SHaMBa: Reducing Bloom Filter Overhead in LSM Trees

Zichen Zhu

supervised by Prof. Manos Athanassoulis
Boston University, MA, USA

Abstract

Bloom Filiters (BFs) are typically employed to alleviate unnecessary disk accesses to faciliate point lookup in LSM trees. They are particularly beneficial when there is a significant performance difference between probing a Bloom filter (hashing and accessing memory) and accessing data (on secondary storage). However, this gap is decreasing as SSDs and NVMs have increasingly lower latency, to the point that the cost of *accessing data* can be comparable to that of *hashing and filter probing*, especially for large key sizes that results in high hashing cost. In addition, BFs are beneficial for empty queries while they are a burden for positive queries (i.e., on an existing key). Also, with larger datasets, the total memory consumed by also increases making it less feasible to keep in memory all BFs. Coupling this, with the increasing price of memory and the need to reduce the memory-to-data ratio in many practical deployments, we are seeing an increased memory pressure. In this setting, fewer BF blocks are cached, thus causing additional storage accesses, since they have to be fetched in memory to answer a query.

In this PhD work, we introduce **SHaMBa** (**S**hared **Ha**sh **M**odular **B**loom Filter), a new LSM-based key-value engine that addresses both (a) the increasing hashing overhead and (b) the suboptimal performance when BFs do not fit in memory. First, SHaMBa decouples the hashing cost from the data size by sharing a single hash digest across different levels. Second, SHaMBa applies a workload-aware BF skipping policy based on Modular Bloom Filter (i.e., a set of mini-BFs that replace a single large BF) to avoid accessing BFs when they are not useful. Our evaluation shows that SHaMBa reduces the CPU cost for BF probing, and substantially outperforms the state of the art under memory pressure, having the same average number of I/Os - using only *one-third of the memory consumption of the state of the art.*

Keywords

LSM trees, Bloom Filter, compaction policy, privacy

1. Introduction

LSM-based Key-Value Stores. Log-Structured Merge-trees (LSM-trees) [1] are widely adopted as one of the core data structures in modern NoSQL storage engines including LevelDB [2], RocksDB [3], and WiredTiger [4]. This is because LSM-trees offer high write throughput by employing *out-of-place* ingestion and also achieve good space utilization via an immutable file structure. In LSM-trees, incoming entries (inserts, updates, and deletes) are buffered within main memory. Once the write buffer becomes full, the contained entries are sorted and flushed to disk as a *sorted run*. The disk-resident sorted runs are organized into a number of levels of increasing sizes. In practice, a sorted run may consist of one or more *immutable Sorted-String Tables* (or *SST* files). To bound the number of files that a point lookup needs to probe, runs of similar sizes in the same level are sort-merged and pushed to the next (deeper) level when the accumulated bytes of similarly sized runs reach a predefined capacity. To avoid unnecessary accesses for point lookups, LSM-trees typically construct a Bloom Filter (BF) for every file that probabilistically allows to skip a file if it does not contain the target key. In addition, every file is also associated with an index block (aka fence pointers) that maintains the min-max range and the offset for each data block (or disk page), which ensures that at most one data block (or disk page) is retrieved when probing a file.

Problem 1: The Benefit of BFs Shrinks When Faster Storage is Used. Contrary to common perception, however, BFs are not always beneficial. The rationale behind the ubiquitous use of BFs in LSM-trees is that there is a considerable cost difference between accessing a BF (in memory) and accessing data (on disk). As the gap in access latency between BFs and data narrows, the advantages of using BFs weaken. If the data is already cached in main memory, BFs are detrimental. Further, as new storage devices like SSDs and non-volatile memories (NVMs) emerge, the latency gap between memory and storage narrows. experiments show that MurmurHash64 calculation (used in production system [3]) is $\sim$1.47$\times$ more expensive than accessing a memory page, thereby, making the use of a BF detrimental. The LSM hashing overhead is further exacerbated as multiple BFs are queried per lookup (at least one per level), and repeated hash calculations turn querying over fast storage (or cached data) into a CPU-intensive operation.

Problem 2: Read Performance in LSM-trees Degrades With Limited Memory. The memory allocated

VLDB 2023 PhD Workshop, co-located with the 49th International Conference on Very Large Data Bases (VLDB 2023), August 28, 2023, Vancouver, Canada
zczhu@bu.edu (Z. Zhu)
https://cs-people.bu.edu/zczhu/ (Z. Zhu)
0000-0002-9197-4649 (Z. Zhu)

to an LSM-based system can be divided into several components: (i) the *block cache* for caching BF blocks, index blocks, and data blocks, (ii) the *memory buffer* for storing incoming data, (iii) temporary memory to *compact data*, and (iv) temporary memory to support *ongoing range queries*. While computing, memory, and storage prices decrease and allow us to facilitate more data, in the last few years, the price drop in memory has been slower than what has been for computing and storage, making it hard to maintain the same memory-to-data ratio. For example, since 2010, the price of SSDs has decreased by a factor of sixty, whereas the price of memory has only decreased by a factor of ten [5]. As a result, BFs may not always be in memory when competing for the resource with index blocks and data blocks, and which there is a cache miss for BFs, a significant number of I/Os may be spent on fetching them.

SHaMBa: Less Hashing on Modular Bloom Filters. We propose *Shared Hashing* and *skipping-based Modular Bloom Filters* - two techniques that reduce the BF overhead in LSM trees and we integrate them into our system, SHaMBa. Specifically, we first propose a *shared hashing* technique [6] that shares a single hash digest across all levels in an LSM tree in order to alleviate the unnecessary cost of re-hashing for every level. Shared hashing decouples the aggregated hashing cost from data size; regardless of the number of LSM tree levels (which depends on the data size, the size ratio, and the memory buffer size), hashing cost is constant. We then identify that not all BFs are equally important, and based on this observation, we propose a skipping mechanism [7] based on Modular Bloom Filter (MBF) that allows us to load part of the filter to alleviate the memory pressure. Our evaluation shows that hash sharing can lead to 20% higher lookup performance in a state-of-the-art PCIe SSD and the skipping mechanism in MBF can achieve the same read throughput with only *one-third of the memory consumed by state-of-the-art*.

Contributions. Our contributions are as follows:

- We identify that BFs dominate the LSM query latency for *fast storage and high hashing cost*, and we decouple the amount of hashing from the data size (height of an LSM tree) by *shared hashing* across different levels.
- We propose *a skipping mechanism based on Modular Bloom Filter (MBF)* that reduces the memory footprint without sacrificing performance.
- We integrate Shared Hashing and Modular Bloom Filters in the state-of-the-art LSM-engine RocksDB, and we show through extensive experimentation with realistic workloads that our proposed techniques reduce the hashing cost, and outperform the state of the art under memory pressure.

2. Shared Hashing

Classical BFs rely on k independent hash functions to generate k indexes, which results in high CPU overhead. Practical implementation uses a single hash digest and generates $k - 1$ indexes by bit rotation. Such an optimization reduces the CPU cost by a factor of k, we now apply hash sharing across multiple LSM levels.

Hash Sharing Across Levels. The key observation is that for a specific query, the same hash digest calculation is repeated across levels. The BFs are different across levels (they have indexed different elements). However, calculating the hash digest is repeated for every queried level until finding the matching key or the tree is entirely searched. Thus, to mitigate this overhead, we *share the hash digest calculation* across levels by re-engineering the BF implementation and allowing the BFs residing in different levels to work in concert during the course of a single query (Fig. 1). As a result, the hashing cost stays constant regardless of the number of levels.

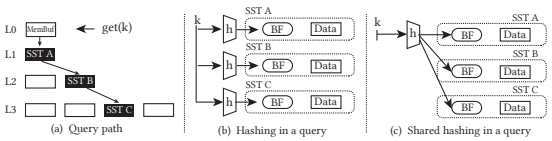
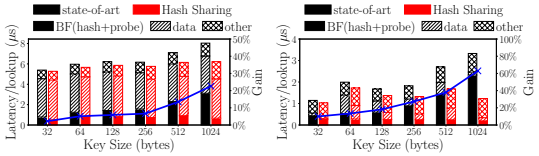


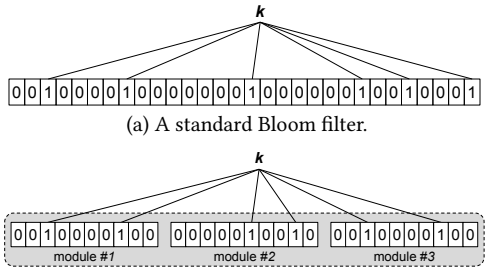
Figure 1: Hash sharing across BFs of different levels.

Evaluation. We build an in-house LSM-tree prototype, which uses RocksDB's fast local Bloom Filter and MurmurHash64. We bulk load our LSM tree with 22GB of key-value pairs (entry size is fixed as 2KB), and report the latency of empty point queries. The experiments are running with a state-of-the-art PCIe SSD that offers 10μs latency for 4KB page access. As shown in Figure 2a, the hashing cost increases for both approaches as the key size grows, however, shared hashing has a performance gain of up to 23% (blue line). The time breakdown shows where this benefit is coming from. The time spent in BFs (both hashing and probing) is drastically reduced for the hash sharing approach, while the cost for accessing data, as well as the other costs (e.g., binary search in fence pointers), virtually remains the same. In addition, larger key sizes have higher hashing cost, hence, hash sharing is more beneficial for them. Besides, when the query workload becomes skewed, we further observe the gain steeply increases for 1KB keys to more than 60% in Figure 2b. This is because the skewed workload has fewer data accesses due to fewer false positives, and as a result, hashing becomes a bottleneck for skewed point queries. More experiments can be found in our full paper [6].



(a) Uniform (b) Zipfian

Figure 2: Hash sharing reduces hashing overhead. The reduction is more pronounced for larger keys and skew workload.



(a) A standard Bloom filter.

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

Figure 3: Modular Bloom filters split the physical representation of a BF into multiple independent modules.

3. A Skipping Mechanism for MBF

In this section, we discuss how we achieve lower memory footprint with our skipping mechanism [7] for MBFs, and we show that, under memory pressure, our skipping algorithm achieves the same read throughput with only one-third of the memory used by the state of the art.

Modular Bloom Filters (MBFs). We first present Modular Bloom filters (MBFs) that divide a normal Bloom Filter into multiple modules. This division allows us to design workload-aware access methods by leveraging the tradeoff between the filter memory and the associated accuracy. Dividing a Bloom filter into smaller chunks is not a new technique, which can be also found in ElasticBF [8]. A Modular Bloom filter (MBF) uses m bits to index n elements in each of D modules. Each module uses m_d bits such that $\sum_{d=1}^{D} m_d = m$. Essentially, an MBF is a collection of D Bloom filters, and every membership test has to go through all modules before it concludes with a positive result if all modules have to be used. On the other hand, a negative response at any module terminates the query without the need to further continue probing the remaining modules. Figure 3 compares a Modular Bloom filter, which is composed of three modules, with a standard BF. By design, a point lookup can use all or any of the available modules without re-indexing. Thus, MBF can maintain only the selected modules in fast memory, leading to smaller memory consumption at the expense of a higher false positive rate (without needing to recalculate the filter).

A Skipping Mechanism. To fully exploit the multiple modules of MBFs, we quantify the *utility* of each module and design a new module skipping mechanism. We define the *utility* as follows.

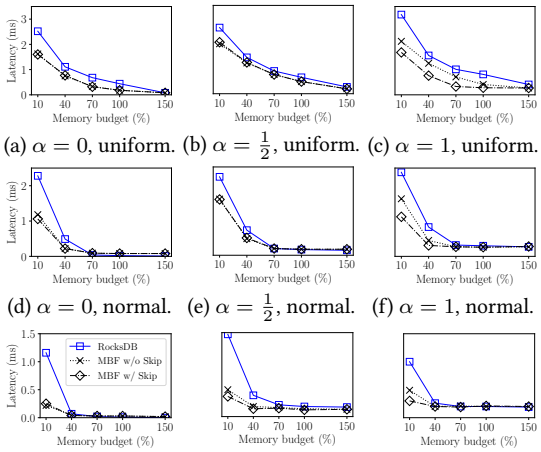
```
QueryMBF (key k, SST_{l,i})
    for d = 1, d ≤ number of modules, d++ do
        //calc module's utility
        u_{l,i,d} = β_{l,i} · (1 − α_{l,i}) · (·f_sm^d − f_sm^{d−1})
        if skip_d ==true & u_{l,i,d} < threshold_d then
            // skipping module, assumes that it returns
              positive
            return true;
        else
            // probe the module like a mini BF
            // this part might cause an I/O if the module is
              not cached
            result = QueryModule(k, module_{l,i,d})
        end
        if result==false then
            return false;
        end
    end
    return result;
end
```

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

$$u_{l,i,d} = \beta_{l,i} \cdot (1 - \alpha_{l,i}) \cdot \left(f_{sm}^{d-1} - f_{sm}^{d}\right) \qquad (1)$$

where $\beta_{l,i}$ ($\alpha_{,i}$) represents the access frequency point lookup (the ratio of true positive point queries, respectively) of the i^{th} file $SST_{l,i}$ at level l and f_{sm}^{d} is the false positive rate when using the first d modules. Next, we propose to skip probing modules if the expected number of I/Os is over a certain threshold (threshold$_d$). Algorithm 1 shows how to query an MBF and how to skip modules using their utility. The core idea of the algorithm is that if a module is expected to lead to an I/O anyway (combining the frequency of the accesses, and the frequency of queries being non-empty on the specific SST file), the system will prefer to go directly to the data since the I/O is inevitable (if we refer to the last module). As the utility of modules in different orders is not comparable, we allow the algorithm to use a different threshold per module slot. Moreover, it is also possible to skip only a subset of the modules since their utility decreases as more modules are accessed. As shown in Algorithm 1, the decision to skip is made per module $skip_d$.

Evaluation. We integrate our skipping algorithm and MBF into RocksDB with two equal-size modules (with thresholds, respectively, threshold$_1$ $=$ 0.02 and threshold$_2$ $=$ 0.01) to showcase the benefit of our skipping mechanism. We stress-test our approach with different workloads: 1) vary the point query patterns to follow the (a)uniform, (b) normal, or (c) Zipfian distribution; 2) vary the proportion of existing point lookups (α). We



(a) $\alpha = 0$, uniform. (b) $\alpha = \frac{1}{2}$, uniform. (c) $\alpha = 1$, uniform.

(d) $\alpha = 0$, normal. (e) $\alpha = \frac{1}{2}$, normal. (f) $\alpha = 1$, normal.

(g) $\alpha = 0$, Zipfian. (h) $\alpha = \frac{1}{2}$, Zipfian. (i) $\alpha = 1$, Zipfian.

Figure 4: Our skipping algorithm reduces the lookup latency of RocksDB under memory pressure.

report the average latency per lookup, as shown in Figure 4. The experimental results show that our skipping algorithm effectively reduces the lookup latency when there is memory pressure, and the benefits persist for both empty ($\alpha = 0.0$) and non-empty queries ($\alpha = 1.0$). More results can be found in our full paper [7].

4. Research Plan

State-of-the-art LSM trees employ a static memory allocation paradigm across levels and across files, which leads to all files having BFs with the same bits-per-key (BPK). Notably, Monkey [9] proposed to allocate more BPK to shallower (smaller) levels which proves to lead to minimal read cost assuming that the workload is uniform. Our main goal moving forward is to use detailed information on the access patterns of the workload to decide the exact BPK at the file and the module level.

[Short-Term] Dynamic BPK Allocation. We are working towards a dynamic BPK allocation strategy for all the BFs in LSM trees, which allows different files to have different BPK. The decision of the BPK per file is implemented at *compaction time*. Unlike prior work that assumes a predefined static workload [9], we will collect statistics of read access patterns (empty and non-empty queries per level) which will allow us to build an accurate cost function at runtime and find out the best local BPK allocation strategy during compaction, thus generalizing prior approaches. Our earlier work [10] has shown that the average compaction latency is mostly affected by moving data, with the creation of BFs at compaction time being a low-overhead process. In other words, we can implement a better BPK re-allocation without any visible increase in compaction latency.

[Long-Term] Holistic Memory Tuning. We are targeting a set of holistic memory tuning algorithms that can navigate the entire designing space of MBF under limited memory. If we allow each module to have different BPK and each file to have different number of modules, a richer designing continuum is constructed for MBF. We explain why the new designing space of MBF can benefit point queries from the following two perspectives: (a) By allowing each module to have a different BPK without changing the total BPK assigned per file, files with more empty point lookups can have more BPK for their in-memory modules while fewer BPK are assigned for the in-memory modules of other files. In this way, more empty point lookups can be blocked by the first module with a higher BPK due to a lower false positive rate. (b) However, note that the above design requires compaction to allocate BPK per module or per file. If we allow files to have three or more modules, we have more flexibility to decide how many modules are needed to access on-the-fly, without needing to re-construct the entire MBF. However, this design sacrifices the read performance of existing point queries since multiple BF queries are required when using several modules in MBF. Our goal is to create a workload-aware solution that can leverage the above trade-off and navigate the designing space of MBF to achieve minimum point query cost for LSM trees.

5. Conclusion

In this PhD work, we propose SHaMBa, a novel LSM-based key-value engine that addresses two key challenges. First, the fact that as we move to faster storage devices, hashing for BFs in LSM-Trees becomes the main bottleneck, and, second, the fact that the benefit from using BFs diminishes when the system is under memory pressure. Our evaluation shows that SHaMBa can reduce the fraction of time spent on hashing during lookups, and it can also exploit the available memory to offer better performance than the state of the art under memory pressure. The long-term goal of this PhD work is to introduce hardware/workload-aware BF management policy to facilitate point queries in write-optimized LSM trees.

References

[1] P. E. O'Neil, et al., The log-structured merge-tree (LSM-tree), Acta Informatica 33 (1996) 351–385.
[2] Google, LevelDB, https://github.com/google/leveldb/ (2021).
[3] Facebook, RocksDB, https://github.com/facebook/rocksdb (2021).
[4] WiredTiger, Source Code, https://github.com/wiredtiger/wiredtiger (2021).
[5] J. C. McCallum, Historical Cost of Computer Memory and Storage, https://jcmit.net/mem2015.htm (2022).
[6] Z. Zhu, et al., Reducing Bloom Filter CPU Overhead in LSM-Trees on Modern Storage Devices, DAMON (2021).
[7] J. H. Mun, et al., LSM-Tree Under (Memory) Pressure, ADMS (2022).
[8] Y. Zhang, et al., ElasticBF: Fine-grained and Elastic Bloom Filter Towards Efficient Read for LSM-tree-based KV Stores, HotStorage (2018).
[9] N. Dayan, et al., Monkey: Optimal Navigable Key-Value Store, SIGMOD (2017).
[10] S. Sarkar, et al., Constructing and Analyzing the LSM Compaction Design Space, PVLDB 14 (2021) 2216–2229.