MCF-KV: Multi-Cuckoo-Filter Index based Key-Value Store with Persistent Memory

Hongjia Zou
Zhejiang University
zouhj@zju.edu.cn

Lidan Shou
Zhejiang University
should@zju.edu.cn

Ke Chen
Zhejiang University
chenk@zju.edu.cn

Xuan Zhou
East China Normal University
xzhou@dase.ecnu.edu.cn

ABSTRACT

A Modern KV store typically employs log structured merge trees (LSM-Tree) for organizing data and bloom filters to speed up read. However, both the multi-layer structure and the boom filters come at a performance cost in certain scenarios. In this paper, we leverage persistent memory (PM) devices to improve the performance of LSMT based KV store. We present a novel KV store engine, the Multi-Cuckoo-Filter KV (MCF-KV), which is a new design that replaces in-memory Bloom filters with a multiple-cuckoo-filter index stored in PM, and uses a single-level LSMT to store real data on SSD. In MCF index, we store fingerprint and location information of a key-value pair. While the MCF index performs well in both read and write and has low write amplification, its duplicate key update is costly as it does not distinguish between items with hash collisions. To address this, we harness Fast and Fair B+-tree, a b+-tree index specifically designed for PM, to store key-value pairs that trigger hash collisions. Also, MCF-KV performs overlap-based compaction to efficiently reclaim storage space on disk while avoiding unnecessary write amplification. Our extensive experiments demonstrate that MCF-KV provides 2.1× and 4.9× higher random write throughput, and 2.2× and 1.4× higher point query performance compared to SLM-DB and LevelDB respectively. It also achieves much lower read latency compared to the baselines.

1 INTRODUCTION

Key-value (KV) stores play an increasingly irreplaceable role in modern data-intensive applications, such as cloud storage, social networking and e-commerce. In write intensive workloads, LSM-tree[25, 36] based KV stores like LevelDB[3], RocksDB[4], BigTable[11] and Cassandra[30] are the main products as they are well optimized for these scenarios.

A KV store with LSM-tree based index is able to achieve high write throughput, which mainly benefits from in-memory data buffer (like Memtable in LevelDB) and sequentially flushing of these buffered data to persistent device. To enhance read performance, most KV stores utilize in-memory Bloom filters to speed up the read process. Specifically, a Bloom filter is built when a file is flushed to persistent device, which enables KV store to quickly decide if an item belongs to the flushed file in a number of *probing* operations.

Today, most industrial and commercial KV store products are deployed on machines with DRAM and persistent SSD devices to provide powerful database services. However, the performance of KV stores are still throttled in different ways. First, the in-memory bloom filter is becoming a performance bottleneck as the speed gap between SSD and DRAM devices gradually narrows [17, 38]. Moreover, bloom filter also introduces long tail latency for read operations due to its probabilistic nature[17, 26, 38]. Second, high write amplification, which is mainly caused by merge-sort operations (i.e, compaction) to reclaim storage space and reorganize KV pairs' physical locations for fast read, results in unpredictable performance and great degradation of foreground user experiences[33, 40, 42].

Byte-addressable persistent devices provide new opportunities to improve the performance of KV store systems. In the past few years, a batch of works have emerged to employ persistent memory (PM) devices to address the above problems. Some works[19, 27, 32] followed traditional LSM-Tree architecture, most of them place level-0 data and Memtable in the PM as an extra storage layer. By placing hot data in a faster and persistent medium and carefully optimizing the compaction scheme, these works achieved remarkable results in performance. But they failed to address the problem of high write amplification and long tail latency in read operations. Some works tried to replace traditional LSM-Tree architecture with their own optimized index and place the index on PM device. They successfully eliminate in-memory bloom filters and get rid of high write amplification. However, at the same time, they introduce new knotty problems that cause severe performance fluctuation. For example, SLM-DB[26] achieves high performance in read and random write, but its sequential write performance is pretty weak due to expensive maintenance on persistent B+-tree index.

In this paper, we leverage PM device to improve the performance of LSM-Tree based KV stores. We present a novel KV store called Multi-Cuckoo-Filter-KV (MCF-KV), which is carefully designed to achieve high performance in both reading and writing with low write amplification. We propose an MCF index, which is equipped with multiple Cuckoo filters[20], to provide stable and consistent probing performance. Specifically, we use multiple Cuckoo filters in our index, since a single Cuckoo filter suffers from unpredictable load factor and long tail latency while rehashing. However, multi-Cuckoo-filters do not secure high and stable performance because of potential hash collisions.

A hash-based index needs to either do a read-before-write operation to store a hash-collision item, or employ an append-only strategy to first accept the item and then delay the checking to a later read or rehash process, which leads to unpredictable performance. To address this, we use a B+-tree index, which is also placed in the persistent memory device, to store collision items. In other words, the Cuckoo filter index only accepts items that do not trigger hash collisions. (For clarity, we will refer to the entire index as the MCF index.) Additionally, to maintain

high write throughput, we stage inserted KV pairs in a memory buffer and flush them to SSD devices in an append-only manner.

In MCF-KV, KV pairs are stored on disk within a single-level LSM-Tree organization. With the location information provided by MCF index, there is no need for on-disk data to provide a strictly sorted order, which enables MCF-KV to significantly reduce the write amplification caused by compaction, contributing to better and stable system performance. However, we still need some mechanism to ensuring some degree of locality of KV pairs on disk to provide acceptable read performance. To address this, we propose well-designed compaction scheme to reorganize the KV pairs in the single level LSM-tree.

The main contributions of this work are as follows:

- We design a new index structure that integrates Cuckoo filters and persistent B+-tree to achieve optimal overall performance. We carefully design a hash-based index using multiple Cuckoo filters, which is placed in PM layer, to store the fingerprint and location information of KV pairs for fast read. Additionally, to overcome the unpredictable performance caused by hash collisions, we introduce a persistent B+-tree index as a fallback to store the full information of keys that cause hash collision and thus eliminate read-before-write or later check operations.
- We devise an overlap-based compaction scheme, which selectively picks up files for compaction based on each file's overlapping ratio, a quantity that measures its overlap with other files.
- We implement MCF-KV based on LevelDB. The main data structures of MCF-KV, namely the multi-Cuckoo-filters and the B+-tree in persistent memory, as well as the single-level LSM-tree in SSD, are carefully designed to provide robust durability guarantees and maintain crash consistency. We also carry out extensive evaluations on MCF-KV using microbenchmarks and YCSB for real-world workloads. Our experimental results show that MCF-KV achieves high write and point query performance with low tail latency.

The rest of this paper is organized as follows: Section 2 discusses the challenging issues of LSM-Tree based KV store, including high write amplification and slow in-memory bloom filter. We also discuss PM technology in section 2. Section 3 demonstrates the detailed design and implementation of MCF-KV. Section 4 discusses the compaction scheme of MCF-KV. Sections 5 talks about the store operations and recovery process of MCF-KV. Section 6 gives evaluations on MCF-KV and presents the experimental results. Section 7 concludes the paper. The source code of MCF-KV is available here [1].

2 BACKGROUND AND MOTIVATION

In this section, we first demonstrate the necessary information and challenging issues of LSM-Tree based KV store by focusing on LevelDB; then we introduce the background of PM device; finally, we talk about related works that leverage PM to optimize LSM-Tree based KV store and their deficiencies.

2.1 LSM-Tree Based KV Stores

LSM-Tree based KV stores typically batch write requests in memory and sequentially flush data to storage devices to fully utilize the high sequential write bandwidth of the storage devices. For read requests, these stores use in-memory bloom filters that are probed to conservatively decide if an item belongs to a file, thereby mitigating actual lookups.

2.1.1 Background of LSMT-based KV Store. In this section, we will focus on LevelDB, a well-studied LSM-Tree based KV store to demonstrate the background of LSMT-based KV store. Figure 1(a) illustrates the architecture of LevelDB, which includes a DRAM component and an SSD component. The DRAM component is composed of *MemTable*, *Immutable Memtable* and bloom filters. The SSD component primarily consists of multi-level sorted *SSTables*. In addition, it utilizes write-ahead log (WAL) on SSDs to safeguard KV store service against system crashes.

For DRAM component, MemTable and Immutable MemTable are basically sorted skiplists. For write requests, LevelDB first buffers incoming data (i.e., KV pairs) in MemTable. Once MemTable is full, LevelDB transfers MemTable into Immutable MemTable and generates a new MemTable for future write operations. KV pairs inside Immutable MemTable are flushed to SSD devices as on-disk data structure SSTable, and at the same time, a bloom filter of this SSTable, which helps to tell if key exists in this SSTable, is constructed to enable fast search for future read operations. Note that deletion and update operations are treated as a special case of write operation. Before actually inserting KV pairs into MemTable, KV pairs must first be appended to the WAL for the purpose of crash consistency. And after KV pairs are persisted in SSTables on disk, the log containing their information can be safely deleted.

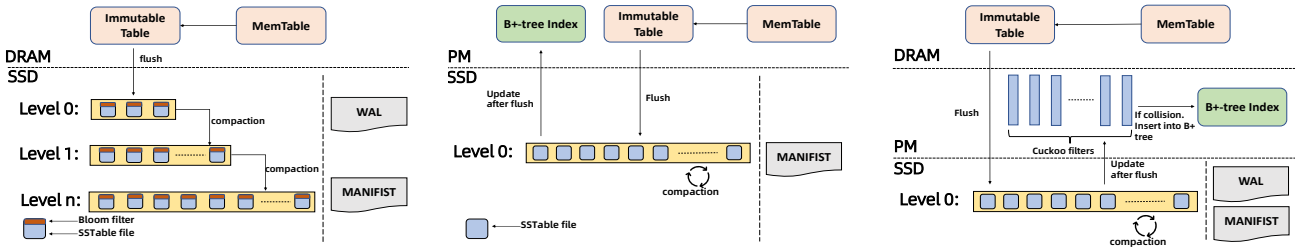
For the SSD component, LevelDB maintains a multiple level LSM-Tree of SSTables. In particular, from L_0 to L_k every level has one or more sorted SSTable files. In each level (except L_0), the key range of a specific SSTable does not overlap with others. Each level can only retain limited number of SSTables, but higher level can keep more SSTable files than lower level.

To maintain such leveled architecture, SSTables are gradually compacted from L_0 to higher levels. When L_x grows beyond its limit or too many read and scan operations are performed in L_x, LevelDB selects a qualified SSTable (called victim SSTable) in L_x and multiple SSTables in L_{x+1} that overlap (in key space) with the victim SSTable. Then a merge sort is performed among these SSTables to generate new ones. All of these new SStables will be flushed into L_{x+1}. The process above is called *Major Compaction*. Since L_0 is not sorted, LevelDB does not perform merge sort in L_0 compaction. Instead, it directly flushes the data in Immutable MemTable to L_0. The compaction of L_0 is called *Minor Compaction*. Compaction mechanism helps reclaim storage spaces and makes each level sorted (except L_0), which reduces the overhead of subsequent reads.

2.1.2 Challenges faced by LevelDB. While LevelDB is widely adopted as KV store, it faces technical challenges. We focus on the following issues.

High Write Amplification. The first issue of LevelDB is its high write amplification (WA)[33, 42]. WA is defined as the ratio between the size of data written to disk and the size of data written by users. To maintain fully sorted order of KV pairs in each level (except level 0), LevelDB needs to conduct compaction operations to move KV items to higher levels via background compaction. As size of two adjacent levels grows by the *amplification factor* (AF, default 10), the WA to compact an SSTable file from l_x to l_{x+1} equals to AF on average. Moreover, for a k-level LSM-Tree, the WA ratio to compact a KV pair from l_0 to l_{k-1} can be higher than $10 \times k$[33, 34, 40].

[1] https://github.com/dilab-zju/MCF-KV



(a) LevelDB architecture (b) SLM-DB architecture (c) MCF-KV architecture

Figure 1: Architecture of different KV store

Slow Build/Read of bloom and Long Tail Latency. Another well known issue of LevelDB is the performance of its per-SSTable built-in bloom filter. (1) bloom filters are known to be slow in both construction and lookup[23, 26, 31]. As the usage of SSD dwarfs the cost of disk I/O operations in the KV store, the cost of in-memory operations, particularly those on bloom filters, emerge as the new bottleneck[7, 16, 35, 39]. The construction of bloom filters in the compaction process of LevelDB takes 15%-70% of the total compaction time[17, 23, 31]. In our own evaluation, as shown in Table 1, bloom filters take about 20% of the total query time. (2) bloom filters also cause long tail latency in read operations due to high false positive rate. In our evaluation, nearly 1.5% read operations incur false positive query and require more than one disk I/O, causing long tail latency that is 8× longer than the average read latency. Such latency leads to unpredictable read performance.

2.1.3 Motivation for Multiple Cuckoo Filters. In recent years, a number of new data structures are proposed as alternatives to bloom filter, such as Cuckoo filter[20], Morton filter[8], Xor filter[21] and others. In this paper, we propose to use Cuckoo filters instead of bloom filters, the reasons are as follows: First, a Cuckoo filter requires lower building costs in memory accesses and computation when compared to bloom filter[20]. Second, Cuckoo filter maintains very good false positive rate[17, 20], which is critical for reducing the long tail latency in read operations. Third, Cuckoo filter is able to achieve an average load factor of 94% [17, 20, 41], which means it consumes storage space efficiently.

Despite the above stated advantages, Cuckoo filters are limited in the following aspects. First, due to the absence of efficient rehashing mechanism, Cuckoo filters may still incur long tail latency in write operations as the number of maintained keys increases. Second, Cuckoo filters may probably suffer from poor space utilization. A Cuckoo filter is considered full when an unsuccessful insertion takes place. At this moment the load factor (i.e. number of entries used in hash Table divided by the total number of entries) reflects the space utilization of the filter. The

Table 1: *LevelDB read cost breakdown*

Operation	time cost	cost percentage
File search	0.71ms	6.1%
Index block search	6.47ms	56.0%
Bloom Filter check	**2.31ms**	**20.0%**
Data block read	2.07ms	17.9%

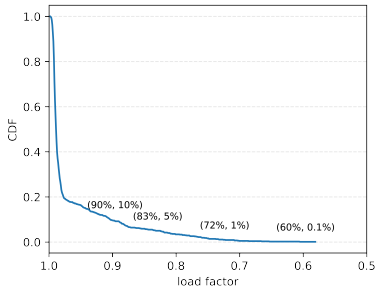


Figure 2: The cumulative distribution of the load factor of 1000 random cuckoo filters, showing 10% of the filters having load factor less than 0.9.

cumulative distribution of the load factors of random Cuckoo filters reveals that a filter has the probability of 5% to have load factor below 83%. To address the above mentioned problems, we propose to use a set of Cuckoo filters as the global index for all key-value pairs in the KV store. Such design statistically retains high load factor and provides high and stable performance.

2.2 Persistent Memory Based LSM KV Stores

Persistent Memory (PM) is a type of storage technology that deployed between traditional Random Access Memory (RAM) and non-volatile storage in the memory hierarchy, such as hard disks or flash memory. It can save data in the event of system power loss or failure, making it more reliable and faster to access. The demand for PM technology is objectively present because it can significantly improve the performance and reliability of computer systems, which make it a popular research object in recent years.

PM such as phase-change memory[6, 37], spin transfer torque MRAM[29] and 3D XPoint[24] provide database service providers new opportunities to improve their quality of services. PM has the following three attractive attributes: fast, byte-addressable and persistent. On the one hand, PM is able to provide disk-like persistence with higher capacity at a much lower cost when compared to DRAM[2, 18, 28]. On the other hand, PM provides $30-50\times$ lower read/write latency and $6-15\times$ higher bandwidth when compared to SSD[2, 5, 10, 18].

In recent years, a batch of works emerged to exploit PM to improve the performance of LSMT-based KV store. But some fail to address the problems mentioned above, some introduce extra knotty problems that make the performance even worse;

and some even eliminate the SSD part and place all data in PM device, such design reaches good performance but fails to provide acceptable recovery performance, which makes it insufficient for industrial use. The rest of this section will discuss related works on PM and their deficiencies.

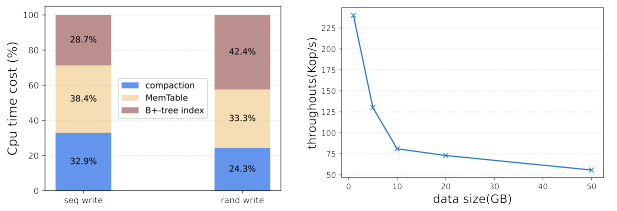
NoveLSM. NoveLSM[27] leverages PM to deliver high throughput. Specifically, it employs PM device as an alternative DRAM to enlarge the MemTable and immutable MemTable, which enables direct update in MemTable and results in less compaction. However, such design only postpones the compaction as it merely increases the size of data buffered in storage but fails to optimize the compaction scheme. Besides, the enlarged MemTable increases the data involved in compaction and leads to even more severe problem of write amplification, which leads to great fluctuation in throughout. Also, it fails to resolve the problem of slow read and long tail latency caused by bloom filter.

SLM-DB. SLM-DB[26] exploits PM to optimize LSM-Tree based KV store and reaches high throughput in both read and random write. As shown in Figure1(b), it places MemTable and Immutable Table on PM device to get rid of WAL, and maintains a persistent B+-tree index to keep KV pairs' meta information, which enables SLM-DB to organize its SSTable files as a single level LSM-Tree. Its high throughput is mainly delivered from the single level structure of SSTable files and elimination of WAL. It also gains benefits from keeping metadata in B+-tree index, which enables it to skip bloom filter for fast and precise query. However, in our evaluation, we find that the cost of maintaining the B+-tree index could be of great expense that exerts a great impact on the overall performance. In our evaluation, we write 40GB data set of 1KB KV pair to SLM-DB. As shown in Figure 3(a), maintaining B+-tree index on PM costs 28.7% cpu time in sequential writes and 42.4% CPU time in random writes, which demonstrate that the B+-tree index in SLM-DB is not efficient enough and turns out to be the performance bottleneck.

Moreover, we also find that SLM-DB's random read performance deteriorates drastically with the data size growing up. In our evaluation, we insert different volume of data into SLM-DB and test its random read performance. As shown in Figure 3(b), the read throughout drops near 4× when the payload grows from 1GB to 50GB.

uTree. uTree[12] differs from other works as it abandons the conventional three-level-architecture (namely memory, persistent memory, and SSD/HDD). uTree discards the SSD/HDD layer and stores all data in the persistent memory. To efficiently search for items in PM, it maintains a B+-tree in DRAM as a global index. While uTree does achieve good performance in throughput and latency, its recovery time after system failure could be unacceptable. It may also incur vast memory consumption when the size of the database expands, since it maintains the entire B+-tree in DRAM. Furthermore, since all data are placed in PM and indexed in DRAM, the hardware price to build such storage scheme could be prohibitive.

To summarize, while PM provides attractive properties and enables potential performance improvement, a stable persistent index to efficiently exploit PM device is still in absence. In this paper, we present MCF index to better leverage PM device as a global index in LSMT-based KV store to reach high and stable throughput with low latency.



(a) Time breakdown for SLM-DB under write workload

(b) SLM-DB random read performance under different data size

Figure 3: SLM-DB performance analysis

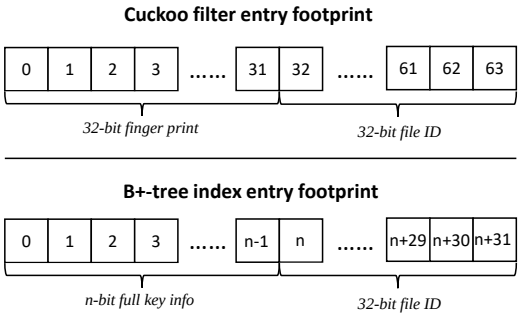


Figure 4: Entry footprint of Cuckoo filter and B+-tree index

3 MULTI-CUCKOO-FILTER KV(MCF-KV) DETAILED DESIGN

In this section we present the design of Multi-Cuckoo-Filter KV (MCF-KV). Figure 1(c) shows the overall architecture of the system. In a nutshell, MCF-KV organizes SSTables as one-level LSM-Tree and keeps the metadata of KV pairs in the PM layer. In the following, we will focus on the MCF index (including multi-Cuckoo filters and a B+tree), and the write/read processes of the index. We also explain the reasons/justifications for the design of the data structures and mechanisms.

3.1 The Design of MCF Index

In MCF-KV, instead of keeping one single Cuckoo filter in PM layer, we maintain N_c Cuckoo filters with different maximal load factor, which store KV pairs' finger prints (i.e. 32-bit hash values of keys) and location information. The reasons for doing so would be explained in section 3.1.

Moreover, we also encounter problems with hash collision. When a filter accepts a key that causes a hash collision, it is unable to determine if the two entries are identical based solely on fingerprint. In this situation, the filter must perform an additional I/O operation to finish the check work, which is bound to incur long tail latency and causes great impact on the system performance. To address this, we introduce an auxiliary B+-tree index to accept keys that cause hash collisions. Among the different versions of persistent B+-tree implementations, we choose FAST and FAIR B-tree[22] because of its excellent read and write performance due to the ordering constraints on store instructions. The reasons for such design would be discussed in section 3.1.

Multiple Cuckoo Filters. As shown in figure 1(c), we employ several Cuckoo filters for the purpose of maintaining KV pair's

meta information. We preserve, as shown in figure 4, a 32-bit finger print of the key and a 32-bit file ID of the SSTable containing the key, which sum to 8 bytes and meet the the granularity demands of PM device[2, 22]. The reasons for doing so are twofold. 1) A 32-bit finger print provides both low false positive rate and low hash collision rate (theoretically $\frac{1}{2^{32}}$); 2) 32-bit file number sets the maximal file number to $2^{32} - 1$, which we believe is sufficient for most scenarios.

The design of multiple Cuckoo filters is used to solve the problem mentioned in section 2.1.2. The reasons are simple and straightforward. First, using multiple Cuckoo filters contributes to more stable and predictable load factor and less potential storage space waste. Assuming the the variance of a single Cuckoo filter's load factor is V, the variance of MCF index's load factor is V_m, and the number of filters inside MCF-index is n, the variance of each filter is V_i, i is from 1 to n, then we have

$$V_m = \frac{V_1 + V_2 + ... + V_n}{n^2} = \frac{n \times V}{n \times n} = \frac{V}{n}$$

a smaller variance of load factor means we are more likely to avoid extreme situations that make the whole filter full with low load factor as we have on-average 94% load factor. Second, we employ a family of Cuckoo filters to amortize the cost of rehash process. Each filter can be regarded as a subassembly of a big filter, therefore, instead of rehashing the big filter, we divide the job into sub-jobs and assign them to each part of the filter and finish the rehash job in different time period, which contributes to lower the peak of tail latency caused by rehashing process.

However, the above mentioned mechanism is not sufficient to avoid long tail latency caused by rehashing. Due to the even distribution of entries, it is possible for most of the Cuckoo filters to start rehashing process simultaneously, which means there would not be enough filters for MCF-KV to use when inserting items. This greatly deteriorates the overall performance of the system.

To overcome such problem, MCF-KV introduces load factor randomization technique. In detail, instead waiting for filters to be full (i.e. reaches maximal Cuckoo count when inserting an item) and starting rehashing process, we give each filter its own maximal load factor, which is $V_l \pm f_l$. In our implement, for example, we set V_l to 0.92 and f_l to 0.02, then each Cuckoo's load factor will be a value from $0.90 - 0.94$. This helps MCF-KV to successfully amortize the rehash cost at the price of little space waste.

Auxiliary B+Tree Index. With the mechanism mentioned above, MCF is able to handle most write operations efficiently. However, hash collision can exert great impact on the system performance. While the probability of hash collisions caused by normal write operation can be as low as $\frac{1}{2^{32}}$, certain operations (i.e. updates and deletes) are bound to cause hash collisions. Some works[38] leverage extra IO to handle hash collisions, some[17] delay the IO and check the validity of items later, which both result in long tail latency.

In our evaluation, we identified two types of operations that result in hash collision. The first type includes write,update and delete operations from foreground; the second type is performed by compaction thread during background compaction. When the compaction thread modifies the physical location of a valid KV pair, it needs to update the corresponding metadata (i.e. file ID) in the Cuckoo filters. This is a special case of update operation and will be named as background update in the rest of this section. Because of the low probability of hash collision between different keys, nearly all hash collisions are caused by update operations,including both foreground update and background update, but Cuckoo filters still need to pay heavy price (i.e. actual I/O) to do the check work.

Note that most hash collisions are caused by duplicate keys, and the most appropriate method to handle these keys is in-place update. Based on the observation above, to avoid incurring extra IO, MCF-KV introduces an auxiliary B+-tree index () as the container of keys that cause hash collision from foreground operation, and performance in-place update for background update operation. As shown in figure 4, the B+-tree index preserves full information of the key and its corresponding 32-bit file ID.

There are several reasons to justify such design. Primarily, it effectively resolves the problem of hash collisions by maintaining full information in a B+-tree index. Moreover, B+-tree index provides efficient read and write performance when tree size is small. As only frequently updated and accessed data is stored in B+-tree index, MCF-KV is able to maintain it at low cost while deriving significant benefits from both its high read and write performance. The impact exerted by B+-tree index on the performance will be discussed in section 7.2.

Algorithm 1: Insert algorithm

```
Input: filters:Multi-Cuckoo filters
       fftree:B+-tree index
       queue:key queue
/* start insertion                                   */
while !queue.empty() do
    key = queue.top();
    queue.pop() ;
    f_id = hash_1(key);
    fp = hash_2(key);
    if filters[f_id].contains(fp) then
        res := fftree.Insert(key) ;
        CHECK(res.ok) ;
    else
        if filters[f_id].IsRehashing then
            queue.push(key);
        else
            res := filters[f_id].insert(key);
            if !res.ok or
              filters[f_id].loadfactor ≥ threshold then
                filters[f_id].Rehash();
            end
        end
    end
end
```

3.2 Operations on MCF index

Now we will present the write/read process of MCF index. We also discuss the rehashing process of Cuckoo filters in MCF index.

Write Process. In MCF-KV, when Immutable MemTable is compacted to SSD, finger prints of keys in that table and the corresponding file ID will be encoded as an item and then inserted into MCF index.

Specifically, in MCF-KV, a queue is used to preserve all keys to be inserted. When MCF-KV attempts to insert a key into a filter, it first decides which filter the item belongs to via a hash

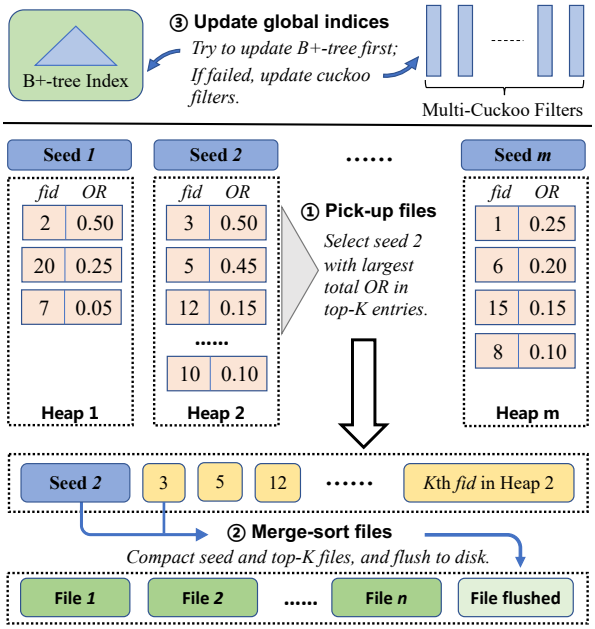


Figure 5: Illustration of the three steps of compaction

function (*line4 in Algorithm* 1) and then calculates the finger print of the key via another hash function (*line5*). Next, MCF-KV checks if the finger print causes a hash collision. If so, it turns to the auxiliary B+-tree and stores full information of the key and it's corresponding file ID in the tree (*line6 - line8*). If the finger print does not cause a hash collision, MCF-KV checks if the filter is being rehashed. If so, it moves the key to the tail of the queue and continues with the next key to avoid waiting (*line10 - line11*). If the filter is not being rehashed, MCF-KV then tries to insert the key into the filter (*line13*), if it fails (i.e. reaches maximum Cuckoo count) or the load factor of this filter reaches the threshold, which means this cuckoo filter needs to be rehashed, MCF-KV creates an extra thread to do the rehashing job and moves on with next key (*line14 - line16*).

Read Process. When searching for an item, MCF-KV first looks in the auxiliary B+-tree since it always maintains the newer version of an item if exist. If it fails to find the item in the B+-tree, MCF-KV then turns to Multi-Cuckoo filters. MCF-KV first uses hash function to locate the appropriate filter and then applies the Cuckoo algorithm to find the target slot with at most two access to the PM device. After locating the target slot, MCF-KV then decodes the file ID read from the slot, which enables it to fetch the value from the corresponding SSTable.

Rehashing Process of Cuckoo Filter. The hash table of a Cuckoo filter has to be rehashed in two cases: 1) when an insertion into this filter fails and 2) when the load factor of this filter reaches threshold after insertion. To perform the rehash, MCF-KV will generate an extra thread. During the rehash process, the thread creates a new hash table, which is double size of the old hash table. It then scans the whole original hash table and transfers all items into the new one via Cuckoo algorithm. All write operations to this filter are delayed during rehashing, but read operations can still be performed normally.

4 COMPACTION SCHEME

In this section we present the compaction scheme of MCF-KV. Since we employ a single-level organization of SSTables, the compaction process that reclaims the space occupied by obsolete KV pairs has to be conducted among the SSTables in the same level. In the remainder of this section, we shall refer to SSTables and files interchangeably. We keep monitoring accesses to all files and newly flushed files while maintaining their metadata. As shown in Figure 5, the compaction process runs in the background in three steps: (1) First, we select files eligible for compaction (also referred to as *compactees*). (2) Second, we merge-sort (i.e. compact) the compactees into new files. (3) Third, we update the global indices (both the multi-Cuckoo filters and the B+-tree index), so that they become consistent with the disk files. Since the second step is straightforward and almost same as those in other LSMT-indices, we shall focus on the first and third steps.

4.1 Pick-up Files For Compaction

Generally, the objective of compaction is to improve the data locality and reduce the space consumption of files. Therefore, our selection strategy aims at selecting: 1) *files with high proportion of obsolete keys*, 2) *files that are frequently accessed*, and 3) *files that significantly overlap with others*.

Files that satisfy the above criteria 1) or 2) are called *seed* files. We can easily determine if a file is a seed file by checking its own metadata as MCF-KV stores the live-key-ratio for every SStable file.

Seed files are considered good candidates for compaction, but do not guarantee an effective compaction can be undertaken right away. We may need to consider other files on disk, which, together with the seeds found so far, could possibly be compacted. As shown in algorithm 2, given a set of m seeds denoted by $S = \{s_1, \ldots, s_m\}$, we maintain for each seed s_i a heap structure h_i, and try to allocate the n files to these m heaps based on how much they overlap with the seed files (*line1 - line13*). We use overlap ratio $OR(f1, f2)$ to denote how much $f1$ overlaps with $f2$ in the rest of this section. The detailed process of computing OR and building h_i is given in *Appendix A*.

Next, for each heap h_i, we select its top-K tuples, and compute their *total OR* as $\sum_{f \in topK} OR(f, s_i)$ (*line16 - line21*). K is an empirical parameter chosen to bound the compaction cost. The seed with its top-K tuples having the maximum total OR, denoted by s_i^{max}, is chosen as the compactee. The K files in the respective tuples are the other compactees. Subsequently, s_i^{max} and its top-K heap files will undergo a compaction and get merge-sorted (i.e. Step 2).

It should be noted that, during the compaction, the validity of KV pair in these files are checked via the global index. Only valid data will be flushed to new files (namely SSTables).

4.2 Index Update For Compaction

After a new file is flushed to disk through compaction, the metadata of these valid keys, namely the (key, file-ID) information, has to be updated as well. As shown in Figure 5 part 3, we first try to update the B+-index as it is supposed to contain the latest file ID information of a KV pair if any collision has happened on that key. If a certain key is not found in the B+-tree, then we turn to the Multi-Cuckoo-Filter index and execute an in-place update with the corresponding (fingerprint, file-ID) information. Note that MCF-KV avoids hash collision via the B+-tree index, therefore if a valid key is absent from B+-tree, its fingerprint is

Algorithm 2: Compactee pickup algorithm

Input: $files$: SSTable files in the current system
S: seed files selected due to low r_l and too many read operations.
H: heaps for each seed file.
Output: $compactees$: files to be compacted.

for $f \in files$ **do**
  $MAX\_OR = 0$;
  $ID = -1$;
  for $s \in S$ **do**
    if $ratio(f, s) \geq MAX\_OR$ **then**
      $MAX\_OR = ratio(f, s)$;
      $ID = s.id$;
    end
  end
  if $ID \neq -1$ **then**
    $h[ID].add(MAX\_OR, f.id)$;
  end
end
$MaxRank = -1$;
/* id of best seed */
$max = -1$;
for $h \in H$ **do**
  if $\sum_{j=0}^{k} h[j] \geq MaxRank$ **then**
    $MaxRank = \sum_{j=0}^{k} h[j]$;
    $max = h.id$;
  end
end
$compactees.add(s_{max})$;
$compactees.add(top\text{-}K\ in\ h_{max})$;
return $compactees$;

certain to appear in the MCF with no hash collision. After the index update process and the creation of new files is complete, MCF-KV commits the metadata of these changed files to the MANIFEST and deletes the obsolete files.

5 IMPLEMENTATION

We implement MCF-KV based on Google's popular open-source KV store LevelDB [3]. We use LevelDB as the code base mainly for two reasons: First, the main comparative works (SLM-DB and NoveLSM) are both implemented based on LevelDB. Therefore it would be more fair and rational to start the implementation from LevelDB. Second, LevelDB is implemented in a highly concise manner and works without much optimization. It would be easier to extend LevelDB to study how our proposed designs influence the system performance. Our code accesses the PM device via the persistent memory development kit (PMDK)[1?] in the APP Direct (DAX) mode, and accesses the SSD via the Posix API. In the following, we shall present the implementation of the write/read processes and the recovery mechanism of MCF-KV.

Write. All KV pairs are written to MCF-KV via a foreground thread. Each KV pair is first buffered in the MemTable in DRAM. When the Memtable grows large enough, its KV pairs will be flushed to an SSTable on disk, and meanwhile the location information of these KV pairs will be recorded in the MCF index (specifically, in either one of the Cuckoo filters or the B+-tree

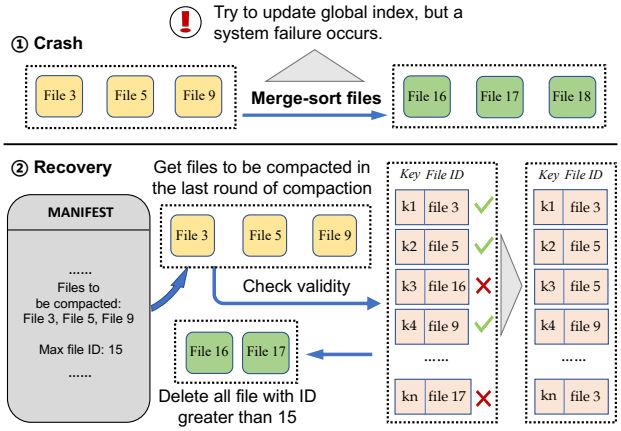


Figure 6: MCF-KV recovery process when crashed during compaction

index, depending on if the key generates a hash collision). The location information of the KV pairs will be updated if the SSTable containing them is compacted.

Read. To read the value of a given key, we first search for the target key in the Memtable and the ImmutableTable. If the key is not found, we shall search for the key in the MCF index. We first look up the key in the B+-index as it always keeps the newest version of the KV pairs. If it is not found, we then search the multiple Cuckoo filters. If the key is found in one of these indices, we read the respective value from disk via the file ID obtained during the above search. If the target key is not found in the MCF index, we return null to the read request.

Compaction. MCF-KV extends the compaction scheme to allow for multi-thread compaction. Specifically, we partition the compaction task by key range into several disjoint jobs that can run independently. For example, given a compaction task of keys ranging from k_1 to k_n, we can spawn x threads and divide the whole compaction into x independent jobs, where the key range of each job does not overlap those of the others. In this way the compaction task can be executed in a multi-thread manner.

Recovery. MCF-KV guarantees strong crash consistency for data structures in PM during system failures[13, 14, 27, 32]. Apart from the recovery mechanism inherited from LevelDB, the PM-resident index of MCF-KV requires additional design to guarantee crash consistency. To address this issue, we need to consider the write and update operations on PM, which may occur in two cases in MCF-KV: namely 1) when flushing a file to disk, and 2) when updating a file during compaction.

Case 1) For the first case, the MemTable is flushed to an SSTable file and the respective index entry is inserted into the MCF index. If a failure happens during the insertion of the index entry, MCF-KV is able to re-process all insertions via WAL.

Case 2) If the system crashes when background update is ongoing, MCF-KV can leverage the MANIFEST file to restart the compaction as LevelDB does. Figure 6 illustrates the recovery process of MCF-KV when there is a system failure during compaction. In MCF-KV, the MANIFEST keeps the essential information of the target database, including the file IDs of the seeds and the compactees. We also maintain in the MANIFEST the maximum file ID M persisted so far. A file with an ID greater than M is considered invalid (or not persisted).

The recovery for case 2 is processed as follows. First, MCF-KV scans the whole file directory and deletes all files with IDs greater than M. Then, MCF-KV retrieves the information of the the compactee files stored in the MANIFEST and checks the file IDs appearing in all entries of the compactee files via MCF index. If any entry contains a file ID greater than M, indicating that it was updated in the last unsuccessful compaction, the file ID is restored to the old one (i.e. the file ID of the compactee file containing the key). Once all the KV pairs in the compactees are checked and duly restored, the system reaches a previous consistent state and is ready to restart the interrupted compaction.

File-ID Recycling. The above recovery scheme relies on an ever-growing maximum file ID M. Since we employ 32 bits to store file IDs, it is possible that M overflows the integer space. To address this issue, we adopt a file-ID recycling strategy. Specifically, we maintain a file ID range $[Min, Max]$, which indicates the valid file ID range of the current database. All valid file IDs fall in this range. We can perform a *permutation* operation which replaces the file IDs of Min by Max+1 for all index entries containing Min. Then the valid range is updated to $[Min_{new}, Max + 1]$, where Min_{new} is the next valid file ID greater than Min. By performing permutations at a reasonable rate, the invalid file IDs can be recycled and the valid range is kept tight. More importantly, if Max overflows, the new ID will start from 0 as it must have been recycled. If a system crashes during file-ID recycling, we can use the MANIFEST to redo the recycling.

Maintaining B+-tree in The Long Run. The B+-tree is inserted mostly when a KV pair is updated for the first time. Further updates to the same pair do not increase the B+-tree. The size of the B+-tree will grow only when cold data are widely updated, which is very rare in production environment [2]. If the B+ tree really grows too large, one possible solution is to freeze the large B+-tree and create a new B+-tree (which replaces the role of the old B+-tree in write path). Then we can move all entries in the frozen B+ tree back into the existing Cuckoo filters. Although this procedure may contend bandwidth with the flushing operations, it results in a reduced B+-tree and frees the space occupied by the old B+-tree.

6 COST ANALYSIS

To illustrate the superiority of MCF-KV, we conduct cost analysis for read/write/update operations made by MCF-KV versus two comparative schemes, namely LevelDB and SLM-DB, in terms of the number of accesses to PM and SSD. Similar to SLM-DB, we assume that the filter blocks and meta blocks of SSTables are cached in DRAM, but the index blocks and data blocks are not. Each of the three KV stores is assumed to have already maintained N KV-pairs before an operation starts.

6.1 Read Costs.

LevelDB first reads an index block to locate the target item and then reads the data block to fetch the data if the bloom filter hits, which incurs two I/Os. If a false positive occurs in the bloom filter, it still conducts the above actions but fails to find target value in the index block (i.e. wasting 2 I/Os) and then it continues reading process in the next level. It is rare to have more than one false positive query during a read operation. Therefore, the expected read cost of LevelDB when false positive happens is 4 I/Os.

SLM-DB and MCF-KV provide constant SSD I/O costs as they avoid false positive query via specific data structures. SLM-DB requires only 1 disk I/O as it stores the detailed location information of KV pairs in the persistent B+-tree. MCF-KV locates the specific SSTable and provides the query result in 2 accesses to SSD.

On the PM part, SLM-DB maintains a persistent B+-tree with height of $log_f N$ where f denotes the fan-out of the tree (default to 16 in SLM-DB), it takes $log_f N + 1$ PM accesses to locate the target value. As shown in Table 2, given a database of 10 million pairs, it takes up to 8 accesses on PM to finally locate a target item. In contrast, MCF-KV only makes one or two PM accesses to fetch the location of the target value via the Cuckoo algorithm. Hence its cost on the PM is significantly lower.

6.2 Write Costs.

For write operation, LevelDB first writes the item in the WAL and then flushes the data into level-0 via minor compaction. The data will be gradually compacted from level-0 to higher levels to maintain its strict multi-leveled SSTable architecture, which incurs great amount of rewriting. We denote by R the average percentage of data in level-x to be compacted with level-$(x + 1)$, then it takes $\frac{1}{R^t}$ I/Os to move a KV pair from level-0 to level-t. In production scenarios, the actual number of I/Os to move a KV pair to its final level may vary from 5 to 8.

MCF-KV and SLM-DB outperform LevelDB because of their single-level SSTables and carefully designed compaction mechanisms, which significantly reduce number of compactions and hence results in $2 - 3$ I/Os on SSD. The cost of PM accesses by MCF-KV may vary from 1 to M where M is the maximum Cuckoo count specified in the Cuckoo algorithm[20]. In the production environment, most items require no more than three PM accesses to find an empty slot. Therefore, the expected number of PM accesses in the write operation of MCF-KV is 3. The cost of PM accesses by SLM-DB is same as that for read operations.

6.3 Update Costs.

For update operation, LevelDB and SLM-DB incur the same costs as write operation. MCF-KV requires extra PM accesses to maintain the auxiliary B+-tree for the duplicate items. Given the number of $D \times N$ duplicate items, it takes $log_f(D \times N)$ accesses to update an item, where f is the fan-out of the B+-tree. Consider only a small portion of the data to be updated in production environment[9], it may take on average 3 to 4 PM accesses to update an item in MCF-KV.

The values of the estimated costs of various operations for the three KV stores are presented in Table 2, assuming that the database already contains 10 million KV pairs.

Table 2: *Estimated costs of various operations with 10 million items(* indicates false positive case in bloom filter)*

Estimated	read		write		update	
# of accesses	PM	SSD	PM	SSD	PM	SSD
LevelDB	\	2(4*)	\	5-8	\	5-8
SLM-DB	8	1	8	2-3	8	2-3
MCF-KV	1-2	2	3-4	2-3	3-4	2-3

[2]Previous work[9] shows that only 20% of data will be updated more than once in most production scenarios

Table 3: Space cost and False Positive Rate of different indices

Symbol	Meaning	Default
N_c	Number of Cuckoo filter	12
V_l	Base load factor	0.92
f_l	Fluctuation ratio of load factor	0.02
R	Files with live key ratio $\leq R$ are considered as seeds	0.7
k	Number of files selected for compaction	3

7 EVALUATION

This section presents the experimental results of MCF-KV. In the experiments, we first evaluate the PM-resident indices of MCF-KV using the micro-benchmarks. Then we report the overall performance of MCF-KV, using both micro and macro-benchmarks.

7.1 Experiment Setup

Our experiments are conducted on a machine with two Intel Xeon Gold 6240C processors (2.60GHz) and 400GB memory. We employ two storage devices, an 1.5TB Intel SSDPEDME016T4F SSD and 128GB NVM of one single 128 GB Intel Optane DC PMM[24]. Ubuntu 20.04 of Linux kernel version of the machine is 5.4.0 is used for the machine.

When running the experiments, we follow the same settings of Matrix[42] and SLM-DB[26]. We restrict the DRAM size to 16GB via memory kernel parameters. For PM device, we leverage the PMDK to manage an NVM pool of 8GB. We also set the default MemTable/SSTable size to 64MB and turn off sync (i.e do not sync the write ahead log for strong durability) in our experiment for better performance. For MCF-KV, the default value of each parameter is shown in Table 3.

We evaluate the performance of MCF-KV and compare with LevelDB, SLM-DB, and NoveLSM. LevelDB represents traditional LSM based KV store engine of a DRAM-SSD hierarchy. NoveLSM is the state-of-the-art hybrid storage scheme based on LevelDB, which combines DRAM, PM, and SSD. SLM-DB is the most similar with MCF-KV in structure, from which we can tell how the differences in indices influence the overall performance. We do not include uTree in our evaluation as its architecture, which consists only of DRAM and PM, is different from ours.

7.2 Microbenchmark on Indices

This section presents the efficiency of the proposed MCF index, including the multi-Cuckoo filters and the auxiliary B+-tree. We compare the index against the traditional bloom filter in LevelDB and the B+-tree index in SLM-DB. We also compare with two variants of MCF index, one with merely a single Cuckoo filter and the other with no B+-tree. To make the evaluation more fair, We implement a persistent version of bloom filter for the test. We focus on both write and read performance of these indices. Besides, we also pay attention to the space efficiency and false positive rate of these indices when we perform write and read operations on them. All experiments are run with one single thread with no actual flash I/O, which enables us to detect the difference among different indices more precisely.

In the experiment, we introduce the concept of update rate. We use parameter d to represent the update rate of the key set. For example, a d value of 0.1 indicates that we will select 10% of the key in the key set to update, which means these keys to be updated will trigger hash collision in these hash-based indices. We later show how the update rate influences the performance of these hash base indices. To evaluate the index performance under production environment, we also use the YCSB workload to reproduce real-world scenarios.

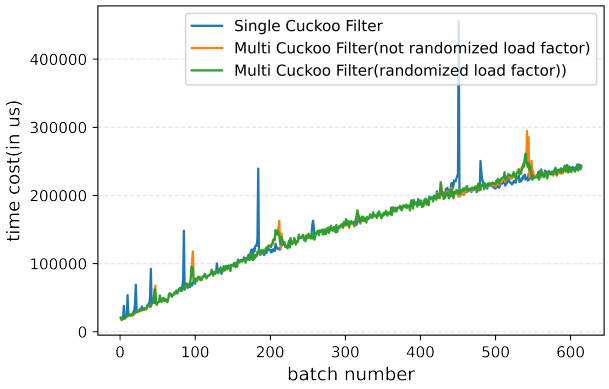


Figure 7: Time cost of every batch insertion of Single Cuckoo filter and MCF-Index with no B+tree

Table 4: Average time cost (in microseconds) of insertion operation under different update ratio

Insertion Cost(usec)	Update rate					
	0	0.1	0.2	0.4	0.8	YCSB
Single Cuckoo filter	0.99	7.94	12.31	22.22	43.11	36.23
MCF (no B+-tree)	0.92	7.72	12.08	22.84	42.29	35.98
B+-tree Index	7.63	7.50	7.42	7.22	6.57	7.10
Bloom filters (PM)	2.02	2.11	2.10	2.10	2.09	2.13
MCF Index	1.07	1.39	1.41	1.60	2.43	2.06

Write Performance. In the write performance test, we insert 125 million pre-generated random 64-bit entries into each index in an ascending order. For the Cuckoo filters (including MCF and single Cuckoo), we carefully design the initial capacity of Cuckoo filter to make it rehash once during the insertion process to ensure a fair evaluation. For the single Cuckoo filter and MCF Index without B+-tree, we simulate the operation of checking keys that cause hash collision by imposing a 50μs latency for each hash collision occurring in the index.

Table 4 shows the insertion latency of different indices under different update ratios. As we can see, MCF Index is faster than all other indices in all conditions except for bloom filter (PM) under 0.8 update rate. This is because in this exceptional case MCF-KV nearly degrades to a B+-tree index. Single Cuckoo filter has even better performance compared to MCF-index with zero update rate, but its insertion cost grows drastically when the update ratio increases because it is not able to handle hash collision without extra I/O. On the other hand, MCF index is faster than B+-tree index since the latter needs more CPU time for finding the target leaf node and performing the extra re-balancing operation.

Moreover, we carry out a test to study the long tail latency of write operations on multiple Cuckoo filters. Specifically, we insert 100MB of 64-bit random keys into MCF-index (no B+-tree) and single Cuckoo filter as mentioned above, but this time with a smaller initial capacity for the filter so that the filter would rehash for several times during the insertion process.

Table 5: Space cost and False Positive Rate of different indices

Index type	Space cost(bits/key)	FP rate
Single Cuckoo filter	35.52	0.1%
MCF (no B+-tree)	33.97	0.1%
Bloom filter	1.41	1.521%
B+-tree index	110.69	\
MCF index	40.24	\

Figure 7 shows that MCF-index with randomized load factor outperforms normal MCF-index and single Cuckoo filter in long-tail latency during the bulk writing operations. This improvement mainly owes to the increased number of filters (and thus more balanced cost for rehashing). With multiple filters and different load factor of each filter, MCF is able to amortize the total cost of rehashing a single filter into many filters in different period, which contributes to a clear decrease in the long-tail latency of write operations.

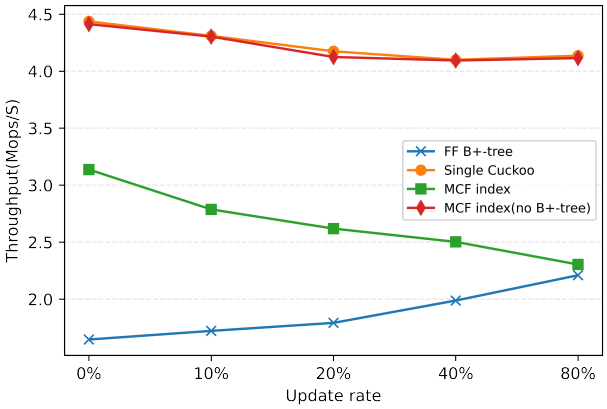


Figure 8: Read Throughput of different indices

Read Performance. To measure the read performance of each index, we execute 6.5 million query operations on each index and report the throughput of the indices being studied. As bloom filter only helps to determine if a key exists in a specific file instead of telling the exact file ID, we remove bloom from the read performance test to make the evaluation more rational. MCF index maintains a B+-tree index to better handle the hash collision, which to some extend sacrifices the read performance. As shown in Figure 8, MCF index is slower than single Cuckoo due to the introduction of the B+-tree and extra software stack. With the increase of update rate, the read performance of MCF index decreases because the B+-tree becomes larger. Different from MCF index, the throughput of FF B+-tree increases with higher update rate. This is because while MCF index keeps most keys in Cuckoo filters and higher update rate means larger B+-tree, FF B+-tree performs in-place updates on identical keys, resulting in a smaller B+-tree and better read performance.

False Positive Rate & Space Cost. As shown in Table 5, Bloom filter reaches minimal space cost, but it fails to provide good false positive rate (FPR), which is responsible for the long tail latency in read. B+-tree index does not have the problem of FPR as it stores the full information of all keys. However, it consumes on average 85× the space of the Bloom filter and 3× the space of Cuckoo filter per key. Single Cuckoo filter and MCF index both reach good performance in space cost and FPR, but the former achieves this at great expenses (i.e. long latency to deal with hash collision).

7.3 Overall Performance Evaluation

In this section, we evaluate the overall performance of the four aforementioned KV stores with *db_bench*, a micro-benchmark for LevelDB, and the widely used YCSB macro-benchmark.

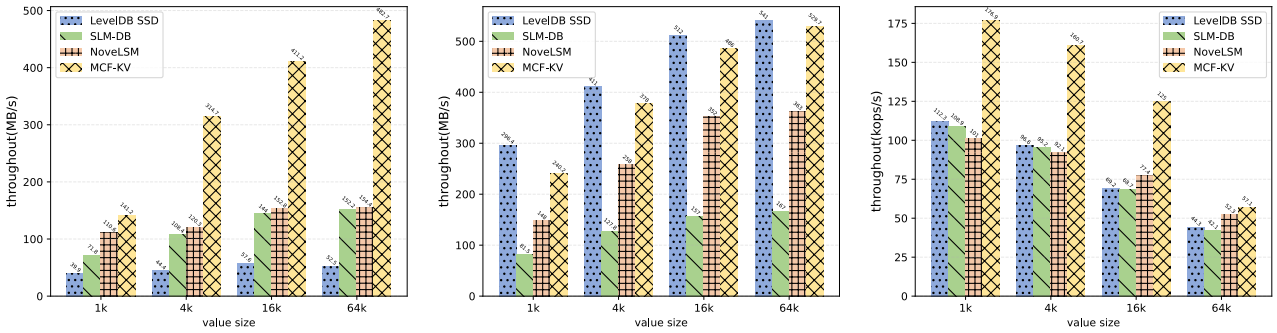
7.3.1 Results with Microbenchmark. Figure 9 shows the throughput with MCF-KV for random write, sequential write and random read. We evaluate these workloads with different value size (1KB, 4KB, 16KB and 64KB). For random read workload, we first run a random write workload to create the database and do the read work until the compaction process is done. Details of the experiment are reported in the following.

Write Performance. For random write, we insert 80GB data in a uniformly distributed random order. Figure 9(a) demonstrates the random write performance of different KV stores with different value size. MCF-KV provides about $3\times - 10\times$ higher throughput than LevelDB, which is mainly derived from reducing the data size written to disk in major compaction. MCF-KV's throughput improvement over SLM-DB ranges from 2× to 3×, this is mainly caused by the fast and efficient MCF index in PM. Although SLM-DB uses a background thread to build the B+-tree index, compaction thread still needs to wait for the finish of former batch of insertion in B+-tree to continue later flush and compaction. Therefore, a comparatively slow index leads to a lower throughput of the whole system. When compared to NoveLSM, MCF-KV provides 1.3× to 3× higher throughput because NoveLSM only gains limited benefits from PM device but leaves the problem of high WA and write stall unaddressed.

We evaluate sequential write performance by inserting 80GB KV items with different value sizes in a sequential order. As shown in Figure 9(b), the throughput of sequential write is higher than random write in every KV store as it causes no major compaction. LevelDB SSD performs the best as the DRAM-SSD architecture is theoretically the fastest without compaction. MCF-KV outperforms SLM-DB and NoveLSM as it utilizes the PM devices more efficiently.

Write amplification. Figure 10 shows the total amount of data written to disk by various KV stores under random write workload of 80GB data. The results are normalized by LevelDB. NoveLSM displays high WA since it incurs significantly higher costs in $L_0 - L_1$ compaction. On the contrary, SLM-DB and MCF-KV reduce the average amount of data written to disk by 60%-70% compared to LevelDB. This is because both KV stores employ single-level SSTable organization and carefully design their compaction procedures to optimize the write process. MCF-KV outperforms SLM-DB because compaction is triggered more frequently in SLM-DB to reach better scan performance (which would be discussed in section 7.3.2).

Read Performance. Random read performance is evaluated by reading 10% of the KV items randomly from an existing 80GB database. As shown in Figure 9(c), MCF-KV outperforms other KV stores in all situations. The improvement of random read performance mainly owes to the efficient lookup in PM indices. With the value size increases, the time of reading from the disk



(a) random write (b) sequential write (c) random read

Figure 9: Microbenchmark results

becomes more dominant in read operations, leading to reduction in performance gap from ~ 60% to ~ 20%. Besides, for SLM-DB, although it employs a B+-tree index to speed up read process, the time spent searching for target in PM increases as the B+-tree grows, which explains its lower throughput than LevelDB and NoveLSM in 1*KB* and 4*KB* value size. Different from SLM-DB, MCF-KV only places keys that incur hash collisions in the B+-tree, resulting in a smaller B+-tree and contributing to stable lookup performance.

7.3.2 Results with YCSB. YCSB[15] is a widely used macro-benchmark suit released by Yahoo!. It consists of six workload that capture different real-world scenarios. During the evaluation, we first write an 80GB data set with fixed 4KB values to load the database (marked as workload a). Then we tested four different workload patterns with one million items from YCSB respectively: 1) The first workload launches one-million point query on randomly selected items (marked as b). 2) The second workload has 50% random reads and 50% updates (marked as c). 3) The third one consists of a 50% random reads and 50% read-and-update operations (marked as d). 4) The fourth workload issues a sorted scan to fetch keys from a starting item (marked as e).

Throughput. As shown in Figure 11, MCF-KV is 2.1× to 4.9× faster than other KV stores in terms of insertion speed. With the help of MCF index, MCF-KV also achieves 1.4× to 2.2× higher

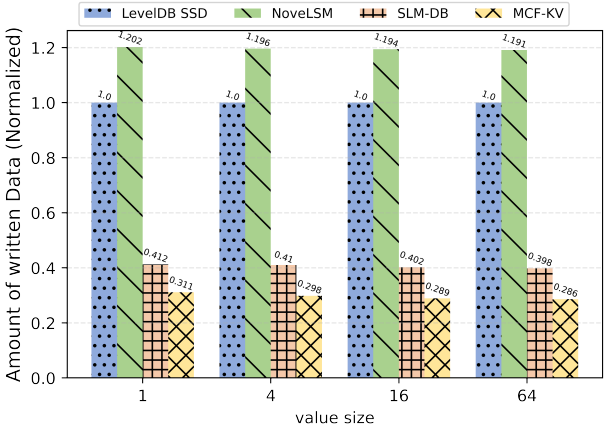


Figure 10: Total amount of written data (normalized)

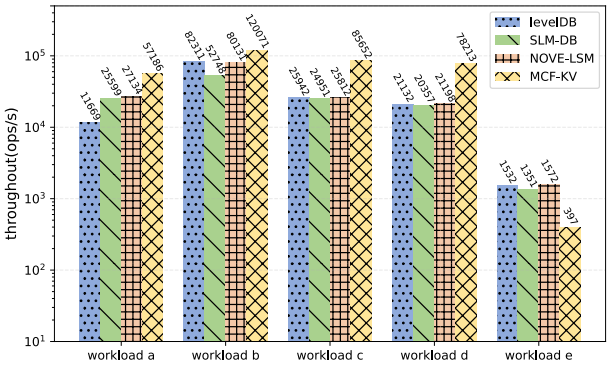


Figure 11: YCSB results. The y-axis shows the throughput of different KV store under different workload. Log-scale is used on y-axis.

point query performance. Moreover, MCF-KV handles update operation with the auxiliary B+-tree to avoid extra I/O and reaches about 3.5× improvement. However, MCF-KV is slower than the other KV stores in workload e, which issues rounds of scans to fetch the data. This is because MCF index is unable to offer valuable information to speed up the sequential read due to its hash-based nature. If a workload is dominated by scan operations, MCF-KV can be configured to launch more *pick-up process* in compaction, which improves the data locality at the cost of write speed for better scan performance.

Read Latency. We specifically focus on the tail latency of read operation as it is crucial for optimizing foreground query experiences. We evaluate the latency with YCSB-B. As shown in Table 6, MCF-KV achieves a 99.9% percentile latency of 19.27us, which is much better than other baselines.

Table 6: Read latency on different KV stores

KV-Store	Avg.	90%	99%	99.9%
LevelDB	10.35	11.27	19.36	44.28
NoveLSM	11.26	13.87	22.03	51.24
SLM-DB	13.14	19.25	19.25	19.25
MCF-KV	6.22	6.22	10.24	10.24

8 CONCLUSION

In this paper, we presented MCF-KV, a key-value store that utilizes an extra PM layer to reach high random write and point query performance. By leveraging persistent multiple Cuckoo filters and single-level LSM-tree, MCF-KV reduces write amplification and achieves high write throughput and stable query performance. Additionally, MCF-KV eliminates hash collision in hash-based Cuckoo filter with an auxiliary B+-tree index, which avoids extra I/O and improves performance. Furthermore, MCF-KV introduces an overlap-based compaction scheme to ensure some extent of data locality in the single-level LSM-tree. Evaluation results demonstrated that MCF-KV outperforms LevelDB, SLM-DB and NoveLSM in most scenarios. The MCF-index, as a general indexing technique, holds potential for application in various key-value stores. We plan to explore this potential in our future work.

ACKNOWLEDGMENTS

This work is supported by the National Key R&D Program of China (No.2022YFB3304100) and Fundamental Research Funds for the Central Universities. The authors are supported by the State Key Laboratory of Blockchain and Data Security, and Key Lab of Intelligent Computing Based Big Data of Zhejiang Province.

A COMPACTION FILE PICK UP

This appendix describes in detail how MCF-KV picks up files for compaction, in addition to Section 4. As mentioned before, MCF-KV selects seed files as good candidates for compaction and builds for every seed file s_i a heap structure h_i sorted by overlap ratio(OR).

Specifically, each heap h_i contains tuples as $(f, OR(f, s_i))$, sorted in descending order of $OR(\cdot)$, where f is the ID of a file, $OR(f, s_i)$ is the *overlap ratio* between file f and seed s_i. The overlap ratio is defined as

$$OR(f, s_i) = \begin{cases} \frac{MIN(max(f), max(s_i)) - MAX(min(f), min(s_i))}{MAX(max(f), max(s_i)) - MIN(min(f), min(s_i))} \\ \\ 0 \qquad \text{if } f \text{ does not overlap } s_i \end{cases} \tag{1}$$

where $min(x)$ and $max(x)$ indicate the minimal and maximal keys of file x, which can be found in the metadata for x.

The allocation of a file (to any one of the heaps) aims at producing the maximum overlap in key space. For each file f, we compute its OR with all the m seeds. Then file f is allocated to the heap of the seed producing maximum OR. Formally, tuple $(f, OR(f, s_i))$ is inserted to the heap of seed

$$\arg\max_i OR(f, s_i) \quad \text{where } i = 1, \ldots, m.$$

Note that the seeds are also disk files, and are therefore also inserted to the heaps. In the end, each seed s_i is hopefully associated with a sorted list of files which maximally overlap s_i.

REFERENCES

[1] 2019. *PMDK*. https://github.com/pmem/pmdk.
[2] 2020. *An empirical guide to the behavior and use of scalable persistent memory.*
[3] 2021. *LevelDB*. https://github.com/google/leveldb.
[4] 2021. *RocksDB*. https://github.com/facebook/rocksdb.
[5] R. H. Arpacidusseau and A. C. Arpacidusseau. 2015. Operating systems: Three easy pieces. (2015).
[6] Meenakshi Sundaram Bhaskaran, Jian Xu, and Steven Swanson. 2014. Bankshot: Caching slow storage in fast non-volatile memory. *ACM SIGOPS Operating Systems Review* 48, 1 (2014), 73–81.
[7] Matias Bjrling, N. Dayan, L. Bouganim, and P. Bonnet. 2013. The Necessary Death of the Block Device Interface. In *CIDR 2013*.
[8] Alex D Breslow and Nuwan S Jayasena. 2018. Morton filters: faster, space-efficient cuckoo filters via biasing, compression, and decoupled logical sparsity. *Proceedings of the VLDB Endowment* 11, 9 (2018), 1041–1055.
[9] Zhichao Cao, Siying Dong, Sagar Vemuri, and David HC Du. 2020. Characterizing, modeling, and benchmarking {RocksDB}{Key-Value} workloads at facebook. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*. 209–223.
[10] A. M. Caulfield, A. De, J. Coburn, T. I. Mollow, and S. Swanson. 2010. Moneta: A High-Performance Storage Array Architecture for Next-Generation, Non-volatile Memories. In *IEEE/ACM International Symposium on Microarchitecture*.
[11] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. 2008. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)* 26, 2 (2008), 1–26.
[12] Youmin Chen, Youyou Lu, Kedong Fang, Qing Wang, and Jiwu Shu. 2020. uTree: a persistent B+-tree with low tail latency. *Proceedings of the VLDB Endowment* 13, 12 (2020), 2634–2648.
[13] Joel Coburn, Adrian M Caulfield, Ameen Akel, Laura M Grupp, Rajesh K Gupta, Ranjit Jhala, and Steven Swanson. 2011. NV-Heaps: Making persistent objects fast and safe with next-generation, non-volatile memories. *ACM SIGARCH Computer Architecture News* 39, 1 (2011), 105–118.
[14] N. Cohen, D. T. Aksun, H. Avni, and J. R. Larus. 2019. Fine-Grain Checkpointing with In-Cache-Line Logging.
[15] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. 2010. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM Symposium on Cloud Computing, SoCC 2010, Indianapolis, Indiana, USA, June 10-11, 2010*.
[16] Niv Dayan, Martin Kjær Svendsen, Matias Bjorling, Philippe Bonnet, and Luc Bouganim. 2014. EagleTree: Exploring the design space of SSD-based algorithms. *arXiv preprint arXiv:1401.6360* (2014).
[17] Niv Dayan and Moshe Twitto. 2021. Chucky: A Succinct Cuckoo Filter for LSM-Tree. In *Proceedings of the 2021 International Conference on Management of Data*. 365–378.
[18] Subramanya R Dulloor, Sanjay Kumar, Anil Keshavamurthy, Philip Lantz, Dheeraj Reddy, Rajesh Sankaran, and Jeff Jackson. 2014. System software for persistent memory. In *Proceedings of the Ninth European Conference on Computer Systems*. 1–15.
[19] Assaf Eisenman, Darryl Gardner, Islam AbdelRahman, Jens Axboe, Siying Dong, Kim Hazelwood, Chris Petersen, Asaf Cidon, and Sachin Katti. 2018. Reducing DRAM footprint with NVM in Facebook. In *Proceedings of the Thirteenth EuroSys Conference*. 1–13.
[20] Bin Fan, Dave G Andersen, Michael Kaminsky, and Michael D Mitzenmacher. 2014. Cuckoo filter: Practically better than bloom. In *Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies*. 75–88.
[21] Thomas Mueller Graf and Daniel Lemire. 2020. Xor filters: Faster and smaller than bloom and cuckoo filters. *Journal of Experimental Algorithmics (JEA)* 25 (2020), 1–16.
[22] Deukyeon Hwang, Wook-Hee Kim, Youjip Won, and Beomseok Nam. 2018. Endurable transient inconsistency in {Byte-Addressable} persistent {B+-Tree}. In *16th USENIX Conference on File and Storage Technologies (FAST 18)*. 187–200.
[23] Junsu Im, Jinwook Bae, Chanwoo Chung, Sungjin Lee, et al. 2020. {PinK}: High-speed In-storage Key-value Store with Bounded Tails. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. 173–187.
[24] Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amirsaman Memaripour, Yun Joon Soh, Zixuan Wang, Yi Xu, Subramanya R Dulloor, et al. 2019. Basic performance measurements of the intel optane DC persistent memory module. *arXiv preprint arXiv:1903.05714* (2019).
[25] HV Jagadish, PPS Narayan, Sridhar Seshadri, S Sudarshan, and Rama Kanneganti. 1997. Incremental organization for data recording and warehousing. In *VLDB*. 16–25.
[26] Olzhas Kaiyrakhmet, Songyi Lee, Beomseok Nam, Sam H Noh, and Young-ri Choi. 2019. SLM-DB: single-level key-value store with persistent memory. In *17th USENIX Conference on File and Storage Technologies (FAST)*. 191–205.
[27] Sudarsun Kannan, Nitish Bhat, Ada Gavrilovska, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. 2018. Redesigning LSMs for nonvolatile memory with NoveLSM. In *2018 USENIX Annual Technical Conference (USENIX/ATC)*. 993–1005.
[28] Sudarsun Kannan, Nitish Bhat, Ada Gavrilovska, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. 2018. Redesigning {LSMs} for Nonvolatile Memory with {NoveLSM}. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. 993–1005.
[29] Takayuki Kawahara, Kenchi Ito, Riichiro Takemura, and Hideo Ohno. 2012. Spin-transfer torque RAM technology: Review and prospect. *Microelectronics Reliability* 52, 4 (2012), 613–627.
[30] Avinash Lakshman and Prashant Malik. 2010. Cassandra: a decentralized structured storage system. *ACM SIGOPS operating systems review* 44, 2 (2010), 35–40.
[31] Baptiste Lepers, Oana Balmau, Karan Gupta, and Willy Zwaenepoel. 2019. Kvell: the design and implementation of a fast persistent key-value store. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. 447–461.

[32] Jianhong Li, Andrew Pavlo, and Siying Dong. 2017. *NVM-rocks: RocksDB on non-volatile memory systems.*

[33] Lanyue Lu, Thanumalayan Sankaranarayana Pillai, Hariharan Gopalakrishnan, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. 2017. Wisckey: Separating keys from values in ssd-conscious storage. *ACM Transactions on Storage (TOS)* 13, 1 (2017), 1–28.

[34] Leonardo Marmol, Swaminathan Sundararaman, Nisha Talagala, and Raju Rangaswami. 2015. {NVMKV}: A Scalable, Lightweight,{FTL-aware}{Key-Value} Store. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*. 207–219.

[35] Yoshinori Matsunobu, Siying Dong, and Herman Lee. 2020. MyRocks: LSM-tree database storage engine serving Facebook's social graph. *Proceedings of the VLDB Endowment* 13, 12 (2020), 3217–3230.

[36] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. 1996. The log-structured merge-tree (LSM-tree). *Acta Informatica* 33, 4 (1996), 351–385.

[37] Simone Raoux, Geoffrey W Burr, Matthew J Breitwisch, Charles T Rettner, Y-C Chen, Robert M Shelby, Martin Salinga, Daniel Krebs, S-H Chen, H-L Lung, et al. 2008. Phase-change random access memory: A scalable technology. *IBM Journal of Research and Development* 52, 4.5 (2008), 465–479.

[38] Kai Ren, Qing Zheng, Joy Arulraj, and Garth Gibson. 2017. SlimDB: A space-efficient key-value storage engine for semi-sorted data. *Proceedings of the VLDB Endowment* 10, 13 (2017), 2037–2048.

[39] Sanam Shahla Rizvi and Tae-Sun Chung. 2010. Flash SSD vs HDD: High performance oriented modern embedded and multimedia storage systems. In *2010 2nd International Conference on Computer Engineering and Technology*, Vol. 7. IEEE, 297–299.

[40] Xingbo Wu, Yuehai Xu, Zili Shao, and Song Jiang. 2015. {LSM-trie}: An {LSM-tree-based}{Ultra-Large}{Key-Value} Store for Small Data Items. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*. 71–82.

[41] Yuhan Wu, Zirui Liu, Xiang Yu, Jie Gui, Haochen Gan, Yuhao Han, Tao Li, Ori Rottenstreich, and Tong Yang. 2021. Mapembed: Perfect hashing with high load factor and fast update. In *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining*. 1863–1872.

[42] Ting Yao, Yiwen Zhang, Jiguang Wan, Qiu Cui, Liu Tang, Hong Jiang, Changsheng Xie, and Xubin He. 2020. {MatrixKV}: Reducing Write Stalls and Write Amplification in {LSM-tree} Based {KV} Stores with Matrix Container in {NVM}. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. 17–31.