A Light-weight Compaction Tree to Reduce I/O Amplification toward Efficient Key-Value Stores

Ting Yao*, Jiguang Wan*, Ping Huang†, Xubin He†, Qingxin Gui*, Fei Wu* and Changsheng Xie*
*Wuhan National Laboratory for Optoelectronics, School of Computer Science and Technology,
Huazhong University of Science and Technology, {tingyao, jgwan, wufei, cs_xie}@hust.edu.cn
†Department of Computer and Information Sciences, Temple University
{templestorager, xubin.he}@temple.edu

***Abstract*—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 light-weight compaction tree (LWC-tree), a variant of LSM-tree index optimized for minimizing the write amplification and maximizing the system throughput. The light-weight compaction drastically decreases write amplification by appending data in a table and only merging the metadata that has much smaller size. We implement three key-value LWC-stores on different storage mediums including Shingled Magnetic Recording (SMR) drives, Solid State Drives (SSD) and conventional HDDs, using our proposed LWC-tree. The LWC-store is particularly optimized for SMR drives as it eliminates the multiplicative I/O amplification from both LSM-trees and SMR devices. Due to light-weight compaction procedures, the LWC-store reduces the write amplification by up to 5× compared to the popular LevelDB key-value store. Moreover, the write throughput of the LWC-tree on SMR drives is significantly improved by up to 467% even compared with LevelDB on HDDs. Furthermore, the LWC-tree has wide applicability and it delivers impressive performance improvement in various conditions, including different storage mediums (i.e., SMR, HDD, SSD), varying value sizes and access patterns (i.e., uniform, Zipfian and latest key distribution).**

I. Introduction

Key-value stores are becoming widespread in modern data centers as they support an increasing diversity of applications [1]. However, currently the majority of key-value stores [2]–[5] are built based on the log-structured merge tree (LSM-tree) [6] which is used as the index structure for key-value stores. The LSM-tree enjoys its advantages over traditional B-trees because it adopts a memory buffer to batch new writes and creates write sequentiality. The LSM-tree does not suffer from random write problem. In the meanwhile, the batched data has to be persisted to the disk sooner or later. LSM-trees initiate compactions to absorb the temporarily buffered content and ensure the key-value pairs in sorted order for future fast lookups.

Compactions unfortunately introduce significant overheads due to I/O amplifications. To compact LSM-tree tables, the LSM-tree has to read a table file in Level L_i, which is called a *victim* table, and several table files in Level L_{i+1}, which have key ranges overlapping with that of the victim table and are called *overlapped* tables. It then merges those tables according to their key ranges and writes back the resultant tables. Such multiple table reads and writes can incur excessive disk I/Os (called I/O amplification) and thus severely affect the performance [1]. As we will see later (Section II-B), the I/O amplification caused by compactions in typical LSM-trees can reach to a factor of $50\times$ or higher [7], [8], causing up to $10\times$ degraded write performance.

Previously proposed solutions alleviate the write amplification and thus improve the overall performance by using large memory to buffer more indexes or KV items [9], leveraging the characteristic of specific devices [1], [7], [10], [11], hashing the KV items step by step [8], reducing the number of levels [12], or simply reducing the amplification factor in adjacent levels. Solving the write amplification problem of LSM-tree in this manner, unfortunately, is not cost-efficient as it requires additional memory or it is limited to specific storage devices. Even worse, the coupling of LSM-based key-value systems and storage devices may further amplify the compaction I/O overhead [1]. The I/O amplification of running the key-value store on SMR drives which are becoming increasingly deployed in data centers is particularly challenging due to the SMR device auxiliary amplification.

In this paper, we present a light-weight compaction tree, a variant of LSM-tree index which mitigates the write amplification during a compaction procedure by appending data in a table and only merging a little metadata, while retaining the level-by-level structure in LSM-tree for acceptable read performance. The LWC-tree introduces the following four new techniques to improve the write throughput without affecting the read performance:

- LWC-tree employs a simplified compaction procedure, named light-weight compaction. Instead of read, sort and rewrite all whole tables in a traditional compaction procedure, LWC-tree divides the 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 LWC-tree, the table is defined as a DTable.
- LWC-tree conducts metadata aggregation in adjacent

* Corresponding author :{jgwan,wufei}@hust.edu.cn

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 light-weigh compaction, the LWC-tree only requires to read a table sized data in a single DTable.

- LWC-tree reorganizes the DTable data structure according to the characteristics of light-weight compaction. DTables absorb the data from light-weight compactions by appending the tables whose table size is variable. DTable provides efficient lookups via binary searching segment indexes using the sequence number.
- LWC-tree balances the workload of DTables in the same level to ensure the balance of LWC-tree and provide an stable and efficient operation performance. Workload balance aims to move the overly-full table to its siblings by adjusting their key range after light-weight compaction, which brings no extra overhead for no data movement.

More importantly, based on the LWC-tree, we design and implement three key-value stores to explore its applicability in modern data centers. Specifically, we implement three LWC-tree based key-value stores on top of three kinds of devices commonly deployed in data centers, including SSDs, SMR drives, and conventional HDDs. For LWC-store on SMR drives, equal division is proposed to avoid a DTable overflowing a band in SMR drives, which reads out a DTable, divides it into several sub-DTables and writes them back to the same level. The experimental results demonstrate that the LWC-tree enjoys substantial performance improvements compared with the popular key-value store LevelDB and the advantages are not dependent on storage medium (SMR, HDD and SSD), value size and access pattern (uniform, Zipf and latest key distribution).

The rest of this paper is organized as follows: the background and motivation of our work are described in Section II. The design and implementation of the LWC-tree and the LWC-stores are presented in Sections III, and IV respectively. Section V presents the evaluation results and analyzes the effectiveness of light-weight compaction. Related work is described in Section VI, and our conclusions from this work are given in Section VII.

II. Background & 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 [13]. Therefore, key-value stores based on LSM-trees which 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 [8]: one goal is that new data has to be quickly admitted into the store to support high-throughput write, which is achieved by the data organization discussed in Section II-A and the other goal is that KV items in the store are sorted to support fast lookups, which is achieved by recurring compactions discussed in Section II-B.

A. LSM-trees and LevelDB

The LSM-tree is a widely used persistent data structure that provides efficient indexing for a key-value store with a high rate of inserts and deletes [6]. 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 [6], 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.

LevelDB, inspired by BigTable [2], [4] is a popular open-source key-value store from Google using the LSM-tree to organize the key-value pairs. We use LevelDB as an example to explain the data structure of LSM-trees. The LSM-tree uses an in-memory buffer, called MemTable, to absorb incoming KV items, which results in high-throughput write in LevelDB. The quickly admitted data is sorted according to their keys simultaneously by the skip-list in memory. Once a MemTable is filled up, it is turned into an immutable MemTable, which still remains in the memory but no longer accepts new data. Later on, the immutable Memtable in 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 items stored in the unit of data block. The KV items can be indexed by the metadata of SSTable and LevelDB can locate the block of a specific KV item via binary searching on the index. In the metadata of an SSTable, for each block there is a Bloom filter to indicate the presence of the KV items in it [8], 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-levels SSTables on disk compose a key-value store based on the LSM-tree. Since the LSM-tree is the core element of LevelDB and other widely used key-value stores, such as RocksDB [14] at Facebook, Cassandra [15], HBase [16] and PNUTS [17] at Yahoo!, the reductions of write amplification in our proposed LWC-tree can be beneficial to all KV stores based on LSM-tree and enjoy wide applicability.

B. 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-trees continuously compact key-value pairs at adjacent levels throughout the lifetime to sort key-value items. The compaction procedure of LSM-trees requires constant reading, sorting and writing

TABLE I
MEANINGS OF THE 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	R/W amplification from LSM-tree or LWC-tree
ARA/AWA	Auxiliary R/W amplification from SMR drives
MRA/MWA	Overall R/W amplification, MWA=MA $\times$ AWA

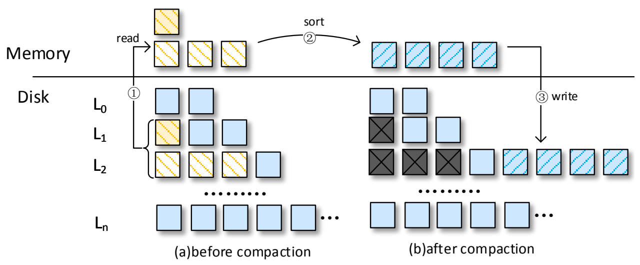


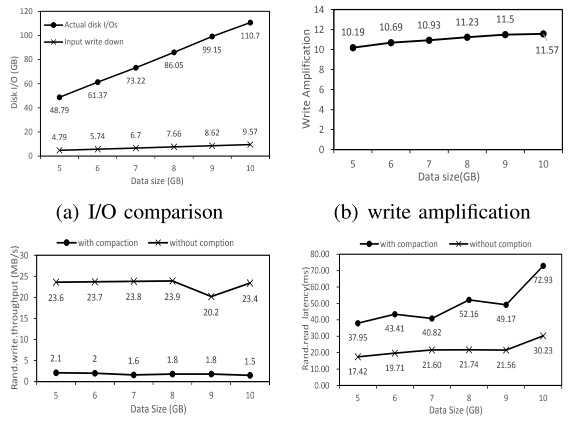
Fig. 1. **The compaction procedure of LSM-trees.** *The compaction involves read, sort and rewrite multiple SSTables, causing serious I/O amplification.*

KV items, which introduces excessive writes and represents a performance bottleneck of LSM-tree based key-value stores.

More specifically, a single compaction has three steps. For the convenience of exposition, let us call the SSTable selected to compact in level L_i as a victim SSTable and the SSTables whose key ranges fall in the key range of the victim SSTable in the next level L_{i+1} as overlapped SSTables. To start a compaction, LevelDB first selects victim SSTables and the overlapped SSTables according to the score of each level, and then it decides whether more victim SSTables in L_i can be added into the compaction by searching the SSTables whose key ranges fall in the ranges of overlapped SSTables. Figure 1 pictorially shows the three steps of a compaction procedure of LSM-trees. As it is shown, during the compaction procedure, LevelDB first reads the victim SSTable in level L_i and overlapped SSTables in level L_{i+1}. 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 amplification factor (AF). Due to this size relationship, 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. The $2\times$ multiplication indicates both read and write the total data. 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 [1].

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

To quantitatively measure the degree of amplification and performance degradation due to compaction in practice with LevelDB, we carry out the following experiments with the same configuration of LevelDB on HDDs in section V. First, we randomly load databases of size 5GB, 6GB, 7GB, 8GB, 9GB and 10GB, respectively. The value size is 4KB by default. Figure 2(a) shows the relationship between input data size and actual disk I/O data size. As can be seen, in all cases, LevelDB incurs significant write amplification. For example, writing 10GB 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 the write amplification factors and a minimum value of 10.19 is observed. Second, to evaluate the random I/O performance with compaction or without compaction, we use 6 different database sizes for the initial random loads as well 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 performing I/Os to eliminate concurrent compaction and mitigate the compaction interferences. Figure 2(c) and Figure2(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 $13.01\times$ and $2.23\times$ the throughputs with compaction, respectively. We design the LWC-tree mainly to eliminate the amplification caused by compactions in LSM-trees.



(a) I/O comparison (b) write amplification

(c) random write (d) random read

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

III. LWC-TREE DESIGN

As demonstrated in the previous section, the conventional LSM-tree incurs excessive I/O amplification when used as the key-value store index. We design the LWC-tree to alleviate the I/O amplification caused by compactions and aim to achieve high write throughput without sacrificing read performance. As a variant of LSM-tree, LWC-tree is also composed of one memory-resident component and multiple disk-resident components. Key-value items are written to the memory component first, then dumped to disk, and finally compacted to lower levels. We keep the sorted tables and the multi-level structure in LWC-tree to guarantee the read efficiency of KV stores.

The LWC-tree has four distinct features from the LSM-tree. First, LWC-tree employs a light-weight compaction mechanism to eliminate I/O amplification by appending data in tables and merging metadata (Section III-A). Second, each table preserves the aggregated metadata of overlapped tables in a lower layer to further reduce small random disk reads during the compactions (Section III-B). Third, the LWC-tree has a new data structure for SSTable, which is named as *DTable*, to improve the lookup performance in a table (Section III-C). Lastly, the LWC-tree creates workload balance for DTables to ensure operation efficiency and the balance of LWC-tree (Section III-D).

A. Light-weight Compaction

Compactions are needed during the entire life in LSM-trees to ensure acceptable read and scan performance. However, compactions bring about excessive I/O amplification as the LSM-tree table files have often to be read and written to disk during compactions. And also the extra I/Os contend for the disk resources, causing degraded performance, as demonstrated in Section II-B. To reduce the excessive I/O amplification and alleviate the degraded system throughput due to compaction, we propose a new approach to implementing light-weight compaction.

Motivated by the observation that only merging and sorting keys is sufficient while values can be managed separately [1], [18], our design of the LWC-tree moves a further step and proposes to merge and sort the metadata of table files to fasten compaction speed. Since the metadata size of each individual table file is much smaller than the table size itself, compacting only metadata can thus significantly reduce the amount of data involving in compactions, resulting in light-weight compaction.

To carry out a light-weight compaction, the LWC-tree first reads the victim DTable into memory and this DTable includes the aggregated metadata of overlapped DTables (Section III-B) as well as its own data and metadata. Then, the LWC-tree 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 metatdata of its associated overlapped DTables. Lastly, the resultant new segments and metadata are appended into the overlapped DTables by overwriting out-of-date metadata.

Figure 3 graphically shows the procedure of light-weight compaction in LSM-trees. As an example, let's 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 range of 'a', 'b', and 'c', respectively. Then instead of reading, merging, sorting all the four involved DTables as in the conventional compaction, the LWC-tree only reads the victim DTable and divides it into three segments corresponding to each of the three overlapped DTables' key ranges, namely, 'a', 'b' and 'c'. After reading the victim table into memory, the LWC-tree clears any invalid key-value pairs and sorts the valid key-value pairs. In the meanwhile, the newly generated metadata of each segment and the metadata of its corresponding overlapped DTables are merged. Finally, the segments are appended into the overlapped DTables in level L_2 together with the updated metadata, as indicated in the right part of the figure.

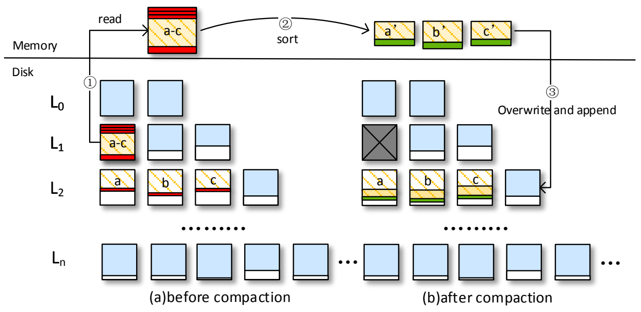


Fig. 3. **The light-weight compaction procedure of LSM-trees.** *The light-weight compaction involves reading one DTable, sorting the metadata and appending newly generated segments, which dramatically reduces the I/O amplification relative to the traditional compaction procedure in LSM-trees.*

In a light-weight compaction, it only needs to read one DTable from disk and appends AF (amplification factor) segments back to disk. Equation 2 calculates the total amount of I/O data size (S_{data}) involved in a light-weight compaction, where S_{dt} represents the size of a DTable and $S_{metadata}$ represents the metadata size. Comparing it with the overhead of the original compaction in LSM-trees given by Equation 1 in Section II-B, we can see that the I/O data size of light-weight compaction could be reduced by $10\times$, assuming the maximum size of a DTable equals to an SSTable and the AF is 10 by default according to LevelDB [4]. In addition to I/O amplification reduction, the light-weight compaction keeps the key ranges of the tables in the same level sorted. In other words, the key ranges of the tables in the same level of the LWC-tree are not overlapped, which helps to provide high table lookup efficiency.

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

B. Metadata Aggregation

As discussed in the preceding section, the LWC-tree employs an efficient compaction policy by considering only metadata during compactions, which could result in up to $10\times$ I/O data reduction. However, so far we have not yet discussed how to efficiently obtain the metadata of overlapped DTables during compactions, which can critically impact compaction speed. An intuitive method would be to read the metadata from the overlapped DTables in the next tree level whenever the metadata is needed. Unfortunately, this method incurs extra cost due to randomly reading the metadata, offsetting the efficiency of our light-weight compactions. Another straightforward solution is to cache all 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

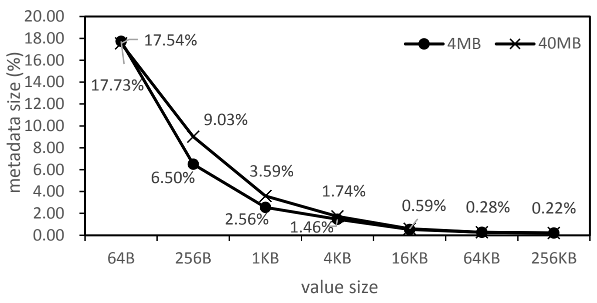


Fig. 4. **The proportion of metadata in the table.** *This figure shows the ratio between metadata and data in a table under different value sizes with two table size being 4MB and 40MB respectively. Small value sizes have significantly higher metadata overhead.*

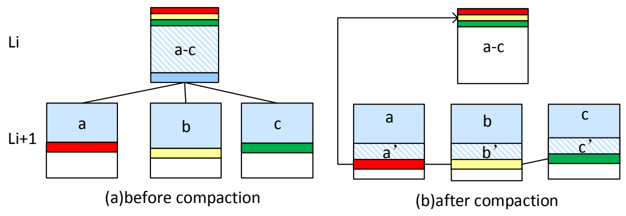


Fig. 5. **The metadata aggregation between adjacent levels.** *This figure shows that before compaction, the metadata of overlapped DTables is collected in the victim DTable and after compaction, the newly produced metadata is aggregated in the empty victim DTable.*

between metadata and data as the value size varies in two scenarios where the table size is 4MB and 40MB, respectively. As it 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 [19]. For instance, the metadata overhead is around 17% and up to 9% for the value size of 64B and 256B, respectively. Equally put, to support a 10T 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 issue, we propose metadata aggregation to cluster the metadata of overlapped DTables into the corresponding victim table, as it is illustrated in Figure 5. After finishing the light-weight compaction, the updated metadata of overlapped DTables in level L_{i+1} which has been merged recently in memory is aggregated and stored in the victim DTable in level L_i so as to avoid the accesses during the next compaction. The victim DTable always contains the most recent metadata of overlapped DTables so that consistency can be guaranteed. Though the metadata aggregation introduces an extra write in a compaction, it however reduces AF times random reads on average during the next compaction. Overall, for a light-weight compaction, it 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.

C. Data structure of DTable

Similar to the SSTable in LSM-trees, the DTable is our proposed basic data management unit in LWC-trees and its

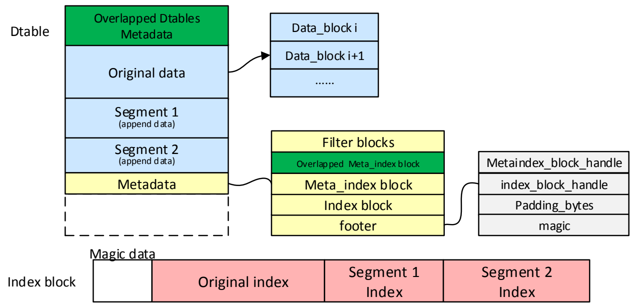


Fig. 6. **The data structure of DTable** *This figure shows the organization in a DTable, which is comprised of the metadata from overlapped DTables, data and metadata.*

detailed data structure is depicted Figure 6 . A DTable contains the aggregated metadata from overlapped DTables, data and metadata. The data of DTable includes several segments, and each segment is composed of the appended data blocks of a light-weight compaction. A light-weight compaction appends the sorted data segment to the overlapped DTables without modifying the data written before. As a result, 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 damage the lookup performance within a table. To facilitate the read and search performance within a DTable, our LWC-tree employs a new method for managing the metadata.

The metadata stored in a DTable includes the bloom filter blocks, overlapped Meta_index blocks for overlapped DTables' metadata, Meta_index blocks for metadata, index blocks for data blocks, and a footer. The index of data block is used to identify a 4KB data block. Each block has a bloom filter to indicate the existence of KV items in this block. When conducting a light-weight compaction, the metadata of new generated segment and the metadata of overlapped DTables are merged in memory. Concretely, first, LWC-tree generate new bloom filters for blocks in segment, and the new bloom filters just append to the filter blocks of the corresponding overlapped DTable; second, the index blocks for blocks in segment are established and write together with the index blocks of the overlapped DTable; third, the index for metadata and overlapped DTables metadata are modified according their new location; fourth, the footer is updated as well.

Segment indexes: The index block is also organized in segments, similar to the data organization. The LWC-tree assigns a sequence number to each segment, with the larger sequence number indicating fresher recency. The indexes in each segment are sorted as well. To find a KV item in a DTable, the LWC-tree starts with checking the largest sequence number segment. It looks for the data block to which the wanted key might belongs. If such a block is found, it then checks the block bloom filter to confirm the existence of the key. If the key exists, then the data block is returned. Other it repeats the same process in decreasing sequence number segments. The LWC-tree ensures that only one data block read is needed to find a KV item.

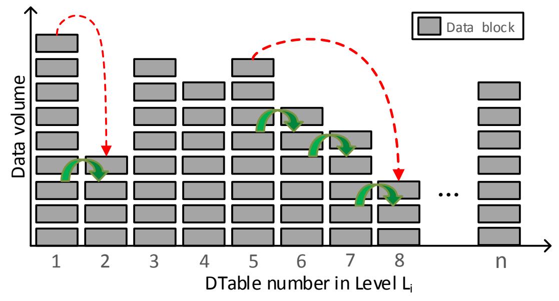


Fig. 7. **The imbalanced workload of DTables in Level L_i.** *This figure shows the unbalanced data distribution of DTables in a level. The LWC-tree chooses to migrate data from Table 1 to Table 2 and from Table 5 to Table 8 to keep balance.*

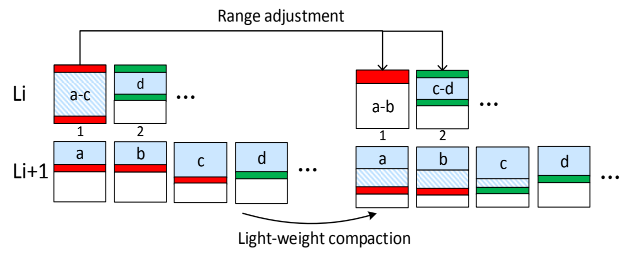


Fig. 8. **The workload balance during a compaction.** *This figure shows the procedure of performing workload balance between two adjacent tables in Level L_i, e.g., Table 1 and Table 2. Workload balance is conducted after light-weight compaction, which modifies the key range of imbalanced tables without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

D. Workload Balance in DTables

It is possible that different DTables of the same level of LWC-tree can have different amount of data due to their represented key ranges. Figure 7 shows the workload distribution across the DTables in the same level at a specific timepoint during the running. The process of light-weight compaction appends data to each DTable according to their key ranges, causing the data volume in DTables being dynamic and imbalanced, which may affect the read performance of LWC-tree. To avoid that, the LWC-tree attempts to construct a balanced tree in which each DTable is filled with the data of the same size. As an example in Figure 7, we may expect to move the data of overly-full table to its siblings, such as migrate the data from Table 1 to Table 5 and from Table 2 to Table 8 to achieve balance.

To this end, the LWC-tree introduces a method to realize workload balance by adjusting the key range of the overly-full table and its siblings in the same level during the light-weight compaction without extra overhead. Concretely workload balance separates the key range of overly-full table into two part, and gives a part of key ranges to its next neighbor. So the incoming workload is replaced according to the new key range of DTables and resulted in a quite balanced workload placing. This works because the LWC-tree (same as the LSM-tree) doesn't have strict subset relationships across levels.

Figure 8 shows an example of ensuring workload balance in Table 1&2. After DTable 1 with excessive data finishing compaction from L_i to L_{i+1}, the key range of Table 1 is separated into 2 parts according to workloads key range distribution. For example, the first part is range 'a-b' and the second part is range 'c' in Figure 8. Noticing that DTable 2 contains the minimal data, the LWC-tree adds the second part key range 'c' into DTable 2, resulting in the key ranges of Table 1 and Table 2 becoming 'a-b' and 'c-d', respectively. Due to the adjustment of the key ranges of DTables, the incoming workload of DTables will be changed and the workload balance is achieved as a result. Similar to the key range adjustment between Table 1 and Table 2, the LWC-tree consecutively conducts the key range adjustments from Table 5 until Table 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 Table 5 to Table 8 directly, as the keys must be sorted for tables in a level. The key range redistribution process during compaction imposes almost no extra overhead, as it incurs no data migration.

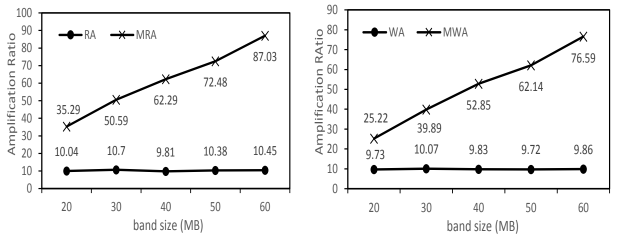
IV. LWC-STORE DESIGN AND IMPLEMENTATION

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 [20]–[23]. 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 LWC-stores on SMR drives.

A. Auxiliary write amplification of SMR drives

Shingled magnetic recording is leading the next generation 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 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 requires a reorganization via performing garbage collection [24]. However, no matter using the in-place or out-of-place update, SMR drives still generate read/write amplification, named Auxiliary Write or Read Amplification (short for AWA or ARA). Even worse, the AWA and the WA induced by applications create a multiplicative effect on write traffic to SMR drives, as it is with flash devices [25].

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 [7]. To reveal the amplification



(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.*

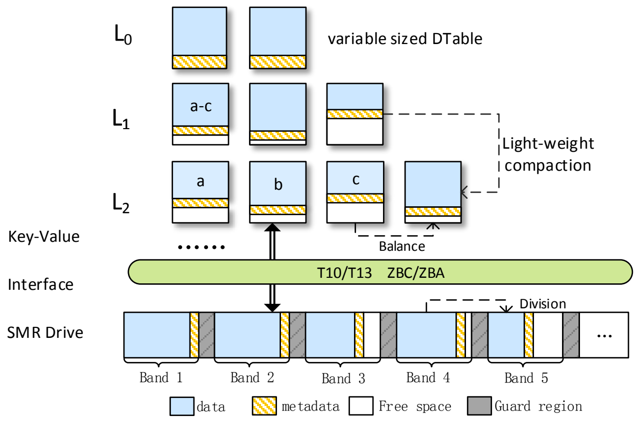


Fig. 10. **The system architecture of the LWC-store on SMR drives.** *In the LWC-store on SMR, the light-weight compaction and workload balance support the efficiency of LWC-tree. The data layout and the division procedure of varying sized DTables support the SMR optimized KV store.*

on SMR drives, we experimentally measure the amount of amplification by running LevelDB on an SMR emulator which is artificially banded on top of a conventional HDD [26]. The multiplicative WA and RA (in short MRA, MWA) is shown in Figure 9. As an example, in a 40MB band size SMR drive, the WA and RA from LSM-tree multiplied by ARA and AWA increase 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 [27] due to random accesses and I/O amplification. Ext4 as a mature and widely used file system, is not sequential-oriented [28]. Given the multiplicative effect of I/O amplification, it is thus important to minimize additional KV store writes (as what we discussed in Section III), and reduce the I/O amplification on SMR drives (Section IV-B).

B. The LWC-store on SMR Drives

As seen in the previous section, building KV stores on SMR drives with 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 by eliminating I/O amplification, the LWC-store assigns each DTable to a sequential physical 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 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 a 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 light-weight compaction. At the device level, the SMR device provides dedicated bands 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 light-weight 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 [22]. In addition, as the metadata is 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-stores not only mitigate the write amplification from compactions but also eliminate the auxiliary I/O amplification from SMR drives.

C. Equal Division

The implementation of LWC-stores on SMR drives places each DTable in an SMR band, therefore the maximum size of a DTable is limited by the SMR band. LWC-stores expect that each DTable is perfectly filling the band at the time of performing compactions, so that the data would not overflow a band and the KV store would not induce extra space overhead. However, for some workloads with the latest key distribution, the data appended to some DTables during light-weight compaction may overflow the bands. Solving this problem by continuously compacting oversized DTables and flushing to lower layers could bring cascading compactions and result in more oversized DTables.

To address the above problem, equal division of LWC-store is proposed, which divides an oversized DTable of into several sub-DTables and set the sub-DTables at 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 that 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 in L_i. As a result, the

TABLE II
BASIC DISK PERFORMANCE MEASUREMENTS.

	SSD	HDD	SMR
Sequence read (MB/s)	1200	169	165
Sequence write (MB/s)	901	155	148
Random read 4KB (IOPS)	8647	64	70
Random write 4KB (IOPS)	19712	143	5-140

key range is separated into n parts accordingly. One benefit of our equal division is to keeping data sorted within a table.

However, the biggest 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 will be. 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 to be 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.

V. EVALUATION

In this section, we conduct extensive experiments to evaluate the LWC-stores. The experiments include main evaluations and sensitivity evaluations. 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.

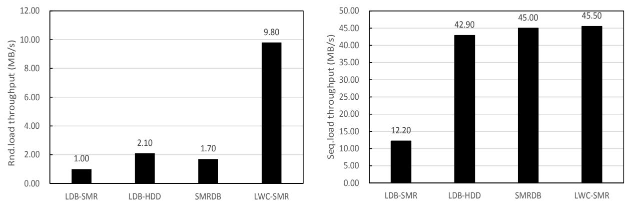
The test machine used for our experimentation has 16 Intel(R) Xeon(R) CPU E5-2660 0 @ 2.20GHz processors and 64GB of memory. The operating system is 64-bit Linux 4.4.0-31-generic, and the file system to support conventional LevelDB is Ext4. The storage devices used for the test are a 1TB Seagate ST1000DM003 HDD, a 5 TB Seagate ST5000AS0011 SMR drive and a 400GB Intel P3700 SSD. Table II 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 comparing to the conventional HDD, while the random write restriction of an SMR drive is obvious, which is consistent with the findings in existing works [26]. In addition, the Intel P3700 SSD exhibits much better performance than the HDD and SMR drive.

A. Main Evaluation

For the first set of experiments, we intend to use the LWC-store on SMR drives to demonstrate the superiority of the design strategies of LWC-store. Main 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.

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 IV-A.



(a) Random load throughput (b) Sequence load throughput

Fig. 11. **Load Performnce** *This figure shows random and sequential load performance of 4 different key-value stores. Our proposed LWC-smr on an SMR outperforms LDB-smr dramatically in both random and sequence load.*

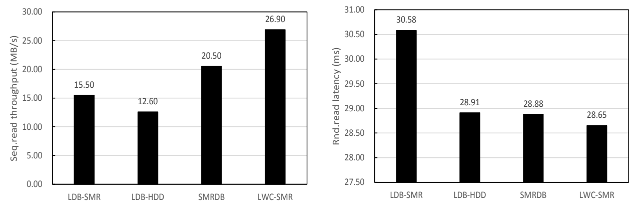
SMRDB: SMRDB is an SMR drive optimized key-value store, which proposes an SMR friendly solution to improve the performance of KV stores on SMR drives [12]. The main design choices of SMRDB are to reduce the LSM-tree levels to only two levels (i.e., L_0 and L_1), allow the key ranges overlapped in the 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 light-weight compaction in LWC-tree and tables with size variation in SMR drive. LWC-smr is implemented based on LevelDB 1.19 [4] and the SMR emulator [26] as discussed in Section IV-B. The default band size is 40MB, and the DTable size is variable but is 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. We use the key size of 16 bytes and the value size of 4 KB.

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 items 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 theses two workloads is that inserting entries in sequence does not incur compactions, while random loading does. Random loading performance is significantly influenced by the compaction efficiency.

Figure 11(a) gives the comparisons of the random load performance of the four key-value systems. From this figure, we can make three observations. First, LWC-smr improves the random load performance to 9.8 times that of LDB-smr. One reason for that is because LevelDB conducts costly compaction procedures constantly, which pull the random load performance down, as aforementioned in Section II-B. Another reason is because LWC-smr is built directly on the SMR drive and is free from file system overheads, while LDB-smr suffers from excessive Ext4 overhead with SMR drives [27], [28], as aforementioned in Section IV-A. Second, LWC-smr outperforms LDB-hdd by 4.67 times, which verifies that by adopting LWC-smr, SMR devices can deliver an even better random load performance than conventional HDDs. Moreover, LDB-hdd also suffers from the amplification from the LSM-tree



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

Fig. 12. **Read Performnce** *This figure shows random and sequential read performance of 4 different key-value sotres. 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 millisecond.*

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, which increase 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 [8]. In general, our LWC-smr achieves significant advantages because of the light-weight compaction in the LWC-tree and the absence of file system overheads between DTables in the LSM-tree and bands in SMR devices. Moreover, the property of ensured sequential writes of the LWC-smr contributes to its leading position in the 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 on which key-value system, the corresponding sequential load performance is much better than the random load performance, which can be mainly attributed to the compactions caused by random loading. Second, LWC-smr and SMRDB both deliver comparable performance as LDB-hdd, which is due to the absence of compactions. Third, LDB-smr is about 3.7$\times$ inferior to the other three systems. This is because of the embedded indirection of drive managed SMR drives and the non-sequential Ext4 file system [27], [28].

2) Read Performance: This experiment is to evaluate the read performance including sequential read and uniformly distributed random read by looking up 100K entries against 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$\times$ 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 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.

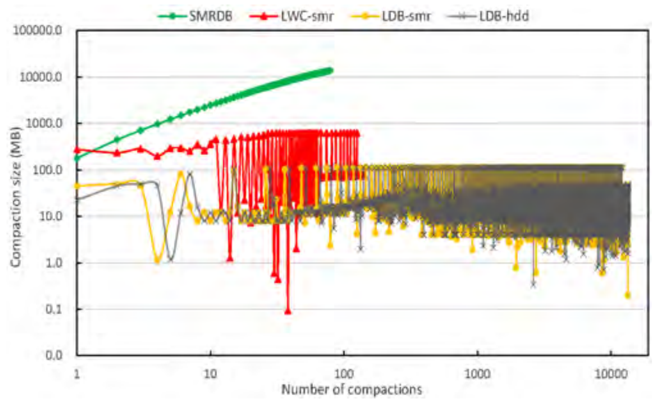


Fig. 13. **The microscopic view of compactions.** *This figure shows the compactions resulting from randomly loading a 40GB database to the four KV systems. The x-axis denotes the the number of compactions and the y-axis denotes the data size in each compaction.*

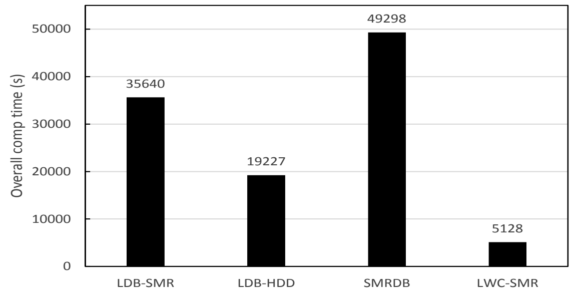
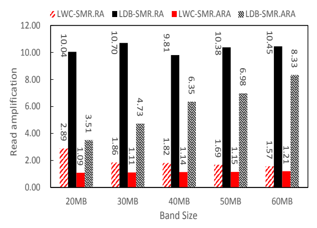


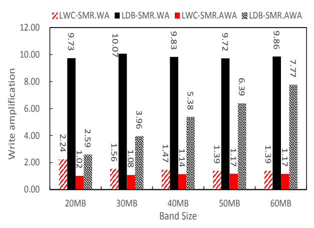
Fig. 14. **The overall compaction time.** *This figure shows the overall consumed time on compactions for randomly loading a 40GB database to four KV systems. The y-axis unit is the time in seconds.*

3) Performance of Light-weight Compaction: The compactions in key-value systems consume a large percentage of device bandwidth [8], [9], which influences the overall performance by blocking foreground requests. Therefore, how efficiently compactions can be conducted seriously affects the performance. 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.

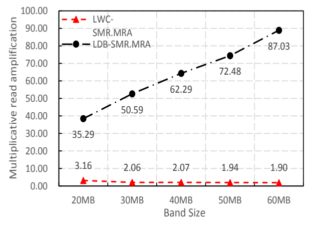
Figure 13 depicts the number of compactions and the corresponding written data size. From this figure, we obtain a primary observation that LDB-smr and LDB-hdd have much more compactions than SMRDB and LWC-smr, while SMRDB has larger compaction size relative to other counterparts. That is to say, although SMRDB reduces the compactions of LSM-tree, it however incurs larger compaction size unfortunately. The reason is because SMRDB allows 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$\times$ less than the size of SMRDB) and fewer compaction numbers, resulting in LWC-smr delivering a better compaction efficiency.



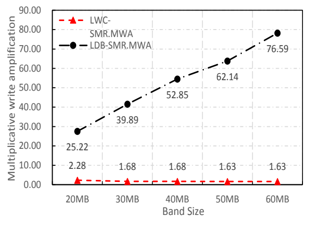
(a) RA and ARA



(b) WA and AWA



(c) MRA



(d) MWA

Fig. 15. **I/O amplification** *Figure (a) and (b) shows the read or write amplification from KV stores (RA/WA) and the read or write amplification from SMR drives (ARA/AWA). Figure (c) and (d) shows 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.*

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 corresponding total compaction time for randomly loading a 40GB database of four KV systems. From this figure, we can draw a conclusion that the light-weight 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.

4) I/O Amplification: The I/O amplification of a key-value system on SMR drives is generated by compaction and multiplied by the auxiliary amplification from SMR drives, as discussed in Section IV-A. In this section, 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.

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, the LWC-tree outperforms the LSM-tree by reducing the read and write amplification up to 5.49× and 6.32× respectively on average. This is because the light-weight compaction in LWC-tree reduces the total compaction size for loading a database. 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. In addition, we make an observation that the auxiliary amplification of LDB-smr increases with the increase of 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. Figure 15(c) and Figure 15(d) depict the multiplicative amplifications with

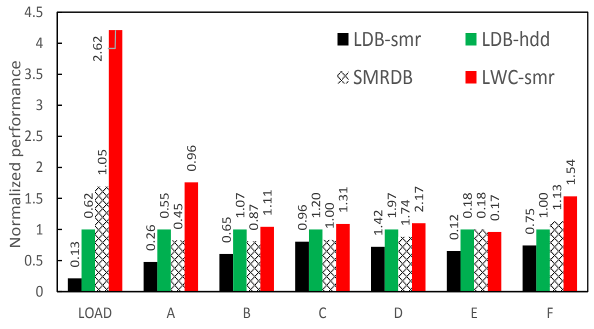


Fig. 16. **YCSB Macro-benchmark Performance.** *This figure shows the performance normalized to LDB-hdd. The x-axis represents different workloads, the y-axis denotes the normalized values, and the number on each bar shows the actual throughput (K ops/s). Workload A is composed with 50% reads and 50% 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.*

different band sizes. MWA is the overall write amplification caused by WA and AWA, so as MRA. This figure shows that LWC-smr mitigates the overall amplification by dozens of times, which validates the effectiveness of the design strategies of LWC-smr.

5) Macro-Benchmark: The YCSB benchmark [29] provides a framework and a set of seven workloads applicable for evaluating the performance of key-value stores. We perform YCSB benchmarking as a supplement to verify the performance advantages of the LWC-smr. We load 100GB database first, and then test the performance in different workloads with 100k entries. Figure 16 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 LWC-smr obtains more advantages for random load than read, which accords with the experiments in preceding sections.

B. Sensitivity Study

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

1) 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 17 shows the random write throughput and the corresponding space usage efficiency of LWC-smr. The space usage efficiency is defined as the ratio between the required storage space of the key-value store and the actually 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 with quite bad random write performance for as it incurs less frequent divisions. On the contrary, a large division number hurts the space usage efficiency badly but delivers a better random write performance. In conclusion, 12

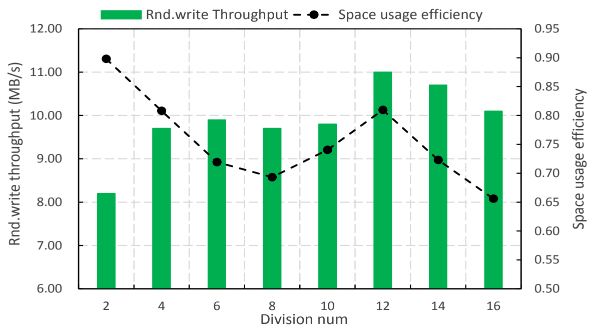


Fig. 17. **Random write throughput and space usage efficiency of different division number.** *This figure shows the influence of division number on the performance and the space usage efficiency of LWC-smr.*

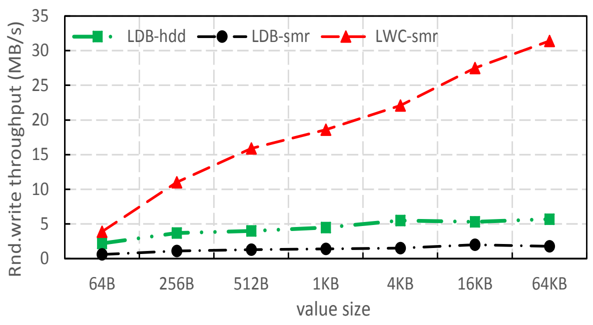


Fig. 18. **Random write throughput in varying value sizes.** *This figure shows that the LWC-tree exhibits advantages in all the examined value sizes and the bigger the value size is, the more prominent the advantage is.*

is a relative good choice for the division number of LWC-smr, as it achieves a good trade-off point between space usage efficiency and random write performance.

2) 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 18 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 increase of value size. Moreover, 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. The phenomenon results from the fact that the LWC-tree operates on metadata while the LSM-tree involves data itself and therefore the LWC-tree merges less metadata with increased value size.

3) Storage Medium (HDD,SSD): To explore the applicability of the LWC-tree in different storage mediums, besides of the experiments of LWC-stores on an SMR device we have done so far, we then evaluate LWC-stores on HDDs and SSDs. We report the performance comparisons in the following two configurations: (1) LevelDB on conventional HDD (LDB-hdd), where the original LevelDB is run on a conventional HDD; and (2) LWC-store on conventional HDD (LWC-hdd), where the LWC-store is run on top of a conventional HDD without a file system in between. Its performance difference relative to LDB-hdd reflects the applicability of our design

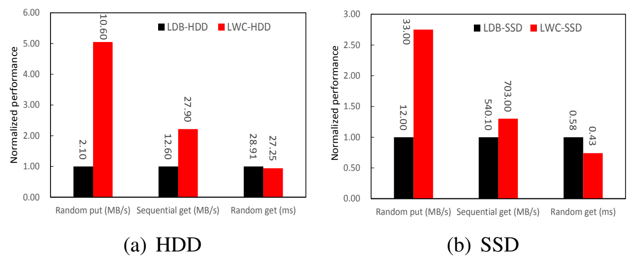


Fig. 19. **Performance comparison in HDD and SSD.** *The x-axis denotes the different performance measurements including: random write, sequential read, and random read. The y-axis denotes the normalized performance relative to LDB-HDD. The number on top of each bar is the actual performance value.*

ideas to HDDs. Figure 19(a) compares the LevelDB and LWC-store on an HDD in terms of the performance of random write, random read, and sequence read. From this figure, we can make three observations. First, LWC-hdd improves the random load performance to 5.05 times that of LDB-hdd. This is attributed to the efficient light-weight compaction which mitigates the I/O amplification in the LWC-tree significantly. Second, LWC-hdd improves the sequential read performance to 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 19(b) gives a similar comparison but on the SSD storage medium. On the high level, we can make similar observations as it is with the HDD medium in Figure 19(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 the policy of LWC-store that 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.

VI. Related Work

Key-value stores have recently become a hot research interest as many data center applications rely on key-value stores. Various key-value stores have been proposed for specific storage medium. Wisckey [1] is a flash optimized key-value store. Its main idea is to separate keys from values to reduce I/O amplification by mitigating the migration of values. NVMKV [7] is an FTL-aware light weight KV store which leverages native FTL capabilities to provide a high performance. SkimpyStash [10] is RAM space skimpy key-value store on flash-based storage, which moves a part of the table to the SSD using a linear chaining. FlashStore [30] is a high throughput persistent key-value store using cuckoo hashing. LOCS [31] is an LSM-tree based KV store on SSD, which exposes its internal flash channels to applications to better work with the LSM-tree based KV store. SILT [11] combines the log-structure, hash-table, and sorted-table layouts to provide a memory-efficient KV store. Memcached [3] and Redis [32] are the popular memory KV implementations. GD-Wheel [33] provides a cost-aware replacement policy for memory based KV stores, which takes access recency and computation cost into account. It is granted that a full understanding of storage devices (such as SSD [34]–[38] and SMR [26], [28], [39]) contribute to a better design of device specific key-value stores. We optimize the key-value stores for SMR drives which are being considered to be widely deployed due to its capacity advantages.

Other researches are dedicated to develop key-value stores for specific scenarios. zExpander [40] dynamically partitions the cache into two parts for high memory efficiency and low miss ratio, respectively, by compressing one of the partitions. ForestDB [41] addresses the performance degradation of large keys by employing a new hybrid index scheme. LSM-trie [8] constructs a prefix tree to store data in a hierarchical structure which helps to reduce the metadata and the write amplification. bLSM [13] proposes a "spring and gear" merge scheduler, which bounds write latency and provides high read and scan performance. Atlas [5] is a key-value storage system for cloud data, which 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.

VII. Conclusion

The constantly occurring compactions in the LSM-tree based key-value stores seriously damage the overall performance. Running KV stores on different storage devices can lead to different degrees of influence, especially on SMR drives with severe multiplicative amplification. In this paper, we propose LWC-tree with light-weight compaction to reduce the amplification and accelerate the compaction procedure by operating the metadata during compactions. we design and implement three high performance KV store systems for SMR drives, HDD and SSD, respectively. Extensive experiments have demonstrated that the LWC-store significantly improves random write throughput compared with other schemes, including SMRDB and LevelDB, by up to 5 times. Furthermore, we have also experimentally demonstrated that the performance improvement remains across different storage mediums, value sizes and access patterns (i.e., uniform, Zipf and latest key distribution).

Acknowledgement

We thank the anonymous reviewers for their constructive comments. This work was sponsored in part 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. This research is partially sponsored by the U.S. National Science Foundation (NSF) under Grant Nos.CNS-1702474 and CNS-1700719. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the funding agencies.

References

[1] L. Lu, T. S. Pillai, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Wisckey: separating keys from values in ssd-conscious storage," in *14th USENIX Conference on File and Storage Technologies (FAST 16)*, 2016, pp. 133–148.

[2] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. Gruber, "Bigtable: A distributed storage system for structured data," in *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI 06)*, 2006, pp. 205–218.

[3] B. Fitzpatrick and A. Vorobey, "Memcached: a distributed memory object caching system," 2011.

[4] S. Ghemawat and J. Dean, "Leveldb," https://github.com/Level/leveldown/issues/298, 2016.

[5] C. Lai, S. Jiang, L. Yang, S. Lin, G. Sun, Z. Hou, C. Cui, and J. Cong, "Atlas: Baidu's key-value storage system for cloud data," in *2015 31st Symposium on Mass Storage Systems and Technologies (MSST)*, 2015, pp. 1–14.

[6] P. ONeil, E. Cheng, D. Gawlick, and E. ONeil, "The log-structured merge-tree (lsm-tree)," *Acta Informatica*, vol. 33, no. 4, pp. 351–385, 1996.

[7] L. Marmol, S. Sundararaman, N. Talagala, R. Rangaswami, S. Devendrappa, B. Ramsundar, and S. Ganesan, "Nvmkv: A scalable and lightweight flash aware key-value store," in *6th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 14)*, 2014.

[8] X. Wu, Y. Xu, Z. Shao, and S. Jiang, "Lsm-trie: An lsm-tree-based ultra-large key-value store for small data," in *Proceedings of the USENIX Annual Technical Conference (USENIX 15)*, 2015.

[9] Y. Yue, B. He, Y. Li, and W. Wang, "Building an efficient put-intensive key-value store with skip-tree," *IEEE Transactions on Parallel and Distributed Systems*, vol. PP, no. 99, pp. 1–1, 2016.

[10] B. Debnath, S. Sengupta, and J. Li, "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, 2011, pp. 25–36.

[11] H. Lim, B. Fan, D. G. Andersen, and M. Kaminsky, "Silt: A memory-efficient, high-performance key-value store," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, 2011, pp. 1–13.

[12] R. Pichumani, J. Hughes, and E. L.Miller, "Smrdb: Key-value data store for shingled magnetic recording disks," in *Proceedings of SYSTOR 2015*, 2015.

[13] R. Sears and R. Ramakrishnan, "blsm: A general purpose log structured merge tree," in *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data (SIGMOD 12)*, 2012.

[14] Facebook, "Rocksdb, a persistent key-value store for fast storage enviroments." http://rocksdb.org/, 2016.

[15] A. Lakshman and P. Malik, "Cassandra: A decentralized structured storage system," in *The 3rd ACM SIGOPS International Workshop on Large Scale Distributed Systems and Middleware*, 2009.

[16] Apache, "Hbase," http://hbase.apache.org/, 2007.

[17] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni, "Pnuts: Yahoo!s hosted data serving platform," in *Proceedings of the VLDB Endowment (PVLDB 2008)*, 2008.

[18] C. Nyberg, T. Barclay, Z. Cvetanovic, J. Gray, and D. Lomet., "Alphasort: A risc machine sort," in *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data (SIGMOD 94)*, 1994.

[19] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny, "Workload analysis of a large-scale key-value store," *Acm Sigmetrics Performance Evaluation Review*, vol. 40, no. 1, pp. 53–64, 2012.

[20] A. Amer, D. D. E. Long, E. L. Miller, J.-F. Paris, and S. J. T. Schwarz, "Design issues for a shingled write disk system," in *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST10)*, 2010.

[21] G. Gibson and G. Ganger, "Principles of operation for shingled disk devices," *Carnegie Mellon University Parallel Data Lab Technical Report CMU-PDL-11-107*, 2011.

[22] T. Feldman and G. Gibson, "Shingled magnetic recording: Areal density increase requires new data management," *USENIX*, vol. 38, no. 3, pp. 22–30, 2013.

[23] I. Tagawa and M. Williams, "High density data-storage using shingled write," in *IEEE International Magnetics Conference (INTERMAG)*, 2009.

[24] Y. Cassuto, M. A. Sanvido, C. Guyot, D. R. Hall, and Z. Z. Bandic, "Indirection systems for shingled-recording disk drives," in *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on*. IEEE, 2010, pp. 1–14.

[25] J. Yang, N. Plasson, G. Gillis, and N. Talagala, "Hec: improving endurance of high performance flash-based cache devices," in *Proceedings of the 6th International Systems and Storage Conference*. ACM, 2013, p. 10.

[26] A. Aghayev and P. Desnoyers, "Skylight—a window on shingled disk operation," in *13th USENIX Conference on File and Storage Technologies (FAST 15)*, 2015, pp. 135–149.

[27] A. Manzanares, N. Watkins, C. Guyot, D. LeMoal, C. Maltzahn, and Z. Bandic, "Zea, a data management approach for smr," in *8th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 16)*, 2016.

[28] A. A. T. Ts'o, G. Gibson, and P. Desnoyers, "Evolving ext4 for shingled disks," in *(to appear)15th USENIX Conference on File and Storage Technologies (FAST 17)*, 2017.

[29] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with ycsb," in *Proceedings of the ACM Symposium on Cloud Computing (SOCC'10)*, 2010.

[30] B. Debnath, S. Sengupta, and J. Li, "Flashstore: high throughput persistent key-value store," *Proceedings of the VLDB Endowment*, vol. 3, no. 1-2, pp. 1414–1425, 2010.

[31] P. Wang, G. Sun, S. Jiang, J. Ouyang, S. Lin, C. Zhang, and J. Cong, "An efficient design and implementation of lsm-tree based key-value store on open-channel ssd," in *Proceedings of the Ninth European Conference on Computer Systems*, 2014, pp. 16:1–16:14.

[32] S. Sanfilippo and P. Noordhuis, "Redis," http://redis.io/, 2009.

[33] C. Li and A. L. Cox, "Gd-wheel: a cost-aware replacement policy for key-value stores," in *Proceedings of the Tenth European Conference on Computer Systems*, 2015, p. 5.

[34] P. Huang, P. Subedi, X. He, S. He, and K. Zhou, "Flexecc: Partially relaxing ecc of mlc ssd for better cache performance." in *USENIX Annual Technical Conference*, 2014, pp. 489–500.

[35] Y. Zhou, F. Wu, P. Huang, X. He, C. Xie, and J. Zhou, "An efficient page-level ftl to optimize address translation in flash memory," in *Proceedings of the Tenth European Conference on Computer Systems*. ACM, 2015, p. 12.

[36] P. Huang, G. Wu, X. He, and W. Xiao, "An aggressive worn-out flash block management scheme to alleviate ssd performance degradation," in *Proceedings of the Ninth European Conference on Computer Systems*. ACM, 2014, p. 22.

[37] H. Wang, P. Huang, S. He, K. Zhou, C. Li, and X. He, "A novel i/o scheduler for ssd with improved performance and lifetime," in *Mass Storage Systems and Technologies (MSST), 2013 IEEE 29th Symposium on*. IEEE, 2013, pp. 1–5.

[38] K. Zhou, S. Hu, P. Huang, and Y. Zhao, "Lx-ssd: Enhancing the lifespan of nand flash-based memory via recycling invalid pages," in *33rd International Conferenceon Massive Storage Systems and Technology (MSST 2017)*, 2017.

[39] W. He and D. H. Du, "Smart: An approach to shingled magnetic recording translation," in *Proceedings of FAST17: 15th USENIX Conference on File and Storage Technologies*, 2017, p. 121.

[40] X. Wu, L. Zhang, Y. Wang, Y. Ren, M. Hack, and S. Jiang, "zexpander: a key-value cache with both high performance and fewer misses," in *Proceedings of the Eleventh European Conference on Computer Systems*. ACM, 2016, p. 14.

[41] J.-S. Ahn, C. Seo, R. Mayuram, R. Yaseen, J.-S. Kim, and S. Maeng, "Forestdb: A fast key-value storage system for variable-length string keys," *IEEE Transactions on Computers*, vol. 65, no. 3, pp. 902–915, 2016.