Building Efficient Key-Value Stores via a Lightweight Compaction Tree

TING YAO and JIGUANG WAN, Wuhan National Laboratory for Optoelectronics, Huazhong University of Science and Technology
PING HUANG and XUBIN HE, Temple University
FEI WU and CHANGSHENG XIE, Wuhan National Laboratory for Optoelectronics, Huazhong University of Science and Technology

Log-Structure Merge tree (LSM-tree) has been one of the mainstream indexes in key-value systems supporting a variety of write-intensive Internet applications in today's data centers. However, the performance of LSM-tree is seriously hampered by constantly occurring compaction procedures, which incur significant write amplification and degrade the write throughput. To alleviate the performance degradation caused by compactions, we introduce a lightweight compaction tree (LWC-tree), a variant of LSM-tree index optimized for minimizing the write amplification and maximizing the system throughput. The lightweight compaction drastically decreases write amplification by appending data in a table and only merging the metadata that have much smaller size. Using our proposed LWC-tree, we have implemented three key-value LWC-stores on different storage mediums including Shingled Magnetic Recording (SMR) drives, Solid State Drives (SSD), and conventional Hard Disk Drives (HDDs). The LWC-store is particularly optimized for SMR drives, as it eliminates the multiplicative I/O amplification from both LSM-trees and SMR drives. Due to the lightweight compaction procedure, LWC-store reduces the write amplification by a factor of up to 5× compared to the popular LevelDB key-value store. Moreover, the random write throughput of the LWC-tree on SMR drives is significantly improved by up to 467% even compared with LevelDB on conventional HDDs. Furthermore, LWC-tree has wide applicability and delivers impressive performance improvement in various conditions, including different storage mediums (i.e., SMR, HDD, SSD) and various value sizes and access patterns (i.e., uniform and Zipfian).

CCS Concepts: • **General and reference → Performance**; • **Information systems → Key-value stores**; *Magnetic disks*;

Additional Key Words and Phrases: Lightweight compaction, LSM-tree, write amplification, SMR drives

A preliminary version of this work was presented at the 33rd International Conference on Massive Storage Systems and Technology (MSST 2017) as *A Light-weight Compaction Tree to Reduce I/O Amplification toward Efficient Key-Value Stores*. This research is partially sponsored by the National Natural Science Foundation of China under Grant No. 61472152, No. 61300047, No. 61432007, and No. 61572209; the Fundamental Research Funds for the Central Universities HUST:2015QN069; the 111 Project (No. B07038); the Director Fund of WNLO; the Key Laboratory of Data Storage System, Ministry of Education; and the U.S. National Science Foundation (NSF) under Grant No. CNS-1702474 and No. CNS-1700719.

Authors' addresses: T. Yao, J. Wan (Corresponding Author), F. Wu (Corresponding Author), and C. Xie, Wuhan National Laboratory for Optoelectronics, School of Computer Science and Technology, Huazhong University of Science and Technology; emails: {tingyao, jgwan}@hust.edu.cn, wufei@hust.edu.cn, cs_xie@hust.edu.cn; P. Huang and X. He, Department of Computer and Information Sciences, Temple University; emails: {templestorager, xubin.he}@temple.edu.

https://doi.org/10.1145/3139922

ACM Reference format:
Ting Yao, Jiguang Wan, Ping Huang, Xubin He, Fei Wu, and Changsheng Xie. 2017. Building Efficient Key-Value Stores via a Lightweight Compaction Tree. *ACM Trans. Storage* 13, 4, Article 29 (November 2017), 28 pages.
https://doi.org/10.1145/3139922

1 INTRODUCTION

Key-value stores are becoming widespread in modern data centers as they support an increasing diversity of applications [25]. Currently, the majority of key-value stores [8, 15, 16, 21] are built based on a *Log-Structured Merge tree* (*LSM-tree*) [29], which is used as the index structure for key-value stores. An LSM-tree demonstrates its advantages over traditional B-trees because it adopts a memory buffer to batch new writes and create write sequentiality. In the meantime, the batched data have to be flushed to disks sooner or later. An LSM-tree initiates procedures called *compaction* to absorb the temporarily buffered content and ensure key-value pairs in a sorted order for future fast lookups. Unfortunately, compactions introduce significant overheads due to I/O amplifications.

To compact a table, an LSM-tree has to read a table file (called *victim table*) in Level L_i and several table files (called *overlapped tables*) in Level L_{i+1}, which have key ranges overlapping with that of the victim table. It then merges those tables according to their key ranges and writes back the resultant tables. Such multiple table reads and writes incur excessive disk I/Os (called *I/O amplification*) and thus severely affect the performance [25]. As we will see later (Section 2.2), the I/O amplification caused by compactions in a typical LSM-tree can reach to a factor of 50× or higher [27, 36], causing up to 10× degraded I/O performance.

Previously proposed solutions alleviate the write amplification and thus improve the overall performance by using large memory to buffer more indexes or key-value (KV) pairs [39], reducing the number of levels [24, 30], or simply reducing the amplification factor in adjacent levels. Solving the write amplification problem of LSM-trees in this manner, unfortunately, are not cost-efficient as they either require additional memory or have a counter effect when the store grows large. Even worse, the coupling of LSM-tree-based key-value stores and storage devices may further amplify the compaction I/O overhead [25, 27]. The I/O amplification of running key-value stores on Shingled Magnetic Recording (SMR) drives, which are becoming increasingly deployed in data centers, is particularly challenging due to the auxiliary amplification of SMR drives, similarly to Solid State Drives (SSDs) [38].

To solve the I/O amplification problem in KV stores, in this article, we present a *lightweight compaction tree (LWC-tree)* to mitigate the I/O amplification during a compaction procedure by appending data in a table and only merging a small amount of metadata. Our LWC-tree significantly improves the write throughput and retains a fast read performance in KV stores via the following four new techniques:

(1) The LWC-tree employs a simplified compaction procedure, named *lightweight compaction*. Instead of reading, sorting, and rewriting all entire tables in a traditional compaction procedure, our LWC-tree divides data of a victim table into segments according to the key range of overlapped tables in the next level. Then it appends segments to the corresponding tables and merges a small amount of metadata simultaneously. As the unit of data management in the LWC-tree, the table is referred to a DTable.
(2) The LWC-tree conducts *metadata aggregation* in adjacent levels to remove small reads on devices. The metadata aggregation policy collects the metadata in overlapped tables

and stores them in the victim table in the upper level of the LWC-tree. Therefore, when performing a lightweight compaction, the LWC-tree only requires to read one table instead of multiple tables in traditional LSM-trees.

(3) The LWC-tree reorganizes the DTable data structure according to characteristics of the lightweight compaction. DTable absorbs the data from lightweight compactions by appending them so the table size is dynamic. DTable provides efficient lookups via binary searching segment indexes using sequence numbers.
(4) The LWC-tree balances the workload of DTables in the same level to provide a consistent and efficient operation performance. Workload balance aims to balance the overly full DTable with its siblings by adjusting their key ranges after lightweight compactions without extra overhead.

To explore the applicability of our LWC-trees in key-value stores used in modern data centers, we further design an LWC-tree-based key value store, called LWC-store. We implement our LWC-store on various storage devices including SSDs, SMR drives, and conventional HDDs. Specifically, for the LWC-store on SMR drives, an equal division is proposed to avoid a DTable overflowing a band on SMR drive, which reads out a DTable, divides it into several sub-DTables and writes them back to the same level. The intensive experimental results using both micro-benchmarks and macro-benchmarks show that our LWC-store demonstrates substantial performance improvement compared to the popular LSM-tree-based key-value store LevelDB on different storage devices (SMRs, SSDs, and HDDs).

The rest of this article is organized as follows: the background and motivation of our work are described in Section 2. The design and implementation of the LWC-tree and LWC-stores are presented in Sections 3 and 4, respectively. Section 5 presents the evaluation results and analyzes the effectiveness of the lightweight compaction. Related work is described in Section 6, and our conclusions of this work are given in Section 7.

2 BACKGROUND AND MOTIVATION

To provide better service quality and responsive user experience for many data-intensive Internet applications, key-value stores have been adopted as the infrastructure in modern data centers. In addition, the performance gap between random I/O and sequential I/O has increased, decreasing the relative cost of the additional sequential I/O and widening the range of workloads that can benefit from log-structure [32]. Therefore, key-value stores based on the LSM-tree that transform random accesses to sequential accesses are gaining increasing popularity and have widespread practical deployments.

A KV store design based on an LSM-tree services two goals [36]: One goal is that new data have to be quickly admitted into the store to support high-throughput writes, which is achieved by the data organization discussed in Section 2.1, and the other goal is that KV pairs in the store are sorted to support fast lookups, which is achieved by recurring compactions discussed in Section 2.2.

2.1 LSM-tree and LevelDB

The LSM-tree is a widely used persistent data structure that provides efficient indexing for key-value stores with the high rate of insertions and deletions [29]. It first batches writes into memory to avoid random writes and exploit the high sequential bandwidth of hard drives and then updates the in-memory content to the LSM-tree at a later time. Since random writes are nearly two orders of magnitude slower than sequential writes on hard drives [29], LSM-trees can thus provide better write performance than traditional B-trees that incur excessive random accesses, even though they perform more writes due to the compaction process.

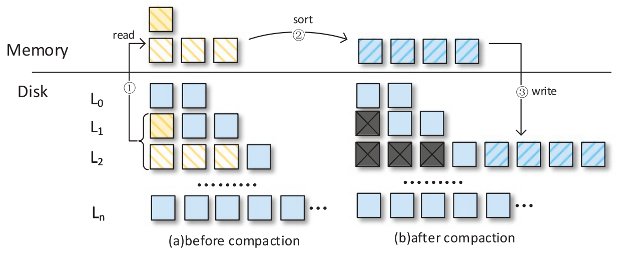


Fig. 1. The compaction procedure of LSM-trees. The compaction involves reading, sorting, and rewriting the victim SSTable and all the overlapped SSTables, leading to serious I/O amplification.

LevelDB, inspired by BigTable [8, 16], is a popular open-source key-value store from Google using LSM-trees to organize key-value pairs. We use LevelDB as an example to explain the data structure of LSM-trees. LevelDB uses an in-memory buffer, called MemTable, to absorb incoming KV pairs, which results in high-throughput writes in LevelDB. The quickly admitted data are sorted according to their keys simultaneously by the skip-list in memory. Once being filled up, a Memtable is turned into an immutable MemTable, which still remains in the memory but no longer accepts new data. Later, the immutable Memtable with the size of several megabytes is dumped to the disk, generating an on-disk data structure called SSTable. SSTables are immutable and each SSTable contains a number of KV pairs stored in the unit of data block. KV pairs can be indexed by the metadata of SSTable and LevelDB can locate the block of a specific KV pair via binary searching on the index. In the metadata of an SSTable, there is a Bloom filter to indicate the presence of the KV pairs for each block [36], which facilitates avoiding unnecessary accesses to data blocks to reduce read latency.

The combination of MemTable and immutable MemTable in the memory, together with the multi-level SSTables on disk compose a key-value store based on LSM-trees. Since the LSM-tree is the core element of LevelDB and other widely used key-value stores, such as RocksDB [13] at Facebook, Cassandra [22], HBase [5], and PNUTS [9] at Yahoo!, the reductions of write amplification in our proposed LWC-tree can be beneficial to all KV stores based on LSM-trees and enjoy wide applicability.

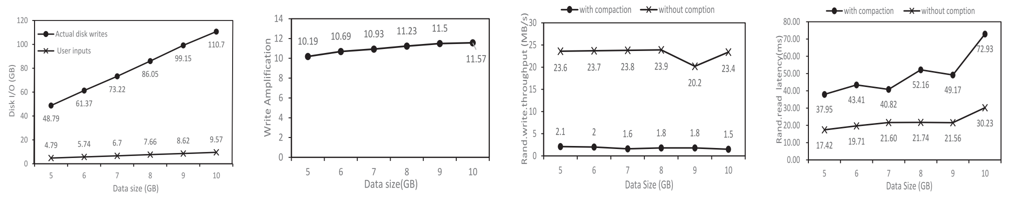
2.2 Performance Degradation by Compaction

As mentioned before, LSM-trees achieve outstanding write performance by batching key-value pairs and later writing them sequentially. Subsequently, to enable future efficient lookups and deliver an acceptable read performance (for both individual keys as well as range queries), LSM-tree continuously compacts key-value pairs at adjacent levels throughout the lifetime to sort key-value pairs. The compaction procedure of an LSM-tree requires constant reading, sorting, and writing KV pairs, which introduces excessive writes and represents a performance bottleneck of LSM-tree-based key-value stores.

Specifically, a single compaction has three steps. For the convenience of exposition, we call the SSTable selected to compact in level L_i as a victim SSTable and the SSTables whose key range fall in the key range of the victim SSTable in the next level L_{i+1} as overlapped SSTables. To start a compaction, LSM-tree first selects the victim SSTable and the overlapped SSTables according to the score of each level and then decides whether more victim SSTables in L_i can be added into the compaction by searching the SSTables whose key range fall in the ranges of overlapped SSTables. Figure 1 shows the three steps of a compaction procedure of an LSM-tree. During the compaction procedure, LevelDB first reads the victim SSTable in level L_i and overlapped SSTables in level L_{i+1}.

Table 1. A List of Symbols

S_{data}	Data size of disk I/O in a compaction
S_{sst}	Size of an SSTable
S_{dt}	Size of a DTable
$S_{metadata}$	Metadata size in a compaction
RA/WA	Read/Write amplification factor from LSM-tree or LWC-tree
ARA/AWA	Auxiliary Read/Write amplification factor from storage devices
MRA/MWA	Overall Read/Write amplification factor, MWA=WA × AWA



(a) I/O comparison (b) write amplification (c) random write (d) random read

Fig. 2. Write amplification and performance degradation due to compactions. (a) The comparison between the write data size and the actual disk I/O size; (b) the write amplification of different write data sizes; (c) the random write performance with and without compaction; (d) the random read performance with and without compaction.

After that, LevelDB merges and sorts SSTables that have been fetched into memory by the first step. Finally, LevelDB writes the newly generated SSTables to disk. According to the size limit of each level in LevelDB, the size of L_{i+1} is 10 times that of L_i, and this size factor is called the amplification factor (AF). Due to the size limitation of levels, on average a victim SSTable in level L_i has AF overlapped SSTables in level L_{i+1}, and thus the total data size involved in a compaction is given by Equation (1), where S_{sst} represents the size of an SSTable and S_{data} represents the data size of disk I/O in a compaction, as listed in Table 1. The 2× multiplication indicates the total data volume of both reading and writing. With a large dataset, the ultimate amplification could be over 50 (10 for each gap between L_1 to L_6), as it is possible for any newly generated SSTable to migrate from L_0 to L_6 through a series of compaction steps [25],

$$S_{data} = (AF + 1) \times S_{sst} \times 2. \tag{1}$$

To quantitatively measure the degree of amplification and performance degradation due to compactions in practice with LevelDB, we carry out the following experiments using a similar configuration of LevelDB on conventional HDDs in Section 5. First, we randomly load six databases of sizes ranging from approximately 5GB to 10GB. The value size is 4KB by default. Figure 2(a) shows the relationship between the input data size and the actual disk I/O data size. As clearly shown, in all cases, LevelDB incurs significant write amplification. For example, writing 9.57GB input data results in 110.7GB actual disk write and the corresponding write amplification ratio is 11.57 as shown in Figure 2(b). Based on Figure 2(a), Figure 2(b) calculates all write amplification factors and a minimum factor of 10.19 is observed. Second, to evaluate the random I/O performance with compaction or without compaction, we use the same six databases ranging from 5GB to 10GB for initial random loads and randomly query 1 million keys following a uniform distribution. For the I/O performance without compaction, we trigger a manual compaction immediately after finishing loading the database and before starting I/Os to eliminate concurrent compaction and mitigate the compaction interferences. Figures 2(c) and (d) show the performance degradation caused by compaction to random write and random read, respectively. The write and read throughputs without compaction on average are 13.01× and 2.23× of the throughputs with compaction, respectively.

As demonstrated previously, the compaction procedure in conventional LSM-trees incurs excessive I/O amplification and degrades the I/O performance when used as the key-value store index. This motivates us to design the LWC-tree to mitigate the I/O amplification caused by compactions in LSM-trees and improve the I/O performance.

3 LWC-TREE DESIGN

We design a lightweight compaction tree (LWC-tree) to alleviate the I/O amplification caused by compactions and aim to achieve high write throughput without significantly sacrificing read performance. As a variant of the LSM-tree, LWC-tree is also composed of one memory-resident table (memtable) and multiple disk-resident tables (SSTables). Key-value pairs are first written to the memory component, then dumped to disks, and finally compacted to lower levels of LWC-trees. We keep the sorted tables and the multi-level structure in the LWC-tree to maintain the read efficiency of KV stores.

The LWC-tree has four distinct features over the LSM-tree. First, an LWC-tree employs a lightweight compaction mechanism to mitigate I/O amplification by appending data in overlapped tables and only merging their metadata (Section 3.1). Second, each table preserves the aggregated metadata of overlapped tables in the lower level to further reduce small random reads during the compaction procedure (Section 3.2). Third, LWC-tree introduces a new data structure for SSTable to improve the lookup performance within a table (Section 3.3); an SSTable with this new data structure in LWC-tree is called *DTable*. Finally, the LWC-tree achieves workload balance among multiple DTables in the same level to ensure read/write efficiency and the balance of an LWC-tree (Section 3.4).

3.1 Lightweight Compaction

Compactions are needed during the entire life of an LSM-tree to ensure acceptable read and scan performance. However, compactions cause excessive I/O amplification as table files of the LSM-tree have often to be read and written to disks during the compaction. Moreover, extra I/Os from compaction contend for disk resources, causing degraded performance, as demonstrated in Section 2.2. To reduce the excessive I/O amplification and alleviate the degraded system throughput due to compactions, we propose a new approach called lightweight compaction.

Motivated by the observation that merging and sorting keys are sufficient for KV store operations (read, write, and delete) while values can be separately managed (e.g., by a log) [25, 28], our LWC-tree moves a further step and proposes to merge and sort the metadata of table files to speed up the compaction procedure. Since the metadata size of each individual table file is typically much smaller than the table size itself, merging only metadata can thus significantly reduce the amount of data involved in the compaction, resulting in a lightweight compaction.

To carry out a lightweight compaction, LWC-tree first reads the victim DTable into memory and this DTable includes the aggregated metadata of overlapped DTables (Section 3.2) as well as its own data and metadata. The LWC-tree then divides the victim DTable into several segments corresponding to the key ranges of each overlapped DTable. Based on the division, it merges and sorts the metadata of each segment and the metadata of its associated overlapped DTables. The resultant new segments and metadata are appended into the overlapped DTables by overwriting the out-of-date metadata.

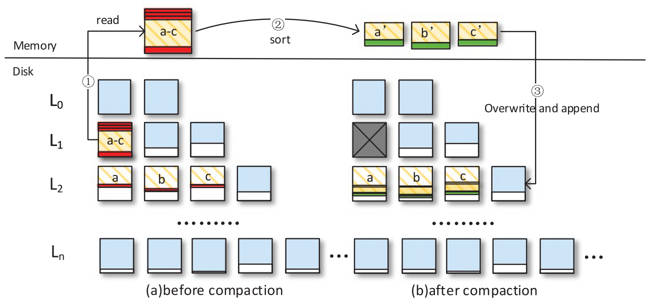


Fig. 3. The lightweight compaction procedure of an LWC-tree. The lightweight compaction involves reading one DTable, sorting the metadata and appending newly generated segments, which drastically reduces the I/O amplification relative to the traditional compaction procedure in an LSM-tree.

Figure 3 graphically shows the procedure of the lightweight compaction in an LWC-tree. As an example, we assume that the DTable having the key range of "a–c" in level L_1 is selected as the victim table for compaction, and its three overlapped DTables in level L_2 have the key ranges of "a," "b," and "c," respectively. First, instead of reading, merging, and sorting all the four involved DTables as in a conventional compaction, LWC-tree only reads the victim DTable into the memory. Second, LWC-tree clears all invalid key-value pairs and sorts the valid key-value pairs. Third, these valid key-value pairs are divided into three segments corresponding to each of the three overlapped DTables' key ranges, namely, segments "a," "b," and "c." Finally, as indicated in the right part of Figure 3, the segments are appended into the overlapped DTables in level L_2. In the meantime, the newly generated metadata of each segment is merged with the metadata of its corresponding overlapped DTable.

Compaction Process: Algorithm 1 demonstrates the overall process of the lightweight compaction. Given a victim DTable, first, key-value pairs are read into the memory in a sorted order by the function *MakeInputIterator()*. Then, for each overlapped DTable, we process every key-value pair (in a sorted order) as follows. (a) Drop the stale KV pairs by comparing their sequence numbers. (b) Drop the deleted KV pairs by comparing their value type and ensuring the object key is not included in any DTable's key ranges at the higher level. (c) Add valid key-value pairs into a segment, as demonstrated in Algorithm 2. (d) As long as a key is out of the key range of the overlapped DTable, a segment is finished right before the out-of-range key. Then we append the segment to the overlapped DTable, with metadata merging, as demonstrated in Algorithm 3. Algorithms 2 and 3 are described specifically in Section 3.3, after introducing the data structure of DTable.

A lightweight compaction only needs to read one DTable from a disk and appends amplification factor (AF) segments back to the disk. Equation (2) calculates the total amount of I/O data size (S_{data}) involved in a lightweight compaction, where S_{dt} represents the size of a DTable and $S_{metadata}$ represents the metadata size. Comparing (S_{data}) with the overhead of the compaction in an LSM-tree given by Equation (1) in Section 2.2, we notice that the I/O data size of the lightweight compaction is reduced by nearly 10×, assuming the maximum DTable size is equal to the size of an SSTable and the AF is 10 by default according to LevelDB [16],

$$S_{data} = 2 \times S_{dt} + AF \times S_{metadata}. \tag{2}$$

In addition to the I/O amplification reduction, the lightweight compaction keeps the key range of tables sorted in the same level. In other words, the key range of tables in the same level of the LWC-tree are not overlapped, which helps to provide a high table lookup efficiency.

ALGORITHM 1: Lightweight Compaction Process

```
Input: victim:victim DTable in $L_n$, overlaps:overlapped DTables in $L_{n+1}$
Output: SUCCESS/FAILURE
iter ← MakeInputIterator(victim); /*Input kv pairs in sorted order.*/
iter ← iter.first();
for i ← 0 to num_overlaps do
    for iter to iter.end() do
        key ← iter.key();
        if key.sequence ≤ MAX_key.sequence then
            drop ←true; /*Drop stale KV pairs.*/
        else
            if (key.type == Deletion) && (tables in higher level !contain key) then
                drop ← true; /*Drop deleted KV pairs.*/
            end
        end
        if !drop then
            Seg.add(key-value);/*Algorithm 2: Generate segment and its metadata.*/
        end
        if key > overlaps[i].largest_key then
            Append(Seg);/*Algorithm 3: Append segment and merge metadata.*/
            break;
        end
    end
end
return SUCCESS; /*Return FAILURE when functions Seg.add() and/or Append() fail.*/
```

3.2 Metadata Aggregation

As discussed in the preceding section, LWC-tree employs an efficient compaction policy by considering only metadata during compactions, which could result in up to 10× I/O data reduction. However, so far we have not yet discussed how to efficiently obtain the metadata of overlapped DTables during the compaction, which can critically impact the compaction efficiency. An intuitive method is reading the metadata from the overlapped DTables in the next tree level whenever the metadata are needed. Unfortunately, this method incurs extra cost due to randomly reading the metadata, offsetting the efficiency of our lightweight compaction. Another straightforward solution is to cache all the metadata in the memory for fast metadata accesses. However, this solution is not cost-effective since the metadata size is non-trivial. Figure 4 shows the ratio between metadata and data as the value size varies in two scenarios where the table size is 4MB and 40MB, respectively. As is shown, the metadata overhead increases as the value size decreases. Particularly, the metadata overhead of small value sizes is significant. Unfortunately, as revealed by existing key-value workloads analysis, small value sizes are dominant in the real world [6]. For instance, the metadata overhead is around 17% and 9% for the value size of 64B and 256B, respectively. Equally put, to support a 10TB key-value store, the total metadata would require 925GB memory when the value size and table size are respectively 256B and 40MB.

To address this problem, we propose metadata aggregation to cluster the metadata of overlapped DTables into the corresponding victim DTable, as illustrated in Figure 5. After finishing the lightweight compaction, when the updated metadata that were just merged in the memory are appended to the overlapped DTables in level L_{i+1}, they are aggregated into the victim DTable in level L_i simultaneously. Thus to avoid the random accesses of metadata for the next compaction. The

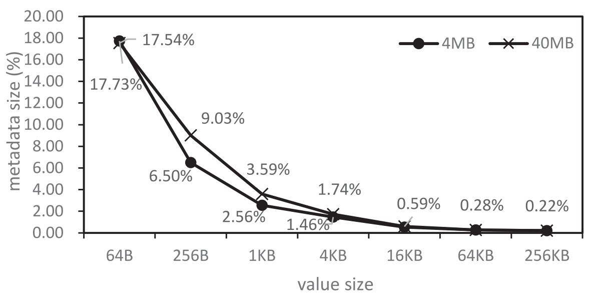


Fig. 4. The proportion of metadata in the table. This figure shows the ratio between metadata and data in a table under different value sizes with two table size being 4MB and 40MB, respectively. Small value sizes have significantly high metadata overhead.

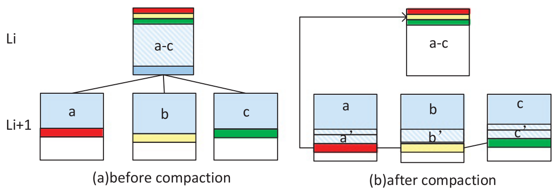


Fig. 5. The metadata aggregation between adjacent levels. This figure shows that before compaction, the metadata of overlapped DTables is collected into the victim DTable after each lightweight compaction.

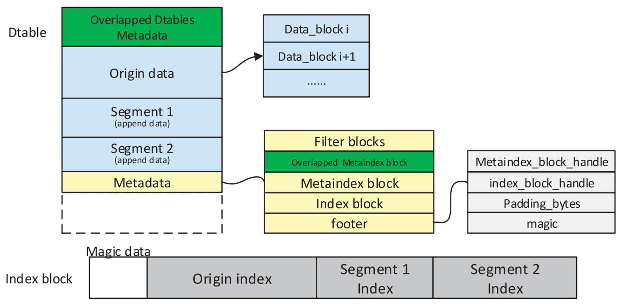


Fig. 6. The data structure of DTable. This figure shows the organization in a DTable, which is comprised of the metadata from overlapped DTables, data and metadata.

victim DTable always contains the most recent updated metadata of overlapped DTables, so consistency can be guaranteed. Though introducing an extra write in the lightweight compaction, the metadata aggregation reduces AF times random reads on average for the next compaction. Overall, the lightweight compaction only requires to read the victim DTable because all the needed metadata and data are already in the victim DTable, which significantly reduces incurred I/O traffic.

3.3 Data Structure of DTable

Similarly to SSTables in an LSM-tree, DTable is our proposed basic data management unit in the LWC-tree, and its detailed data structure is depicted in Figure 6. A DTable contains the aggregated metadata from overlapped DTables, data, and metadata. The data of DTable includes several segments. Each segment is composed of the appended data blocks of a lightweight compaction. The sequence of key-value pairs are stored in a sorted order and partitioned into a sequence of

ALGORITHM 2: Segment Generation.

```
Input: key,value
Output: SUCCESS/FAILURE
while key do
    filter_block.Add(key);
    data_block.Add(key,value);
    if data_block.size() ≥ block_size then
        SaveBlock(data_block); /*write data in segment in memory.*/
        index_block.Add(block_key, handle);
        data_block.New;
    end
end
SaveBlock(filter_block);
Seg.generator(metaindex); /* Generate and save the metaindex block according to filter blocks. */
SaveBlock(index_block);
Seg.generator(footer); /* Generate and save the footer block according to metadata. */
return SUCCESS; /* Return FAILURE when data block and/or metadata generation fail. */
```

data blocks. These blocks come one after another at the beginning of the segment [16]. Since the lightweight compaction just appends the sorted data segment to the overlapped DTables without modifying the data written before, the appended data from compaction is sorted in each individual segment but the key ranges of different segments may be overlapped. These overlapped segments in the DTables could potentially degrade the lookup performance within a table. To facilitate the read and search performance within a DTable, our LWC-tree employs a new method to manage the metadata.

The metadata stored in a DTable includes bloom filter blocks, overlapped metadata index blocks for overlapped DTables' metadata (referred as Overlapped metaindex), metadata index blocks for filter blocks (referred as metaindex), index blocks for data blocks, and a footer. (1) The filter block stores a sequence of filters, where one filter contains the results of all keys that are stored in a block. In this way, a bloom filter can indicate the existence of KV pairs in a block. (2) The overlapped metaindex block contains one entry for the handle of every overlapped DTable's metadata, where each entry contains the name of overlapped DTable, offset, and the size of overlapped DTable's metadata. (3) The metaindex block contains an entry that maps from the filter name to the block handle for a filter block. (4) The index block contains one entry per data block, where each entry is used to index a data block. The index block is organized in segments, corresponding to the data organization. Within each segment, the index blocks are sorted in a sequence order. In addition, LWC-tree assigns a sequence number to each index segment to indicate the recency of the appended segment. (5) At the very end of the file, a fix-sized footer contains the block handle of the metaindex and index blocks.

Segment Generation: Algorithm 2 demonstrates the process of segment generation. For every valid key-value pair, (a) the key is first added into a filter; (b) then, the key-value pair is added to the data block; (c) once a data block is full, an index entry of this block is generated and added to the index block. In the meantime, a bloom filter entry of the data block is finished. Once finishing the process for all keys, (1) the filter of each data block is formed as a filter block; (2) the metaindex is generated referring to the handle of the filter block; (3) an index-block is added to the segment; (4) a corresponding footer of the segment is formed and added at the tail of the segment.

ALGORITHM 3: Append Segment and Merge Metadata.

```
Input: table: overlapped DTable, seg: segment
Output: SUCCESS/FAILURE
for i ← 0 to seg.datablocknum do
    append(table, seg.block[i]);
end
append(table_filterblock);
append(seg_filterblock);
append(table_overlapped metaindex block);
append(table_indexblock);
append(seg_indexblock);
footer.index_handle ← new_index_handle;
footer.metaindex_handle ← new_meta_index_handle;
append(footer);
return SUCCESS; /* Return FAILURE when any append process fails. */
```

ALGORITHM 4: Seek KV Pairs in a DTable.

```
Input: key
Output: value
/* For each candidate DTable, seeking from segment with the Max.Sequence Number to Minimum. */
for j ← max.seqnum to min.seqnum do
    data_block ← index_block[j].Seek(key);
    if  data_block != NULL && filter.Seek(key) == true then
        value ← data_block.Seek(key);
        return value;
    end
end
```

Append Process: Algorithm 3 demonstrates the process of appending the segment as well as merging the metadata of the segment and the metadata of its overlapped DTable. For the metadata aggregation, the metadata of overlapped DTable is already fetched into memory along with the victim DTable. A segment with its newly generated metadata is appended to the overlapped DTable as follows: (1) append data blocks and overwrite the stale metadata, (2) append the bloom filter block of original overlapped DTable (in memory) to the overlapped DTable, (3) append the new bloom filter block of the segment, (4) append the overlapped metaindex block, (5) append the index blocks of the overlapped DTable together with the index blocks of the segment, (6) form the footer on the basis of the merged metadata, and (7) append the footer to the tail of the DTable.

Seek Process Within a DTable: Due to the benefit from the design of index segment and its sequence number, the seek process within a DTable is demonstrated in Algorithm 4. To find a KV pair in a DTable, the LWC-tree starts with checking the index segment with the largest sequence number. First, it searches the data block whose key range includes the object key. If such a block is found, then it checks the block bloom filter to confirm the existence of the key. If the key exists, then the object value is returned. Otherwise, it repeats the same process in index segment with a decreasing sequence number. With the help of index sequence number, the LWC-tree ensures that only one data block read is needed to find a KV pair.

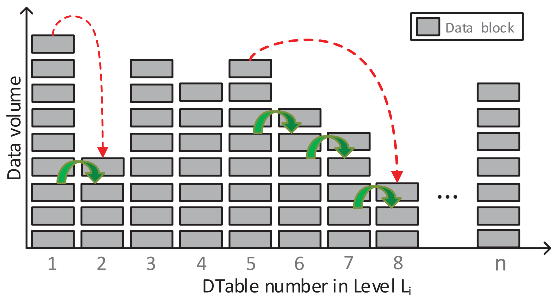


Fig. 7. The imbalanced workload of DTables in Level L_i. This figure shows the imbalanced data distribution of DTables in a level.

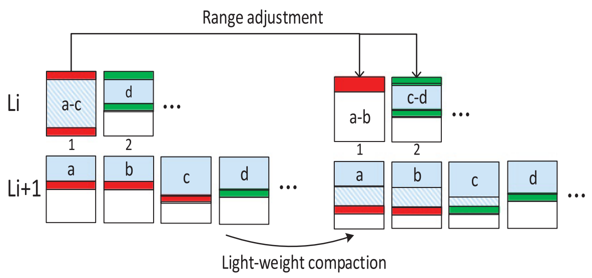


Fig. 8. The workload balance during a compaction. This figure shows the procedure of performing workload balance between two adjacent tables in Level L_i, for example, Table 1 and Table 2. Workload balance is conducted after lightweight compaction, which modifies the key range of imbalanced tables without actual data mitigation and introduces no extra overhead.

3.4 Workload Balance in DTables

Different DTables in the same level of the LWC-tree may have different amount of data due to their represented key ranges. Figure 7 shows the workload distribution across DTables in the same level at a specific time point at runtime. The process of lightweight compaction appends data to each DTable according to its key range, causing the data volume in DTables to become dynamic and imbalanced, which may affect the read performance of the LWC-tree. To address this problem, the LWC-tree attempts to construct a balanced tree in which each DTable has the same data size. As an example in Figure 7, we expect to move the data of overly full table to its siblings, such as migrate the data from Table 1 to Table 2 and from Table 5 to Table 8 to achieve balance.

Therefore, workload balance is proposed to realize the balance of the LWC-tree by delivering the key range of the overly full table to its siblings after lightweight compaction without moving data. Workload balance separates the key range of an overly full table into two parts and shares a part of key ranges to its next neighbor. Therefore, the incoming workload is relocated according to the new key range of DTables, resulting in a quite balanced workload placement. This works because the LWC-tree does not have strict subset relationships across levels.

Figure 8 shows an example of ensuring workload balance in DTables 1 and 2. After DTable 1 with excessive data finishing compaction from L_i to L_{i+1}, the key range of DTable 1 is separated into two parts according to workloads key range distribution. For example, the first part has range "a–b" and the second part has range "c" in Figure 8. Noticing that DTable 2 contains the much smaller data, LWC-tree adds the second part key range "c" into DTable 2, so key ranges of DTable 1 and DTable 2 become "a–b" and "c–d," respectively. Due to the adjustment of the key range of DTables, the incoming workload of DTables is changed, and the workload balance is achieved as a result. Similarly to the key range adjustment between DTable 1 and DTable 2, the LWC-tree consecutively conducts the key range adjustments from DTable 5 until DTable 8 step by step as indicated by the green arrows. Please note that it is not allowed to give a part of key range of DTable 5 to DTable 8 directly, as the keys must be sorted for tables in the same level. The key range redistribution process during compaction imposes almost no extra overhead, as it incurs no data migration.

4 LWC-STORE DESIGN AND IMPLEMENTATION

The demand for high-capacity KV stores in an individual KV server keeps increasing. This rising demand is not only due to data-intensive applications but also because of the virtualization purpose and the cost benefit choice of using fewer servers to host a distributed KV store. Today, it is an economical choice to host a multi-terabytes KV store on one server using either hard disks or SSDs. The proposed LWC-tree is an optimized variant of the LSM-tree and thus is generally applicable for various devices. Therefore, we implement three key-value stores, named LWC-stores, on HDD, SSD, and SMR drives, respectively, based on the LWC-tree. Out of the three storage devices, the SMR drive is one of the most elegant choices to provide large capacity with no significant cost impact [4, 14, 17, 18, 33]. Even though SMR drives have the random write limitation, the LWC-tree is still able to fully utilize the SMR drive and eliminate its weakness on I/O constraint. In this section, we focus on the design and implementation of the LWC-store on an SMR drive.

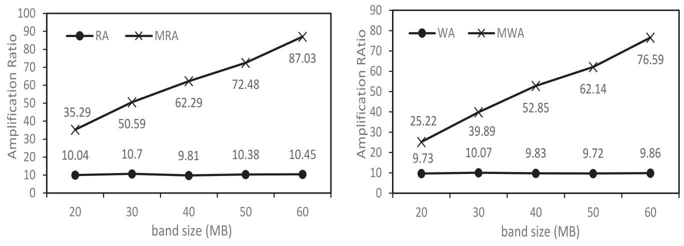
4.1 Auxiliary Write Amplification of SMR Drives

Shingled magnetic recording is leading the next generation of disk technology due to its increased disk areal density. SMR drives achieve disk capacity expansion by overlapping tracks on one another like shingles on a roof. The overlapped tracks of SMR drives bring a serious I/O constraint, where random I/O or in-place update may overwrite the valid data in the band. A solution to handle the access restriction of SMR drives is to enforce an out-of-place update policy, in which the data in SMR drives require a reorganization via performing garbage collection [7]. However, no matter of using the in-place or out-of-place update, SMR drives still generate read/write amplification, named Auxiliary Write or Read Amplification (AWA or ARA). Even worse, the AWA and WA induced by applications create a multiplicative effect on write traffic to SMR drives, as it is with flash devices [38].

Previous work on flash devices has demonstrated an example of this amplification with LevelDB, where a small amount of user writes can be amplified into as much as 40× more writes to the flash device [27]. To reveal the amplification on SMR drives, we experimentally measure the amount of amplification by running LevelDB on an SMR emulator that is artificially banded on top of a conventional HDD [1]. The multiplicative WA and RA (in short MRA, MWA) are shown in Figure 9. As an example, in a 40MB band size SMR drive, the WA and RA from LSM-trees are multiplied by ARA and AWA, increasing from 9.81 and 9.83 to 62.29 and 52.85 respectively. This severe amplification is attributed to two factors: traditional key-value stores being built on top of a shingled disk and Ext4 delivering suboptimal performance [26] due to random accesses and I/O amplification. Ext4 as a mature and widely used file system is not sequentially oriented [2]. Given the multiplicative effect of I/O amplification, it is thus important to minimize additional KV store writes (as what we discussed in Section 3) and reduce the I/O amplification on SMR drives (Section 4.3).

4.2 Previous Work for KV Stores on SMR Drives

In this section, we explore a related prior effort, which represents the state-of-the-art attempt to optimize the key-value storage system on SMR drives.



(a) MRA (b) MWA

Fig. 9. The multiplicative read and write amplification on drives. This figure shows the average I/O amplification and multiplicative I/O amplification at different SMR band sizes.

SMRDB [30] is an SMR friendly Key-value store, which adopts SMR drives for LSM-tree based KV stores to demonstrate that SMR drives are capable of meeting modern storage needs. The main design choices of SMRDB follow. First, SMRDB adopts a fixed-sized model that sizes the SSTable to match the band size with entire band allocation and deallocation. Second, the SMR friendly organization only has two levels (L_0 and L_1), and the SSTables in both levels can have overlapping key range. Third, the background compaction tries to minimize overlapping ranges in level L_1. We re-implement SMRDB as faithfully as possible according to the descriptions in the article. SMRDB is evaluated in Section 5, which basically reflects the performance improvements of limiting the table file size to the band size. However, the design of SMRDB has three shortcomings. First, there exists a dilemma between the band size and the SSTable size in SMRDB, that is, the SSTable in LSM-trees prefers a small size in order to mitigate the data volume of a compaction and reduce the compaction overhead. But the related small band size could damage the space utilization of SMR drives. Second, limiting the number of LSM-tree levels to reduce compaction overhead could have a counter effect in large-scale stores, since it induces more compaction data in each compaction as studied in rRef. [36], which is also confirmed by our experiments in Section 5. Moreover the overlapped key range in both Level L_0 and Level L_1 aggravate the data size of the compaction as well. Generally, SMRDB shows a splendid potential for the interactive of LSM-tree-based KV store and SMR drives, but it demonstrates little improvement on a large dataset.

4.3 The LWC-store on SMR Drives

As shown in the previous section, building KV stores on SMR drives with a traditional Ext4 file system introduces serious multiplicative write amplification. Therefore, developing KV stores on the Ext4 file system based on the LWC-tree can also lead to random disk I/Os due to its non-sequential properties, which aggravates the write amplification of SMR drives. For this reason, we implement the LWC-store on SMR drives without having a file system.

The LWC-store on SMR drives is a key-value store system, which runs directly on top of an SMR drive and manages the underlying disk in an SMR-friendly manner. To get around the I/O constraint of SMR drives and eliminate the I/O amplification, the LWC-store assigns each DTable to a sequential physical block address (PBA) space. In SMR drives, a sequential PBA space is organized as bands, and the invalid data at the tail of a band can be overlapped without suffering from the overwrite penalty. Subsequently, each DTable of LWC-store is mapped to a band, where the table size could be variable but should not be larger than the band size. Using the band as the disk management unit is cost-effective as it does not suffer garbage collection overhead. Figure 10 depicts the overall system architecture of LWC-store on SMR drives. At the high level, the LWC-tree provides a lightweight compaction. At the device level, the SMR device provides dedicated bands

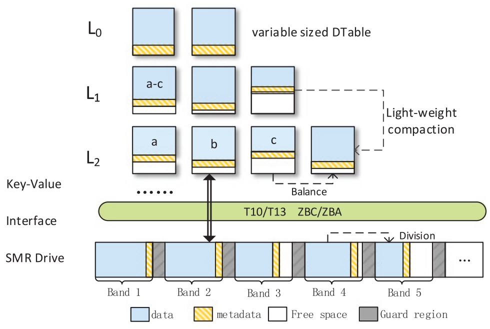


Fig. 10. The system architecture of the LWC-store on SMR drives. In the LWC-store on SMR, the lightweight compaction and workload balance support the efficiency of LWC-tree. The data layout and the division procedure of varying sized DTables support the SMR optimized KV store.

to store the dynamically sized DTables. The interactions between the key-value store and the SMR device can be realized via the ZBC/ZAC interfaces.

In a lightweight compaction, the new segment and metadata are appended to the corresponding band by overwriting the out-of-date metadata. This is viable because SMR drives write data in sequence and overlaps happen in only one direction [14]. In addition, as the metadata are always located at the end of the DTable, overwriting the metadata at the tail of a band would neither damage the valid data nor bring I/O amplification. Comparing to the traditional LSM-tree-based KV stores, LWC-store not only mitigates the write amplification from compactions but also eliminates the auxiliary I/O amplification from SMR drives.

4.4 Equal Division

The implementation of the LWC-store on SMR drives places each DTable in an SMR band, and therefore the maximum size of a DTable is limited by the SMR band. LWC-store expects that each DTable is perfectly filling the band when it becomes the victim DTable and ready to perform compaction, so the data in a DTable would not overflow a band and the KV store would not induce extra space wastage. However, for some workloads with the latest key distribution, the data appended to some DTables during the lightweight compaction may overflow the SMR band. Addressing this problem by continuously compacting the oversized DTable and flushing to lower layers could bring cascading compactions and result in more oversized DTables.

To address the previous problem, equal division of the LWC-store is proposed, which divides an oversized DTable into several sub-DTables and puts the sub-DTables back to the same level. The key range of each sub-DTable is therefore limited to a smaller scope. The division is designed to reserve a suitable free space in a band for the DTable, so the DTable would not be oversized and at the same time the band space would not be wasted. More specifically, when conducting a division, the LWC-store reads out the oversized DTable in L_i from disk, separates the data into n DTables of equal data size and writes the divided DTables back to disk still staying in L_i. As a result, the key range is separated into n parts accordingly. One benefit of our equal division is to keep data sorted within a table. A DTable overflows an SMR band only when it is appended by a lightweight compaction, so division is a part of some compaction process. When a lightweight compaction meets an oversized DTable, it has to wait for a division process to divide the overflowed DTable first. Then the appended data is rearranged and appended to the newly generated DTables.

Table 2. Basic Disk Performance Measurements

	SSD	HDD	SMR
Sequential read (MB/s)	1,200	169	165
Sequential write (MB/s)	901	155	148
Random read 4KB (IOPS)	8,647	64	70
Random write 4KB (IOPS)	19,712	143	5–140

The challenge of implementing equal division is to determine the optimal division number "*n*." Generally, the larger the division number is, the more adequate band space there remains. The sufficient free space in the band ensures high performance by reducing the subsequent divisions, although it may lead to a lower space usage efficiency. On the contrary, a smaller division number could save the band space in some degree but damage the performance due to more frequent divisions. By default, we choose the division number *n* as the amplification ratio between adjacent levels. However, we have also conducted a sensitivity study on the division number to see how it affects the performance and space usage efficiency.

The algorithm of division can be optimized by considering more parameters when we choose a division number, for example, the hotness and the age of the DTable. By having a set parameters, we can dynamically pick a more concise division number for each overflowed DTable, which not only improves the space utilization but also decreases further divisions. This will be a part of our future works.

5 EVALUATION

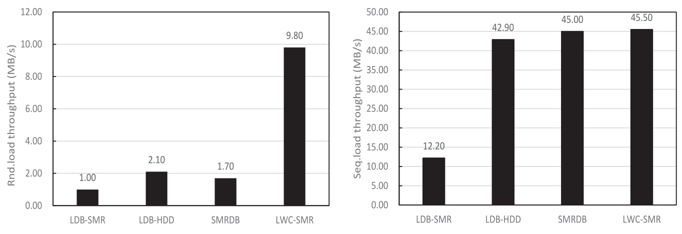
To evaluate the efficiency and applicability of our LWC-tree, we conduct extensive experiments on LWC-stores that build on LWC-trees. The aim of evaluations is to demonstrate that the LWC-store reduces the I/O amplification and improves overall performance on a wide variety of system configurations and various storage devices (SMR, SSD, and HDD), especially for SMR drives. The experiment result of our LWC-store on SMR drives is reported in Section 5.1 and the LWC-store on SSDs and HDDs are described in Section 5.2.2.

The test machine used for our experiments has 16 Intel(R) Xeon(R) CPU E5-2660 0 @ 2.20GHz processors and 64GB of memory. The kernel version is 64-bit Linux 4.4.0-31-generic, and the file system to support conventional LevelDB is Ext4. The storage devices used for the test are a 1TB Seagate ST1000DM003 HDD, a 5TB Seagate ST5000AS0011 SMR drive, and a 400GB Intel P3700 SSD. Table 2 lists the performance characteristics of these devices. As we can see, the SMR drive shows a similar sequential I/O performance and random read performance compared with the conventional HDD, while the random write restriction of an SMR drive is obvious, which is consistent with the findings in existing works [1]. In addition, the Intel P3700 SSD exhibits much better performance than the HDD and SMR drive. To compare the key-value stores fairly, we limit the memory size in 4GB for all the competitors to fully expose their properties, instead of showing the performance upgraded by the memory.

5.1 Experiment Results

In our experiments, we first demonstrate the superiority of the design strategies of the LWC-store using the LWC-store on SMR drives. The evaluations are conducted in the following configurations:

LevelDB on HDDs (LDB-hdd): Running the original LevelDB on a conventional HDD, which represents the baseline for our evaluations.



(a) Random load throughput (b) Sequential load throughput

Fig. 11. Load Performance. This figure shows random and sequential load performance of four different key-value stores. Our proposed LWC-smr on an SMR drive outperforms LDB-smr dramatically in both random and sequential load.

LevelDB on SMR drives (LDB-smr): Running the original LevelDB on a Seagate drive-managed SMR drive, which induces serious multiplicative I/O amplification due to the incompatibility between the traditional LSM-tree-based key-value store and the SMR device as mentioned in Section 4.1.

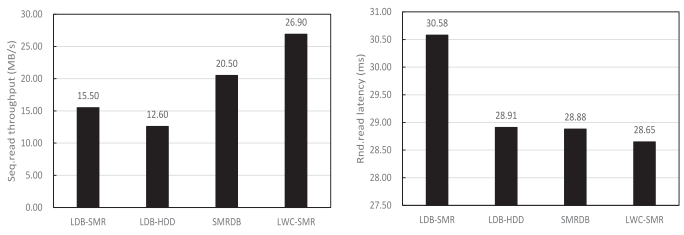
SMRDB: SMRDB is an SMR drive optimized key-value store, which proposes an SMR-friendly solution to improve the performance of the KV store on SMR drives [30]. The main design choices of SMRDB are to reduce the LSM-tree levels to only two levels (i.e., L_0 and L_1), allow the key ranges overlapped in the tables at the same level, and match the SSTable size with the band size (40MB by default).

LWC-store on SMR drives (LWC-smr): LWC-smr is our proposed key-value system, which includes the lightweight compaction in the LWC-tree and tables with size variation in the SMR drive. LWC-smr is implemented based on LevelDB 1.19 [16] and the SMR emulator [1], as discussed in Section 4.3. The default band size is 40MB, and the DTable size is variable but not larger than the band size.

In the main evaluation, we use the default micro-benchmarks in LevelDB to evaluate the overall performance of the four configurations. The default key size and value size are 16B and 4KB, respectively.

5.1.1 Load Performance. We first examine the random load and sequential load performance in each KV store system by inserting a total size of 100GB KV pairs with sequential keys and uniformly distributed keys. For both sequential load and random load, the benchmark is constructed by 25 million entries. The major difference of the two workloads is that inserting entries sequentially does not incur compactions, while random load does. Random load performance is significantly influenced by the compaction efficiency.

Figure 11(a) gives comparisons of the random load performance of the four key-value systems. From this figure, we have three observations. First, LWC-smr improves the random load performance by 9.8 times that of LDB-smr. One reason for that is LevelDB conducts costly compaction procedures constantly, which pull the random load performance down, as mentioned in Section 2.2. Another reason is that LWC-smr is built directly on the SMR drive and free from file system overheads, while LDB-smr suffers from excessive Ext4 overhead with the SMR drive [2, 26], as mentioned in Section 4.1. Second, LWC-smr outperforms LDB-hdd by 4.67 times, which verifies that by adopting LWC-smr, SMR devices deliver an even better random load performance than conventional HDDs. Moreover, LDB-hdd also suffers from the amplification from the LSM-tree in LevelDB. Third, LWC-smr delivers a better random load performance than SMRDB due to the restrictions of SMRDB's two-level construction and the key range overlaps in the same level,



(a) Sequential read throughput (b) Random read latency

Fig. 12. Read Performance. This figure shows random and sequential read performance of four different key-value stores. The y-axis unit of sequential read is the throughput in MB/s, and the y-axis unit of random read is the latency per operation in milliseconds.

which increases the data size for a compaction and has limited effect to improve the random load performance. In addition, as the database grows larger, the weakness of the two-level LSM-tree becomes more evident [36]. In general, our LWC-smr achieves significant advantages because of the lightweight compaction in the LWC-tree and the absence of file system overheads. Moreover, the sequential write property of the LWC-smr contributes to its leading position in write performance comparisons.

Figure 11(b) compares the sequential load throughput of the four key-value store systems. From this figure, we can obtain three conclusions. First, no matter which key-value system is used, the corresponding sequential load performance is much better than the random load performance, which can be mainly attributed to the compactions caused by random load. Second, LWC-smr and SMRDB both deliver comparable performance as LDB-hdd, which is due to the absence of compactions in sequential load. Third, LDB-smr is about 3.7× inferior to other three systems. This is because of the embedded indirection of drive managed SMR drives and the non-sequential Ext4 file system [2, 26].

5.1.2 Read Performance. This experiment is to evaluate the read performance including sequential read and uniformly distributed random read by looking up 100K entries in a 100GB random load database. Figure 12 presents the sequential and random read performance. We make three conclusions from the results of sequential lookup. First, LWC-smr obtains the best performance compared to the other counterparts due to the big DTable and the sequential data layout on the SMR drive. Second, LWC-smr improves the sequential read performance to 1.31× that of SMRDB since LWC-smr has no overlapped tables in the same level. Third, LDB-smr outperforms LDB-hdd in sequential read due to the internal cache of drive-managed SMR drives. The major difference between the random lookup and the sequential read is that LDB-smr suffers from degraded performance compared to the other three key-value stores, as the internal cache of drive-managed SMR has limited efficiency for random read. In addition, both LWC-smr and SMRDB exhibit a comparable random read performance, meaning that if designed properly, one can expect SMR devices to deliver the same level of performance as HDDs while enjoying their capacity benefits.

5.1.3 Performance of Lightweight Compaction. Compactions in a key-value system consume a large percentage of device bandwidth [36, 39], which influences the overall performance by blocking foreground requests. Therefore, the efficiency of compactions affects the system performance significantly. To microscopically investigate the compaction behaviors, we randomly load a 40GB database and record the number of compactions, data size of each compaction, and the total time taken to complete the compactions of the four key-value systems.

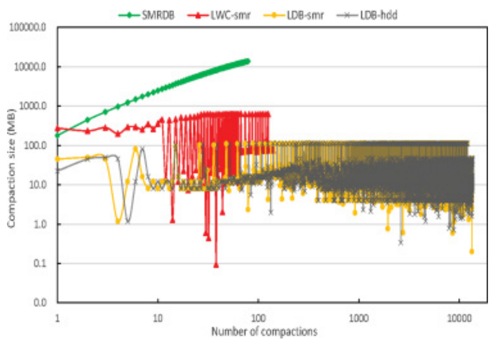


Fig. 13. The microscopic view of compactions. This figure shows the compactions resulting from randomly loading a 40GB database to the four KV systems. The *x*-axis denotes the number of compactions and the *y*-axis denotes the data size in each compaction.

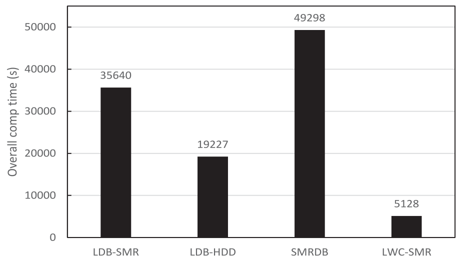
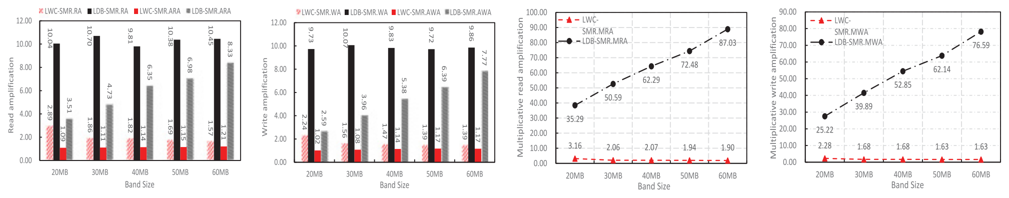


Fig. 14. The overall compaction time. This figure shows the overall consumed time on compactions for randomly loading a 40GB database to four KV systems. The *y*-axis unit is the time in seconds.

Figure 13 depicts the number of compactions and the corresponding written data size. From this figure, we obtain a primary observation that LDB-smr and LDB-hdd have much more compactions than SMRDB and LWC-smr, while SMRDB has larger compaction data size relative to other counterparts. That is to say, although SMRDB reduces the number of compactions of LSM-tree, it, however, incurs larger compaction size, unfortunately. The reason is that SMRDB allows a key range to overlap in Level L_0 and Level L_1, and a compaction in SMRDB involves all the tables with overlapped key ranges. By contrast, LWC-smr has smaller compaction data size (about 45× less than the size of SMRDB) and fewer compaction numbers, resulting in LWC-smr delivering a better compaction efficiency.

The overall time spent on compactions fully reflects the influence of data size and the number of compactions of different KV stores. Figure 14 gives the overall compaction time for randomly loading a 40GB database of four KV systems. From this figure, we can draw a conclusion that the lightweight compaction in LWC-smr gets the highest efficiency, which is 7×, 3.75×, and 9.61× faster than that of LDB-smr, LDB-hdd, and SMRDB respectively, explaining the performance advantages observed in preceding sections.

5.1.4 I/O Amplification. The I/O amplification of a key-value system on an SMR drive is generated by compaction and multiplied by the auxiliary amplification from SMR drives, as discussed in Section 4.1. In this subsection, we evaluate the amplification (RA/WA), auxiliary amplification (ARA/AWA) and the multiplicative amplification (MRA/MWA), respectively, to explore why LWC-smr improves the performance over the LSM-tree-based key-value store.



(a) RA and ARA (b) WA and AWA (c) MRA (d) MWA

Fig. 15. I/O amplification. Panels (a) and (b) show the read or write amplification from KV stores (RA/WA) and the read or write amplification from SMR drives (ARA/AWA). Panels (c) and (d) show the multiplicative read or write amplification (MRA/MWA) of LWC-smr and LDB-smr. The *y*-axis denotes the read/write amplification and the *x*-axis denotes different SMR band sizes.

Figure 15(a) and Figure 15(b) present the read/write amplifications of trees and auxiliary amplification of SMR drives in both LWC-smr and LDB-smr with different band sizes. From these figure, we can draw two conclusions. First, LWC-tree outperforms the LSM-tree by reducing the read and write amplification 5.49× and 6.32× on average. This is because the lightweight compaction in the LWC-tree reduces the total compaction data size for writing key-value pairs. Second, in the device level the LWC-smr outperforms LDB-smr by eliminating AWA and ARA. This is because LWC-smr observes the random write constraint of SMR and never overlaps valid data on disk. On the one hand, the compaction process adapt append basically while only overlap the stale metadata at the tail of a table; on the other hand, it assign the table to dedicate band to prevent the cross-bands table, which aggravates the write amplification from SMR drives. In addition, we observe that the auxiliary amplification of LDB-smr increases with the band size, while LWC-smr does not. For example, the AWA of LDB-smr increases from 2.59 to 7.77, while the AWA of LWC-smr remains around 1.14. Figures 15(c) and (d) depict the multiplicative amplifications with different band sizes. The overall write amplification, MWA, is WA multiplied by AWA, so as MRA (i.e., MRA=RA × ARA). This figure shows that LWC-smr reduces the overall read amplification (MRA) by a factor of 11.17 to 45.80, and the overall write amplification (MWA) by a factor of 11.06 to 46.99, which validates the effectiveness of the design strategies of the LWC-smr.

5.1.5 Performance Benefit Analysis. Above results show that our LWC-store significantly improves performances. In this section, we analyze where the performance benefit comes from. Write amplification is a dominant factor influencing and reflecting the overall performance variation consistently, so we take write amplification to evaluate and analyze the contribution of our design choices separately. Specifically, the performance benefit of the LWC-store on SMR drives comes from two parts. The first part is the optimization on the index structure, which is the backbone of key-value stores. Compared with the WA (write amplification) of original LDB-smr, this part of contribution is defined by the WA ratio of LDB-smr to LWC-smr. The second contributor is the data layout and the data management on SMR drives, where LWC-smr changes all read-modify-write operations into appends and accommodate each table with a dedicate band. The contribution of design choices on SMR drives is defined by the mitigation of auxiliary write amplification compared with the original LDB-smr, which is figured by the AWA ratio of LDB-smr to LWC-smr. The overall write amplification improvement is defined as the MWA ratio of LDB-smr to LWC-smr.

Figure 16 shows the performance (i.e., write amplification) improvement ratio of LWC-smr to LDB-smr on an emulated SMR drive with five different band sizes. The WA and AWA ratio of LDB-smr to LWC-smr reflect effects of two contributors. Generally, the index structure mitigates write amplification by 6.3× on average, and the data layout and data management on SMR drives

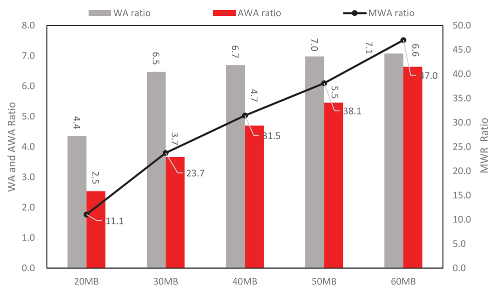
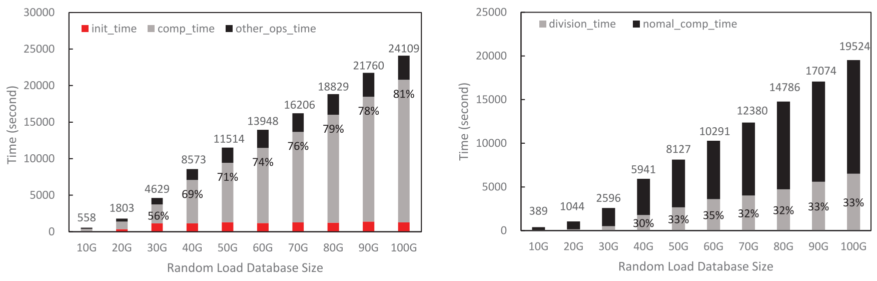


Fig. 16. Contribution analysis of two design choices. This figure depicts the contributions of two factors, that is, the lightweight compaction tree and the data management on SMR drives. The contribution of LWC-tree is defined by the write amplification ratio of LDB-smr to LWC-smr. The contribution on SMR drives is defined by the auxiliary write amplification ratio of LDB-smr to LWC-smr. These two contributors have a multiplicative effect in overall performance improvement.



(a) Load time breakdown (b) Compaction time breakdown

Fig. 17. Execution time breakdown. (a) The overall execution time for randomly loading 10 databases. The overall execution time breaks into three parts, that is, the time for LWC-tree initialization, the time for lightweight compactions, and the time for other operations. The number on each bar is the overall execution time, and the percentage of compaction shows the proportion of lightweight compactions in overall execution time. (b) A concrete compaction time of each random load database. The compaction time comes from two parts, that is, the original lightweight compactions and the compactions waiting for division. The number on each bar shows the overall compaction time of each database, and the percentage of division shows the proportion of divisions in overall compaction time.

mitigate the auxiliary amplification by 4.6× on average. The overall performance gain grows with the band size, and it is the result of the WA times the AWA. In conclusion, optimizations on both index structures and SMR drives improve the system performance significantly and they result in a multiplicative effect.

To illustrate the performance improvement explicitly, we now evaluate the overall random load execution time of LWC-smr in 10 different databases and record the time consumption of each kind of operation. The random load process is composed of three kinds of operations generally, initialization operations, lightweight compaction procedures, and other access operations. In LWC-smr, we initiate the key value store by dividing the victim DTable into AF overlapped Dtables and putting them into next level, when the next level has no DTable whose key range overlapped the victim DTable. Figure 17(a) shows the execution time of three kinds of operations when randomly load 10 different databases. From this figure, we can make two observations. First, the compaction process dominates the overall execution time, which explains why the optimization of compactions

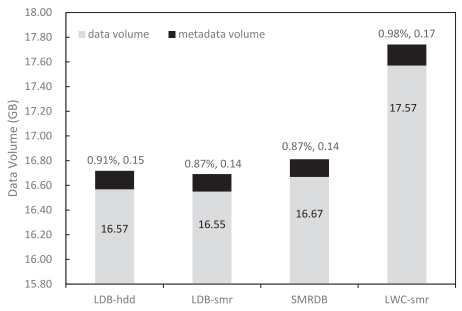


Fig. 18. Space overhead of data and metadata. This figure shows the physical space overhead for storing the 40GB randomly loaded database of four different key-value stores. The number on the bar of data is the data volume. The two numbers on the bar of metadata are the metadata volume and the ratio of metadata to data.

leads to a significant performance improvement. Second, with the growth of the data volume, the proportion of compaction increases, because for a large database, more data are compacted into the lower level of LWC-trees.

Except for design choices that benefit the system performance, we have made some compromise for the system integrity, for example, division (as described in Section 4.4). Although the process of division consumes parts of bandwidth and sacrifices the performance, it has to be executed when a table overflows an SMR band for the data layout on SMR drives. In LWC-smr, division takes part of the lightweight compaction time, since an SMR band only overflows when a segment appends during a lightweight compaction. Hence, a compaction requires to wait until the overflowed DTable finishes its division. Figure 17(b) shows the time consumption of division and the normal compaction without divisions. The number on each bar is the overall compaction time, and the percentage of divisions shows the overhead of divisions in compactions. From this figure, we can make two conclusions. First, the division time takes up to 34% of overall compaction time in databases of different sizes. Second, although division procedures takes part of the compaction time, the overall compaction time of lightweight compaction is less than that of the conventional compaction process in other competitors, as demonstrated in Section 5.1.3.

5.2 Further Discussion

5.2.1 Space Overhead. Space overhead is a significant concern for all systems with space to exchange for time. Our proposed LWC-tree has a tradeoff between space and time for saving the bandwidth of key-value storage systems. The space overhead of LWC-tree comes from two parts. First, LWC-tree stores the metadata of overlapped DTables to its victim DTable, thus the duplicated metadata of overlapped DTables introduces space overhead. However, in this way we gather all the required data and metadata in a single victim DTable, mitigating the random read and write for metadata in each lightweight compaction. It brings significant improvement on performance and I/O amplification as shown in prior sections. Second, the stale data in prior appended segments only can be cleaned when the DTable is selected as a victim DTable according to the design of the lightweight compaction; thus invalid data stay in DTables for a period and require space overhead.

To clarify the space overhead of LWC-tree, we take an example of LWC-smr that applies LWC-trees on SMRs. First, we randomly load a 40GB database for each key-value store and then evaluate their space overhead of data and metadata, respectively. Figure 18 shows results of our evaluation. Three observations can be concluded from this figure. First, LDB-hdd, LDB-smr, and SMRDB have

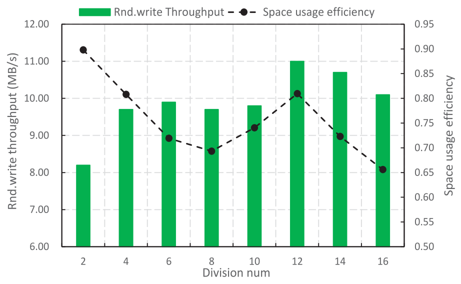


Fig. 19. Random write throughput and space usage efficiency of different division number. This figure shows the influence of division number on the performance and the space usage efficiency of LWC-smr.

similar amount of data as they load the same volume database and share the same index structure, that is, the LSM-tree. Nevertheless, the data volume of our LWC-smr is larger than the other three counterparts. The data amount of LWC-smr is 6.05% more than that of the LDB-hdd, because LWC-smr saves stale data in appended segments, as mentioned previously. Second, the counterparts of LWC-smr have similar amount of metadata, while LWC-smr has more metadata for storing the duplicated metadata in victim DTables. Third, the ratio of metadata to data of LWC-smr is 0.98%, which is more than that of SMRDB, LDB-smr, and LDB-hdd.

In conclusion, LWC-smr slightly increases the amount of data, metadata, and the ratio of metadata compared to LDB-hdd, LDB-smr, and SMRDB. Such a small space overhead (up to 6%) is acceptable for persistent hard disk drives, considering that LWC-stores bring significant performance improvement (e.g., random load increased by up to 467% compared to LDB-hdd) as demonstrated in prior sections.

5.2.2 *Sensitivity Study.*

In this section, we perform a sensitivity study on several parameters, including division numbers, value size, and storage medium. For this sensitivity study, we use the micro benchmarks distributed with LevelDB code and set the database size to be 100GB.

Division Numbers: To examine the impacts of division number and pick an optimal division number for LWC-smr, we conduct experiments with division number ranging from 2 to 16. Figure 19 shows the random write throughput and the corresponding space usage efficiency of LWC-smr. The space usage efficiency is defined as the ratio of required storage space to actual used disk space. From this figure, we can make two observations. First, a small division number helps to maintain a high space usage efficiency but hurts the random write performance as it incurs divisions frequently. On the contrary, a large division number hurts the space usage efficiency badly but delivers a better random write performance. In conclusion, 12 is a relative good choice for the division number of LWC-smr, as it achieves a good tradeoff point between the space usage efficiency and the random write performance.

Value Size: To examine the impacts of value size, we perform another set of 100GB random write with the value size of 64B, 256B, 512B, 1KB, 4KB, 16KB, and 64KB, respectively. Figure 20 shows the random write throughput of three key-value store systems. Generally, the throughput of LWC-smr, LDB-hdd, and LDB-smr increase with the value size. The relative improvement of LWC-smr increases with the value size as well. For example, the random write throughput of LWC-smr increases from 1.77× to 5.51× that of LDB-hdd when the value size increases from 64B to 64KB. This is because LWC-trees merge less metadata as the value size increased, while compaction data volume do not change in LSM-trees.

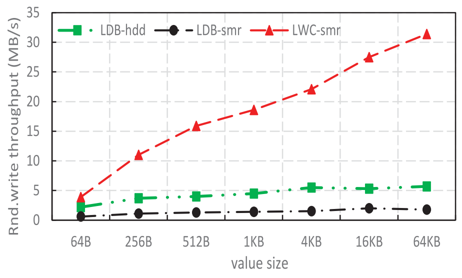


Fig. 20. Random write throughput in varying value sizes. This figure shows that the LWC-tree exhibits advantage in all the examined value sizes and this performance advantage becomes more prominent with the growth of the value size.

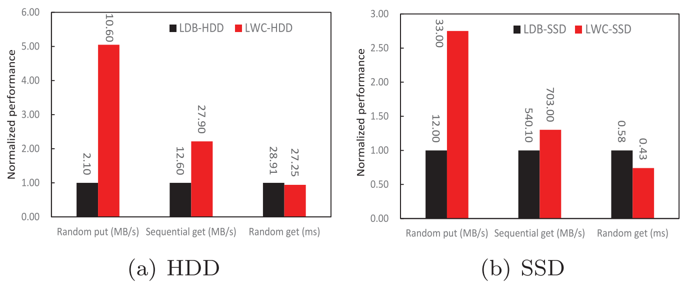


Fig. 21. Performance comparison in HDD and SSD. The *x*-axis denotes the different performance measurements including: random write, sequential read, and random read. The *y*-axis denotes the normalized performance relative to LDB-HDD. The number on top of each bar is the actual performance value.

Storage Medium (HDD and SSD): To explore the applicability of LWC-trees in different storage mediums, besides the experiments of the LWC-store on SMR drives we have done so far, we then evaluate LWC-stores on HDDs and SSDs. We report the performance comparisons in following two configurations: (1) LevelDB on conventional HDD (LDB-hdd), where the original LevelDB runs on a conventional HDD, and (2) LWC-store on conventional HDD (LWC-hdd), where the LWC-store runs on top of a conventional HDD without a file system in between. LWC-hdd's performance difference relative to LDB-hdd reflects the applicability of our designs to HDDs. Figure 21(a) compares the LevelDB and LWC-store on an HDD in terms of the performance of random write, random read, and sequential read. From this figure, we can make three observations. First, LWC-hdd improves the random load performance by 5.05 times that of LDB-hdd. This is attributed to the efficient lightweight compaction which mitigates the I/O amplification in the LWC-tree significantly. Second, LWC-hdd improves the sequential read performance by 2.01 times that of LDB-hdd for LWC-hdd's sequential on-disk data layout. Third, LWC-hdd delivers a comparable random read performance as LDB-hdd. Figure 21(b) gives a similar comparison but on an SSD. On the high level, we can make similar observations as it is with the HDD medium in Figure 21(a). However, a major difference is that, the absolute performance improvement ratio in an SSD is slightly smaller than that in an HDD, which is attributed to that LWC-store is designed to improve the throughput by reducing amplification. Since the bandwidth of HDD is far less than that of SSD, the impact of write amplification under SSD is not as serious as that under HDD and SMR drives, limiting the potential improvement of the LWC-store on an SSD.

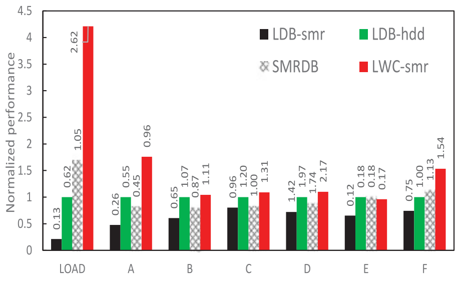


Fig. 22. YCSB Macro-benchmark Performance. This figure shows the performance normalized to LDB-hdd. The *x*-axis represents different workloads, the *y*-axis denotes the normalized values, and the number on each bar shows the actual throughput (K ops/s). Workload A is composed with 50% reads and 50% updates, Workload-B has 95% reads and 5% updates, and Workload-C includes 100% reads. Workload-D has 95% reads and 5% insert latest keys. Workload-E has 95% range queries and 5% insert new keys, Workload-F includes 50% reads and 50% RMW.

5.2.3 *Macro-Benchmark.*

The YCSB benchmarks [10] provide a framework and a set of seven workloads applicable for evaluating the performance of key-value stores. We perform YCSB benchmarks as a supplement to verify the performance advantages of the LWC-smr. We first load 100GB database and then test the performance in different workloads with 100k entries. Figure 22 compares the throughput of the four key-value systems. The *y*-axis in the figure gives the normalized comparison results relative to LDB-hdd. The primary conclusion is that LWC-smr outperforms other systems in all workloads, especially for random load. This is because the LWC-smr has greater advantages for random load over read, which accords with the experiment results in preceding sections.

6 RELATED WORK

As the backbone of many data center applications, key-value stores have recently become a hot research interest. Various key-value stores have been proposed for specific storage medium.

Wisckey [25] is a flash optimized key-value store, which separates keys from values to reduce the I/O amplification by mitigating migrations of values. The unsorted values degrade the sequential read performance of LSM-trees, so Wisckey is not optimal for conventional HDDs and SMR drives for their poor random read performance. NVMKV [27] is an FTL-aware light weight KV store that leverages native FTL capabilities to provide high performance. SkimpyStash [12] is RAM space skimpy key-value store on flash-based storage, which moves a part of tables to the SSD using a linear chaining. FlashStore [11] is a high-throughput persistent key-value store using cuckoo hashing. LOCS [35] is an LSM-tree-based KV store on SSD that exposes the internal flash channels to applications for a better cooperation. SILT [24] combines the log-structure, hash-table, and sorted-table layouts to provide a memory-efficient KV store. ZEA [26] is a data management approach for SMR. It finds that the widely applicable Ext4 file system is not strictly sequential, and hence it designed a zone-based extent allocator that provides a building block for higher-level abstractions by mapping ZEA logical blocks to LBAs of the disk. Specifically, ZEA illustrates its advantages over SMR drives by a specifically designed file system, the ZEAFS file system. ZEAFS offers serialized accesses to the defined LBA regions and delivers similar performance as the LevelDB on conventional HDDs with ext4. However, ZEAFS has several deficiencies in space utilization and reclamation since it induces free space debris and engenders the complexity of garbage collection. Memcached [15] and Redis [31] are the popular memory KV implementations. GD-Wheel [23] provides a cost-aware replacement policy for memory-based KV stores, which takes access recency and computation cost into account. It is reasonable that a full understanding of storage devices, for example, SSD [19, 20, 34, 40, 41], contributes to a better design of device specific key-value stores. Our work optimizes the key-value store for SMR drives that are widely deployed due to its capacity advantages.

Other researches are dedicated to develop key-value stores for specific scenarios. zExpander [37] dynamically partitions the cache into two parts for the high memory efficiency and low miss ratio, respectively. ForestDB [3] addresses the performance degradation of large keys by employing a new hybrid index scheme. LSM-trie [36] constructs a prefix tree to store data in a hierarchical structure that helps to reduce the metadata and the write amplification. bLSM [32] proposes a "spring and gear" merge scheduler, which bounds write latency and provides a high read and scan performance. Atlas [21] is a key-value storage system for cloud data that stores keys and values on different hard drives. Among these works, bLSM, LSM-trie, Wisckey, VT-tree, Altas, and LOCS are optimized for traditional LSM-tree-based key-value stores.

7 CONCLUSIONS

The overall performances of the LSM-tree-based key-value stores are seriously decreased by the constantly occurring compactions. Running key-value stores on different storage devices can lead to different degrees of influence, especially on SMR drives with severe multiplicative I/O amplification. In this article, we first propose the LWC-tree with lightweight compaction to reduce the I/O amplification by appending data during the compaction. Then, we design and implement three high performance key-value store systems for SMR drives, conventional HDD and SSD, respectively. Extensive experiments have demonstrated that our LWC-store significantly improves the random write throughput compared with other schemes, including SMRDB and LevelDB, by up to 4.67 times. Furthermore, we have also experimentally demonstrated that the performance improvement remains across different storage mediums, value sizes and access patterns.

REFERENCES

[1] Abutalib Aghayev and Peter Desnoyers. 2015. Skylight a window on shingled disk operation. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST'15)*.

[2] Abutalib Aghayev, Theodore Tso, Garth Gibson, and Peter Desnoyers. 2017. Evolving Ext4 for shingled disks. In *Proceedings of 15th USENIX Conference on File and Storage Technologies (FAST'17)*, Vol. 1. 105.

[3] Jung-Sang Ahn, Chiyoung Seo, Ravi Mayuram, Rahim Yaseen, Jin-Soo Kim, and Seungryoul Maeng. 2016. ForestDB: A fast key-value storage system for variable-length string keys. *IEEE Trans. Comput.* 65, 3 (2016), 902–915.

[4] Ahmed Amer, Darrell D. E. Long, Ethan L. Miller, Jehan-Francois Paris, and S. J. Thomas Schwarz. 2010. Design issues for a shingled write disk system. In *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST'10)*.

[5] Apache. 2007. HBase. Retrieved from http://hbase.apache.org/.

[6] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. 2012. Workload analysis of a large-scale key-value store. *ACM Sigmetr. Perf. Eval. Rev.* 40, 1 (2012), 53–64.

[7] Yuval Cassuto, Marco A. A. Sanvido, Cyril Guyot, David R. Hall, and Zvonimir Z. Bandic. 2010. Indirection systems for shingled-recording disk drives. In *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST'10)*. IEEE, 1–14.

[8] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Michael Burrows, Tushar Chandra, Andrew Fikes, and Robert Gruber. 2006. Bigtable: A distributed storage system for structured data. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI'06)*. 205–218.

[9] Brian F. Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, Hans-Arno Jacobsen, Nick Puz, Daniel Weaver, and Ramana Yerneni. 2008. PNUTS: Yahoo!s hosted data serving platform. In *Proceedings of the VLDB Endowment (PVLDB'08)*.

[10] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *Proceedings of the ACM Symposium on Cloud Computing (SOCC'10)*.

[11] Biplob Debnath, Sudipta Sengupta, and Jin Li. 2010. FlashStore: High throughput persistent key-value store. *Proc. VLDB Endow.* 3, 1–2 (2010), 1414–1425.
[12] Biplob Debnath, Sudipta Sengupta, and Jin Li. 2011. SkimpyStash: RAM space skimpy key-value store on flash-based storage. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*. ACM, 25–36.
[13] Facebook. 2016. RocksDB, A persistent key-value store for fast storage enviroments. Retrieved from http://rocksdb.org/.
[14] Tim Feldman and Garth Gibson. 2013. Shingled magnetic recording: Areal density increase requires new data management. *USENIX* 38, 3 (2013), 22–30.
[15] Brad Fitzpatrick and Anatoly Vorobey. 2011. Memcached: A distributed memory object caching system. https://memcached.org/.
[16] Sanjay Ghemawat and Jeff Dean. 2016. LevelDB. Retrieved from https://github.com/Level/leveldown/issues/298.
[17] Garth Gibson and Greg Ganger. 2011. Principles of operation for shingled disk devices. Carnegie Mellon University Parallel Data Lab Technical Report CMU-PDL-11-107 (2011).
[18] Weiping He and David H. C. Du. 2017. SMaRT: An approach to shingled magnetic recording translation. In *Proceedings of the 15th USENIX Conference on File and Storage Technologies (FAST'17)*. 121.
[19] Ping Huang, Pradeep Subedi, Xubin He, Shuang He, and Ke Zhou. 2014. FlexECC: Partially relaxing ECC of MLC SSD for better cache performance. In *Proceedings of the USENIX Annual Technical Conference*. 489–500.
[20] Ping Huang, Guanying Wu, Xubin He, and Weijun Xiao. 2014. An aggressive worn-out flash block management scheme to alleviate SSD performance degradation. In *Proceedings of the 9th European Conference on Computer Systems*. ACM, 22.
[21] Chunbo Lai, Song Jiang, Liqiong Yang, Shiding Lin, Guangyu Sun, Zhenyu Hou, Can Cui, and Jason Cong. 2015. Atlas: Baidu's key-value storage system for cloud data. In *Proceedings of the 2015 31st Symposium on Mass Storage Systems and Technologies (MSST'15)*. 1–14.
[22] Avinash Lakshman and Prashant Malik. 2009. Cassandra: A decentralized structured storage system. In *Proceedings of the 3rd ACM SIGOPS International Workshop on Large Scale Distributed Systems and Middleware*.
[23] Conglong Li and Alan L. Cox. 2015. GD-wheel: A cost-aware replacement policy for key-value stores. In *Proceedings of the 10th European Conference on Computer Systems*. 5.
[24] Hyeontaek Lim, Bin Fan, David G. Andersen, and Michael Kaminsky. 2011. SILT: A memory-efficient, high-performance key-value store. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles*. 1–13.
[25] Lanyue Lu, Thanumalayan Sankaranarayana Pillai, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2016. WiscKey: Separating keys from values in SSD-conscious storage. In *Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST'16)*. 133–148.
[26] Adam Manzanares, Noah Watkins, Cyril Guyot, Damien LeMoal, Carlos Maltzahn, and Zvonimr Bandic. 2016. ZEA, A data management approach for SMR. In *Proceedings of the 8th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage'16)*.
[27] Leonardo Marmol, Swaminathan Sundararaman, Nisha Talagala, Raju Rangaswami, Sushma Devendrappa, Bharath Ramsundar, and Sriram Ganesan. 2014. NVMKV: A scalable and lightweight flash aware key-value store. In *Proceedings of the 6th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage'14)*.
[28] Chris Nyberg, Tom Barclay, Zarka Cvetanovic, Jim Gray, and Dave Lomet. 1994. AlphaSort: A RISC machine sort. In *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data (SIGMOD'94)*.
[29] Patrick ONeil, Edward Cheng, Dieter Gawlick, and Elizabeth ONeil. 1996. The log-structured merge-tree (LSM-tree). *Acta Inform.* 33, 4 (1996), 351–385.
[30] Rekha Pichumani, James Hughes, and Ethan L. Miller. 2015. SMRDB: Key-value data store for shingled magnetic recording disks. In *Proceedings of the ACM International Systems and Storage Conference (SYSTOR'15)*.
[31] Salvatore Sanfilippo and Pieter Noordhuis. 2009. Redis. Retrieved from http://redis.io/.
[32] Russell Sears and Raghu Ramakrishnan. 2012. bLSM: A general purpose log structured merge tree. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data (SIGMOD'12)*.
[33] I. Tagawa and M. Williams. 2009. High density data-storage using shingled write. In *Proceedings of the IEEE International Magnetics Conference (INTERMAG'09)*.
[34] Hua Wang, Ping Huang, Shuang He, Ke Zhou, Chunhua Li, and Xubin He. 2013. A novel I/O scheduler for SSD with improved performance and lifetime. In *Proceedings of the 2013 IEEE 29th Symposium on Mass Storage Systems and Technologies (MSST'13)*. IEEE, 1–5.
[35] Peng Wang, Guangyu Sun, Song Jiang, Jian Ouyang, Shiding Lin, Chen Zhang, and Jason Cong. 2014. An efficient design and implementation of LSM-tree based key-value store on open-channel SSD. In *Proceedings of the 9th European Conference on Computer Systems*. 16:1–16:14.
[36] Xingbo Wu, Yuehai Xu, Zili Shao, and Song Jiang. 2015. LSM-trie: An LSM-tree-based ultra-large key-value store for small data. In *Proceedings of the USENIX Annual Technical Conference (USENIX'15)*.

[37] Xingbo Wu, Li Zhang, Yandong Wang, Yufei Ren, Michel Hack, and Song Jiang. 2016. zExpander: A key-value cache with both high performance and fewer misses. In *Proceedings of the 11th European Conference on Computer Systems*. ACM, 14.

[38] Jingpei Yang, Ned Plasson, Greg Gillis, and Nisha Talagala. 2013. HEC: Improving endurance of high performance flash-based cache devices. In *Proceedings of the 6th International Systems and Storage Conference*. ACM, 10.

[39] Yinliang Yue, Bingsheng He, Yuzhe Li, and Weiping Wang. 2017. Building an efficient put-intensive key-value store with skip-tree. *IEEE Transactions on Parallel and Distributed Systems* 28, 4. IEEE.

[40] Ke Zhou, Shaofu Hu, Ping Huang, and Yuhong Zhao. 2017. LX-SSD: Enhancing the lifespan of NAND flash-based memory via recycling invalid pages. In *Proceedings of the 2017 IEEE 33rd Symposium on Massive Storage Systems and Technology (MSST'17)*.

[41] You Zhou, Fei Wu, Ping Huang, Xubin He, Changsheng Xie, and Jian Zhou. 2015. An efficient page-level ftl to optimize address translation in flash memory. In *Proceedings of the 10th European Conference on Computer Systems*. ACM, 12.

Received August 2017; accepted September 2017