LSM-Trees Under (Memory) Pressure

Ju Hyoung Mun, Zichen Zhu, Aneesh Raman, Manos Athanassoulis
Boston University

ABSTRACT

Log-structured merge trees (LSM-trees) are widely used in modern key-value stores since they offer efficient data ingestion. To accelerate point lookups, LSM-trees employ filters such as Bloom filters (BFs) to reduce unnecessary storage accesses to levels that do not contain the desired key. BFs are particularly beneficial for empty queries while they might be a small burden for non-empty queries. Further, with larger datasets, the size of metadata like index and filters also increases, making it less feasible to keep all BFs in cache. Coupling this, with the increasing price of memory and the need to reduce the memory-to-data ratio in many practical deployments, we are seeing an increased *memory pressure*. In this setting, fewer BF blocks are cached, thus causing additional storage accesses, since they have to be fetched in memory to answer a query.

In this paper, we introduce SHaMBa, a new LSM-based key-value engine that addresses the suboptimal performance when BFs do not fit in memory. SHaMBa integrates a new variation of BF, called Modular Bloom filters (MBFs) that replace a single Bloom filter with a set of mini-BFs (*modules*) having the same aggregate size and requiring the same total number of probes, distributed among the modules. Querying MBFs accesses the modules sequentially, resulting in the first module being more frequently in memory while the remaining modules compete with data blocks in case of positive queries. Further, we propose a new memory management policy and two BF-skipping strategies to avoid accessing BFs when they are ineffective. Our evaluation shows that SHaMBa substantially outperforms the state of the art under memory pressure, having the same average number of I/Os, needing only *one-third of the memory consumed by the state of the art.*

1 INTRODUCTION

Log-Structured Merge-trees (LSM-trees) [31] are widely adopted in state-of-the-art key-value engines including RocksDB [18] at Facebook, LevelDB [20] and BigTable [10] at Google, HBase [21] and Cassandra [4] at Apache, WiredTiger [42] at MongoDB, X-Engine [22] at Alibaba and Dynamo [16] at Amazon, as they offer high ingestion rate and fast reads. LSM-trees have several tuning knobs like the compaction policy, the size ratio between the levels, and the use of metadata - typically Bloom filters and fence pointers. As a result, tuning LSM-trees has been a key goal, and various optimizations have been introduced for querying [1, 49], compaction [2, 14, 26, 37, 38, 47], filters and memory management [7, 12, 23, 25, 28, 44, 46, 48, 50], and holistic tuning [13–15, 33].

The Structure of LSM-trees. LSM-trees buffer incoming data in a memory buffer that, when full, is sorted and flushed to disk in the form of a *sorted run*. When a sorted run is flushed to disk, it may be iteratively merged with existing runs of the same size. Overall, as a result of such iterative merges, the sorted runs on disk, also termed *Sorted-String Table* or *SST files* in the literature [10, 17], form a collection of *levels* of exponentially increasing size with potentially overlapping key ranges across the levels. As a result, a lookup may need to search all levels in the tree until it finds a match, which can lead to multiple I/Os per lookup. In order to accelerate read performance, LSM-trees employ metadata such as fence pointers and Bloom filters to reduce the number of storage accesses [27].

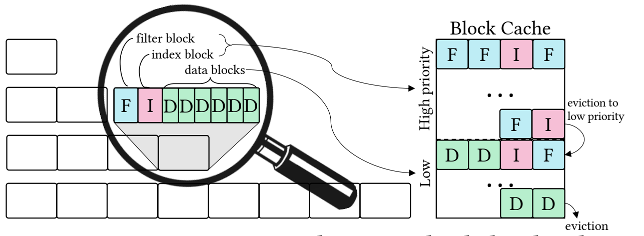
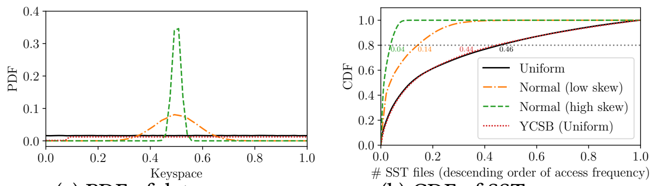


Figure 1: LSM-trees organize data at multiple levels. The capacity of each level grows exponentially by the predefined size ratio and each level consists of immutable sorted *runs*, often composed of multiple sorted files. Each file maintains filter (F) and index (I) metadata along with data blocks (D) to accelerate lookups. To maximize the temporal and spatial locality, these blocks are stored in a block cache. Often, the block cache prioritizes F and I blocks since they have a larger performance impact than D blocks.

Bloom Filters and Caching in LSM-trees. In order to avoid unnecessary data accesses during querying, LSM-trees employ a Bloom filter (BF) [6] for every level or sorted run (or even per file in case of partial compactions). BFs are used to identify levels that can be safely skipped because they do not contain the queried key. Figure 1 shows the basic structure of the physical storage of an LSM-tree. Each level consists of a number of SST files, and each SST file contains three types of blocks: *filter blocks (F)*, that contain the Bloom filters, *index blocks (I)*, that contain the fence pointers (i.e., a collection of triplets (min, max, offset) of each block), and *data blocks (D)*. Every time an SST file is accessed, it is beneficial to cache the accessed filter blocks (to find out whether we should continue searching this SST file), the accessed index blocks (to help us identify the data blocks that contain the desired data), and the data blocks (in case they are accessed again in the near future). Practical systems like RocksDB employ a two-level block cache where the filter and index blocks are cached with a higher priority, whereas data blocks are cached with a lower priority [35]. Essentially, the block cache has two zones: the first zone caches filter and index blocks, and the second zone caches data blocks. Contrary to data blocks, filter and index blocks are not directly evicted. They are first pushed to the second zone for one more round of LRU caching while competing with data blocks. Note that while BF and index



(a) PDF of data accesses (b) CDF of SST accesses

Figure 2: Even with uniform access patterns in the data domain, the access frequency of the SST files is not uniform, hence, SST files require a non-uniform memory allocation strategy for their BFs.

blocks are cached in memory to maximize their efficacy, they are ultimately part of the sorted runs for persistence.

Memory Pressure. The memory allocated to an LSM-based system is used by several different components: (i) the block cache for caching BF blocks, index blocks, and data blocks, (ii) the memory buffer for storing incoming data, (iii) temporary memory to compact data, and (iv) temporary memory to support ongoing range queries. While compute, memory, and storage prices decrease and allow us to facilitate more data, in the last few years, the price drop in memory has been slower than what has been for compute and storage [29], making it hard to maintain the same memory-to-data ratio. As a result, BFs may not always be in memory, and a significant number of I/Os may be spent on fetching them. For example, consider an LSM-tree with one billion 1KB key-value pairs and a size ratio of 10. It requires about 1.1TB of storage (1TB for the base data and 11% of space amplification due to the inherent LSM-tree duplication [17]). Assuming 10 bits per key are applied for each BF and a 64B key size, around 1.25GB is required to cache all BFs and approximately 17.19GB is necessary for index. If the feasible memory-to-data ratio is 1:100, the available block cache would be up to 1% of the 1TB data (10GB), and it can only cache a subset of the metadata. For every metadata block not cached, the cost of fetching it from the disk to memory is prohibitively expensive. As the cost of storage is projected to keep getting cheaper faster than memory, we expect the memory pressure to be further exacerbated.

Are All (Filter) Blocks Equally Important? As memory becomes a scarcer resource, one should reconsider whether some of the cached blocks are not as useful. For instance, LSM-tree tuning for BFs does not consider the workload. Primarily, only the lookup vs. the ingestion ratio is used in tuning, however, the access distribution may also affect the benefit gained from using BFs. In fact, this benefit from BFs across different SST files varies significantly, *even when the workload is perfectly uniform.* Figure 2 shows how the read access patterns on the domain are translated into access requests on various SST files of the LSM-tree. Figure 2(a) shows four different read workloads - a YCSB-based read workload with uniform accesses, and three synthetic workloads following the uniform, and the normal distribution with two levels of skewness. Figure 2(b) shows the cumulative distribution function (CDF) of SST accesses for each workload. Note that the x-axis contains all the SST files sorted by the number of accesses in descending order. The horizontal dotted gray line corresponds to CDF value 0.8, i.e., it shows the fraction of the SST files that get 80% of the accesses. A striking observation is that even in a perfectly uniform workload, 80% of the lookups are directed to 44%-46% of the SST files, while for more skewed workloads described in Figure 2(a), the numbers decrease to 14% and 8%. Overall, only a few BFs are accessed during most point lookups, therefore, the majority of the BFs consume (possibly scarce) memory resources without offering significant performance advantages. *The challenge here is to reduce the memory consumption of unpopular BFs without hurting overall performance.*

Modular Bloom Filters. To address this, we propose a new variation of BF, termed Modular Bloom filter (MBF) that divides the Bloom filter into multiple modules. Each module is also a Bloom filter by itself, i.e., an MBF is equivalent to a set of BFs. A query sequentially accesses all modules until a negative result is obtained or all modules have been queried, and the false positive rate of the MBF is the same as the standard BF of the same aggregate size. MBFs can accelerate point lookups by accessing a different number of modules adaptively. A lookup can start with only a single module, therefore, it does not require the whole filter to start probing. In fact, an empty point query does not need (on average) to access all modules. For example, in the common case of 10 bits per key, having three modules will lead to accessing only one module on average. Having multiple BF modules, allows us to evaluate the *utility of each module* and use it only when it is beneficial.

MBFs in LSM-trees: SHaMBa. BFs in LSM-Trees are initially stored on storage and fetched in block cache on demand. The entire BF block of a file has to be brought into the cache. A key benefit of MBFs is that by allowing them to bring only specific modules of the overall filter, they navigate the memory vs. performance trade-off of BFs without re-hashing. Further, MBFs build on the concept of hash sharing [50] to amortize the hashing overhead due to using multiple BFs. By controlling the number of modules accessed and cached, **S**haring **Ha**shing with **M**odular **B**loom filters, or SHaMBa for short, enables variable memory footprint and highly tunable false positive rate with no additional overhead.

Controlling the Number of Modules Using Their Utility: To decide how many modules to access, we quantify their *utility*, i.e., a measure of the benefit of a module. We propose a novel lookup policy that skips a module if there is no benefit from its filter. Skipping a module avoids fetching unnecessary filter blocks from the disk and, more importantly, prevents non-beneficial modules from polluting the cache. SHaMBa prioritizes fetching in block cache the modules that correspond to frequently read parts of the LSM-tree. Since the size, and consequently, the false-positive rate of an MBF can easily be adjusted by varying the number of modules, our proposed algorithm can diversify the false-positive rate across files in order to maximize the benefit from BFs under a given memory budget.

Contributions. Our contributions are as follows.

- We propose Modular Bloom filters (MBF), a novel Bloom filter variant, which navigates the memory/accuracy BF tradeoff without having to re-hash. Using MBF, we implement a rich set of filter management policies with no overhead.
- We quantify the *utility* of a module that enables skipping parts of the BF with no benefit. We show that module skipping reduces memory utilization without sacrificing performance.
- We integrate Modular Bloom Filters in the state-of-the-art LSM-engine RocksDB, and we show through extensive experimentation with realistic workloads that our proposed techniques outperform the state-of-the-art under memory pressure.

2 BACKGROUND ON LSM-TREES

LSM Structure. LSM-trees are widely adopted by modern key-value stores since they offer high ingestion rate [4, 10, 16, 18, 20–22, 31, 42]. LSM-trees store all inserts into a memory buffer with a predefined size. Note that updates and deletes on existing keys are treated similarly to inserts. Once the buffer is full, it is flushed to secondary storage as an immutable sorted *run*, composed of multiple immutable stored files referred to as SST files. Similar to a buffer flush, a sort-merge is triggered when a level reaches its maximum size, merging the saturated level with the next level. During this process, obsolete entries are removed and the two levels are merged into a single sorted level. The capacity of each level exponentially grows by a factor T, termed *size ratio*. Thus, shallower levels have newer entries but smaller sizes.

Compaction: Tiering vs. Leveling and Full vs. Partial. The sort-merging between consecutive levels, termed *compaction*, can be done either eagerly to optimize for future reads (*leveling*) or lazily to increase the write throughput (*tiering*) [26]. Hybrid compaction strategies that mix leveling and tiering on different levels have also been proposed [9, 14, 15, 34]. Leveling restricts the number of runs to 1 in each level, and once there is a merge operation between Level i and Level $i+1$, all overlapping data files in Level $i+1$ are considered. In contrast, when it comes to tiering, the number of runs within the same level can be as large as $T-1$ and the sort-merge takes place between files of the same level when it reaches its maximum allowed size. In addition, compaction can be applied globally to the whole level, or partially to a small partition of a level. The first, called *full compaction*, merges the entire Level i with the entire Level $i+1$, while the latter, called *partial compaction*, selects a small partition of Level i (e.g., a single SST file) and compacts it with the overlapping SST files of Level $i+1$ [17]. In this paper, we focus on partial leveling compaction, which is used by various state-of-the-art systems including RocksDB.

Point Queries in LSM-trees. As LSM updates are out-of-place, multiple versions of entries with the same key may co-exist. Note that the newer version is always at a higher level. Thus, a lookup sequentially proceeds from the highest (smallest) to the lowest (largest) level and terminates when the first match is found. If there are multiple runs at a level, the search goes from the youngest to the oldest run and similarly returns the first matching value.

Auxiliary In-Memory Metadata. LSM-trees accelerate reads using Bloom filters and fence pointers in every SST file.

Bloom Filters: For each run, a Bloom filter (BF) [6, 40] stores the membership information of all keys comprising this run. To avoid multiple (potentially unnecessary) I/Os when querying LSM-trees, a Bloom filter (per level) is queried in advance of accessing a run to determine if it may contain the target key. Only for positive results, the search continues to access the run on secondary storage. However, BFs can have false positives (not false negatives). As a result, the worst-case lookup cost depends on the sum of false positives for all levels of the tree.

Fence Pointers: Every run residing on the disk is sorted by key; thus, the key range of each disk page does not overlap with any other page of the same run. In order to ensure at most one I/O during a lookup for a single run, LSM-tree engines deploy fence pointers, which are min-max ranges for each disk page. That way, before accessing the run on storage, an efficient search in the fence pointers would tell us which page to access on storage.

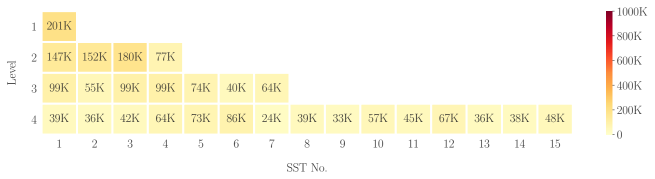
Block cache: In addition to fence pointers and Bloom filters, every LSM-based key-value engine uses a block cache to hold recently or frequently read blocks in memory. These blocks may come from the fence pointers (index blocks), from the BFs (filter blocks), or from data (data blocks). State-of-the-art LSM engines like RocksDB use a block cache that has two priority queues where each one follows a specific (e.g., LRU) eviction policy. The first queue is called the high-priority one, and the second the low-priority one. Upon eviction from the high-priority queue, blocks are moved to the low-priority one, and eviction from the low-priority queue follows standard eviction. In this scheme, data blocks are cached in the low-priority queue, while filter and index blocks are cached in the high-priority queue and are given a second chance before they are evicted.

Discussion on Partitioned Index and Filters In order to have a more fine granular memory management, systems like RocksDB support partitioned index and filters [43], where each SST file is organized in smaller partitions and each one has its own index and filter blocks. This approach creates the need for a two-level index and necessitates the use of the index prior to the filter, slightly increasing both the space amplification and the CPU utilization to allow for finer memory management. The design of MBF presented in this paper can co-exist with partitioned filters (as long as each filter is more than one disk page), however, it can also be used *instead of* partitioned filters, since it can allow for fine-grained memory usage without increasing space or CPU consumption.

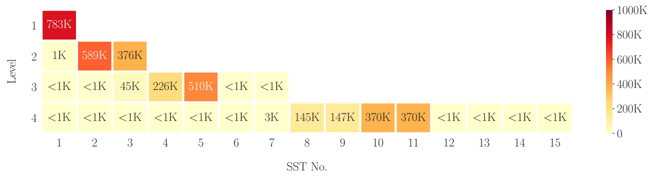
3 MOTIVATION

All BFs Are Not Equally Beneficial State-of-the-art LSM designs treat all BFs as equally beneficial, having a uniform memory allocation policy across all BFs. However, not all BFs are accessed equally frequently. Prior work highlights that even in uniform read patterns, to ensure minimum access cost (in terms of the total number of I/Os), BFs of different levels should be tuned with different false positive rates, hence, with variable bits per element [12, 13]. The intuition is that the SST files at the shallower (smaller) levels are accessed first, hence, they will be accessed more frequently, in the same way that the B$^+$-Tree root node is accessed by every query, and as we move downwards, the access frequency reduces. We take this observation a step further and point out that when the workload is not uniform on the key domain, the access skew of the SSTs is exacerbated (as shown in Figure 2). In addition, the type of point queries that a BF receives (empty vs. non-empty) affects its benefit. As the fraction of empty queries increases, the benefit of BFs gets higher because they help avoid unnecessary I/Os. Conversely, had we known that all queries are seeking existing values, we would not need a BF at all. Putting everything together, the fraction of existing queries that an SST file at level i receives, termed α_i^{ex}, depends on both the level i and the read access distribution. As a result, the benefit of using the BF of a specific SST file depends on both (a) the level it is located and (b) the part of the key domain it contains.

Figure 3 shows a heat map of the access frequency of the SST files of the top 4 levels of an LSM-tree running (a) a uniform and (b) a skewed read workload. For simplicity, we show the first four levels of an LSM-tree with 8 levels and a size ratio of 2. The uniform workload is based on YCSB access patterns, and the skewed



(a) YCSB uniform



(b) Normal (low skewed)

Figure 3: Heat map of access frequency of the SST files of the top 4 levels of an LSM-tree with size ratio 2, for (a) uniform, and (b) skewed reads. In both cases, a few files receive most access requests, hence, their metadata should have priority.

workload is the same as the one used in the introduction, with low skew. Figure 3(a) shows that for any uniform workload, the SST access frequency varies *depending on the level they reside*, following the intuition of prior work that builds BFs with a smaller false positive rate in the top levels [12]. Figure 3(b) shows that for a skewed workload, a small number of SSTs receive the vast majority of the accesses. In level 2, 2 out of 3 SSTs receive more than 99.9% of the accesses, in level 3, 2 out of 7 SSTs receive 94.5% of the accesses (and 3 out of 7 more than 99.99%), and in level 4, 4 out of 15 SSTs receive 99.7% of the accesses. This highlights that not all BFs are equally important. If we only cache the BFs of the darker SSTs from Figure 3, we will already reap most of the BFs benefits. This motivates us to quantify the utility of each BF, and, further, to break down a BF in modules and quantify each module's utility (discussed in §4).

BFs Crack under (Memory) Pressure In a typical LSM-based system the goal is to keep all auxiliary metadata in memory. In state-of-the-art systems like RocksDB, the way to handle memory pressure is to use the *block cache* similarly to how a bufferpool operates. The block cache would keep the most frequently used blocks from SST files, which could either be a data block, or a metadata block (fence pointers or BFs). Most LSM-systems opt to pin the BFs in memory and evict cold data blocks. When memory is insufficient to hold all BFs, the performance degrades quickly. Essentially, access to a BF that is on the disk could be even more expensive than accessing the data block (since a BF is not ensured to fit in one data block) and searching for the target key. Under the standard caching scheme the BF will be fetched and cached, hoping to capture a future empty query on the same SST file. Contrary to traditional database systems that update in-place and an update keeps the pages in the bufferpool, SST files are invalidated upon compaction and the cached Bloom filter will quickly become worthless [44]. This motivates us to quantify the utility of a BF not only using its access frequency but also whether the expectation is to be used as part of empty or non-empty queries.

To showcase how an LSM-tree behaves under memory pressure, we run an experiment using state-of-the-art RocksDB where we measure the average bytes read per (empty) query including fence pointers and BFs, while varying the block cache size between 10% and 100% of the aggregate size of the metadata of the LSM-tree. Before measuring the reported numbers using the low skewed workload, we run a warm-up workload to load the most frequently accessed metadata blocks in memory. Note that a memory budget of 100% means that the block cache fits exactly all metadata blocks. Table 1 shows that as the available memory decreases, the read bytes per query increase rapidly from 12KB to 279KB, a 23.2× increase. For 100% memory budget, we expect that some of the BFs and index blocks might not be in memory since the warm-up workload does not always touch the whole database. The available memory is enough for holding BFs and fence pointers, but as we bring some data blocks, BFs might also be replaced. As the memory budget decreases to 70%, the amount of data read increases to 17KB, which is 1.4× of the 100% case. For a memory budget of 40%, we see a 7.3× increase, and for 10% a 23.2× increase. This experiment further motivates us to design a new BF variant that does not always need to be loaded in its entirety before it can be used to answer queries.

Memory budget:	10%	40%	70%	100%
Reads/lookup:	279KB	88KB	17KB	12KB
Normalized:	23.2×	7.3×	1.4×	1×

Table 1: Lookup cost under memory pressure increases precipitously due to Bloom filters not being always in memory.

Entry	Filter	Index Size (key sizes)					
Size	Size	8B	32B	64B	96B	128B	256B
64B	80KB	0.1×	0.4×	–	–	–	–
128B	40KB	0.2×	0.8×	1.6×	2.4×	–	–
256B	20KB	0.4×	1.6×	3.2×	4.8×	6.4×	–
512B	10KB	0.8×	3.2×	6.4×	9.6×	12.8×	25.6×
1KB	5KB	1.6×	6.4×	12.8×	19.2×	25.6×	51.2×

Table 2: Here we consider a 4MB SST file, $page\_size = 4KB$, and thus, $P = 1024$. Further $bpk = 10$, and we vary K and E. Unless we have a large entry size and a rather large key size, the filter is larger than the index.

Index Size vs. Filter Size. Since both the index and the filter blocks need to be cached in memory to offer fast reads we now discuss the relative size of the filter and the index of an SST file. Note that the size of the filter depends on the overall number of entries in the SST file, while the size of the index depends on the number of data pages. Consider an SST file with P data pages of size $page\_size$ each, entry size E, key size K, and bpk bits per element. The index size is $P \cdot K$ bytes, while the filter size is $P \cdot page\_size/E \cdot bpk/8$ bytes. So if we assume that we have a 4MB data portion of an SST file, an entry size anywhere between 64B and 1KB, a key size between 8B and 256B, and 10 bits per element for the Bloom filter, the index may be anywhere between 0.1× to 51.2× the filter size, as shown in Table 2. While the index is not always smaller than the filter in an SST file, in this paper, we focus on managing the filter blocks for two reasons: (a) in most practical use-cases the index is indeed smaller than the filter, and (b) the index needs to be fully accurate when answering queries. Further, the index can be optimized via partitioning, essentially, by building a packed two-level search tree.

4 MODULAR BLOOM FILTERS

Next, we present Modular Bloom filters that divide a BF into multiple modules to flexibly navigate its space vs. accuracy tradeoff.

4.1 The Structure of a Modular Bloom filter

A Modular Bloom filter (MBF) uses m bits to index n elements in each of D modules. Each module uses m_d bits such that $\sum_{d=1}^{D} m_d = m$. Essentially, an MBF is a collection of D Bloom filters, and every membership test has to go through all modules before it concludes with a positive result if all modules have to be used. On the other hand, a negative response at any module terminates the query without the need to further continue probing the remaining modules. Figure 4 compares a Modular Bloom filter, which is composed of three modules, with a standard BF.

Every module is also a Bloom filter, hence, a lookup can use any of the available modules without re-indexing. Thus, MBF can maintain only the selected modules in fast memory, leading to smaller memory consumption at the expense of a higher false positive rate (without recalculating the filter). There have been several approaches that divide a Bloom filter into smaller chunks [24, 30, 32, 39, 45, 48]. The innovation for MBFs is that by indexing all elements in all modules, the membership test can move forward with any subset of the modules.

Different modules of an MBF may have arbitrary sizes, however, in practice, it may be easier to consider equally-sized modules. Consider the case of a classic BF that uses 10 bits per element (a typical value for BFs deployed in LSM-trees). The number of index k_{opt} probes that minimize the false positive rate f is given by

$$f = \left(\frac{1}{2}\right)^{\frac{m}{n}\ln 2} \approx \left(1 - e^{-kn/m}\right)^{k} \quad \text{where} \quad k_{opt} = \frac{m}{n}\ln 2 \qquad (1)$$

and for $\frac{m}{n} = 10$, $k_{opt} = 7$. Every empty query on a BF will return a negative result (or a positive result with a false positive probability f). For the case of $k_{opt} = 7$, the average number of index probes before it finds a bit set to 0 and terminates is 1.93 (further discussed below). Hence, an MBF can have a module with m_1 bits and $k_1 = 2$ index probes to capture the bits needed on average before a negative query terminates, and then load the remaining modules when the first result is positive. Next, we show that while the size of the modules can vary, in order to minimize the overall false positive, each k_i has to follow the same rule as in Eq. (1).

MBF False Positive. The false positive rate of an MBF is equal to the product of the false positive rate of each module, which is equal to that of a smaller Bloom filter using Eq. (1).

$$f_{MBF}(\{m_d\}, \{k_d\}) = \prod_{d=1}^{D} f_d = \prod_{d=1}^{D} \left(1 - e^{-k_d \cdot \frac{n}{m_d}}\right)^{k_d} \qquad (2)$$

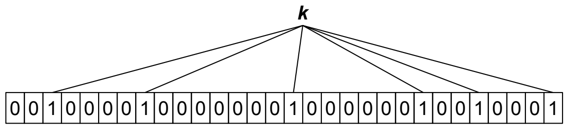
The problem of minimizing f_{MBF} from Eq. (2) is equivalent to minimizing the logarithm of the false positive $\ln(f_{MBF})$.

$$\min_{\{m_d, k_d\}} \sum_{d=1}^{D} k_d \ln(1 - e^{-k_d n/m_d}), \quad s.t. \sum_{d=1}^{D} m_d = m \qquad (3)$$

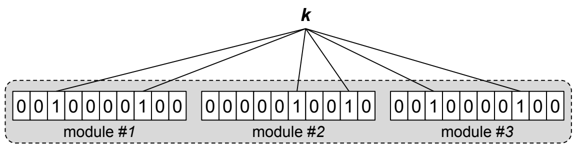
Using the Lagrangian multiplier we get that in order to minimize the false positive of an MBF we need to have:

$$\forall d, \quad k_d = \frac{m_d}{n} \cdot \ln 2 \qquad (4)$$

This means that an MBF is a generalization of the classic Bloom filter, and when there is one module ($D = 1$) an MBF is a classic BF. In addition, the false positive of an MBF with optimal k_d, $\forall d$



(a) A standard Bloom filter.



(b) A modular Bloom filter with three modules of equal size.

Figure 4: Modular Bloom filters split the physical representation of a BF into multiple independent modules.

is identical with the false positive of a classic BF with the same overall number of bits available:

$$f_{MBF}^{opt} = \prod_{d=1}^{D} e^{-\frac{m_d}{n}\cdot(\ln 2)^2} = e^{-\frac{\sum_{d=1}^{D} m_d}{n}\cdot(\ln 2)^2} = e^{-\frac{m}{n}\cdot(\ln 2)^2} = f_{BF}^{opt} \qquad (5)$$

Average Filter Probes for an Empty Query. Now we revisit the question of how fast an empty query terminates. First, we note that the optimal number of indexes (k_{opt}) that minimizes the false positive rate, leads to a BF that has half its bits set, as shown in Eq (1). After the first probe, with probability 1/2 there will be a second one. Now, in case we had a second probe with a conditional probability 1/2 there will be a third probe, with a total probability of $1/2^2$. This logic continues until we reach the k^{th} probe with total probability $1/2^{k-1}$. Hence, the expected number of filter probes can be given by the geometric-like series in Eq. (6).

$$probes_{empty} = \sum_{d=1}^{k} \frac{1}{2^{d-1}} = 2 - \frac{1}{2^{k-1}} \qquad (6)$$

For every k, $probes_{empty}$ is less than 2, hence, an MBF with the first module that accounts for 2 index probes ($k_d = 2 \Rightarrow \frac{m_d}{n} \approx 3$) is enough to account for the average index probes of an empty query. If the first module accounts for 3 filter probes then it will capture the $\sum_{d=1}^{3} \frac{1}{2^d} = 0.875$ of all the accesses without requiring fetching the remaining modules.

Experimental Verification of False Positive Ratio. In practice, if we have a memory budget for Bloom filters, for example, 10 bits per key, we do not want to break it down to too many modules. We want to map at least one index probe per module, so for $\frac{m}{n} = 10$ and $k_{opt} = 7$, we would choose 1, 2, 3, or 7 modules, depending on the flexibility we want to achieve. We implement MBFs with 1, 2, 3, or 7 modules, using accordingly the following number of index probes per module: $\{7\}$, $\{3, 4\}$, $\{2, 2, 3\}$, and $\{1, 1, 1, 1, 1, 1, 1\}$. Table 3 compares the theoretical expectation and the experimentally measured false positive of a Modular Bloom filter that uses 10 bits per element. The experiment indexes 500K elements in $5 \cdot 10^6$ bits ($\sim$ 625KB), and measures the average false positive after running 2M empty queries. We observe that the difference from the theoretical false positive (which is the same as for the classical BF with 10 bits per element) is negligible. The last line of Table 3 shows the average number of module accesses per empty query. We discussed above that in a classical Bloom filter the average number of index

Modules:	1	2	3	7
Theoretical FP:	0.819%	0.819%	0.819%	0.819%
Experimental FP:	0.825%	0.847%	0.887%	0.830%
Avg. Module Accesses:	1	1.093	1.251	1.997
Module Size as % of the BF:	100%	50%	33%	17%

Table 3: MBFs achieve a false positive rate close to the theoretical rate of a BF with the same aggregate size. As MBFs have more modules, the average number of module accesses grows, however, the average size of each module decreases.

symbol	Explanation
$SST_{l,i}$	i-th SST file in level l
$\alpha_{l,i}$	the ratio of existing keys for $SST_{l,i}$
$\beta_{l,i}$	the access frequency for $SST_{l,i}$
$u_{l,i,d}$	The utility of the d-th module for $SST_{l,i}$
$expIO_{l,i,d}$	The expected number of I/O when using d modules for $SST_{l,i}$

Table 4: Notation for calculating the *utility* of BFs

probes per empty query is less than two. However, when the MBF has one module (hence, it is a classical BF), it has to load the entire module, while, when we have two modules, it loads on average 1.093 modules, meaning that most queries are answered using only the first module (which is 40% of the memory). Finally, in the extreme case of having 7 modules and one index probe per module, we only have to load on average 2 modules out of 7 (28% of the memory footprint). This allows for more efficient memory utilization in multiple ways. For example, even when all BF modules are loaded in memory the cache memory is not polluted with all modules. When the modules do not all fit in memory, the system can decide to pin some modules and sporadically use the last modules that have a marginal benefit. We further discuss the applicability of MBF for full-blown systems in the next section.

4.2 Modular Bloom Filters In LSM-trees

Modular Bloom filters offer a flexible alternative to Bloom filters for LSM-trees. Instead of relying on a monolithic fixed-size Bloom filter, LSM-trees can navigate the memory availability vs. performance continuum by providing the MBFs a specific memory budget to use. MBFs are different as each level or even each SST file can make an independent decision as to whether it will use all or a subset of its modules. In practice, because the metadata blocks of realistic systems are used in equal-sized pages which are also part of the SST file, we resort to MBFs with equal-sized modules. Note that we also implement MBFs with modules of various sizes in the evaluation. Typically, in our experiments, an MBF of a single SST has two or three modules each one occupying one or more pages. In the examples discussed below, we assume that each SST file has an MBF with three modules of the same size. This leads to a simplification with respect to the expected false positive contribution per single module f_{sm}, which is expected to be also equal $f_1 = f_2 = f_3 = f_{sm}$. Hence, the false positive rate of the MBF f_{MBF} is given as follows.

$$f_{MBF} = f_1 \cdot f_2 \cdot f_3 = f_{sm}^3 \quad (7)$$

Table 4 summarizes the notation for the following *utility* and skipping algorithm.

BF *Utility.* As we have seen in Section 3 not all BFs are equally beneficial, because their access frequency varies widely. In addition, the benefit of a BF comes from the unnecessary I/Os avoided for

```
QueryMBF (key k, $SST_{l,i}$)
    for $d = 1$, $d \leq$ number of modules, $d$++ do
        //calc module's utility
        $\text{expIO}_{l,i,d} = \beta_{l,i} \cdot (\alpha_{l,i} + (1 - \alpha_{l,i}) \cdot f_{sm}^d)$;
        $u_{l,i,d} = \text{expIO}(l, i, d) - \text{expIO}(l, i, d-1)$;
        if $skip_d$==true & $u_{l,i,d} < threshold_d$ then
            // skipping module, assumes that it returns positive
            return true;
        else
            // probe the module like a mini BF
            // this part might cause an I/O if the module is not cached
            result = QueryModule($k$, $\text{module}_{l,i,d}$)
        end
        if result==false then
            return false;
        end
    end
    return result;
end
```

Algorithm 1: Querying an MBF uses the utility of each module (along with utility thresholds per module) to decide whether accessing a module is beneficial.

empty queries. However, there is still a lot of room for improvement. To fully exploit the multiple modules of MBFs, we quantify the *utility* of each module and design a new module management policy. We define the *utility* as follows. We consider for every SST file i at level l two quantities to calculate the potential benefit from a BF: (1) the ratio of existing keys $\alpha_{l,i}$, and (2) the access frequency $\beta_{l,i}$. The utility of a module estimates the impact on the expected number of I/Os when the specific number of modules is in use. For a module d on an MBF at level l of SST file i, the utility is given as follows:

$$u_{l,i,d} = \text{expIO}_{l,i,d} - \text{expIO}_{l,i,d-1} \quad (8)$$

where $\text{expIO}_{l,i,d}$ is the expected number of I/Os when using d modules at the specific SST file.

$$\text{expIO}_{l,i,d} = \beta_{l,i} \cdot \left(\alpha_{l,i} + \left(1 - \alpha_{l,i}\right) \cdot f_{sm}^d\right) \quad (9)$$

Eq. (9) quantifies the number of I/Os at a specific SST file by taking into account the popularity of the file $\beta_{l,i}$, the amount of true positive queries reaching this SST file $\alpha_{l,i}$, and the false-positive ratio as a function of the number of the modules (d) used for that file. Note that for any SST file, the expected I/O when using no modules is $\text{expIO}_{l,i,0} = \alpha_{l,i} \cdot \beta_{l,i}$, and that the total number of base data accesses is given by the fraction of positive queries $\alpha_{l,i}$ summed up with the negative queries for which there is a false positive $(1 - \alpha_{l,i}) \cdot f_{sm}^d$.

State-of-the-art systems maintain statistics about the number of accesses already. We add another counter for positive queries. In case an SST file was just created after compaction or flushing, the necessary quantities can be approximated through the corresponding metadata of overlapping files from other levels as follows. The popularity of the SST file can be calculated using the popularity of the overlapping files from the last level, and the ratio of positive lookups per SST file can be calculated using the ratio of positive lookups in the overlapping files from the last level divided by a factor of T for each level between the current level and the last one.

Skipping Modules Using Their *Utility.* The expected number of I/Os and the *utility* of a module, help us estimate its benefit. Data access is inevitable when the expected number of I/Os is equal to one, therefore, accessing the module is unnecessary. Hence, we propose to skip probing modules if the expected number of I/Os is over a certain threshold. When the block cache is large enough to hold all available BFs there is no benefit from skipping a module. However, with limited memory, probing a module with low utility may evict from the cache other modules with higher utility. Therefore, skipping low-utility modules reduces the stress of block cache competition and increases the probability of high-utility modules staying in the cache. Note that the utility of a filter module corresponds to the number of I/Os that it can save if used. Algorithm 1 shows how to query an MBF and how to skip modules using their utility. The core idea of the algorithm is that if a module is expected to lead to an I/O anyway (combining the frequency of the accesses, and the frequency of queries being non-empty on the specific SST file), the system will prefer to go directly to the data since the I/O is inevitable (if we refer to the last module). As the utility of modules in different orders is not comparable, we allow the algorithm to use a different threshold per module slot. Moreover, it is also possible to skip only a subset of the modules since their utility decreases as more modules are accessed. As shown in Algorithm 1, the decision to skip is made per module $skip_d$. Note that the modules are always accessed sequentially, thus the decision to skip the i-th module affects the rest of the modules.

When are MBFs Feasible? In order to be able to employ MBFs, we need to guarantee that the aggregate size of Bloom filters of an SST file is equal to or larger than the number of modules intended to use. Specifically, the following equation should hold true:

$$\frac{\frac{\text{fileSize}}{\text{keyValueSize}} \cdot \frac{\text{bpk}}{8}}{\text{pageSize}} \geq \text{numModules} \tag{10}$$

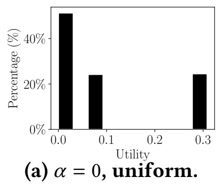
This equation can guide the LSM-Tree tuning and specifically the file size picked, a parameter that is often neglected when discussing tuning.

$$\text{fileSize} \geq \frac{\text{numModules} \cdot \text{pageSize} \cdot \text{keyValueSize}}{\frac{\text{bpk}}{8}} \tag{11}$$

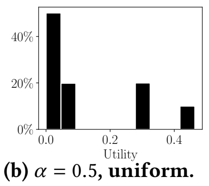
For example, to support 2 modules for a system with a 4KB page size, 256KB key-value entry size, and 10 bits per element, the file size has to be larger than 1.6MB. Note that MBF also can be implemented for partitioned index/filter if Eq. (10) is satisfied. Eq. (10) can be easily modified for partitioned index/filter by simply modifying the SST file size to be equal to the size of each partition.

How to Choose the Thresholds for Skipping Algorithm. To decide the skipping threshold, we conduct a micro-benchmark to measure the observed utility distribution. Figure 5 shows the histogram of utility modules for uniform access patterns while varying the ratio of empty queries α. As α grows, the range of the utility widens. However, even when the workload has only empty queries ($\alpha = 1$), the utility of different modules is not static because the SST access frequency is not uniform even for uniform access distribution (Figure 3). For simplicity, we use as threshold $utility = 0.2$, which in our micro-benchmark serves as a clear separation point of the modules with zero utility and the remaining modules.

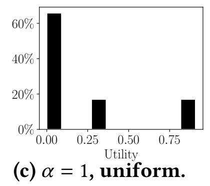
Updating α and β. Our proposed skipping algorithm is based on the statistical information per SST file. We assume that these statistics are inherited during compactions and flushes as discussed earlier. During query execution, we combine the inherited (α_i and β_i) and measured (α_m and β_m) statistics. In order to cope with the workload shifts, we assign more weight (w) to α_m and β_m as the age of SST files increases as per Eq. (12). Note that w is a function of time; when the SST file is just created, the weight is zero ($w(0) = 0$), but it is tuned to choose the measured statistics when the age of the SST file is over a predefined threshold. That way, we have the time needed to collect enough query statistics without sacrificing accuracy in the meantime.



(a) $\alpha = 0$, **uniform.**



(b) $\alpha = 0.5$, **uniform.**



(c) $\alpha = 1$, **uniform.**

Figure 5: Histogram of *utility* of uniform distribution with various α. As the ratio of empty queries (α) grows, the range of utility also increases. However, the utilities of modules even for all empty queries are not equally beneficial since the access frequency of SST files varies.

$$\alpha = \alpha_m \cdot w(t) + \alpha_i \cdot (1 - w(t)) \tag{12}$$

The same approach is used to approximate β as well.

Different Priorities for Modules. Another advantage of MBF over BF is that modules can have different priorities. The different modules of an MBF have different access frequencies; therefore, their utility and hence, their probability of staying in the cache also differs. In addition, since practical systems like RocksDB today support multiple priority queues when caching, we explicitly assign different priorities to modules. Specifically, we assign high priority to the first module, and low priority to all other modules.

5 EXPERIMENTAL EVALUATION

In this section, we present the experimental evaluation and analysis of SHaMBa. We demonstrate that SHaMBa can significantly outperform the state of the art when there is memory pressure.

Experimental Platform. We run our experiments in our in-house server, which is equipped with two sockets each with an Intel Xeon Gold 6230 2.1GHz processor with 20 hardware threads and 40 threads with virtualization enabled. The server is equipped with two 7200RPM hard drives and one off-the-shelf SSD In our experiments we use the SSD devices as secondary storage.

Workloads. We conduct our experiments with different workloads to stress-test all our approaches. Since our algorithms only affect read performance, we first prepare an LSM-Tree with 1GB of key-value pairs with 64B entry size, 4MB write buffer, and size ratio 4. We then execute a read-only workload varying the data access skew, and the fraction of empty lookups. With respect to skew, we vary the access patterns to follow (a) a uniform, (b) a normal distribution with a standard deviation of 5 keys in the key space, or (c) a Zipfian distribution with a skew factor of 2. A pictorial representation of the access frequency in the domain is shown in Figure 2(a).

Note that for queries that return a positive result, any BF access is essentially an overhead that we have to pay to optimize for the empty queries. Our skipping algorithms from Section 4.2 are addressing exactly this problem and they attempt to reduce the

Term	Value	Explanation
E	64	entry size (B)
K	32	key size (B)
B	64	block size (#entries)
P	1024	buffer size/file size (#blocks)
T	4	size ratio
b	10	bits per key for filters
S_D	4KB	data block size
S_I	32KB	index block size
S_F	80KB	filter block size

Table 5: Experiment Settings

Term	Value
number of modules	1, **2**, 3, or 7
size of each module	**equal** or proportional
skipping algorithm	none, partially ($\mathcal{P}$), or **fully ($\mathcal{F}$)**

Table 6: Experimental setup and knobs of SHaMBa

overhead of BF for positive queries without losing their benefit for empty queries. In order to evaluate the efficacy of skipping, we also test workloads that vary the fraction of empty queries between 100% ($\alpha = 0$), 50% ($\alpha = 0.5$), or 0% ($\alpha = 1$). As discussed in Section 4.2, the fraction of empty queries drastically varies in different parts of the domain and in different levels of the LSM-Tree (i.e., each SST file maintains a separate account of the empty queries it observes).

System Settings. Table 5 summarizes the LSM tuning parameters that we used in our experimentation. The table includes some specific LSM-tree parameters that relate to the block cache to better explain the impact of caching. Specifically, we enable the block cache with LRU as an eviction policy, and in our experiments, we vary its capacity between 10% and 150% of total metadata size.

Metrics. SHaMBa does not affect data ingestion hence we focus on analyzing its query performance. For each experiment, we populate the data and then repeat read-only experiments five times. We report the average number of I/Os and the latency per query.

Approaches Tested. We compare SHaMBa with our in-house LSM-tree prototype based on the state-of-the-art RocksDB [18] to showcase the benefits of Modular Bloom filters. SHaMBa has multiple tuning knobs such as the number of modules, the size of each module (modules can have identical or variable size), lookup policies (with or without the skipping algorithm), and thresholds for the skipping algorithm. In order to thoroughly evaluate the SHaMBa, we conduct four sets of experiments in the following subsection. First, we conduct two sets of experiments that vary two tuning knobs of SHaMBa (Table 6): the way to choose the size of modules (SHaMBa-eq or SHaMBa-prop) and the number of modules. Additionally, we implement and evaluate our SHaMBa with partitioned index and filter and Monkey. Finally, we evaluate SHaMBa implemented in RocksDB. In each set of experiments, we test two main variations of SHaMBa, one that has modules of equal size (SHaMBa-eq), and one that has modules with variable size (SHaMBa-prop) . The latter is always tuned so the total size of the first module can fit in memory to avoid eviction, that is, its size is *proportional* to the available memory. In addition, for each of the two module sizes, we compare three lookup policies: (i) one that uses all available modules, (ii) one that always queries the first part of modules and based on their utility decides whether to skip the second part ($\mathcal{P}$), and (iii) one that might decide to skip all modules ($\mathcal{F}$). Overall, we test the following systems:

- *state-of-art*: an LSM-engine that uses a single BF per SST file.
- *SHaMBa-eq*: uses MBFs with multiple equal-size modules.
- *SHaMBa-eq-$\mathcal{P}$*: uses MBFs with multiple equal-size modules, and conditionally skips the part of modules if the *utility* of modules is less than the predefined thresholds.
- *SHaMBa-eq-$\mathcal{F}$*: uses MBFs with multiple equal-size modules, and conditionally skips full or part of modules if the *utility* of modules is less than the predefined thresholds.
- *SHaMBa-prop*: uses MBFs with size proportional to the available memory, so that the first modules fit in the cache.
- *SHaMBa-prop-$\mathcal{P}$*: uses MBFs with size proportional to available memory, and conditionally skips part of modules if the *utility* of modules is less than the predefined thresholds.
- *SHaMBa-prop-$\mathcal{F}$*: uses MBFs with size proportional to available memory, and conditionally skips full or part of modules if their *utility* is less than the predefined thresholds.

5.1 MBFs Under Memory Pressure.

We now show how SHaMBa performs compared to our in-house LSM-tree prototype based on the state of the art LSM-engine. We compare the lookup performance of SHaMBa against the state of the art. For each experiment, before we report measurements, we run 30K read queries with the same characteristics to warm up the block cache. Note that the metadata for module skipping is populated and maintained throughout the warmup phase. SHaMBa integrates Modular Bloom filters to provide performance that is resilient to memory pressure, diverse workloads, and access patterns.

5.1.1 Equal-sized modules vs. proportional sized modules.

In this set of experiments, we use two modules and compare the two main variations of SHaMBa: SHaMBa-eq and SHaMBa-prop. We compare three lookup policies discussed earlier: (1) one that always uses two modules, (2) one that always queries the first module and using the utility might decide to skip the second module ($\mathcal{P}$), and (3) one that might decide to skip both modules ($\mathcal{F}$). Unless otherwise noted, the thresholds used for the two skipping algorithms are 0.1 for skipping the second module and 0.2 for skipping the entire filter. The rationale behind the thresholds is that when we are using the partial skipping algorithm ($\mathcal{P}$), the first module is already positive and we have to do an I/O for the second module as well. With respect to performance, it is inconsequential whether the I/O happens for the second module or the data, so, we have an aggressive threshold. For full skipping ($\mathcal{F}$) we test whether to skip both modules; so, we ensure that with high probability, both modules are not in memory or the query is indeed positive. Thus, we use an increased threshold.

SHaMBa-eq Outperforms the State of the Art. Figures 6(a) through (i) compare SHaMBa-eq with the state of the art as we vary the access distribution (uniform, normal, Zipfian), and the fraction of non-empty queries α (0, 0.5, 0.1). The x-axis of each graph shows the available memory budget that varies between 10% and 150% of the total size of the fence pointers (index blocks) and Bloom filters (filter blocks) assuming 10 bits per element. The y-axis is the number of I/Os per lookup. Note that the latency graphs follow very similar patterns but we show the I/O that is the main

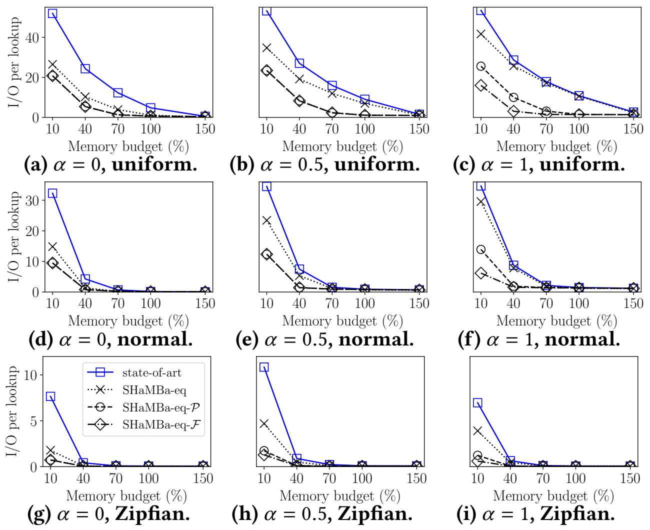


Figure 6: SHaMBa-eq reduces the I/O per lookup for different access distributions and exploits workload monitoring to avoid fetching modules with low utility.

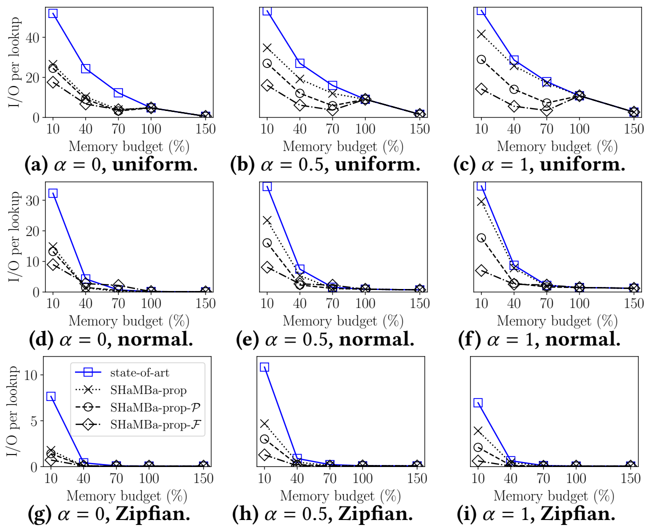


Figure 7: SHaMBa-prop matches state of the art for memory ≥100%, however, as the first modules always fit in cache, it outperforms all approaches under severe memory pressure.

bottleneck as we encounter memory pressure. In every graph, we observe that the variant of SHaMBa (black lines) consumes a significantly lower number of I/Os per lookup than the state-of-art (blue line) for a memory budget of less than 100%. This is on par with our expectation - when the available memory is enough for all filter blocks, MBFs and utility-based skipping is not necessary. As we reduce the available memory, however, modules in MBFs allow the BFs to remain useful even when they only partially fit in the cache. In Figure 6(a) we observe that for all empty queries on uniform data distribution, by retrieving a smaller number of modules based on their utility we reduce unnecessary expensive I/Os. In Figures 6(b) and (c) we see this trend continuing, however, we now observe a differentiation between SHaMBa-eq, SHaMBa-eq-$\mathcal{P}$, and SHaMBa-eq-$\mathcal{F}$. Here, as more queries access data that exist (and have always positive results), we avoid fetching an increased number of modules using their utility. For the more skewed distributions in Figures 6(d)-(i) we see similar results but SHaMBa's benefits kick in only at smaller block cache sizes because skewed accesses require a smaller cache to capture the working set.

SHaMBa-prop Shines under Severe Memory Pressure. Figures 7(a)-(i) show how SHaMBa-prop outperforms the previous approach under extreme memory pressure. The key difference between SHaMBa-prop and SHaMBa-eq is that the size of the first module in SHaMBa-prop depends on the total available memory. If the memory budget is 70%, the bits-per-element of the first module is 7, and for the second module 3, to always have enough memory for all first modules. Note that when the memory budget is greater or equal to 100%, SHaMBa-prop is identical to the state-of-the-art.

We observe trends similar to the previous experiments with respect to access distribution and workload. SHaMBa-prop has one key difference when compared with SHaMBa-eq. The first modules always fit in the cache, so when the system is under extreme memory pressure (e.g., memory budget <50%), it outperforms SHaMBa-eq. This is further pronounced for SHaMBa-prop-$\mathcal{P}/\mathcal{F}$.

MBFs are Beneficial even with High Memory Budget. A key outcome of the experiments presented in Figure 6 is that SHaMBa-eq is beneficial when compared to the state of the art even when available memory budget is more than 100% – e.g., Figures 6(a)-(c). This is because even though the cache is large enough to hold all index and filter blocks, they still compete with data blocks especially when non-empty queries form a large fraction of the workload. On the contrary, SHaMBa-prop reverts to the standard single module when the memory budget exceeds 100%. Taking into account that SHaMBa-prop is beneficial mostly for very low memory budget, we can select to use SHaMBa-eq when the memory budget is at 50% or more of the index and filter size, and SHaMBa-prop when the expected memory budget is smaller.

Skipping Boosts Existing Lookups. A key motivation of our work is that BFs are not equally beneficial. BFs are useful for empty queries, and less so, for existing queries. However, our module skipping algorithm takes the existing ratio as an input parameter when quantifying the utility of a BF module. That way, even when the existing ratio increases (α grows), skipping modules avoids accessing unnecessary blocks. Note that even a query on an existing key will perform searches for keys that do not exist in some levels. Throughout the experiments shown in Figures 6 and 7, we observe that the skipping algorithm reduces the number of I/Os per lookup significantly both for lower and higher values of α.

5.1.2 *Impact of number of modules.*

We now evaluate the impact of varying the number of modules for SHaMBa. For simplicity, we test only the equal-sized modules. We implement MBFs with 1, 2, 3, or 7 modules as shown in Table 3. Here, we apply the skipping algorithm to all modules.

SHaMBa Performs Best with Smaller Modules. Figure 8 shows that the smaller the module size the higher the performance for SHaMBa-eq, because it allows for memory management at a finer granularity. Note that the smaller module size corresponds to an

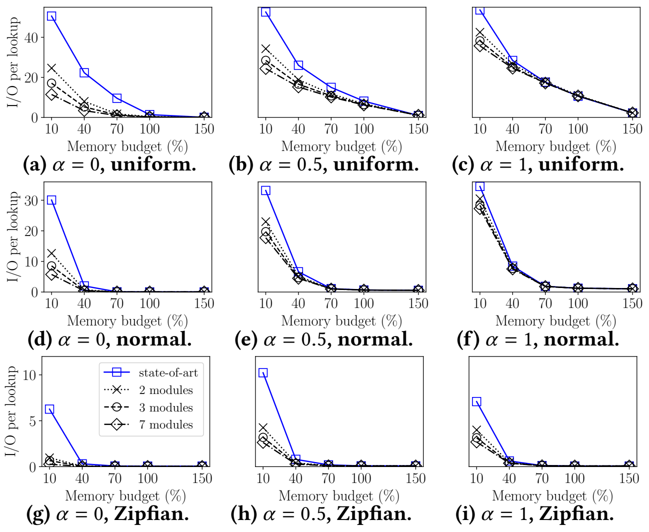


Figure 8: SHaMBa-eq achieves the minimal number of I/Os with smaller modules.

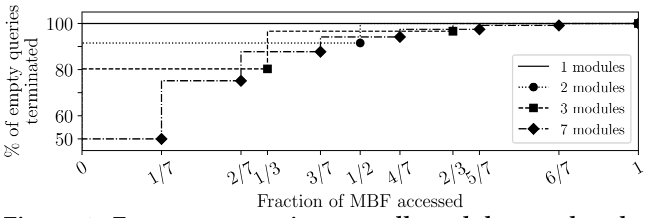


Figure 9: For empty queries, not all modules need to be accessed. For a 2-module MBF, the percentage of empty queries that terminate after accessing only one module is 91%. Similarly, for a 7-module MBF, 50% of the queries terminate after accessing only one module, and more than 94% terminate after accessing four modules (57% of the MBF). This allows us to have a finer control of the space vs. performance trade-off as the number of modules grows.

MBF with seven modules. In Section 4.1, we conducted a microbenchmark to show the average number of module accesses for empty queries in Table 3. We observe that the average number of module accesses increases as the number of modules grows; however, the size of each module decreases, leading to fewer overall I/Os for smaller modules. Figure 9 shows on the y-axis the fraction of module accesses terminating after accessing each module for uniform empty queries. A classical BF (having one module), needs to access the entire BF to answer any query. A MBF with 2 modules, can answer ∼91% of the empty queries only using the first module, that is, 50% of the overall BF. Similarly, if we focus on the 7-module MBF, ∼74% of the queries terminate using only 2 modules (∼29% of the BF), while more than 94% of the queries terminate by using 4 modules (accessing 57% of the BF). Overall, as we increase the number of modules, we can perform more fine granular memory management. In practical implementations, we create MBFs with two modules, since having a very small module size is not always feasible as shown by Eq. (10).

Aggressively Skipping Modules Reduces the Impact of the Number of the Modules. Figure 10 shows the results from the

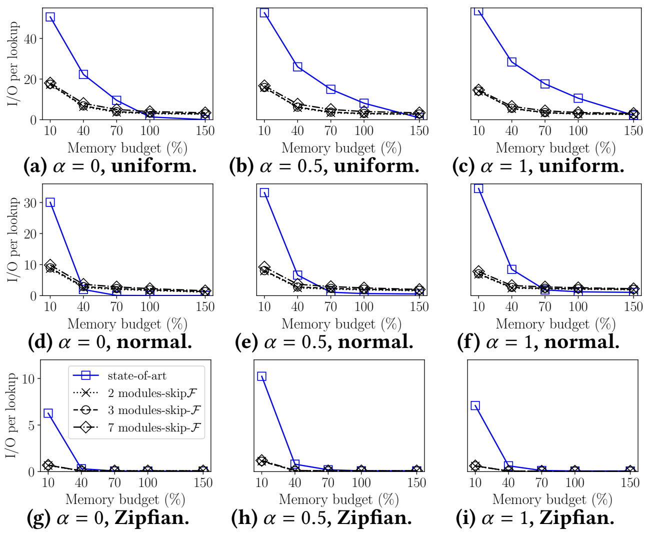


Figure 10: SHaMBa-eq-$\mathcal{F}$ shows the number of modules has a smaller impact if we allow to skip any number of modules based on their utility.

same set of experiments with our full skipping algorithm ($\mathcal{F}$) as we vary the number of modules. SHaMBa$-eq-\mathcal{F}$ evaluates each module's utility before accessing it, and skips it if there is not enough benefit. We observe that SHaMBa$-eq-\mathcal{F}$ reduces the impact of having more modules in the MBF, which further reinforces our decision to have two modules in practical implementations.

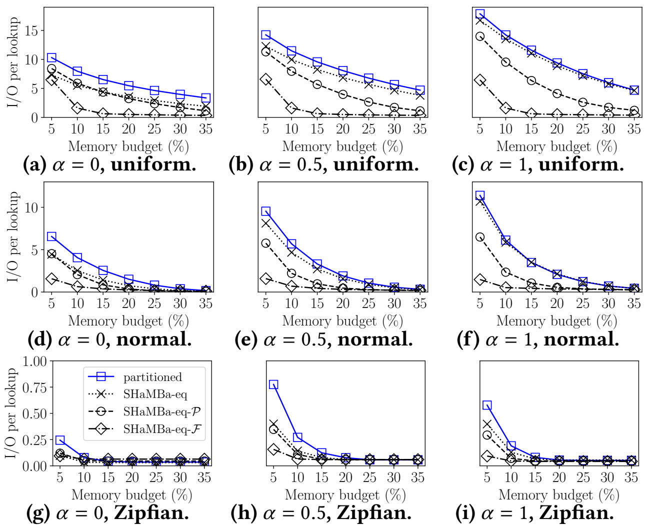
5.1.3 *Experiments with Partitioned Index/Filter.*

In the experiment until now, we use a single BF per SST. In our setting, the index amounts to eight 4KB data pages (index size is $P{\cdot}K$; thus, 32KB). If we use partitioned index and filter [43] to ensure that the index of a partition fits in a page, we end up with eight partitions. We now experiment with this setup, where each SST file is partitioned eight ways, and the BF of each partition is 10KB, spanning 3 pages (filter size is $\frac{bpk}{8} \cdot B \cdot P/\text{number\_of\_partitions}$; i.e. 10KB). As a result, we can employ MBF on top of Partitioned SST since Eq. (10) would still be satisfied. Since partitioning the BF is already addressing memory pressure, we focus our experiments on a smaller memory budget between 5% and 35%. We use MBFs with two equally sized modules in these experiments.

SHaMBa Boosts Partitioned Index/Filter under Severe Memory Pressure. Figure 11 compares the number of I/Os per lookup of the state-of-the-art with partitioned index and filters and SHaMBa on partitioned index and filter. The results highlight that all variants of SHaMBa reduce the number of I/Os, as they enable fine-grained cache management. Note that SHaMBa-eq-$\mathcal{F}$ outperforms all variants as it aggressively skips modules with low utility, thus, reducing I/Os per query under severe memory pressure.

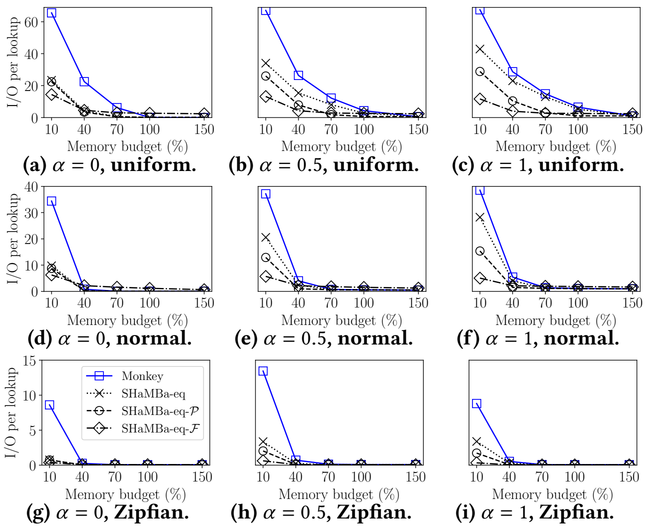
5.2 Experiments with Monkey

In our experimentation, we also consider Monkey [12], that allocates more bits per element in the shallower levels to aggressively reduce their false positives with a very small penalty for the last level, leading to fewer overall I/Os per query. Since the first few levels have larger BFs, Monkey is also a good candidate for MBFs.



(a) $\alpha = 0$, **uniform.** (b) $\alpha = 0.5$, **uniform.** (c) $\alpha = 1$, **uniform.**
(d) $\alpha = 0$, **normal.** (e) $\alpha = 0.5$, **normal.** (f) $\alpha = 1$, **normal.**
(g) $\alpha = 0$, **Zipfian.** (h) $\alpha = 0.5$, **Zipfian.** (i) $\alpha = 1$, **Zipfian.**

Figure 11: SHaMBa reduces I/Os per query for partitioned index and filter, especially under severe memory pressure.



(a) $\alpha = 0$, **uniform.** (b) $\alpha = 0.5$, **uniform.** (c) $\alpha = 1$, **uniform.**
(d) $\alpha = 0$, **normal.** (e) $\alpha = 0.5$, **normal.** (f) $\alpha = 1$, **normal.**
(g) $\alpha = 0$, **Zipfian.** (h) $\alpha = 0.5$, **Zipfian.** (i) $\alpha = 1$, **Zipfian.**

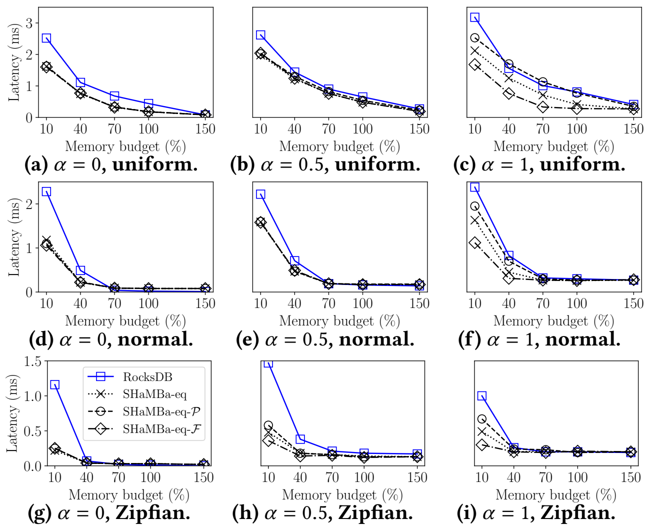
Figure 12: SHaMBa-eq reduces the lookup latency of Monkey under memory pressure.

SHaMBa Further Improves Monkey's Performance. Because of the large BFs, Monkey performs worse than state-of-the-art under extreme memory pressure. Figure 12 shows that SHaMBa improves Monkey's lookup performance significantly. As shown in Table 3 and Figure 9, the average number of modules accesses when using two modules is close to one. Thus, the number of I/Os for empty lookups is halved. For existing queries, SHaMBa-eq-$\mathcal{P}/\mathcal{F}$ reduce the filter accesses by avoiding modules with low utility.

5.3 SHaMBa with RocksDB

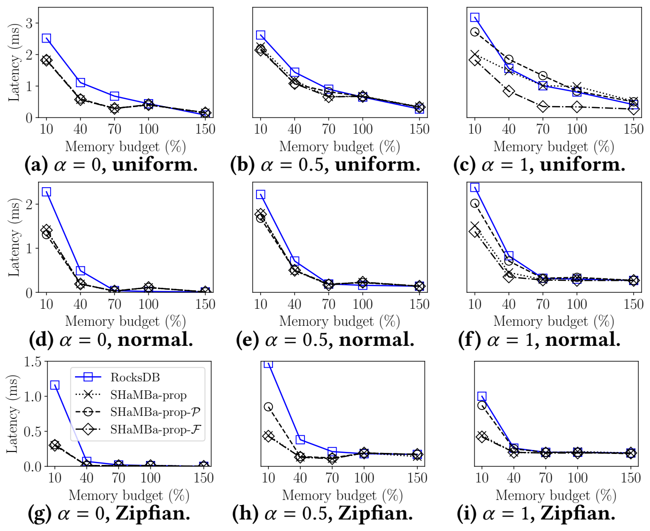
For our last experiment, we integrate our approach into RocksDB (version 6.19.3), a state-of-the-art LSM engine, to showcase the benefits of Modular Bloom filters. We run the same set of experiments of Section 5.1.1, and report the average latency per lookup.

SHaMBa Accelerates Point Lookups. Figure 13 compares the lookup latency of SHaMBa-eq to RocksDB engine. The experimental



(a) $\alpha = 0$, **uniform.** (b) $\alpha = 0.5$, **uniform.** (c) $\alpha = 1$, **uniform.**
(d) $\alpha = 0$, **normal.** (e) $\alpha = 0.5$, **normal.** (f) $\alpha = 1$, **normal.**
(g) $\alpha = 0$, **Zipfian.** (h) $\alpha = 0.5$, **Zipfian.** (i) $\alpha = 1$, **Zipfian.**

Figure 13: SHaMBa-eq reduces the lookup latency of RocksDB under memory pressure.



(a) $\alpha = 0$, **uniform.** (b) $\alpha = 0.5$, **uniform.** (c) $\alpha = 1$, **uniform.**
(d) $\alpha = 0$, **normal.** (e) $\alpha = 0.5$, **normal.** (f) $\alpha = 1$, **normal.**
(g) $\alpha = 0$, **Zipfian.** (h) $\alpha = 0.5$, **Zipfian.** (i) $\alpha = 1$, **Zipfian.**

Figure 14: SHaMBa-prop improves the lookup performance of RocksDB especially under severe memory pressure.

results using RocksDB show the same trends as Figure 6 for the I/Os per lookup. Particularly, our skipping algorithms effectively reduce the lookup latency when there is memory pressure, and the benefits persist for both empty ($\alpha = 0$) and non-empty queries ($\alpha = 1.0$). Similarly, Figure 14 compares the lookup latency of SHaMBa-prop to RocksDB. The results are noisier because it is hard to account perfectly accurately for the memory usage of RocksDB and SHaMBa-prop relies on having the first module always in memory. In the big picture, however, SHaMBa-prop significantly reduces the lookup latency under memory pressure, especially when also considering the full skipping algorithm.

SHaMBa Performs Best When Filters Are Larger Than Indexes. We now show how SHaMBa performs as we vary the relative size between the index and the filter. As discussed in Section 3, the index size can vary dramatically, being anywhere between 0.1× to 51.2× of the filter size. The relative size depends on the key size,

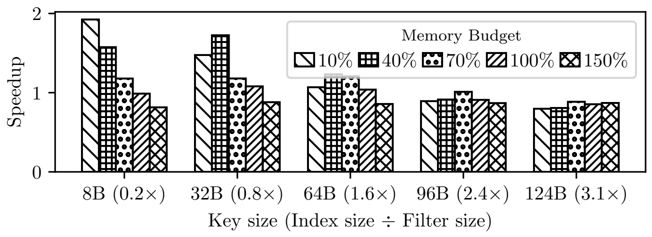


Figure 15: SHaMBa leads to significant performance benefits under memory pressure when the key size is 64B or smaller, and the number of filter blocks is comparable to or larger than the number of index blocks (entry size is 128B).

the entry size, and the bits-per-element of the BF. In this experiment, we use as entry size 128B, and we vary the key size from 8B (where the index is only 0.2× of the total BF size) and 124B (3.1×). The workload consists of all-empty uniform queries ($\alpha = 0$). Figure 15 shows the speedup of SHaMBa-eq for different memory budgets (10%-150%). When the filter size is much larger than the index block, SHaMBa-eq leads to 2× speedup. However, since SHaMBa only targets the filter blocks, its benefits are reduced when the index size grows bigger than the filter.

SHaMBa Also Benefits Faster Storage Devices. Our last experiment shows that SHaMBa leads to performance improvements for faster storage devices. We experiment with a standard SSD, a fast PCIe SSD, and a RAM-disk (that serves as an emulation of NVMe), using uniform all-empty ($\alpha = 0$) read queries. Figure 16 shows that SHaMBa leads to 1.4×-2× performance improvement when compared to the RocksDB baseline. We also note that the benefit is high even in the RAM-disk case, showing that the utility-based block skipping is beneficial even when slow I/Os are not the bottleneck.

6 RELATED WORK

Memory Allocation in LSM-Trees. A key aspect of LSM-Tree tuning is memory allocation. Recent work has proposed new ways to allocate memory across different levels [12], to allocate memory between Bloom filters and the write buffer [13], and also to allocate memory across multiple LSM-Trees [25]. In addition, recent work has focused on the benefit of the block cache for LSM-Trees since all the data pages are immutable and short-lived [44]. Contrary to prior work on LSM memory management, SHaMBa proposes a new variant of Bloom filters that allows for incremental, workload-tailored memory utilization for Bloom filters, and more effective navigation of the memory vs. performance trade-off.

Elastic Bloom Filter. The Elastic Bloom filter (EBF) [48] uses a collection of BFs similar to the MBF we introduce. The segmentation strategy employed by EBF exacerbates the hashing overhead that has been gradually dominating key-value workloads. Hence, the EBF increases the hashing cost as it requires a new hash digest calculation for each filter segment. On the other hand, SHaMBa reduces the overall hashing overhead by carefully re-using hash calculations. Secondly, the EBF always uses all of its BFs, while SHaMBa can skip modules with no importance. Lastly, the EBF adversely impacts compaction, as for every SST file, each segment needs to hash and index all keys of the SST file.

Membership-Test Filters. The various membership-test filters developed recently [3, 5, 8, 11, 19, 32, 36, 40, 41] can be classified into

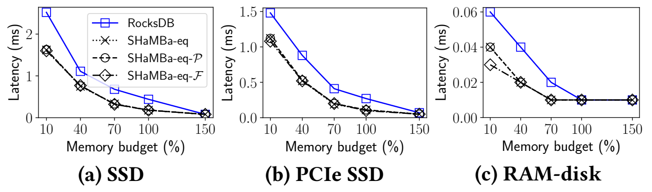


Figure 16: SHaMBa benefits remain as we vary storage devices from an SSD, to a fast PCIe SSD, to a RAM-disk.

BF variants and fingerprint-based variants. Generally, BF variants do not store information about inserted elements while fingerprint-based variants keep track of the fingerprint of elements that can help with in-place updates and deletions, which are not needed for SST files, since they are immutable. Below we compare our approach with the most relevant representative from each category.

Blocked Bloom Filter. A Blocked Bloom filter (BBF) [32], similar to MBF, divides the BFs into multiple blocks. While MBF uses all partitions to insert or query keys, BBF uses only one partition in order to reduce the number of memory probes to random locations generated by multiple hash digests. The partition to use for a specific key is selected by the first hash calculation. To maximize access locality, the size of each partition is a few cache lines. BBFs are ideal for reducing data movement from L3 to L1.

Cuckoo Filter. A Cuckoo filter [19] is a fingerprint-based filter based on Cuckoo hashing. It stores the fingerprint of each element in a bucket, and each bucket stores at most b signatures. Although Cuckoo hashing has a constant expected cost per insertion, even considering the rehashing case, it assumes the use of two (c, k)-universal functions. On the contrary, practical BF implementations use only one expensive hash function, and MBFs use only one hash function for an entire query. Another benefit of using Cuckoo filters is the ability to update in-place (with a small probability of failure) and to delete, however, these features are not necessary for supporting LSM-Trees. Further, Cuckoo filters face the risk of insertion failure, which would require a restart of the SST file creation process (e.g., at compaction time), and a rehash anew with a different hash function, which again, cannot guarantee successful insertion. Overall, the complexity of implementing and deploying Cuckoo filters in LSM-Trees outweigh their benefits.

7 CONCLUSIONS

In this paper, we introduce SHaMBa, a novel LSM-based key-value engine that is specifically designed to address performance loss due to memory pressure. When the available memory is not enough to hold all filters, using them hurts performance. This trend is expected to continue as data size increases. To address this problem, we propose Modular Bloom filters (MBFs), a BF variant that consists of multiple *modules* with the same aggregate size, the same aggregate false positive, and the same maximum number of probes per query. MBFs enable smooth navigation of the memory vs. performance trade-off through their modules that can be queried independently. Overall, using MBFs and a utility-based module skipping strategy, SHaMBa exploits the available memory more efficiently to offer better performance than the state of the art under memory pressure.

Acknowledgements. This work is funded by NSF Grants IIS-2144547 and IIS-1850202, a Facebook Award, and a Meta gift.

REFERENCES

[1] Ildar Absalyamov, Michael J Carey, and Vassilis J Tsotras. 2018. Lightweight Cardinality Estimation in LSM-based Systems. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 841–855.

[2] Wail Y Alkowaileet, Sattam Alsubaiee, and Michael J Carey. 2020. An LSM-based Tuple Compaction Framework for Apache AsterixDB. *Proceedings of the VLDB Endowment* 13, 9 (2020), 1388–1400.

[3] Paulo Sérgio Almeida, Carlos Baquero, Nuno Preguiça, and David Hutchison. 2007. Scalable Bloom Filters. *Inform. Process. Lett.* 101, 6 (mar 2007), 255–261.

[4] Apache. 2021. Cassandra. *http://cassandra.apache.org* (2021).

[5] Michael A. Bender, Martin Farach-Colton, Rob Johnson, Russell Kraner, Bradley C. Kuszmaul, Dzejla Medjedovic, Pablo Montes, Pradeep Shetty, Richard P. Spillane, and Erez Zadok. 2012. Don't Thrash: How to Cache Your Hash on Flash. *Proceedings of the VLDB Endowment* 5, 11 (2012), 1627–1637.

[6] Burton H Bloom. 1970. Space/Time Trade-offs in Hash Coding with Allowable Errors. *Commun. ACM* 13, 7 (1970), 422–426.

[7] Edward Bortnikov, Anastasia Braginsky, Eshcar Hillel, Idit Keidar, and Gali Sheffi. 2018. Accordion: Better Memory Organization for LSM Key-Value Stores. *Proceedings of the VLDB Endowment* 11, 12 (2018), 1863–1875.

[8] Andrei Z. Broder and Michael Mitzenmacher. 2002. Network Applications of Bloom Filters: A Survey. *Internet Mathematics* 1 (2002), 636–646.

[9] Mark Callaghan. 2016. Compaction priority in RocksDB. *http://smalldatum.blogspot.com/2016/02/compaction-priority-in-rocksdb.html* (2016).

[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.

[11] Adina Crainiceanu. 2013. Bloofi: a hierarchical Bloom filter index with applications to distributed data provenance. In *Proceedings of the International Workshop on Cloud Intelligence (CloudI)*. 4:1–4:8.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] Siying Dong, Mark Callaghan, Leonidas Galanis, Dhruba Borthakur, Tony Savor, and Michael Strum. 2017. Optimizing Space Amplification in RocksDB. In *Proceedings of the Biennial Conference on Innovative Data Systems Research (CIDR)*.

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

[19] 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.

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

[21] HBase. 2013. Online reference. *http://hbase.apache.org/* (2013).

[22] 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.

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

[24] Hyesook Lim, Jungwon Lee, and Changhoon Yim. 2015. Complement Bloom Filter for Identifying True Positiveness of a Bloom Filter. *IEEE Communications Letters* 19, 11 (2015), 1905–1908.

[25] 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.

[26] Chen Luo and Michael J Carey. 2019. On Performance Stability in LSM-based Storage Systems. *Proceedings of the VLDB Endowment* 13, 4 (2019), 449–462.

[27] Chen Luo and Michael J. Carey. 2020. LSM-based Storage Techniques: A Survey. *The VLDB Journal* 29, 1 (2020), 393–418.

[28] Siqiang Luo, Subarna Chatterjee, Rafael Ketsetsidis, Niv Dayan, Wilson Qin, and Stratos Idreos. 2020. Rosetta: A Robust Space-Time Optimized Range Filter for Key-Value Stores. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 2071–2086.

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

[30] Ju Hyoung Mun, Jungwon Lee, and Hyesook Lim. 2017. A new Bloom filter structure for identifying true positiveness of a Bloom filter. In *Proceedings of the IEEE International Conference on High Performance Switching and Routing (HPSR)*. 1–5.

[31] 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.

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

[33] 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.

[34] RocksDB. 2020. Universal Compaction. *https://github.com/facebook/rocksdb/wiki/Universal-Compaction* (2020).

[35] RocksDB. 2021. Block Cache. *https://github.com/facebook/rocksdb/wiki/Block-Cache* (2021).

[36] Christian Esteve Rothenberg, Carlos Macapuna, Fabio Verdi, and Mauricio Magalhaes. 2010. The deletable Bloom filter: a new member of the Bloom family. *IEEE Communications Letters* 14, 6 (jun 2010), 557–559.

[37] 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.

[38] 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.

[39] Kulesh Shanmugasundaram, Hervé Brönnimann, and Nasir D Memon. 2004. Payload attribution via hierarchical bloom filters. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*. 31–41.

[40] 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.

[41] Xiujun Wang, Yusheng Ji, Zhe Dang, Xiao Zheng, and Baohua Zhao. 2015. Improved Weighted Bloom Filter and Space Lower Bound Analysis of Algorithms for Approximated Membership Querying. In *Proceedings of the International Conference on Database Systems for Advanced Applications (DASFAA)*. 346–362.

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

[43] Maysam Yabandeh. 2017. Partitioned Index/Filters. *http://rocksdb.org/blog/2017/05/12/partitioned-index-filter.html* (2017).

[44] 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.

[45] MyungKeun Yoon, JinWoo Son, and Seon-Ho Shin. 2014. Bloom tree: A search tree based on Bloom filters for multiple-set membership testing. In *Proceedings of the IEEE Conference on Computer Communications (INFOCOM)*. 1429–1437.

[46] Huanchen Zhang, Hyeontaek Lim, Viktor Leis, David G Andersen, Michael Kaminsky, Kimberly Keeton, and Andrew Pavlo. 2018. SuRF: Practical Range Query Filtering with Fast Succinct Tries. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 323–336.

[47] Teng Zhang, Jianying Wang, Xuntao Cheng, Hao Xu, Nanlong Yu, Gui Huang, Tieying Zhang, Dengcheng He, Feifei Li, Wei Cao, Zhongdong Huang, and Jianling Sun. 2020. FPGA-Accelerated Compactions for LSM-based Key-Value Store. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*. 225–237.

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

[49] Wenshao Zhong, Chen Chen, Xingbo Wu, and Song Jiang. 2021. REMIX: Efficient Range Query for LSM-trees. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*. 51–64.

[50] 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.