware model, as defined in Eq. (3), should be prioritized over IdealMonkey if the workload access pattern per file is known.

4 BF Reallocation in An Evolving Tree

In this section, we first discuss how we efficiently solve Eq. (3), and, based on our solution, we then introduce Mnemosyne, which can dynamically reallocate BFs in an evolving tree without excessively overusing the user-defined average bits per key.

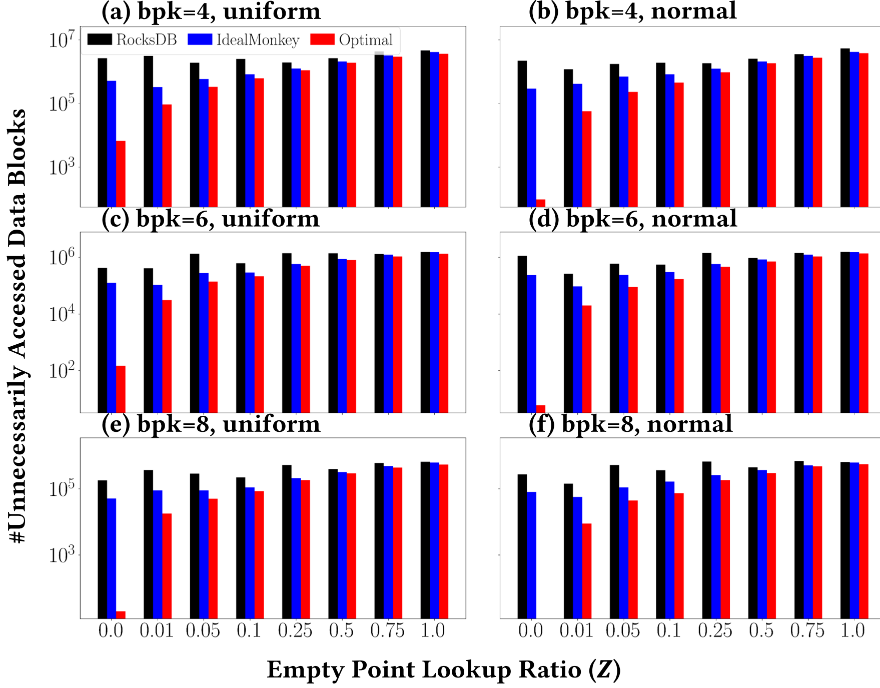


Fig. 4. Comparing the optimal model with RocksDB and IdealMonkey, the average number of unnecessarily accessed data blocks is much fewer for workloads that have all non-empty point queries or exhibit skew access pattern. Z is the proportion of zero-result queries ($Z = 0$ means all queries are existing queries and $Z = 1$ means all are zero-result queries).

4.1 Ordered Property

We observe that we cannot directly apply Eq. (4) to Eq. (3) as files with $\text{bpk}_i = 0$ ($\epsilon_i = 1$) may exist. To identify them, we use the ORDERED THEOREM that we present and prove in this section. The core result is that we can use basic file statistics, z_i and n_i, to identify whether the solution in Eq. (4) is applicable to a file. Specifically, such files are contiguous if we sort them according to the z_i/n_i ratio. Using this result, we derive an algorithm with $O(F \cdot \log F)$ complexity that identifies which files have $\epsilon_i = 1$ in the optimal solution.

THEOREM 4.1 (ORDERED THEOREM). *In the optimal solution of the objective function in Eq.* (3), *the value* $\frac{z_i}{n_i}$ *of files with* $\epsilon_i = 1$ *should be all smaller than the one of any file with* $\epsilon_i < 1$.

PROOF. We prove Theorem 4.1 using duality. We note that the optimization problem is a *convex* problem, as proved by Truncated BF [48]. We now use Slater's condition to show that strong duality holds. To do this, we define the *Lagrangian* function $\mathcal{L}$ as follows:

$$\mathcal{L}(\epsilon_1, \epsilon_2, ..., \epsilon_F, \lambda, \nu_1, \nu_2, ..., \nu_F) = Cost(\{\epsilon_i\}) + \lambda \cdot \left(-\frac{1}{(\ln 2)^2} \cdot \sum_{i=1}^{F} n_i \ln \epsilon_i - M \right) + \sum_{i=1}^{F} \nu_i \cdot (\epsilon_i - 1), \quad (5)$$

where λ and $\{\nu_i\}$ are *Lagrange multipliers* we introduce for the constraints ($\lambda \geq 0$ and $\nu_i \geq 0\ \forall i \in [F]$). We do not consider the constraint $\epsilon_i > 0$ because it is implied by $\ln \epsilon_i$ in the memory constraint.

We also define the dual function $g(\lambda, \nu_1, ..., \nu_F)$ as follows:

$$g(\lambda, \nu_1, ..., \nu_F) = \inf_{\epsilon_1, \epsilon_2, ..., \epsilon_F} \mathcal{L}(\epsilon_1, \epsilon_2, ..., \epsilon_F, \lambda, \nu_1, \nu_2, ..., \nu_F) \tag{6}$$

As long as the user-specified average bpk > 0 and the LSM tree is not empty, we always have $M > 0$ since $M = \text{bpk} \cdot \sum_{i=1}^{F} n_i$. Otherwise, we can set all $\text{bpk}_i = 0$ if bpk $= 0$. To satisfy Slater's condition, we need to find a feasible point that makes the convex constraint to strictly hold. We can achieve this by setting all $\text{bpk}_i = 0$. Synthesizing the above discussion and Slater's condition, the optimization problem in Eq. (3) has *strong duality*. Therefore, the optimal solution $\epsilon_1, \epsilon_2, ..., \epsilon_F, \nu_1, \nu_2, ..., \nu_F$ of Eq. (6) should satisfy:

$$\begin{cases} \nu_i > 0, & \text{if } \epsilon_i = 1 \\ \nu_i = 0, & \text{otherwise (i.e., } \epsilon_i < 1) \end{cases} \tag{7}$$

Note that, we should always have $\lambda > 0$ and $-\frac{1}{(\ln 2)^2} \cdot \sum_{i=1}^{F} \ln \epsilon_i = M$ to minimize $Cost(\{\epsilon_i\})$. Furthermore, to minimize $\mathcal{L}$, we can take its derivative with respect to ϵ_i, and set it equal to 0:

$$z_i - \frac{\lambda \cdot n_i}{(\ln 2)^2 \cdot \epsilon_i} + \nu_i = 0 \tag{8}$$

By solving the equation, we have:

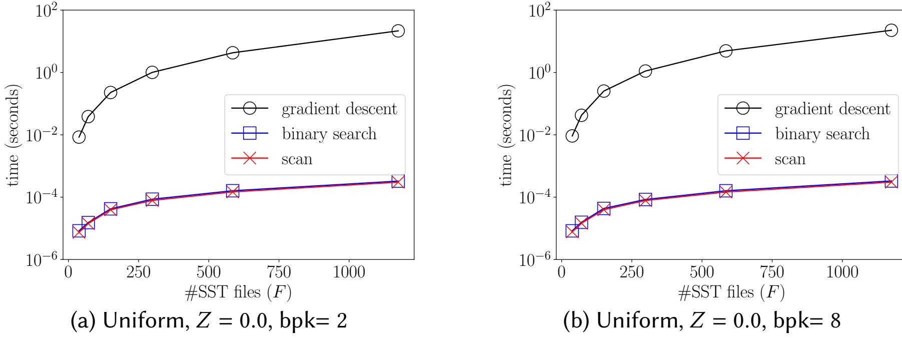
$$\begin{cases} C > \frac{z_i}{n_i} & if\, \epsilon_i = 1 \\ C < \frac{z_i}{n_i} & if\, \epsilon_i < 1 \end{cases}, \tag{9}$$

where $C = \lambda/(\ln 2)^2$, obtained by Eq. (4) without considering BFs with $\epsilon_i = 1$, that is, we will not consider all F files in $\sum_{i=1}^{F} n_i$ and $\sum_{i=1}^{F} n_i \cdot \ln \frac{n_i}{z_i}$, but only those with $\epsilon_i < 1$. Proof completes. □

Connection with Monkey. In Monkey, shallower levels have larger bpk_j than deeper levels, which is consistent with Theorem 4.1 when the workload is uniform and only has empty queries. Specifically, when all the files have the same number of entries (i.e., n_i is constant), our model always assigns more bpk to files with large z_i, because shallower levels have fewer files but approximately the same key range, compared to the deeper levels. However, if the workload only contains existing (non-empty) queries, all the files in the deepest level then have $z_i = 0$, and thus $\text{bpk}_i = 0$, while Monkey wastes $1/L$ BF memory budget in the deepest level where L is the number of levels. As such, when the overall bpk is small, bpk_j for shallower levels in Monkey can be much smaller than bpk_i derived from our model, and thus Monkey may result in more unnecessary data block accesses in those levels, as shown in Figure 2. Besides, when the workload exhibits higher skew, files in the same level could have significantly different z_i. Since Monkey assumes files in the same level have the same z_i, the bpk assignment in Monkey can deviate further away from the optimal one.

Sort-And-Search. We now present Algorithm 1, our sort-and-search algorithm based on Theorem 4.1. We first sort all the files according to $\frac{z_i}{n_i}$ in descending order, and then we do reversely linear searching to find the maximum C so that the maximum false positive rate ϵ_{max} is smaller than 1. During this process, we also filter out files with small $\frac{z_i}{n_i}$ (that is, we do not construct BFs for these files, i.e., $\text{bpk}_i = 0$). After that, we calculate bpk using z_i, n_i and C for the remaining files. The overall complexity $O(F \cdot \log F)$ is dominated by sorting. In fact, we can also use binary search after sorting to further accelerate the algorithm. Since the complexity is dominated by the sorting, we do not observe that binary search shows significant improvements over linear scan (see a micro-benchmark in Figure 5), we thus only implement the linear scan algorithm in Mnemosyne.

Implementation. In practical systems such as RocksDB, when bpk ≤ 1, the assigned bpk is actually `round(bpk)`. For example, if bpk < 0.5, it rounds down to 0. This implementation restricts the minimum bpk_i to be 1 if $\text{bpk}_i > 0$. To achieve this, we replace the condition ($\bullet > 0$) in line 10



(a) Uniform, $Z = 0.0$, bpk= 2

(b) Uniform, $Z = 0.0$, bpk= 8

Fig. 5. Our algorithms (both binary search and scan) are much faster than gradient descent.

of Algorithm 1 with $\bullet > e^{-(\ln 2)^2}$ (the false positive rate $\epsilon = e^{-(\ln 2)^2}$ when bpk = 1), and then we naturally have $\text{bpk}_i \geq 1$ if $\text{bpk}_i > 0$ in the optimal solution.

Algorithm 1: SORT-AND-SEARCH($\{z_i\}, \{n_i\}, F, M$)

1. $\mathcal{Z} \leftarrow \{z_i\}$
2. $\mathcal{N} \leftarrow \{n_i\}$
3. $result \leftarrow \{\}$
4. Initialize a pair vector V so that $V[i] = (\frac{z_i}{n_i}, i)$
5. Sort V according to $\frac{z_i}{n_i}$ in a descending order
6. $S_1 \leftarrow \sum_{i=1}^{F} n_i$
7. $S_2 \leftarrow \sum_{i=1}^{F} n_i \cdot \ln \frac{n_i}{z_i}$
8. $C \leftarrow \frac{-M\cdot(\ln 2)^2 - S_2}{S_1}$
9. **for** $i^* \leftarrow n$ *to* 1 **do**
10.   **if** $C - \ln V[i^*].first > 0$ **then**
11. $i_{tmp} \leftarrow V[i^*].second$
12. $result[i_{tmp}] \leftarrow 0$
13. $S_1 = S_1 - \mathcal{N}[i_{tmp}]$
14. $S_2 = S_2 - \mathcal{N}[i_{tmp}] \cdot \ln \frac{N[i_{tmp}]}{\mathcal{Z}[i_{tmp}]}$
15. $C \leftarrow \frac{-M\cdot(\ln 2)^2 - S_2}{S_1}$
16.   **else**
17.   break;
18. **for** $i \leftarrow i^*$ *to* 1 **do**
19. $i_{tmp} \leftarrow V[i].second$
20. $result[i_{tmp}] = -\frac{1}{(\ln 2)^2} \cdot \left(\ln \frac{\mathcal{N}[i_{tmp}]}{\mathcal{Z}[i_{tmp}]} + C\right)$
21. Return *result*

Efficiency of the Optimization Solver. We also examine the difference in the execution time between the gradient descent algorithm (proposed by Monkey) and our sort-and-search algorithms (including both linear-scan and binary-search versions). We set the SST file size as 32MB and the

size ratio as 4, and vary bpk, the workload characteristics (including both Z and the distribution), and also the number of SST files (F). Since most experimental results have similar patterns when fixing F, we only present a subset of results ($Z = 0.0$, bpk = 2 and $Z = 0.0$, bpk = 8) in Figure 5. As shown in Figures 5a and 5b, the gradient descent approach is slower than our algorithms by $2 \sim 4$ orders of magnitude. Note that the execution time to find the optimal solution shall not exceed one second because this searching algorithm is executed during each compaction as mentioned later and the median compaction latency is just one second [56] when populating a 10GB LSM tree with a 4000 IOPS-provisioned SSD. Since our optimization solvers only take around 0.1 *ms* when $F \approx 1K$, our algorithms are more practical to be deployed during compactions. However, obtaining the input of Algorithm 1 is not trivial since it has to traverse the metadata of all files. This process can result in around 2% throughput reduction in write-intensive workloads.

4.2 Mnemosyne

Since we do not yet have the estimated access statistics, we assume that all files within the same sorted run are accessed uniformly, resulting in the same bpk_i for each file within the same sorted run. Based on this assumption, we can reformulate Eq. (3) by optimizing ϵ_j and setting $z_j = 1$ for all sorted runs, where ϵ_j represents the false positive rate for each sorted run. As Algorithm 1 remains applicable under the adjusted model, we integrate the BF reallocation algorithm into RocksDB to construct *Mnemosyne*.

BF Reconstruction During Flushes and Compactions. As all SST files in LSM trees are immutable, we cannot directly apply Algorithm 1 to rebuild BFs. Therefore, BF reallocation is only applied for files generated by flushes and compactions and all other BFs remain unchanged. Whenever a flush or compaction is triggered, we estimate the future LSM tree shape based on its current structure. The shape is represented by the list $\{n_j\}$ of length L', where each element records the number of entries in a sorted run, and L' denotes the total number of sorted runs. In RocksDB, Level 0 can contain multiple sorted runs, and starting from Level 1, each level forms a separate sorted run. Note that while users can configure the number of files in Level 0 to trigger compactions, this threshold is not strictly enforced due to *write stalls*, which occur when compaction from Level 0 to Level 1 is blocked by ongoing compaction between Level 1 and Level 2. Write stalls thus allow files to accumulate in Level 0 beyond the configured limit. In such cases, there might be unexpectedly high overlap between files. Given the current LSM tree shape $\{n_j\}$ and a newly flushed file with n^{new} entries, the updated LSM tree shape becomes $\{n^{new}\} \cup \{n_j\}$. For compaction, we adjust the number of entries between adjacent levels accordingly. Specifically, when a compaction moves a set of files containing n' entries from one sorted run j_1 to another sorted run j_2, we update the LSM tree shape by setting $n_{j_1} = n_{j_1} - n'$ and $n_{j_2} = n_{j_2} + n'$. This formulation applies to normal compactions between levels, user-triggered compactions, and intra-L0 compactions. Once the estimated LSM tree shape $\{n_j\}$ is determined, we compute $M = \text{bpk} \cdot \sum_{i=1}^{L'} n_j$, and apply Algorithm 1 with the input $(\{1\}, \{n_j\}, L', M)$ to obtain bpk^{new} for newly generated files.

Bits-per-key Adjustment. If $\text{bpk}_j \in [0.5, 1)$, we round it up to 1. Otherwise, we set it to 0, as is already done in RocksDB. Knowing the number of entries n^{new} in the upcoming new file, we can compute bpk^{new} using Algorithm 1 and estimate the average bpk for the entire LSM tree if bpk^{new} is applied. Note that we allow a BF to be skipped during a point query if Algorithm 1 says the associated bpk is 0, even if a BF exists. As such, we ignore this BF when computing the actual average bpk in Mnemosyne. After we get the average bpk, we then adjust bpk^{new} to ensure that the average bpk usage does not exceed the target by more than one bit-per-key. We allow bpk to be temporarily overused so that future compactions at lower levels do not underutilize bpk.

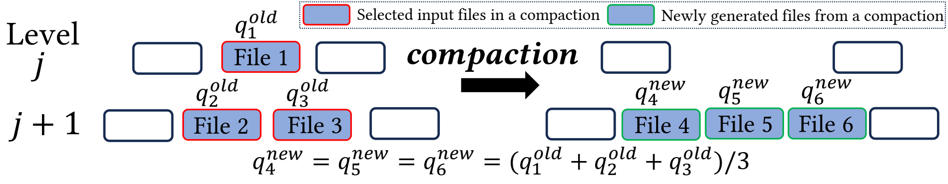


Fig. 6. The naïve strategy to inherit statistics during a compaction. $q_{i_1}^{old}$, $q_{i_2}^{new}$ denote the number of point queries for an input File i_1, and the estimated number of point queries for a newly generated File i_2, respectively.

Limited Block Cache. When the block cache is constrained, BF cache misses may occur because it is not always possible to fit all BFs into the available cache. Therefore, BFs should not be reconstructed during queries, as re-creating and subsequently evicting BFs would necessitate persisting them to disk, leading to excessive random disk writes. As such, in Mnemosyne, every BF construction is associated with a flush or a compaction, and BFs do not need to be re-created during restarts. Additionally, a limited block cache may require large BFs to be retrieved from disk, further exacerbating I/O overhead. To mitigate unnecessary reads spent on retrieving large BFs, we adopt the ModularBF [49], which decomposes large BFs into smaller BFs with different hash functions. In this way, only the first small BF needs to be loaded in the case of a cache miss. If the first small BF returns a positive result, the remaining components are then retrieved as needed, significantly reducing the number of disk I/Os. In our implementation, a ModularBF is built when $\text{bpk}^{new} > \text{bpk} + 1$ where bpk^{new} is the bits-per-key value assigned by Algorithm 1.

5 Access Estimation in Mnemosyne$^+$

Since Mnemosyne cannot perform workload-aware BF allocation due to the lack of access statistics, we now describe how to estimate, at runtime, the number of zero-result (empty) queries per file (z_i) in an LSM tree. This estimation is crucial for instantiating our workload-aware model in Eq. (3) and build Mnemosyne$^+$. Tracking the exact value of z_i per file would introduce significant maintenance overhead as the LSM tree continues to evolve with more data ingestion. Therefore, we focus on *estimating* z_i. To do this, we first estimate the number of queries q_i and the number of existing queries x_i, and compute z_i as $z_i = q_i - x_i$.

5.1 A Naïve Strategy

We first introduce a naïve strategy for estimating statistics. With this strategy, we maintain two counters for the number of point reads q_i and the number of existing point reads x_i, respectively, per file. At every compaction, we use the average counter of $\{q_i^{old}\}$ of all the input files as the estimated q_i^{new} for every newly generated file to avoid a cold start (a method similar to the one used by ElasticBF [41] to monitor $\{q_i\}$). In this example, shown in Figure 6, Files 1, 2, and 3 are the input files for a compaction, and Files 4, 5, and 6 are generated from this compaction. Using the naïve strategy, the number of point queries of newly generated files (i.e., q_4^{new}, q_5^{new}, q_6^{new}) are all estimated as $\sum_{i=1}^{3} q_i^{old}/3$. We can also use a similar strategy to estimate z_i^{new} or x_i^{new}. However, the naïve strategy has two problems that lead to inaccurate estimation, in which case the memory allocation would significantly deviate from the desired one. We detail these two problems below.

Problem 1: The Query Counters Are Initiated From an Inconsistent Starting Point. First, the naïve strategy does not consider a critical discrepancy, that is, $\{q_i\}$ are *not* initiated from the same starting counting point. If we simply count q_i per file, new SST files cannot record the workload statistics before they are generated, and thus $\{q_i\}$ of new files are usually smaller than older files. For example, in Figure 7, we consider a scenario where we start with the left-top LSM

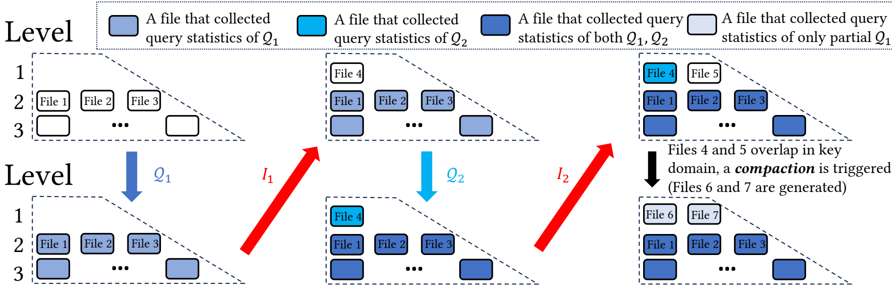


Fig. 7. An example that shows how the counter discrepancy could occur with the naïve approach. Darker color indicates more query statistics are collected when maintaining q_i.

tree, and sequentially execute workloads Q_1, I_1, Q_2, I_2 where Q_1, Q_2 are query workloads and I_1, I_2 are insertion workloads (I_1, I_2 both trigger a flush). In the final (right-bottom) state, the point query counters q_1, q_2, q_3 of older Files 1, 2, 3 are much larger than q_6, q_7 of newer Files 6, 7 because q_1, q_2, q_3 count for both workloads Q_1, Q_2, while q_6, q_7 only count Q_2. Besides, q_6, q_7 are inaccurate even if we only consider Q_2. In the naïve strategy, $q_6 = q_7 = (q_4 + q_5)/2$ (Files 6,7 are generated by compacting Files 4, 5). Since File 5 does not track any query workload ($q_5 = 0$), q_6 and q_7 deviate a lot from the actual ones if we replay Q_2. Ideally, q_6 and q_7 should take into account both Q_1 and Q_2 so that $\{q_i\}$ of all SST files have the same starting counting point.

Problem 2: Averaging Query Statistics at Compaction Leads to Overestimation. A query that does not find the desired key in level j proceeds to search in level $j+1$, which may lead to counting this query twice when calculating the average for newly generated files. Therefore, when we average the query statistics from the files in a compaction, the newly calculated statistics overestimate the number of queries by double counting the number of empty queries from level j. For example, consider a query workload Q on keys $\in [k_2, k_5]$ in Figure 8 (assuming $k_2 < k_3 < k_4 < k_5$). The empty queries that arrive before the compaction in File 1 – key $\in [k_2, k_5]$ – may have also accessed File 2 – if key also $\in [k_2, k_3]$ – or File 3 – if key also $\in [k_4, k_5]$ (only queries $\in (k_3, k_4)$ skip level $j+1$ because no files in level $j+1$ overlap with this range). We mark these two sets of double-counted queries using different shaded patterns in Figure 8. In the compaction, the averaging approach counts the queries in the shaded areas two times: once from File 1 and once from File 2 or File 3. After compaction, workload Q skips level j since no files overlap with $[k_2, k_5]$ in level j, and the query statistics are only collected once in Files 4, 5, and 6.

5.2 Merlin

We now introduce Merlin, a tracking and estimation mechanism for point queries per file. Although we only use Merlin for memory allocation in this paper, it can also benefit other decisions like caching and compaction priority. We summarize new notations of Merlin in Table 2.

5.2.1 Estimation of Zero-Result Point Queries. As discussed, we decompose the estimation of z_i into two steps: (1) estimating q_i and (2) estimating x_i. z_i is then approximated by $q_i - x_i$ (we explain why we need to estimate x_i instead of directly estimating z_i in §5.2.2).

Estimation of q_i. We maintain a global point query sequence number (noted by D^{global}) that represents the total number of point queries issued so far, and then we maintain a sliding window Ω_i per file, where Ω_i is implemented as a First-In-First-Out 64-length queue of D^{global} that accesses file i. In addition, for File i, we also maintain two counters q'_i and x'_i that store the number of queries and the number of non-empty queries before the first element of Ω_i (noted by $\Omega_i.front$), i.e., q'_i

Table 2. Notations in Merlin.

Notation	Description
D^{global}	the global point query counter
q'_i	a 64-bit point query counter of File i
x'_i	a 64-bit non-empty point query counter of File i
Ω_i	a queue for the most recent point queries of File i
x^{Ω_i}	a bitmap that represents if queried keys in Ω_i are found
β	the adaptation rate in statistics estimation

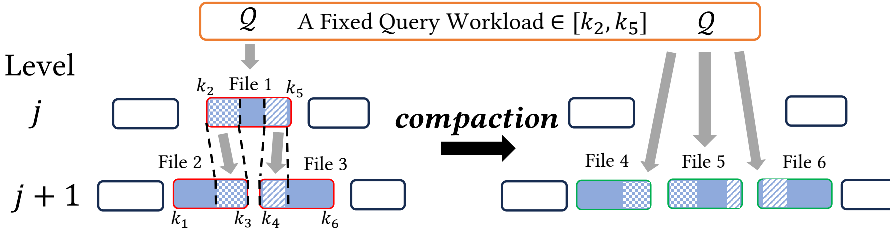


Fig. 8. In the naïve strategy, most empty queries counted in File 1 (z_1) are counted again in z_2 and z_3, which could result in inaccurate estimation for newly generated files, i.e., Files 4, 5, and 6 in the above example.

does not count the number of point queries in Ω_i. With these access statistics, q_i is estimated as:

$$q_i = D^{global} \Bigg/ \left(\beta \cdot \frac{\Omega_i.last - \Omega_i.first}{\|\Omega_i\| - 1} + (1-\beta) \cdot \frac{\Omega_i.first}{q'_i + 1}\right)$$

where $\Omega_i.last$ represents the last point query that accesses File i, $\|\Omega_i\|$ represents the length of Ω_i, and β represents the adaptation rate to control how aggressive the estimation adapts to most recent queries. In the above estimation model, $(\Omega_i.last - \Omega_i.first)/(\|\Omega_i\| - 1)$ is used to estimate the access interval in Ω_i, which is essentially a Maximum Likelihood Estimation that has been proposed by φ failure detector [32] and applied in Cassandra [4].

Estimation of x_i. Based on Ω_i and q'_i, we use a 64-length bitmap x^{Ω_i}, where each bit in x^{Ω_i} denotes whether the queried key in Ω_i is found in File i. We estimate x_i as follows:

$$x_i = \left(\beta \cdot \frac{\texttt{__builtin_popcount}(x^{\Omega_i})}{\|\Omega_i\|} + (1-\beta) \cdot \frac{x'_i}{q'_i}\right) \cdot q_i,$$

where `__builtin_popcount`(x^{Ω_i}) is a built-in function in gcc compiler that returns the number of non-zero bits for a given integer.

Workload Shifting. The workload characteristics (e.g., the proportion of empty queries, the distribution of queries) may change over time [7, 33, 46], and the old statistics thus become outdated if we only use the counter to estimate q_i and x_i. Merlin addresses this by maintaining query counters for older point queries while also using a sliding window to track the most recently accessed queries and the current ratio of existing queries. The adaptive ratio $\beta \in [0, 1]$ further enables Merlin to balance between preserving historical statistics and rapidly adapting to changing workload patterns, where a larger β indicates quicker adaptation to the most recent workload.

Concurrency. Although we can use `atomic` variables for all the counters when there are multiple querying threads, it is hard to maintain a thread-safe queue Ω_i for each file, because adding a lock can bring significant overhead. As such, we use a sampling-based approach for estimation. We start by randomly picking a thread to record the statistics (other threads cannot update Ω_i and q'_i, x'_i, x^{Ω_i}). In each query, the picked thread has low probability (i.e., 1%) to transfer the write

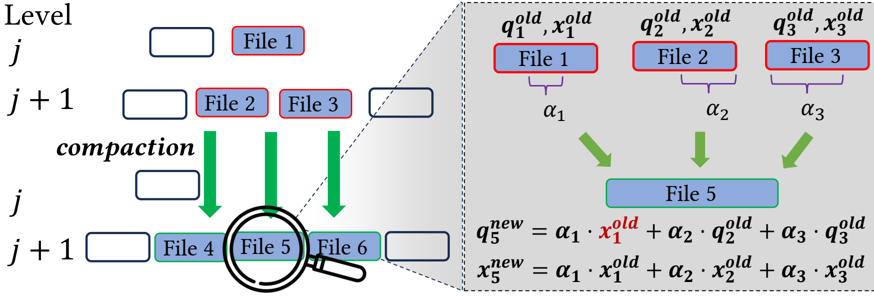


Fig. 9. In our inheritance model, File 5 is generated when merging files 1, 2, and 3. $\alpha_1, \alpha_2, \alpha_3 \in [0, 1]$ indicates the fraction of files 1, 2, and 3 that are used when generating file 5.

permission to another randomly picked thread. Note that D^{global} is implemented by an `atomic` counter, and thus we allow each thread to update it without conflicts.

Complexity. Maintaining such a queue for each SST file does not bring much CPU overhead during the point query, because evicting or inserting one element takes only $O(1)$ complexity. In addition, before accessing each file, we may need to estimate $z_i = q_i - x_i$ to determine if we can skip the BF (we discuss this in §5.2.3), but the estimation also only takes a constant time. In terms of space complexity, we need a 64-length queue where each element (the point query sequence number) is a 64-bit unsigned long variable, and we also have q'_i, x'_i, x_i^w, and bpk_i. Overall, we need $64 \cdot (64+4)/8 = 544$ extra bytes per file. For a 100GB database with file size of 32MB, the overall memory footprint only increases by 1.7MB.

5.2.2 Statistics Inheritance during Compactions. When new files are generated in compactions, we need an inheritance mechanism to estimate q_i^{new} of them to avoid cold start. To achieve this, the estimated q_i^{new}, x_i^{new} are assigned to q'_i, x'_i, all new files start with an empty Ω_i and x^{Ω_i}. We observe that only the non-empty queries are inherited during a compaction, according to the example shown in Figure 8 (only the queries for keys in File 1, are added in level $j+1$). In fact, the number of queries of newly generated files in compaction should mostly depend on q_i in the deeper (i.e., level $j+1$) level and x_i in the shallower level (i.e., level j). To improve the estimation accuracy, we keep track of what proportion α_i of input files is used to form the newly generated file, and we use the weighted combination between $\{x_i^{old}\}$ (typically one file) in shallower level and $\{q_i^{old}\}$ (one or more files) in deeper level to estimate $q_{i^*}^{new}$. Formally, when a compaction merges a set of files from level j and $j+1$, and writes a new file to level $j+1$, we have:

$$q_{i^*}^{new} = \sum_{i \in I_j} \alpha_i \cdot x_i^{old} + \sum_{i \in I_{j+1}} \alpha_i \cdot q_i^{old}, \tag{10}$$

where I_j, I_{j+1} represent the set of file IDs in level j and $j+1$ that get compacted. For example, in Figure 9, x_1^{old} is the number of existing queries in the shallower level j, and we use x_1^{old} instead of q_1^{old} to estimate q_5^{new} to avoid the double counting issue. In addition to $q_{i^*}^{new}$, the number of existing queries x_i does not have the double counting issue because all the existing queries for the file in level j terminate at the same level, thus, they do not *contaminate* the x_i of the deeper level. Replacing the average model in the naïve strategy with our weighted model, we formally have:

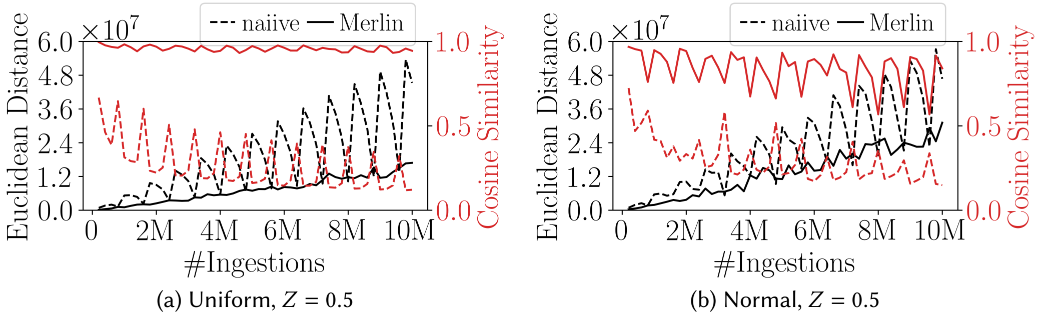
$$x_{i^*}^{new} = \sum_{i \in I_j \cup I_{j+1}} \alpha_i \cdot x_i^{old} \tag{11}$$

As we do not count $q_i^{old}(i \in I_j)$, we may underestimate $q_{i^*}^{new}$. To avoid this, we calculate $\min_{i \in I_j}\{q_i^{old}/n_i\}$, which stands for the minimum average number of point queries per entry in a compaction, and then obtain a lower bound for $q_{i^*}^{new}$ by multiplying $n_{i^*}^{new}$ with the minimum average whenever a new file is being created.

5.2.3 Mnemosyne+: Integrating Merlin with Our Cost Model into Mnemosyne. We add the queue and other counter variables into the metadata of the `FileMetaData` object to implement the sliding-window-based estimation. During each compaction, we calculate q_i^{avg}, the average number of queries per key for each input File i, and we embed q_i^{avg} into the compaction iterator. When the compaction iterator adds a key-value pair to a new File i^* generated from the compaction, we accumulate q_i^{avg} to q'_{i^*}. To determine bpk_{i^*}, we estimate $\{z_i\}$ for all the files during each flush and compaction, and run Algorithm 1 to get C from Eq. (9). After that, we use C, n_{i^*}, and $z_{i^*} = q'_{i^*} - x'_{i^*}$ to calculate bpk_{i^*} (following line 20 in Algorithm 1). We apply the same bpk adjustment as we do in Mnemosyne. Benefited by empty query estimation, Mnemosyne+ is able to skip BFs when most point queries become non-empty, which can be achieved even without compactions. Currently, Mnemosyne+ does not have a more advanced adaptive mechanism that adapts to changing skew. However, as we break a large BF into several small BFs, future work can consider adaptively using small BFs for read-only workloads.

5.3 Micro-benchmark for Estimation

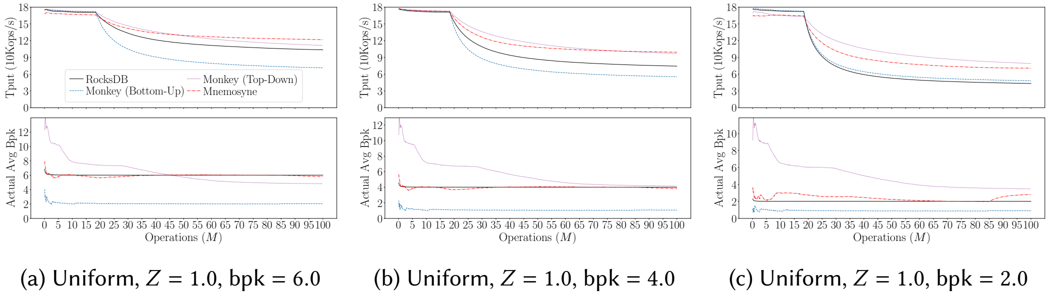
We now evaluate the accuracy of the estimated number of zero-result point queries (z_i) with Merlin.
Experimental Methodology. We first populate an LSM tree with $21M$ 512B key-value pairs with size ratio 4 and file size 16MB. To examine the accuracy of a tracking strategy, we copy the database and execute a mixed workload with $31M$ point queries and $10M$ updates. For every $200K$ updates, we copy the entire database with resetting all statistics counters, and re-execute all the point queries issued so far. By doing this, we obtain the *ground-truth* access statistics for every $200K$ updates, and then we can compare the estimated statistics with the ground truth. To quantify the accuracy, we build two $\{z_i\}$ vectors that record the number of estimated/ground-truth z_i per file and compare two vectors using both Euclidean distance and cosine similarity. We do not consider as a baseline the tracking mechanism in ElasticBF [41] because it eliminates statistics after *expiredTime* and thus the Euclidean distance could deviate further away compared to naïve tracking. We then report how the distance and the similarity change as the database receives more updates for different query workloads. We repeat the experiment three times, and we report the average. In the interest of space, we only show our experimental results for $Z = 0.5$ (that is, half of the queries are zero-result queries) when LSM trees are built in a bottom-up manner in Figure 10. Our findings for $Z = 0, 1$ or for top-down LSM tree construction are similar, and thus the figures are omitted. In Figure 10, black lines stand for Euclidean distance (lower black lines mean higher accuracy) and red lines stand for cosine similarity (higher red lines mean higher accuracy).
Observations. As shown in Figure 10a, Merlin has much smaller Euclidean distance from the ground truth than the naïve strategy, and more than 2× larger cosine similarity of the naïve strategy to the ground truth, supporting that **Merlin offers more accurate statistics estimation**. Note that Merlin cannot be perfectly accurate because no strategy can get the exact z_i for new files during compactions with affordable CPU overhead. In mixed workloads, where flushes and compactions occur alongside queries, statistics can deviate further from the ground truth. Both strategies show increasing Euclidean distance as ingestion grows. We also observe oscillating patterns, particularly in the naïve strategy. This is mainly because the gap between the naïve approach and the ground truth stems from statistical differences at Level 0, where past queries are not counted. At checking points, when statistics are evaluated, Level 0 files are just compacted to Level 1. The averaging



(a) Uniform, $Z = 0.5$

(b) Normal, $Z = 0.5$

Fig. 10. The Euclidean distance (black lines) represents the absolute error, where Merlin (dashed line) shows significantly lower error compared to naïve tracking (solid line). Furthermore, when comparing cosine similarity (red lines), Merlin also outperforms naïve tracking by exhibiting higher similarity.



(a) Uniform, $Z = 1.0$, bpk = 6.0

(b) Uniform, $Z = 1.0$, bpk = 4.0

(c) Uniform, $Z = 1.0$, bpk = 2.0

Fig. 11. Mnemosyne outperforms RocksDB and Monkey without violating the user-defined memory budget.

approach, which tends to overestimate the number of queries, reduces the impact of missing counters. Consequently, a local maximum in Euclidean distance (or minimum in cosine similarity) occurs when Level 0 has files, while a local minimum Euclidean distance (or maximum cosine similarity) occurs when Level 0 is empty. A similar explanation applies to Merlin, as its largest statistical discrepancies also occur in the shallower levels. However, the oscillation amplitude of Merlin is significantly smaller compared to the naïve strategy, with the maximum similarity of Merlin remaining close to 1. However, when it comes to skew queries, we observe a much drastic oscillating pattern of Merlin in Figure 10b. This is because Merlin still has to average the access frequency to every key during compactions, where the actual accesses may be centralized in a small key range. Unless we consume more memory and build another global histogram to record the access frequency, we cannot be more accurate just using the current extra meta data we maintain.

6 Evaluation

We implement Mnemosyne on top of RocksDB (v8.9.1), and we integrate it with Merlin and our workload-aware cost model to construct Mnemosyne$^+$. In this section, we first compare Mnemosyne with Monkey (Top-Down), Monkey (Bottom-Up), and RocksDB, i.e., the same bpk per file. We also compare Mnemosyne$^+$ and Mnemosyne with RocksDB in other experiments. Note that we cannot run IdealMonkey for evolving LSM trees, as IdealMonkey requires the exact number of entries per level, which cannot be obtained in end-to-end experiments. For offline benchmark on IdealMonkey, readers can refer to Section 3.2.

Environment. We use an in-house server, configured with two Intel Xeon Gold 6230 2.1GHz processors, each having 20 cores with virtualization enabled, and 375GB of main memory. By

default, we use a 350GB Optane P4800X SSD with direct I/O enabled (reading a 4KB page takes around 15 μs) as our disk storage. We use gcc version 12.3.1 with optimization level -O2 enabled.

Table 3. Synthetic Workloads.

Type	Description
I	$40M$ inserts mixed with $40M$ empty queries, and $20M$ updates
II	$21M$ inserts, followed by mixed $31M$ queries and $10M$ updates

Experimental Methodology. We experiment with two synthetic workloads produced by an existing workload generator [71] and an industry-grade benchmark YCSB [14]. The synthetic workloads in our experiments are classified in Table 3 where queries in these two workload types can follow different distributions (i.e., uniform or normal distribution with $\sigma = 3.0$). In workload type I, each key-value pair consists of 128 bytes (32 bytes for the key and 96 bytes for the value), whereas in workload type II, each key-value pair comprises 512 bytes (128 bytes for the key and 384 bytes for the value). Consequently, the total workload sizes for workload types I and II are 4.8 GB and 10 GB, respectively. We fix the size ratio $T = 4$ and the target SST file size as 16MB in all the experiments. By default, we enable Write-Ahead-Log and use the default compaction policy of RocksDB (i.e., tiering in Level 0 and leveling in all other levels) in all the experiments. More detailed RocksDB configuration can be found in our code repository. In addition, we focus on bpk $\in [2, 7]$ in our experiments, which corresponds to a practical system setup under memory pressure. To emulate the memory pressure, the block cache is set as around 5% of the workload size accordingly. When testing Mnemosyne$^+$ with synthetic workloads, we reduce the block cache for Mnemosyne$^+$ by the amount of extra memory occupied by our statistics, to ensure the overall memory budget is approximately the same as other baselines. The extra memory is estimated according to Section 5.2.1, where the total number of files is collected after running RocksDB with the same workload. As Mnemosyne does not rely on extra statistics, we do not apply the above block cache adjustment for Mnemosyne.

Mnemosyne Achieves Higher Throughput Without Excessive Bits-per-key Usage. We first compare Mnemosyne with RocksDB, Monkey (Top-Down), and Monkey (Bottom-Up) by measuring throughput and the actual average bpk. The experiment uses workload type I, where approximately $5M$ inserts are ingested first, followed by another $5M$ inserts interleaved with other operations. In this experiment, we vary the bpk between 2 and 6 and construct Mnemosyne and RocksDB in a top-down manner. The results are shown in Figure 11 (similar results can be reproduced for bottom-up construction). The red, black, purple, and blue lines represent Mnemosyne, RocksDB, Monkey (Top-Down), and Monkey (Bottom-Up), respectively. Since the first $20M$ operations consist of pure inserts, the initial throughput is high and gradually decreases as more point queries are introduced. As shown in Figure 11, **the write throughput of Mnemosyne only decreases by no more than 2% during the first 20M inserts**. When more point queries and updates are issued, the performance of all systems drops, however, Mnemosyne begins to outperform all other baselines, without excessive use of bpk. In Figure 11a, when bpk = 6, the actual bpk is underutilized for both versions of Monkey, and their overall throughput is less than standard RocksDB (which uses the same bpk for all files). In contrast, the average bpk in Mnemosyne is closer to the intended value, and it achieves higher throughput than all baselines. The trends observed in Figure 11a are also visible for designs with smaller bpk in Figures 11b and 11c.

Additionally, we observe that bpk utilization in both versions of Monkey is inconsistent, deviating from the user-defined bpk as we vary bpk. This inconsistency is due to the fact that Monkey requires an estimation of the LSM tree shape to compute a static bpk list, but the actual LSM tree shape can

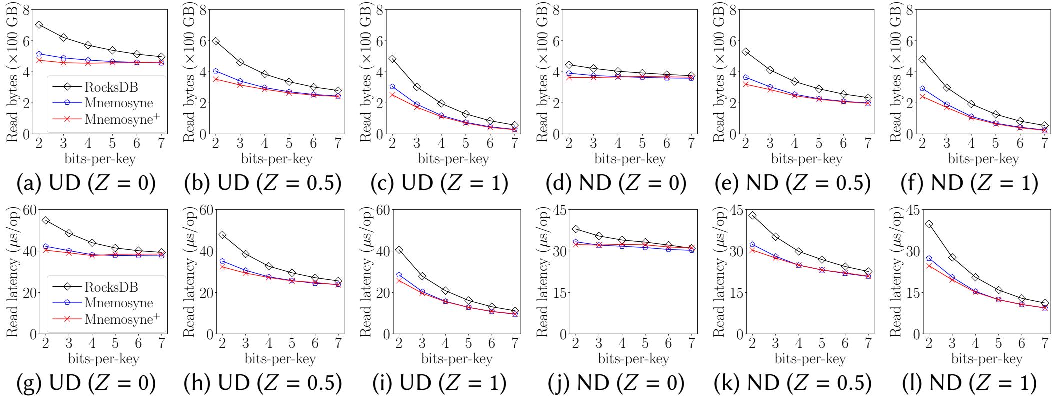


Fig. 12. Mnemosyne$^+$ and Mnemosyne significantly reduce the read bytes and the query latency of RocksDBfor small bpk where UD and ZD stand for uniform and Normal distribution.

vary significantly due to concurrent compactions and flushes. As a result, the actually allocated bpk fluctuates due to inaccuracies in the LSM tree shape estimation. Furthermore, although we set the overuse threshold at 20% in Mnemosyne, we do not see the actual bpk exceeding this threshold when bpk is large, as shown in Figures 10a and 10b. However, when bpk = 2, as the smallest non-zero bpk is 1, the largest actual bpk in Mnemosyne approaches 2.5, contributing to more noticeable oscillations in Figures 11c.

Mnemosyne$^+$ have Higher Benefit for Small Bits-per-key. We test six workloads of type II where we vary the query distribution and the fraction of empty queries (Z), and report the total number of read bytes from disk and the average latency per query in Figure 12. We first observe that Mnemosyne$^+$ has higher benefit for small bpk (e.g., 2, 3 for Mnemosyne and $2 \sim 6$ for RocksDB). As the workload becomes more skewed (moving from Uniform distribution to Normal distribution), the benefit of Mnemosyne$^+$ remains for both $Z = 0.5$ and $Z = 1.0$, which is consistent with our headroom experiments in §3.2. However, we do not observe significant performance improvement for $Z = 0$. This is because, when all queries are non-empty queries, the query latency is dominated by retrieving data blocks from the disk. As false positive results from BFs only trigger a small portion of all the data block accesses for workloads with all non-empty queries, Mnemosyne$^+$ thus has very similar performance to Mnemosyne. Nevertheless, Mnemosyne and Mnemosyne$^+$ still outperform RocksDB for all tested bpk and all different workloads. Specifically, when bpk = 2.0, Mnemosyne and Mnemosyne$^+$ can achieve 1.42× and 1.61× speedup over RocksDB for skewed empty query workload, as shown in Figure 12l.

Latency Benefits Remain With a Different SSD. To further study the impact of the underlying storage, we re-run the all empty workloads with normal and uniform distributions using a slower NVM SSD (Intel P4510 SFF), of which reading a 4KB page takes 36 μs, 2.4× slower than our default SSD. We compare the average query latency and the total number of read bytes in Figure 13. Although we have a smaller I/O reduction for skewed workloads due to more cache hits, as we already discussed, reading a data block now becomes 2.4× more expensive, and thus, we observe a larger benefit of Mnemosyne$^+$ over Mnemosyne and RocksDB. Compared with RocksDB, Mnemosyne$^+$ achieves up to 1.9× and 2× speedup for uniform and normal distribution when bpk = 2. Further, Mnemosyne$^+$ dominates Mnemosyne when the query distribution becomes more skewed. When the query workload follows a uniform distribution, Mnemosyne dominates RocksDB with lower latency but it nearly overlaps with Mnemosyne$^+$, as shown in Figure 13a. For queries that follow normal distribution, we observe higher benef it for the slower device. Specifically, when

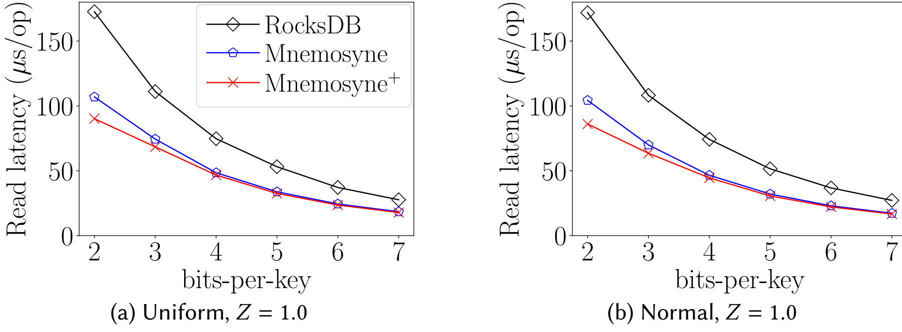


Fig. 13. Mnemosyne+ and Mnemosyne achieve higher latency benefits when using a slower SSD.

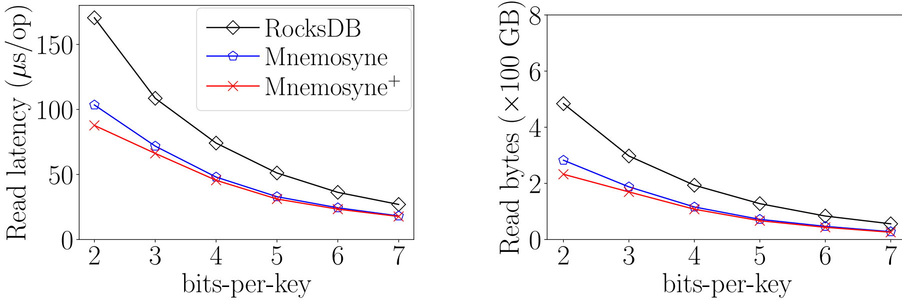
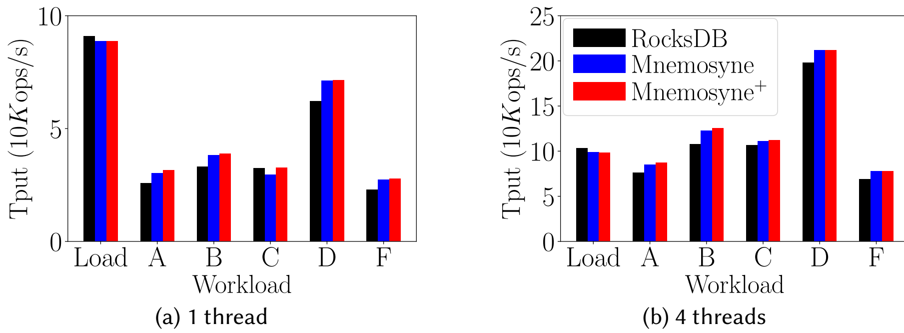


Fig. 14. Mnemosyne+ and Mnemosyne have a similar benefit over RocksDB when using Ribbon Filter.

bpk = 3, Mnemosyne+ has 10% lower query latency than Mnemosyne in a slower SSD while the latency reduction becomes 5% for the faster one, as we compare Figure 13b with Figure 12l.

Mnemosyne and Mnemosyne+ Can be Seamlessly Integrated with the Ribbon Filter and still Outperform RocksDB. Although Mnemosyne and Mnemosyne+ were originally proposed to distribute bpk for BFs, the assignment strategy can also be applied for alternative filters, such as Ribbon Filters. Benefited by the well-decoupled code interface provided by RocksDB, we can specify the filter implement policy as Ribbon Filter in RocksDB without changing anything else. By default, the Ribbon Filter construction interface allows to specify BF-equivalent bpk, which achieves the same false positive rate of BF. Using this interface, we can easily compare Mnemosyne and Mnemosyne+ with RocksDB on top of Ribbon Filters. In this experiment, we still use the slower SSD and test with the workload with uniform all empty queries. The experimental results are summarized in Figure 14. While the observed patterns are consistent with Figures 13a and 12c, we highlight that the exact query latency and the number of read bytes both decrease by 2% on average for all tested approaches. From the above experimental results, we see that Mnemosyne and Mnemosyne+ can co-exist with alternative filters and remain dominant over RocksDB.

Mnemosyne+ Outperforms Mnemosyne and RocksDB in Most YCSB Workloads. We use YCSB [14] (specifically its C++ version [37]) to compare the performance of Mnemosyne+, Mnemosyne, and RocksDB. For this experiment, the `fieldlength` is set to 9 bytes, resulting in a key-value size of $24 + 10 \cdot 9 = 114$ bytes. With bpk = 2.0, direct I/O enabled, and a block cache of 200MB, we run each workload (except workload E, a range query workload) with `operationcount` of $60M$ and `recordcount` of $30M$. To evaluate multi-threaded performance, we respectively report the throughputs of experiments using a single and four threads. The results are summarized in Figure 15. We observe that Mnemosyne+ and Mnemosyne exhibit a 2% lower loading (insertion)



(a) 1 thread

(b) 4 threads

Fig. 15. Mnemosyne+ has 2% lower throughput than RocksDB when loading YCSB, but Mnemosyne+ gains up to 20% improvement in all other workloads.

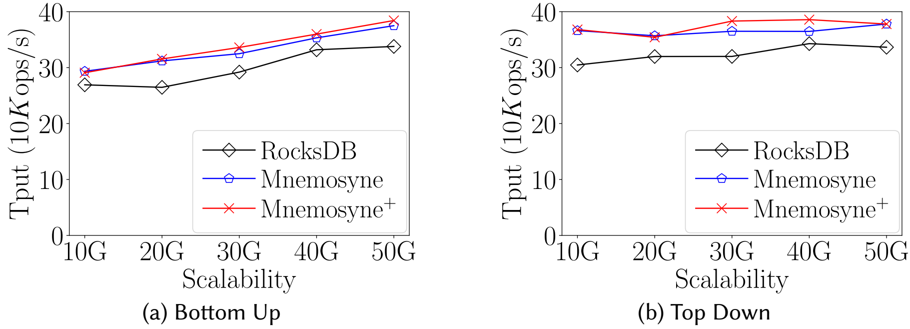
throughput compared to RocksDB, primarily due to the dynamic bits-per-key reallocation during each flush and compaction. However, Mnemosyne+ outperforms RocksDB in all other workloads by up to 20% and shows superior performance over Mnemosyne in workloads A, B, and C. For workloads D and F, there is no significant performance difference between Mnemosyne+ and Mnemosyne. In workload F, where `readmodifywrite` operations exhibit skew, many queries are also targeting newly updated entries, many queries also target recently updated entries, resulting in comparable performance between Mnemosyne+ and Mnemosyne. Nevertheless, as shown in Figure 15b, both Mnemosyne+ and Mnemosyne achieve substantial performance gains over RocksDB, with improvements of 15% and 21% for workloads D and F respectively.

Mnemosyne and Mnemosyne+ Dominates RocksDB When Scales up. We also use YCSB to examine the scalability of Mnemosyne and Mnemosyne+. In this experiment, we choose workload type b in YCSB and use 16 threads by default. We vary the database size from 10GB to 50GB (i.e., varying both `operationcount` and `recordcount` from 100M to 500M). The block cache size is fixed as 5% of the database size, ranging from 512MB to 2.5GB. We repeat the above experiments using both bottom-up and top-down LSM tree construction for all three methods. In Figure 16, we observe that in both bottom-up and top-down construction methods, Mnemosyne+ and Mnemosyne remain dominant over RocksDB. Moreover, while Mnemosyne+ achieves up to 5% higher throughput than Mnemosyne, their performance is also closely comparable. This is due to the maximum overused bpk limitation, where Mnemosyne+ assigns a small bpk to files that should be assigned with a larger bpk. Although this limitation ensures the average bpk remains controlled, it can lead to performance regression. However, as more entries are ingested and deeper level compactions are triggered, the actual average bpk decreases, allowing larger bpk values to be assigned to new files.

7 Related Work

Memory Allocation in LSM trees. In addition to memory allocation within BFs, memory can also be reallocated among BFs, fence pointers, and the write buffer [16, 36]. Further, memory can be further reallocated between multiple LSM trees [43, 44] in LSM-based storage systems. In fact, our sort-and-search algorithm (§4.1) to find the optimal bits-per-key per BF can be seamlessly integrated into these techniques to achieve a better holistic memory tuning.

Membership-Testing Filters. Many membership-testing filters have been proposed in the past literature including Blocked Bloom Filter [52], Cuckoo Filter [27], Quotient Filter [51], Morton Filter [9], Vacuum Filter [62], Xor Filter [30], Ribbon Filter [23], and InfiniFilter [17]. To replace Bloom Filters in LSM trees with any other fingerprint-based filter, we can replace the FPR definition in Eq. (1) and the core idea of our sort-and-search algorithm still applies. In addition, Chucky [20]



(a) Bottom Up

(b) Top Down

Fig. 16. Mnemosyne and Mnemosyne+ dominates RocksDB when the database grows in YCSB workload B.

and SlimDB [53] construct a global Cuckoo Filter for an LSM tree that does not work well under memory pressure. Specifically, when the filter does not fit in cache, every compaction updates the on-disk part of the filter. Furthermore, Partitioned Learned Bloom Filter (PLBF) [59] targets finding how to partition the key space and how to allocate bpk per partition to find the optimal FPR with constrained memory. Although PLBF has a more general optimization function than Mnemosyne, our system aims optimize BFs in the context of LSM-trees instead of the key domain where the key ranges of all the files are also pre-determined by ingestion, iterative compactions, and file sizes.

Skew-Aware Key-Value Stores. Existing skew-aware read optimizations for LSM trees primarily focus on caching policies [61, 66, 68] or allocating extra memory to frequently queried keys [31, 69]. When memory is limited, BFs can be partitioned into smaller components — as in ElasticBF [41] and ModularBF [49] — to avoid loading the entire filter into memory. To determine how many of these smaller filters are needed to answer a query within an SST file, some systems maintain per-file query statistics, enabling skew-aware optimization in LSM trees. All of these skew-aware techniques are compatible with Mnemosyne+. Additionally, there are skew-aware methods designed for hash-table-based and purely in-memory key-value stores [38, 39, 42]; however, these are tailored to their specific data structures and are not directly applicable to LSM trees.

8 Conclusion

In this paper, we propose a workload-aware Bloom Filter memory allocation strategy with accurate statistics estimation in LSM trees. We build a more general cost model that considers the access pattern per file and we further design Merlin, a novel and accurate query statistics tracking mechanism. We implement Mnemosyne+ by integrating Merlin and Mnemosyne into RocksDB. With limited memory, Mnemosyne+ achieves up to 2× improvement over RocksDB and 10% latency reduction over Mnemosyne.

Acknowledgments

We thank the anonymous reviewers for their constructive feedback. This work is partially funded by the National Science Foundation under Grants No. IIS-2144547 and CCF-2403012, a Facebook Faculty Research Award, and a Meta Gift.

References

[1] Sattam Alsubaiee, Yasser Altowim, Hotham Altwaijry, Alexander Behm, Vinayak R. Borkar, Yingyi Bu, Michael J. Carey, Inci Cetindil, Madhusudan Cheelangi, Khurram Faraaz, Eugenia Gabrielova, Raman Grover, Zachary Heilbron, Young-Seok Kim, Chen Li, Guangqiang Li, Ji Mahn Ok, Nicola Onose, Pouria Pirzadeh, Vassilis J. Tsotras, Rares Vernica, Jian Wen, and Till Westmann. 2014. AsterixDB: A Scalable, Open Source BDMS. *Proceedings of the VLDB Endowment* 7, 14 (2014), 1905–1916. http://www.vldb.org/pvldb/vol7/p1905-alsubaiee.pdf

[2] Amazon. [n.d.]. EC2 Instance Types. https://aws.amazon.com/ec2/instance-types/.

[3] Apache. 2023. Accumulo. *https://accumulo.apache.org/* (2023).

[4] Apache. 2023. Cassandra. *http://cassandra.apache.org* (2023).

[5] Apache. 2023. HBase. *http://hbase.apache.org/* (2023).

[6] Raja Appuswamy, Renata Borovica-Gajic, Goetz Graefe, and Anastasia Ailamaki. 2017. The Five minute Rule Thirty Years Later and its Impact on the Storage Hierarchy. In *Proceedings of the International Workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architectures (ADMS).*

[7] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. 2012. Workload Analysis of a Large-scale Key-value Store. In *Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems.* 53–64.

[8] Burton H Bloom. 1970. Space/Time Trade-offs in Hash Coding with Allowable Errors. *Commun. ACM* 13, 7 (1970), 422–426. http://dl.acm.org/citation.cfm?id=362686.362692

[9] Alexander Breslow and Nuwan Jayasena. 2018. Morton Filters: Faster, Space-Efficient Cuckoo Filters via Biasing, Compression, and Decoupled Logical Sparsity. *Proceedings of the VLDB Endowment* 11, 9 (2018), 1041–1055.

[10] Zhichao Cao, Siying Dong, Sagar Vemuri, and David H C Du. 2020. Characterizing, Modeling, and Benchmarking RocksDB Key-Value Workloads at Facebook. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST).* 209–223.

[11] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. 2006. Bigtable: A Distributed Storage System for Structured Data. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI).* 205–218. http://dl.acm.org/citation.cfm?id=1267308.1267323

[12] CockroachDB. 2021. CockroachDB. *https://github.com/cockroachdb/cockroach* (2021).

[13] Alex Conway, Martin Farach-Colton, and Rob Johnson. 2023. SplinterDB and Maplets: Improving the Tradeoffs in Key-Value Store Compaction Policy. *Proceedings of the ACM on Management of Data (PACMMOD)* 1, 1 (2023), 46:1–46:27. https://doi.org/10.1145/3588726

[14] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC).* 143–154. http://doi.acm.org/10.1145/1807128.1807152

[15] Niv Dayan, Manos Athanassoulis, and Stratos Idreos. 2017. Monkey: Optimal Navigable Key-Value Store. In *Proceedings of the ACM SIGMOD International Conference on Management of Data.* 79–94. http://doi.acm.org/10.1145/3035918.3064054

[16] Niv Dayan, Manos Athanassoulis, and Stratos Idreos. 2018. Optimal Bloom Filters and Adaptive Merging for LSM-Trees. *ACM Transactions on Database Systems (TODS)* 43, 4 (2018), 16:1–16:48. https://doi.org/10.1145/3276980

[17] Niv Dayan, Ioana O Bercea, Pedro Reviriego, and Rasmus Pagh. 2023. InfiniFilter: Expanding Filters to Infinity and Beyond. *Proceedings of the ACM on Management of Data (PACMMOD)* 1, 2 (2023), 140:1–140:27. https://doi.org/10.1145/3589285

[18] 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 ACM SIGMOD International Conference on Management of Data.* 505–520. http://doi.acm.org/10.1145/3183713.3196927

[19] Niv Dayan and Stratos Idreos. 2019. The Log-Structured Merge-Bush & the Wacky Continuum. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD).* 449–466.

[20] Niv Dayan and Moshe Twitto. 2021. Chucky: A Succinct Cuckoo Filter for LSM-Tree. In *Proceedings of the ACM SIGMOD International Conference on Management of Data.* 365–378. https://doi.org/10.1145/3448016.3457273

[21] 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. https://www.vldb.org/pvldb/vol15/p3071-dayan.pdf

[22] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. 2007. Dynamo: Amazon's Highly Available Key-value Store. *ACM SIGOPS Operating Systems Review* 41, 6 (2007), 205–220. http://dl.acm.org/citation.cfm?id=1323293.1294281

[23] Peter C. Dillinger, Lorenz Hübschle-Schneider, Peter Sanders, and Stefan Walzer. 2022. Fast Succinct Retrieval and Approximate Membership Using Ribbon. In *Proceedings of the International Symposium on Experimental Algorithms (SEA)*. 4:1–4:20. https://doi.org/10.4230/LIPIcs.SEA.2022.4
[24] Facebook. 2023. Level Compaction Dynamic Level Bytes in RocksDB. https://github.com/facebook/rocksdb/wiki/Leveled-Compaction#option-level_compaction_dynamic_level_bytes-and-levels-target-size. (Accessed: 2025-01-16).
[25] Facebook. 2023. MyRocks. *http://myrocks.io/* (2023).
[26] Facebook. 2024. RocksDB. *https://github.com/facebook/rocksdb* (2024).
[27] Bin Fan, David G Andersen, Michael Kaminsky, and Michael Mitzenmacher. 2014. Cuckoo Filter: Practically Better Than Bloom. In *Proceedings of the ACM International on Conference on emerging Networking Experiments and Technologies (CoNEXT)*. 75–88. http://doi.acm.org/10.1145/2674005.2674994
[28] Guy Golan-Gueta, Edward Bortnikov, Eshcar Hillel, and Idit Keidar. 2015. Scaling Concurrent Log-Structured Data Stores. In *Proceedings of the ACM European Conference on Computer Systems (EuroSys)*. 32:1–32:14. http://doi.acm.org/10.1145/2741948.2741973
[29] Google. 2021. LevelDB. *https://github.com/google/leveldb/* (2021).
[30] Thomas Mueller Graf and Daniel Lemire. 2019. Xor Filters: Faster and Smaller Than Bloom and Cuckoo Filters. *CoRR* abs/1912.0 (2019). http://arxiv.org/abs/1912.08258
[31] Xiangpeng Hao and Badrish Chandramouli. 2024. Bf-Tree: A Modern Read-Write-Optimized Concurrent Larger-Than-Memory Range Index. *Proceedings of the VLDB Endowment* 17, 11 (2024), 3442–3455. https://www.vldb.org/pvldb/vol17/p3442-hao.pdf
[32] Naohiro Hayashibara, Xavier Défago, Rami Yared, and Takuya Katayama. 2004. The ϕ Accrual Failure Detector. In *Proceedings of the International Symposium on Reliable Distributed Systems (SRDS)*. 66–78. https://doi.org/10.1109/RELDIS.2004.1353004
[33] Marc Holze, Ali Haschimi, and Norbert Ritter. 2010. Towards workload-aware self-management: Predicting significant workload shifts. *Proceedings of the IEEE International Conference on Data Engineering (ICDE)* (2010), 111–116.
[34] Gui Huang, Xuntao Cheng, Jianying Wang, Yujie Wang, Dengcheng He, Tieying Zhang, Feifei Li, Sheng Wang, Wei Cao, and Qiang Li. 2019. X-Engine: An Optimized Storage Engine for Large-scale E-commerce Transaction Processing. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 651–665.
[35] Andy Huynh, Harshal A. Chaudhari, Evimaria Terzi, and Manos Athanassoulis. 2024. Towards flexibility and robustness of LSM trees. *The VLDB Journal* (2024), 1–24. https://doi.org/10.1007/s00778-023-00826-9
[36] Stratos Idreos, Niv Dayan, Wilson Qin, Mali Akmanalp, Sophie Hilgard, Andrew 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)*. https://www.cidrdb.org/cidr2019/papers/p143-idreos-cidr19.pdf
[37] Ren Jinglei, Kjellqvist Chris, and Deng Long. 2024. YCSB-C. (2024). https://github.com/basicthinker/YCSB-C
[38] Konstantinos Kanellis, Badrish Chandramouli, and Shivaram Venkataraman. 2023. F2: Designing a Key-Value Store for Large Skewed Workloads. *CoRR* abs/2305.0 (2023). https://doi.org/10.48550/arXiv.2305.01516
[39] Hongbo Kang, Yiwei Zhao, Guy E Blelloch, Laxman Dhulipala, Yan Gu, Charles McGuffey, and Phillip B Gibbons. 2022. PIM-tree: A Skew-resistant Index for Processing-in-Memory. *Proceedings of the VLDB Endowment* 16, 4 (2022), 946–958. https://www.vldb.org/pvldb/vol16/p946-kang.pdf
[40] Hiwot Tadese Kassa, Jason Akers, Mrinmoy Ghosh, Zhichao Cao, Vaibhav Gogte, and Ronald G Dreslinski. 2021. Improving Performance of Flash Based Key-Value Stores Using Storage Class Memory as a Volatile Memory Extension. In *Proceedings of the USENIX Annual Technical Conference (ATC)*. 821–837. https://www.usenix.org/conference/atc21/presentation/kassa
[41] Yongkun Li, Chengjin Tian, Fan Guo, Cheng Li, and Yinlong Xu. 2019. ElasticBF: Elastic Bloom Filter with Hotness Awareness for Boosting Read Performance in Large Key-Value Stores. In *Proceedings of the USENIX Annual Technical Conference (ATC)*. 739–752.
[42] Hyeontaek Lim, Dongsu Han, David G Andersen, and Michael Kaminsky. 2014. MICA: A Holistic Approach to Fast In-Memory Key-Value Storage. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. 429–444. https://www.usenix.org/conference/nsdi14/technical-sessions/presentation/lim
[43] Chen Luo. 2020. Breaking Down Memory Walls in LSM-based Storage Systems. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 2817–2819. https://dl.acm.org/doi/10.1145/3318464.3384399
[44] 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. http://www.vldb.org/pvldb/vol14/p241-luo.pdf
[45] Chen Luo and Michael J. Carey. 2020. LSM-based Storage Techniques: A Survey. *The VLDB Journal* 29, 1 (2020), 393–418. https://link.springer.com/article/10.1007%2Fs00778-019-00555-y

[46] Lin Ma, Dana Van Aken, Ahmed Hefny, Gustavo Mezerhane, Andrew Pavlo, and Geoffrey J Gordon. 2018. Query-based Workload Forecasting for Self-Driving Database Management Systems. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 631–645.

[47] John C. McCallum. 2022. Historical Cost of Computer Memory and Storage. *https://jcmit.net/mem2015.htm* (2022).

[48] Gabriel Mersy, Zhuo Wang, Stavros Sintos, and Sanjay Krishnan. 2024. Optimizing Collections of Bloom Filters within a Space Budget. *Proceedings of the VLDB Endowment* 17, 11 (2024), 3551–3564. https://www.vldb.org/pvldb/vol17/p3551-mersy.pdf

[49] Ju Hyoung Mun, Zichen Zhu, Aneesh Raman, and Manos Athanassoulis. 2022. LSM-Tree Under (Memory) Pressure. In *Proceedings of the International Workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architectures (ADMS)*. 23–35. https://adms-conf.org/2022-camera-ready/ADMS22_mun.pdf

[50] Patrick E. O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth J. O'Neil. 1996. The log-structured merge-tree (LSM-tree). *Acta Informatica* 33, 4 (1996), 351–385. http://dl.acm.org/citation.cfm?id=230823.230826

[51] Prashant Pandey, Alex Conway, Joe Durie, Michael A Bender, Martin Farach-Colton, and Rob Johnson. 2021. Vector Quotient Filters: Overcoming the Time/Space Trade-Off in Filter Design. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 1386–1399. https://doi.org/10.1145/3448016.3452841

[52] Felix Putze, Peter Sanders, and Johannes Singler. 2009. Cache-, hash-, and space-efficient bloom filters. *ACM Journal of Experimental Algorithmics* 14 (2009).

[53] Kai Ren, Qing Zheng, Joy Arulraj, and Garth Gibson. 2017. SlimDB: A Space-Efficient Key-Value Storage Engine For Semi-Sorted Data. *Proceedings of the VLDB Endowment* 10, 13 (2017), 2037–2048. http://www.vldb.org/pvldb/vol10/p2037-ren.pdf

[54] RocksDB. 2023. Expanding Picked Files Before Compaction. (2023). https://github.com/facebook/rocksdb/blob/8.9.fb/db/compaction/compaction_picker.cc#L497

[55] Subhadeep Sarkar, Tarikul Islam Papon, Dimitris Staratzis, and Manos Athanassoulis. 2020. Lethe: A Tunable Delete-Aware LSM Engine. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 893–908.

[56] Subhadeep Sarkar, Dimitris Staratzis, Zichen Zhu, and Manos Athanassoulis. 2021. Constructing and Analyzing the LSM Compaction Design Space. *Proceedings of the VLDB Endowment* 14, 11 (2021), 2216–2229. http://vldb.org/pvldb/vol14/p2216-sarkar.pdf

[57] Russell Sears and Raghu Ramakrishnan. 2012. bLSM: A General Purpose Log Structured Merge Tree. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 217–228. http://doi.acm.org/10.1145/2213836.2213862

[58] Sasu Tarkoma, Christian Esteve Rothenberg, and Eemil Lagerspetz. 2012. Theory and Practice of Bloom Filters for Distributed Systems. *IEEE Communications Surveys & Tutorials* 14, 1 (2012), 131–155. http://ieeexplore.ieee.org/xpl/login.jsp?arnumber=5751342

[59] Kapil Vaidya, Eric Knorr, Michael Mitzenmacher, and Tim Kraska. 2021. Partitioned Learned Bloom Filters. In *Proceedings of the International Conference on Learning Representations (ICLR)*. https://openreview.net/forum?id=6BRLOfrMhW

[60] Chen Wang, Jialin Qiao, Xiangdong Huang, Shaoxu Song, Haonan Hou, Tian Jiang, Lei Rui, Jianmin Wang, and Jiaguang Sun. 2023. Apache IoTDB: A Time Series Database for IoT Applications. *Proceedings of the ACM on Management of Data (PACMMOD)* 1, 2 (2023), 195:1–195:27. https://doi.org/10.1145/3589775

[61] Kefei Wang and Feng Chen. 2023. Catalyst: Optimizing Cache Management for Large In-memory Key-value Systems. *Proceedings of the VLDB Endowment* 16, 13 (2023), 4339–4352. https://www.vldb.org/pvldb/vol16/p4339-chen.pdf

[62] Minmei Wang, Mingxun Zhou, Shouqian Shi, and Chen Qian. 2019. Vacuum Filters: More Space-Efficient and Faster Replacement for Bloom and Cuckoo Filters. *Proceedings of the VLDB Endowment* 13, 2 (2019), 197–210. http://www.vldb.org/pvldb/vol13/p197-wang.pdf

[63] Ran Wei, Zichen Zhu, Andrew Kryczka, Jay Zhuang, and Manos Athanassoulis. 2024. Codebase for Benchmark, Analyze, Optimize Partial Compaction in RocksDB. (2024). https://github.com/BU-DiSC/Benchmark-Analyze-Optimize-Partial-Compaction-in-RocksDB-Codebase

[64] Ran Wei, Zichen Zhu, Andrew Kryczka, Jay Zhuang, and Manos Athanassoulis. 2025. Benchmarking, Analyzing, and Optimizing WA of Partial Compaction in RocksDB. In *Proceedings of the International Conference on Extending Database Technology (EDBT)*. 425–437. https://doi.org/10.48786/edbt.2025.34

[65] WiredTiger. 2021. Source Code. *https://github.com/wiredtiger/wiredtiger* (2021).

[66] Fenggang Wu, Ming-Hong Yang, Baoquan Zhang, and David H C Du. 2020. AC-Key: Adaptive Caching for LSM-based Key-Value Stores. In *Proceedings of the USENIX Annual Technical Conference (ATC)*. 603–615. https://www.usenix.org/conference/atc20/presentation/wu-fenggang

[67] 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 *Proceedings of the USENIX Annual Technical Conference (ATC)*. 71–82. https://www.usenix.org/conference/atc15/technical-session/presentation/wu

[68] Lei Yang, Hong Wu, Tieying Zhang, Xuntao Cheng, Feifei Li, Lei Zou, Yujie Wang, Rongyao Chen, Jianying Wang, and Gui Huang. 2020. Leaper: A Learned Prefetcher for Cache Invalidation in LSM-tree based Storage Engines. *Proceedings*

of the VLDB Endowment 13, 11 (2020), 1976–1989.

[69] Jianshun Zhang, Fang Wang, and Chao Dong. 2022. HaLSM: A Hotspot-aware LSM-tree based Key-Value Storage Engine. In *IEEE 40th International Conference on Computer Design, ICCD 2022, Olympic Valley, CA, USA, October 23-26, 2022*. 179–186. https://doi.org/10.1109/ICCD56317.2022.00035

[70] 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 International Workshop on Data Management on New Hardware (DAMON)*. 1:1–1:10.

[71] Zichen Zhu, Arpita Saha, Manos Athanassoulis, and Subhadeep Sarkar. 2024. KVBench: A Key-Value Benchmarking Suite. In *International Workshop on Testing Database Systems (DBTest)*. 9–15. https://doi.org/10.1145/3662165.3662765

Received October 2024; revised January 2025; accepted February 2025