Reducing Bloom Filter CPU Overhead in LSM-Trees on Modern Storage Devices

Zichen Zhu, Ju Hyoung Mun, Aneesh Raman, Manos Athanassoulis
zczhu@bu.edu,jmun@bu.edu,aneeshr@bu.edu,mathan@bu.edu
Boston University

ABSTRACT

Bloom filters (BFs) accelerate point lookups in Log-Structured Merge (LSM) trees by reducing unnecessary storage accesses to levels that do not contain the desired key. BFs are particularly beneficial when there is a significant performance difference between querying a BF (hashing and accessing memory) and accessing data (on secondary storage). This gap, however, is decreasing as modern storage devices (SSDs and NVMs) have increasingly lower latency, to the point that the cost of *accessing data* can be comparable to that of filter *probing* and *hashing*, especially for large key sizes that exhibit high hashing cost. In an LSM-tree, BFs are employed when querying each level of the tree, thus, exacerbating the CPU cost as the data size – and thus, the tree height – grows. To address the increasing CPU cost of BFs in LSM-trees, we propose to *re-use hash calculations* aggressively *within* and *across* BFs, as well as *between different levels*, and we show both analytically and experimentally that we can maintain a close-to-ideal false positive rate while significantly reducing the runtime. The reduced CPU cost for queries using the proposed *hash sharing* leads to 10% higher lookup performance in an LSM-tree with 22GB of data (5 levels) stored in a state-of-the-art PCIe SSD. The benefit further increases for faster underlying storage. Specifically, we show that for faster NVM devices, hash sharing leads to performance gains up to 40%.

1 INTRODUCTION

LSM-Trees are Everywhere. Log-Structured Merge-trees (LSM-trees) [29] are the core data structure of several state-of-the-art key-value engines like RocksDB [14] at Facebook, LevelDB [17] and BigTable [7] at Google, HBase [18] and Cassandra [3] at Apache, WiredTiger [44] at MongoDB, X-Engine [19] at Alibaba, and DynamoDB [12] at Amazon. LSM-trees are widely adopted because they offer high ingestion rate and support fast reads. In addition to the systems that are mentioned above, in the past few years, various LSM-tree optimizations on compaction, membership filtering, and memory management have been proposed [1, 2, 5, 8, 10, 11, 20, 24, 25, 27, 28, 36, 41, 45–47].

The Structure of LSM-Trees. LSM-trees maintain sorted runs across multiple levels with exponentially increasing capacity, which have potentially overlapping key ranges. The performance of LSM-trees is determined by many tuning knobs, including compaction policy, size ratio between levels, and metadata used to accelerate read queries. In particular, LSM-trees employ fence pointers and Bloom filters (BFs) [4] to reduce unnecessary storage accesses [26].

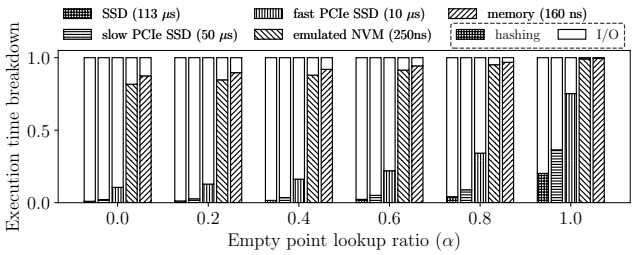


Figure 1: The fraction of the time spent on hashing (vs. probing the BF, accessing data blocks, and other operations) in an LSM-tree increases as the empty point lookup ratio (α) increases, and as the underlying storage becomes faster. Hashing is becoming a key performance bottleneck for LSM-trees on fast storage devices.

Bloom Filters in LSM-Trees. Since key-value pairs are spread across multiple levels, a point query might need to probe every level of a tree, thereby, requiring multiple I/Os for a single lookup. To avoid unnecessary accesses, LSM-trees typically employ BFs [4] to identify whether the target key belongs to a given level. A BF is associated with each level on the secondary storage (or a file that belongs to a given level, in case of partitioned LSM-trees [13]), and is often prefetched in main memory, to be readily available during a point query, before accessing slow storage. The cost of querying a BF is two-fold: (a) hashing and (b) probing of the filter's bits. On the other hand, accessing data on secondary storage, e.g., hard disk drives (HDD) or solid-state drives (SSD), is several orders of magnitude more expensive than probing the filter in memory. This performance gap always renders it worthwhile to consult BFs before accessing data. Overall, BFs reduce the number of disk accesses and the overall query latency at the price of additional memory footprint and CPU computation.

What About Faster Storage? Contrary to common perception, however, BFs are not always beneficial [39]. The rationale behind the ubiquitous use of BFs in LSM-trees is that there is *a considerable cost difference between accessing a BF (in memory) and accessing data (on disk).* As the gap in access latency between BFs and data narrows, the advantages of using BFs weaken. If the data is already cached in main memory, BFs are detrimental. Further, as new storage devices like SSDs and non-volatile memories (NVMs) [35] emerge, the latency gap between memory and storage narrows. Typically, a BF query requires an expensive hash calculation and one or more memory accesses for a total cost in the order of 1μs for 1KB keys.

DAMON'21, June 20–25, 2021, Virtual Event, China

ACM ISBN 978-1-4503-8556-5/21/06…$15.00
https://doi.org/10.1145/3465998.3466002

Meanwhile, a potentially unnecessary disk access is 2ms and 100μs for our in-house HDDs and SSDs, respectively. As a result, for each query within a level, an additional 1μs can help to avoid an I/O, and thus, to significantly reduce the query latency, especially for empty lookups [8]. The benefit from using BFs (and thus, hashing), however, reduces for as the data access cost reduces. Overall, even for workloads with only empty queries, as storage devices become faster, the across-the-board benefit of using BFs is challenged.

Bloom Filters Incur High CPU Overhead. Ideally, a BF that supports a probabilistic membership test requires multiple independent hash functions in order to achieve the minimal error rate. In practice, however, the different BF indexes are often computed using a single hash function, followed by much cheaper bitwise operations (rotations and modulo) to generate the remaining indexes to be probed[1]. Taking into account that current SSD devices have several orders of magnitude lower access latency than disk, and that future SSDs and NVMs are bound to be faster, *hashing latency is on its way to becoming comparable with data access latency*. For example, accessing a 4KB data page on our off-the-shelf SSD needs 113μs, while the cost of hashing a 1KB-key using MurmurHash64 (which is used in production systems [14]) is 235ns, making storage about 480× more expensive than hashing. However, accessing a data page of our PCIe SSD device takes 10μs [31] (7μs when bypassing the file system using Intel's SPDK [42]), reducing this gap to 42× (30× without file system). In addition, future NVM devices are expected to offer access latency as low as ~250ns, being only 1.6× slower than DRAM [32], hence, making storage access *comparable* to hashing. Note that when data is cached in main memory, our experiments show that a single hash function calculation is ~1.47× more expensive than accessing a memory page, thereby, making the use of a BF detrimental. The LSM hashing overhead is further exacerbated as multiple BFs are queried per lookup (at least one per level), and repeated hash calculations turn querying over fast storage (or cached data) into a CPU-intensive operation. Fig. 1 shows the execution time breakdown of point queries in a state-of-the-art LSM-tree with size ratio 10 that has 5 levels, for a total of 22GB of data. We focus on the time spent on hashing and time spent on other parts of the code including waiting for I/O completion. We vary the fraction of empty queries between 0% ($\alpha = 0$) and 100% ($\alpha = 1$), and we compare five different devices as base data storage (an SSD, a PCIe SSD, a fast PCIe SSD, an emulated NVM, and main memory), while BFs are always in memory. For a fast PCIe SSD that has an access latency of 10μs, for all non-empty queries ($\alpha = 0$) about 15% of the query time is spent on hashing. The fraction of time spent hashing increases to more than 75% for empty queries ($\alpha = 1$). For faster devices this will further increase as we observe with the emulated NVM, which reaches almost 100%. While hashing for $\alpha = 1$ helps to avoid unnecessary I/Os, it takes the majority of query execution. As a result, our goal is to maintain its benefits (avoiding unnecessary I/Os), while reducing its cost. In addition, recent designs like ElasticBF [22] and Rosetta [28] employ *a higher number of smaller* BFs. Both approaches introduce four or more additional BFs per SST (Sorted-String Table) file, thus increasing the hashing overhead.

[1]E.g., https://github.com/rockset/rocksdb-cloud/blob/master/util/bloom_impl.h.

Hash Sharing. To reduce the CPU overhead, we propose to aggressively re-use hash computations *within* a BF and *across different BFs residing in different levels*. We reproduce state-of-the-art results for BFs that use only one hash calculation and perform cheaper computations (termed *pseudo-hashing*) for the remaining positions of the BF's bit-vector (a technique already employed by practical implementations of BFs in LSM systems like RocksDB). We take this a step further by (i) showing how to share hash computations across multiple LSM levels, and (ii) across the series of *smaller* BFs that forms a single logical BF, in the case of ElasticBF [22].

The aggregate cost of hashing in state-of-the-art LSM-trees depends on the height of the tree (which, in turn, depends on the data size and the system tuning), because for each point query, a BF per level is typically accessed. The proposed approach of *hash sharing across levels* decouples the aggregate hashing cost from data size, since, regardless of the number of LSM-tree levels, the amount of hashing remains constant. Similarly, hash sharing for ElasticBF allows us to decouple its CPU cost from its design, by internally using pseudo-hashing.

Contributions. Our work offers the following contributions.

- We identify that BFs dominate the LSM query latency for *fast storage* and *high hashing cost*.
- We decouple the amount of hashing from the data size (height of an LSM-tree) by hash-sharing across different levels.
- We decouple the amount of hashing from the design complexity of ElasticBF, an approach that can be used by other BF variants.
- We show through analytical and experimental results that hash sharing improves LSM query performance by 10% on PCIe SSD and 40% on emulated NVM devices.

2 BACKGROUND

LSM-Tree Basics. Many modern key-value stores adopt LSM-trees as their storage layer in order to handle write-intensive workloads, because LSM-trees are designed for fast ingestion [3, 7, 12, 14, 17–19, 29, 44]. To support fast writes, LSM-trees buffer all inserts (including the ones updating or deleting existing entries) in a memory buffer, typically referred to as level 0. When the buffer reaches a predetermined capacity, it is flushed to secondary storage in the form of a sorted *run*, consisting of multiple files stored as immutable SST files. All runs in the secondary storage are organized in a tree-like structure where each level has exponentially larger capacity according to a pre-defined size ratio T. The number of LSM-tree levels L depends on the total data size, the size of the memory buffer, and the size ratio [28]. Shallower levels store more recent updates and have a smaller capacity. Similar to buffer flushing, whenever a level fills up, a sort-merge operation is triggered between sorted runs from this newly-saturated level and the next one, and obsolete entries are removed during this process. This sort-merge operation operation, 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 in different levels have also been proposed [6, 10, 11, 38]. Leveling restricts the number of runs to 1 in each level, while tiering allows the number of runs be as large as $T - 1$.

Point Queries in LSM-Trees. As mentioned above, updating or deleting an entry is essentially a new insertion (for delete, we insert

Operation	Latency	Normalized
4KB I/O on HDD	4.6 ms	28750×
4KB I/O on SDD	113 μs	706×
4KB I/O on PCIe SSD	10 μs	62.5×
4KB I/O on PCIe SSD (using SPDK)	7 μs	43.75×
4KB I/O on emulated NVM	250 ns	1.56×
4KB access on Memory	**160 ns**	**1×**
CITY of 1KB-key	176 ns	1.1×
Murmur Hash 64 (MM64) of 1KB-key	235 ns	1.47×
CRC of 1KB-key	323 ns	2×
XXHash (XX) of 1KB-key	874 ns	5.46×
MD5 of 1KB-key	2.85 μs	17.81×
SHA-256 of 1KB-key	5.17 μs	32.31×

Table 1: The decreasing access latency of new storage devices makes the hashing cost of a 1KB-key comparable with accessing a page in NVM (within one order of magnitude).

a tombstone). As such, multiple entries with the same key may exist in the tree. However, point lookups can terminate safely after finding the first entry with a matching key, because any matching keys in older levels are guaranteed to be obsolete. Therefore, a point query first consults the in-memory buffer and then traverses the tree from the shallowest to the deepest level until it finds the first match. In the case of tiering, which has multiple overlapping runs per level, searching within a level goes from the youngest to the oldest run and terminates if there is a match.

Auxiliary In-Memory Data Structures. To boost query performance, LSM-trees maintain two in-memory auxiliary data structures for each SST file: fence pointers and Bloom filters.

Fence Pointers: Since entries within a disk-resident run are sorted by key, the min-max range of each page does not overlap with any other page. Fence pointers are the min-max ranges for each disk page, along with their aggregation at the level of each SST file and each level. They ensure that at most one I/O occurs when searching for a target key within a single run.

Bloom Filters: Each SST file is also equipped with a BF to avoid unnecessary I/Os. A BF is a membership test data structure that uses an m-bit vector and originally k *independent* hash functions to store and query the membership of n elements [4, 43]. All negative responses to membership queries are always correct, however, positive responses might either be *true positives*, or *false positives* with a small probability which is a function of k, m, and n. The expected false positive rate (f_p) and the optimal number of hash functions to use are shown in Eq. (1).

$$f_p \approx \left(1 - e^{-k \cdot n/m}\right)^k \quad \text{and} \quad k_{opt} = \left\lceil \frac{m}{n} \cdot \ln 2 \right\rceil \tag{1}$$

Overall, the impact of false positives in LSM-trees can be calculated by considering the disk accesses due to false positives across all levels [9]. All LSM-based key-value stores employ BFs [28] or other variations like ElasticBF [22] and Rosetta [28].

Storage Access vs. Hashing. Next, we put into context, the comparison between storage access and hashing latency. Table 1 shows the access latency for a 4KB page in various devices (HDD, SSD, PCIe SSD, NVM, and memory) as well as the hashing latency of a 1KB key using six representative hash functions: MurmurHash64 (MM64), XXHash (XX), MD5, SHA-256, CRC, and CITY64 (CITY). We use the RocksDB [14] implementation of MM64, XX, and CRC, and Google's implementation of CITY [16]. As MD5 [37] and SHA-256 [40] are more than one order of magnitude expensive than other hash functions they are rarely used for practical BF implementations. Overall, we observe that the hashing vs. data access latency gap reduces for newer storage devices (e.g., PCIe SSD), and that even the most efficient hash functions are comparable with the expected access latency of NVM devices.

3 THE CPU COST OF BLOOM FILTERS

We now analyze the point query cost in an LSM-tree focusing on the amount of time spent on hashing for the BFs. We consider both the leveling and the tiering compaction policies.

Leveling. A point query in an LSM-tree can be classified as either *empty* or *non-empty*. The latter will have to do at least one disk access as it targets existing keys [9]. We first analyze the cost of querying a single LSM level. The cost of querying level i, $\mathcal{T}(i)$, is modeled using (a) the fraction of empty queries over all point queries α_i, (b) the BF query cost (CPU cost of hashing and memory cost of probing the BF indexes) T_{BF}, and (c) the data page access cost T_D. $\mathcal{T}(i)$ is the sum of the cost of querying the BF and accessing the data for non-empty queries, $(1-\alpha_i)\cdot(T_{BF}+T_D)$, and of the cost of querying the BF and accessing the data due to false positive for the empty queries, $\alpha_i \cdot f_p \cdot T_D$, where f_p is the false positive rate):

$$\begin{aligned}\mathcal{T}(i) &= (1-\alpha_i)\cdot(T_{BF}+T_D) + \alpha_i \cdot (T_{BF} + f_p \cdot T_D) \\ &= T_{BF} + (1-\alpha_i)\cdot T_D + \alpha_i \cdot f_p \cdot T_D \end{aligned} \tag{2}$$

The BF cost, T_{BF}, consists of two components: (a) the hash calculation T_H, and (b) the BF probing T_P. T_P depends on where the BFs are stored. Since BFs are usually cached in main memory, T_P is often negligible compared to the I/Os. In fact, the hot BFs of an LSM-tree reside higher in the cache hierarchy, making T_P negligible even compared to T_H. For generality, we assume that the BFs are in main memory and that T_P is small, but not negligible. On the other hand, the T_H depends on the CPU power and the key size. Putting everything together, the cost for a read workload $\mathcal{T}(i)$ on level i with α_i fraction of empty queries is shown in Eq. (3).

$$\mathcal{T}(i) = T_H + T_P + (1-\alpha_i)\cdot T_D + \alpha_i \cdot f_p \cdot T_D \tag{3}$$

Using Eq. (3), we can now understand what the main bottleneck is for point lookups. When $\alpha_i << 1$, the time spent to retrieve data dominates the overall LSM-tree lookup cost, because T_D corresponds to expensive accesses on the slow storage. Even when α_i is close to 1, f_p contributes to a number of slow storage accesses, making them the bottleneck for high T_D. However, as novel storage devices have dramatically reduced access latency, even for low α_i, the impact of hashing is pronounced, as T_D and T_H are comparable.

Full LSM-Tree Query Cost. We now synthesize the overall query cost using the cost per level. Note that while we use α_i to denote the empty queries per level; the workload is oblivious to the structure of the tree, so it has an overall fraction of empty queries denoted as α. To compute α_i, we need the total number of queries reaching level i and the number of negative results in that level. Thus, we introduce a new parameter, β_i quantifying the probability that a level i has matching elements for the workload. The overall fraction of non-empty queries is $1-\alpha$. Further, $\sum_{i=1}^{L} \beta_i$ quantifies the queries

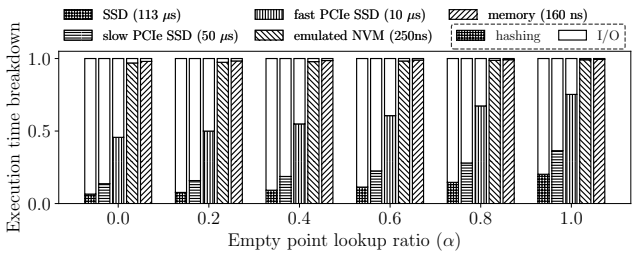


Figure 2: The amount of time spent on hashing for tiered LSM-trees is higher than that of leveled LSM-trees (Fig. 1).

that found an element in any of the L levels, hence, $\sum_{i=1}^{L} \beta_i = 1 - \alpha$. Assuming that the keys in the LSM-tree with size ratio T are uniformly distributed, we have $\beta_i = T^{i-1} \cdot \beta_1$ because every level is T times larger than the previous one. Using the size of the first level relatively to the remaining of the tree we get $\beta_1 = \frac{1-\alpha}{\sum_{j=1}^{L} T^{i-1}}$. Hence, the probability that a query terminates in level i is β_i, and the fraction of queries that reach level i is $1 - \sum_{j=1}^{i-1} \beta_j$, and we can now calculate $\alpha_i = 1 - \frac{\beta_i}{1-\sum_{j=1}^{i-1} \beta_j}$, with $\alpha_1 = 1 - \beta_1$. Overall, the cost of a lookup in the entire LSM-tree is shown in Eq. (4).

$$cost = \mathcal{T}(1) + \sum_{i=2}^{L} \left(1 - \sum_{j=1}^{i-1} \beta_j\right) \cdot \mathcal{T}(i) \tag{4}$$
$$\approx \left(L - \frac{1-\alpha}{T-1}\right) \cdot (T_H + T_P) + (1-\alpha) \cdot T_D + \left(L - \frac{1-\alpha}{T-1} - 1 + \alpha\right) \cdot (f_p \cdot T_D)$$

The interested reader can find in the Appendix the derivation of Eq. (4). Note that the storage access cost due to true positives stays constant, while the BF related costs increase with L. This implies that as we have more data (in faster storage devices), the fraction of the time spent hashing will keep increasing.

Tiering. We now proceed to present the query cost with an LSM-tree that uses tiering. The difference from the more classical leveled LSM-tree is that each level contains T runs which may overlap with one another. Therefore, an empty point lookup has to query all T runs in each level, while a non-empty lookup should search runs from the oldest to the newest until the key is found. Assuming the keys are uniformly distributed across all runs, the cost of querying a level i in tiering, $\mathcal{T}(i)$, is modeled as shown in Eq. (5).

$$\mathcal{T}(i) = (1-\alpha_i) \cdot \frac{T+1}{2} \cdot (T_H + T_P) + (1-\alpha_i) \cdot T_D + \alpha_i \cdot T \cdot \left(T_H + T_P + f_p \cdot T_D\right) \tag{5}$$

The total cost of a lookup with tiering is shown in Eq. (6).

$$cost = \mathcal{T}(1) + \sum_{i=2}^{L} \left(1 - \sum_{j=1}^{i-1} \beta_j\right) \cdot \mathcal{T}(i) \approx \frac{T+1}{2} \cdot (1-\alpha) \cdot (T_H + T_P) + (1-\alpha) \cdot T_D + (T \cdot L - (1-\alpha) \cdot (T+1)) \cdot \left(T_H + T_P + f_p \cdot T_D\right) \tag{6}$$

Fig. 2 shows the fraction of hashing cost for a tiered LSM-tree with 5 levels and 10 runs per level, while varying α. A lookup in tiering needs to search more runs, hence, the hashing cost amounts to a higher fraction when compared to leveling (shown in Fig. 1).

Hash Function	FPR (%)	Hash Function	FPR (%)
MurmurHash (MM)	0.850%	SHA-256	0.868%
MurmurHash64 (MM64)	0.853%	CRC	0.819%
XXHash (XX)	0.794%	CITY	0.850%
MD5	0.921%	All k	0.899%

Table 2: Using a single hash digest and bit-rotation does not negatively impact the experimentally measured FPR when compared with a BF with k different hash functions.

4 SHARING BLOOM FILTER HASHING

We now discuss the benefits of hash sharing (a) in a single BF, (b) across the series of *smaller* BFs that forms a single logical BF, and (c) across multiple BFs residing in different levels of an LSM-tree.

4.1 Hash Sharing in a Single BF

Classical BFs [4] rely on k independent hash functions to generate k indexes, which results in high CPU overhead. Practical BF implementations share a single hash calculation for their k indexes [14]. For example, RocksDB uses a single hash digest and multiple indexes by rotating the hash digest. Specifically, *given a hash function* $h(x)$, $\delta = h(x) << 17 \mid h(x) >> 15$, *and the* $i^{th}(0 \leq i \leq k-1)$ *hash function* $g_i(x)$ *is calculated using* $g_i(x) = h(x) + i \cdot \delta$. Such an optimization reduces T_H by a factor of k, since it computes only a single hash digest and the bit rotation cost is negligible.

To showcase the impact of this optimization on performance and the false positive rate (FPR), we conduct a micro-benchmark on a single BF. We use seven popular hash functions shown in Table 2, and we vary the key size between 8B and 512B. We populate the BF with 10K keys, with 10 bits per key, and thus, the optimal $k = 7$ hash functions. We execute 100K empty point queries and measure both FPR and query performance.

The first experiment measures the impact of hash sharing via bit-rotation on FPR. We fix the key size to 512B and report the measured FPR in Table 2. We compare a BF that uses all seven hash functions to guarantee that we have independent hash digests, indicated with "All k", along with BF implementations that use a single hash function and calculate the remaining indexes with bit-rotations. We observe that the bit-rotation optimization does not affect the FPR, rather, BFs with bit-rotation achieve close-to-ideal FPR. Note that the theoretical optimal FPR for 10 bits per element is $e^{-10 \cdot (ln(2))^2} \approx 0.819\%$. To further validate the efficacy of bit-rotation, we conduct another experiment for which we vary the bits per key between 1 and 13 (requiring accordingly 1 to 9 hash indexes, as shown in Eq. (1)). Fig. 3 shows that for any number of hash functions using one hash function and bit-rotation also leads to an empirical false positive that is close to the theoretical expectation.

Next, we compare the lookup latency with and without bit-rotation as we vary the key size in Fig. 4. We compare the lookup latency of a BF that uses all seven hash functions (black bars), a BF that uses MD5 and bit-rotation (red bars), and a BF that uses MM64 and bit-rotation (blue bars). As expected, the lookup cost increases as the key size grows, and it is dominated by the hashing cost for all seven hash functions. Using only MD5 reduces the cost significantly, but when using the more efficient MM64, the average

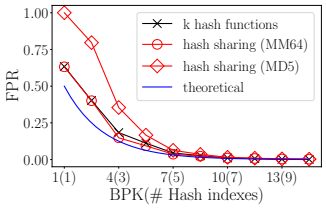


Figure 3: Using a signle hash function and the bit-rotation technique leads to a false positive rate close to the theoretical expectation.

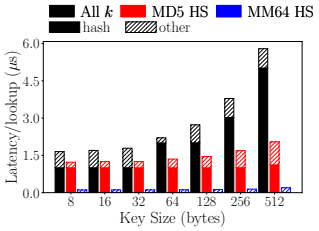


Figure 4: The benefit of hash sharing increases as the key size grows. Less hashing significantly reduces the average query latency.

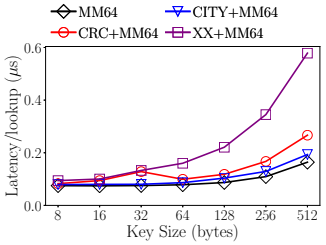


Figure 5: A single hash function with bit rotation significantly outperforms double hashing, especially as the key size increases.

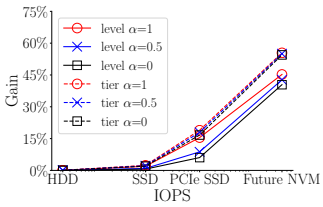


Figure 6: While the gain from hash sharing is negligible for HDDs and modest for SSDs, it rapidly grows for faster devices like NVM.

single hash and bit-rotation

Hash Function	FPR (%)
MM64	0.829%
XX	0.897%
CRC	0.841%
CITY	0.834%

double hashing

Hash Function	FPR (%)
XX + MM64	0.761%
CRC + MM64	0.808%
CITY + MM64	0.842%

Table 3: ElasticBF normally achieves almost the same false positive rate as a normal Bloom filter

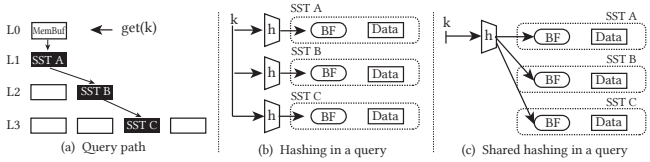


Figure 7: Hash sharing across BFs of different levels.

lookup cost is drastically reduced. For the remainder of the paper, we focus on hash functions that can be computed fast (MM64, XXHash, CRC, and CITY) discarding the slow hash functions (MD5, SHA-256). We use MM64 as our primary hash function because of its efficient execution, low FPR, and wide usage in production, notably by RocksDB.

4.2 Hash Sharing in ElasticBF

In addition to the classical BFs, several BF variants have been proposed for LSM-trees that increase the hashing overhead to address more complex workloads like short-range queries [28, 46], and data skew for point queries [22]. As our target workload is point queries, we focus on ElasticBF [22] which consists of multiple small filter units per BF (essentially *small* BFs) to address access skew. By default, each unit uses a unique hash function to ensure that they are all independent, thus substantially increasing the hashing overhead. As a result, the hashing cost of ElasticBF increases with both the number of *filter units* and the number of *levels* in the LSM-tree.

We first address the increased hashing cost due to the number of units. The bit-rotation optimization is directly applicable to ElasticBF. In addition, we use the double hashing scheme [21] that ensures that each unit will get a provably independent hash function. The double hashing scheme bounds the expected FPR compared to the standard BF by $O(1/n)$ where n is the number of inserted elements. Formally, according to the double hashing scheme, *given two independent hash functions* $h_1(x)$ *and* $h_2(x)$, *the* $i^{th}(0 \leq i \leq k-1)$ *hash function* $g_i(x)$ *is defined as* $g_i(x) = h_1(x) + i \cdot h_2(x)$.

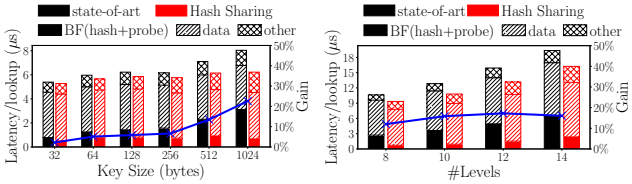
To investigate how much hash sharing can affect the FPR and the CPU overhead, we emulate ElasticBF and conduct a micro-benchmark that compares the bit-rotation and the double-hashing schemes. The benchmark is similar to the previous. We populate the ElasticBF with 10K keys and then issue 100K empty point queries. Note that the bits per key is 10, the number of filter units is 7 and each filter unit uses a single index. For the double hashing scheme, we use MM64 as the primary and XX, CRC, and CITY as secondary hash functions. Table 3 shows that the FPR is similar (with double hashing being marginally better), while Fig. 5 shows that bit-rotation leads to lower query latency, up to 3×.

4.3 Hash Sharing Across Multiple LSM Levels

We now apply hash sharing across multiple LSM levels, a design that can benefit any BF variant employed in an LSM-tree. The key observation is that for a specific query, the same hash digest calculation is repeated across levels. The BFs are different across levels (they have indexed different elements). However, calculating the hash digest is repeated for every queried level until finding the matching key or the tree is entirely searched. Thus, to mitigate this overhead, we *share the hash digest calculation* across levels by re-engineering the BF implementation and allowing the BFs residing in different levels to work in concert during the course of a single query (Fig. 7). As a result, the hashing cost stays constant regardless of the number of levels, shaving off a factor of L from the hashing cost in Eq. (4). The new cost is shown in Eq. (7). Note that the hash sharing technique can also be used across separate runs in tiering, leading to no increase in hashing for tiering compared to leveling even though we are probing $T\times$ more runs.

$$cost^{sh} = T_H + \left(L - \frac{1-\alpha}{T-1}\right) \cdot T_P + (1-\alpha) \cdot T_D + \left(L - \frac{1-\alpha}{T-1} - 1 + \alpha\right) \cdot (f_p \cdot T_D) \quad (7)$$

Performance Implications and Discussion. Hash sharing decouples the amount of time spent on hashing from the number of LSM levels, and as a result, from the data size. In our experiments, we have seen that there is no difference in the empirical FPR across different levels of the LSM-tree between the state-of-the-art design and hash sharing, while the hashing cost of an empty query drops



(a) Varying key size **(b) Varying levels**

Figure 8: Hash sharing reduces hashing overhead. The reduction is more pronounced for larger keys and taller trees.

by a factor of L. We further define the "Gain" from hash sharing to quantify the performance improvement as shown in Eq. (8).

$$\text{Gain} = \frac{cost - cost^{sh}}{cost} \times 100\% \tag{8}$$
$$= \frac{\left(L - 1 - \frac{1-\alpha}{T-1}\right) \cdot T_H}{\left(L - \frac{1-\alpha}{T-1}\right) \cdot (T_H + T_P) + (1-\alpha) \cdot T_D + \left(L - \frac{1-\alpha}{T-1} - 1 + \alpha\right) \cdot f_p \cdot T_D} \times 100\%$$
$$= \frac{L - 1 - \frac{1-\alpha}{T-1}}{\left(L - \frac{1-\alpha}{T-1}\right) \cdot \left(1 + \frac{T_P}{T_H}\right) + (1-\alpha) \cdot \frac{T_D}{T_H} + \left(L - \frac{1-\alpha}{T-1} - 1 + \alpha\right) \cdot f_p \cdot \frac{T_D}{T_H}} \times 100\%$$

Eq. (8) shows that the performance gain is affected by many factors including T_H, T_D, α and L. The gain for tiering can similarly be derived from Eq. (6). To quantify the theoretical gain, we use the latency numbers from Table 1 and plot the expected performance gain from hash sharing as we are moving to faster storage devices for storing our data. Fig. 6 shows the theoretical gain for both leveled and tiered LSM-trees for different fractions of empty vs. non-empty queries. We observe that while the gain is negligible for HDDs and small for SSDs, it is expected to grow rapidly to more than 40%, as we are moving towards faster storage devices. Further, the gain is higher for tiered LSM-trees.

Note that the same benefit applies when ElasticBFs are used in an LSM-tree. In addition to that, hash sharing allows ElasticBFs to benefit from a further reduction in hashing overhead by a factor equal to the number of filter units, without harming the empirically measured FPR. Finally, *any BF variant* that is employed in a hierarchical manner, like in an LSM-tree, can benefit from hash sharing as long as the same hash digest calculation offers the desires results. Hence, filters like Rosetta [28], Cuckoo filters [15], and Counting filters [30] can also benefit. In the following section, we experimentally show the benefits of hash sharing in an LSM-tree that employs BFs with bit-rotation.

5 EXPERIMENTAL EVALUATION

We now present the benefits of *hash sharing* across BFs in different levels in LSM-trees. In our experimentation, we vary the key size, the height of the LSM-tree, the bits per key allocated in the BFs, and the workload characteristics.

Hardware Environment. 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 (40 threads with virtualization). The size of the main memory and the L3 cache in our machine is 384GB and 27.5MB. The server is equipped with two 7200RPM hard drives, one off-the-shelf SSD (240GB S4610), and two state-of-the-art PCIe SSD devices, 1TB PCIe P4510 SSD and Optane 375GB P4800X SSD, which can offer 600K and 1M IOPS accordingly, and 10μs access latency for 4KB page accesses. Unless otherwise specified, we use the 1TB PCIe P4510 SSD with direct I/O enabled in our experimentation.

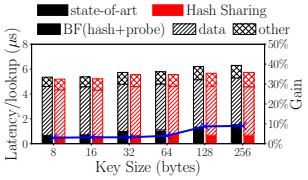


Figure 9: The benefit diminishes for smaller key sizes.

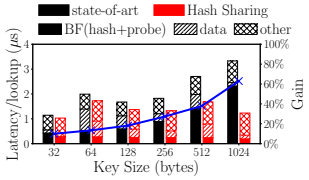


Figure 10: The gain is higher for a Zipfian workload.

Experimental Platform, Workloads, and Metrics. We build an in-house LSM-tree prototype system[2] based on the architecture of RocksDB [14], which uses RocksDB's fast local Bloom filter (format_version = 5, only supports 64-bit hash digest). We stress-test our system by varying workloads and the execution environment. Since our design optimizes read performance, we focus on read-only workloads. We bulk load our LSM-tree with 22GB of key-value pairs. The default size of a key-value entry is 1KB with 512B for the key. In our experimentation, we vary the key size and the height of the tree. With respect to tuning the LSM-tree, we set the file size to 2MB, the size ratio of the LSM-tree to 10, and the bits per key for the BFs to 10. For each experiment, we measure 1 million point queries and report the average lookup latency along with a detailed breakdown of the time spent on hashing, accessing data, or other parts of the code. Each reported measurement is the average of five executions, with negligible standard deviation.

Hash Sharing Scales Better with Key Size. In order to highlight the impact of key size on hashing overhead, we conduct an experiment varying the key size from 8B to 1024B. For this experiment, we increase the key-value entry size to 2KB (in order to accommodate key sizes up to 1KB), with the resulting tree having five levels. Fig. 8a shows the lookup latency (y-axis) of empty queries for variable key sizes (x-axis). Here, we compare the state-of-the-art with a system that employs hash sharing. As expected, the hashing cost increases for both approaches as the key size grows, however, hash sharing has a performance gain of up to 23% (blue line). The time breakdown shows where this benefit is coming from. The time spent in BFs (both hashing and probing) is drastically reduced for the hash sharing approach, while the cost for accessing data, as well as the other costs (e.g., binary search in fence pointers), virtually remains the same. In addition, larger keys have higher hashing overhead, hence, hash sharing is more beneficial for larger key sizes. This observation is inline with Eq. (8); as T_H increases, both T_D/T_H and T_P/T_H will decrease. For 1KB keys, the total hashing cost is reduced to less than half. Fig. 9 shows the benefits from hash sharing for smaller keys. In this experiment we set the entry size equal to 512B and each SST file can now hold more keys. We observe that the benefit diminishes since hashing small keys only constitutes a negligible portion of the whole query.

Hash Sharing Benefit is Higher for Skewed Queries. We further evaluate the effect of a skewed query workload that follows Zipfian. As shown in Fig. 10, the gain steeply increases for 1KB keys to more than 60%. Note that the data access time becomes unstable since a Zipfian workload can be "stuck" requesting keys

[2] Our codebase can be found at https://github.com/BU-DiSC/BF-Shared-Hashing.

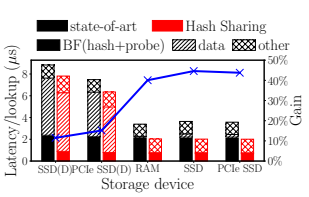


Figure 11: For faster storage, the benefit of hash sharing is pronounced.

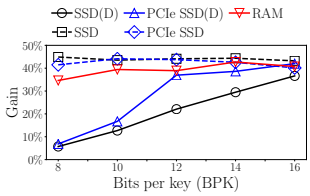


Figure 12: For fewer data accesses (lower FPR), the benefit of hash sharing is larger.

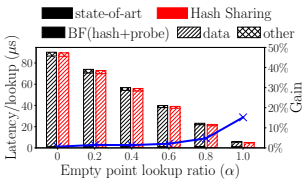


Figure 13: Hash sharing benefits on SSD increase up to 20%, as the fraction of empty queries (α) increases.

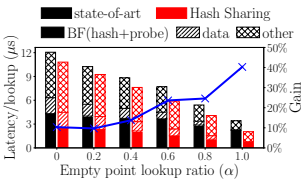


Figure 14: The gain on a RAM disk increases up to 40%, following a similar pattern as in the case of the SSD.

that cause false positives. Overall, the observed false positive was lower than for the uniform workload (0.08% instead of 0.8%), hence, the workload has fewer data accesses due to false positives. As a result, hashing is the main bottleneck for skewed lookups, and hash sharing helps to aggressively reduce it.

Hash Sharing Scales Better with Data Size. We now vary the data size, resulting in LSM-trees that have between 8 and 14 levels (with size ratio 2). As shown in Fig. 8b, the difference in the cost of hashing (solid bars) increases for larger data size (more levels), showing that hash sharing scales with data size. The state-of-the-art system RocksDB performs one hash calculation per level, thus the hashing cost accumulates with the number of levels. On the contrary, using hash sharing, every query performs only one hash calculation, thereby decoupling the hashing cost from the height of the tree. In hash sharing, the increase in the total BF cost is a result of the unavoidable (yet much cheaper than hashing) filter probing. We further observe a small improvement in terms of gain, which is consistent with our gain definition. To explain this, we reformulate Eq. (8) as Eq. (9).

$$\text{Gain} = \frac{T_H}{(T_H + T_P + f_p \cdot T_D) + \frac{T_H+T_P+(1-\alpha)\cdot T_D+\alpha\cdot f_p\cdot T_D}{L-1-\frac{1-\alpha}{T-1}}} \times 100\% \quad (9)$$

According to Eq. (9), as the number of levels increases, the denominator is dominated by the constant $T_H + f_p \cdot T_D$. Even though the performance gain increases for higher number of levels, it eventually approaches a constant value.

Hash Sharing has Higher Impact for Faster Devices. For faster storage devices, the fraction of time spent on hashing increases to the point that it dominates the point query latency. In this experiment, we vary the underlying storage device (SSD, PCIe SSD, and RAM-disk). We use RAM-disk to emulate the behavior of a future non-volatile memory with performance close to DRAM. We also experiment with our SSD and PCIe SSD with direct I/O both enabled and disabled. In Fig. 11, SSD(D) and PCI SSD(D) indicate that direct I/O is enabled, while for SSD and PCI SSD it is disabled. We observe that as the storage latency reduces, hashing dominates query time, and the benefit of hash sharing increases from 10% for the off-the-shelf SSD to more than 40% for an emulated NVM. Note that this is consistent with Eq. (8). Specifically, the smaller the value of T_D/T_H, the higher the gain, and for fixed T_H, the gain monotonically increases for faster storage devices (smaller T_D).

Hash Sharing has Higher Impact for Lower FPR. Using more bits per key leads to lower false positive rate, thus, it reduces the number of data accesses due to false positives, further highlighting the benefit of hash sharing. In Fig. 12, we vary the bits per key from 8 to 14, and we report the performance gain for different storage devices. We observe that the benefit of hash sharing increases for more bits per key (lower false positive rate). The observation is consistent with Eq. (8), in which f_p is in the denominator with a positive coefficient and thus, lower f_p will result in a higher performance gain. When the bits per key is set to 10 (a typical setting in state-of-the-art LSM-trees), the performance is improved by more than 10% on PCIe SSD and by more than 40% on an emulated NVM.

Hash Sharing has Higher Impact for Empty Queries. The previous experiments focus on empty point queries, for which hash sharing is most effective. We now examine the benefit of hash sharing as we vary the fraction of empty queries, α. Note that empty queries will (mostly) avoid doing any I/Os because of BFs, while positive queries will have to read data at least once. Fig. 13 and Fig. 14 show the latency breakdowns of the state-of-the-art and hash sharing when using an SSD and a RAM disk as data storage accordingly. We observe that the fraction of time spent on hashing reduces as the fraction of empty queries (α) increases. The elapsed time spent hashing is constant, however, the time spent retrieving data increases as we have more empty queries. The benefit when using an SSD for data storage is negligible for $\alpha = 0$ and increases to 17% for $\alpha = 1$. In the case of a RAM disk (used to emulate NVM), we observe that the gain starts from 10% for $\alpha = 0$, and grows to more than 40% for $\alpha = 1$. These findings are consistent with Eq. (9). We note the relationship between the term $\frac{T_H+T_P+(1-\alpha)\cdot T_D+\alpha\cdot f_p\cdot T_D}{T-1-\frac{1-\alpha}{T-1}}$ and α. As α increases, the numerator decreases and the denominator increases, hence, the term overall is monotonically decreasing. This means that for more empty queries (higher α) the performance gain is expected to increase, which is corroborated by our experiments.

6 RELATED WORK

The textbook implementation of a BF requires k independent hash functions, however, their cost is prohibitively high. To mitigate this cost, Less Hashing Bloom Filters (LHBF) [21] and One-Hashing Bloom filters (OHBF) [23] aim to achieve the same false positive rate with reduced hashing cost. LHBF divides the filter into k partitions of identical size, and calculates the index for a partition i using two hash functions $h_1(x)$ and $h_2(x)$, $g_i(x) = h_1(x) + i \cdot h_2(x)$. Similarly, OHBF divides the filter into k partitions of uneven sizes and calculates the index for partition i using a single hash function, $g_i(x) = h(x)\%m_i$. The OHBF is implemented using one hash function and a few modulo operations.

Orthogonal to the hashing cost, there have been efforts to reduce the probing cost focusing on the locality of bit-vector accesses [33, 34]. Blocked Bloom filters (BBF) [33] split the filter into a sequence of blocks to reduce memory probing for different locations

generated by the k hash functions. BBFs partitions have a small fixed size of one (or a few) cache lines, and unlike classical BFs, the first hash calculation points to a specific block, and all subsequent probes are performed in the same cache line (or group of cache lines). Bloom-1 [34] filter maps k bits in a single word, instead of mapping to an entire filter, in order to reduce the probing cost. Thus, Bloom-1 can achieve membership identification with only one memory access. While BBFs and Bloom-1 are a great match for in-memory workloads, their locality does not benefit disk-resident workloads where the benefit from being cache-efficient is masked by the latency to retrieve data from the disk.

The aforementioned approaches aim to optimize a single BF, while our work aims to optimize a collection of multiple BFs, by sharing hashing not only *within a BF*, but also *across different BFs*. Hence, our design can be combined with any technique that reduces the hashing cost of a single BF.

7 CONCLUSIONS

In this paper, we observe that as we move to faster storage devices, hashing for BFs in LSM-trees becomes a key performance bottleneck. We address this by decoupling the hashing overhead from the number of distinct levels in the tree (and, as a result, the data size), by sharing a single hash digest across different levels. Our technique reduces the fraction of time spent on hashing during lookups and leads to performance benefits varying from 10% for our PCIe SSD to more than 40% for an emulated NVM.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their valuable feedback. We are particularly grateful to Subhadeep Sarkar for his feedback. This work was supported by the National Research Foundation of Korea (NRF-2019R1A6A3A12032099), NSF Grant No. IIS-1850202, a Facebook Faculty Research Award, and RedHat.

A COST MODEL DERIVATIONS

In this section we provide more details about the derivations of the cost model in Section 3. The total query cost from Eq. (4) can be expanded to Eq. (10) using Eq. (3).

$$\begin{aligned} cost = {} & T_H + T_P + (1-\alpha_1)\cdot T_D + \alpha_1 \cdot f_p \cdot T_D \\ & + \sum_{i=2}^{L}\left(1 - \sum_{j=1}^{i-1}\beta_j\right)\cdot\left[T_H + T_P + (1-\alpha_i)\cdot T_D + \alpha_i \cdot f_p \cdot T_D\right] \end{aligned} \tag{10}$$

Since $\alpha_i = 1 - \frac{\beta_i}{1-\sum_{k=1}^{i-1}\beta_k}$ and $\alpha_1 = 1-\beta_1$, Eq. (10) can be reformulated as follows, leading to Eq. (11).

$$\begin{aligned} cost = {} & T_H + T_P + \beta_1 \cdot T_D + (1-\beta_1)\cdot f_p \cdot T_D \\ & + \sum_{i=2}^{L}\left(1 - \sum_{j=1}^{i-1}\beta_j\right)\cdot\left[T_H + T_P + \frac{\beta_i \cdot T_D}{1-\sum_{k=1}^{i-1}\beta_k} \right. \\ & \left. + \left(1 - \frac{\beta_i}{1-\sum_{k=1}^{i-1}\beta_k}\right)\cdot f_p \cdot T_D\right] \Rightarrow \end{aligned}$$

$$\begin{aligned} cost = {} & \left(L - \sum_{i=2}^{L}\sum_{j=1}^{i-1}\beta_j\right)\cdot(T_H + T_P) + \sum_{i=1}^{L}\beta_i \cdot T_D \\ & + \left(L - \sum_{i=2}^{L}\sum_{j=1}^{i-1}\beta_j - \sum_{i=1}^{L}\beta_i\right)\cdot f_p \cdot T_D \end{aligned} \tag{11}$$

Since $\sum_{i=1}^{L}\beta_i = 1-\alpha$, Eq. (11) becomes Eq. (12).

$$\begin{aligned} cost = {} & \left(L - \sum_{i=2}^{L}\sum_{j=1}^{i-1}\beta_j\right)\cdot(T_H + T_P) + (1-\alpha)\cdot T_D \\ & + \left(L - \sum_{i=2}^{L}\sum_{j=1}^{i-1}\beta_j - 1 + \alpha\right)\cdot f_p \cdot T_D \end{aligned} \tag{12}$$

If we assume that keys in the LSM-tree are perfectly uniform, then, β_i depends on α and the size of level i, i.e., $\beta_i = T^{i-1}\cdot\beta_1$, while $\beta_1 = \frac{1-\alpha}{\sum_{j=1}^{L}T^{i-1}}$. Thus, $\sum_{i=2}^{L}\sum_{j=1}^{i-1}\beta_j$ can be approximated as follows.

$$\begin{aligned} \sum_{i=2}^{L}\sum_{j=1}^{i-1}\beta_j &= \sum_{i=2}^{L}\sum_{j=1}^{i-1}T^{j-1}\cdot\beta_1 \\ &= \beta_1 \cdot \sum_{i=2}^{L}\frac{T^{i-1}-1}{T-1} \\ &= \frac{\beta_1}{T-1}\cdot\left(\sum_{i=2}^{L}\left(T^{i-1}\right) - (L-1)\right) \\ &= \frac{\beta_1}{T-1}\cdot\left(\sum_{i=1}^{L}\left(T^{i-1}\right) - 1 - (L-1)\right) \\ &= \frac{\beta_1}{T-1}\cdot\left(\sum_{i=1}^{L}\left(T^{i-1}\right) - L\right) \\ &= \frac{1-\alpha}{\sum_{k=1}^{L}T^{k-1}}\cdot\frac{1}{T-1}\cdot\left(\sum_{i=1}^{L}\left(T^{i-1}\right) - L\right) \\ &= \frac{1-\alpha}{T-1}\cdot\left(1 - \frac{L}{\sum_{k=1}^{L}T^{k-1}}\right) \\ &= \frac{1-\alpha}{T-1}\cdot\left(1 - \frac{L}{\frac{T^L-1}{T-1}}\right) \\ &= \frac{1-\alpha}{T-1}\cdot\left(1 - \frac{L\cdot(T-1)}{T^L-1}\right) \approx \frac{1-\alpha}{T-1} \\ &\qquad \text{(for } T>3 \text{ or } L>3\text{)} \end{aligned} \tag{13}$$

Therefore, the cost of full LSM-tree lookup can be simplified as in Eq. (4). The analysis discussed above concerns the query cost for a leveled LSM-Tree. When considering tiering, every level contains T runs which may overlap with each other. As such, a point lookup has to query all the runs from the oldest to the newest unless a key is found. Assuming the keys are uniformly distributed across all runs, the number of BF queries for empty lookup is T, while a non-empty lookup needs to lookup on average $\frac{T+1}{2}$ (the average of 1 and T). Thus, the cost of querying level i in tiering, $\mathcal{T}(i)$, is modeled as Eq. (5), and the cost of a lookup in tiered LSM-tree is shown in Eq. (14).

$$
\begin{aligned}
cost &= \mathcal{T}(1) + \sum_{i=2}^{L}\left(1 - \sum_{j=1}^{i-1}\beta_j\right)\cdot\mathcal{T}(i) \\
&= (1-\alpha_1)\cdot\frac{T+1}{2}(T_H + T_P) \\
&+ (1-\alpha_1)\cdot T_D + \alpha_1\cdot T\cdot(T_H + T_P + f_p\cdot T_D) \\
&+ \sum_{i=2}^{L}\left(1 - \sum_{j=1}^{i-1}\beta_j\right)\cdot\left((1-\alpha_i)\cdot\frac{T+1}{2}(T_H + T_P)\right) \\
&+ \sum_{i=2}^{L}\left(1 - \sum_{j=1}^{i-1}\beta_j\right)\cdot((1-\alpha_i)\cdot T_D) \\
&+ \sum_{i=2}^{L}\left(1 - \sum_{j=1}^{i-1}\beta_j\right)\cdot\left(\alpha_i\cdot T\cdot(T_H + T_P + f_p\cdot T_D)\right)
\end{aligned}
\tag{14}
$$

Since $\alpha_i = 1 - \frac{\beta_i}{1-\sum_{j=1}^{i-1}\beta_j}$ and $\alpha_1 = 1-\beta_1$, Eq. (14) becomes Eq. (15).

$$
\begin{aligned}
cost &= \beta_1\cdot\frac{T+1}{2}\cdot(T_H + T_P) \\
&+ \beta_1\cdot T_D + (1-\beta_1)\cdot T\cdot(T_H + T_P + f_p\cdot T_D) \\
&+ \sum_{i=2}^{L}\left(1 - \sum_{j=1}^{i-1}\beta_j\right)\cdot\left(\frac{\beta_i}{1-\sum_{j=1}^{i-1}\beta_j}\cdot(T_H + T_P)\right) \\
&+ \sum_{i=2}^{L}\left(1 - \sum_{j=1}^{i-1}\beta_j\right)\cdot\left(\frac{\beta_i}{1-\sum_{j=1}^{i-1}\beta_j}\cdot T_D\right) \\
&+ \sum_{i=2}^{L}\left(1 - \sum_{j=1}^{i-1}\beta_j\right) \\
&\qquad\cdot\left(\left(1 - \frac{\beta_i}{1-\sum_{j=1}^{i-1}\beta_j}\right)\cdot T\cdot(T_H + T_P + f_p\cdot T_D)\right) \\
&= \sum_{i=1}^{L}\left(\beta_i\cdot\frac{T+1}{2}\cdot(T_H + T_P)\right) + \sum_{i=1}^{L}(\beta_i\cdot T_D) \\
&+ \left(L - \sum_{i=2}^{L}\sum_{j=1}^{i-1}\beta_j - \sum_{i=1}^{L}\beta_i\right)\cdot T\cdot\left(T_H + T_P + f_p\cdot T_D\right)
\end{aligned}
\tag{15}
$$

Using Eq. (13) and $\sum_{i=1}^{L}\beta_i = 1-\alpha$, Eq. (15) is simplified to Eq. (6).

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

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

[5] 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.

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

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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)*.

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

[15] 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.

[16] Google. 2021. CityHash. *https://github.com/google/cityhash* (2021).

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

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

[19] 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.

[20] 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.

[21] Adam Kirsch and Michael Mitzenmacher. 2008. Less hashing, same performance: Building a better Bloom filter. *Random Structures & Algorithms* 33, 2 (2008), 187–218.

[22] 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.

[23] Jianyuan Lu, Tong Yang, Yi Wang, Huichen Dai, Xi Chen, Linxiao Jin, Haoyu Song, and Bin Liu. 2018. Low Computational Cost Bloom Filters. *IEEE/ACM Trans. Netw.* 26, 5 (2018), 2254–2267.

[24] 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.

[25] 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.

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

[27] Chen Luo, Pinar Tözün, Yuanyuan Tian, Ronald Barber, Vijayshankar Raman, and Richard Sidle. 2019. Umzi: Unified Multi-Zone Indexing for Large-Scale HTAP. In *Proceedings of the International Conference on Extending Database Technology (EDBT)*. 1–12.

[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] 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.

[30] Prashant Pandey, Michael A. Bender, Rob Johnson, and Robert Patro. 2017. A General-Purpose Counting Filter: Making Every Bit Count. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 775–787.

[31] Tarikul Islam Papon and Manos Athanassoulis. 2021. A Parametric I/O Model for Modern Storage Devices. In *Proceedings of the International Workshop on Data Management on New Hardware (DAMON)*.

[32] Georgios Psaropoulos, Ismail Oukid, Thomas Legler, Norman May, and Anastasia Ailamaki. 2019. Bridging the Latency Gap between NVM and DRAM for Latency-bound Operations. In *Proceedings of the International Workshop on Data Management on New Hardware (DAMON)*. 13:1–13:8.

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

[34] Yan Qiao, Tao Li, and Shigang Chen. 2011. One memory access bloom filters and their generalization. In *INFOCOM 2011. 30th IEEE International Conference on Computer Communications, Joint Conference of the IEEE Computer and Communications Societies, 10-15 April 2011, Shanghai, China*. 1745–1753.

[35] Simone Raoux, Geoffrey W Burr, Matthew J Breitwisch, Charles T Rettner, Yi-Chou Chen, Robert M Shelby, Martin Salinga, Daniel Krebs, Shih-Hung Chen, Hsiang-Lan Lung, and Chung Hon Lam. 2008. Phase-change random access memory: A scalable technology. *IBM Journal of Research and Development* 52, 4-5 (2008), 465–480.

[36] 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.

[37] Ronald L Rivest. 1992. The MD5 Message-Digest Algorithm. *RFC* 1321 (1992), 1–21.

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

[39] Ori Rottenstreich and Isaac Keslassy. 2015. The Bloom Paradox: When Not to Use a Bloom Filter. *IEEE/ACM Trans. Netw.* 23, 3 (2015), 703–716.

[40] Somitra Kumar Sanadhya and Palash Sarkar. 2008. New collision attacks against up to 24-step SHA-2. In *International conference on cryptology in India*. Springer, 91–103.

[41] 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.

[42] Jonathan Stern. 2016. Introduction to the Storage Performance Development Kit (SPDK). *https://software.intel.com/content/www/us/en/develop/articles/introduction-to-the-storage-performance-development-kit-spdk.html* (2016).

[43] 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.

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

[45] 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.

[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 *18th USENIX Conference on File and Storage Technologies, FAST 2020, Santa Clara, CA, USA, February 24-27, 2020*. 225–237.