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 probing a BF (hashing and accessing memory) and accessing data (on secondary storage). However, this gap 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 of the levels of the tree, thus, exacerbating the CPU cost as the data size grows (and, thus, the tree height). 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 close-to-ideal false positive rate while significantly reducing the runtime. The reduced CPU cost of queries using the proposed *hash sharing* leads to 20% 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 when NVM devices will be available the improvement can increase up to 65%.

1 INTRODUCTION

LSM-trees are Everywhere. Log-Structured Merge-trees (LSM-trees) [28] are the core data structure of several state-of-the-art key-value engines like RocksDB [13] at Facebook, LevelDB [15] and BigTable [6] at Google, HBase [17] and Cassandra [3] at Apache, WiredTiger [39] at MongoDB, X-Engine [18] at Alibaba and DynamoDB [11] at Amazon. LSM-trees are widely adopted because they offer high ingestion rate and support fast reads. In addition to the systems developed in industry 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, 7, 9, 10, 19, 23, 24, 26, 27, 34, 36, 40, 41, 43, 44].

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) to reduce unnecessary storage accesses [25].

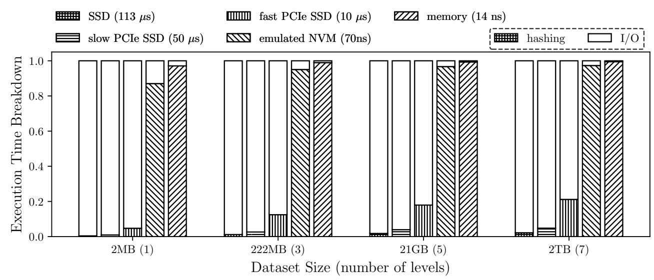


Figure 1: The percentage of time spent in hashing to probe BF in an LSM-tree increases as either the data size increases and/or as the underlying storage becomes faster. Thus, hashing becomes the main performance bottleneck for LSM-trees that hold large datasets on fast storage devices.

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 [12]), and is often pre-fetched 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) the hash calculation and (b) the probing of the filter's bits. Often, the BFs fit in main memory, and thus, the probing cost is negligible. 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. BFs reduce the number of data accesses and the overall query latency at the price of additional memory footprint and hashing.

What About Faster Storage? Contrary to common perception, however, BFs are not always beneficial [35]. 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 of access latency between BFs and data narrows, the advantage of using a BF weakens. If the data is already cached in main memory, BFs are detrimental. Further, as new storage devices like SSDs and non-volatile memories (NVMs) [33] emerge, the latency gap between memory and storage narrows. Typically, a BF probe requires an expensive hash calculation and one or more

Under Submission, 2021

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM.
https://doi.org/10.1145/1122445.1122456

memory accesses for a total cost on the order of 1μs for 1KB keys. Meanwhile, a potentially unnecessary disk access is on the order of 2ms for HDD and 100μs for SSDs. As a result, for each query within a level, an additional 1μs can help to avoid an I/O and, thus, significantly reduces the query latency, especially for zero-result point lookups. However, faster PCIe SSD devices offer access latency as low as 10μs access latency or even lower (e.g., we measured 7μs when using Intel's SPDK [37] to bypass the file system on our PCIe SSD device), making the access latency comparable to the hash calculation cost. In addition, future NVM devices are expected to have much faster latency, being only 5× slower than DRAM, in the order of 70ns [30]. Overall, as storage devices become faster, they challenge the across-the-board benefit of using BFs.

Bloom Filters Incur High CPU Overhead. Ideally, a BF requires multiple independent hash functions in order to achieve the theoretical false positive rate (FPR). In practice, however, the different BF indexes are often computed using a single hash function to calculate a hash digest, followed by much cheaper bitwise operations (rotations and modulo) to generate the remaining index probes[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 become 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 [13], is 235ns, making storage about 480× more expensive than hashing. However, accessing a data page of our PCIe SSD device takes 10μs (7μs when bypassing the file system), reducing this gap to 42× (30× without file system). In addition, future NVM devices are expected to offer access latency in the order of 70*ns*, making accessing storage 3× *faster* than hashing. Note that when data is cached in main memory, a single hash function calculation is about 17× more expensive than accessing a memory page making the use of the BF detrimental. The LSM hashing overhead is further exacerbated as multiple BFs are queried per lookup (one per level), and repeated hash calculations turn querying over fast storage (or cached data) into a CPU-intensive operation. Figure 1 shows the execution time breakdown of point queries (half of which are empty queries) in a state-of-the-art LSM-tree. We focus on the time spent hashing and time spent waiting for I/O completion. We compare different devices for storing the data (an SSD, a PCIe SSD, a fast PCIe SSD, an emulated NVM, and main memory) and different dataset sizes, ranging from 22MB (2 level) to 2TB (7 levels), with size ratio 10. For a fast PCIe SSD that has access latency of 10μs, about 80% of the lookup time is consumed by hashing for 2GB of data. As the access latency of NVM is expected to be reduced, hashing overhead will inevitably increase. The fraction of time spent on hashing accumulates over all levels, and eventually becomes the main bottleneck. Further, recent designs deliver higher and more flexible accuracy for point queries [21] and short range queries [27], using *more smaller* BFs. Both approaches introduce four or more additional BFs per SST file, thus increasing the hashing overhead.

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 bitvector. We take this a step further by (i) showing how to share hash computations across multiple LSM levels, and (ii) across different independent modules of a single logical BF, in the case ElasticBF [21].

The aggregate cost of hashing in state-of-the-art LSM-Trees depends on the height of the tree, which depends on the data size, since for each point query a BF per level is typically accessed. However, *hash sharing across levels* decouples the aggregated 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, and internally use pseudo-hashing.

Contributions. Our work offers the following contributions.

- We identify that BFs dominate LSM query latency for *fast storage* and *high hashing cost*.
- We decouple the amount of hashing from the data size (height of a LSM-tree) by hash sharing across different levels.
- We decouple the amount of hashing from the design complexity of Elastic BF, 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 20% on PCIe SSD and 65% 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, 6, 11, 13, 15, 17, 18, 28, 39]. 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 Sorted-String Tables (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 user-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 [27]. Shallower levels store more recent updates and have smaller capacity. Similar to buffer flushing, whenever a level fills up, a sort-merge operation is triggered between this newly-saturated level and the next one, and obsolete entries are removed during this process.

Point Queries in LSM-trees. As LSM-updates are out-of-place, multiple entries with the same key may exist. However, searching can terminate safely after finding the first matching entry, because matching keys in the 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 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.

[1] For example, see the implementation of `LegacyNoLocalityBloomImpl` at https://github.com/rockset/rocksdb-cloud/blob/master/util/bloom_impl.h.

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, 38]. 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).

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

Overall, the impact of false positives in LSM-trees can be calculated by considering the disk accesses due to false positives across all levels [8]. All LSM-based key-value stores employ BFs [27] or other variations like ElasticBF [21], SuRF [41, 42] and Rosetta [27].

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: 64-bit MurmurHash64 (MM64), XXHash (XX), MD5, SHA-256, CRC and CITY64 (CITY). We use the RocksDB implementation of MM64, XX, and CRC, and Google's implementation of CITY [16]. As MD5 and SHA-256 are more than one order of magnitude more expensive than other hash functions they are rarely used for practical implementations for BFs. Overall, we observe that even the most efficient hash functions are comparable with accessing data on PCIe SSDs, where hashing accounts for 10% of the access latency, and it is expected to worsen when NVM devices with DRAM-like latency become available.

Operation	Latency	Normalized
4KB I/O on HDD	4.6 ms	328571×
4KB I/O on SDD	113 μs	8071×
4KB I/O on PCIe SSD	10 μs	714×
4KB I/O on PCIe SSD (using SPDK)	7 μs	500×
4KB I/O on NVM	70 ns	5×
4KB access on Memory	**14 ns**	**1×**
CITY of 1KB-key	176 *ns*	13×
Murmur Hash 64 (MM64) of 1KB-key	235 *ns*	17×
CRC of 1KB-key	323 *ns*	23×
XXHash (XX) of 1KB-key	874 *ns*	62×
MD5 of 1KB-key	2.85 μs	203×
SHA-256 of 1KB-key	5.17 μs	378×

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

3 THE CPU COST OF BLOOM FILTERS

We now analyze the query cost in an LSM-tree focusing on the amount of time spent hashing for the BFs.

Point query in an LSM-Tree can be classified as either *empty* or *non-empty*. The latter will have to do a disk access as they target existing keys [8]. We first analyze the cost of querying a single LSM-level, level i. The cost of querying level i, $\mathcal{T}(i)$, is modeled using (i) the fraction of non-empty queries over all point queries α_i, (ii) the BF access cost (CPU cost of hashing and memory cost of probing the BF indexes) T_{BF}, and (iii) the data page access cost T_D. $\mathcal{T}(i)$ is the sum of the cost of accessing both the BF and the data for non-empty queries, $\alpha \cdot (T_{BF} + T_D)$, and of the cost of accessing the BF and the data due to false positive for the empty queries, $(1-\alpha_i) \cdot f_p \cdot T_D$, where f_p is the false positive ratio):

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

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. On the contrary, 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 non-empty queries is shown in Eq. (3).

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

Using Eq. (3), we can now understand what is the main bottleneck for point lookups. When α_i is non-negligible, the time spent to retrieve data dominates the overall LSM-tree lookup cost, because T_D corresponds to expensive accesses on slow storage. Even when α_i is very small 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 high α_i, the bottleneck shifts to hashing.

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 non-empty queries per level, the workload is oblivious to the structure of the tree, so it has an overall fraction of non-empty queries denoted as α. To compute α_i, we need the total number of queries reaching level i and the number of positive 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. Since we know that the overall fraction of non-empty queries is α, $\sum_{i=1}^{L} \beta_i$ quantifies all matched and $\sum_{i=1}^{L} \beta_i = \alpha$. Assuming that the key in the LSM-tree with size ratio T are uniformly distributed, $\beta_i = T^{i-1} \cdot \beta_1$ because every level is $T\times$ larger than the previous one, and using the size of the first level relatively to the remaining of the tree $\beta_1 = \frac{\alpha}{\sum_{j=1}^{L} T^{i-1}}$. Hence, the probability that a level finished 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 $\alpha_i = \frac{\beta_i}{1-\sum_{j=1}^{i-1} \beta_j}$, with $\alpha_1 = \beta_1$. Overall, the cost of lookup in the whole LSM-tree is shown in Eq. (4). The detailed simplifying process is shown in the appendix.

$$\begin{aligned} cost &= \mathcal{T}(1) + \sum_{i=2}^{L}(1 - \sum_{j=1}^{i-1} \beta_j) \cdot \mathcal{T}(i) \\ &= (L - \frac{\alpha}{T-1}) \cdot T_H + \alpha \cdot T_D + (L - \frac{\alpha}{T-1}) \cdot f_p \cdot T_D \end{aligned} \tag{4}$$

As shown in Eq. (4), the storage access due to true positives stay constant, while the BF related costs increase with L. Hence, the fraction of time spent on hashing increases with growing data size (and, hence, tree height) as shown earlier in Fig. 1.

4 SHARING BLOOM FILTER HASHING

We now discuss the benefits of hash sharing (a) in a single BF, (b) across BF partitions, 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 shares a single hash calculation for their k indexes [13]. For example, RocksDB uses a single hash digest and multiple indexes by rotating the hash digest. Specifically, *given hash function* $h(x)$*, we define* $\delta = h(x) << 17 | h(x) >> 15$*, 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 ratio (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, we use 10 bits per key, and thus, the optimal $k = 7$ hash functions. We execute 100K empty point queries and we 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 we report the experimentally 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 functions 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 $e^{10 \cdot (ln(2))^2} \approx 0.819\%$.

Next, we compare the lookup latency with and without bit-rotation as we vary the key size in Figure 2. 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

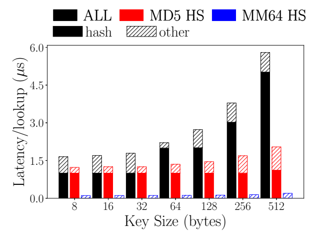


Figure 2: The benefits of hash sharing increase as the key size grows. Cheap hashing significantly reduces the average query latency.

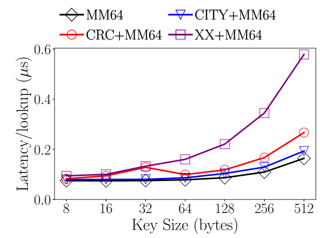


Figure 3: A single hash function with bit rotation significantly outperforms double hashing, especially as the key size increases.

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

4.2 Hash Sharing in ElasticBF

In addition to the classical BF, several BF variants have been proposed for LSM-trees that increase the hashing overhead to address more complex workloads like short range queries [27, 41], and data skew for point queries [21]. As our target workload is point queries, we focus on ElasticBF [21] which consists of multiple small filter units per BF to address access skew. By default, each filter unit employs unique a hash function to ensure that they are independent, thus increasing substantially 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 [20] 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.

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.

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 normal Bloom filter

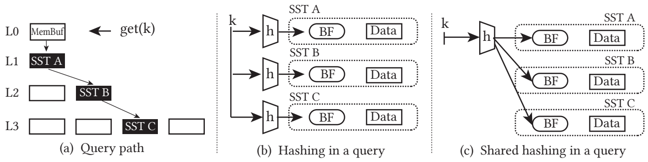
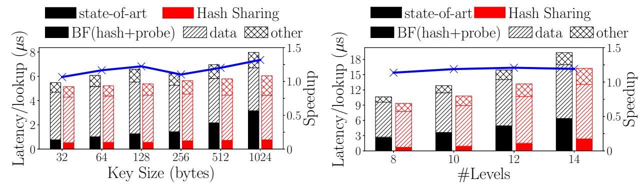


Figure 4: Hash sharing across BFs of different levels.



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

Figure 5: Hash sharing reduces hashing overhead, which is more pronounced for larger keys and higher trees.

Note that 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. 3 shows that bit-rotation leads to significantly lower query latency, up to 3× as the key size increases.

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 BF are different across levels, (the have indexed different elements), however, the expensive part of probing them, the calculation of the hash digest, is repeated for each queried level until the key is found or the tree is entirely searched. Hence, in order 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. 4). 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. (5).

$$cost^{share} \approx T_H + \alpha \cdot T_D + (L - \frac{\alpha}{T-1}) \cdot f_p \cdot T_D \tag{5}$$

Performance Implications and Discussion. Hash sharing decouples the amount of time spend 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 the 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 by a factor of L. 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 [27], Cuckoo filters [14], and Counting filters [29], can benefit from hash sharing. 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 of our machine is 384GB, and of the L3 cache is 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, with access latency less than 15μs for 4KB page accesses. Unless otherwise specified, we use the 1TB PCIe P4510 SSD with direct I/O enabled in our experimentation.

Experimental Platform, Workloads and Metrics. We build an in-house LSM-tree prototype system[2] based on the architecture of RocksDB [13], which uses RocksDB's fast local Bloom filter (format_version = 5, only supports 64-bit hash digest). We stress-test our system by varying the workload, and the execution environment. Since our design optimizes read performance, all focus on read-only workloads. We bulk load our LSM-Tree with 22GB worth of key-value pairs. The default size of the key-value entry 1KB with 512B for the key. In our experimentation, we vary the key size from 8b to 1KB, and the height of the tree from 5 to 14 levels. 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 and report the lookup latency along with a detailed breakdown of the time spent hashing, accessing data, or in other parts of the code. Each reported measurement is the average of five executions.

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. Figure 5a 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 speedup between 1.1× and 1.3× (blue line). The time breakdown shows where this benefit is coming from. The amount spend in the BFs (both hashing and probing) drastically reduced for the hash sharing approach, while the cost for accessing data, as well the remaining costs (e.g., binary search in fence pointers), remain virtually the same. In addition, larger keys have higher hashing overhead, hence, hash sharing is more beneficial for larger key sizes. The outlier in the speedup line for key size 256

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

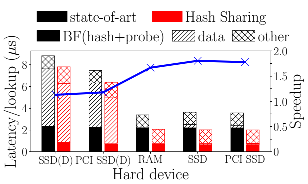


Figure 6: For faster storage, the benefit of hash sharing is pronounced.

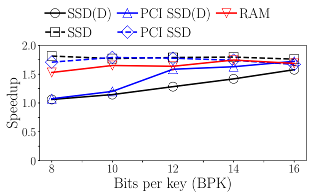


Figure 7: For fewer data accesses (lower FPR), the benefit of hash sharing is larger.

matches our expectation that MM64 has a sweet-spot for key size 256, so the overall benefit from skipping hashing is diminished. For 1KB keys, the total hashing cost is reduced to less than half ad the query performance is improved by more than 30%.

Hash Sharing Scales Better with Data Size. We now vary the bulk loaded data size, resulting in LSM-Trees that have between 8 and 14 levels (with size ratio 2). Figure 5b shows that hash sharing is increasingly helpful as the number of levels increases. The state-of-the-art system 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, hence decoupling the hashing cost from the height of the tree. In hash sharing, the increase in the BF cost is a result of the unavoidable (yet much cheaper than hashing) filter probing.

Hash Sharing Has Higher Impact for Faster Devices. For faster storage devices, the fraction of time spent hashing increases, to the point it dominates point query latency. In this experiment we vary the underlying storage device (SSD, PCIe SSD, and RAM-disk). We use the 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 the direct I/O both enabled and disabled. In Figure 6, SSD(D) and PCI SSD(D) indicates that direct I/O is enabled, while for SSD and PCI SSD it is disabled. We observe that as the storage latency reduces, hashing dominated query time, and the benefit of hash sharing increases from 20% for the off-the-self SSD to more than 80% for an emulated NVM.

Hash Sharing Has Higher Impact for Lower FPR. Using more bits per key leads to lower FPR, hence reducing the number of data accesses due to false positives, and further highlighting the benefits of hash sharing. In Figure 7, we vary the bits per key from 8 to 14 and we report the speedup for different storage devices. We observe that the benefit of hash sharing increases for more bits-per-key (lower FPR). When BPK is 10 (a typical setting in state-of-the-art LSM-trees), hash sharing improves the query performance by 20% on PCIe SSD, and by more than 65% 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 increase the ratio of queries that will return a positive result, α, and as a result, they will make at least one disk access. Figure 8 shows the latency breakdowns of the state-of-the-art and hash sharing on the SSD. As expected, we observe that that percentage of time spent hashing reduces as α increases. The cost of hashing is constant, however, the average data retrieval cost increases as the number of non-empty queries increases. Figure 9 shows the same experiment on the emulated NVM device. We now observe

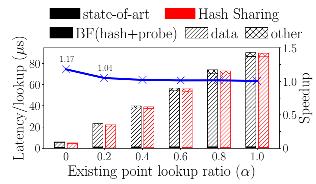


Figure 8: Hash sharing benefits reduce as the existing point lookup ratio grows.

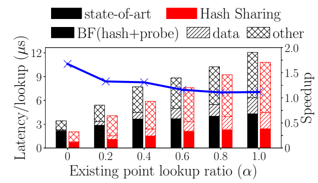


Figure 9: The speedup on RAM disk is pronounced and has similar pattern as SSD

that while the benefit reduces as α increases, it starts from 65% for $\alpha = 0$ and it stays more than 15% for $\alpha = 1$.

6 RELATED WORK

The hashing cost of BFs has been identified as a key optimization, especially when the cost a false positive is low. 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) [20] and One-Hashing Bloom filters (OHBF) [22] aim to achieve the same false positive ratio while reducing the hashing cost. LHBF divides the filter into k partition with identical size, and calculates the index for partition i using a hash function, $g_i(x) = h_1(x) + i \cdot h_2(x)$, based on two hash functions, $h_1(x)$ and $h_2(x)$. 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 can be implemented by using only one hash function and a few modulo operations.

Orthogonally to the hashing cost, there have been efforts to reduce the probing cost focusing on the locality of bit-vector accesses [31, 32]. Blocked Bloom filters (BBF) [31] 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 [32] 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 BFs*. Hence, our design can be combined with any techniques 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 the main 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 leading and leads to performance benefits varying from 20% for an our PCIe SSD to more than 65% for an emulated NVM.

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. [n.d.]. Cassandra. *http://cassandra.apache.org* ([n. d.]).

[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] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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)*.

[13] Facebook. [n.d.]. RocksDB. *https://github.com/facebook/rocksdb* ([n. d.]).

[14] 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.

[15] Google. [n.d.]. LevelDB. *https://github.com/google/leveldb/* ([n. d.]).

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

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

[18] 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.

[19] 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.

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

[21] 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.

[22] 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.

[23] 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.

[24] 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.

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

[26] 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.

[27] 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.

[28] 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.

[29] 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.

[30] 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.

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

[32] 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.

[33] 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.

[34] 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.

[35] 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.

[36] 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.

[37] 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).

[38] 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.

[39] WiredTiger. [n.d.]. Source Code. *https://github.com/wiredtiger/wiredtiger* ([n. d.]).

[40] 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.

[41] 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.

[42] Huanchen Zhang, Hyeontaek Lim, Viktor Leis, David G Andersen, Michael Kaminsky, Kimberly Keeton, and Andrew Pavlo. 2020. Succinct Range Filters. *ACM Transactions on Database Systems (TODS)* 45, 2 (2020), 5:1—-5:31.

[43] 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.

[44] 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)*.

Appendices

The cost of lookup in the whole LSM-tree in Eq. (4) can be expanded to (6) using Eq. (3) assuming that memory probing (T_P) is negligible.

$$\begin{aligned} cost \approx& T_H + \alpha_1 \cdot T_D + (1-\alpha_1) \cdot f_p \cdot T_D \\ &+ \sum_{i=2}^{L} \left(1 - \sum_{j=1}^{i-1} \beta_j\right) \cdot \left[T_H + \alpha_i \cdot T_D + (1-\alpha_i) \cdot f_p \cdot T_D\right] \end{aligned} \tag{6}$$

Since $\alpha_i = \frac{\beta_i}{1-\sum_{j=1}^{i-1}\beta_j}$ and $\alpha_1 = \beta_1$, Eq. (6) becomes Eq. (7).

$$\begin{aligned} cost \approx& T_H + \beta_1 \cdot T_D + (1-\beta_1) \cdot f_p \cdot T_D \\ &+ \sum_{i=2}^{L} \left(1 - \sum_{j=1}^{i-1} \beta_j\right) \cdot \left[T_H + \frac{\beta_i \cdot T_D}{1-\sum_{j=1}^{i-1}\beta_j} + (1 - \frac{\beta_i}{1-\sum_{j=1}^{i-1}\beta_j}) \cdot f_p \cdot T_D\right] \\ &= \left(L - \sum_{i=2}^{L}\sum_{j=1}^{i-1}\beta_j\right) \cdot T_H + \sum_{i=1}^{L}\beta_i \cdot T_D + \left(L - \sum_{i=2}^{L}\sum_{j=1}^{i-1}\beta_j - \sum_{i=1}^{L}\beta_i\right) \cdot f_p \cdot T_D \end{aligned} \tag{7}$$

Since $\sum_{i=1}^{L}\beta_i = \alpha$, Eq. (7) becomes Eq. (8).

$$cost \approx \left(L - \sum_{i=2}^{L}\sum_{j=1}^{i-1}\beta_j\right) \cdot T_H + \alpha \cdot T_D + \left(L - \sum_{i=2}^{L}\sum_{j=1}^{i-1}\beta_j - \alpha\right) \cdot f_p \cdot T_D \tag{8}$$

If we assume that keys in the LSM-tree is 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 $\beta_1 = \frac{\alpha}{\sum_{j=1}^{L} T^{i-1}}$. Thus, $\sum_{i=2}^{L}\sum_{j=1}^{i-1}\beta_j$ can be approximated as Eq. (9).

$$\begin{aligned} &\sum_{i=2}^{L}\sum_{j=1}^{i-1}\beta_j = \sum_{i=2}^{L}\sum_{j=1}^{i-1} T^{j-1} \cdot \beta_1 = \\ &= \beta_1 \cdot \sum_{i=2}^{L} \frac{T^{i-1}-1}{T-1} = \frac{\beta_1}{T-1} \cdot \left(\sum_{i=2}^{L}\left(T^{i-1}\right) - (L-1)\right) = \\ &= \frac{\beta_1}{T-1} \cdot \left(\sum_{i=1}^{L}\left(T^{i-1}\right) - 1 - (L-1)\right) = \frac{\beta_1}{T-1} \cdot \left(\sum_{i=1}^{L}\left(T^{i-1}\right) - L\right) = \\ &= \frac{\alpha}{\sum_{k=1}^{L} T^{k-1}} \cdot \frac{1}{T-1} \cdot \left(\sum_{i=1}^{L}\left(T^{i-1}\right) - L\right) = \frac{\alpha}{T-1} \cdot \left(1 - \frac{L}{\sum_{k=1}^{L} T^{k-1}}\right) = \\ &= \frac{\alpha}{T-1} \cdot \left(1 - \frac{L}{\frac{T^L-1}{T-1}}\right) = \frac{\alpha}{T-1} \cdot \left(1 - \frac{L \cdot (T-1)}{T^L - 1}\right) \\ &(\text{for } T > 3 \text{ or } L > 3) \approx \frac{\alpha}{s-1} \end{aligned} \tag{9}$$

Therefore, the cost of lookup in the whole LSM-tree can be simplified as in Eq. (4).