ElasticBF: Elastic Bloom Filter with Hotness Awareness for Boosting Read Performance in Large Key-Value Stores

Yongkun Li, Chengjin Tian, Fan Guo, Cheng Li, and Yinlong Xu,
University of Science and Technology of China

https://www.usenix.org/conference/atc19/presentation/li-yongkun

This paper is included in the Proceedings of the
2019 USENIX Annual Technical Conference.

July 10–12, 2019 • Renton, WA, USA

ISBN 978-1-939133-03-8

ElasticBF: Elastic Bloom Filter with Hotness Awareness for Boosting Read Performance in Large Key-Value Stores

Yongkun Li, Chengjin Tian, Fan Guo, Cheng Li, Yinlong Xu
University of Science and Technology of China

Abstract

LSM-tree based key-value (KV) stores suffer from severe read amplification because searching a key requires to check multiple SSTables. To reduce extra I/Os, Bloom filters are usually deployed in KV stores to improve read performance. However, Bloom filters suffer from false positive, and simply enlarging the size of Bloom filters introduces large memory overhead, so it still causes extra I/Os in memory-constrained systems. In this paper, we observe that access skewness is very common among SSTables or even small-sized segments within each SSTable. To leverage this skewness feature, we develop ElasticBF, a fine-grained heterogeneous Bloom filter management scheme with dynamic adjustment according to data hotness. ElasticBF is orthogonal to the works optimizing the architecture of LSM-tree based KV stores, so it can be integrated to further speed up their read performance. We build ElasticBF atop of LevelDB, RocksDB, and PebblesDB, and our experimental results show that ElasticBF increases the read throughput of the above KV stores to 2.34×, 2.35×, and 2.58×, respectively, while keeps almost the same write and range query performance.

1 Introduction

With the exponential growth of data volume, traditional relational database meets challenges in scalability in dealing with extremely large-scale data. As an alternative, key-value (KV) store is widely used as the fundamental storage infrastructure in many applications, such as cloud systems [25], advertising [8, 16], social networks [3, 37], search indexing [8, 21], online gaming [13], etc. According to the used index structures, KV stores can be categorized into hash index based design [12, 13, 26], B-tree based design [20, 31] and LSM-tree based design [27, 30, 32, 36]. Because hash index based design requires large memory and can not well support range query, and B-tree based design involves an abundance of random writes, so most modern KV stores use LSM-tree, e.g., LevelDB [21] at Google, RocksDB [16] at Facebook, Dynamo [14] at Amazon, and Cassandra [1] at Apache.

An LSM-tree based KV store is typically composed of two components, and we take LevelDB as an example to illustrate. One resides in memory to cache KV pairs, and it includes a MemTable and an Immutable MemTable. The other is stored in secondary storage, which is divided into multiple levels consisting of multiple SSTables. Each SSTable contains a set of sorted KV pairs and necessary metadata. When a level reaches its size limit, its SSTables will be compacted into the next level via compaction, which first reads out the SSTables in the two levels, then performs a merge sort, and finally writes back the new SSTables into the next level. As a result, compaction induces severe write amplification [27, 32].

Various designs are recently proposed to mitigate the compaction overhead [5, 27, 32, 42]. The most recent work PebblesDB [32] proposes a novel data structure called Fragmented Log-Structured Merge-Trees (FLSM), which follows the LSM-tree structure, but relaxes the fully sorted constraint in each level. Thus, compaction just needs to append data to multiple fragments in the next level, and there is no need to merge with the SSTables in the next level. Therefore, PebblesDB can greatly reduce the write amplification. However, sacrificing the sorted nature of KV pairs in each level, PebblesDB inevitably degrades the read performance.

On the other hand, LSM-tree based KV stores also suffer from severe read amplification [27, 31, 39]. This is because when lookup a KV pair, KV store needs to check multiple SSTables from the lowest level to the highest level until the key is found or all levels have been checked. Furthermore, it is required to read multiple metadata blocks to really check whether a KV pair exists in one SSTable. Thus, read amplification can reach a factor of over 300× [27]. To reduce extra I/Os induced by checking multiple SSTables, modern designs use Bloom filters in KV stores [35] to quickly check the existence of a KV pair in an individual SSTable. However, Bloom filters suffer from false positive, so they may return a positive result even when a KV pair does not exist in the SSTable, and this incurs unnecessary I/Os. We also conduct experiments to measure the impact of false positive by using Bloom filters with four `bits-per-key` in a 100GB KV store. Other parameter settings and system configurations are presented in §4.1. Results show that a key lookup incurs 7.6 inspections to SSTables on average, but 1.3 disk accesses are needless and induced by false positive.

Allocating more bits for each key can reduce the false positive rate, but the volume of all Bloom filters also increases, e.g., a 10TB KV store with 100B KV pairs requires 128GB Bloom filters under 10 `bits-per-key` setting. If the volume of Bloom filters exceeds the memory capacity, some filters will be swapped out to secondary storage, which induces extra I/Os and increases the read amplification.

It is a consensus that data access is usually skewed in real applications [4, 9]. Some KV pairs are hot and frequently accessed, while most KV pairs are seldom accessed. Thus, if we allocate more bits to the Bloom filters for hot KV pairs, then we can reduce the overall false positive rate in the whole process of running an application, while still limit the volume of all Bloom filters. However, it is not an easy task to dynamically adjust the setting of Bloom filters in KV stores due to the following reasons, which are revealed by our experiments. First, different levels exhibit significantly different access unevenness. Even though KV pairs in a lower level are usually more frequently accessed, there are still a considerable proportion of SSTables in a higher level that are evidently hotter than SSTables in a lower level. Second, even for KV pairs within the same SSTable, the access unevenness is also serious. Last but not least, the hotness of KV pairs dynamically changes during the running time of applications. Monkey [10] proposes a heterogeneous scheme which allocates more bits to Bloom filters in lower levels, but it uses the same setting for all filters in the same level and fails to dynamically adjust the setting according to data hotness.

In this paper, we propose ElasticBF, a fine-grained and elastic Bloom filter management scheme. Its basic idea is assigning multiple small-sized Bloom filters to each small group of KV pairs when building SSTables, and these Bloom filters reside in secondary storage and are dynamically loaded into memory to activate according to the hotness of KV pairs. However, to realize dynamical allocation of Bloom filters, the following key issues must be addressed: (1) How to accurately measure and record the hotness of KV pairs with low storage and CPU overheads? (2) How to dynamically change the ability of Bloom filters based on hotness with low memory and computation overheads? (3) How to efficiently inherit the hotness of SSTables with low metadata overhead when SSTables are reorganized during compaction? ElasticBF carefully addresses the above issues by developing multiple techniques to limit its overhead, including *fine-grained allocation*, *hotness inheritance* and *in-memory management optimization*.

We emphasize that ElasticBF is orthogonal to existing works focusing on the optimization of KV store structures, so it can be combined with these designs to further accelerate the lookup of KV pairs. To demonstrate its efficiency, we carefully build ElasticBF atop three commonly used or state-of-the-art KV stores, LevelDB, RocksDB and PebblesDB. Experiments show that for all of the above KV stores, ElasticBF increases the read throughput to 2.34×, 2.35×, and 2.58×, respectively, while keeps almost the same write performance. Also, for workloads with mixed reads and writes, ElasticBF reduces read latency by 38.9% - 51.8% without affecting writes. Compared with Monkey, which is the state-of-the-art heterogeneous Bloom filter management scheme, ElasticBF achieves up to 2.20× throughput. In summary, our contributions are as follows.

- **Fine-grained allocation.** We find that the hotness of KV pairs varies significantly in different ranges within the same SSTable. So we divide each SSTable into different segments, measure and record their hotness with acceptable storage and CPU overheads, so as to receive relatively accurate hotness estimation and enable fine-grained Bloom filter allocation.

- **Hotness inheritance.** We design an effective scheme to estimate the access frequencies of new SSTables during compaction by inheriting from the outdated SSTables. So ElasticBF can preserve the hotness during the whole execution process of applications, and avoids frequent cold start of hotness due to compaction and consistently improves read performance.

- **In-memory management optimization.** We propose a Multi-Queue to manage the in-memory Bloom filters, and use parallel I/Os to accelerate the adjustment. Thus, we can dynamically adjust the Bloom filters in memory according to the hotness of KV pairs with negligible CPU overhead.

The rest of this paper is as follows. §2 introduces LSM-tree based KV stores and shows our observations on access skewness to motivate this work. §3 presents the design details of ElasticBF and §4 evaluates its performance. Finally, §5 reviews related work and §6 concludes this paper.

2 Background & Motivation

In this section, we first briefly introduce LSM-tree, then analyze the read amplification in LSM-tree based KV stores, and finally present our observations about data access locality in KV stores to motivate this work.

2.1 Log-Structured Merge Trees

Figure 1 illustrates the structure of an LSM-tree based KV store, which mainly consists of two components. One is in memory, which includes a *MemTable* and an *Immutable MemTable*. The other is in secondary storage, which is divided into multiple levels, say L_0, L_1, $\cdots$, L_k, where k depends on the KV store size. Besides, the size limit of level L_i is usually m times of that of level L_{i-1} for $1 \leq i \leq k$, e.g., $m = 10$ in LevelDB by default.

Now we illustrate how data is stored. Specifically, KV pairs are first written to the *MemTable* which works as a cache. When *MemTable* is filled up, it will be converted to an *Immutable Memtable*, which can not be written any more.

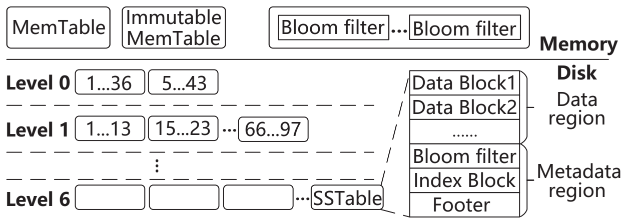


Figure 1: Structure of LSM-tree based KV store.

Later, *Immutable Memtable* will be packed into an SSTable and appended into L_0 in the secondary storage. Note that keys in each SSTable are sorted, but they are not sorted between SSTables in L_0 so as to make the writes to disk fast. Thus, this write policy degrades read performance, so the size of L_0 is usually limited, e.g., it is limited as 12 SSTables in LevelDB. To balance read and write performance, SSTables are organized into a multi-level tree, and if one level (say L_{i-1}) is filled up, its SSTables will be merged into its higher level (say L_i) by compaction, which merges all KV pairs in L_{i-1} into L_i, so data are sorted in every level except for L_0.

To find a key in the secondary storage, we need to search it level by level from L_0 to L_k. Note that we should check all SSTables in L_0 because they are not sorted, while we only need to check one SSTable in each of the other levels until we find the key or all levels are checked. Thus, we usually need to read multiple SSTables to find a key, which induces read amplification, and Bloom filters are commonly used to reduce the read amplification. As a result, besides KV pairs, each SSTable also includes Bloom filters and other metadata (see Figure 1). For performance consideration, the Bloom filters are usually required to be also buffered in memory.

However, Bloom filters suffer from false positive because of hash collision, and thus incur extra I/Os to read out data from SSTables for key comparison. The false positive rate of a Bloom filter is $(1-e^{-k/b})^k$, where b is the number of bits allocated to each key, i.e., `bits-per-key`, and k means the number of hash functions [24]. Since $(1-e^{-k/b})^k$ is minimized when $k = ln2 \cdot b$, false positive rate can be simply represented as 0.6185^b. Thus, the value of b directly determines the memory usage of a Bloom filter. We can reduce the false positive rate by allocating more `bits-per-key` for Bloom filters, but allocating more bits to each keys will increase the volume of all Bloom filters and thus consumes more memory. Even worse, if the volume of all Bloom filters exceeds the memory capacity, some Bloom filters will be swapped out to secondary storage, and this will induce extra I/Os and further aggravate read amplification.

2.2 Motivation

Uneven accesses are still very common in KV stores [9, 22], where only a small proportion of the KV pairs are frequently accessed, while the majority of the KV pairs are seldom accessed. Therefore, if we allocate more bits to the Bloom filters for hot KV pairs and fewer bits for cold ones, then the overall positive false rate during the whole execution process of applications will be reduced. Clearly, we will face to a series of challenges to realize such heterogeneous Bloom filters and enable dynamic adjustment. In this subsection, we present our observations on the access skewness of KV stores to motivate this work. The detailed design of ElasticBF will be presented in §3.

We run experiments with RocksDB to validate the access unevenness in KV stores. We use YCSB [9] to load a 256GB

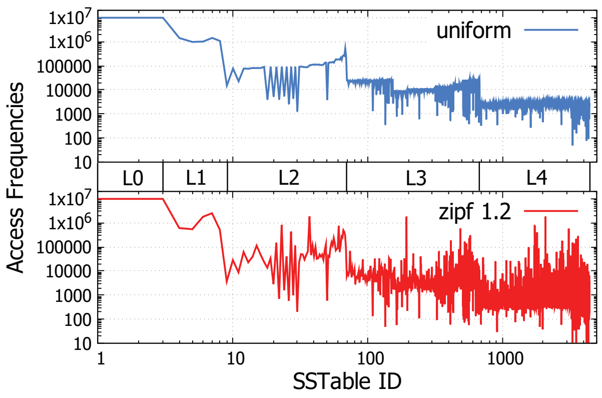


Figure 2: File access frequencies under different workloads.

database in the experiments, where the size of each SSTable is set as 64MB, which is the default configuration, and the size of each key value pair is 1KB. Note that the maximum size of L_1 is configured as 256MB in RocksDB, and the size of L_i is 10 times of that in L_{i-1} ($i \geq 2$), therefore 5 levels are enough to keep 256GB data. We then generate two representative workloads with uniform and Zipfian distributions, and each workload contains ten million Get requests. Note that there are about 4400 SSTables in the tested KV store, so issuing ten million Get requests is enough to study the access pattern. To make the evaluation of the hotness of SSTables and the hotness of different regions in the same SSTable accurate, we disable the Bloom filters in these experiments. That is, we search the keys level by level, and at each level, we compare the target key with the key ranges of SSTables. If the key falls into the range of an SSTable, we will read the data out and check whether the key exists or not until the key is found or all levels are checked.

We first show the file-level access characteristics, and Figure 2 shows the access frequency of each SSTable. The x-axis represents the identities of SSTables which are numbered sequentially from the lowest level to the highest level, and the y-axis shows the number of accesses to each SSTable. From the results, we can have two observations. First, on average, the access frequencies of SSTables in lower levels are higher than those in higher levels. This is because lookup always flows from lower levels to higher levels. Second, if we zoom in one particular level, we can find that the access frequencies vary very significantly from SSTables, i.e., some SSTables are much hotter than others within each level. Besides, when we compare the access frequencies of SSTables in adjacent two levels, we can find that it is very common to have some SSTables in level L_{i+1} which are even hotter than some SSTables in level L_i, especially for the skewed workload with Zipf distribution. That is, SSTables in higher levels may also be hotter than those in lower levels. For example, 21% of SSTables in L_4 is even hotter than 11% of SSTables in L_3. More importantly, since more than 98% SSTables are stored in the highest two levels, i.e., L_3 and L_4 in this example, we can conclude that the hotness of most SSTables can not be

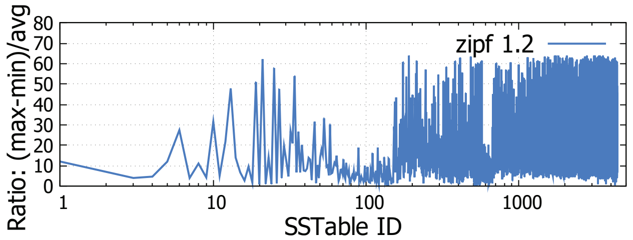


Figure 3: The ratio of the difference between the maximum and the minimum access frequencies of different regions to their average access frequency within each file.

accurately characterized according to which level they are placed. This observation implies that Monkey's level-based coarse-grained heterogeneous Bloom filters can not take full advantage of the access skewness, and finer-grained Bloom filter design is necessary.

Since the size of an SSTable may still be configured to be large to leverage sequential I/O bandwidth, e.g., RocksDB uses 64MB or even larger SSTables, the access skewness may still be serious within each SSTable. Note that it will bring very large memory and CPU overheads to record the hotness of each KV pair, so we divide each 64MB SSTable into 64 regions, each of which has 1MB, and record the access frequencies of different regions in each SSTable. Figure 3 shows the ratio of the difference between the maximum and the minimum access frequencies of all regions in the same SSTable to their average access frequency. We can see the ratio value is very large for many SSTabes, e.g., greater than 10 for 73% SSTables. So the access unevenness is very serious even within the same SSTable.

In summary, access skewness is very serious among different SSTables, and even among different regions within the same SSTable. This offers an opportunity to develop finer-grained heterogeneous Bloom filters by allocating more `bits-per-key` for hot SSTables or regions so as to reduce the overall false positive rate without increasing the volume and memory overhead of Bloom filters.

3 Design

The main idea of ElasticBF is to construct multiple Bloom filters for each SSTable, but allocate less `bits-per-key` to each filter. Note that the Bloom filters in SSTables are stored in secondary storage, and they are just reserved for future use. That is, Bloom filters become active only after being loaded into memory, which is a dynamical process according to the hotness of SSTables. If an SSTable becomes hot, we will load more of its Bloom filters into memory, and we may also disable some of its Bloom filters in memory when it becomes cold. Thus, we can dynamically adjust Bloom filters while avoiding heavy I/O and CPU overheads for computing the hash functions when we change Bloom filters according to the hotness of SSTables. Thanks to the dynamic allocation and adjustment of Bloom filters, we can reduce the overall false positive rate, while keeping the same memory usage.

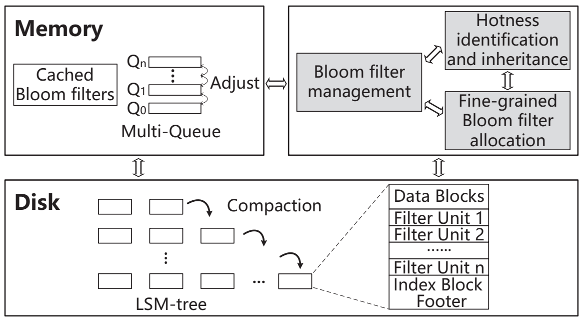


Figure 4: The architecture of ElasticBF.

3.1 Overview

Figure 4 depicts the architecture of ElasticBF, which mainly contains three components, *fine-grained Bloom filter allocation*, *hotness identification and inheritance*, and *Bloom filter management in memory*. For the design of fine-grained Bloom filter allocation, we mainly face to the problems of how many Bloom filters should be allocated to each SSTable and how many bits should be assigned to each filter so as to achieve low false positive rate and low memory usage. We also need to carefully design the data structure and management scheme with low I/O overhead. For hotness identification, our goal is to achieve relatively accurate estimation of hotness with low overhead. Finally, hotness inheritance is designed to avoid cold start of hotness identification after compaction, and Bloom filter management in memory is to efficiently adjust Bloom filters according to hotness.

Remarks: ElasticBF realizes an elastic Bloom filter management scheme with little extra memory usage and small CPU and I/O overheads. It is orthogonal to existing works focusing on optimizing the structure of KV stores and can be integrated to accelerate their read performance. Besides, ElasticBF may also be applied in other scenarios to improve object lookups, and the management technique for hot/cold adaption is applicable to other summary data structures.

3.2 Fine-grained Bloom Filter Allocation

The read performance of KV stores can be improved by reducing the I/Os caused due to false positive of Bloom filters. In the following, we first analyze the expected false positive rate by dynamically activating Bloom filters according to hotness, then we describe how to construct multiple Bloom filters for SSTables to realize fine-grained allocation.

Construction of multiple Bloom filters. ElasticBF generates multiple Bloom filters for each SSTable by using different and independent hash functions. Each filter is allocated with less `bits-per-key`, and we call it a *filter unit*. All filter units assigned to an SSTable are named as a *filter group*, as shown in Figure 5. Since the multiple filters within a filter group are independent, a key is certainly not in an SSTable as long as one filter unit returns a negative answer. That is, if multiple filter units are enabled, then only when all enabled

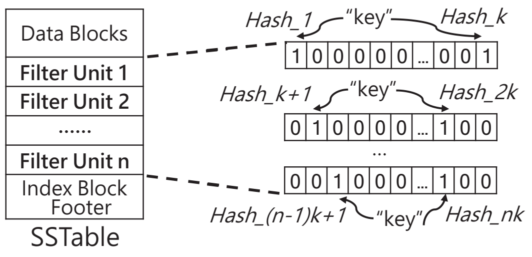


Figure 5: Construction of multiple Bloom filters.

filter units indicate the existence of a key, we need to read out the SSTable to search the key.

As pointed out in [24], the false positive rate of a filter group is equivalent to a single Bloom filter which has the same `bits-per-key` to all filter units within the filter group. We call this feature *separability*, which can be further justified as follows. Assume that a filter group consists of *n* filter units, each of which is a b/n `bits-per-key` filter, then the false positive rate of each filter unit is $0.6185^{b/n}$. Since the filter units in a group are generated by different independent hash functions, the false positive rate of n filter units in a group is $(0.6185^{b/n})^n = 0.6185^b$, which is exactly the same with that of a single Bloom filter with b `bits-per-key`.

Based on the separability feature, we should determine b and n to optimize the setting of multiple Bloom filters. As we should enable all the filter units in a group for the hottest SSTables such that the false positive rate 0.6185^b closes to zero, in our configuration, we set $b = 24$ with a false positive rate of about 0.001%. On the other hand, n indicates the maximum number of hotness categories to distinguish different SSTables. Increasing n will more accurately differentiate the hotness of SSTables, but it needs more I/Os to load filter units to achieve low false positive rate. Thus, we set $n = 6$, i.e., a filter group has 6 filter units, each is allocated with 4 `bits-per-key`. We will analyze the impact of `bits-per-key` of each filter unit on the read performance in §4. Notice that ElasticBF allocates multiple filter units to SSTables, which induces extra storage usage. Assuming that the size of KV pairs is 1KB, thus one group of filter units cost about 192KB storage, which is only 0.3% of a 64MB SSTable, and the filters are stored in secondary storage, so the storage overhead of ElasticBF is negligible.

Benefit analysis. Now we analyze the benefit of using multiple Bloom filters. Suppose that there are N SSTables, $s_1, s_2, ..., s_N$, in a KV store, and we use static setting to set b `bits-per-key` for all SSTables, i.e., the memory usage to reside Bloom filters is b bits for each key. Suppose that a workload needs to access SSTable s_i with p_i times, then the expected number of times to really read out data from all SSTables due to false positive with static setting is

$$R_{static} = \sum_{i=1}^{N} p_i \cdot 0.6185^b. \tag{1}$$

In a contrary, if we dynamically set multiple Bloom filters with the same memory usage as the static setting, and suppose

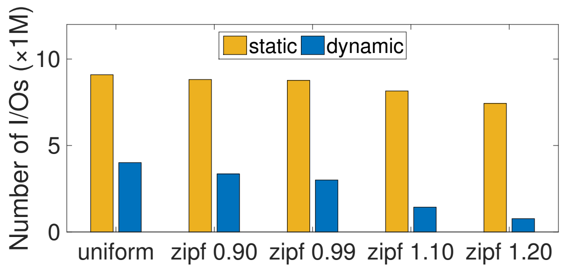


Figure 6: Number of I/Os caused due to false positive.

we load n'_i filter units for SSTable s_i according to its hotness, each allocates b' `bits-per-key`, then under the assumption of the same memory usage with static setting, the expected number of times to read out data from SSTables is

$$\begin{aligned} R_{dynamic} &= \sum_{i=1}^{N} p_i \cdot \left(0.6185^{b'}\right)^{n'_i}, \\ \text{subject to} &\quad \sum_{i=1}^{N} n'_i \times b' \le Nb, \end{aligned} \tag{2}$$

where the inequality $\sum_{i=1}^{N} n'_i \times b' \le Nb$ represents the same memory usage constraint.

To better understand why dynamical allocation can reduce I/Os, i.e., $R_{dynamic} < R_{static}$, we count the number of I/Os due to false positive by conducting experiments on RocksDB. We set the average `bits-per-key` as 4 for both Bloom filter allocation schemes. For the dynamical allocation scheme, we generate 6 filter units for each SSTable and still use 4 `bits-per-key` for each filter unit. We first issue ten million Get requests on a 100GB database using static setting of Bloom filters, half of the Get operations request non-existent items, and we count the real number of I/Os issued due to false positive as R_{static}. Then we classify SSTables into 7 categories (C_0, C_1, ..., C_6) according to their access frequencies, and initially load i filter units for SSTables in C_i in the dynamical allocation. We use this configuration to replay the ten million Get requests and count $R_{dynamic}$. Figure 6 shows the results under different workloads, we find that with the same memory usage, dynamic setting of Bloom filters reduces the number of I/Os caused due to false positive under different workloads by 55.9% - 89.7% compared to static setting, and the reduction becomes larger for more skewed workloads. Note that in practical systems, Bloom filters may be configured with larger `bits-per-key` so as to achieve very low false positive rate, dynamical allocation still has its benefit, e.g., it can use much less memory to achieve similar false positive rate.

Finer-grained design with chunking. As mentioned in Section 2.2, access unevenness is still serious within an SSTable. So we may further reduce the false positive rate by differentiating the hotness of keys within the same SSTable. However, this will bring too large extra overheads of memory usage and CPU for recording the hotness of individual keys. To balance the accuracy of measuring hotness and the extra

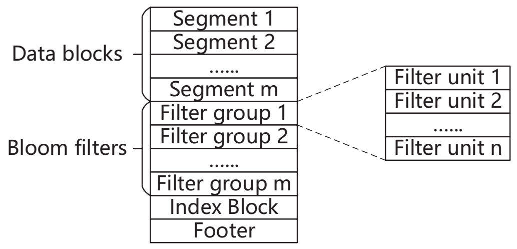


Figure 7: Segments in one SSTable.

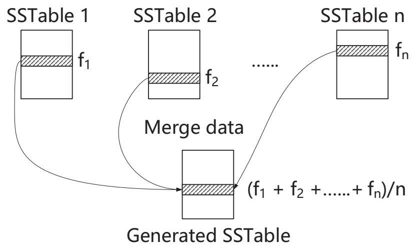


Figure 8: Hotness inheritance after compaction.

overheads to KV stores, we further divide each SSTable into multiple regions called *segments* and record the hotness at the granularity of the segment. Each segment is then allocated a group of filter units, as shown in Figure 7. From §2.2, we know the hotness of different segments still significantly varies, so we are still expected to reduce the false positive rate by differentiating the hotness of segments.

The challenging issue is to optimize the size of segments, as large segment can not accurately reflect the hotness of different KV pairs in an SSTable and small segment will bring large overhead to KV stores. Our rule to configure the size of segment is to make the size of each Bloom filter be close to the device block size, e.g., 4KB, so as to reduce the I/O overhead when loading Bloom filters. We will analyze the impact of segment size on the read performance in §4.

Finally, since we only need several bytes to record the hotness for each segment, the memory overhead is smaller than 1% of the Bloom filter size. For storage overhead, if the KV pair size is 1KB and ElasticBF uses 4 `bits-per-key` for each filter unit, then one filter unit only costs around 2KB storage space, which is only 0.05% of a 4MB segment.

Remarks: All Bloom filters allocated to an SSTable are stored in its metadata area and kept in secondary storage. Upon reading an SSTable, the default number of Bloom filters are also loaded into memory, so loading Bloom filters at initialization does not induce extra I/Os.

3.3 Hotness Identification and Inheritance

Hotness identification. The hotness of a segment is determined by its access frequency and the time duration since its last access. Specifically, we propose an *expiring policy* to differentiate hot/cold segments. We maintain a global variable named `currentTime`, which is defined as the total number of Get requests issued to the whole KV store so far, and we also associate a variable named `expiredTime` with each segment to denote the time point at which the segment will be "expired". Precisely, `expiredTime` is defined as `lastAccessedTime + lifeTime`, where `lastAccessedTime` denotes the time of the most recent access to the segment and `lifeTime` is a fixed constant. Note that the "time" concept here means logical time which is actually represented by the number of accesses. Each time when a segment is accessed, we increase the `currentTime` by one and update the `expiredTime` of this segment by setting `lastAccessedTime` as the updated value of the `currentTime`. We define a segment as "expired" if `currentTime` already becomes larger than `expiredTime`. The physical meaning of the above policy is that if a segment is not accessed during a fixed number of Get requests which is defined by `lifeTime`, then it is expired and considered as cold. Note that the time complexity to update the hotness metadata of a segment is only $O(1)$. Besides, the memory overhead is also small, e.g., for a 100GB KV store, there are around 25K segments whose size is 4MB, assume that 4 bytes are used to record the `expireTime` of each segment, then the total memory overhead is only around 100KB.

Hotness inheritance after compaction. Compaction will trigger merge sort between SSTables to generate new SSTables. So the segments in new SSTables are also newly generated and their hotness should be changed. If we set the hotness of new segments as 0, then ElasticBF will face to cold start of hotness and this may degrade the performance of future read from the new segments. To inherit the hotness, ideally, we can accurately estimate the hotness of new segments based on the hotness of all keys within it, but this will bring into KV stores too large overhead.

Instead, ElasticBF uses the hotness of old segments to estimate the hotness of new segments. Specifically, as illustrated in Figure 8, when a new segment is generated, we first find out the old segments which are involved in the procedure of generating the new segment and also have overlapped key ranges, then we estimate the hotness value of the new segment by simply using the mean of the hotness of all old segments. At last, we enable some filter units for the new segment accordingly. Experiments in §4 show that this simple scheme is efficient to improve the read performance for workloads with mixed reads and writes in KV stores.

3.4 Bloom Filter Management in Memory

Now the final issue is to determine how many filter units should be enabled for each segment. Although we can address this issue by formulating an optimization problem to minimize the overall false positive rate, but this will consume lots of CPU resources. Besides, every access changes the access frequency of some segment, so it needs to recompute the optimal solution and incurs lots of I/Os to adjust the optimal configuration. To address this issue, ElasticBF develops a lightweight and efficient adjustment scheme.

Bloom filter adjusting rule. We use a metric which is defined as the expected number of I/Os caused by false positive to guide the adjustment, and we denote this amount of extra I/Os as $E[\text{Extra\_IO}]$, which can be expressed as

$$E[\text{Extra\_IO}] = \sum_{i=1}^{M} f_i \times r_i, \quad (3)$$

where M means the total number of segments in the KV store, f_i denotes the access frequency of segment i, r_i denotes the false positive rate and it is determined by the number of filter units loaded in memory for segment i. Here, the rule of thumb is to adjust the number of filter units, and thus changes r_i, so as to make $E[\text{Extra\_IO}]$ be decreased.

The procedure of adjusting Bloom filters is as follows. Each time when a segment is accessed, we update its access frequency and the $E[\text{Extra\_IO}]$, then we check whether $E[\text{Extra\_IO}]$ could be decreased if we enable one more filter unit for this segment and disable one unit for other segment to guarantee the same memory usage. If the $E[\text{Extra\_IO}]$ could be decreased, then we apply the adjustment, otherwise, we do nothing. Note that in this adjusting procedure, one key issue is to find out which filter unit should be disabled, and we address this problem by maintaining an in-memory index based on Multi-Queue, which will be described later.

Realizing dynamic adjustment with Multi-Queue. Recall that the challenging issue in the adjust procedure is to decide which filter unit should be disabled. We extend Multi-Queue (MQ) [33,46] to address this problem. Specifically, ElasticBF maintains multiple in-memory Least-Recently-Used (LRU) queues to manage the metadata of each segment as shown in Figure 9. We denote these queues as Q_0,...,Q_n, where n is equal to the maximum number of filter units allocated to each segment. Each element of a queue corresponds to one segment, and it manages the filter units enabled for the segment. Precisely, each element in queue Q_i indicates that i filter units are enabled for the corresponding segment, i.e., only these i filter units are used to check the existence of keys. To keep the LRU feature of each queue, each time when a segment is accessed, we move the corresponding element to the MRU side within the same queue.

To find out which filter unit should be disabled and then removed from memory, we use the hotness information defined by the *expiring policy* described in §3.3. Specifically, we search "expired" segments from Q_n to Q_1, and for each queue, we search from the LRU side to the MRU side, since an "expired" segment must be the least recently used one. When we find an "expired" segment, and if the $E[\text{Extra\_IO}]$ can be decreased when we disable one filter unit of this segment, we then downgrade it to the next lower-level queue to release one filter unit. Note that the access frequency of the "expired" segment does not change, while the $E[\text{Extra\_IO}]$ could be decreased because of the change of false positive rates by adjusting the Bloom filters in corresponding segments. If there is no "expired" segment,

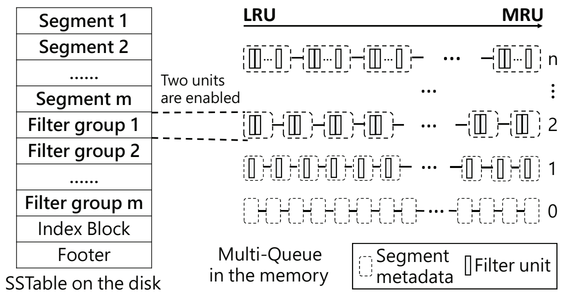


Figure 9: The in-memory Multi-Queue in ElasticBF.

we skip the Bloom filter adjustment this time, this is a conservative strategy to prevent us from degrading the lookup efficiency of possible hot segments (which are not "expired"), and combined with checking if $E[\text{Extra\_IO}]$ (which is related to the access frequency) can be decreased, the adjustment overhead is limited as the adjustment frequency is limited, we also analyze the adjusting overhead in §4. On the other hand, we set `lifeTime` as the same order of magnitude as the total number of segments. The rationale is that if there is no "expired" segment, it means almost all the segments have been accessed recently, so they may have similar hotness during that time and we do not need to do the adjustment.

3.5 Implementation Issues

We implement ElasticBF on top of various commonly used KV stores, including LevelDB, RocksDB and PebblesDB. Here, we briefly describe the issues in the implementation.

ElasticBF keeps multiple filter units for each segment in each SSTable, to make minimum changes to SSTables, each filter unit is treated as a *meta block* in original SSTable organization, and the offset information in the file are recorded in *meta index block*. Besides, as generating multiple Bloom filters may add latency to writes, ElasticBF leverages multi-threading via *threadpool* to generate multiple filter units simultaneously so as to further reduce the computation time. On the other hand, ElasticBF maintains a background thread to manage Multi-Queue, so loading filter units and fetching data from secondary storage can be done in parallel, therefore the device bandwidth can be efficiently exploited.

4 Evaluation

In this section, we evaluate ElasticBF to validate its efficiency. We build ElasticBF atop LevelDB [21], RocksDB [16], and PebblesDB [32], and compare the performance of these systems with and without ElasticBF so as to study how much improvement ElasticBF can achieve. We point out that LevelDB is the classical LSM-tree based design, and RocksDB further improves the performance of LevelDB with multiple optimizations. Both of them are widely used as baselines for performance comparison [27, 32]. Besides, PebblesDB is developed based on the new and state-of-the-art index called fragmented LSM-tree, so we also take it as a baseline to demonstrate the effectiveness of ElasticBF. We emphasize that since ElasticBF is orthogonal to the works optimizing architecture of KV stores, it can also be integrated to other KV stores to further speedup their read performance. In the evaluation, we try to address the following questions.

- How much improvement does ElasticBF achieve to speedup the read performance of KV stores? (§4.2)
- How is the performance of ElasticBF under the workloads of YCSB benchmark? (§4.3)
- What is the performance impact of dynamically allocating bits to Bloom filters in ElasticBF, as compared to the static heterogeneous scheme in Monkey? (§4.4)
- What is the performance impact of different configurations on ElasticBF? (§4.5)

4.1 Experiment Setup

We run experiments on a Dell PowerEdge R730 with an 12-cores Intel Xeon CPU E5-2650 v4 with 2.20GHz processor, 64GB RAM, and Ubuntu 16.04 OS with Linux 4.15 kernel. The testbed is equipped with one 500GB SSD and one 1TB 7200RPM HDD. By default, we run experiments on the SSD. We build ElasticBF on top of LevelDB (v1.20), RocksDB (v5.14) and PebblesDB. As RocksDB and PebblesDB use 64MB or larger SSTables as their default configuration, we also set the SSTable size by modifying *max_file_size* to 64MB in LevelDB. To make a fair comparison, only the management strategy of Bloom filters was changed accordingly, while the memory usage is limited as the same and other parameters are also set as the same with the default values.

In the experiment, we use the benchmark YCSB-C [31, 34], which is the C++ version of YCSB [9] with low overhead. Unless specifically mentioned, we use the following default configuration. We set the size of each KV pair as 1KB, and load a 100GB database with randomly generated distinct keys. For the benchmarked workload, we generate 10M Get operations by following the Zipf distribution with a Zipfian constant 0.99 by default. Note that there is no warm up phase, i.e., we immediately issue the benchmarked workload to the randomly loaded database. We also point out that the performance is already stable after issuing 10M operations. By default, we assume that half of the Get operations request non-existent items (i.e., zero lookup), mainly because lookups of non-existent KV pairs are very common in practical systems [6, 23, 35, 38]. As many KV stores provide their own cache mechanisms, thus we enable *direct I/O* [19] to better manage memory. For the default setting, we disable the *block cache* [17] to minimize the influence of cache. This represents the scenario in which a KV store runs within a memory-constrained environment [11, 25], we also show the performance impact of block cache size in §4.5. For ElasticBF, the segment size of each SSTable is set as 4MB, and the `lifeTime` is set as 10K as there are around 30K segments in total. The average Bloom filter space for each key (i.e., `bits-per-key`) is set as four bits, this is because allocating a large number of bits for each key is not cost-efficient and it may be impractical in very large KV stores. Cassandra [2] also uses a similar setting of about 5 `bits-per-key` by configuring the Bloom filter to have the false positive rate of 0.1 in its *LeveledCompactionStrategy*. We also study the impact of different system configurations on the performance of ElasticBF in §4.5.

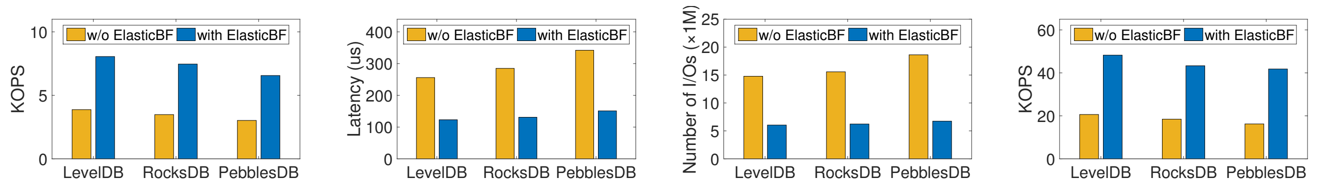
4.2 Micro-benchmarks

We first evaluate the performance of ElasticBF with micro-benchmarks. To evaluate the read performance, we consider both read-only workload and mixed workloads with different read/write ratios so as to validate the effectiveness of the hotness inheritance technique in ElasticBF. Finally, we also show the performance impact on writes and range queries.

Read-only workload. We use one thread to run the YCSB benchmark to perform 10M Get requests. Figure 10(a)-(c) show the results of read throughput, average read latency, and total number of I/Os. We can see that ElasticBF improves the read performance of different KV stores. Specifically, the read throughput with ElasticBF is increased to 2.08×, 2.15× and 2.17× compared to the results without ElasticBF under LevelDB, RocksDB and PebblesDB, respectively. For average read latency, ElasticBF can reduce the latency of LevelDB, RocksDB and PebblesDB by 51.9%, 54.0% and 55.8%, respectively. The improvement of ElasticBF is mainly because the reduction of extra I/Os caused by false positive of Bloom filters. To validate this, we also count the total number of I/Os generated to serve the Get requests, and the results are shown in Figure 10(c). We can see ElasticBF reduces the number of I/Os issued under different KV stores by 59.1% - 63.8%. As a result, ElasticBF can further improve the read performance of KV stores. To validate the effectiveness of the *expiring policy* and the adjusting rule, we count the total number of I/Os issued by loading filter units, it is only about 1% of the total number of I/Os generated to serve the Get requests, thus the adjusting overhead is small.

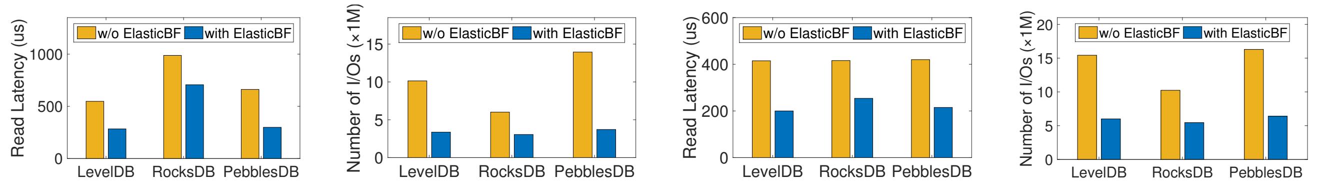
We further study the concurrent read performance of ElasticBF by using 16 threads to run the YCSB benchmark, and each thread performs 1M Get requests. Since we observe similar results for throughput, latency and total number of I/Os, we only show the throughput results in Figure 10(d) for the interest of space. In particular, ElasticBF increases the read throughput to 2.34× - 2.58× in these KV stores. Note that the improvement is slightly larger than that in single-threaded scenario, this is because multi-threaded reads can better utilize the I/O bandwidth.

Mixed workloads. Now we show the performance of ElasticBF under mixed workloads with different read/write ratios. The goal of this experiment is to validate that ElasticBF can still achieve a consistent improvement for reads due to the hotness inheritance design, even though compaction continuously generates new SSTables.



(a) Throughput (b) Latency (c) Total number of I/Os (d) Multi-threaded throughput

Figure 10: Read performance of KV stores with and without ElasticBF under read-only workload.

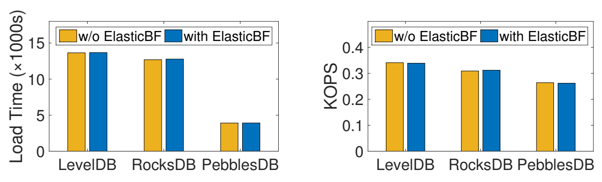


(a) Read latency (50% reads) (b) Number of I/Os (50% reads) (c) Read latency (90% reads) (d) Number of I/Os (90% reads)

Figure 11: Read performance of KV stores with and without ElasticBF under mixed workloads.

Figure 11 (a)-(b) show the results under the workload with 50% reads and 50% writes, and Figure 11 (c)-(d) show the results under the workload with 90% reads and 10% writes. Note that the total number of requests in the workload is 10M. We can see that ElasticBF can help reduce the read latency by 48.2%, 28.4%, and 54.8% for LevelDB, RocksDB and PebblesDB, respectively, under the workload with 50% reads, and the corresponding reduction ratios are 51.8%, 38.9% and 48.8% under the workload with 90% reads. We also count the total number of I/Os issued by Get operations. Specifically, ElasticBF reduces 66.8% (61.1%), 49.1% (46.7%), and 73.3% (60.7%), for LevelDB, RocksDB and PebblesDB, respectively, for workloads with 50% (90%) reads. Note that the reduction of read latency is smaller than that under read-only workload, the reason is that different KV stores use different compaction strategies, e.g., RocksDB enables multiple threads to do compaction and PebblesDB reduces compaction I/Os by avoiding rewriting SSTables to the same level, so the background compaction I/Os that are competed with the foreground Get I/Os are varied from KV stores. Note that the write performance does not decrease, and we will evaluate the write performance later.

Write and range query performance. Now we study the impact on write and range query performance. For different KV stores, we first randomly load a 100GB database and then issue 10M scan requests. We compare the time of loading the database and performing scan requests to evaluate the write and range query performance, and the results are shown in Figure 12. We can see that both the write and range query performance keep almost the same (the difference is less than 1%) even when ElasticBF is integrated in KV stores. The main reason is that Bloom filters are organized into blocks in SSTables, and ElasticBF also uses multi-threading to speedup the generation of Bloom filters. For range query, since it needs to fetch all blocks overlapped with the given range, Bloom filters are not involved in this procedure. Thus, ElasticBF has negligible impact on write and range query performance.



(a) Time to load the KV store (b) Throughput of range query

Figure 12: Put and Scan performance.

4.3 YCSB Benchmarks

Now we evaluate the performance of ElasticBF with YCSB benchmarks, which provide a set of six workloads with different combinations of KV operations. Specifically, Workload A consists of 50% reads and 50% updates, Workload B consists of 95% reads and 5% updates, Workload C consists of 100% reads, Workload D consists of 95% reads and 5% inserts, Workload E consists of 95% scans and 5% inserts, and Workload F consists of 50% reads and 50% read-modify-writes. Note that Workload D uses the Latest distribution [9], while others follow Zipfian distribution. Each of the six workloads consists of 10M operations, which are issued on a 100GB database, and other settings are the same as before.

We first compare the performance of LevelDB, RocksDB and PebblesDB with and without ElasticBF. Figure 13(a), 13(b) and 13(c) show the throughput results. We can see that ElasticBF improves the performance for all workloads except for Workload E (95% scans), this is because Workload E is a scan-dominated workload, and ElasticBF does not affect the performance of write and scan. In particular, for read-only Workload C, ElasticBF achieves 1.99× - 2.11× throughput in different KV stores due to the optimized Bloom filter design. For Workload A (50% reads) and Workload B (95% reads), ElasticBF improves the throughput by 7.4% - 36.8% and 52.6% - 71.5% for different KV stores, respectively. The reason why the improvement under Workload A and B is smaller than that under Workload C is because the request keys are Zipfian distributed, and the updates make most of

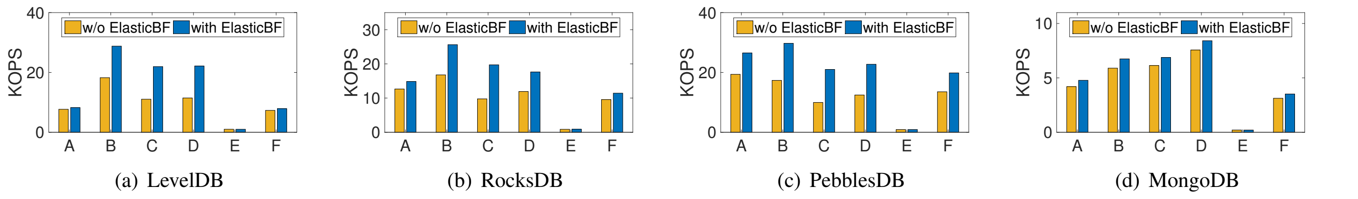


Figure 13: Performance comparison of different KV stores with and w/o ElasticBF under YCSB benchmarks.

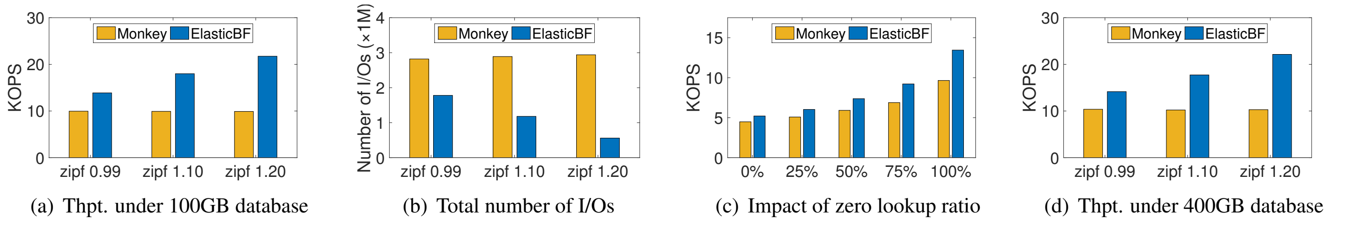


Figure 14: Performance compared with Monkey under micro-benchmarks.

the accessed keys issued by Get are stored in lower levels, and thus leads to a smaller number of SSTables that need to be checked during read than that in read-only Workload C, so ElasticBF has a smaller improvement. The above reason also leads to the results under Workload F which has 50% reads and 50% read-modify-writes, and ElasticBF improves the throughput by 7.8% - 46.6%. Finally, for Workload D, ElasticBF increases the throughput by 47.9% - 93.0%.

We also study the performance impact of ElasticBF on MongoDB [29], which is a popular open-source NoSQL database using WiredTiger [41] and RocksDB as its storage engine. Since WiredTiger is not based on LSM-tree, we choose RocksDB as the storage engine, and evaluate the performance improvement when integrate ElasticBF in MongoDB. The client of YCSB benchmark is running on the same machine with the MongoDB server. Figure 13(d) shows the results. We find that the improvements are only about 11% - 15% except for the scan-dominated Workload E. This is because the YCSB workloads issue only one read/update/insert per request, while MongoDB adds a lot of latency in the critical path of read operations, e.g., the query planner, and thus the latency induced by RocksDB accounts for only about 20% of the total latency. As a result, optimization in the KV storage layer does not result in a large improvement, and this is also observed in PebblesDB [32]. However, we point out that MongoDB may issue batch reads to hide extra latency in real-world scenarios, and in this case, ElasticBF must bring in larger improvement.

4.4 Comparison with Monkey

Now we compare the performance of ElasticBF with Monkey [10], both are built atop LevelDB. As Monkey mainly focuses on zero lookups, which are very common in practice, e.g., the insert-if-not-exist queries, we also assume that all Get operations request non-existent items in the experiments.

We first evaluate the performance of micro-benchmarks. We conduct the evaluation by issuing 10M Get requests to 100GB KV stores. Figure 14(a) shows the results. We can

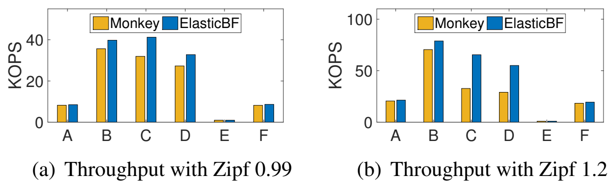
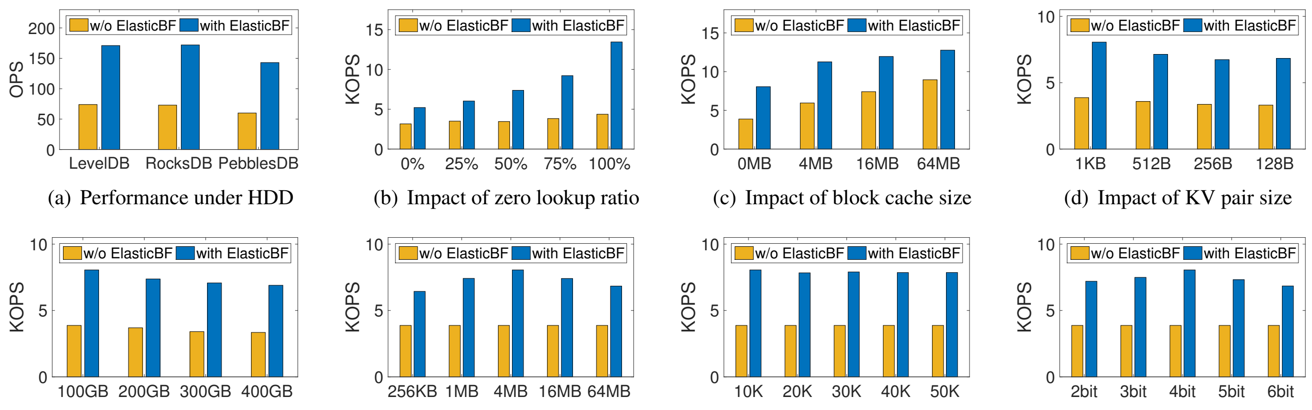


Figure 15: Compared with Monkey under YCSB benchmarks.

see that ElasticBF increases the throughput to 1.39× - 2.20× across different workloads. In particular, the improvement ratio increases if the workload is more skewed, and this validates the efficiency of taking into account data locality. However, the performance of Monkey is flat across workloads, the reason is that each zero lookup will traverse all levels of the LSM-tree regardless of the access skewness, thus the number of I/Os caused by each Get request due to false positive keeps almost the same under the static setting in Monkey. To further justify, we also count the total number of I/Os issued by Get operations. As shown in Figure 14(b), ElasticBF reduces the number of I/Os by 36.9% - 80.9% compared with Monkey. We then study the impact of zero lookup ratio, and Figure 14(c) shows the result by fixing the Zipfian constant as 0.99. ElasticBF improves the throughput to 1.16× - 1.39×. In particular, as the number of I/Os to find the relevant items accounts for a higher proportion when there are fewer zero lookups, ElasticBF results in smaller benefits in the case of lower zero lookup ratio. Finally, we also study the performance impact on 400GB KV stores, as shown in Figure 14(d), ElasticBF preserves similar improvement, e.g., it increases the throughput to 1.36× - 2.15×.

Next, we study the performance of ElasticBF under the YCSB benchmarks with four threads, each workload of YCSB issues 10M operations on a 100GB KV store. To consider the impact of access skewness, we conduct two sets of experiments by setting the Zipfian constant as 0.99 and 1.2, respectively. Figure 15(a) and Figure 15(b) show the results. We can see that ElasticBF outperforms Monkey for all read-dominated workloads. In particular, for Workload C (100%



(a) Performance under HDD (b) Impact of zero lookup ratio (c) Impact of block cache size (d) Impact of KV pair size

(e) Impact of database size (f) Impact of segment size (g) Impact of `lifeTime` (h) Impact of filter unit size

Figure 16: Read throughput under different system and parameter configurations.

reads) and Workload D (95% reads and 5% inserts), when the Zipfian constant is 0.99, ElasticBF obtains 1.29× and 1.20× throughput, respectively, and the improvement ratios increase to 1.99× and 1.89× when the workload becomes more skewed. For other workloads, such as Workload A, B and F, the improvement of ElasticBF is small, this is because the updates in these workloads make most of the keys requested by Get be at lower levels, and Monkey can gain the benefits even with a level-based static scheme, and thus the further improvement of ElasticBF is small.

ElasticBF performs better than Monkey mainly because of the finer-grained hotness identification strategy, i.e., the segment-level hotness identification and Bloom filter adjustment. We will further evaluate the impact of the granularity of hotness identification in details by varying the segment size in the next subsection. Besides the performance improvement, ElasticBF is also a more general Bloom filter management scheme, e.g., it can be deployed in KV stores which use complicated compaction strategies [15, 18], while in these cases, the LSM-tree structure becomes hard to predict, so it is hard for Monkey to find the optimal setting of Bloom filters, and Monkey also fails to dynamically adjust the setting when the structure of the LSM-tree changes.

4.5 Impact of Different Configurations

In this subsection, we study the performance impact of different configurations on ElasticBF. We use one thread to run YCSB benchmark, and issue 10M Get requests to a 100GB database. Due to page limit, we only show the performance improvement over LevelDB, and improvements over RocksDB and PebblesDB are also similar.

Performance under HDD. Figure 16(a) shows the read throughput under HDD. We can see that ElasticBF achieves 2.31× - 2.38× better throughput for LevelDB, RocksDB and PebblesDB. Besides, we also note that the improvement ratio is similar to that under SSD. That is, ElasticBF can improve the read performance of KV stores for all the main-stream storage devices, including both SSD and HDD.

Impact of zero lookup ratio. Figure 16(b) studies the impact of different ratios of non-existent items. Specifically, the x-axis represents the ratio of Get requests which access non-existent keys, i.e., zero lookup. Specifically, ElasticBF increases the read throughput to 1.65× - 3.08× when the zero lookup ratio increases from 0% to 100%. Besides, we can see that with the increasing of zero lookup ratio, ElasticBF achieves larger improvement. This is because looking up non-existent keys leads to the search of all levels in the KV store, and so it needs to check more SSTables. Thus, optimizing the Bloom filter can bring larger benefit to reduce the I/Os, so ElasticBF can get a larger improvement from zero lookups.

Impact of block cache size. We study the impact of cache size. As mentioned before, we enable direct I/O in the experiments. Figure 16(c) shows the results with different block cache sizes, note that the default block cache size is 8MB [17], and we set its size as 64MB which is slightly larger than the amount of Bloom filters (50MB). We can see that with a larger block cache, the read performance improves. Besides, ElasticBF still improves the performance when cache is enabled, e.g., it increases the throughput to 1.43× - 2.08×.

Impact of KV pair size and database size. Figure 16(d) studies the impact of KV pair size. ElasticBF increases the read throughput for different KV pair sizes, e.g., it increases the throughput to 1.99× - 2.08× when we vary the KV pair size from 1KB to 128B. Similarly, Figure 16(e) shows that ElasticBF consistently increases the read throughput for large databases, e.g., it increases the throughput to 2.00× - 2.08× when we vary the database size from 100GB to 400GB.

Impact of segment size, `lifeTime` length and filter unit size. Finally, we study the impact of configuration parameters on the read performance of ElasticBF. First, we consider the impact of segment size, and Figure 16(f) shows the read throughput versus the segment size. Note that the SSTable size is 64MB, if the segment size is also set as 64MB, then it means that we measure hotness and adjust Bloom filters in unit of an SSTable. The results show that the improvement is the largest under the 4MB setting, e.g., the throughput under the 4MB setting is 17.8% higher than that under the 64MB setting. The reason is that as the segment size decreases, ElasticBF can perform a finer-grained hotness recognition, so it can gain more benefits from the adjustment of Bloom filters. This also demonstrates the effectiveness of the finer-grained design in ElasticBF. However, if the size of each filter unit is too small (e.g., less than a block size), then each load of the filter unit is wasting for the I/Os, so the throughput drops. Second, from the results in Figure 16(g), which show the impact of `lifeTime` length, we can see that and the performance improvement has no big difference. That is, ElasticBF is not sensitive to the `lifeTime` parameter, e.g., we can simply set `lifeTime` according to the total number of segments. Finally, Figure 16(h) shows the impact of filter unit size, the x-axis represents the `bits-per-key` of each filter unit, and we configure the total `bits-per-key` of a filter group as 24 (or 25 for 5bit). The improvement is the largest when using 4bit. This is because if each filter unit uses fewer `bits-per-key`, then it can have more filter units for each SSTable, and this implies to have more hotness categories, but too many categories will also require more I/Os to load enough filter units to achieve low false positive rate.

To summarize this section, we find that ElasticBF can effectively boost the read performance of various KV stores under different storage mediums and database scales, but it mainly focuses on the memory-constrained environment. That is, if the memory capacity is not a bottleneck, then one can simply allocate more bits to each Bloom filter and keeps all of them in memory, in this case, the false positive rate can be very small and the benefit of ElasticBF is limited.

5 Related Work

In recent years, many studies have proposed new designs based on LSM-tree [30]. WiscKey [27] reduces the compaction I/Os by using key-value separation technique to manage keys and metadata in LSM-tree, while stores values into an appended-only log. HashKV [7] further optimizes the value management for key-value separation based design by using hash-based data organization. LSM-trie [42] focuses on small key value pairs, and organizes data into a hash-based trie structure to reduce write amplification. bLSM [35] uses a new merge scheduler to reduce the impact of the compaction on the front-end write performance, and also uses Bloom filters to help efficient lookup. TRIAD [5] reduces write I/Os by leveraging the skewed data popularity and delayed compaction strategy. PebblesDB [32] reorganizes the storage layout inspired from skip lists, thereby avoiding data rewriting in the same level to reduce the compaction overhead. We point out that these works mainly focus on improving the write performance, and they still follow the basic structure of LSM-tree and require Bloom filter, so our work is orthogonal to them, and can be used to further improve the read performance by adaptively adjusting Bloom filters.

Some other studies aim to better utilize the features of emerging storage devices to improve the performance of KV stores. For example, RocksDB [16] utilizes the parallelism of SSDs by scheduling multiple compaction operations concurrently. LOCS [40] leverages the multi-channel of SSDs to exploit the abundant parallelism for efficient compaction and data access. NVMKV [28] cooperates with FTL by mapping KV pairs in physical address space to decrease the redundant work between the store layer and the device layer. HiKV [43] also leverages NVRAM by using a hybrid index. In contract, ElasticBF mainly focuses on the Bloom filter design, and it can improve the read performance on both HDDs and SSDs.

At last, there are also several works considering Bloom filter optimization. In particular, RocksDB [15] uses prefix Bloom filter to reduce read amplification on range queries. SuRF [44] is based on a succinct data structure to reduce I/Os by filtering requests of point queries and range queries. Heterogeneous Bloom filter design is also considered to configure different Bloom filters for different levels or files [10, 45]. However, Monkey [10] adopts a coarse-grained scheme which allocates the same number of filters for SSTables within the same level and also fails to dynamically adjust according to hotness. ElasticBF further leverages the access locality in a finer granularity with dynamical adjustment, and our extensive experiments demonstrate its benefit, especially for skewed workloads. Finally, compared with our previously published workshop paper [45], we also make multiple novel optimizations: (1) we develop a fine-grained heterogeneous scheme by further differentiating segments within each SSTable, (2) we propose a hotness inheritance scheme to quickly obtain the accurate hotness information of the newly generated SSTables during compaction, (3) we implement ElasticBF on top of various KV stores and conduct extensive experiments to demonstrate its efficiency and generality, and (4) we also leverage multi-threading and parallel I/Os in the implementation for performance optimization.

6 Conclusion

In this paper, we developed a fine-grained heterogeneous Bloom filter management scheme called ElasticBF by leveraging the access skewness within workloads. ElasticBF measures the hotness information with a lightweight method and also supports dynamic adjustment of Bloom filters at a fine granularity. As a result, ElasticBF can greatly reduce the expected overall false positive rate without increasing the volume and memory overhead of Bloom filters, and thus speeds up the read performance in KV stores. Finally, we also conducted extensive experiments to demonstrate the efficiency of ElasticBF by building it atop of various KV stores.

Acknowledgements

The work was supported by National Key R&D Program of China under Grant No. 2018YFB1003204, National Nature Science Foundation of China (61772484, 61832011, and 61772486).

References

[1] Apache. Cassandra. http://cassandra.apache.org/.

[2] Apache. Tuning Bloom filters. http://cassandra.apache.org/doc/4.0/operating/bloom_filters.html, 2018.

[3] Timothy G Armstrong, Vamsi Ponnekanti, Dhruba Borthakur, and Mark Callaghan. LinkBench: A Database Benchmark based on the Facebook Social Graph. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. ACM, 2013.

[4] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload Analysis of a Large-Scale Key-Value Store. In *ACM SIGMETRICS Performance Evaluation Review*, volume 40, pages 53–64. ACM, 2012.

[5] Oana Maria Balmau, Diego Didona, Rachid Guerraoui, Willy Zwaenepoel, Huapeng Yuan, Aashray Arora, Karan Gupta, and Pavan Konka. TRIAD: Creating Synergies Between Memory, Disk and Log in Log Structured Key-Value Stores. In *USENIX ATC 17*, 2017.

[6] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry C Li, et al. TAO: Facebook's Distributed Data Store for the Social Graph. In *USENIX Annual Technical Conference*, pages 49–60, 2013.

[7] Helen HW Chan, Yongkun Li, Patrick PC Lee, and Yinlong Xu. HashKV: Enabling Efficient Updates in KV Storage via Hashing. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. USENIX Association, 2018.

[8] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. Bigtable: A Distributed Storage System for Structured Data. *ACM Transactions on Computer Systems (TOCS)*, 26(2):4, 2008.

[9] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 143–154. ACM, 2010.

[10] Niv Dayan, Manos Athanassoulis, and Stratos Idreos. Monkey: Optimal Navigable Key-Value Store. In *Proceedings of the 2017 ACM International Conference on Management of Data*. ACM, 2017.

[11] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[12] Biplob Debnath, Sudipta Sengupta, and Jin Li. FlashStore: High Throughput Persistent Key-Value Store. *Proceedings of the VLDB Endowment*, 3(1-2):1414–1425, 2010.

[13] Biplob Debnath, Sudipta Sengupta, and Jin Li. SkimpyStash: RAM Space Skimpy Key-Value Store on Flash-based Storage. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, pages 25–36. ACM, 2011.

[14] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's Highly Available Key-Value Store. In *ACM SIGOPS operating systems review*. ACM, 2007.

[15] Siying Dong, Mark Callaghan, Leonidas Galanis, Dhruba Borthakur, and Tony Savor. Optimizing Space Amplification in RocksDB.

[16] Facebook. RocksDB. http://rocksdb.org/.

[17] Facebook. Block Cache. https://github.com/facebook/rocksdb/wiki/Block-Cache, 2017.

[18] Facebook. Universal Compaction. https://github.com/facebook/rocksdb/wiki/Universal-Compaction, 2017.

[19] Facebook. Direct IO. https://github.com/facebook/rocksdb/wiki/Direct-IO, 2018.

[20] Peter Frühwirt, Marcus Huber, Martin Mulazzani, and Edgar R Weippl. InnoDB Database Forensics. In *2010 24th IEEE International Conference on Advanced Information Networking and Applications*. IEEE, 2010.

[21] Sanjay Ghemawat and Jeff Dean. LevelDB. https://github.com/google/leveldb, 2011.

[22] Tyler Harter, Dhruba Borthakur, Siying Dong, Amitanand S Aiyer, Liyin Tang, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. Analysis of HDFS under HBase: A Facebook Messages Case Study. In *FAST*, 2014.

[23] Yangwook Kang, Rekha Pitchumani, Thomas Marlette, and Ethan L Miller. Muninn: A Versioning Flash Key-Value Store Using an Object-based Storage Model. In *Proceedings of International Conference on Systems and Storage*, pages 1–11. ACM, 2014.

[24] Adam Kirsch and Michael Mitzenmacher. Less Hashing, Same Performance: Building a Better Bloom Filter. In *ESA*, volume 6, pages 456–467. Springer, 2006.

[25] Chunbo Lai, Song Jiang, Liqiong Yang, Shiding Lin, Guangyu Sun, Zhenyu Hou, Can Cui, and Jason Cong. Atlas: Baidu's Key-Value Storage System for Cloud Data. In *Mass Storage Systems and Technologies*

(MSST), 2015 31st Symposium on, pages 1–14. IEEE, 2015.

[26] Guanlin Lu, Young Jin Nam, and David HC Du. BloomStore: Bloom-Filter based Memory-efficient Key-Value Store for Indexing of Data Deduplication on Flash. In *Mass Storage Systems and Technologies (MSST), 2012 IEEE 28th Symposium on*, pages 1–11. IEEE, 2012.

[27] Lanyue Lu, Thanumalayan Sankaranarayana Pillai, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. WiscKey: Separating Keys from Values in SSD-Conscious Storage. In *FAST*, pages 133–148, 2016.

[28] Leonardo Marmol, Swaminathan Sundararaman, Nisha Talagala, Raju Rangaswami, Sushma Devendrappa, Bharath Ramsundar, and Sriram Ganesan. NVMKV: A Scalable and Lightweight Flash Aware Key-Value Store. In *HotStorage*, 2014.

[29] MongoDB. MongoDB. `https://www.mongodb.com/`.

[30] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. The Log-Structured Merge-Tree (LSM-tree). *Acta Informatica*, 33(4):351–385, 1996.

[31] Anastasios Papagiannis, Giorgos Saloustros, Pilar González-Férez, and Angelos Bilas. Tucana: Design and Implementation of a Fast and Efficient Scale-up Key-value Store. In *USENIX ATC*, pages 537–550, 2016.

[32] Pandian Raju, Rohan Kadekodi, Vijay Chidambaram, and Ittai Abraham. PebblesDB: Building Key-Value Stores using Fragmented Log-Structured Merge Trees. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 497–514. ACM, 2017.

[33] Luiz E Ramos, Eugene Gorbatov, and Ricardo Bianchini. Page Placement in Hybrid Memory Systems. In *Proceedings of the international conference on Supercomputing*, pages 85–95. ACM, 2011.

[34] J. REN. YCSB-C. `https://github.com/basicthinker/YCSB-C`, 2015.

[35] Russell Sears and Raghu Ramakrishnan. bLSM: A General Purpose Log Structured Merge Tree. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*. ACM, 2012.

[36] Pradeep Shetty, Richard P Spillane, Ravikant Malpani, Binesh Andrews, Justin Seyster, and Erez Zadok. Building Workload-Independent Storage with VT-Trees. In *FAST*, pages 17–30, 2013.

[37] Roshan Sumbaly, Jay Kreps, Lei Gao, Alex Feinberg, Chinmay Soman, and Sam Shah. Serving Large-scale Batch Computed Data with Project Voldemort. In *Proceedings of the 10th USENIX conference on File and Storage Technologies*, pages 18–18. USENIX Association, 2012.

[38] Guido Urdaneta, Guillaume Pierre, and Maarten Van Steen. Wikipedia Workload Analysis for Decentralized Hosting. *Computer Networks*, 53(11):1830–1845, 2009.

[39] Hoang Tam Vo, Sheng Wang, Divyakant Agrawal, Gang Chen, and Beng Chin Ooi. LogBase: A Scalable Log-structured Database System in the Cloud. *Proceedings of the VLDB Endowment*, 5(10):1004–1015, 2012.

[40] Peng Wang, Guangyu Sun, Song Jiang, Jian Ouyang, Shiding Lin, Chen Zhang, and Jason Cong. An Efficient Design and Implementation of LSM-tree based Key-Value Store on Open-Channel SSD. In *Proceedings of the Ninth European Conference on Computer Systems*, page 16. ACM, 2014.

[41] WiredTiger. WiredTiger. `http://www.wiredtiger.com/`.

[42] Xingbo Wu, Yuehai Xu, Zili Shao, and Song Jiang. LSM-trie: An LSM-tree-based Ultra-Large Key-Value Store for Small Data. In *USENIX ATC 15*, pages 71–82. USENIX Association, 2015.

[43] Fei Xia, Dejun Jiang, Jin Xiong, and Ninghui Sun. HiKV: A Hybrid Index Key-Value Store for DRAM-NVM Memory Systems. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 349–362, 2017.

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

[45] Yueming Zhang, Yongkun Li, Fan Guo, Cheng Li, and Yinlong Xu. ElasticBF: Fine-grained and Elastic Bloom Filter Towards Efficient Read for LSM-tree-based KV Stores. In *USENIX HotStorage 18*, 2018.

[46] Yuanyuan Zhou, James Philbin, and Kai Li. The Multi-Queue Replacement Algorithm for Second Level Buffer Caches. In *USENIX Annual Technical Conference, General Track*, pages 91–104, 2001.