SHaMBa: Reducing Bloom Filter Overhead in LSM Trees

Zichen Zhu

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

Abstract

Bloom Filters (BFs) are typically employed to alleviate unnecessary disk accesses to facilitate 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 existing keys). Also, with larger datasets, the total consumed memory also increases, making it less feasible to keep all BFs in memory. 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 sub-optimal 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.

Keywords

LSM trees, Bloom Filter, compaction policy, storage, memory pressure

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. 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 built with an index block (also termed 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 For Faster Storage. Contrary to common perception, 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 new storage devices like SSDs and non-volatile memories (NVMs) emerge, the latency gap between memory and storage narrows, and thus the advantages of using BFs weaken. If the data is already cached in main memory, BFs are even detrimental. Experiments show that MurmurHash64 calculation (used in production systems [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. 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

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)

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 when 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. To address these challenges, 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 the levels in an LSM tree to alleviate the unnecessary cost of re-hashing for every level. Shared hashing decouples the aggregated hashing cost from data size. With SHaMBa, regardless of the number of LSM tree levels (which depends on the data size, the size ratio, and the memory buffer size), the hashing cost is constant. We then identify that not all the BFs are equally important, and based on this observation, we propose a skipping mechanism [7] based on the Modular Bloom Filter (MBF) design 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 when using a state-of-the-art PCIe SSD, and the skipping mechanism in MBF increases read throughput by more than 50% compared with the state of the art when memory is constrained to 10% of the total size of Bloom Filters.

Contributions. Our contributions are as follows:

- We identify that BFs dominate LSM query latency for *fast storage and high hashing cost*, and we decouple the amount of hashing from data size (height of 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 experiments 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, which results in high CPU overhead when probing a BF. Practical implementations [3] use a single hash digest and generates $k - 1$ indexes by bit rotation. This optimization is based on the double hashing scheme [8], for which it is shown that it can achieve nearly the same accuracy obtained by independent k hash functions. Such

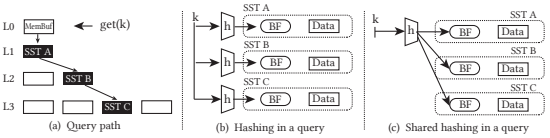


Figure 1: Hash sharing across BFs of different levels.

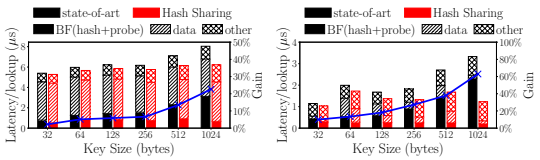


Figure 2: Hash sharing reduces hashing overhead. The reduction is more pronounced for larger keys and skew workload.

an optimization reduces the CPU cost by a factor of k when probing a single BF. Here, we apply a form of hash sharing across multiple BFs from different LSM levels.

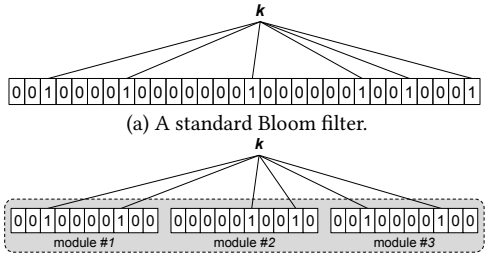
Hash Sharing Across Levels. The key observation is that for a specific query, the same hash digest calculation is repeated across levels when fence pointers cannot filter the query. The BFs are different across levels (they have indexed different elements), however, in order to probe them for a given key, the same hash digest is calculated for every level until the key is found or the tree is entirely searched. 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 collaboratively during the execution of a single query (Fig. 1). As a result, the hashing cost remains constant regardless of the number of 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].

3. Skipping-Based Modular BFs

In this section, we discuss how we achieve a lower memory footprint with our skipping mechanism [7] for MBFs, and we show that, under memory pressure, our skipping algorithm achieves higher read throughput compared to 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. MBF is a generalized version of ElasticBF [9] since MBF allows the size of each module to be different. 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 tries to sequentially go through all the modules before it concludes with a positive result. A negative response at any module terminates the query without further probing the remaining modules.



(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.

Figure 3 compares an MBF (using three modules) with a standard BF. By design, a point lookup can use all or any subset of the D modules without re-indexing. Thus, MBF can navigate the tradeoff between accuracy and memory footprint without re-calculating the filter.

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

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

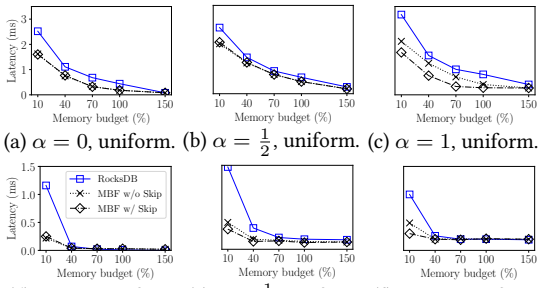
where $\beta_{l,i}$ ($\alpha_{l,i}$) represents the point lookup frequency (the ratio of true positive point queries, respectively) of the i^{th} file at level l, $SST_{l,i}$, and f_{sm}^{d} is the false positive rate when using the first d modules. By definition, $u_{l,i,d}$ quantifies how many I/Os (on average) can be avoided using the d^{th} module. Based on the utility, we propose to skip probing modules if the utility is lower than a certain threshold (threshold_d). Algorithm 1 shows how to query an MBF. The core idea is to skip a module if it does not lead to a reduction of I/Os. In other words, if we anticipate that querying a module leads anyway to an I/O, we will skip the rest of the MBF as an I/O is inevitable. Note that since the modules are accessed sequentially, the decision to skip the d-th module affects the remaining modules. We also allow the algorithm to use a different threshold per module slot for more flexibility.

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

Algorithm 1: MBF decides to skip a module based on its utility (along with threshold_d).

Updates and Deletes. In an update-heavy or delete-heavy setting, our skipping algorithm still works by actively maintaining $\alpha_{l,i}, \beta_{l,i}$ for each SST file and inheriting these values when building new SST files during compactions. For example, if point queries mainly target frequently updated/deleted keys, these queries mostly terminate in shallower levels, leading to fewer accesses to the files in deeper levels that contain obsolete entries ($\beta_{l,i}$ of the files in deeper levels decreases). Therefore, according to the utility definition in Eq. (1), $u_{l,i,d}$ also decreases, which possibly leads to skipping modules when other queries access these files. On the other hand, if point queries concentrate on a few infrequently updated/deleted keys (in deeper levels), $\alpha_{l,i}$ of the files that contain these keys may increase since other point queries that target frequently updated/deleted keys terminate in shallower levels, which again yields lower utility because most of the queries for these SST files are likely to be non-empty queries. As a result, for non-empty queries, BFs can be skipped, and data blocks can be directly loaded, thus saving memory space and I/Os for BFs.

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



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

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

Figure 4: Our skipping algorithm reduces the lookup latency of RocksDB under memory pressure where x-axis represents the percentage of the total BF size (for example, 100% memory budget is 20 MB when 2^{24} keys are inserted).

non-empty queries ($\alpha = 1.0$). More experiments (e.g., varying the number of modules, and allowing modules to be different-sized) 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 files, which leads to all files having BFs with the same bits-per-key (BPK, the ratio between BF size in bits and the number of keys). A larger BPK indicates larger space to hash keys and thus a lower false positive ratio. Notably, to achieve minimal read cost without changing the overall memory space of BFs, Monkey [10] proposes to allocate more BPK to shallower levels. Our main goal moving forward is to use detailed access pattern information 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 (different false positive ratio). The decision of the BPK per file is implemented at *compaction time*. Unlike prior work that assumes a predefined static workload [10], we will employ machine learning techniques (e.g., kernel density estimation) to estimate read access statistics (empty and non-empty queries per level), and this will allow us to identify the best BPK allocation strategy at compaction-time, thus generalizing prior approaches. Our earlier work [11] 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. Notably, the dynamic BPK re-allocation strategy can also be potentially applied when other types of filters (e.g., Cuckoo Filter [12] and XOR Filter [13]) are employed, by simply replacing the false positive rate calculation in our cost function.

[Long-Term] Holistic Memory Tuning. In the long term, we are targeting a set of holistic memory-tuning algorithms that can navigate the entire design space of MBF under limited memory. If we allow each module to have a different BPK and allow each file to have a varying number of modules, we create a more expansive design continuum for MBF, and point queries can be further accelerated in the following two ways: (a) By allowing each module to have a different BPK without changing the total memory size for BFs, files with more empty point lookups can have larger BPK for their first modules while smaller BPK are assigned for the first modules of other files. In this way, most empty point lookups can be blocked by the first modules due to a lower false positive rate. (b) The above design requires compaction to re-construct the MBF. If each file can have three or more modules, we may have higher flexibility when deciding how many modules are required when answering point queries, even before compaction occurs. However, more modules also indicate multiple BF probes for non-empty queries. Our goal is to create a workload-aware solution that leverages the above trade-off and navigates the design space of MBF to achieve minimum point query cost.

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 bottleneck, and, second, the fact that the benefit of BFs diminishes 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 LSM trees, and to study data systems under memory pressure.

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] P. C. Dillinger, P. Manolios, Bloom Filters in Probabilistic Verification, Formal Methods in Computer-Aided Design (2004).
[9] Y. Zhang, et al., ElasticBF: Fine-grained and Elastic Bloom Filter Towards Efficient Read for LSM-tree-based KV Stores, HotStorage (2018).
[10] N. Dayan, et al., Monkey: Optimal Navigable Key-Value Store, SIGMOD (2017).
[11] S. Sarkar, et al., Constructing and Analyzing the LSM Compaction Design Space, PVLDB 14 (2021) 2216–2229.
[12] B. Fan, et al., Cuckoo Filter: Practically Better Than Bloom, CoNEXT (2014).
[13] P. C. Dillinger, S. Walzer, Ribbon filter: practically smaller than Bloom and Xor, CoRR 2103.02515 (2021).