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

Zichen Zhu
Boston University
zczhu@bu.edu

Yanpeng Wei
Tsinghua University
weiyp21@mails.tsinghua.edu.cn

Manos Athanassoulis
Boston University
mathan@bu.edu

ABSTRACT

Log-structured merge (LSM) trees employ Bloom Filters (BFs) or other filters 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, the existing model has two limitations: (1) the closed-form solution only works for a *perfectly-shaped* LSM-tree, and (2) the model assumes a uniform query distribution in the key domain. Due to these two 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 propose a new more general LSM cost model that considers the access pattern *per file* for an arbitrary LSM-tree (i.e., the LSM-tree does not have to be in perfect shape). Furthermore, we develop an optimal memory allocation algorithm that converges 3 orders of magnitude faster than the gradient descent algorithm used by the state of the art. Further, to make this approach applicable in real systems, we need to use accurate access statistics per file. We find that no system accurately maintains such statistics. Moreover, obtaining statistics by simply maintaining an access counter per file significantly deviates from the ground truth. To address this issue, we propose Merlin, a dynamic sliding-window-based tracking mechanism that accurately captures the access statistics per file. Combining Merlin together with our new cost model, we build Mnemosyne on top of a state-of-the-art LSM-based key-value store RocksDB. In our evaluation, Mnemosyne reduces query latency by up to 2× compared to the production-grade LSM, RocksDB, under memory pressure for query workloads that exhibit skew or contain a high fraction of existing vs. empty queries.

1 INTRODUCTION

LSM-Tree-Based Key-Value Stores. Log-Structured Merge-trees (**LSM-trees**) [45] have emerged as the core data structure in most modern key-value stores [1, 3–5, 11, 12, 21, 23, 24, 26, 27, 31, 52, 55, 57]. 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

Proceedings of the VLDB Endowment, Vol. 18, No. 1 ISSN 2150-8097.
doi:XX.XX/XXX.XX

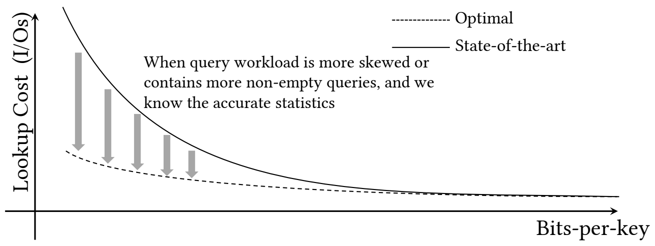


Figure 1: When the query workload becomes more skewed or has more non-empty queries, we can achieve lower lookup cost by optimally assigning bits-per-key per BF if we know the exact number of empty queries per BF, especially when the system has limited memory (i.e., smaller bits-per-key).

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 stores typically maintain metadata such as fence pointers and Bloom filters to reduce the number of storage accesses [41].

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 [27] and RocksDB [23], a sorted run can span multiple *Sorted-String Table* (SST) files, in which case every 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 have 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 (including deletion entries, also called tombstones) in LSM-trees. Since 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 the BFs are prefetched 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 filters from disk, adversely impacting query performance [44].

Challenge 1: Memory Pressure. 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 (the unit price per MB between memory and storage increased from 1.67 to 10 for Flash SSD), leading to decreasing memory-to-storage ratios [6, 43, 44]. 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 configured small enough to ensure most BFs fit in the block cache. Besides, BFs also need to compete for memory resources with fence pointers. For example, for $21M$ 512-byte key-value pairs (128-byte key and 384-byte value), if we use 103 MB block cache ($\approx$ 1% data size), the minimum fence pointer granularity is 16KB so that all index blocks fit in the block cache (one fence pointer needs 140 bytes as discussed later). In that scenario, bits-per-key must be less than six to ensure both BFs and fence pointers fit in memory.

Challenge 2: Workload Skew and Non-empty Queries. Prior work on memory allocation for LSM with memory constraints forces all the files in the same run to have the same bits-per-key, assuming that the query workload is uniform across the key domain. However, real-world workloads exhibit skewed access patterns [7, 10, 13], that is, a small subset of keys are frequently queried, while most keys are rarely or never queried. For skewed workloads, assigning more bits-per-key to files with higher access frequency (even across a single level) can lead to fewer unnecessary I/Os compared to the state-of-the-art approaches. Further, memory allocation in prior work considers non-empty queries (i.e., queried keys are found in LSM-trees) opportunistically [15], while a more accurate cost model has been recently proposed for holistic tuning [32]. In practice, if we know that the queries targeting a file will be exclusively non-empty, no BF is needed for this file since there are no empty queries to be skipped (i.e., bits-per-key should be 0). Rather, any hashing in the BF will only negatively impact query latency due to its CPU cost [62]. As a result, prior work does not systematically consider query skew and the fraction of non-empty queries, leaving a large headroom to improve when determining the bits-per-key per file during BF construction, as shown in Figure 1.

Challenge 3: Imperfect LSM-trees. Existing memory allocation models [14, 17, 19] assume that the given LSM-tree has a *perfect shape*, meaning that the number of entries per level grows *exactly* exponentially with a constant size ratio T. However, this is not true in general, especially when the compaction granularity is a full level or run [20, 51]. There are a few reasons that make an LSM-tree shape imperfect. First, 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 [51], multiple files can be selected to be compacted together into the next level (e.g., the expansion mechanism in RocksDB [49], or partition-based compactions for deeper levels in Spooky [20]). When more than one file is picked, the data size changes more drastically, and so does the size ratio between two adjacent levels. Additionally, compactions are not necessarily triggered by level capacity [51], instead, they can also be triggered by the number of tombstones, and the expiring tombstones with specified Time-To-Live (TTL) [31, 50]. Last but not least, 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 is typically assumed by idealistic models.

Challenge 4: Expensive Optimization. There is a variant of state-of-the-art memory allocation algorithm for BFs that does not rely on perfectly-shaped LSM-trees and it uses a gradient descent algorithm to find the optimal bits-per-key assignment [15], with complexity $O(L^2 \cdot \log M)$ where L is the number of levels, and M is the total memory budget for BFs. It assumes that all the files in the same level have the same bits-per-key and the limited height of the tree (no more than 8 levels in practice) allows the gradient descent algorithm to terminate quickly. However, when we allow for different decisions per file, the complexity becomes $O(F^2 \cdot \log M)$ where F is the total number of BFs (files), which brings a substantial CPU cost when the database grows.

Challenge 5: 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 may seem feasible to maintain query counters and propagate statistics at compaction time (e.g., via averaging), this mechanism overlooks the evolving access patterns during the continuous processes of flushes and compactions (§4). This leads to a significant discrepancy between the tracked accesses and the ground truth. The challenge is estimating the accesses of a newly generated file after a compaction. While 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 was generated earlier. An accurate statistics estimation mechanism can facilitate other workload-specific tuning decisions in LSM-trees, in addition to what we propose in this paper.

Mnemosyne: Optimal BF Memory Allocation via Accurate Statistics Tracking. To address the aforementioned challenges, we build a new, more faithful cost model for tuning with constrained memory. We also develop an accurate LSM-tree statistics tracking mechanism called Merlin[1], using which we build a system that offers workload-aware memory allocation mechanism, Mnemosyne[2]. 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. Note that working with access statistics captures the exact tree structure (whether it is leveling, tiering, or hybrid and how full each level is), thus, Mnemosyne does need to make any assumptions regarding the LSM tree shape. Therefore, Mnemosyne only relies on the user-provided bits-per-key without assuming a perfectly shaped tree or a predefined tree structure. To minimize the overhead when searching the optimal BF size per file, we develop an efficient algorithm with a worst-case complexity of $O(F \cdot \log F)$ that has negligible CPU overhead during flushes and compactions. Overall, Mnemosyne is a practical approach that uses historical workload data to achieve near-optimal memory allocation 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. To address Challenges 1-3, the new model takes into account

[1]A wizard who could foresee the outcome of battles, in Celtic mythology.

[2]The ancient Greek goddess of memory.

workload skew and empty queries per file with a limited memory budget, without assuming a perfectly-shaped LSM-tree (§3.1).

- To overcome Challenge 4, we propose a design navigation algorithm with worst-case complexity $O(F \cdot \log F)$, where F is the number of input files, while the existing gradient descent method has complexity $O(F^2 \cdot \log M)$ where M is the total memory budget (§3.2). Since our search algorithm is applicable to the state of the art, we also augment our baseline system.
- We find that averaging access counters at compaction time for each BF to estimate access frequency leads to a large discrepancy between the estimated frequency and the ground truth (§4.1).
- 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 reduces the estimation error by half for different workloads, thus, also addressing Challenge 5 (§4.2).
- We build Mnemosyne by integrating Merlin and our new cost model into RocksDB. We thoroughly examine Mnemosyne by comparing it with a production-ready RocksDB system and the state-of-the-art memory allocation strategy, Monkey. Mnemosyne outperforms RocksDB by up to 2× and Monkey by up to 15% under memory pressure, when workloads contain non-empty queries or exhibit high skew (§5).

2 BACKGROUND

In this section, we first review the LSM-tree background [41, 45], and then we discuss the state-of-the-art cost models used for optimal memory allocation in LSM-trees [14]. We summarize the most commonly used notations in Table 1.

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)[3]
$n_i (n_j)$	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 i^{th} 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 $\mathcal{Q}$

2.1 LSM-trees

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 the runs are organized in a tree-like structure where each level has exponentially larger capacity according to a predefined size ratio T. A compaction is triggered for a level whenever its accumulated data size reaches the predefined capacity.

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 between the costs associated with reads, writes, and space [17, 18, 20, 48, 51, 59]. 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 compaction strategies [18, 51] that allow different levels to have different compaction policies. In the case of leveling, there could be a significant latency spike when it compacts data from two entire levels, especially for deeper (larger) levels. To amortize the compaction cost, real systems like LevelDB, RocksDB, and CockroachDB 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 compaction to the next level. Note that although we focus on file-based partial leveling compaction in this paper, our new workload-aware bits-per-key assignment model is applicable to any compaction policy.

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, 53] 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 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

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

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.

2.2 Monkey

A Point Lookup Cost Model for Empty Queries. While many LSM-based key-value stores (e.g., LevelDB and RocksDB) use the same bits-per-key for all the files by default, Monkey [14] uses a cost model for point lookups, noted by $Cost(\{\epsilon_j\})$, to obtain the bits-per-key assignment per level. In the leveling case of Monkey, the expected number of accessed data blocks by an empty query is the sum of the FPRs in each level. We can thus define:

$$Cost(\{\epsilon_j\}) = \sum_{j=1}^{L} \epsilon_j, \tag{2}$$

where L is the number of levels, ϵ_j is the false positive rate in level j. Considering the memory budget constraint for the filter size, Monkey formalizes the optimization problem as follows:

$$\begin{aligned} \min_{\epsilon_1, \epsilon_2, \ldots, \epsilon_L} \quad & Cost(\{\epsilon_j\}) \\ \text{subject to} \quad & -\frac{1}{(\ln 2)^2} \cdot \sum_{j=1}^{L} n_j \cdot \ln \epsilon_j \leq M \\ & 0 < \epsilon_j \leq 1, \forall j \in [L], \end{aligned} \tag{3}$$

where n_j represents the number of entries in level j, and M represents the total number of bits for all the filters. When the overall bits-per-key is configured as bpk, we have $M = \text{bpk} \cdot \sum_{j=1}^{L} n_j$.

Closed-form Solutions. The original Monkey cost model assumes a *perfectly-shaped LSM-tree*, that is, the number of entries in each level grows exponentially with a size ratio T. Formally, we have:

$$n_j = n_1 \cdot T^{j-1}, \forall j \in [L]$$

Monkey then derives the optimal point lookup cost (noted by $Cost_{min}$) and the optimal $\{\epsilon_j\}$ with a closed-form solution:

$$\begin{aligned} Cost_{min} &\overset{def}{=} \min\left\{Cost(\{\epsilon_j\})\right\} = e^{-\text{bpk}\cdot(\ln 2)^2 \cdot T^Y} \cdot \frac{T^{\frac{T}{T-1}}}{T-1} + Y, \\ \epsilon_j &= \begin{cases} 1, & \text{for } j > L - Y \\ (Cost_{min} - Y) \cdot \frac{T-1}{T}, & \text{for } j = L - Y \\ (Cost_{min} - Y) \cdot \frac{T-1}{T} \cdot \frac{1}{T^{L-Y-j}}, & \text{for } 1 \leq j < L - Y \end{cases}, \end{aligned} \tag{4}$$

where Y specifies that the false positive rate ϵ_j for the deepest Y levels converges to 1 (when the associated $\{\text{bpk}_j\}$ for these levels are 0, i.e., no BFs). Y can also be calculated in a closed-form manner:

$$Y = \left\lfloor \log_T \left(\frac{\ln T}{(\ln 2)^2 \cdot \text{bpk} \cdot (T-1)} \right) \right\rfloor$$

While the above model only works for the leveling policy (i.e., one single sorted run per level), there exists a general model and a closed-form solution when the number of sorted runs can vary across levels, see more details in Dostoevsky [17].

Monkey$^+$ (an Extension of Monkey for Imprefectly-Shaped LSM-trees). The closed-form solutions discussed above in Monkey and Dostoevsky assume the LSM-tree is *perfectly-shaped.* However, LSM-trees do not always have a perfect shape. As mentioned earlier, compactions in LSM-trees are not necessarily triggered by the level capacity [31, 50, 51], and each compaction may select more than one file to compact [20, 49]. Besides, the entry size can also vary across levels even if the size of each level grows exponentially. As such, most of the time, we need to deal with an *imperfectly-shaped* LSM-tree. Monkey has an extended version Monkey$^+$ that takes as input a list that stores the number of entries in each level [15]. To solve the optimization problem in Eq. (3), Monkey$^+$ designs a gradient descent algorithm with complexity of $O(L^2 \cdot \log M)$ where L is the number of levels and M is the memory budget in bits.

3 WORKLOAD-AWARE BITS-PER-KEY ALLOCATION FOR A STATIC LSM-TREE

We now study the optimal workload-aware bits-per-key allocation model for a *static* LSM-tree. 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 corresponding files are newly created and have not received any queries yet. Thus, this section provides a theoretically optimal bits-per-key assignment that provides a **headroom** of improvement for our practical algorithm later.

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 for a given workload). Note that all unnecessary accesses to a file i are triggered by false positive results of the BF for empty queries z_i. As such, we only need to add z_i as a coefficient for each BF in Eq. (3) to build our new objective function:

$$Cost(\{\epsilon_i\}) = \sum_{i=1}^{F} z_i \cdot \epsilon_i, \tag{5}$$

where F is the number of total BFs (files). Note that if a file receives no empty queries ($z_i = 0$), it does not affect the objective function, and it would also not benefit from a BF. We treat this as a special case, and 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 re-allocate the BF budget for those files to files with $z_i > 0$. We formalize the problem statement as follows:

$$\begin{aligned} \min_{\epsilon_1, \epsilon_2, \ldots, \epsilon_F} \quad & Cost(\{\epsilon_i\}) \\ \text{subject to} \quad & -\frac{1}{(\ln 2)^2} \cdot \sum_{i=1}^{F} n_i \cdot \ln \epsilon_i \leq M \\ & 0 < \epsilon_i \leq 1, \forall i \in [F] \end{aligned} \tag{6}$$

Note that we now want to find F false positive rates (one per file), as opposed to L false positive rates in Monkey/Monkey$^+$ (one per level). 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 point when we discuss the efficiency of solving the optimization problem. Similar to prior work [14, 15, 54], we can use a *Lagrangian* multiplier for the memory constraint to obtain:

$$\epsilon_i = \frac{n_i}{z_i} \cdot e^C \text{ where } C = -\frac{M \cdot (\ln 2)^2 + \sum_{i=1}^{F} n_i \cdot \ln \frac{n_i}{z_i}}{\sum_{i=1}^{F} n_i} \tag{7}$$

Note that we cannot enforce the constraint $\epsilon_i \leq 1, \forall i \in [F]$ in Eq. (7), so 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. 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. (7) for the remaining files. We discuss how to develop this algorithm in the next section.

3.2 Ordered Property & BF Allocation

To identify which files can be tuned using Eq. (7), we use a property they have, which derives from 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. (7) is applicable to a file. Specifically, such files are contiguous if we sort them based on 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 3.1 (Ordered Theorem). *In the optimal solution of the objective function in Eq. (6), 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 3.1 using duality. We first write the *Lagrangian* function $\mathcal{L}$ as follows:

$$\mathcal{L}(\epsilon_1, \epsilon_2, ..., \epsilon_F, \lambda, \nu_1, \nu_2, ..., \nu_F) = Cost(\{\epsilon_i\}) + \lambda \cdot \left(-\frac{1}{(\ln 2)^2} \cdot \sum_{i=1}^{F} n_i \ln \epsilon_i - M \right) + \sum_{i=1}^{F} \nu_i \cdot (\epsilon_i - 1), \tag{8}$$

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:

$$g(\lambda, \nu_1, ..., \nu_F) = \inf_{\epsilon_1, \epsilon_2, ..., \epsilon_F} \mathcal{L}(\epsilon_1, \epsilon_2, ..., \epsilon_F, \lambda, \nu_1, \nu_2, ..., \nu_F) \tag{9}$$

Next, we show this is a convex problem because the only nonlinear constraint specifies a convex feasible region. For any two ϵ_{i_1} and ϵ_{i_2} that satisfy $-\frac{1}{(\ln 2)^2} \cdot n_i \cdot \ln \epsilon \leq M$, we always have:

$$-\frac{1}{(\ln 2)^2} \cdot n_i \cdot \ln \epsilon_\delta \leq -\frac{1}{(\ln 2)^2} \cdot n_i \cdot \left(\sigma \cdot \ln \epsilon_{i_1} + (1 - \sigma) \cdot \ln \epsilon_{i_2}\right) \leq M$$

where $\epsilon_\delta = \delta \cdot \epsilon_{i_1} + (1 - \delta) \cdot \epsilon_{i_2}$ for any $\delta \in [0, 1]$. The above inequality holds because $-\ln \epsilon$ is a convex function for ϵ. Now, we use Slater's condition to show that strong duality holds. 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 $\text{bpk} = 0$. 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. (6) has *strong duality*. Therefore, the optimal solution $\epsilon_1, \epsilon_2, ..., \epsilon_F, \nu_1, \nu_2, ..., \nu_F$ of Eq. (9) should satisfy:

$$\begin{cases} \nu_i > 0, & \text{if } \epsilon_i = 1 \\ \nu_i = 0, & \text{otherwise (i.e., } \epsilon_i < 1) \end{cases} \tag{10}$$

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:

$$z_i - \frac{\lambda \cdot n_i}{(\ln 2)^2 \cdot \epsilon_i} + \nu_i = 0 \tag{11}$$

In the case of $\epsilon_i = 1$, we have:

$$\frac{\lambda \cdot n_i}{(\ln 2)^2} - z_i = \nu_i > 0 \Rightarrow \frac{\lambda}{(\ln 2)^2} > \frac{z_i}{n_i}$$

Otherwise, we have $\epsilon_i < 1$ and thus:

$$\frac{\lambda \cdot n_i}{(\ln 2)^2 \cdot \epsilon_i} - z_i = 0 \Rightarrow \epsilon_i = \frac{\lambda \cdot n_i}{(\ln 2)^2 \cdot z_i} \Rightarrow \frac{\lambda}{(\ln 2)^2} < \frac{z_i}{n_i}$$

As such, in the optimal solution of λ and $\{\epsilon_i\}$, we always have:

$$\begin{cases} C > \frac{z_i}{n_i} & if\, \epsilon_i = 1 \\ C < \frac{z_i}{n_i} & if\, \epsilon_i < 1 \end{cases}, \tag{12}$$

where $C = \lambda/(\ln 2)^2$, obtained by Eq. (7) 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_i than deeper levels, which is consistent with Theorem 3.1 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 3.1. 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

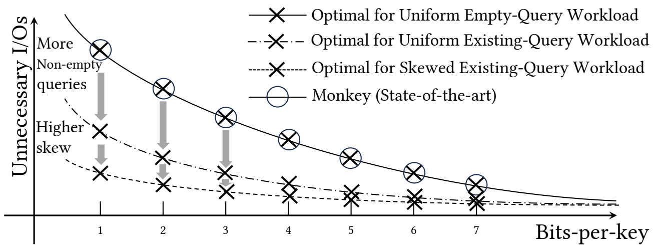


Figure 2: A conceptual graph that compares the state-of-the-art bits-per-key allocation solution from Monkey and the solution from our workload-aware model.

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 4), we thus only implement the linear scan algorithm in Mnemosyne.

Implementation. In practical systems such as RocksDB, when $\text{bpk} \leq 1$, the assigned bits-per-key is actually `round(bpk)`. For example, if $\text{bpk} < 0.5$, 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 6 of Algorithm 1 with $\bullet > e^{-(\ln 2)^2}$ (the false positive rate $\epsilon = e^{-(\ln 2)^2}$ when $\text{bpk} = 1$), and then we naturally have $\text{bpk}_i \geq 1$ if $\text{bpk}_i > 0$ in the optimal solution (the proof is similar to the proof for the constraint $\epsilon \leq 1$).

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

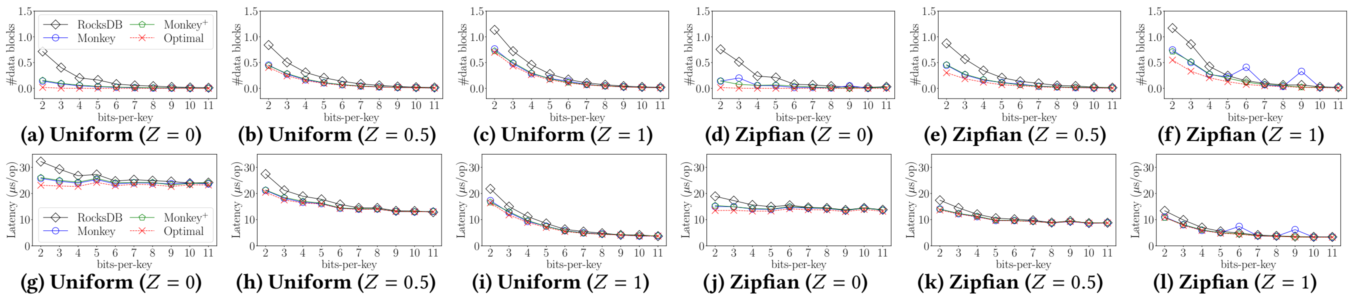
```
1  $\mathcal{Z} \leftarrow \{z_i\};\ \mathcal{N} \leftarrow \{n_i\};\ result \leftarrow \{\}$ ;
2  Initialize a pair vector $V$ so that $V[i] = (\frac{z_i}{n_i}, i)$
3  Sort $V$ according to $\frac{z_i}{n_i}$ in a descending order
4  $S_1 \leftarrow \sum_{i=1}^{F} n_i;\ S_2 \leftarrow \sum_{i=1}^{F}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

3.3 Headroom for Improvement

To investigate the headroom for improvement, we conduct a micro-benchmark that compares the number of accessed data blocks among the default BF allocation strategy in RocksDB (the same bpk for all BFs), the closed-form solution in Monkey (which assumes that all levels have their nominal size), Monkey$^+$ (that allows imperfect LSM shapes), and the optimal one obtained by our model in Eq. (6). Further, since the gradient descent algorithm by Monkey$^+$ can also be used to solve Eq. (6), we further compare the execution time between the gradient descent algorithm and Algorithm 1.

Experimental Methodology. We populate an LSM-tree with $2M$ 512-byte entries (1GB) 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 8KB, and the number of entries per file as 8192 (4MB per file), which produces a 3-level LSM-tree. Then we allocate a 32MB block cache, which ensures all the filters and indexes fit in the cache except the data blocks, and we execute a read workload of $4M$ point queries, which consists of either only existing queries ($Z = 0$) or only empty queries ($Z = 1$). Having 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), we then build another LSM-tree that results from the same sequence of compactions but has different bpk_i per file. We vary the bpk_i based on the decisions of the four strategies. For Monkey, bpk_i is obtained using the closed-form solution Eq. (4). For Monkey$^+$ and the optimal strategy, we calculate bpk_i by solving the associated optimization problem with Algorithm 1. Then, we execute the same query workload again on top of the new LSM-tree and measure the execution time (with direct I/O turned on) and the number of accessed data blocks. Finally, we report the average access latency per query and 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 repeat the above procedure by varying overall bpk from 2 to 11. The experimental results are summarized in Figure 3.

Observations. In Figures 3a, 3c, 3d, 3f, while Monkey/Monkey$^+$ reduce a lot of unnecessary data block accesses compared with RocksDB for all kinds of workloads, **the optimal strategy further reduces the number of unnecessarily accessed data blocks** when the workload contains only existing queries or exhibits higher skew. Similar patterns can be observed in the latency comparison in Figures 3g, 3i, 3j, 3l. As expected, when all point queries are empty queries ($Z = 1$) and are uniformly distributed (two assumptions in Monkey/Monkey$^+$), the optimal solution is close to what Monkey/Monkey$^+$ propose, as shown in Figures 3c and 3i. Note the benefit of the optimal over Monkey/Monkey$^+$ is more pronounced for fewer bits-per-key (i.e., between 2 and 6), which indicates that the optimal strategy will have a higher benefit in scenarios with memory pressure. Further, although Monkey and Monkey$^+$ have very similar performance (the green and the blue lines nearly overlap in most figures), Monkey$^+$ is more stable (the green line is smoother than the blue line) because Monkey$^+$ does not assume a perfect LSM shape. Specifically, the spikes for Monkey in Figures 3f and 3l show that Monkey can be occasionally worse than RocksDB. This typically happens when all files in Level 1 are compacted (RocksDB may select multiple files to reduce write amplification [49]), resulting in an empty Level 1 and, thus, a *imperfectly-shaped* LSM-tree. Note that, in the closed-form solution of Monkey, each level is supposed to be close to full and consume $1/L$ of the



(a) Uniform ($Z = 0$) (b) Uniform ($Z = 0.5$) (c) Uniform ($Z = 1$) (d) Zipfian ($Z = 0$) (e) Zipfian ($Z = 0.5$) (f) Zipfian ($Z = 1$)

(g) Uniform ($Z = 0$) (h) Uniform ($Z = 0.5$) (i) Uniform ($Z = 1$) (j) Zipfian ($Z = 0$) (k) Zipfian ($Z = 0.5$) (l) Zipfian ($Z = 1$)

Figure 3: The average number of unnecessarily accessed data blocks and the latency per query for different workloads where 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).

memory budget for BFs. When Level 1 becomes empty, its memory is not redistributed to other levels, meaning that the overall BF size is smaller (and, thus, the queries are slower) than the baseline strategy. Since Monkey$^+$ is always superior to Monkey, we omit Monkey in the rest of this paper. We only compare our method with Monkey$^+$ in our experiments and refer to Monkey$^+$ as Monkey.

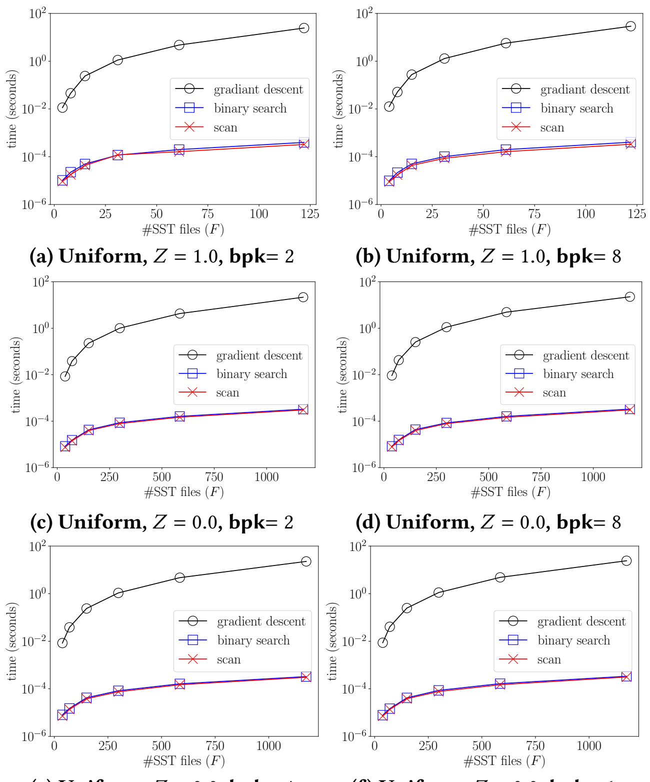
Efficiency of the Optimization Solver. We also examine the difference of the execution time between the gradient descent algorithm (proposed by Monkey) and our sort-and-search algorithms (including both linear-scan and binary-search version). We fix the SST file size as 32MB and the size ratio as 4, 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 4. As shown in Figures 4c and 4d, 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 flush and compaction, given that the median compaction latency is just one second [51] when populating a 10GB LSM-tree with a 4000 IOPS-provisioned SSD (with direct I/O). Since our algorithms only take around 0.1 ms when $F \approx 1K$, our algorithms are more practical to be deployed with negligible overhead during compactions.

4 ACCESS STATISTICS ESTIMATION

Now that we have seen the headroom for improvement, assuming we have perfect statistics, we present how to estimate at runtime the number of zero-result (empty) queries per file (z_i) in LSM-trees, which is necessary to instantiate our workload-aware bits-per-key model in Eq. (6). Note that keeping track of the exact z_i per file can lead to a huge maintenance overhead, and thus, we can only *estimate* z_i. To achieve this, we estimate the number of queries q_i and the number of existing queries x_i, and approximate z_i as $q_i - x_i$.

4.1 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 [37] to monitor $\{q_i\}$). In this example, shown in Figure 5, 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.,

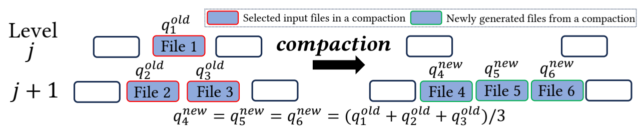


(a) Uniform, $Z = 1.0$, bpk$= 2$ (b) Uniform, $Z = 1.0$, bpk$= 8$

(c) Uniform, $Z = 0.0$, bpk$= 2$ (d) Uniform, $Z = 0.0$, bpk$= 8$

(e) Uniform, $Z = 0.0$, bpk$= 4$ (f) Uniform, $Z = 0.0$, bpk$= 6$

Figure 4: Our algorithms (both binary searching and scan) are faster than gradient descent by $2 \sim 4$ magnitude.



Selected input files in a compaction
Newly generated files from a compaction
Level
j
$j+1$
q_1^{old}
File 1
q_2^{old}
File 2
q_3^{old}
File 3
compaction
q_4^{new}
File 4
q_5^{new}
File 5
q_6^{new}
File 6
$q_4^{new} = q_5^{new} = q_6^{new} = (q_1^{old} + q_2^{old} + q_3^{old})/3$

Figure 5: An illustration of the inheritance mechanism in the naïve strategy. $q_{i_1}^{old}$, $q_{i_2}^{new}$ denote the number of counted queries for an input File i_1, and the estimated number of queries for a newly generated File i_2, respectively.

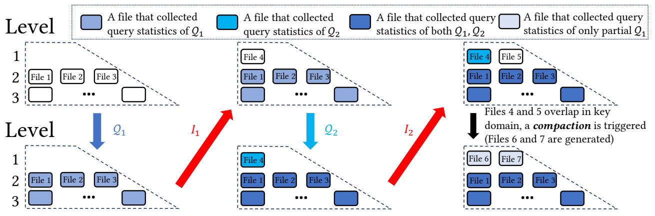

Level
A file that collected query statistics of Q_1
A file that collected query statistics of Q_2
A file that collected query statistics of both Q_1, Q_2
A file that collected query statistics of only partial Q_1
1
2
3
File 1
File 2
File 3
File 4
File 5
File 6
File 7
Q_1
I_1
Q_2
I_2
Files 4 and 5 overlap in key domain, a *compaction* is triggered (Files 6 and 7 are generated)

Figure 6: 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.

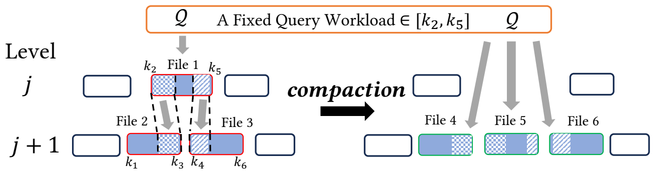


Figure 7: In the naïve approach, most empty queries in File 1 (z_1) are counted twice when estimating q_i^{new} ($i = 4, 5, 6$).

$q_4^{new}, q_5^{new}, q_6^{new}$) 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 6, we consider a scenario where we start with the left-top LSM-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. 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 7 (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 7. 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.

4.2 Merlin

We now introduce Merlin, a tracking and estimation mechanism of the number of zero-result point queries per file. While we use accurate file statistics for memory allocation, they can also benefit other decisions like caching and compaction priority.

4.2.1 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 §4.2.2).

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 per file to estimate the associated access interval. Specifically, for File i, we maintain a w-length window, which stores the last w sequence numbers that accessed File i, noted by $\Omega^{(i)} = \{D_1^{(i)}, ..., D_w^{(i)}\}$. This window is essentially a First-In-First-Out queue, and thus we have $D_1^{(i)} > \cdots > D_w^{(i)}$ in $\Omega^{(i)}$. Logically, $\Omega^{(i)}$ records $w-1$ intervals during the last w accesses, noted by $\{\Delta_1^{(i)}, .., \Delta_{w-1}^{(i)}\}$ ($\Delta_l^{(i)} = D_l^{(i)} - D_{l+1}^{(i)}$ for $l \in [1, w-1]$). In addition, we also maintain a counter q_i' that stores the number of queries before $D_w^{(i)}$ (q_i' does not count the number of point queries in $\Omega^{(i)}$). When File i is accessed by a query D^{global}, we first check if $\Omega^{(i)}$ is full. If not, we just need to push D^{global} into $\Omega^{(i)}$. Otherwise, we evict the oldest query $D_w^{(i)}$ and increase q_i' by 1, before we push D^{global}. To estimate the access interval of a File i during the last w accesses, we use a similar idea in φ failure detector [29] (applied in Cassandra [4]). We assume that the access interval for File i between every two adjacent accesses (noted by $\Delta^{(i)}$) follows an exponential distribution with parameter ζ_i, that is, $\Delta^{(i)} \sim Exp(\zeta_i)$ (ζ_i can vary for different files). Then, the Maximum Likelihood Estimation (MLE) of ζ_i using the last observed $w-1$ intervals is $(w-1)/\sum_{l=1}^{w-1} \Delta_l^{(i)}$. Since $E[\Delta^{(i)}] = 1/\zeta_i$, $\Delta^{(i)}$ can thus be estimated as $\sum_{l=1}^{w-1} \Delta_l^{(i)}/(w-1)$ using the last w accesses. When calculating $E[\Delta^{(i)}]$, we can replace $\sum_{l=1}^{w-1} \Delta_l^{(i)}$ with $D_1^{(i)} - D_w^{(i)}$ by definition. In addition, we can also estimate the access interval for queries older than $D_w^{(i)}$ by $D_w^{(i)}/(q_i' + 1)$. Combining these two estimated intervals using β ($\beta \in [0, 1]$ which controls how aggressively the interval estimation adapts to the last w accesses in case of workload shifting), we have:

$$\Delta_{est}^{(i)} = \beta \cdot \frac{D_1^{(i)} - D_w^{(i)}}{w-1} + (1-\beta) \cdot \frac{D_w^{(i)}}{q_i' + 1}$$

Finally, we estimate q_i by $D^{global}/\Delta_{est}^{(i)}$.

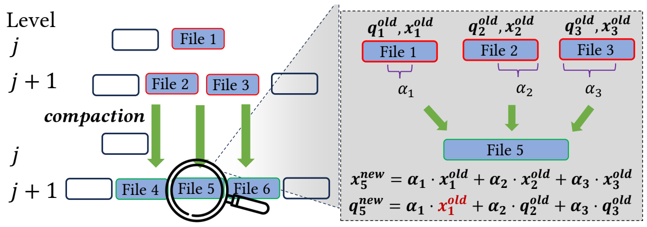

Level
j
j + 1
File 1
File 2
File 3
compaction
j
j + 1
File 4
File 5
File 6
q_1^{old}, x_1^{old}
q_2^{old}, x_2^{old}
q_3^{old}, x_3^{old}
File 1
File 2
File 3
α_1
α_2
α_3
File 5
$x_5^{new} = \alpha_1 \cdot x_1^{old} + \alpha_2 \cdot x_2^{old} + \alpha_3 \cdot x_3^{old}$
$q_5^{new} = \alpha_1 \cdot x_1^{old} + \alpha_2 \cdot q_2^{old} + \alpha_3 \cdot q_3^{old}$

Figure 8: 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.

Estimation of x_i. Based on $\Omega^{(i)}$ and q'_i, we use a 64-length bitmap x_i^w and an additional counter x'_i, where each bit in x_i^w denotes whether the associated query in $\Omega^{(i)}$ is non-empty or not, and x'_i is the number of non-empty queries in q'_i (the number of queries older than $D_w^{(i)}$). We estimate x_i as follows:

$$x_i = \left(\beta \cdot \frac{\texttt{\_\_builtin\_popcount}(x_i^w)}{w} + (1-\beta) \cdot \frac{x'_i}{q'_i}\right) \cdot q_i,$$

where `__builtin_popcount`(x_i^w) is a built-in function in `gcc` compiler that returns the number of non-zero bits for a given integer. We also limit the maximum window size to 64, and thus a 64-bit x_i^w is sufficient to record all the query results in $\Omega^{(i)}$.

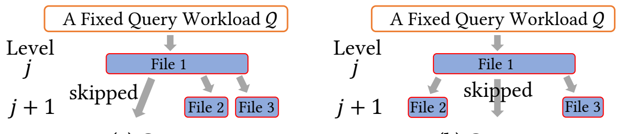
Workload Shifting. The workload characteristics (e.g., the proportion of empty queries, the distribution of queries) may change over time [7, 30, 42], and the old statistics thus become outdated if we only use the counter to estimate q_i and x_i. ElasticBF has a threshold *expiredTime*, which specifies when the statistics become invalid, however, this approach hastily eliminates statistics older than *expiredTime*, which could result in inaccurate estimation. The sliding window employed by Merlin keeps track of most-recently accessed queries, and thus it can naturally adapt to shifting workloads.

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 $\Omega^{(i)}$ 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 $\Omega^{(i)}$ and q'_i, x'_i, x_i^w). In each query, the picked thread has 0.05% probability to transfer the write 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 §4.2.3), but the estimation also only takes constant time. In terms of space, we need a 64-length queue where each element (the point query sequence number) is also 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.

4.2.2 Statistics Inheritance during Compactions.

When new files are generated in compactions, we need an inheritance mechanism



(a) Case 1 **(b) Case 3**

Figure 9: Two example corner cases (when some zero-result queries are counted in level j but not counted in level $j + 1$).

to estimate q_i^{new} for them (the estimated q_i^{new}, x_i^{new} are assigned to q'_i, x'_i, all the new files start with an empty $\Omega^{(i)}$ and $x_i^w = 0$) to avoid cold start. We observe that only the non-empty queries are inherited during a compaction, according to the example shown in Figure 7 (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:

$$q_{i^*}^{new} = \sum_{i \in I_j} \alpha_i \cdot x_i^{old} + \sum_{i \in I_{j+1}} \alpha_i \cdot q_i^{old}, \tag{13}$$

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 8, 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:

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

Some Corner Cases. In our design, the new counter $q_{i^*}^{new}$ only inherits x_i in a shallower level because we assume z_i are counted twice in most cases. However, there are three corner cases in which some empty queries are only counted once in the shallower level. We list 2 example cases in Figure 9. In case 1, the smallest key in level j is much smaller than the smallest key in level $j + 1$, and a large proportion of queries targeting keys larger than the smallest key in level $j + 1$ are not collected by level $j + 1$. Similarly, we could also have such a scenario for the largest key (case 2). In case 3, the gap in the key domain in level $j + 1$ is so huge that many zero-result queries in level j skip level $j + 1$. To deal with all the above cases, we adjust Eq. (13) as follows[4]: when there is a compaction between Files 1, 2, 3 where File 1 is in a shallower level, we change x_1^{old} into $\max\{q_1^{old} - q_2^{old} - q_3^{old}, x_1^{old}\}$ when estimating $q_{i^*}^{new}$.

4.2.3 Mnemosyne: Integrating Merlin with Our Cost Model into RocksDB.

We add the queue and other counter variables into the

[4]See our implementation (https://github.com/BU-DiSC/Mnemosyne) for more details.

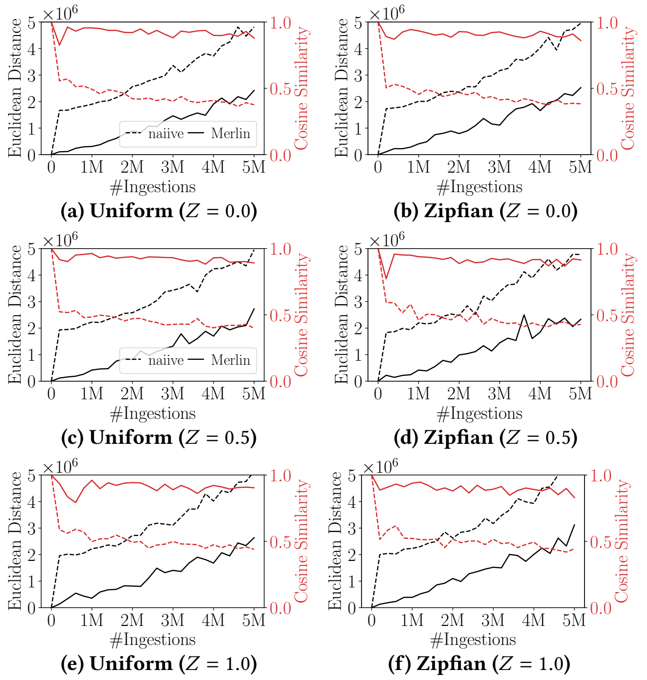
metadata of the `FileMetaData` object to implement the sliding-window-based estimation. During each compaction, we calculate q_i^{avg}, the average number of queries per key for each input File i, and we embed q_i^{avg} into the compaction iterator. When the compaction iterator adds a key-value pair 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 Alg. 1 to get C from Eq. (12). After that, we use C, n_{i^*}, and $z_{i^*} = q'_{i^*} - x'_{i^*}$ to calculate bpk_{i^*} (following line 15 in Algorithm 1). When $\text{bpk}_{i^*} \in [0.5, 1)$, we round up bpk_{i^*} to 1. If $\text{bpk}_{i^*} < 0.5$, we set $\text{bpk}_{i^*} = 0$ just as RocksDB already does. Note that we cannot rebuild BFs for other files since there might be concurrent compactions relying on them. Despite this, when executing any point query, we can still use the constant C and the statistics (n_i and estimated z_i) to decide if we need to skip a BF (to emulate $\text{bpk}_i = 0$) when accessing a file (we skip the BF if $n_i > z_i \cdot e^{-C}$). We allow C to be overwritten by concurrent compactions because they run Algorithm 1 based on nearly the same input data, and thus produce similar C.

4.3 Micro-benchmark for Estimation

Next, we compare the accuracy of the estimated number of zero-result point queries (z_i) between the naïve strategy and Merlin.

Experimental Methodology. We first populate an LSM-tree with $10M$ 1KB key-value pairs with size ratio 4 and file size 64MB. To examine the accuracy of a tracking strategy, we copy the database and execute a mixed workload with $10M$ point queries and $5M$ updates. For every $200K$ updates, we issue a manual flush and copy the entire database while resetting all statistics counters and re-executing all the point queries issued so far by maintaining a z_i counter per file. By doing this, we obtain the ground-truth 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 [37] 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 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 10, Merlin has half the Euclidean distance from the ground truth than the naïve strategy, and 2× the cosine similarity of the naïve strategy to the ground truth, supporting that **Merlin offers more accurate statistics estimation**. Another key benefit of Merlin is that by access, statistics already capture the impact of the exact LSM shape, so no assumptions have to be made (e.g., like the assumption by Monkey that every level is full), not even whether the tree follows leveling, tiering, or a hybrid layout. 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. Having said that, the cosine similarity



(a) Uniform ($Z = 0.0$) **(b) Zipfian** ($Z = 0.0$)
(c) Uniform ($Z = 0.5$) **(d) Zipfian** ($Z = 0.5$)
(e) Uniform ($Z = 1.0$) **(f) Zipfian** ($Z = 1.0$)

Figure 10: The Euclidean distance (black lines) indicates the absolute error, of which Merlin has 50% less error over the naïve tracking. Besides, when comparing the cosine similarity (red lines), Merlin is also superior to the naïve tracking.

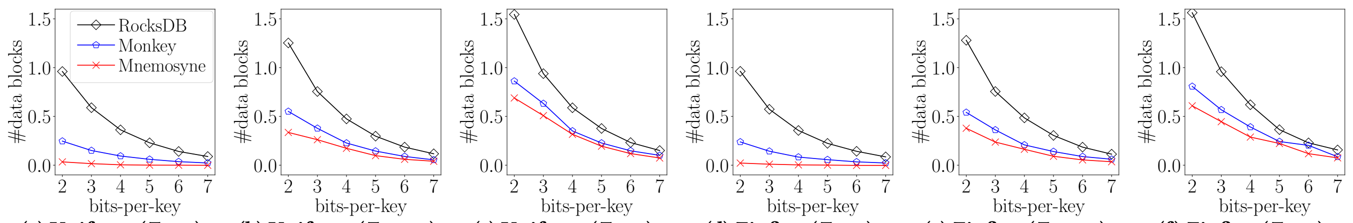
of Merlin stabilizes between 0.9 and 0.95, while the naïve one stabilizes around 0.5. Finally, we observe that for both approaches, the Euclidean distance increases with more updates while cosine similarity is more stable. This is expected since cosine similarity is normalized while Euclidean distance depends on the value (number of queries), which is ever-increasing.

5 EVALUATION

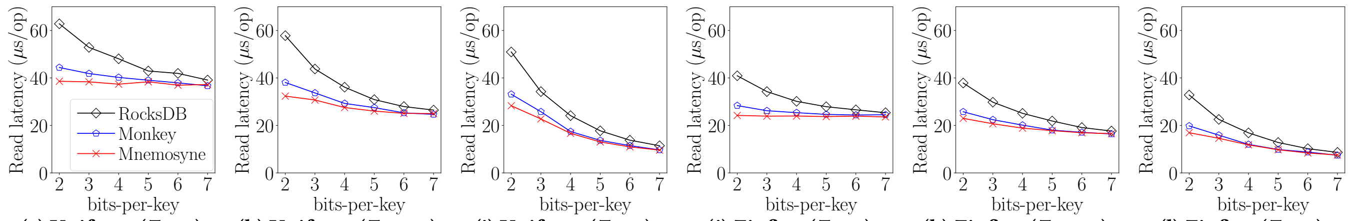
We implement Mnemosyne, which integrates Merlin with our workload-aware cost model into RocksDB (v8.9.1). We *augment* Monkey$^+$ with Algorithm 1 and implement it in RocksDB (referred to as Monkey). In this section, we implement Mnemosyne and Monkey on RocksDB, and compare them to the default bits-per-key strategy in RocksDB, which assigns the same bits-per-key per file.

Environment. We use a 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 PCIe P4510 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.

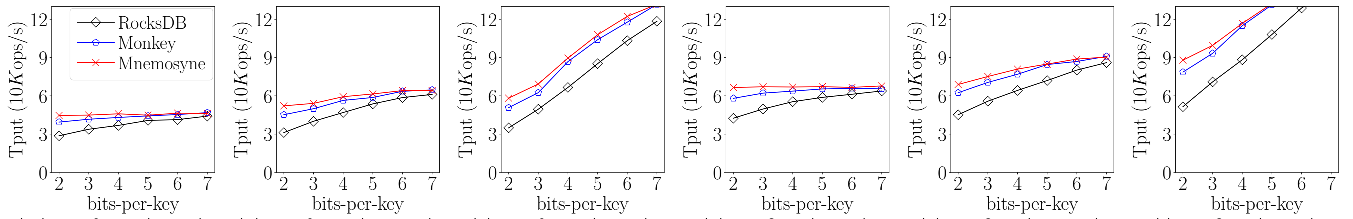
Experimental Methodology. We experiment both with synthetic data and industry-grade benchmarks like YCSB [13]. Our synthetic workload first ingests $21M$ 512-byte key-value pairs (128-byte key and 384-byte value), and then proceeds with mixed operations consisting of $31M$ point queries and $10M$ updates. With size ratio $T = 4$ and factoring in a space amplification factor of $1/T$, the total number of key-value pairs (noted by N) is $N = 21M \cdot (1 + 1/T)$ and thus the database size (noted by $\|DB\|$) is $N \cdot 512$ bytes $\approx$



(a) Uniform ($Z = 0$) **(b) Uniform ($Z = 0.5$)** **(c) Uniform ($Z = 1$)** **(d) Zipfian ($Z = 0$)** **(e) Zipfian ($Z = 0.5$)** **(f) Zipfian ($Z = 1$)**

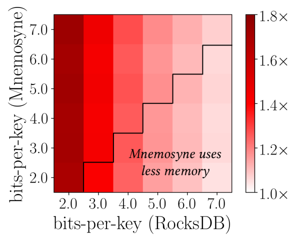


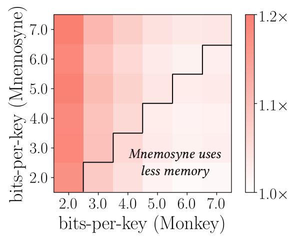
(g) Uniform ($Z = 0$) **(h) Uniform ($Z = 0.5$)** **(i) Uniform ($Z = 1$)** **(j) Zipfian ($Z = 0$)** **(k) Zipfian ($Z = 0.5$)** **(l) Zipfian ($Z = 1$)**



(m) Uniform ($Z = 0$) **(n) Uniform ($Z = 0.5$)** **(o) Uniform ($Z = 1$)** **(p) Zipfian ($Z = 0$)** **(q) Zipfian ($Z = 0.5$)** **(r) Zipfian ($Z = 1$)**

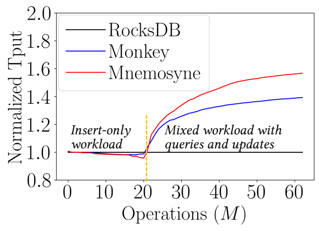
Figure 11: Mnemosyne significantly reduces the average number of unnecessarily accessed data blocks and the query latency of Monkey and RocksDB for small bits-per-key, and the reduction of accessed data blocks is more prominent when $Z = 0.0$.

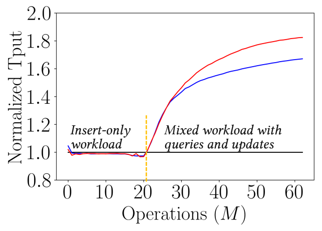




(a) Mnemosyne v.s. RocksDB **(b) Mnemosyne v.s. Monkey**

Figure 12: For Zipfian workload with $Z = 0$, Mnemosyne still outperforms RocksDB, even using $2/7$ space for BFs.



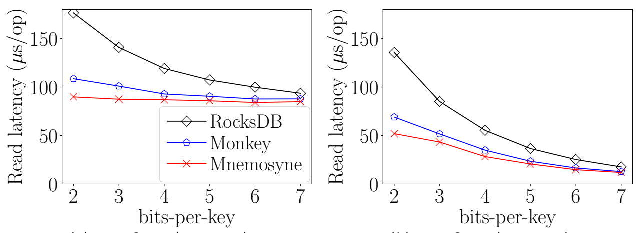


(a) Zipfian ($Z = 0.0$) **(b) Zipfian ($Z = 1.0$)**

Figure 13: Mnemosyne and Monkey have slightly worse write throughput than RocksDB, but much higher overall throughput when more queries come. Mnemosyne has more prominent improvement for empty queries ($Z = 1$).

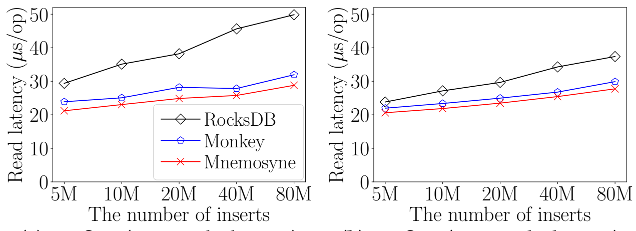
12.5GB. As many AWS EC2 instances have less than 3% memory-to-storage ratio [2], the block cache size is set as 256MB ($\approx$ 2% of the data size) to match production-level memory pressure. By default, RocksDB allocates 50% block cache exclusively for BFs and index blocks (`high_pri_pool_ratio=0.5`). We bound the index block size such that all index blocks fit in the block cache as follows: $(\|DB\|/block_{size}) \cdot index\_entry_{size} < \|block\_cache\|$. In our setup, $index\_entry_{size}$ = 140 bytes (128-byte key, 8-byte internal sequence number, and 4-byte offset of the associated data block) and the block cache for index blocks and BFs (noted by $\|block\_cache\|$), is 128MB, so $block_{size} >$ 13.7KB. We round up the block size to 16KB as it has to be a multiple of 4KB. The total size of index blocks is $(\|DB\|/block_{size}) \cdot index\_entry_{size} \approx$ 109MB, so we have 128MB − 109MB = 19MB for BFs. To ensure that all indexes and BFs fit in block cache, the maximum bpk can be calculated as the available memory in bits, divided by the number of keys, i.e., $19\text{MB}/N \cdot 8\text{bits/byte} \approx 5.96 < 6$. Further, from Figure 3 we know that neither Monkey nor the optimal strategy outperform RocksDB when bpk > 7. As such, in our experiments, we focus on bpk $\in [2, 7]$, which corresponds to a practical system setup under memory pressure.

Mnemosyne has Higher Benefit with Small bits-per-key for Non-empty Queries. We test six workloads where we vary the query distribution and the fraction of empty queries (Z), and report the average number of unnecessary data block accesses and the average latency per query in Figure 11. We first observe that Mnemosyne has higher benefit for low bits-per-key (e.g., 2 ∼ 4 for Monkey and 2 ∼ 6 for RocksDB). As the workload has fewer empty queries (moving from $Z = 1$ to $Z = 0$), Mnemosyne's benefit is pronounced for both uniform and Zipfian workloads, which is consistent with our headroom experiments in §3.3. In addition, we observe that for uniform workloads, the number of unnecessary data block accesses decreases following the query latency trend. However, when comparing Figures 11a, 11b, 11c with Figures 11g, 11h, 11i, we do not observe a similar pattern for Zipfian workloads, due to the significantly higher cache hit rates of the



(a) Zipfian ($Z = 0.0$) **(b) Zipfian ($Z = 1.0$)**

Figure 14: The latency reduction between Mnemosyne and Monkey (or RocksDB) is prominent on a slower NVM SSD.



(a) Zipfian ($Z = 0.0$, bpk = 2.0) **(b) Zipfian ($Z = 0.0$, bpk = 4.0)**

Figure 15: Mnemosyne scales better than RocksDB and remains dominant over Monkey when the database grows.

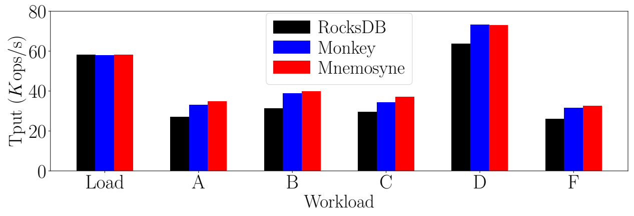


Figure 16: Mnemosyne still outperforms Monkey and RocksDB in workloads A, B, C, and F of YCSB.

frequently accessed data blocks in skewed workloads. As such, even though Mnemosyne reduces the data block accesses more than for the uniform workload, the latency benefit is not as significant.

Mnemosyne Saves Up to 66% Space Under High Workload Skew. A key benefit of Mnemosyne is that it consumes *less space than Monkey and RocksDB to offer the same or better performance*. To show this, we compare Mnemosyne with the two baselines in Figure 12 by comparing the performance benefits vs. each system bits-per-key allocation. Red and white cells indicate performance speedup and same performance, respectively. Figure 12a shows that Mnemosyne always outperforms RocksDB. Another way to read this graph is that we have the same performance even with 3.5× less memory for BFs if we focus on the bottom-right point of the graph, where bpk = 2 for Mnemosyne and bpk = 7 for RocksDB. Similarly, Figure 12b shows that, Mnemosyne with bpk = 2 can offer the same performance as Monkey with bpk = 6, thus using 3× less memory for BFs. By navigating these two graphs, we can also find points where we trade off *performance benefits* for *memory savings*, always having the same or better *raw performance*.

Write Throughput Does Not Degrade in Mnemosyne. We now examine the statistics estimation overhead that Merlin might incur during compaction. We re-run the experiment with bpk = 2 for Zipfian workloads and report the overall throughput. As shown in Figure 13, in the first 21M inserts, Mnemosyne and Monkey maintain nearly identical write throughput (only slightly worse than RocksDB) . In the next 41M operations (a mixed workload of 31M queries and 10M updates), the average throughput of Mnemosyne is significantly higher than Monkey and RocksDB for both $Z = 0$ and $Z = 1$, driven by the faster queries due to the optimal BF allocation of Mnemosyne. In addition, we also observe that the benefit of Mnemosyne over RocksDB is more prominent for $Z = 1.0$ because non-empty queries ($Z = 0$) always need to access at least one data block, leading to a smaller relative latency benefit.

Latency Benefits are Pronounced on a Slower SSD. To further study the impact of the underlying storage, we re-run the Zipfian workloads with $Z = 0$ and $Z = 1$ using a slower NVM SSD (Intel Optane 4800X), of which reading a 4KB page takes 36 μs, 2.4× slower than our default SSD. We compare the average query latency in Figure 14. Although we have a smaller I/O reduction for skewed workloads due to the higher cache hit ratio, as we already discussed, reading a data block now becomes 2.4× more expensive, and thus, we observe a larger benefit of Mnemosyne over Monkey and RocksDB. Compared with RocksDB, Mnemosyne achieves up to 2× speedup when bpk = 2, and for Monkey, Mnemosyne achieves up to 17% lower latency. Specifically, when bpk = 2, 3, 4, we have 17%, 14.6%, and 6% lower latency in Figure 14a compared to 14.5%, 8.7%, and 5.5% lower latency in Figure 11j.

Mnemosyne Dominates Monkey and RocksDB When Scales up. We scale up the number of inserts from 5M to 80M, and the number of queries (updates) from 7.5M to 120M (2.5M to 40M). We use the Zipfian workload with non-empty queries and evaluate the query latency for bpk 2 and 4. Figure 15 shows that the query latency of Mnemosyne increases slowly with the exponentially growing data size, while RocksDB's latency increases much faster. Mnemosyne benefits over Monkey scale with the data size.

Mnemosyne Remains Beneficial over Monkey and RocksDB in Most Workloads of YCSB. Now we employ YCSB [13] (use its C++ version [34]) to compare Mnemosyne, Monkey, and RocksDB in an AWS EC2 instance (*i3.4xlarge*). We set the `fieldlength` as 9 bytes (the key-value size is $24 + 10 \cdot 9 = 114$ bytes). With direct I/O turned on and block cache of 384MB, we use 4 threads to run each workload (except workload E, a range query workload) with `operationcount` of 80M and `recordcount` of 80M. The experimental results are summarized in Figure 16. We observe in Figure 16 that Mnemosyne has the same loading (insertion) throughput as Monkey and RocksDB. Furthermore, Mnemosyne outperforms RocksDB in all other workloads and dominates Monkey in workloads A, B, C, F. We do not observe significant improvement for workload D (Monkey occasionally has a bit higher throughput), because the query distribution is "`latest`", meaning that the queries mostly target recently inserted keys. Since Monkey gives most bits-per-key to newly flushed files (in level 0, the smallest level), this strategy is a good fit for Workload D. Note that Mnemosyne naturally does that for this workload based on the collected file access statistics.

6 RELATED WORK

Memory Allocation in LSM-trees. In addition to memory allocation within BFs, memory can also be re-allocated among BFs, fence pointers, and the write buffer [15, 33]. Further, memory can be further re-allocated between multiple LSM-trees [39, 40] in LSM-based storage systems. In fact, our sort-and-search algorithm (§3.2) to find the optimal size 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 [47], Cuckoo Filter [25], Quotient Filter [46], Morton Filter [9], Vacuum Filter [56], Xor Filter [28], Ribbon Filter [22], and InfiniFilter [16]. Most filters are designed based on fingerprints, and their FPR are typically linear to $2^{-l} = e^{-(\ln 2)\cdot l}$ where l can be also termed as bpk. 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 [19] and SlimDB [48] 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) [54] mainly targets finding how to partition the key space and how to allocate bits-per-key per partition to find the optimal FPR with constrained memory. Its optimization algorithm searches the optimal FPR in a heuristic way which may result in a sub-optimal solution. As such, our sort-and-search algorithm can also be used to augment PLBF.

Skew-Aware Key-Value Stores. Existing skew-aware read optimizations in LSM-trees mostly focus on the caching policy [58, 60] or extra memory for frequently queried keys [61]. When the available memory is insufficient, a BF can be broken into smaller BFs (e.g., ElasticBF [37] and ModularBF [44]), so that we can avoid loading the entire BF into memory. We highlight that all these techniques can co-exist with Mnemosyne. In addition, there exist skew-aware techniques in hash-table-based and purely in-memory key-value stores [35, 36, 38], which are customized for the associated data structures and, thus, cannot be directly applied to LSM-trees.

7 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 design an algorithm that finds the optimal solution with negligible CPU overhead. To apply our model in real systems, we further design Merlin, a novel and accurate query statistics tracking mechanism. We implement Mnemosyne by integrating Merlin and our cost model into RocksDB. With limited memory, Mnemosyne achieves up to 2× improvement over the production-ready RocksDB and up to 15% improvement over Monkey for query workloads that exhibit skew or contain more non-empty queries.

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. https://doi.org/10.14778/2733085.2733096

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

[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. https://doi.org/10.1145/2254756.2254766

[8] Burton H Bloom. 1970. Space/Time Trade-offs in Hash Coding with Allowable Errors. *Commun. ACM* 13, 7 (1970), 422–426. https://doi.org/10.1145/362686.362692

[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. https://doi.org/10.14778/3213880.3213884

[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. https://doi.org/10.5555/1267308.1267323

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

[13] 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. https://doi.org/10.1145/1807128.1807152

[14] 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. https://doi.org/10.1145/3035918.3064054

[15] 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

[16] Niv Dayan, Ioana O Bercea, Pedro Reviriego, and Rasmus Pagh. 2023. InfiniFilter: Expanding Filters to Infinity and Beyond. *Proc. ACM Manag. Data* 1, 2 (2023), 140:1—-140:27. https://doi.org/10.1145/3589285

[17] 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. https://doi.org/10.1145/3183713.3196927

[18] 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. https://doi.org/10.1145/3299869.3319903

[19] 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

[20] 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

[21] 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. https://doi.org/10.1145/1323293.1294281

[22] 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) (LIPIcs)*, Vol. 233. 4:1—-4:20. https://doi.org/10.4230/LIPIcs.SEA.2022.4

[23] Facebook. 2021. RocksDB. *https://github.com/facebook/rocksdb* (2021).

[24] Facebook. 2023. MyRocks. *http://myrocks.io/* (2023).

[25] 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. https://doi.org/10.1145/2674005.2674994

[26] 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. https://doi.org/10.1145/2741948.2741973

[27] Google. 2021. LevelDB. *https://github.com/google/leveldb/* (2021).

[28] 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

[29] Naohiro Hayashibara, Xavier Défago, Rami Yared, and Takuya Katayama. 2004. The \(Φ\) Accrual Failure Detector. In *23rd International Symposium on Reliable Distributed Systems (SRDS 2004), 18-20 October 2004, Florianpolis, Brazil*. 66–78. https://doi.org/10.1109/RELDIS.2004.1353004

[30] 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. https://doi.org/10.1109/ICDEW.2010.5452738

[31] 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. https://doi.org/10.1145/3299869.3314041

[32] Andy Huynh, Harshal A. Chaudhari, Evimaria Terzi, and Manos Athanassoulis. 2024. https://doi.org/10.1007/s00778-023-00826-9. *The VLDB Journal* Towards fl (2024). https://doi.org/10.1007/s00778-023-00826-9

[33] 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

[34] Ren Jinglei, Kjellqvist Chris, and Deng Long. [n.d.]. YCSB-C. ([n. d.]). https://github.com/basicthinker/YCSB-C

[35] 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

[36] 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://doi.org/10.14778/3574245.3574275

[37] 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.

[38] 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

[39] 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://doi.org/10.1145/3318464.3384399

[40] 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. https://doi.org/10.5555/3430915.3442425

[41] Chen Luo and Michael J. Carey. 2020. LSM-based Storage Techniques: A Survey. *The VLDB Journal* 29, 1 (2020), 393–418. https://doi.org/10.1007/s00778-019-00555-y

[42] 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. https://doi.org/10.1145/3183713.3196908

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

[44] 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)*. https://adms-conf.org/2022-camera-ready/ADMS22{_}mun.pdf

[45] 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. https://doi.org/10.1007/s002360050048

[46] 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

[47] Felix Putze, Peter Sanders, and Johannes Singler. 2009. Cache-, hash-, and space-efficient bloom filters. *ACM Journal of Experimental Algorithmics* 14 (2009). https://doi.org/10.1145/1498698.1594230

[48] 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. https://doi.org/10.14778/3151106.3151108

[49] RocksDB. 2023. Expanding Picked Files Before Compaction. (2023). https://github.com/facebook/rocksdb/blob/8.9.fb/db/compaction/compaction{_}picker.cc{#}L497

[50] 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. https://doi.org/10.1145/3318464.3389757

[51] 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. https://doi.org/10.14778/3476249.3476274

[52] 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. https://doi.org/10.1145/2213836.2213862

[53] 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. https://doi.org/10.1109/SURV.2011.031611.00024

[54] 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

[55] 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. *Proc. ACM Manag. Data* 1, 2 (2023), 195:1—-195:27. https://doi.org/10.1145/3589775

[56] 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. https://doi.org/10.14778/3364324.3364333

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

[58] 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

[59] Xingbo Wu, Yuehai Xu, Zili Shao, and Song Jiang. 2015. LSM-trie: An LSM-tree-based Ultra-Large Key-Value Store for Small Data Items. In *Proceedings of the USENIX Annual Technical Conference (ATC)*. 71–82. https://www.usenix.org/conference/atc15/technical-session/presentation/wu

[60] 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.

[61] 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

[62] 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.