SlimDB: A Space-Efficient Key-Value Storage Engine For Semi-Sorted Data

Kai Ren, Qing Zheng, Joy Arulraj, Garth Gibson
Carnegie Mellon University
{kair,zhengq,jarulraj,garth}@cs.cmu.edu

Abstract

Modern key-value stores often use write-optimized indexes and compact in-memory indexes to speed up read and write performance. One popular write-optimized index is the Log-structured merge-tree (LSM-tree) which provides indexed access to write-intensive data. It has been increasingly used as a storage backbone for many services, including file system metadata management, graph processing engines, and machine learning feature storage engines. Existing LSM-tree implementations often exhibit high write amplifications caused by compaction, and lack optimizations to maximize read performance on solid-state disks. The goal of this paper is to explore techniques that leverage common workload characteristics shared by many systems using key-value stores to reduce the read/write amplification overhead typically associated with general-purpose LSM-tree implementations. Our experiments show that by applying these design techniques, our new implementation of a key-value store, SlimDB, can be two to three times faster, use less memory to cache metadata indices, and show lower tail latency in read operations compared to popular LSM-tree implementations such as LevelDB and RocksDB.

1. INTRODUCTION

Key-value stores have become important underpinnings of modern storage systems. Advantages provided by key-value stores include their efficient implementations, which are thin enough to provide the performance levels required by many highly demanding data-intensive applications. Many companies have built in-house key-value stores as a critical building block for their services (e.g., RocksDB [29] at Facebook, LevelDB [20] and BigTable [7] at Google, and Dynamo [13] at Amazon). To optimize for write-intensive workloads, most of these distributed and local key value stores are based on Log-Structured Merge-Tree (LSM-tree) [26]. The main advantage of LSM-trees over other external indexes (such as B-trees) is that they use extensive buffering to maintain sequential access patterns for writes. Small updates on B-trees may involve many random writes, making them inefficient on storage devices.

However, there are several workload trends that are challenging LSM-tree implementations for high performance and high scalability [33]. First, small entries are widespread, and total capacity requirements for individual key-value stores is increasing. For example, Facebook reported that 90% of its Memcached pools store key-value entries whose values are smaller than 500 bytes [2]. For a key-value store of a given capacity, a smaller entry size means that more metadata is needed, which includes indexes for locating entries in a data block and filters for determining keys' existence in the block [5]. The increasing demand on key-value store's capacity makes it an economical choice to use fewer servers, each of which has multi-terabytes hard disks or SSDs. Therefore, both factors create memory constraints for metadata of key-value stores.

Second, the key orders used by many applications are very diverse. Many existing key-value stores exclusively support either hash order or sorted order. However, we observed that keys do not need to be strictly sorted in a wide range of applications [17, 21, 30, 28, 1]. For these applications, the primary key used in the key-value store can be divided into two fragments: a prefix x and a suffix y. If range queries only need to iterate through all the keys that share the same prefix x, without any ordering requirement on y, we define the workload as **Semi-Sorted**. Semi-sorted order is stronger than hash order because semi-sorted scans do not fetch all data. Notable examples of applications that use semi-sorted keys include:

- *Feature storage for recommendation systems:* Recommendation systems need to fetch features from a large collection of entities such as users, news stories and websites, and feed these features into a machine learning algorithm to generate a prediction [17, 21]. The key schema used for feature storage can be in the format of a pair of ids like (entity id, feature id).
- *File system metadata management:* The key schema used in some file systems [30, 28] has two parts: (parent directory's inode id, hash of file name). Using this schema, a `readdir` operation in the file system only needs to list the entries that have the same parent directory's inode id as the prefix.
- *Graph-based systems:* Graph edges can be represented as a pair of nodes (source node, destination node) [1].

This work is licensed under the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nc-nd/4.0/. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org.
Proceedings of the VLDB Endowment, Vol. 10, No. 13
Copyright 2017 VLDB Endowment 2150-8097/17/08.

Many graph systems list all neighbors of a node without ordering restrictions.

Third, many workloads require key-value stores to be efficient for both reads and writes. Yahoo's research paper shows that the ratio of read and write requests in its typical low-latency workloads has shifted from anywhere between 2 and 9 to around 1 [31]. Moreover, most write operations performed in many applications fall into one category called **Non-Blind Writes**: these writes first perform a read operation on the same key before inserting or updating a key. For example, all write operations in POSIX file system metadata workloads are non-blind writes such that the file creation operation needs to first check the existence of a file according to the POSIX standard. In the feature storage for recommendation systems, counters that summarize user behaviors such as clicks and impressions are often updated only by operations that increase or decrease the prior value. In such workloads, a better balance between read and write performance for key-value stores is more desired.

In this paper, we propose a space-efficient sorted store, SlimDB, that can be used as the core storage engine for applications with semi-sorted keys. SlimDB integrates three novel techniques that exploit the above workload characteristics for optimizing reads, writes, and the memory footprint: 1) a redesign of data block indexes with space-efficient data structures specialized for semi-sorted data; 2) a novel membership filter that bounds the worst-case read latency in multi-level log-structured key-value stores; and 3) an analytical model for generating layouts of in-memory indexes and on-disk data storage with desired read and write amplifications. Experiments show that SlimDB is two to three times faster for key-value workloads, while taking up less space for caching indexes and filters, and exhibiting better tail latency in read operations, relative to key-value stores using a general-purpose LSM-tree implementation including LevelDB and RocksDB.

2. BACKGROUND AND MOTIVATION

This section discusses the basics of the LSM-tree and its variants, as well as opportunities for further improvement.

2.1 LSM-Tree and LevelDB Implementation

An LSM-tree contains multiple append-only sorted tables, each of which is created by sequential writes, and often implemented as SSTables [7, 20]. As shown in Figure 1, these SSTables are organized into multiple levels based on the time when entries are inserted, that is, if k_i is a key found in level i matches k_j, $(j > i)$, then k_i was written after k_j.

The LSM-tree exploits the relatively fast sequential write speed of modern storage devices. On hard disk and flash drives, sequential writes are an order of magnitude faster than random writes. By using an in-memory buffer and multi-level structure, the LSM-tree stages and transforms random writes into sequential I/O. A new entry is first inserted into the in-memory buffer (and is also logged to the disk for crash recovery). When the buffer limit is reached, the entry and other changes cached in the buffer are spilled into the disk to generate a SSTable in Level 0. The new entry is then migrated over time from Level i to Level $i+1$ by a merge process called "compaction".

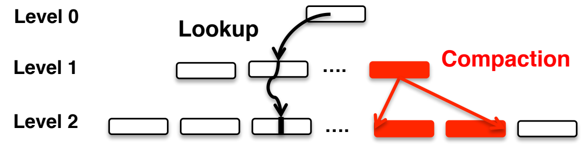

Figure 1: *Illustration of the level structure of LSM-tree and its compaction procedure. Each block denotes an SSTable file.*

The total size of each level follows an exponential growth pattern such that the size of Level i is r times larger than the size of Level $i-1$. Common values of r are between 8 and 16. With this exponential pattern, there are at most $O(log_r N)$ levels, where N is the total number of unique keys. To search for an entry, the LSM-tree has to search multiple levels because the entry can exist in any of these levels. In practice, searching for an entry within an uncachable large level usually requires two random disk I/Os to fetch the SSTable's index and the block that contains the entry. Thus the worst-case lookup incurs $O(log_r N)$ random I/O by accessing all levels. Modern implementations of LSM-trees usually use in-memory Bloom filters [5] to avoid unnecessary searching in some levels.

The compaction strategies that move data between levels are based upon certain criteria. In an LSM-tree, if the size limit of a level is reached, data in this level will be compacted into the next level by merge-sorting SSTables from the two levels. In the worst case, the key range of Level i overlaps the entire key range of Level $i+1$, which requires merge-sorting all of the SSTables in both levels. Since Level $i+1$ has r times more data than Level i on average, migrating an entry requires reading and writing r times more data than itself. Thus, the write amplification per insertion is $O(r log_r N)$. Since entries are transferred in batches (of size B) during compaction, the amortized I/O cost per insertion is $O(\frac{1}{B} r log_r N)$ in the worst case.

2.2 The Stepped-Merge Algorithm

Stepped-Merge is an LSM variant that uses a different organization and compaction strategy to manage SSTables [19]. The main purpose of compaction is to make room for recently inserted entries by integrating SSTables from Level i to Level $i+1$. The major source of write amplification comes from the fact that the compaction procedure has to merge-sort at least one SSTable in level i with all of the overlapping SSTables in the next level, amplifing the compaction overhead.

Based on this observation, as shown in Figure 2, Stepped-Merge divides the SSTables in each level into r sub-levels. The size limit of each level is still the same as the LSM-tree. However, when compacting SSTables in Level i, Stepped-Merge does not merge-sort SSTables in Level i with tables in Level $i+1$ as the LSM-tree does. Instead, all sub-levels in

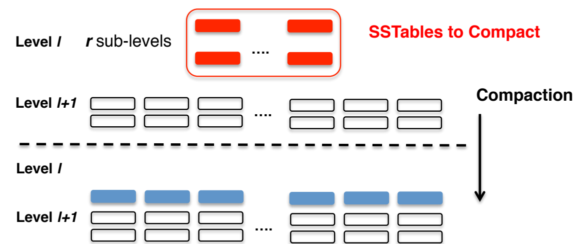

Figure 2: *Illustration of Stepped-Merge algorithm.*

Level i are r-way merge-sorted and inserted into Level $i+1$ as a new sub-level. The total amount of transferred data during merge-sorting r sub-levels is roughly the same as the total amount of data stored in these sub-levels. By doing so, the amortized cost of migrating an entry from Level i to Level $i+1$ is reduced to only two times its size. Since each long-lived, inserted entry is written at each level only once, the write amplification for an entry to reach level $i+1$ is i. Thus, the amortized I/O cost of an insertion in Stepped-Merge decreases to $O(\frac{1}{B}log_r N)$.

On the other hand, a lookup operation in Stepped-Merge has to check $rlog_r N$ sub-levels to locate a key, which costs $O(rlog_r N)$ random reads from disk in the worst case.

2.3 Optimizing Indexes and Filters

Although the stepped-merge algorithm can reduce write amplification, its read performance degrades because the algorithm has multiple overlapping sub-levels within each level for a lookup operation to read in order to find a particular entry. To avoid high read latency while maintaining low write amplification, one potential solution is to increase the effectiveness of a store's in-memory indexes and filters. These enhanced in-memory data structures can better pinpoint where entries might and will not be, and therefore can avoid unnecessary disk accesses.

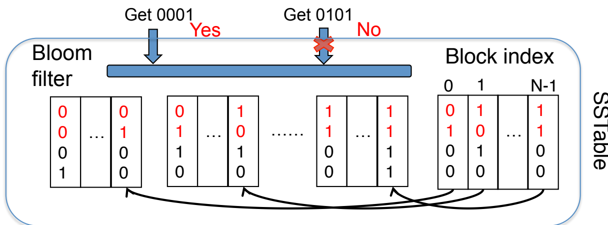

Figure 3: *Illustration of the basic index format of LevelDB's SSTable and its read path. The keys follow a semi-sorted order, so each key has two parts: the prefix (red) and the suffix (black).*

Figure 3 shows the main components of a typical LevelDB SSTable, which is the format used by LevelDB to store data. Each SSTable stores its data in sorted order across an array of data blocks. The size of each data block is configurable and is usually 4KB. In addition to these data blocks, a special index block is created that maps each key range to its data block. This index block consists of all the largest keys of every data block. Along with this index block, a Bloom filter is also used to record the existence of all the keys in the table [5]. For each lookup operation, LevelDB first checks the Bloom filter to ascertain the non-existence of a key, else it uses the index block to find the right data block. In Figure 3, for example, to lookup key "0001" the lookup process will first go through the Bloom filter and will find that the key may be in the SSTable. It then checks the corresponding index block, which will lead the lookup to data block 0. To lookup key "0101", the lookup process will be stopped by the Bloom filter as key "0101" was never inserted into this example table.

Since the Bloom filter is a probabilistic data structure with a false positive rate, a lookup operation may fetch a data block that does not contain the target key, thus adding additional read latency. On the other hand, all SSTable indexes and filters in many key-value stores are often stored in memory in a LRU cache with a fixed memory limit. Making high quality indexes and filters more compact will allow more entries can be precisely indexed and avoid loading block indexes and filters from the disk. Because the quality and the size of block indexes and filters are key to ensuring good read performance, we will show in the following sections how to improve indexes and filters by leveraging common key-value workload characteristics.

3. DESIGN AND IMPLEMENTATION

The following summarizes the characteristics of the indexes and filters used in SlimDB:

- *Three-level Block Index*: Our three-level block index replaces the original block index used in LevelDB's SSTable. This new index is specially optimized for semi-sorted data. It features a memory cost that is as small as 0.7 bits per key.
- *Multi-level Cuckoo Filter*: The multi-level cuckoo filter is a replacement of Bloom filters for the stepped-merge algorithm. When searching for a key using a multi-level cuckoo filter, the filter returns the most recent sub-level containing the target key if the key appears to exist. Similar to Bloom filters, the multi-level cuckoo filter is a probabilistic data structure which may give the wrong answer if the key does not exist. But even in the worst case, the lookup procedure will only need to access a SSTable in one sub-level in a workload with only blind writes.

Combining different data layouts and indexes gives rise to key-value stores with different read, write amplification, and memory costs. For example, we can combine a multi-level cuckoo filter with a stepped-merge algorithm. Together they can have lower write amplification than an original LSM-tree but may require more memory resources. There is no one combination that is strictly better than all other combinations. However, the multi-level structure used by many log-structured store designs allows for a flexible use of different key-value store combinations at each level [24]. As we shall show, these multi-level stores are able to leverage a mix of key-value store designs to balance read amplification, write amplification, and memory usage. Figure 4 gives an example of the multi-store design in SlimDB. Levels 0, 1 and 2 all use the data layout of the stepped-merge algorithm, with multi-level cuckoo filters and three-level block indexes. All filters and indexes are cached in memory. Level 3 and Level 4 use the data layout of the original LSM-tree, and cache three-level block indexes in memory. Further, Level 3 but not Level 4 caches Bloom filters in memory.

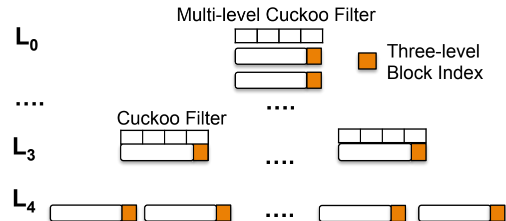

Figure 4: *The use of multi-store design in SlimDB. Filters and indexes are generally in-memory, and for a large store SSTables are mostly on disk.*

The following sections explains our novel indexes and filters. Section 4 will show how to use our proposed analytic model to automatically select basic key-value store designs for each level to meet resource and performance constraints.

3.1 Space-efficiency of an SSTable Index

In LevelDB's original SSTable format, key-value pairs are sorted and packed into data blocks. As shown in Figure 3, each SSTable file contains an index block at the end of the file that stores the full key of each data block's last entry. Without caching the block index, reading an entry from an SSTable requires two block reads: one to load the index block and the other to read the actual entry. Since the size of an SSTable data block is usually set to 4KB and the typical size of an entry in many applications (e.g, file system metadata, feature storage in recommendation system) might be smaller than 256 bytes [24], each block stores, say, at most 16 entries. As LevelDB's block index stores a full key (e.g. 16B) for each data block, the average space required to store a key might be 16B / 16 = 8 bits. The block index representation can be replaced without changing the general LSM organization and execution. Our goal is to to apply sophisticated compression schemes on the block index to trade more CPU cycles for fewer storage accesses through higher cache hit rates when fetching a random entry from an on-disk SSTable.

Different from LevelDB, which is designed for totally ordered keys, SlimDB only needs to support semi-sorted keys that consist of a prefix and a suffix. This means that the keys in a SlimDB SSTable only need to be sorted by their prefixes. This enables us to use entropy-coded tries (ECT) [24] to compress prefixes and suffixes separately in the index block. ECT can efficiently index a sorted list of fixed-sized hash keys using only 2.5 bits per entry on average. In this section, we construct a semi-sorted block index with ECT to use only 1.9 bits to index an entry to its block, which is 4X smaller than the LevelDB method.

Entropy-Encoded Trie Basics: Given an array of n distinct keys that are sorted by their hash order, an ECT data structure is able to map each input key to its rank ($\in [0, n-1]$) in the array. As shown in Figure 5, each ECT is a radix tree that stores a set of keys where each leaf node represents one key in the set and each internal node denotes the longest common prefix shared by the keys under the subtree rooted by this internal node. For each key stored, ECT only preserves the shortest partial key prefix that is sufficient to differentiate it from other keys. Although ECT can index a set of keys, it cannot check key membership, so additional data structures (such as bloom filters) are still needed to avoid false lookups.

Because all keys are hashed, ensuring a uniform distribution, a combination of Huffman coding and Elias-gamma

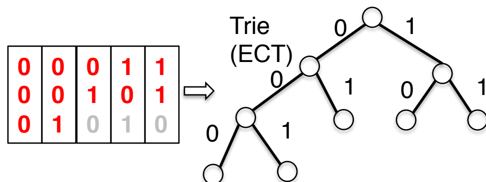

Figure 5: *ECT transforms a list of sorted key hashes into a radix tree that only keeps the shortest prefix of each key that is enough to distinguish it from other keys.*

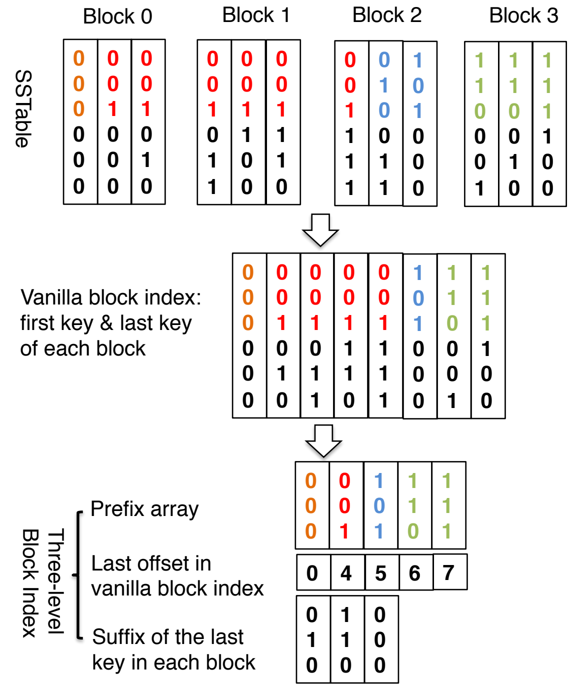

Figure 6: *An example three-level block index for an SSTable.*

coding is able to greatly compress the keys in each trie. Details of how to compress the trie are described in [24].

Three-Level Index Design: Unlike hash tables, in order to retain the semi-ordering of keys, using ECT alone is not sufficient to serve as a block index for SlimDB. In SlimDB both key fragments are hashed, so it is possible to use ECT to index each key fragment individually, leading to a two-step search procedure: first find a group of SSTable blocks that contain all keys sharing the same prefix as the sought key; then locate the specific block having the sought key from the group of blocks returned by the first step.

As shown in the example in Figure 6, the construction of three-level index is based on compressing the vanilla block index that stores the first and the last keys of all the data blocks. First, the prefix of these block keys are stored separately in a prefix array, where only one prefix is preserved and duplicated prefixes are removed. We use ECT to compress this prefix array, which constitutes the first level of our index. This level allows us to map each key, using its prefix, to its rank in the prefix array, and this rank in turn becomes the input to the second level of our three-level index, which we now describe.

The second level of our three-level index takes the rank in the prefix array and maps it to one or more SSTable data blocks that contain entries matching this specific prefix key. In this integer array each element stores the offset in the vanilla block index of the last block key containing the corresponding prefix in the prefix array. For example, the last block containing "001" is Block 2, and its offset in the vallina block index is 4. Therefore its corresponding element in the second level is 4.

Through the mappings defined by the first two levels, a lookup procedure is able to retrieve a list of potential SSTable blocks that contain the sought key's prefix. To finally locate the SSTable block whose range covers the sought key, the last step is to binary search through all potential SSTable blocks using the suffix of the last entry from each

block. Similar to the first level index, the array of suffixes of block keys sharing the same prefix can be compressed by using ECT.

To optionally speed up the lookup process without using ECT, our three-level index can store an array of partial suffixes instead: each partial suffix is the shortest unique prefix of the original suffix that distinguishes a pair of suffixes from the two adjacent SSTable blocks. For example, when searching for the key "001000", we find it must reside between Block 0 to Block 2 inclusive, based on the first two level indexes, as shown in Figure 6. To locate its block, we use its suffix "000" to complete a binary search among the array of suffixes ("010", "110") that differentiate the three candidate block groups. Since "000" is smaller than "010", "001000" can only be stored in Block 0.

Analysis of Three-Level SSTable Index: For the first level, the prefix array needs to store at most two prefixes per SSTable block, and all prefixes are hash sorted which can be used for ECT encoding. Since ECT costs 2.5 bits per prefix key on average, the first level costs no more than $2 \times 2.5 = 5$ bits per SSTable block.

For the second-level index that records, the last block's offset per prefix, we can represent it with a rank/select dictionry[18]. It first uses delta encoding to calculate the difference between two offsets. Because the sum of these deltas cannot exceed the number of blocks in the SSTable, we can then use unary coding to represent the delta as a bit vector, with no more than two bits per block in an SSTable. Optionally, to speed up searching in this array, a sum and pointer enables quick skipping one set of k deltas, this can be added to the bit vector for every k deltas. If the size of the sum and a pointer is 16 bits and $k = 32$, then building this array costs $2 + 16/k = 2.5$ bits per group.

The third-level index that records per-block last suffixes costs 2.5 bits per block on average if using ECT. If using an array of partial suffixes instead of ECT, Monte carlo simulation of all possible arrays of partial keys shows that the average length of the partial key that separates two adjacent suffixes is about 16 bits. Another 6 bits is used to record the length of each partial key. So the average cost of the faster lookup third-level index is 22 bits per block (using the array of partial keys).

Summing the average-case cost of all three index levels, the three-level SSTable consumes 10 $(5 + 2.5 + 2.5)$ bits per SSTable block using ECT on the third-level index. Using 16 key-value items per block, memory overhead is $10/16 = 0.7$ bits per key, much smaller than LevelDB's 8 bits per key. If using the array of partial keys for faster lookup in third-level index, the memory overhead is $(5 + 2.5 + 22)/16 = 1.9$ bits per key (still 4X better than LevelDB).

3.2 Multi-Level Cuckoo Filter

In-memory filters are data structures commonly used by many high performance key-value stores to efficiently test whether a given key is not found in the store before accessing the disk. Most of these filters are probablistic data structures that perform false positive accesses. One main source of long tail latency in the read path of a stepped-merge store lies in false positive answers given by Bloom filters at multiple levels when looking for a key. We propose a new filter design, called a multi-level cuckoo filter, that can limit the number of disk reads in such cases. This new filter design uses the cuckoo filter as a building block. Cuckoo filters are similar to Bloom filters but have properties like lower memory cost and fingerprint-based filtering [14]. This section introduces how the design of our multi-level cuckoo filter improves the read tail latency of key-value stores by leveraging these properties.

Cuckoo Filter Basics: A cuckoo filter extends standard cuckoo hash tables [27] to provide membership information. As shown in Figure 7, a cuckoo hash table is a linear array of key buckets where each key has two candidate buckets calculated by two independent hash functions. When looking up a key, the procedure checks both candidate buckets to see if the entry exists. An entry can be inserted into any one of two candidate buckets that is vacant. If both are full, then the procedure displaces one existing entry in either bucket and re-inserts the victim to its alternative bucket. The displacement procedure repeats until a vacant bucket is found or the maximum number of displacements is reached (e.g, hundreds of tries). In the latter case, the hash table is declared to be full, and an expansion process is executed. Although cuckoo hashing may execute a series of displacements, the amortized I/O cost of insertion operation is $O(1)$.

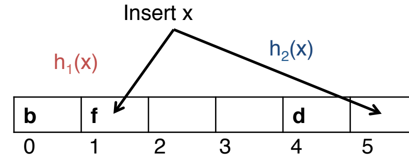

Figure 7: *Illustration of Cuckoo Hashing.*

Cuckoo hashing can achieve higher space occupancy by using more hash functions as well as extending the buckets to have more than one slot to allow several entries to co-exist. Nikolaos et. al. present an analysis of the maximum possible occupancy ratio, showing that with 2 hash functions and a bucket of size 4, the table space can be 95% filled [16].

Cuckoo hashing can be used directly to implement a membership query. But since the hash table stores the full key, it has high space overhead compared to a Bloom filter. To save space, a cuckoo filter [14] only stores a constant-sized hash fingerprint of any inserted entry instead of its original full key. This results in changes to the insertion procedure. Storing only fingerprints in the hash table prevents inserting entries using the standard cuckoo hashing approach, since it prevents the algorithm from calculating the entry's alternative position. To overcome this limitation, the cuckoo filter uses the fingerprint to calculate an entry's alternative bucket rather than the key itself. For example, the cuckoo hash indexes of the two candidate buckets of an entry x ($h_1(x)$ and $h_2(x)$) are calculated as follows:

$$h_1(x) = hash(x),$$

$$h_2(x) = h_1(x) \oplus hash(x\text{'s fingerprint})$$

Obviously, the two functions are symetric since $h_1(x) = h_2(x) \oplus hash(x\text{'s fingerprint})$. This design causes the two hash functions to be less independent of each other, therefore the collision rate in the hash table is higher than that of the standard cuckoo hashing table. However, by selecting an approriate fingerprint size f and bucket size b, it can be shown that the cuckoo filter is more space-efficient than the Bloom filter when the target false postive rate is smaller than 3% [14].

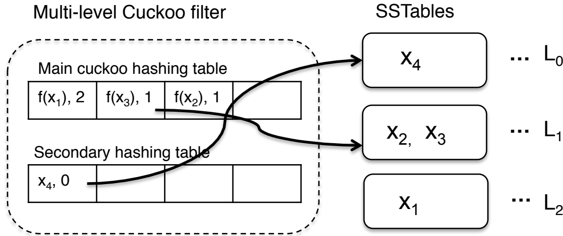

Figure 8: *Illustration of integrating cuckoo filters with multi-level indexes and a secondary table. If a key has a hash collision with some key in the primary hashing table, the key will be put into the secondary hashing table. Each lookup first checks in the secondary table and then the primary table.*

Integration with Multi-level Stores: Figure 8 depicts the integration of a multi-level cuckoo filter with multi-level key-value stores. The multi-level cuckoo filter has two separate tables: the primary table and the secondary table. The primary table is a cuckoo filter variant that stores both the fingerprint, $f(x)$, and the level number of each entry, $l(x)$. Different from the basic cuckoo filter, the level number stored in the primary table can be used to locate the sub-level in the LSM-tree in which the target entry is actually present.

The secondary table is used to bound tail latency by storing special entries with their full keys. The reason for having the secondary table is to cope with the case that multiple entries may have the same fingerprint. In such cases, the primary table would need to keep multiple copies of the same fingerprint with all the associated level numbers. To locate the level that actually contains the sought entry, it would be necessary to perform a disk read for each level associated with the conflicting fingerprint in the worst case. A straightfoward method to reduce the number of disk reads in the worst case is to avoid fingerprint conflicts in the primary table, which means that each fingerprint must be unique and can only be associated with one level number in the primary table. To maintain this property, the multi-level cuckoo filter uses the secondary table to record the full key for each entry, after the first, having a conflicting fingerprint in the primary table.

With a secondary table for conflicting entries, the procedure of looking up an entry in multi-level cuckoo filter is still straightforward, as shown in Algorithm 1. When searching for an entry, first search the secondary table to see if there is any matching full key. If found, the level number of this entry can be retrieved from the secondary table; otherwise, continue to check the primary table of the multi-level cuckoo filter. With the secondary table in memory, the worst case lookup performs at most one disk read since it is guaranteed that there is only one copy of each fingerprint in the primary table.

To maintain uniqueness of fingerprints in the primary table, the insertion procedure in multi-level cuckoo filters must follow certain rules as shown in Algorithm 2. The multi-level cuckoo filter is built along with the multi-level stores, meaning that newer entries are always inserted into newer (lower) levels. When inserting an entry, if its fingerprint already exists, then we check whether the fingerprint in the primary table is derived from the same key. If it is derived from the same key, then the entry must have a newer level number, and therefore we only update the level number associated with the key in the primary table. Otherwise, the entry is put into the secondary table due to the conflict of the fingerprint. If the fingerprint does not exist, then the entry can be safely inserted into the primary table. For example, as shown in Figure 8, when inserting x_4, it might happen that x_4's fingerprint is the same as x_1's. Thus, x_4 has to be put into the secondary table. However, if we insert x_1 with level 0 that is newer than level 2, then only its level number in the primary table needs to be updated.

Algorithm 1: lookup(key x)

```
Data: c: primary table; t: secondary table
l = t.lookup(x)
if l is not NULL then
    return l
else
    f = fingerprint(x)
    return c.lookup(f)
end
```

Algorithm 2: insert(key x, level l)

```
Data: c: primary table; t: secondary table; store:
        on-disk store
f = fingerprint(x)
l' = c.lookup(f)
if l' is not NULL then
    if store has no x in level l' then
        t.insert(x, l)
        return
    end
end
c.insert(f, l)
```

To verify whether the conflicting fingerprint comes from the same key in the primary table, it is not necessary to perform disk reads to retrieve the full key if writes are non-blind. Our strategy then is to take advantage of non-blind writes to avoid unnecessary disk traffic when possible. For example, in file system metadata workloads under the POSIX standard, all metadata write operations are non-blind, which means that a read operation will have always been performed on a key before any write of that key. The existence check operation done by prior read avoids additional disk reads needed for the multi-level cuckoo filter to verify whether the same key exists in other levels.

If blind writes do happen in the workload, the same key can be inserted into both the primary and secondary table if the insertion procedure does not read the full key from the disk. In this case, however, the number of keys stored in the secondary table will exceed reasonable space targets, so our algorithm will stop inserting new entries into the secondary table, and the multi-level cuckoo filter will exhibit similar false positives and tail latency distribution as the original Bloom filters.

Memory Footprint Analysis: While it might seem that the multi-level cuckoo filter may use $log_2(L), L \approx \log(N)$ more bits per entry compared to the traditional Bloom filters used in LevelDB, the primary table of the multi-level

cuckoo filter actually has the same memory cost even in the worst case. To see why, assume the desired false positive rate for the multi-level cuckoo filter is ϵ. For traditional methods [20] that use a Bloom filter for each SSTable, the overall false positive rate is at least $1-(1-\alpha)^L \approx L \cdot \alpha$ if the false positive rate in each level is α and they are independently distributed. In order to achieve the same false positive rate as the multi-level cuckoo filter, a Bloom filter's α must be ϵ/L. The space requirement for any Bloom filter to achieve a false positive rate α is at least $\log_2 1/\alpha$. So the overall cost of the traditional method is $log_2(1/\alpha) = log_2(1/\epsilon) + log_2(L)$ per entry, which is the same as the multi-level cuckoo filter.

The average size of the secondary table is proportional to the false positive rate of the primary table. To see why, assume that there are n elements that need to be inserted into the primary filter, and the primary filter has a false positive rate ϵ. The expected number of entries stored in the secondary table is the expected number of entries that generate false positive anwers, which is $n \times \epsilon$.

SlimDB uses a multi-level cuckoo filter with a less than 0.1% false positive rate. The size of each item stored in the secondary table is the sum of the length of the full key and the size of the level number which is $128 + 8 = 136$ bits. The secondary table increases the memory overhead by $136 \times 0.1\% = 0.136$ bit per entry, which is $0.136/16 = 0.8\%$ of the original Bloom filter's cost.

3.3 Implementation

The implementation of SlimDB is based on RocksDB [29], and has about 5000 lines of code changes. RocksDB has a modular architecture where each component of the system exports the same, basic key-value interface including the memory-index and on-disk SSTable. This allows us to easily add a new filter policy and block index into its SSTables. In SlimDB, items are sorted according to their hashed prefix and suffix. Thus, point queries do not require additonal changes to RocksDB other than in the filters and block indexes. For prefix scan (e.g. list all entries sharing the same prefix) with stepped-merge algorithms, SlimDB has to maintain an SSTable iterator in each sub-level, which is slower than a traditional LevelDB. Since items are sorted by hashed prefix, SlimDB cannot support scan across prefixes with one index. To support fully ordered key scans in such a workload, SlimDB could maintain another secondary index that stores all the prefixes without hashing [25].

The use of the stepped-merge algorithm in SlimDB is similar to the procedure described in LSM-trie [33]. In each sublevel, semi-sorted items are grouped into SSTables based on their hash-key range as well as the size limit of each SSTable. During each compaction, the procedure will pick all SSTables within a hash-key range from all sub-levels to do merge-sorting and put newly merged SSTables into the next level.

4. ANALYTIC MODEL FOR SELECTING INDEXES AND FILTERS

The level structure of an LSM-tree allows for flexible use of different storage layouts and in-memory indexes on a per-level basis. This section presents an analytic model that selects storage layout and in-memory indexes to achieve low memory usage while maintaining target performance.

The key idea of the analytic model is to account for the hardware resources utilized by the index structure in each level, including the memory cost and the I/O cost of all types of requests (such as positive reads, negative reads, and insertions). To unify I/O costs of different types of requests, we use the time spent on a random 4KB disk read as the basic measurement unit. For most storage devices, the cost of writing a 4KB block sequentially compared to random block read (denoted as w) is actually quite small. For example, the solid state disk used in our evaluation performs 4000 4KB random reads per second, and delivers 107 MB/sec sequential writes. The I/O time to write a 4KB block sequentially is $4/107/1024 \approx 0.0000365$ seconds. The I/O time of reading a 4KB block randomly is roughly $1/4000 \approx 0.00025$ seconds. In such cases, w equals 0.146. If a 4KB block can store B entries, then the cost of inserting an entry is w/B because insertion and compaction in the log-structure design only require sequential writes.

Table 1: *The space and disk access cost of using three types of indexes. CF means Cuckoo Filter. TL means three-level SSTable index. MLCF means multi-level cuckoo filter.*

#	Level structure	Mem. C_M	Pos. C_{PR}	Neg. C_{NR}	Writes C_W
0	LSM-Tree	0b	2	2	$\frac{rw}{B}$
1	LSM+CF	13b	2	$2f$	$\frac{rw}{B}$
2	LSM+CF+TL	15b	1	f	$\frac{rw}{B}$
3	LSM+TL	2b	1	1	$\frac{rw}{B}$
4	Stepped-Merge	0b	$r+1$	$2r$	$\frac{w}{B}$
5	SM+MLCF	15b	2	$2f$	$\frac{w}{B}$
6	SM+MLCF+TL	17b	1	f	$\frac{w}{B}$
7	SM+TL	2b	$\frac{r+1}{2}$	r	$\frac{w}{B}$

Table 1 summarizes the costs of using different combinations of indexes and data layout on a per-level basis. For simplicity, our model assumes that all the SSTables within a level use the same types of index and filter, and that key queries follow a uniform distribution. For each level, if that level follows the design of an LSM-tree and has only one sub-level, then it is labeled as an LSM-tree style data layout. Otherwise, for any level having multiple sub-levels, it is labeled as a Stepped-Merge (SM) style data layout. Without any in-memory indexes, the costs of the two styles are calculated as in Section 2. When equipped with only a cuckoo filter or a multi-level cuckoo filter, the cost of a negative read (a read that does not find the sought key) is $2f$, where f is the false-positive rate of the filter. By caching a three-level SSTable index additionally, the average cost of retrieving an entry is reduced to f.

Once there is a cost model, then the index selection problem becomes an optimization problem whose goal is to minimize the I/O cost under memory constraints. Assume there are $l+1$ levels, and the number of entries in level i is N_i. With N_0 and the growth factor r as input parameters, the size of each level can be calculated as $N_i = N_0 \cdot r^i$. The total number of entries in the store is $N = \sum N_i$. The type of index used by level i is denoted as t_i. For the index of type t_i, its memory cost in bits per entry is denoted as $C_M[t_i]$. $C_{PR}[t_i]$, $C_{NR}[t_i]$, and $C_W[t_i]$ denote the cost of a positive read, the cost of a negative read and the cost of write in disk access per operation. We also assume that the ratio of different operations in the workload are known beforehand: the ratio of positive reads, negative reads, and writes in the workload are r_{PR}, r_{NR}, and r_W, respectively. By choosing different types of indexes for each level, the goal is to meet a memory constraint and reduce the overall I/O cost.

The overall average cost for each type of operations can be summarized as below:

$$S_{PR} = \sum_{0 \le i < l} \frac{N_i}{N} \times (C_{PR}[t_i] + \sum_{0 \le j < i} C_{NR}[t_j]),$$

$$S_{NR} = \sum_{0 \le i < l} C_{NR}[t_i], \quad S_W = \sum_{0 \le i < l} \frac{N_i}{N} \times \sum_{0 \le j \le i} C_W[t_j]$$

Therefore, the average I/O cost of a random operation within a particular workload is:

$$C = r_{PR} \times S_{PR} + r_{NR} \times S_{NR} + r_W \times S_W$$

With a memory budget of M bytes, the constraints for this optimization problem are:

$$\sum N_i * C_M[t_i] \le M, 0 \le t_i \le 7$$

By using a heuristic search, the optimal value can be easily found for the above optimization problem.

Figure 9 shows the average cost of a key-value operation under different file system metadata workloads and memory constraints. In this figure, $l = 5$, $r = 8$, and $N_0 = 2^{17}$. So the key-value store has about a half billion entries in total. The figure shows four file system metadata workloads as an example: file creation in an empty file system (creation), updating inode attributes (update), querying inode attributes (stat), and a mix of reads and writes (mix). The ratio of key-value operations is calculated by designating file metadata operations into read and write operations. For example, since the creation workload creates files from an empty file system, all existence checks are negative reads, which means that $r_{NR} = 0.5$ and $r_W = 0.5$. From Figure 9, we can see that the average cost gradually decreases as the memory budget increases. For a creation workload, when the memory budget allows the key-value store to cache a filter at each level, the creation cost reaches the lowest point. For other workloads dominated by positive reads, one disk read is the lower bound. When the memory budget is between 256MB and 900MB, the four workloads use the same layout: the first three levels use a stepped-merge layout, three-level SSTable indexes and multi-level cuckoo filters; level 3 has only one sub-level with a three-level index and cuckoo filter; and level 4 has one sub-level with only the three-level index. The multi-store layout is illustrated in Figure 4. Traversing down the level hierarchy, the memory cost of the index decreases from the multi-level cuckoo filter to only caching block index.

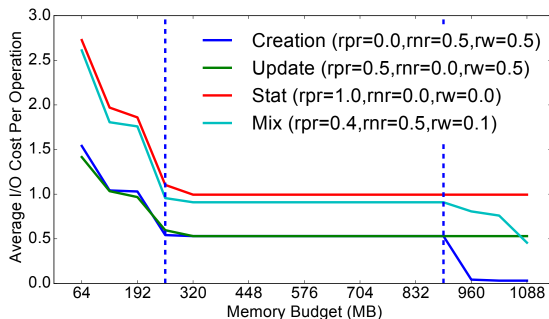

Figure 9: *The per-operation cost estimated by the model.*

5. EVALUATION

Using macro and micro-benchmarks, we evaluate SlimDB's overall performance and explore how its system design and algorithms contribute to meeting its goals. We specifically examine (1) the performance of SlimDB's in-memory indexing data structures in isolation; and (2) an end-to-end evaluation of SlimDB's throughput, memory overhead, and latency.

5.1 Evaluation System

All our experiments are evaluated on a Linux desktop configured as is listed in Table 2.

Table 2: *Hardware configuration for experiments.*

Linux	Ubuntu 12.10, Kernel 3.5.0 64-bit
CPU	Intel Core 2 Quad Proc. Q9550
DRAM	8GB DDR SDRAM
SSD	128GB Intel 520 Solid State Drive Random (op/s): 4K (R), 2.5K (W) Sequential (MB/s): 245 (R), 107 (W)

We compare the performance of SlimDB against two popular LSM tree variants: LevelDB (version 1.6) [20] and its fork RocksDB (version 4.11) [29]. All key-value stores are configured to use filters at 16 bytes per key. The growth factor for all key-value stores is 8, the size limit for Level-0 is 20 SSTables, and the size of each SSTable is 32MB. In all benchmarks, we use a 16-byte key and a 256-byte value. Each key has two fragments, each fragment is a 64-bit value, and all workloads meet the semi-sorted requirement.

5.2 Microbechmark on SSTable Indexes

This section demonstrates the effectiveness of compacting the index for each SSTable. The experiments compare LevelDB's original SSTable index against our three-level index. Analysis of the experiments results focuses on two main metrics: the memory cost and lookup latency per key. It is expected that the enhanced three-level index uses less memory but incurs additional CPU costs for decompression.

Experiment Design: In the first experiment, an empty SSTable is first filled according to given key distribution for each case; the second stage of this experiment opens this newly-created SSTable, loads its index block into the memory to obtain its memory consumption, then generates 1 million random queries to measure average lookup latency. In the experiment, each SSTable only consists of a list of data blocks and a single index block excluding other types of blocks holding information such as bloom filters.

In all experiments, an SSTable has a fixed size of 32MB and each data block has a size of 4KB. All inserted key-value entries have a fixed size of 256 bytes. With table formatting and key prefix compression provided by original LevelDB, SSTables generated in the experiments (with or without compact index) have 148639 entries and 8744 data blocks, with each block on average holding approximately 17 entries. Four different prefix group size distribution patterns are used to evaluate the index: each prefix group has either fixed 16, 32, or 64 entries sharing the same prefix, or has a Zipfian distributed prefix group size with a maximum of 8000 entries. According to the experimental results, these different patterns generated 9290, 4645, 2323, and 26 distinct prefix groups within a single SSTable, respectively.

Memory Consumption: Table 3 shows the experimental results in terms of memory consumption between SlimDB and LevelDB. LevelDB's default indexing mechanism is built upon a sorted array of the last keys of each data blocks. Experiments show that this mechanism can take up to almost 18 bits per key in order to store the entire index in the memory. However, if key compression is applied to LevelDB — which means storing only the unique key prefix that can distinguish the last key of a data block from the first key of the next data block, instead of storing the entire key — LevelDB's memory consumption can be reduced from 18 to about 10.5 bits per key. LevelDB also assumes that each data block has a variable and unpredictable size. So in the indexing structure, LevelDB stores the offset and length of each data block in order to later locate and read those data blocks. In SlimDB design, all data blocks have a fixed size, which allows it to skip storing block locations within the index. With this assumption, LevelDB's memory consumption can be reduced from 10.5 to about 8 bits per key, which we see as the best memory usage that can be achieved from an LevelDB style array-based indexing mechanism. In contrast, SlimDB's compact three-level indexing only requires 1.5 to 2.5 bits per key to represent the entire index, which is as little as 8% to 14% of the memory space needed by LevelDB out of the box. Another observation from Table 3 is that the memory savings depend on workload patterns. SSTables with a lot of small prefix groups require relatively larger memory footprints compared to SSTables storing only a few large prefix groups. This is because a larger set of distinct prefix groups leads to a larger prefix ECT structure is used as the first level of the three-level index.

Table 3: *The memory consumption of different SSTable indexes measured as bits per key, for various patterns of prefix groups (3 fixed sized and a Zipfian distribution)*

	Fix.16	Fix.32	Fix.64	Zipf
SlimDB	2.56	1.94	1.57	1.56
LDB_FixedBlock	7.58	7.82	8.18	8.43
LDB_KeyCompress.	10.37	10.61	10.98	11.23
LDB_Default	17.39	17.39	17.39	17.39

Lookup Performance: Table 4 shows the experimental results in terms of in-memory lookup throughput against these indexes. As can be seen from the table, the three-level indexing mechanism has a longer lookup time, about 5 to 7 times slower than the original LevelDB index. This is because our more compact indexing requires a more sophisticated search algorithm to complete each query. A closer analysis found that decoding the first-level index consumes 70% of the CPU cycles for fixed prefix group sized workloads. For Zipfian-prefix-group size workloads, decoding the third-level index occupies 60% of the CPU cycles. The three-level index trades greater CPU cost for saving memory space, which is worthwhile when the gap between CPU resources and memory resources is large.

Table 4: *Average SSTable index lookup speed for SlimDB and the original LevelDB in thousands of lookups per second.*

	Fix.16	Fix.32	Fix.64	Zipf
SlimDB (Kops/s)	147.5	143.6	149.5	288.5
LevelDB (Kops/s)	1042.7	1019.4	1016.3	1046.0

5.3 Microbenchmark on Filters

While the high throughput of flash disks often limits the CPU budget available for in-memory indexing, this section demonstrates the computation efficiency of our multi-level cuckoo filters. The multi-level cuckoo filter is compared against traditional ways of using cuckoo filters in LevelDB and other key-value stores. In this section, the multi-level cuckoo filter is denoted as "MLCF", and the cuckoo filter is denoted as "CF".

Experiment Design: We focused on the bulk insertion and lookup speed of multi-level cuckoo filters. All experiments are single-threaded programs.

The benchmark builds a filter for an 8-sub-level key-value store. Each sub-level has 10M random 16-byte keys. This micro-benchmark involves only in-memory accesses (no flash I/O). Keys are pre-generated, sorted, and passed to our filters. There is also a parameter d called "duplication ratio" that controls the ratio of duplicated keys at each level. If $d = 10\%$, this means that for any level i $(0 \leq i < 7)$, 10% of the keys are selected from the first 10% of keys in level 8, and the other 80% of keys are distinct from all other keys in the key-value store. Later, we show the impact of this ratio of duplicated keys on the performance of in-memory filters.

To measure lookup performance, we use both true-positive queries and false-positive queries. For true-positive queries, the benchmark issues 80M random requests on positive keys, meaning that the keys are present in the key-value store. For false-positive queries, the benchmark issues 80M random requests on negative (not present) keys.

Space Efficiency and Achieved False Positive Rate: In this experiment, CF is configured to use a 16-bit hash fingerprint. MLCF uses a 14-bit hash fingerprint and a 3-bit value field to index the level number. Table 5 shows the actual memory cost per key and the achieved false positive rate for these configurations. The actual memory cost includes the space inflation caused by a less than 100% load factor for the cuckoo hashing table, as well as such as the secondary table used by MLCF. By comparing the memory cost of MLCF and CF, the memory overhead introduced by the secondary table is negligible because the false positive rate is low.

Table 5: *RAM usage and false positive rate of different filters.*

	CF	MLCF
RAM Cost (bits/key)	16.78	16.67
False Positive Rate	0.001	0.002

Insert Performance: Table 6 shows the bulk insertion performance of different filter implementations under different duplication ratios. CF is faster than MLCF in all cases. This is because MLCF has to check whether an entry has already been inserted into other levels when inserting a new entry. When the duplication ratio becomes large, the average insertion throughput of MLCF becomes higher. The insertion speed of Cuckoo hash tables is higher when its table occupancy is lower. Since MLCF only uses a single table to store all entries, a high duplication ratio leads to low table occupancy.

Lookup Performance: Figure 10 shows the lookup performance of different filter implementations under differ-

Table 6: *Bulk insertion throughput of three filters under different duplication ratio in millions of inerstions per second.*

Duplication Ratio	0%	10%	30%	50%	70%	90%
Cuckoo Filter	2.0	2.05	2.05	2.05	2.05	2.05
Multi-level CF	0.86	0.90	0.98	1.11	1.31	1.40

ent duplication ratios. MLCF is faster than CF in all cases. Since there is only a single hash table in MLCF, the lookup operation requires fewer memory references than the other two alternatives. When the duplication ratio grows larger, the table occupancy in all three filters becomes lower and therefore, all three filters gain higher lookup throughput. Under a higher duplication ratio, most entries in CF are stored in the higher level, so the lookup procedure can find these entries earlier. However, the duplication ratio does not affect the performance of negative queries in CF. This is because the lookup procedure still needs to check each level. A single MLCF has better average lookup performance compared to CF in multiple levels.

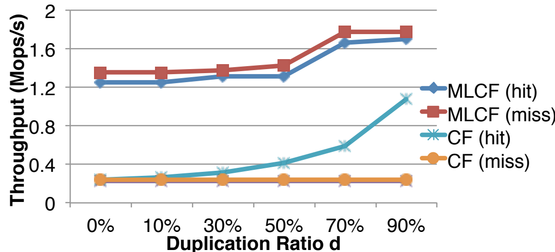

Figure 10: *Average lookup throughput of three filters under different duplication ratios.*

5.4 Full System Benchmark

In this section, we report experiments that evaluate the key-value stores in an end-to-end YCSB benchmark suite [9] consisting of two phases: a create phase initializing the key-value namespace and a query phase reading/writing key-value entries. We modified YCSB's key generator such that the key namespace is divided into small groups such that each group has the same number of entries. The key schema consists of an 8-byte prefix and an 8-byte suffix. All entries in the same group share the same hashed prefix. By default, we use the uniform key distribution in YCSB, as it represents the least locality, and use minimal overwrites in the workload, which helps increase a stores' write pressure.

SlimDB Improves Insertion and Lookup Cost: We evaluate our tested systems through multiple query workloads. During the creation phase, the YCSB benchmark inserts 100 million entries into the empty key-value store in a random order. These insertions are blind writes. The value size of each entry is fixed to be 100 bytes. And each prefix group has exactly 128 entries.

During the query phase, we tested four different workload patterns from YCSB: a) The first workload issues one million lookup operations on randomly selected entries (labeled as Random Read); b) The second workload consists of a mix of 50% random reads and 50% update operations (labeled as 50%R+50%BW). The update request is a blind write, which inserts the entry into the key-value store without checking the existence of the key; c) The third workload consists of a

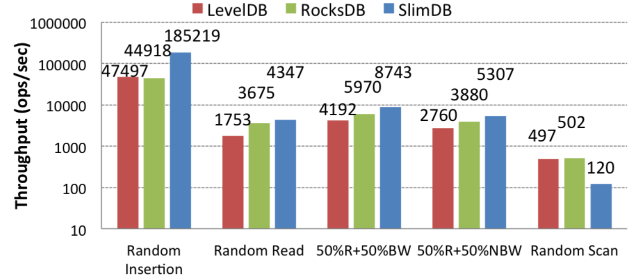

Figure 11: *Average throughput during five different workloads for three tested systems. Log-scale is used on the y-axis.*

mix of 50% random reads and 50% read-and-update operations (labeled as 50%R + 50%NBW). The read-and-update is a non-blind write, reading the old value of the sought key before updating it; d) The fourth workload does a sorted scan starting from a random prefix to fetch all entires of one prefix group (labeled as Random Scan).

To test out-of-RAM performance, we limit the machine's available memory to 4GB so the entire test does not fit in memory. Therefore picking an entry randomly will render key-value store's internal cache effect ineffective. All tests were run three times, and the coefficient of variation was less than 1%.

Figure 11 shows the test results average over three runs. SlimDB is much faster (≈4X) than LevelDB and RocksDB in terms of insertion speed because of the use of Stepped-merge algorithm. With the help of the compact in-memory index, SlimDB is also 1.4 to 2 times faster for point queries. Even for blind writes, the multi-level cuckoo filter will limit the number of entries inserted into the secondary table to avoid space explosion. In such case, multi-level cuckoo filter will just behave like a traditional Bloom filter. However, SlimDB is slower for scan operations since it has to search many more sub-levels when using the Stepped-Merge algorithm. If a workload is dominated by scan operations, SlimDB can be configured to use fewer sub-levels to exchange faster scans for slower insertions.

Figure 12 shows the test results for the insertions and scan entries workload when the prefix group size varies from 128 to 512. The insertion througphut of SlimDB is not affected by the group size. For all the tested system, the scan throughput descreases when group size increases since each scan operation fetches more entries. The scan time spent fetching each entry actually decreases since more entries are grouped in the same SSTable when group size is larger.

SlimDB Needs Less Main Memory: To show that SlimDB uses a smaller memory footprint, we repeat the

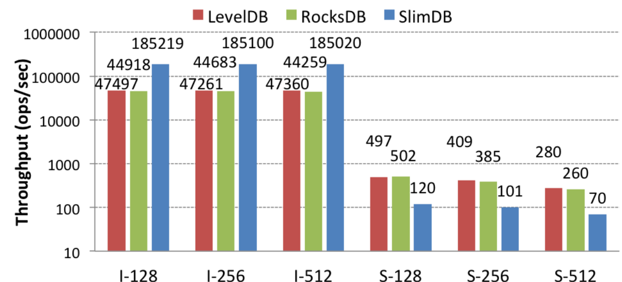

Figure 12: *Average throughput when the prefix group sizes are 128, 256, and 512. Log-scale is used on the y-axis.*

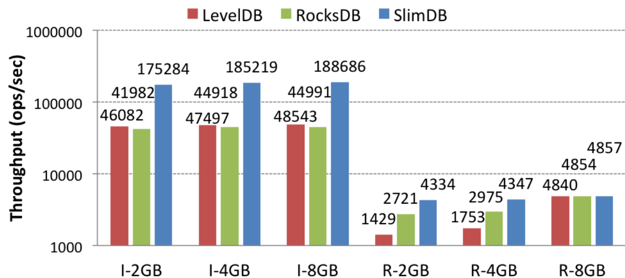

Figure 13: *Average throughput of creation and random reads when the memory sizes are 2GB, 4GB, and 8GB. Log-scale is used on the y-axis.*

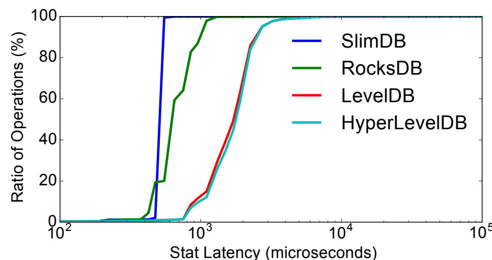

Figure 14: *The latency distribution of lookup operations for all tested systems during the query phase.*

above benchmark under different available memory size. We changed the machine memory sizes through Linux boot parameters. Figure 13 shows the results of the create phase and query phase (random read workload). For all three tested systems, creation performance is insensitive to the memory size. This is because the creation workload contains only blind writes, and does not need to check the existence of an entry. For random reads, unlike other two systems, SlimDB's read performance is similar when memory size is between 2GB and 8GB, because SlimDB's indexes and filters are compact and more easily cached in memory. When the memory size is 8GB, all tested systems can cache most of datasets and achieve similar read performance.

SlimDB Scales Better with Data Volume: Because the scalability of key-value store is of topical interest, we modified the create phase to insert 450 million entries in total with 256 byte values in each entry. The prefix group size is still 128. The total size of all entries is about 110GB, which almost saturates the solid-state disk used in this evaluation.

Table 7: *The average insertion throughput and write amplification of tested system over the entire 110GB create phase.*

	Insertion (Kops/s)	Write Amp.
SlimDB	40.07	3.74
RocksDB	15.86	13.40
LevelDB	11.12	31.80

Table 7 lists the average insertion throughput and write amplification over the entire create phase. Since the create phase starts with an empty key-value store and entries are created without name collisions, most existence checks do not result in disk I/Os with the help of LSM-tree's in-memory filters. SlimDB still shows higher throughput than other tested stores. The write amplification is calculated as the ratio between the actual amount of written data, was seen by the device and the total size of entries. The write amplifications shown in the Table 7 match well to their theoretical bounds. SlimDB has 4 levels and its write amplification is 3.74, which is very close to $\log_r N = 4$, the theoretical bound of the Stepped-Merge algorithm. Other key value stores have much higher write amplification. For example, the write amplification of LevelDB, which is closest to the standard LSM-tree, is 31.8. This matches its theoretical bound $r \log_r N = 32$.

Figure 14 shows the distribution of latencies of lookup operations for the random read workload after the creation phase. The x-axis presents the latency in a log scale. For SlimDB, its 99.9% percentile latency is 0.33ms, which is significantly better than other tested systems. It demonstrates the the bounded latency advantage by using compact indexes in SlimDB. The tail latencies in other tested systems are greatly affected by concurrent compaction procedure.

6. RELATED WORK

Modern key-value stores support fast writes/updates by using log-based write or log-structured method such as FlashStore [11], SkimpyStash [12], SILT [24], LevelDB [20] and bLSM [31]. Though log-appending is efficient for admitting new data, it leaves the data not well organized and produces a large metadata set, which leads to performance degradation for reads. Thus most systems such as LevelDB, SILT and bLSM employ compaction procedure to re-organize the internal data layout to (remove garbage from the log and) reduce metadata size for fast reads. However, compaction algorithms can generate a large write amplification. RocksDB [29] compacts more than two contiguous levels at once intending to sort and push data faster to the lower level [29]. B^ϵ trees, FD-Tree, and TokuDB [6, 22, 4] are based on the fractal tree and its variants, which buffer updates in internal nodes where data is pushed to its next level by appending it to log files. Without more sorting of its data, TokuDB needs to maintain an index with a generally larger memory footprint. In contrast, with the support of dynamic use of a stepped merge algorithm and optimized in-memory indexes, SlimDB minimizes write amplification without sacraficing read performance.

There are other variants of the LSM-tree that exploit workload characteristics to enhance write performance. LSM-trie [33] combines the Stepped-Merge algorithm with hash-range based compaction procedure to de-amortizing compaction overhead. VT-tree [32] exploits sequentiality in the workload by increasing the number of indirections (called stitching) to avoid merge-sorting all aged SSTables during compaction. Walnut and WiscKey [8, 25] use similar partition techniques that store small objects into LSM-trees and large objects into append-only logs to reduce write amplification by avoiding compacting large objects. This design trades more disk reads for fewer disk writes. Most of these technqiues are orthogonal to our work, and they can be used in conjunction with SlimDB.

Compact in-memory indexes and filters are often used in key-value stores to reduce unnecessary disk accesses for read operations. Bloom filter [5] is the most widely used filters. But it cannot support deletion of an item and has a bigger memory footprint than Cuckoo filters. Perfect hashing indexes, such as CHD [3] and ECT [24], use fewer than 2.5

bits per key to map any given key in a set to a distinct location. Such indexes support positive lookups but do not answer negative lookups. Set separator [15] is another space-efficient index structure that maps a large set of keys into a small range of values. Its per-key memory cost is proportional to the size of the value range. However, these indexes only support pure hashing keys instead of semi-sorted keys, do not preserve the sequentiality of prefix keys.

Existing LSM-tree based key-value stores do not allow trading among read cost, write cost and main memory footprint. For example, LevelDB [20], RocksDB [29] and LSM-trie [33] only enable leveled compaction or stepped merge algorithm respectively, and they use a fixed number of bits per element for all Bloom filters and SSTable indexes. Similar to our work, Monkey [10] uses worst-case closed-form models that enable optimizing throughput by automatically tuning the size ratio between levels and bit cost of Bloom filters. Another complementary work [23] uses a numeric simulation method to quantify update cost in different LSM-tree variants when there is a skew in the update pattern. We go one step further here; our analytical model quantifies costs for both filters and SSTable indexes thereby searching over a bigger design space.

7. CONCLUSION

General-purpose LSM-tree implementations usually lack optimizations for read and write amplification for key-value workloads running on solid-state disks. We present techniques that allow key-value workloads with semi-sorted data and non-blind writes to run more efficiently, in terms of both I/O activities and memory consumption. To improve read performance, we present two ideas for shrinking an index: a three-level compact index that only costs 1.9 bits per key to locate the block position of a particular key inside the SSTable; and the design of a multi-level cuckoo filter that not only bounds the worst-case latency of lookup operations, but also improves the average latency. Through integration with the Stepped-Merge algorithm, experiments show that our new key-value store implementation, SlimDB, can achieve a better balance between read and write amplification. We also proposed an analytical framework allowing optimal selection of data layout and in-memory index in key-value stores for general workloads.

8. REFERENCES

[1] T. G. Armstrong, V. Ponnekanti, D. Borthakur, and M. Callaghan. Linkbench: a database benchmark based on the facebook social graph. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2013.

[2] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny. Workload analysis of a large-scale key-value store. In *ACM SIGMETRICS Performance Evaluation Review*, 2012.

[3] D. Belazzougui, F. C. Botelho, and M. Dietzfelbinger. Hash, displace, and compress. In *Proceedings of the European Symposium on Algorithms*, 2009.

[4] M. A. Bender, M. Farach-Colton, J. T. Fineman, Y. R. Fogel, B. C. Kuszmaul, and J. Nelson. Cache-oblivious streaming B-trees. In *Proceedings of annual ACM symposium on parallel algorithms and architectures*, 2007.

[5] B. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communication of ACM*, 1970.

[6] G. S. Brodal and R. Fagerberg. Lower bounds for external memory dictionaries. In *Proceedings of the annual ACM-SIAM symposium on Discrete algorithms*, 2003.

[7] F. Chang, J. Dean, S. Ghemawat, and et al. Bigtable: a distributed storage system for structured data. In *the USENIX Symposium on Operating Systems Design and Implementation*, 2006.

[8] J. Chen, C. Douglas, and et al. Walnut: A unified cloud object store. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2012.

[9] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking Cloud Serving Systems with YCSB. In *the ACM Symposium on Cloud Computing (SOCC)*, 2010.

[10] N. Dayan, M. Athanassoulis, and S. Idreos. Monkey: Optimal navigable key-value store. In *the ACM International Conference on Management of Data (SIGMOD)*, 2017.

[11] B. Debnath, S. Sengupta, and J. Li. Flashstore: high throughput persistent key-value store. *Proceedings of the VLDB Endowment*, 2010.

[12] B. Debnath, S. Sengupta, and J. Li. SkimpyStash: RAM space skimpy key-value store on flash-based storage. In *ACM International Conference on Management of data*, 2011.

[13] G. DeCandia, D. Hastorun, and et al. Dynamo: Amazon's highly available key-value store. In *Proceedings of ACM SIGOPS Symposium on Operating Systems Principles*, 2007.

[14] B. Fan, D. G. Andersen, M. Kaminsky, and M. D. Mitzenmacher. Cuckoo filter: Practically better than bloom. In *Proceedings of the ACM International on Conference on Emerging Networking Experiments and Technologies*, 2014.

[15] B. Fan, D. Zhou, H. Lim, M. Kaminsky, and D. G. Andersen. When cycles are cheap, some tables can be huge. In *Proceedings of the USENIX conference on Hot Topics in Operating Systems*, 2013.

[16] N. Fountoulakis, M. Khosla, and K. Panagiotou. The multiple-orientability thresholds for random hypergraphs. In *the ACM-SIAM symposium on Discrete Algorithms*, 2011.

[17] X. He, J. Pan, O. Jin, and et al. Practical lessons from predicting clicks on ads at facebook. In *The International Workshop on Data Mining for Online Advertising*, 2014.

[18] G. J. Jacobson. *Succinct Static Data Structures.* PhD thesis, Pittsburgh, PA, USA, 1988. AAI8918056.

[19] H. V. Jagadish, P. P. S. Narayan, S. Seshadri, S. Sudarshan, and R. Kanneganti. Incremental organization for data recording and warehousing. In *Proceedings of the International Conference on Very Large Data Bases*, 1997.

[20] LevelDB. A fast and lightweight key/value database library, 2011. http://code.google.com/p/leveldb/.

[21] C. Li, Y. Lu, Q. Mei, D. Wang, and S. Pandey. Click-through prediction for advertising in twitter timeline. In *Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2015.

[22] Y. Li, B. He, R. J. Yang, Q. Luo, and K. Yi. Tree indexing on solid state drives. *Proceedings of the VLDB Endowment*, 2010.

[23] H. Lim, D. G. Andersen, and M. Kaminsky. Towards accurate and fast evaluation of multi-stage log-structured designs. In *Proceedings of the Usenix Conference on File and Storage Technologies (FAST)*, 2016.

[24] H. Lim, B. Fan, D. G. Andersen, and M. Kaminsky. SILT: a memory-efficient, high-performance key-value store. In *the ACM Symposium on Operating Systems Principles*, 2011.

[25] L. Lu, T. S. Pillai, and et al. Wisckey: Separating keys from values in ssd-conscious storage. In *Proceedings of the 14th Usenix Conference on File and Storage Technologies*, 2016.

[26] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil. The log-structured merge-tree (LSM-tree). *Acta Informatica*, 1996.

[27] R. Pagh and F. F. Rodler. Cuckoo hashing. *Journal of Algorithms*, 2004.

[28] K. Ren and G. Gibson. TableFS: Enhancing metadata efficiency in the local file system. *Usenix Annual Technical Conference*, 2013.

[29] RocksDB. A facebook fork of leveldb which is optimized for flash and big memory machines, 2013. https://rocksdb.org.

[30] O. Rodeh, J. Bacik, and C. Mason. BRTFS: The Linux B-tree Filesystem. *IBM Research Report RJ10501*, 2012.

[31] R. Sears and R. Ramakrishnan. bLSM: a general purpose log structured merge tree. *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2012.

[32] P. Shetty, R. Spillane, R. Malpani, B. Andrews, J. Seyster, and E. Zadok. Building workload-independent storage with VT-Trees. In *Proccedings of the conference on File and Storage Technologies*, 2013.

[33] X. Wu, Y. Xu, Z. Shao, and S. Jiang. LSM-trie: An LSM-tree-based ultra-large key-value store for small data. In *USENIX Annual Technical Conference*, 2015.