Dynamic Workload-Aware BF Tuning via Accurate Statistics Estimation 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 decent 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**) [43] have emerged as the core data structure in most modern key-value stores [1–4, 10, 11, 20, 22, 23, 25, 26, 30, 50, 53, 55]. LSM trees are widely adopted because they achieve high ingestion throughput via an *out-of-place* update strategy. With this paradigm, data ingestion operations (including inserts, deletes, and updates) are buffered in memory and eventually flushed to disk as a *sorted immutable run* whenever the buffer fills up. While LSM trees use

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

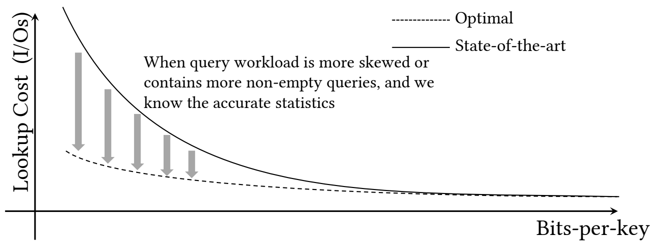


Figure 1: When the query workload becomes more skewed or has more non-empty queries, we can achieve lower lookup cost by optimally assign bits-per-key for each BF if we know the exact number of empty queries per BF, especially when the system has limited memory (i.e., smaller bits-per-key).

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 [39].

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 [26] and RocksDB [22], 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 [42].

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 [5, 41, 42]. 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, while a BF with 8 bits-per-key needs at least 5% memory-to-storage ratio for 20-byte key-value pairs [6]. As a result, the bits-per-key for BFs has to be configured low 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 [6, 9, 12], 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 [14], while a more accurate cost model has been recently proposed for holistic tuning [31]. 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 [60]. 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 [13, 16, 18] 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 [19, 49]. 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 [49], multiple files can be selected to be compacted together into the next level (e.g., the expansion mechanism in RocksDB [47], or partition-based compactions for deeper levels in Spooky [19]). 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 [49], instead, they can also be triggered by the number of tombstones, and the expiring tombstones with specified Time-To-Live (TTL) [30, 48]. 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 which does not rely on perfectly-shaped LSM trees and it uses a gradient descent algorithm to find the optimal bits-per-key assignment [14], 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 properly allocate memory for it, 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, besides 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. Therefore, Mnemosyne only relies on the user-provided bits-per-key without assuming a perfectly shaped tree. 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, considering limited memory. To address Challenges 1-3, the new model takes into account workload skew and empty queries per file with a limited memory budget, without assuming a perfectly-shaped LSM tree (§3.1).

[1]A wizard who could foresee the outcome of battles, in Celtic mythology.
[2]The ancient Greek goddess of memory.

- 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 [39, 43], and then we discuss the state-of-the-art cost models used for optimal memory allocation in LSM trees [13]. 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 the i^{th} file (the j^{th} level[3])
$n_i (n_j)$	#entries in the i^{th} file (the j^{th} level)
$\text{bpk}_i (\text{bpk}_j)$	bits-per-key for the i^{th} file (the j^{th} level)
z_i	#zero-result (empty) queries for the i^{th} file
x_i	#existing (non-empty) queries for the i^{th} file
q_i	#queries for the i^{th} file ($q_i = z_i + x_i$ by definition)
m_i	the filter size of i^{th} file ($m_i = n_i \cdot \text{bpk}_i$)
$\mathcal{Q}$	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 [16, 17, 19, 46, 49, 57]. 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 [17, 49] 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 also utilize Bloom Filters (BFs) [7, 51] to accelerate point lookups. Similar to index blocks, each SST file has filter blocks that store the associated BF (filter blocks are often prefetched in the block cache). The Bloom filter (per file) is queried before accessing any data blocks to determine 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 each 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 (noted by ϵ). Formally, by selecting the optimal $k_{opt} = \lceil m/n \cdot \ln 2 \rceil$, the false positive rate ϵ is:

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

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

2.2 Monkey

A Point Lookup Cost Model for Empty Queries. While the default setting in many LSM-based key-value stores (e.g., LevelDB and RocksDB) uses the same bits-per-key for all the files (we also call this by uniform assignment), Monkey [13] proposes that the uniform strategy yields sub-optimal point query performance. Instead, Monkey aims to minimize #data blocks accessed by empty queries, noted by $Cost(\{\epsilon_j\})$. In the leveling case of Monkey, the expected number of data blocks accessed by an empty query is the sum of the FPRs of BFs accessed 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:

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

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. In the origin model of Monkey, the given LSM tree needs to be *perfect*, that is, #entries in each level grows exponentially with a constant 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:

$$Cost_{min} \overset{def}{=} \min\left\{Cost(\{\epsilon_j\})\right\} = e^{-\text{bpk}\cdot(\ln 2)^2 \cdot T^{Y}} \cdot \frac{T^{\frac{T}{T-1}}}{T-1} + Y,$$

$$\epsilon_j = \begin{cases} 1, & \text{for } j > L - Y \\ (Cost_{min} - Y) \cdot \frac{T-1}{T}, & \text{for } j = L - Y \\ (Cost_{min} - Y) \cdot \frac{T-1}{T} \cdot \frac{1}{T^{L-Y-j}}, & \text{for } 1 \le j < L - Y \end{cases}, \tag{4}$$

where Y specifies that the false positive rate ϵ_j for the deepest Y levels converges to 1 (the associated $\{\text{bpk}_j\}$ for these levels are set to 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 compaction policy (i.e., one single sorted run per level), there exists a general model and a closed-form solution for a Fluid LSM tree (K runs are allowed in the first $L-1$ level, and Z runs are allowed in the last level where $K, Z \in [1, T-1]$), see more details in Dostoevsky [16].

Monkey+ (an Extension of Monkey for Imprefectly-Shaped LSM trees). Both closed-form solutions in Monkey and Dostoevsky assumes the LSM tree is *perfectly-shaped* (i.e., #entries per level grows exponentially by a constant size ratio T). However, LSM trees are not always in the ideal perfect shape. As mentioned earlier, compactions in LSM trees are not necessarily triggered by the level capacity [30, 48, 49], and each compaction may select more than one file to compact due to different optimizations [19, 47]. Besides, the entry size can also vary across levels even if the size of each level grows exponentially. As such, most of time, we need to deal with an *imperfectly-shaped* LSM tree. Monkey proposes an extension Monkey+ that takes as input a list that stores #entries in each level. To solve the optimization problem in Eq (3), Monkey+ designs a gradient decent 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 discuss our workload-aware bits-per-key allocation model for a static LSM tree, assuming we know all the 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 it is **unrealistic** to know the exact query statistics (i.e., $\{z_i\}$ and $\{q_i\}$) when building filters, because filters are constructed when the associated files are being created before any query can access. This is why we consider it a theoretically optimal algorithm to decide the bits-per-key assignment, which helps us understand the lower bound of any feasible bits-per-key allocation strategy but cannot be converted into a practical algorithm.

3.1 Problem Definition

We aim to minimize #data blocks that are unnecessarily accessed, which is equivalent to minimizing accessed #data blocks in total, because #data blocks that are necessary to access is constant (i.e., the number of existing point queries) for a given workload, regardless of data obtained from the write buffer. Note that all unnecessary accesses to a file i are triggered by false positive result of the BF for empty point 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 files in a static LSM tree. In addition, the memory constraint specifies that the summed BF size should be no more than $M(M = \text{bpk} \cdot \sum_{i=1}^{F} n_i)$, where bpk is the user-defined average bits-per-key for the entire LSM tree. For simplicity, we assume $z_i > 0$ for all the files (otherwise, we directly assign $\text{bpk}_i = 0$ for files with $z_i = 0$), because building a BF for a file that only has existing queries (or no queries) does not help reduce #I/Os. Note that we still consider n_i when calculating M even if $z_i = 0$, which indicates that we re-allocate the BF size budget for file i with $z_i = 0$ to other files. After the above pre-processing, we can now formalize the objective function as follows:

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

After obtaining the optimal false positive rate ϵ_i for each file, we can then derive the bpk_i using Eq (1). Note that building a BF for a file with $z_i = 0$ does not help reduce #I/Os (i.e., all queries are existing queries), and thus in the optimal solution, we must have $\text{bpk}_i = 0$ for files with $z_i = 0$. We can first exclude all the files with $z_i = 0$ and then try to solve the optimal ϵ_i for other files with $z_i > 0$. If we know in advance that $\epsilon_i < 1\ \forall i \in [F]$, we could have a closed-form solution using *Lagrangian* multipliers for each ϵ_i:

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

However, the above solution does not hold if there are some files with $\epsilon_i = 1$ in the optimal solution. As such, we need an algorithm that can quickly identify which files have $\epsilon_i = 1$ in the optimal case, after which we can build the objective function for all other files with $\epsilon_i < 1$, and directly re-use Eq (7) to derive the optimal bits-per-key assignment. In literature, Partitioned Learned Bloom Filter (PLBF) [52] proposes an algorithm to find BFs with $\epsilon_i < 1$ but it assumes that C is monotonically increasing when F decreases. However, as shown in Eq (7), this assumption does not hold, and thus we need to re-design the searching algorithm to ensure that it can quickly and correctly find out the set of BFs with $\epsilon_i < 1$.

3.2 Duality

In this section, we present the ordered theorem with its proof, and derive an algorithm with $O(F \cdot \log F)$ complexity that can help us identify which files have $\epsilon_i = 1$ in the optimal solution.

Theorem 3.1 (Ordered Theorem). *In the optimal solution of the objective function of 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 dual variables 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 already implied by $\ln \epsilon_i$ in the memory constraint. We can 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 that this is convex programming 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 the strong duality holds. It can be seen that, 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 strictly hold. Again we can easily achieve this by setting all $\text{bpk}_i = 0$. Synthesizing all the above discussion and Slater's condition, we show that the optimization problem in Eq (6) has *strong duality*. Therefore, according to the strong duality, 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 let 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:

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

where $C = \lambda/(\ln 2)^2$, that can be calculated by Eq (7) with only considering BFs with $\epsilon_i < 1$. Proof completes. □

Connection with Monkey. In Monkey, shallower levels have larger bpk_j than deeper levels, which is consistent with Theorem 3.1 when the workload is uniform and only has empty queries. Specifically, when all the files have the same #entries (i.e., n_i is constant), we always assign more bpk to files with large z_i according to our model, 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_i 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 shallower 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 a descending order, and then we do reversely linear searching to find the maximum C so that the maximum false positive rate ϵ_{max} is smaller than 1. During this process, we also filter out files with small $\frac{z_i}{n_i}$ ("filter out" means not constructing BFs for these files, i.e., $\text{bpk}_i = 0$). After that, we can calculate bpk

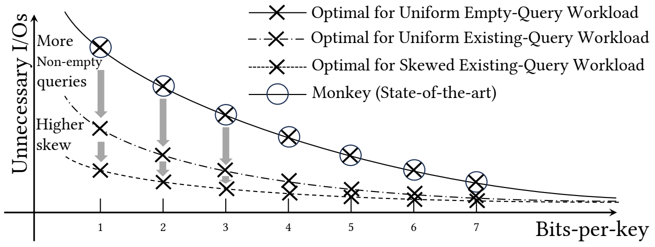


Figure 2: A conceptual graph that compares the state-of-the-art bits-per-key allocation solution from Monkey and the solution from our workload-aware model.

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 observe that C is monotonically increasing with respect to the number of files that have $\text{bpk}_i > 0$, and thus we can apply a binary searching algorithm to further reduce the searching time. However, the binary search does not bring significant improvement, as we see in Figure 4 in the micro-benchmark. As such, we only implement the linear search algorithm when implementing 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 in fact. This implementation restricts the minimum bpk_i to be 1 if $\text{bpk}_i > 0$. To achieve this, we just need to replace the stop condition (larger than 0) in line 8 of Algorithm 1 with larger than $e^{-(\ln 2)^2}$ (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 (we omit this proof because it is similar to the proof for the constraint $\epsilon \leq 1$, as mentioned above).

3.3 Micro-benchmark for Optimal BF Allocation

In this section, to investigate the improvement headroom for state-of-the-art solutions, we conduct a micro-benchmark that compares the number of accessed data blocks among the default BF allocation strategy (the same bpk for all BFs), the closed-form solution from Monkey (assuming all the levels are close to full), Monkey+ (the extension of Monkey), and the optimal one obtained by our model in Eq (6). Besides, since the gradient decent algorithm by Monkey+ can be also used to solve Eq (6), we further compare the execution time between the gradient decent algorithm and Algorithm 1.

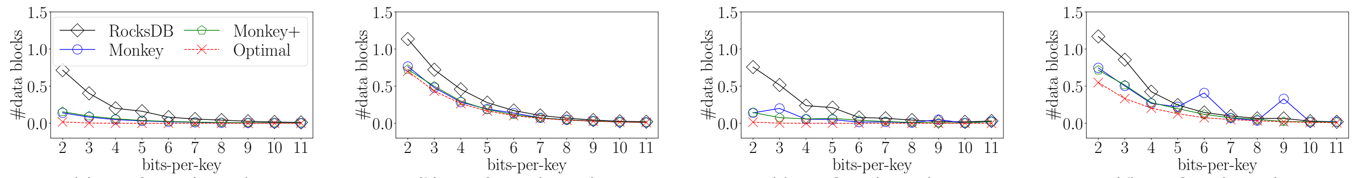
Experimental Methodology. We populate an LSM tree with $2M$ 512-byte entries (1GB) using the default BF allocation strategy (all the BFs have the same bpk). We set the size ratio $T = 4$, the data block size as 8KB, and the number of entries per file as 8192 (4MB per file), which produces a 3-level LSM tree. Then we allocate 32MB block cache, which ensures all the filters and indexes fit in cache except the data blocks, and we execute a read workload of $4M$ point queries (either all are existing queries or empty queries). By collecting 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 can initiate another LSM tree by recreating each SST file using the same amount of raw data but with different bpk_i per file. The bpk_i is derived using different allocation strategy. 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

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  $S_1 \leftarrow \sum_{i=1}^{F} n_i$;
5  $S_2 \leftarrow \sum_{i=1}^{F} n_i \cdot \ln \frac{n_i}{z_i}$;
6  $C \leftarrow \frac{-M \cdot (\ln 2)^2 - S_2}{S_1}$;
7  for $i^* \leftarrow n$ to 1 do
8      if $C + \ln V[i^*].first > 0$ then
9          $i_{tmp} \leftarrow V[i^*].second$;
10         $result[i_{tmp}] \leftarrow 0$;
11         $S_1 = S_1 - \mathcal{N}[i_{tmp}]$;
12         $S_2 = S_2 - \mathcal{N}[i_{tmp}] \cdot \ln \frac{\mathcal{N}[i_{tmp}]}{\mathcal{Z}[i_{tmp}]}$;
13         $C \leftarrow \frac{-M \cdot (\ln 2)^2 - S_2}{S_1}$;
14     else
15         break;
16 for $i \leftarrow i^*$ to 1 do
17     $i_{tmp} \leftarrow V[i].second$;
18     $result[i_{tmp}] = -\frac{1}{(\ln 2)^2} \cdot \left(\ln \frac{\mathcal{N}[i_{tmp}]}{\mathcal{Z}[i_{tmp}]} + C\right)$;
19 Return $result$;
```

the associated optimization problem with Algorithm 1. Then we execute the same query workload again on top of the new LSM tree, measure the execution time (with direct I/O turned on) and accessed #data blocks. Finally we report the average access latency per query and the average number of unnecessarily accessed data blocks (which is essentially 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, 3b, 3c, 3d, while Monkey/Monkey+ reduce a lot of unnecessary data block accesses compared with the default one for all kinds of workloads, the optimal strategy can further reduce the number of unnecessarily accessed data blocks when the workload contains all the existing queries or the workload exhibits higher skew. Similar patterns can be observed in the latency comparison in Figures 3e, 3f, 3g, 3h. As we expect, When all the point queries are empty queries and are uniformly distributed (two assumptions in Monkey/Monkey+), the optimal solution becomes nearly the same as Monkey/Monkey+, as shown in Figures 3b and 3f. We also observe that the I/O and latency gap between Monkey/Monkey+ and the optimal one becomes more evident when smaller bits-per-key (i.e., 2 $\sim$ 6) is used, which indicates more performance improvement headroom in this scenario. Besides, although Monkey and Monkey+ nearly perform the same (the green and the blue lines nearly overlap in most figures), Monkey+ is more stable (the green line is more smooth than the blue line) because Monkey+ does not require each level is close to full, and thus is more robust for an imperfect LSM-tree. Specifically, the bumpy points for Monkey in Figures 3d and 3h show that Monkey can


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

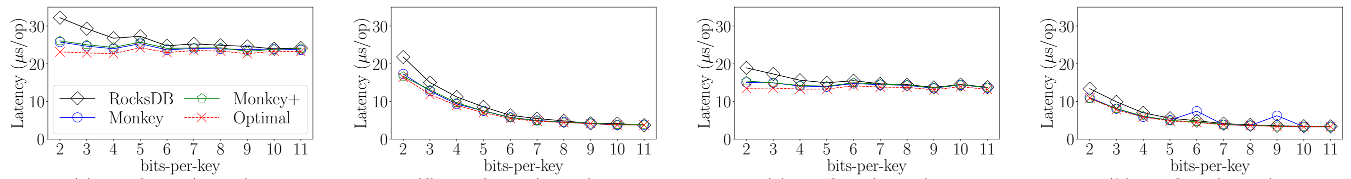

(e) Uniform ($Z = 0$) **(f) Uniform ($Z = 1$)** **(g) Zipfian ($Z = 0$)** **(h) Zipfian ($Z = 1$)**

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).

be occasionally worse than the uniform strategy. This typically happens when level 1 becomes empty when all the files in level 1 are compacted (the expansion mechanism in RocksDB may select multiple files to reduce write amplification [47], which results in an *imperfectly-shaped* LSM tree). Note that, in the closed-form solution of Monkey, one level is supposed to be close to full and consume $1/L$ memory budget for BFs. When level 1 becomes empty, the overall BF size is then smaller than the one with uniform strategy and thus the performance even degrades. Since Monkey+ is always superior to Monkey, we omit Monkey in the rest of this paper and only compare our method with Monkey+ in our experiment (we also refer Monkey+ as Monkey in the following text).

Efficiency of the Optimization Solver. We conduct another micro-benchmark that compares 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 4a and 4b, the gradient decent approach is slower than our algorithms by 2 to 4 orders of magnitude. Note that the execution time to find the optimal solution should not be larger than 1 second if we want to deploy it in real systems, because this process is executed during each flush and compaction. As the median compaction latency when populating a 10GB LSM-tree with a 4000 IOPS-provisioned SSD (with direct I/O) is just 1 second [49], the gradient decent algorithm is impractical to be used. On the contrary, our sort-and-search algorithms only take around 0.1 *ms* when $F > 1K$, which is negligible during compactions.

4 ACCESS STATISTICS ESTIMATION

In this section, we discuss how we estimate the number of zero-result (empty) queries per file (z_i) in LSM trees, which will be further used 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 huge maintenance overhead, and thus, we can only *estimate* z_i. To

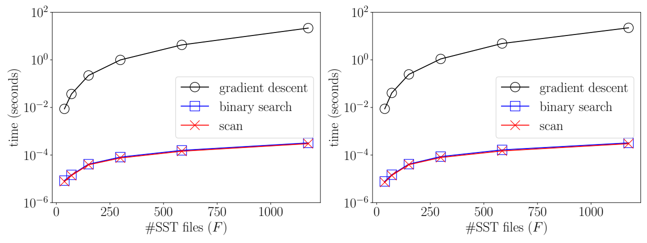

(a) $Z = 0.0$, bpk= 2 **(b) $Z = 0.0$, bpk= 8**

Figure 4: The execution time to solve the workload-aware objective function from Eq (6) for uniform workload.

achieve this, we estimate the number of queries q_i and the number of existing queries x_i, and use $q_i - x_i$ to approximate z_i.

4.1 A Naïve Strategy

We first introduce a naïve strategy for access estimation. 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. Whenever there is a compaction, we use the average counter of $\{q_i^{old}\}$ of the input old files as the estimated q_i^{new} for every newly generated file to avoid cold start (a method akin to this is used by ElasticBF [35] to monitor $\{q_i\}$). In this example of Figure 5, files 1, 2, and 3 are the input files in a compaction, and files 4, 5, and 6 are generated from this compaction. Using the naïve strategy, the number of point queries of newly generated files (i.e., $q_4^{new}, q_5^{new}, q_6^{new}$) are all estimated as $(\sum_{i=1}^{3} q_i^{old})/3$. We can also use a similar strategy to

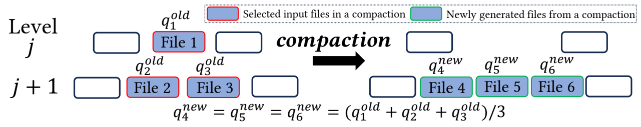

Selected input files in a compaction
Newly generated files from a compaction
Level j
$j+1$
q_1^{old}
File 1
q_2^{old}
q_3^{old}
File 2
File 3
compaction
q_4^{new}
q_5^{new}
q_6^{new}
File 4
File 5
File 6
$q_4^{new} = q_5^{new} = q_6^{new} = (q_1^{old} + q_2^{old} + q_3^{old})/3$

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.

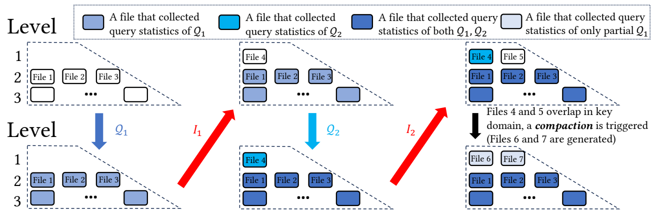

Level
A file that collected query statistics of $\mathcal{Q}_1$
A file that collected query statistics of $\mathcal{Q}_2$
A file that collected query statistics of both $\mathcal{Q}_1, \mathcal{Q}_2$
A file that collected query statistics of only partial $\mathcal{Q}_1$
1
2
3
File 1
File 2
File 3
File 4
File 5
File 6
File 7
$\mathcal{Q}_1$
$\mathcal{I}_1$
$\mathcal{Q}_2$
$\mathcal{I}_2$
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.

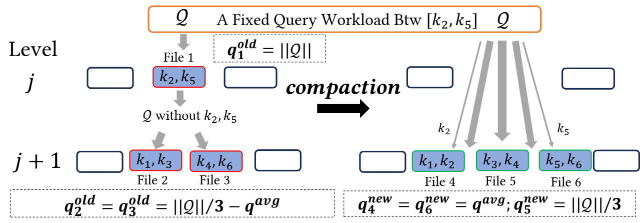

$\mathcal{Q}$ A Fixed Query Workload Btw $[k_2, k_5]$ $\mathcal{Q}$
Level
j
$j+1$
File 1
$q_1^{old} = ||\mathcal{Q}||$
k_2, k_5
$\mathcal{Q}$ without k_2, k_5
k_1, k_3
k_4, k_6
File 2
File 3
compaction
k_2
k_5
k_1, k_2
k_3, k_4
k_5, k_6
File 4
File 5
File 6
$q_2^{old} = q_3^{old} = ||\mathcal{Q}||/3 - q^{avg}$
$q_4^{new} = q_6^{new} = q^{avg}; q_5^{new} = ||\mathcal{Q}||/3$

Figure 7: An example that showcases the naïve model double counts z_1 (empty queries in File 1), assuming that $k_1 < \cdots < k_6$. $\|Q\|$ denotes the total number of queries in Q, and q^{avg} denotes the average number of queries per key in Q.

estimate z_i^{new} or x_i^{new}. However, the naïve strategy has two problems that lead to inaccurate estimation, in which case the solution of bits-per-key allocation model deviates a lot from the desired one. We detail these two problems as follows.

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, when we start with the left-top LSM tree, and sequentially execute workloads $\mathcal{Q}_1, \mathcal{I}_1, \mathcal{Q}_2, \mathcal{I}_2$ where $\mathcal{Q}_1, \mathcal{Q}_2$ are query workloads and $\mathcal{I}_1, \mathcal{I}_2$ are insertion workloads ($\mathcal{I}_1, \mathcal{I}_2$ 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 $\mathcal{Q}_1, \mathcal{Q}_2$, while q_6, q_7 only count $\mathcal{Q}_2$. Besides, q_6, q_7 are are also inaccurate even we only consider $\mathcal{Q}_2$. 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 (i.e., $q_5 = 0$), which makes q_6, q_7 deviate a lot from the actual ones if we replay $\mathcal{Q}_2$. Ideally, q_6, q_7 should take into account both query workload $\mathcal{Q}_1$ and $\mathcal{Q}_2$ 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 a level j proceeds to search in level $j+1$, which may lead to counting this empty query multiple times. Therefore, when we average the query statistics from the files involved into 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 on keys $\in [k_2, k_5]$ with k_2, k_3, k_4, k_5 equidistantly distributed. As we see in Figure 7, 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 $\in [k_1, k_3]$ – or File 3 – if key also $\in [k_4, k_6]$. If, after the compaction, we distributed all accesses in File 1 between the new Files 4 through 6, then we have counted the aforementioned queries two times: once from File 1 and once from File 2 or File 3.

4.2 Merlin

In this section, we introduce Merlin, a tracking and estimation mechanism of the number of zero-result point queries per file.

4.2.1 Estimation of Zero-Result Point Queries.

As mentioned earlier, in our estimation mechanism, we decompose the estimation of z_i into two steps – 1) estimating q_i, 2) estimating x_i. z_i can be then approximated by $q_i - x_i$ (we will 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 number that access 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 [28] (also applied in Cassandra [3]). 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 Δ_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 interval 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:

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

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

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 as 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 [6, 29, 40], and the old statistics thus become outdated if we only use the counter to estimate q_i and x_i. While ElasticBF has a threshold "expiredTime", which specifies when the statistics become invalid, this approach hastily eliminates all the older statistics than "expiredTime", which could result in inaccurate estimation. A sliding window in Merlin keeps tracking of most-recently accessed queries, and thus it can naturally adapt to shifting workload.

Concurrency. Although we can use `atomic` variables for all the counters when there are multiple querying threads, it is hard to maintain a thread-safe queue $\Omega^{(i)}$ 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 around 0.5% probability to switch the write permission to other threads. Note that D^{global} is implemented by an `atomic` counter, and thus we can still estimate q_i and x_i using D^{global} without concurrency issues.

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 q_i, x_i to decide if we use the BF according to Eq (12), but this also only takes constant time (some files should have $\text{bpk}_i = 0$ but no compactions involve them to change bpk_i, using Eq (12) can avoid loading their filters to alleviate the memory resource competition). In terms of space, we need a 64-length queue of which each element (the point query sequence number) is also 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 as 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 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 k_2, k_5, 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 how much 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 new file to level $j+1$, we have:

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

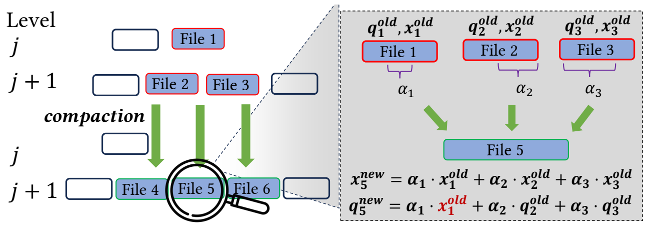
where I_j, I_{j+1} represent the set of file IDs in level j and $j+1$. For example, in Figure 8, x_1^{old} is the number of existing queries in shallower level, and we use x_1^{old} instead of q_1^{old} to estimate q_5^{new} to avoid the double counting issue. In addition to $q_{i^*}^{new}$, the number of existing queries x_i does not have the double counting issue because all the existing queries for the file in the shallower level terminates at the same level and thus they do not *contaminate* the x_i counter in 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, 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 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 as well (case 2). In case 3, the gap in the key domain in level $j+1$ is so huge that a large proportion of zero-result queries in level j skips level $j+1$. To deal with all the above cases, we adjust Eq (13) as follows: 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 calculating $q_{i^*}^{new}$. More details can be found in our implementations[4].

4.2.3 Mnemosyne: Integrating Merlin with Our Cost Model into RocksDB. We add the queue and other counter variables into the metadata of the `FileMetaData` object to implement the sliding-window-based estimation. During each compaction, we calculate q^{avg}, the average number of queries per key for each input file, and we embed q^{avg} into the compaction iterator. When adding key-value pairs to a new file i, we accumulate q^{avg} to q'_i (similar to x'_i). To determine bpk_i for each new file, we estimate $\{z_i\}$ for all files in the LSM tree during each flush and compaction, and run Algorithm 1 to get a constant C, defined in Eq (12). After that, we use C, n_i, and $z_i = q'_i - x'_i$ to calculate bpk_i (following line 18 in Algorithm 1). When $\text{bpk}_i < 1$, we round up bpk_i to 1 if $\text{bpk}_i > 0.5$, otherwise we set $\text{bpk}_i = 0$, just as RocksDB already does. Note that we cannot rebuild BFs for other old files since there might be concurrent compactions relying on them. Despite this, 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]https://github.com/BU-DiSC/Mnemosyne



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

Figure 8: An example for our inheritance model. File 5 is generated when merging files 1, 2, and 3. $\alpha_1, \alpha_2, \alpha_3$ indicates the proportion of files 1, 2, and 3 that is used when generating file 5. By definition, $\alpha_1, \alpha_2, \alpha_3 \in [0, 1]$ and $\alpha_1 + \alpha_2 + \alpha_3 > 0$.

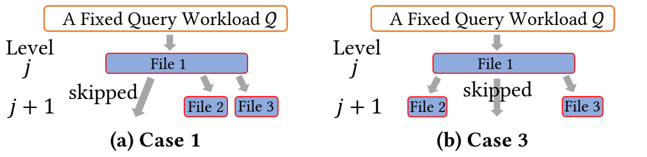


Figure 9: Two example corner cases (when some zero-result queries are counted in level j but not counted in level $j+1$).

4.3 Micro-benchmark for Estimation

In this section, we conduct a micro-benchmark that compares 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 paris with size ratio of 4 and file size of 64 MB. To examine the accuracy of a tracking strategy, we copy the database and we execute a mixed workload with $10M$ point queries and $5M$ updates (also referred as ingestion in the following text). For every $200K$ ingestion, we manually issue a flush operation, and we copy the whole database with resetting all statistics counters and re-issue all the point queries that have been issued so far. By doing this, we obtain a database with ground-truth statistics every $200K$ ingestion, and then we are able to compare the estimated statistics with the ground-truth one. Since two databases have exactly the same tree structure, we can build a $\{z_i\}$ vector that records the number of estimated/ground-truth z_i per file. Next, we can compare the distance between the estimated vector and the ground-truth vector (two vectors should have exactly the same length and each dimension stands for the estimated/ground-truth z_i for a specific file) using Euclidean distance and cosine similarity. We then report how the distance and the similarity changes as the database has more ingestions for different query workloads. We repeat the experiment for three times and we report the average one. We only list the experimental results for $Z = 0.5$ (i.e., half of queries are zero-result queries) in Figure 10, because results are similar when $Z = 0, 1$. 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. Inspecting all the comparison between our mechanism and the naïve strategy in Figure 10, Merlin nearly reduces the Euclidean distance of the naïve strategy by half, and the cosine similarity of Merlin is twice as large as the naïve strategy. A common trend of both strategies is that the Euclidean distance increases with

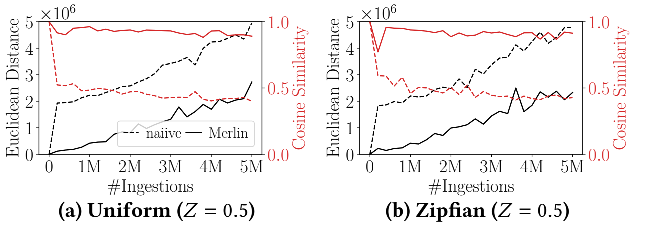


Figure 10: The Euclidean Distance (black lines)/Cosine Similarity (red lines) between the estimated z_i per file and the ground truth for the naïve tracking strategy and Merlin under uniform and zipfian access distribution.

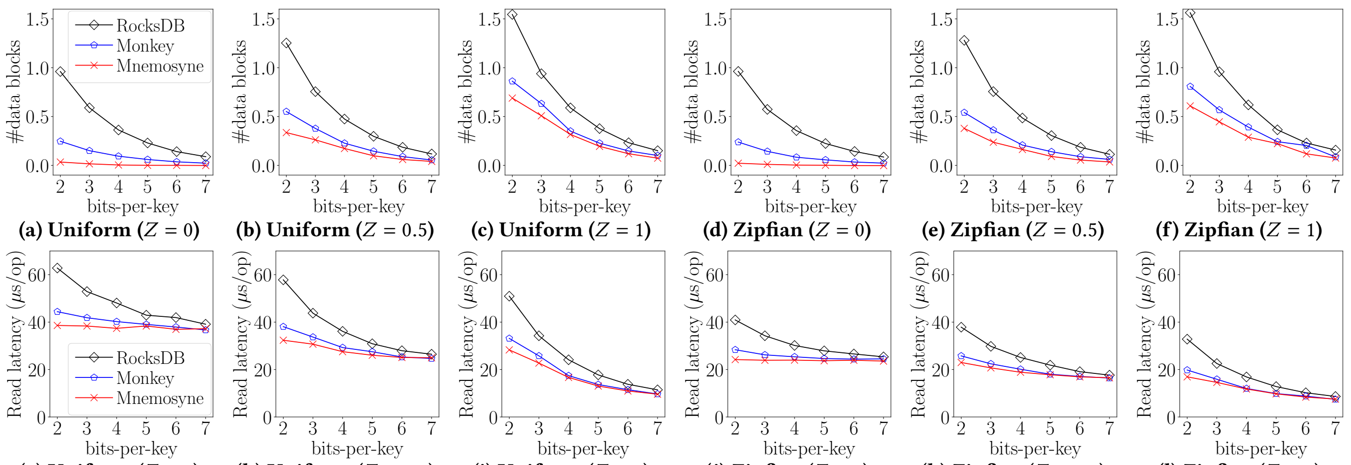
more ingestion but the cosine similarity is more stable (the cosine similarity of Merlin remains between 0.85 and 0.95, and the naïve one becomes stable at around 0.5). This is because both Merlin and the naïve strategy cannot get the exact z_i for new files during compaction. As such, the absolute error (Euclidean distance) between the estimated vector and the ground truth is stably increasing with more ingestions (i.e., more compactions). Compared to Euclidean distance, cosine similarity is a normalized metric that measures the angle between two vectors, regardless of the magnitude of each dimension. Recall our workload-aware objective function in Eq (6), we can also replace z_i with $z_i / \sum_{i=1}^{F} z_i$ by assuming $\sum_{i=1}^{F} z_i$ is a constant. In this way, we actually only require the relative relationship between z_i is accurately captured instead of the exact z_i. Therefore, the bits-per-key assignment solution using $\{z_i\}$ obtained from Merlin could be much closer to the actual optimal one, which can thus lead to better read performance.

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 to make it applicable in RocksDB (noted by Monkey now). In this section, we present the experimental results that compare Mnemosyne, Monkey, and the default strategy.

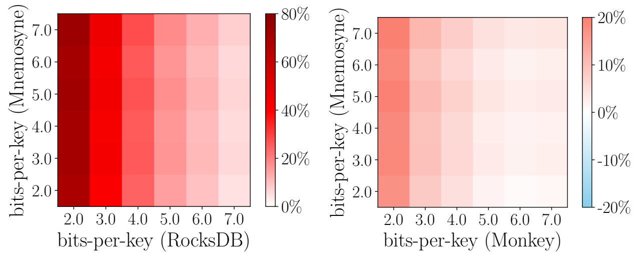
Environment. We use our in-house server, which has 375GB memory and two Intel Xeon Gold 6230 2.1GHz processors, each having 20 cores with virtualization enabled. By default, we use a 350GB PCIe P4510 SSD with direct I/O enabled (reading a 4KB page takes around 15 μs) as our secondary storage. All the codes are complied using gcc (12.3.1) with optimization level -O2 enabled.

Experimental Methodology. We generate a workload that starts with $21M$ 512-byte key-value pairs (128-byte key and 384-byte value) and is followed by a mixed workload of $31M$ point queries and $10M$ updates. With the size ratio $T = 4$, the overall database size is $21 \cdot 10^6 \cdot 512 \cdot (1 + 1/4)$ bytes $\approx$ 12.5GB, considering the space amplification factor $1/T$ from updates. With the write buffer and file size set as 64MB, the LSM tree has 4 levels (excluding level 0). As many storage-optimized and compute-optimized EC2 instances in AWS have less than 3% memory-to-storage ratio, we configure the block cache size as 256MB ($\approx$ 2% of the data size) to emulate the memory pressure scenario (the block cache cannot be as large as 3% of the data size because we need to reserve some memory for other components in both the LSM tree and the operating system, e.g., the write buffer also takes up 64MB). By default, RocksDB



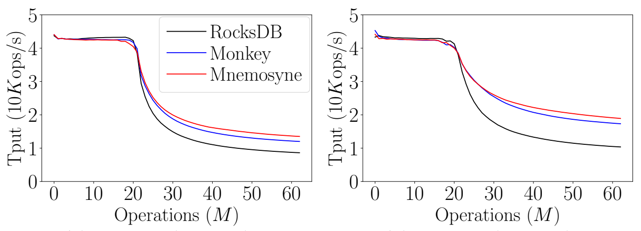
(a) Uniform ($Z = 0$) **(b) Uniform ($Z = 0.5$)** **(c) Uniform ($Z = 1$)** **(d) Zipfian ($Z = 0$)** **(e) Zipfian ($Z = 0.5$)** **(f) Zipfian ($Z = 1$)**
(g) Uniform ($Z = 0$) **(h) Uniform ($Z = 0.5$)** **(i) Uniform ($Z = 1$)** **(j) Zipfian ($Z = 0$)** **(k) Zipfian ($Z = 0.5$)** **(l) Zipfian ($Z = 1$)**

Figure 11: The average number of unnecessarily accessed data blocks and the average latency per query.



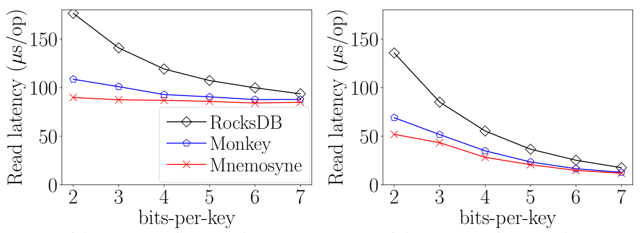
(a) Mnemosyne v.s. RocksDB **(b) Mnemosyne v.s. Monkey**

Figure 12: Speedup for Zipfian workload ($Z = 0$).



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

Figure 13: Throughput (including ingestions) for bpk = 2.



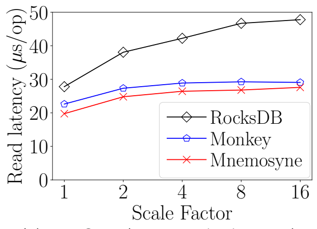
(a) Zipfian ($Z = 0.0$) **(b) Zipfian ($Z = 1.0$)**

Figure 14: The average query latency for slower NVM SSD.

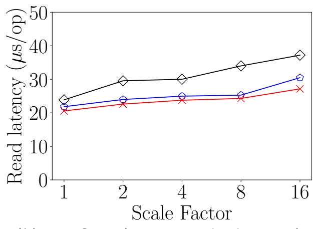
uses half of the block cache to store all the filter and index blocks (`high_pri_pool_ratio=0.5`), and thus, to ensure that all the index blocks fit in a 128MB cache, we set the data block size as 16KB. This is derived as follows: one entry in the index block takes 140 bytes (128-byte key, plus 8-byte internal sequence number, and 4-byte offset of the associated data block), and thus the data block is at least 12.5GB/(128MB/140B) $\approx$ 13.7KB (normally we set the data block size as a multiple of 4KB, and thus the data block size should be 16KB). Note that the maximum bpk is $(128 \cdot 2^{20} - 12.5\text{GB}/16\text{KB} * 140) \cdot 8/(21 \cdot 10^6 \cdot 1.25) \approx 5.96 < 6$ so that both indexes and BFs fit in the cache. As we observe from Figure 3, both monkey and the optimal bits-per-key allocation strategy do not have explicit performance improvement over the default strategy when bpk is larger than 7. As such, in most experiments, we vary bpk between 2 and 7 (bpk=7 for a 128MB cache may trigger additional I/Os to read fence pointers or BFs) for a given workload using three different bits-per-key allocation strategy, and compare the average query latency. We only present a subset of our experiments due to limited space, please refer to our full version for more experimental results.

Mnemosyne has Higher Benefit with Small bits-per-key for Non-empty Queries. We test 6 different workloads, and we report the average number of unnecessarily accessed data blocks and the average latency per query, as shown in Figure 11. We first observe that we have higher benefit for small bits-per-key (e.g., $2 \sim 4$) for every workload. As the workload contains more non-empty queries (from $Z = 1$ to $Z = 0$), the benefit of Mnemosyne is also pronounced for uniform and zipfian query workloads, which is also consistent with what we observe when we compare the optimal and Monkey in the offline setting (see §3.3). Besides, we see that, for uniform workloads, the patterns of different bits-per-key allocation strategy for #unnecessarily accessed data blocks are almost the same as the patterns for latency (comparing Figures 11a, 11b, 11c with Figures 11g, 11h, 11i) But when it comes to a zipfian workload, latency reduction percentage is much less than the percentage of reduced unnecessarily accessed data blocks, which is because frequently accessed data blocks are cached and thus reducing accessed data blocks does not necessarily mean fewer I/Os, and thus Mnemosyne has less latency reduction when the workload becomes skewed.

Mnemosyne Saves Up to 70% Space to Achieve the Same Performance as Monkey. Despite that Mnemosyne has less latency reduction for skew workloads, we emphasize that Mnemosyne uses smaller space to maintain the same performance as Monkey. We re-plot Figure 11j in Figure 12. Red color indicates positive speedup,



(a) Zipfian (Z = 0.0, bpk = 2.0)



(b) Zipfian (Z = 0.0, , bpk = 4.0)

Figure 15: The average query latency for Zipfian workload.

blue color indicates negative speedup, and white color indicates no speedup. As we can see, when comparing Mnemosyne with the default strategy, all cells are positive speedup , meaning that we can use bpk = 2 for Mnemosyne to achieve lower query latency than the default one with bpk = 7. Similarly, in Figure 12b, Mnemosyne with bpk = 2 can achieve the same performance as Monkey with bpk = 7, indicating over 70% space savings.

Write Throughput Is Not Impacted in Mnemosyne. Due to the inheritance mechanism in Merlin and Algorithm 1 in each compaction, the ingestion performance might be negatively affected. To examine how much overhead Mnemosyne could have, 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, the write throughput of Mnemosyne remains nearly the same as Monkey and the default one. In the next $41M$ operations (a mixed workload of $31M$ queries and $10M$ updates), the average throughput of Mnemosyne becomes superior to Monkey and the default one. This is because Mnemosyne still has similar ingestion throughput in $10M$ updates but it also has lower query latency, compared to Monkey and the default strategy.

Latency Reduction is Pronounced on a Slower NVM SSD. To magnify the latency reduction, we re-run the experiments for 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 the default SSD. We compare the average query latency in Figure 14. Although some frequently accessed data blocks can be cached, and we may have fewer I/O reduction for skew workloads, reading a data block now becomes 2.4× more expensive, and thus we can observe larger difference between Monkey and Mnemosyne using a slower SSD. Specifically, when bpk = 2, 3, 4, we have 17%, 14.6%, and 6% latency reduction in Figure 14a while we only have 14.5%, 8.7%, and 5.5% latency reduction in Figure 11j.

Mnemosyne Scales Better than Default and Remains Beneficial Over Monkey. We vary the workload scale from 1 to 16 to examine the scalability of Mnemosyne. The base workload starts with $500M$ 512-byte key-value inserts, and is follow by a mix of $750M$ point queries and $250M$ updates. Similar to other experiment, we set the block cache size as 2% of the data size. We select the zipfian workload with all non-empty queries to plot and execute the scalability experiment for bpk = 2 and bpk = 4, respectively. As shown in Figure 15, for $Z = 0.0$, the query latency of Mnemosyne increases slightly when the database grows exponentially while the default method has more explicit increasing query latency. The speedup of Mnemosyne over Monkey is nearly a constant (8% for bpk = 2 and 5% for bpk = 4) when the database grows.

6 RELATED WORK

Memory Allocation in LSM-trees. In addition to allocate memory between all the Bloom Filters in LSM trees, memory can also be re-allocated among Bloom Filters, fence pointers, and the write buffer [14, 32]. Further, for LSM-based storage systems (multiple LSM trees exist), memory can be further re-allocated between different LSM trees [37, 38]. In this paper, we extract the constrained BF size re-allocation as an optimization problem and design an efficient sort-and-search algorithm to solve it. The proposed algorithm can be applied as a sub-routine in other memory re-allocation problems to accelerate the optimization procedure.

Membership-Testing Filters. There are various membership-testing filters proposed in the past a few decades including Blocked Bloom Filter [45], Cuckoo Filter [24], Quotient Filter [44], Morton Filter [8], Vacuum Filter [54], Xor Filter [27], Ribbon Filter [21], InfiniFilter [15], etc. Most filters are designed based on fingerprints, and the false positive rate of these fingerprint-based filters are linear to $2^{-l} = e^{-(\ln 2)\cdot l}$ where l can be also termed as the bpk in these filters. If we want to replace Bloom Filters in LSM trees with any other fingerprint-based filter, we can replace the false positive rate definition in Eq (1) and our sort-and-search algorithm still applies. In addition, Chucky [18] and SlimDB [46] construct a global Cuckoo Filter for an LSM tree, but this design cannot scale to a low-memory system because it assumes the entire filter fits in memory, and the filter has to be updated during every compaction (which means that compactions may have to update the in-disk part of the global filter). Furthermore, although Partitioned Learned Bloom Filters (PLBF) [52] builds a similar objective function and also proposes an algorithm to find the optimal overall false positive, it mainly targets finding out how to partition the key space, and its optimization algorithm to 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 [56, 58] or using extra memory for frequently queried keys [59]. When the available memory is not sufficient to load all the BFs, other works like ElasticBF [35], ModularBF [42] break a BF into a few smaller BFs and also propose workload-aware filter access mechanism to alleviate the memory pressure by avoiding loading the entire filter into block cache. We highlight that all the existing skew-aware techniques can co-exist with Mnemosyne to accelerate the point lookups. In addition to LSM trees, there are also some workload-aware techniques in other kinds of key-value stores (e.g. hash-table-based or purely in-memory) [33, 34, 36], which are mostly specifically designed for the associated data structures, and thus cannot be directly applied in LSM trees.

7 CONCLUSION

In this paper, we study the workload-aware Bloom Filter Tuning with accurate statistics estimation in LSM trees. We construct a more general cost model that considers the access pattern per file, and propose an efficient algorithm that can quickly find the optimal size per BF with negligible CPU overhead. To apply our model in real systems, we further design a novel and accurate query statistics tracking mechanism Merlin. We implement Mnemosyne by integrating Merlin and our new cost model into the state-of-the-art LSM-tree-based key-value store RocksDB. When we have limited memory resource, Mnemosyne achieves up to 15% throughput improvement over Monkey and 50% improvement over the default RocksDB for query workloads that exhibit skew or contain a high fraction of existing vs. 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] Apache. 2023. Accumulo. *https://accumulo.apache.org/* (2023).

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

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

[5] 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)*.

[6] 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

[7] 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

[8] 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

[9] 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.

[10] 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

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

[12] 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

[13] 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

[14] 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), 1–16. https://doi.org/10.1145/3276980

[15] 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

[16] Niv Dayan and Stratos Idreos. 2018. Dostoevsky: Better Space-Time Trade-Offs for LSM-Tree Based Key-Value Stores via Adaptive Removal of Superfluous Merging. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 505–520. https://doi.org/10.1145/3183713.3196927

[17] 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

[18] 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

[19] 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

[20] 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

[21] 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

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

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

[24] 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

[25] 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)*. 1–32. https://doi.org/10.1145/2741948.2741973

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

[27] 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

[28] 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

[29] 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

[30] 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

[31] 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

[32] 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

[33] 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

[34] 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

[35] 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.

[36] 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

[37] 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

[38] 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

[39] 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

[40] 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

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

[42] 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

[43] 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

[44] 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

[45] 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

[46] 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

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

[48] 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

[49] 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

[50] 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

[51] 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

[52] 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

[53] 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

[54] 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

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

[56] 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

[57] 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

[58] 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.

[59] 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

[60] 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.