A Set-aware Key-Value Store on Shingled Magnetic Recording Drives with Dynamic Band

Ting Yao[1 3], Zhihu Tan[2*], Jiguang Wan[1*], Ping Huang[3], Yiwen Zhang[1], Changsheng Xie[1], and Xubin He[3]
[1] Wuhan National Laboratory for Optoelectronics, HUST, China
[2] School of Computer Science and Technology, HUST, China
Email: {tingyao, stan, jgwan, cs_xie}@hust.edu.cn
[3] Department of Computer and Information Sciences, Temple University, USA
Email: {ting.yao, templestorager, xubin.he}@temple.edu

***Abstract*—Key-value (KV) stores play an increasingly critical role in supporting diverse large-scale applications in modern data centers hosting terabytes of KV items which even might reside on a single server due to virtualization purpose. The combination of ever growing volume of KV items and storage/application consolidation is driving a trend of high storage density for KV stores. Shingled Magnetic Recording (SMR) represents a promising technology for increasing disk capacity, but it comes at a cost of poor random write performance and severe I/O amplification. Applications/software working with SMR devices need to be designed and optimized in an SMR-friendly manner.**

In this work, we present SEALDB, a Log-Structured Merge tree (LSM-tree) based key-value store that is specifically optimized for and works well with SMR drives via adequately addressing the poor random writes and severe I/O amplification issues. First, for LSM-trees, SEALDB concatenates SSTables of each compaction, and groups them into *sets*. Taking sets as the basic unit for compactions, SEALDB improves compaction efficiency by mitigating random I/Os. Second, SEALDB creates varying size bands on HM-SMR drives, named *dynamic bands*. Dynamic bands not only accommodate the storage of sets, but also eliminate the auxiliary write amplification from SMR drives.

We demonstrate the advantages of SEALDB via extensive experiments in various workloads. Overall, SEALDB delivers impressive performance improvement. Compared with LevelDB, SEALDB is $3.42\times$ faster on random load due to improved compaction efficiency and eliminated auxiliary write amplification on SMR drives.

***Keywords*-LSM-tree; SMR; KV store; Set; Dynamic band**

I. Introduction

Nowadays, huge volume of data is being generated at an accelerating rate, of which key-value systems are gaining increasing popularity and have become an important component in modern data center infrastructure [1], [2]. The ever-growing storage requirement is continually pushing the storage capacity envelope of storage systems, calling for efficient solutions to expanding storage capacity. Adding more disks and/or servers to expand storage space should be pursued as a last resort, because it not only incurs added monetary investment, but also counteracts virtualization platforms from consolidating applications on fewer servers or less storage space. Facing this dilemma, innovators from both academia and industry have been relentlessly researching for more effective solutions. Among the proposed disk technologies, Shingled Magnetic Recording (SMR) [3] technology provides a feasible solution to enlarging the disk capacity, as it significantly increases storage space on the same disk platters as conventional disks, while requiring minimal changes to the manufacture process [4]. As a fact, Seagate [5] has already released 5 TB and 8 TB SMR drives, while HGST [6] has recently announced its HM-SMR drive in the size of 14 TB.

An SMR drive is built on the same magnetic recording and manufacture process as conventional disks, with the major difference that disk tracks are overlapped. Although overlapped tracks enable significant disk capacity improvement, data management complexity does increase in SMR drives. One particularly challenging problem is the random write constraint imposed on SMR drives [3], [7], [8], which causes rather poor random write performance and severe I/O amplifications. Though we believe that SMR will play an increasingly important role in massive storage systems due to its areal density advantage [9], [10], system researchers and SMR drive vendors should work in tandem to ensure the smooth adoption of SMR drives. It is generally recognized that new technology has been better adopted to a system in an evolving approach rather than a revolutionary one [8], [11]. High-level systems can craft friendly access patterns presented to the underlying storage system, while the storage system exposes informational hints to provide flexibility. When it comes to SMR technology, these research efforts include designing SMR-oriented on-disk data layout [12], reducing internal data movement [13], developing SMR-friendly file systems [14], [15], and providing an abstract layer of data management [16].

In this paper, we choose key-value systems as our target applications to demonstrate how SMR drives can satisfy the growing storage requirement since key-value stores are becoming more and more important as the backbone supporting a large variety of modern applications, including web indexing [1], [17], social networking [18], [19], photo stores [20], and cloud store [21]. The sizes of key-value stores are skyrocketing and trillion-items key-value systems are not uncommon [22] in data centers. The majority of modern key-value store implementations, including BigTable [1], LevelDB [17], Cassandra [23], RocksDB [20], etc., are based on the Log-Structured Merge trees (LSM-trees) due to its optimized sequential write property. However, even though

LSM-trees buffer writes to create sequentiality in individual files (i.e., SSTables), access patterns seen at the disk level are yet not SMR-friendly [16]. As a result, building LSM-tree based key-value stores on SMR drives still imposes significant challenges due to LSM-trees' inherent I/O amplification problem and the operational peculiarities of SMR drives. LSM-trees periodically perform compactions by reading, sorting, and rewriting files with overlapping key ranges, while these processes cause internal device I/O amplifications on SMR drives.

To reconcile compactions in LSM-trees and I/O amplifications in SMR drives, we introduce *sets* in an LSM-tree to group related files that are involved in a compaction and suggest *dynamic bands* in HM-SMR drives to accommodate the storage of sets, which together comprise the key ideas of our proposed key-value store named SEALDB. SEALDB in its design is aware of the characteristics of both SMR drives and LSM-trees, entailing a good cooperative optimization of the system and device to deliver a high density and high performance key-value store. The technical contributions of our work are summarized as follows:

- SEALDB exploits the inherent relationship between SSTables in an LSM-tree and groups relevant SSTables of each compaction into a *set*. A set essentially aggregates SSTables scattered around the disk into a coarse-grained unit, which helps to serialize I/O patterns to the shingled disk and improve compaction efficiency.
- SEALDB eliminates the auxiliary I/O amplification in HM-SMR drives by employing the varying size *dynamic bands*. Dynamic bands not only store sets in contiguous disk space to gather semantically related data for LSM-trees, but also increase the utilization of disk space by mitigating space wastage.

To the best of our knowledge, SMRDB is one of the very few research works on investigating the applicability of SMR drives for key-value stores [16], [24], [25]. Experimental results demonstrate SEALDB shows impressively better performance than LevelDB and SMRDB for both micro-benchmarks and the cloud YCSB workloads.

II. Background and Motivation

The high storage density is highly demanded by KV stores. As the representative techniques of high capacity storage device and KV stores, both SMR drives and LSM-trees have their merits and demerits. In this section, we first present the background knowledge of SMR technologies and LSM-trees. Then, we discuss challenges in building LSM-tree based key-value stores on SMR drives.

A. Shingled Magnetic Recording

Shingled Magnetic Recording (SMR) is a recently proposed technology to increase the disk areal density by spacing tracks more closely. In SMR drives, tracks overlap in one direction like shingles on a roof. A group of overlapped tracks is called a *band* and two adjacent bands are separated from each other by a dedicated protection region called *guard region*. Reads and

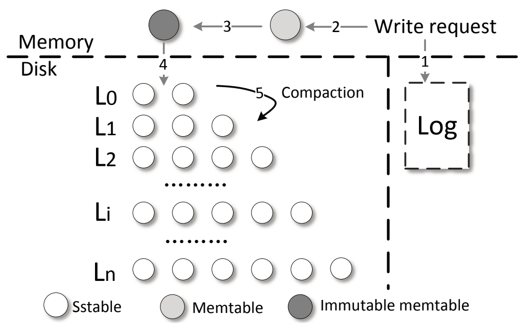


Fig. 1. **LevelDB Architecture.** *This figure shows steps of LevelDB serving write requests: (1) log file; (2) Memtable; (3) immutable Memtable (4) SSTable in L_0; (5) compaction.*

sequential writes in SMR drives are carried out as with conventional hard disks. However, random writes require expensive read-modify-write operations on a whole band to protect the data residing on the subsequently overlapped tracks [3], [4], which results in a large write amplification. To diminish this imperfection, researchers have proposed various techniques in the following three categories: DM-SMR (drive-managed) addresses random writes in a shingled translation layer (STL) [8], [12], [26]; HM-SMR (host-managed) only serves SMR-friendly writes from specific host applications [14], [15], [24]; and HA-SMR (host-aware) takes benefits from both host and STL [27]. Specifically, to provide host applications various parameters of SMR drives, there are ongoing effort on standardized interfaces for HM-SMR and HA-SMR drives (e.g., the ZBC/ZAC in T10/T13 [16], [28]). In this paper, we build SEALDB on a raw HM-SMR drive without physically divided bands and persistent cache, so we can utilize all tracks and get the maximum capacity of SMR drives. This raw HM-SMR is preferably written sequentially and allowed to write anywhere with the promise of never overlapping valid data. The HM-SMR drive in our study is similar to Caveat-Scriptor [29], but it has no extra restrictive parameters.

B. LSM-tree and LevelDB

Multi-level structured LSM-trees are widely deployed in modern key-value stores, such as the open sourced LevelDB [17] from Google Inc. LSM-trees provide a competitive write performance by batching data writes into Memtables in memory. Memtables are then flushed to disks, providing sequential writes in each individual file. The architecture of an LSM-tree is shown in Figure 1, where the size limit of L_{i+1} is typically 10 times of L_i, and this size factor is called amplification factor (AF). To compact an SSTable from L_i to L_{i+1}, LevelDB first reads required SSTables into memory, including an SSTable in Level L_i called victim SSTable, and several SSTables in L_{i+1} which have key ranges overlapping that of the victim SSTable (called overlapped SSTables); then LevelDB merges and sorts SSTables being fetched into memory; finally it writes the resultant SSTables back to L_{i+1}. Compactions are frequently conducted in the background throughout the lifetime of an LSM-tree to maintain the balance

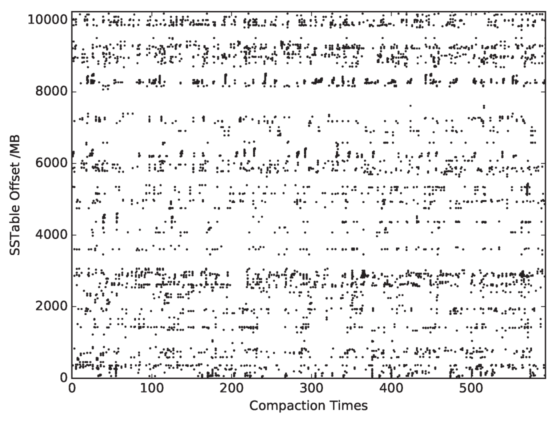


Fig. 2. **SSTables' distribution for each compaction.** *This figure shows SSTables' distributions in the disk space for each compaction when randomly loading a 10 GB database. The accesses of each compaction span the disk space causing the random I/Os problem.*

and enable an efficient lookup. However, it has been observed that, due to compactions, write amplifications of an LSM-tree are $10\times$ on average [30], [31].

C. Challenges

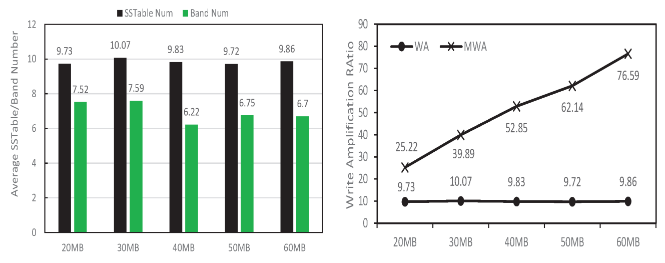
As aforementioned, SMR drives provide enlarged capacity for key-value stores, and LSM-trees offer file sized sequential writes for SMR drives. These properties can potentially benefit each other of them. However, current KV store is not a perfect fit for SMR due to the random I/Os between files, which exactly motivates us to propose SEALDB to pave a way for deploying SMR drives in KV stores. The challenges in designing an LSM-tree aware and SMR friendly key-value store are as following.

1) Random I/Os of LSM-trees: Existing key-value stores generally access on-disk data via a file system. However, the mature and widely used file system Ext4 is not strictly sequential. This HDD oriented file system is not SMR-friendly and it delivers sub-optimal performance for SMR drives as demonstrated in several researches [8], [16]. Although it tries to put all blocks of a file in the same block group [32], different files even they are semantically related can be placed separately, inducing random write penalty in SMR drives. As for the LSM-trees based Key-value store on Ext4, SSTables of one compaction are separately stored on disks, resulting in disperse reads and writes during compactions. We define these disperse accesses of each compaction as random I/Os of LSM-trees.

To demonstrate the random I/Os of LSM-trees, we experimentally measure the distribution of SSTables during each compaction by running LevelDB on Ext4 and a hard disk drive (Seagate ST1000DM003). We randomly load a 10 GB key-value store and record the physical address of each SSTable for every compaction via a Linux tool named "Ext4 Magic". Other evaluation parameters are the same with LevelDB in Section IV. Figure 2 shows that 600 compactions are conducted during this random load. For each compaction, SSTables are separately written to different locations, almost scattered around

TABLE I
MEANINGS OF THE SYMBOLS

WA	Write amplification from an LSM-tree
AWA	Auxiliary write amplification from SMR drives
MWA	Multiplicative overall write amplification
	$MWA = WA \times AWA$



(a) SSTables in bands (b) Write amplification

Fig. 3. **SSTables' distribution and corresponding write amplification.** *Figure (a) shows the average number of SSTables and bands involved in a compaction on SMR emulators with different band sizes. Figure (b) shows the average I/O amplification at different SMR band sizes.*

the first 10 GB disk space. The dispersedly located SSTables degrade compaction efficiency as well as system performance, since random accesses lead to poor performance comparing to the sequential one. It should be noted that simply adopting a sequential-only log-structured file system cannot address this problem, as it will induce complexity and large overhead on garbage collections. Therefore, SEALDB tries to solve the random write problem on Key-value store in itself.

2) Multiplicative write amplification: For SMR drives, random accesses of SSTables not only bring an inferior performance but also result in a large multiplicative write amplification. Repeating the above experiments on five emulated conventional SMR drives with band sizes ranging from 20 MB to 60 MB, we obtained the distribution of SSTables on SMR bands. Figure 3(a) shows the average number of SSTables written in one compaction and the average number of involved bands. Taking the 40 MB band size SMR drive as an example, on average a compaction rewrites 9.83 SSTables and these SSTables are written back separately to 6.22 bands.

The random accesses of SSTables on SMR bands severely increase write amplification of LSM-trees. For a clear description, write amplification from LSM-trees is denoted as (WA), which is the ratio of data size in compactions to user write data size; the auxiliary write amplification from SMR drives is denoted as AWA, which is the ratio of data size in disk writes to the data size in compactions; the overall write amplification of LSM-trees on SMR drives is denoted as (MWA), which is the multiplication of WA and AWA, as shown in Table I. To quantitatively evaluate I/O amplification from LSM-trees and SMR drives, we record the data volume from users, compactions and SMR drives separately. Figure 3(b) shows the write amplification from LSM-trees and multiplicative amplification of the system. As an example, for a 40 MB band sized SMR drive, write amplification deteriorates from $9.83\times$ to $52.85\times$ (i.e., from WA to MWA). This serious MWA of

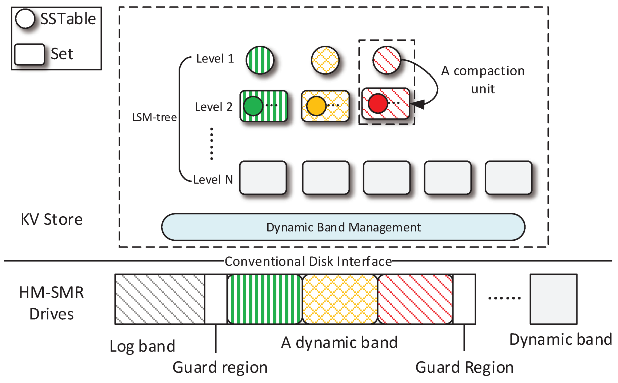


Fig. 4. **SEALDB System Architecture.** *This figure depicts the overall system architecture of SEALDB. At the high level, SSTables involved in a compaction are tied together to form a* ***set****. At the device level, band sizes are not fixed to accommodate the storage of sets, named* ***dynamic band****.*

building LSM-trees on SMR drives motivates us to develop a more efficient solution for their integration. Remarkably, existing SMR drives with a media cache cannot address the MWA problem, since cache cleaning processes induce large latency as well as write amplification and bring a bimodal behavior, as demonstrated in [8], [27].

III. System Design

The severe multiplicative write amplification of building LSM-trees on SMR drives can be mitigated by employing co-operative optimizations to bridge the semantic gap of upper-layer LSM-trees and underlying SMR drives. To achieve that, we propose SEALDB, a novel LSM-tree aware and SMR friendly key-value store. SEALDB realizes a better synergy between LSM-trees and SMR drives by employing the following two critical techniques.

First, for LSM-trees, we concatenate SSTables involved in each compaction, and group them into a *set*. Taking sets as the basic unit of compactions, it refrains multiple SSTables from spreading around disks during compactions. The compaction efficiency thus improved by creating a larger sequential read and write granularity. Second, for HM-SMR drives, we introduce variable sizes bands, named *dynamic bands*. To fully comply with the constraints of SMR drives, SEALDB manages data in dynamic bands by storing sets sequentially and allowing inserts without overlapping valid data. With the assistance of sets, dynamic bands eliminate the auxiliary write amplification from SMR drives, which significantly mitigates the multiplicative write amplification. As shown in Figure 4, sets and dynamic bands are located in the high level LSM-tree and the device level HM-SMR drive respectively.

A. Sets in LSM-trees

LSM-trees merge, sort, and rewrite SSTables in two adjacent levels to compact data to deeper levels. SSTables in a compaction have approximate key ranges, but they are stored dispersedly almost spanning the disk space (Figure 2), which induces large random accesses during compactions and

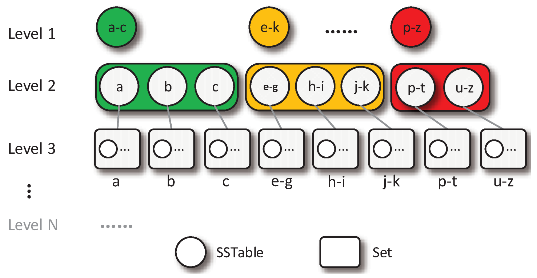


Fig. 5. **Set.** *This figure shows the organization of sets in an LSM-tree. Each SSTable in Level 1 have its sets in Level 2, each SSTable in Level 2 has its set in Level 3, and so on. The SSTable and set in the same color shares the same key range and forms a compaction unit.*

degrades system performance (Section II-C). To address the random write problem of LSM-trees, we propose *set*.

Since the victim SSTable and its overlapped SSTables share approximate key ranges, SSTables involved in a compaction can be known in prior. Leveraging this regularity, we preprocess overlapped SSTables and avoid semantic related SSTables being read/written separately. SEALDB groups overlapped SSTables of each compaction and defines them as a ***set***. More specifically, for every SSTable in Level L_i, its corresponding overlapped SSTables in Level L_{i+1} form a *set*. The key range of a set is approximate to the victim SSTable, and the key range of each overlapped SSTable within a set is part of the victim SSTable. The victim SSTable in question together with its set comprise a 'compaction unit', as shown in Figure 4. According to the default amplification factor (AF) between two adjacent levels in LSM-trees, a set contains 10 SSTables theoretically. That is to say, if an SSTable is 4 MB, the set size is 40 MB. However, set sizes heavily depends on workload distributions. We record set sizes in our evaluation (Section IV-B1), and experimental results show that a set contains 6.87 SSTables and the set size is 27.48 MB on average.

Figure 5 shows an example demonstrating the organization of *sets* in LSM-trees. As it is shown, the key range of the first SSTable in Level 1 is '$a - c$'. Its overlapped SSTables in Level 2 highlighted in green color form a set. This set includes three SSTables, and their key ranges fall into the range of '$a - c$'. Similarly, the next 3 SSTables in Level 2 form a set corresponding to the victim SSTable '$e - k$' in Level 1, and the last 2 SSTables in L_2 form a set corresponding to the key range of SSTable '$p - z$' in Level 1. By analogy, for each SSTable in L_2, it has a corresponding set in L_3, for each SSTable in L_3, it has a corresponding set in L_4, and so on. An LSM-tree thus contains lots of compaction units between adjacent levels, and every level contains multiple sets where each set includes several SSTables. As a result, a set groups overlapped SSTables in each compaction and allows them to be read and written as a unit. However, be minded that sets do not exist in L_0 and L_1. Because some SSTables in L_0 are overlapped with key ranges, an overlapped SSTable in L_1 might belong to several victim SSTables in L_0.

As a set is produced or faded by a compaction, the SSTables in a set are either valid or invalid at the same time. Comparing

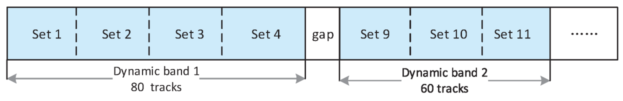


Fig. 6. **Dynamic bands on an SMR drive.** *This figure shows an example of two dynamic bands on an SMR drive, where each band contains different numbers of tracks and sets.*

to a conventional compaction, a set only helps to assemble overlapped SSTables into a unit, while SSTables involved in each compaction remain the same. Hence, LSM-trees still keep balance by compactions, and **multiple random accesses on scattered SSTables are turned into a large sequential one by sets**. Sets can be applied to various LSM-tree based key-value stores. The performance of key-value stores is improved as a result of more efficient data accesses during compactions. Moreover, in this study, sets not only improve the compaction efficiency but also provide a good unit for designing an LSM-tree aware and SMR friendly KV store.

B. Dynamic bands on SMR drives

Random accesses from conventional LSM-tree based key-value stores induce large auxiliary write amplification on SMR drives as aforementioned in Section II-C. However, following the semantic information of LSM-trees conveyed by *sets*, SMR drives can obviate random accesses and auxiliary write amplification. In this section, we introduce ***dynamic bands***, the varying size bands on HM-SMR drives, to accommodate the storage of sets and eliminate auxiliary write amplification.

1) Dynamic band: Different from conventional SMR drives with fixed size physical bands and guard regions, SEALDB works on a primitive HM-SMR drive only with shingled tracks, in which no write address restrictions are enforced. Instead, the host is aware of drive characteristics that enable it to make safe data placement decisions [29]. For SEALDB, the size of each band is dynamically changed according to data accesses of LSM-trees, so we name it *dynamic band*. In a dynamic band, multiple sets are settled sequentially. Invalid space is reused as free space. Write requests with suitable size can be inserted into those free space, as long as a safety guard region is reserved by leaving tracks unwritten to avoid overlapping valid data in the shingled direction. As a result, disk space between two guard regions form a *dynamic band*. Hence, each set is stored in a contiguous address space within a dynamic band, and every dynamic band includes multiple sequentially located sets. Figure 6 shows an example of dynamic bands on an SMR drives, where the first dynamic band contains four sets, and the second one includes three sets.

There are three theoretical foundations for our *dynamic band*. First, the optimized LSM-tree provides a coarse access unit 'set' for compactions. To take this benefit, a set must be stored on SMR drives in a contiguous physical space for efficient accesses, thus we store a set within a dynamic band entirely. Second, the property of SMR drives enables sequential writes in the shingled direction, and sequential writes never overlap data. Hence, multiple sets can be appended in a dynamic band without guard regions. Third, as long as a 'guard region' is reserved between the inserted data and the previously written one in the shingled direction, inserting to SMR drives does not overlap or induce write amplification [29]. Hence, a 'guard region' is only needed when an inserting set may damage subsequent valid data.

2) Dynamic band management: Based on the above discussion, we propose *'dynamic band management'* to manage dynamic bands on SMR drives. Generally, *dynamic band management* writes sets to an HM-SMR drive through appending, but inserts are allowed with the promise of no overlaps to valid data for fully utilizing disk space. The 'dynamic band management' policy serves write requests and manages free space as follows. In the beginning, SEALDB appends data sequentially. When a compaction takes place, the corresponding invalid set is marked as free space. The free space from faded sets is organized by a sorted array of double linked list, named *free space list*, and each array element is aligned with an SSTable size (4 MB). Free space regions with similar sizes are tracked on an array element by a double linked list. The residual space indicates the whole not-yet banded disk space. After that, when allocating space for an incoming set, SEALDB first searches in the free space list by binary searching the sorted array and picking the first free space in its linked list with the complexity of $O(logn)$. The size of a free space region (S_{free}) is required to be not less than the sum of a request size (S_{req}) and a guard region (S_{guard}), as shown in Equation 1. Once an exact size matched free space region is found, this free space is split into a data writing region and a guard region. If a larger free space region is found, data is inserted to that region and the remaining space is returned to the free space list. If there is no suitable free space, SEALDB just appends data to the tail of valid data on disks, at the non-banded region. Using dynamic band management, subsequent valid data will not be overlapped and no auxiliary write amplification is caused. Comparing to managing space in the unit of SSTable, the large size set makes dynamic band management more efficiently as it tremendously reduces metadata and unusable fine fragments on disks.

$$S_{free} \geq S_{req} + S_{guard} \tag{1}$$

The connection between dynamic bands and sets provides HM-SMR drives and LSM-trees an optimized synergy. For LSM-trees, **dynamic bands deliver a suitable data layout, where a set is always stored in a continuous physical space within a dynamic band**. Hence, **random write and corresponding auxiliary write amplification is eliminated**. For SMR drives, the coarse granularity access unit 'set' helps to **simplify the space management of dynamic bands and reduce small fragments**. In addition, **dynamic bands can fully utilize bands and contribute to improving usable disk space in HM-SMR drives**. On the contrary, storing sets in conventional SMR drives with fixed bands and gaps or adopting a buddy system results in space wastage due to partially used bands and unnecessary guard regions, since set sizes are variably changed with workloads.

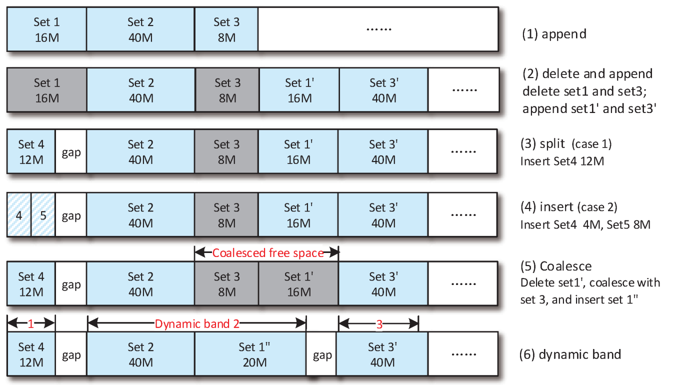

Set 1 16M
Set 2 40M
Set 3 8M
......
(1) append
Set 1' 16M
Set 3' 40M
(2) delete and append delete set1 and set3; append set1' and set3'
Set 4 12M
gap
(3) split (case 1) Insert Set4 12M
4
5
(4) insert (case 2) Insert Set4 4M, Set5 8M
Coalesced free space
(5) Coalesce Delete set1', coalesce with set 3, and insert set 1''
1
Dynamic band 2
3
Set 1'' 20M
(6) dynamic band

Fig. 7. **On-disk operations of dynamic bands.** *This figure shows an example series of operations occurring to SMR space managed by dynamic band management. Appending is the primary operation for write requests; compactions generate invalid data; inserting a write request splits a free space region and results in a guard region (gap) or another free space region; coalescing combines adjacent free space regions.*

C. On-disk operations of dynamic bands

We keep the interface of key-value stores unchanged (i.e., get, put, delete) because optimizations in LSM-trees and SMR drives are behind that interface. In this section, we abstract each of the main on-disk operations from the interface of LSM-trees and SMR drives, including *read, write, delete, coalesce,* and *split*. SEALDB serves read requests from LSM-trees without significant changes as SMR drives impose no constraints on read.

Write SEALDB serves write requests through append or insert. SEALDB appends data to disk at first. Afterwards, once some sets become invalid, they are reused as free space and added into free space list by dynamic band management. Write requests that meet the space constraint of Equation 1 are inserted into free space, otherwise they are appended to the disk space.

Delete Delete happens when a victim SSTable and its set compact and regenerate a new set. For an invalid set, SEALDB deletes it by adding to the free space list. For an invalid victim SSTable, we only mark it as invalid and record its key range in the set it belongs to. The space of an invalid victim SSTable is recycled until the set it belongs to becomes invalid. SEALDB gives priority to compact the set with more invalid SSTables, hence fragments can be recycled implicitly with no overhead.

Coalesce When a set becomes invalid, SEALDB checks free space at its adjacent locations and coalesces free space before updating the free list.

Split When a write request inserts into a large free space region (i.e., exceeding the sum size of the request and a gap), this free space is split into a used space and another free space. We add the unused free space back to the free list to serve other write requests.

For a better understanding, Figure 7 gives an example showing the evolution of a shingled disk space experiencing a series of operations. Please note that we assume the size of a guard region is 4 MB. In subfigure (1), the first three sets are written to SMR drives through appending. In subfigure (2), *set 1* undergoes a compaction, thus *set 1* is deleted, and the regenerated *set 1'* is appended to SMR drives, so does *set 3*. In subfigure (3), *set 4* (12 MB) meets the insert requirement of no overlaps (reserving a guard region), so *set 4* is inserted into the free space region previously occupied by *set 1*. The space is split into a used space and a guard region. If the size of *set 4* is 4 MB, in subfigure (4), the remaining region can serve another write request *Set 5* (8MB). In this case, only one gap is needed to guarantee not to overlap *set 2*, while the guard region between *set 4* and *Set 5* is obviated, since *Set 5*'s append does not damage any data. In subfigure (5), *Set 1'* is deleted, thus two adjacent invalid free space regions can be coalesced into a larger one to serve write requests. In subfigure (6), after a series of operations, three resultant dynamic bands with varying sizes are shown in the space of SMR drives, i.e., 12 MB, 60 MB, and 40 MB.

D. Implementation

Theoretically our proposed techniques (dynamic bands and sets) can be applied and implemented in any KV stores based on LSM-trees, because they focus on SMR drives and the structure of LSM-tree instead of any specific KV stores. For demonstration purpose, we implement SEALDB based on LevelDB 1.19, a popular and reputable LSM-tree based key-value store. We believe this implementation shows the benefits of our schemes and the potential of deploying our techniques in other LSM-trees based KV stores. First, for the implementation of sets, we revise the code of compaction processes in LevelDB to change the get/put unit from SSTables to sets. Second, since SEALDB is a direct-on-disk key-value storage system without file systems, we add an indirection from file name to disk location (i.e., physical block address, PBA) for KV stores accessing SMR drives. Third, we implement dynamic bands on an emulated primitive HM-SMR drive due to the needs of defining dynamic bands and controlling the PBA. The emulated SMR drive for SEALDB only has shingled tracks in disk platters and no physically partitioned zones and bands. SMR constraints in the emulated HM-SMR drive is implemented according to [27], [29]. The overall disk platter is managed by *dynamic band management*, and guard regions are assigned by reserving non-written shingled tracks.

IV. Evaluation

In this section, we present evaluation results of SEALDB and its competitors. All experiments are run on a testing machine with 16 Intel(R) Xeon(R) CPU E5-2660 @ 3.30GHz processors and 8-GB of memory. The operating system is 64-bit Linux 4.4.0. The storage device used for emulating SMR drives is a 1 TB Seagate ST1000DM003 HDD, which has the same read and sequential write performance as the SMR drive available on market (Seagate ST5000AS0011). Table II gives a performance comparison of the two disks. Configurations of evaluation include LevelDB, SEALDB, and SMRDB. SMRDB [24] is an SMR friendly KV store based on LSM-trees. The design choices of SMRDB include enlarging SSTables to the band size, assigning SSTables to dedicated

TABLE II
PERFORMANCE COMPARISON ON HDD AND SMR

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

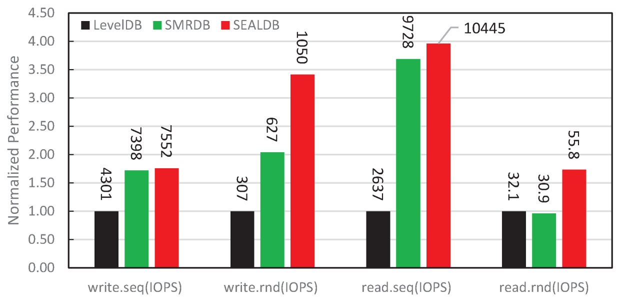


Fig. 8. **Basic performance on micro-benchmark.** *This figure shows the basic performance of sequential/random read and write normalized to LevelDB's performance. The number on top of each bar shows the actual throughput achieved (IOPS).*

bands and reserving only two levels for LSM-trees where key ranges of SSTables in the same level may be overlapped. We re-implement SMRDB as faithfully as possible according to the descriptions in its paper for comparison. Both LevelDB and SMRDB are implemented on a fix-banded emulated SMR drive. The key size, value size, SSTable size, and default band size are 16 bytes, 4 KB, 4 MB, and 40 MB respectively. Particularly, the SSTable size in SMRDB is 40 MB, as it is aligned with the band size according to its design strategies. The band size in SEALDB is variably changed with requests.

A. Basic Performance

In this section, we first evaluate the basic read and write performance of three configurations, using micro-benchmarks distributed with the LevelDB. For write performance, micro-benchmarks generate 25 millions key-value records amounting to a total of 100 GB in a sequential order or a uniformly distributed random order. For read performance, it is evaluated by randomly or sequentially querying 100K key-value pairs on the 100 GB random load database. All evaluated performance is normalized to LevelDB, as shown in Figure 8.

1) Random write performance: Random write performance mainly reveals the optimization of our design strategies, as random writes incur compactions. There are two observations for random write performance: first, SEALDB improves it by $3.42\times$ relative to LevelDB. This is because a compaction in LevelDB accesses SSTables that span multiple SMR bands, bringing a large auxiliary write amplification on SMR drives (Section II-C). SEALDB benefits random write by improving compaction efficiency with sets, eliminating auxiliary write amplification with dynamic bands, and removing redundant software overhead. Second, SEALDB outperforms SMRDB by $1.67\times$, owing to the poor compaction performance of SMRDB. SMRDB's design strategies that enlarging SSTable

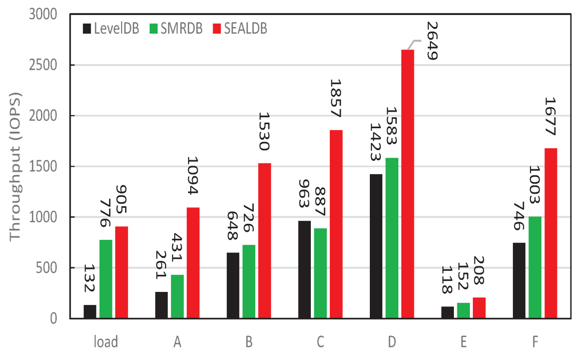


Fig. 9. **YCSB macro-benchmark performance.** *This figure shows the performance on seven YCSB workloads. Workload A is composed of 50% reads and 50% updates, Workload-B has 95% reads and 5% updates, and Workload-C includes 100% reads. Workload-D has 95% reads and 5% insert latest keys. Workload-E has 95% range queries and 5% insert new keys, Workload-F includes 50% reads and 50% RMW.*

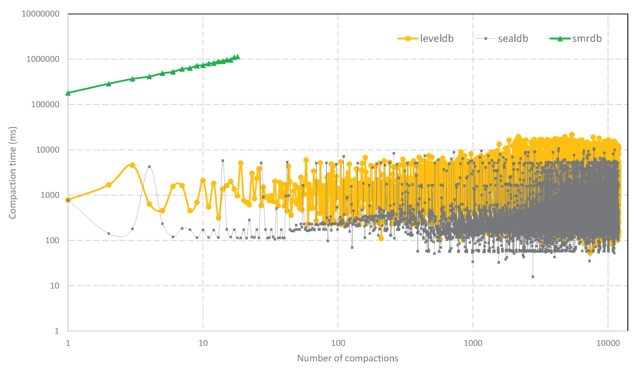
and allowing key range overlap at the two-level structured LSM-tree tremendously increase the amount of data involved in compactions, which heavily slows down its random write performance.

2) Sequential write performance: Overall, sequential write performance is much better than random write, because sequential writes never compact and thus have no write amplification. SMRDB and SEALDB exhibit quite similar sequential load performance. Their performance improvements over LevelDB come from the absence of redundant software overhead and the concise data layout.

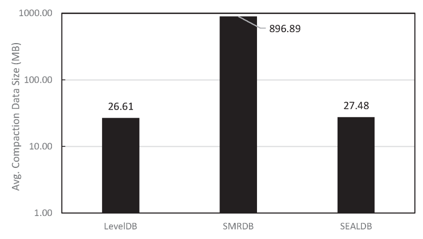
3) Sequential read performance: For sequential read performance, SEALDB surpasses LevelDB by $3.96\times$, and SMRDB is slightly inferior to SEALDB. The low sequential read throughput of LevelDB results from its non-sequential data layout, where semantic related SSTables span in multiple bands. A sequential user read request might incur multiple random reads on disks. On the contrary, both SMRDB and SEALDB store SSTables in continuous PBAs, hence the coarse sequential granularity helps to access data more efficiently. Concretely, SEALDB gathers semantic related SSTables by sets, while SMRDB enlarges each SSTable into a band size.

4) Random read performance: SMRDB has a similar random read performance as LevelDB, while SEALDB transcends them by about $1.80\times$. To get a KV item, SMRDB needs to check more SSTables as some SSTables are overlapped in its two-level structure, and LevelDB needs to check more bands as SSTables span multiple bands. However, SSTables in SEALDB are well-sorted and sequentially stored within dynamic bands.

Finally, we investigate the performance of SEALDB and other key-value stores using YCSB [33], a standard macro-benchmark suit from Yahoo!. To evaluate the overall performance, we load 25 million entries first, and then test the performance in different workloads with 100k entries. Figure 9 shows the performance and patterns of every workload. We find 1) SEALDB enjoys a larger performance improvement in random load/write dominated workloads; 2) the behaviors



(a) Compation latency



(b) Average compaction size

Fig. 10. **Compaction detail.** *Figure (a) shows the latency of every compaction when randomly loading the first 40 GB database. Figure (b) shows the average data size of compaction in three key-value stores.*

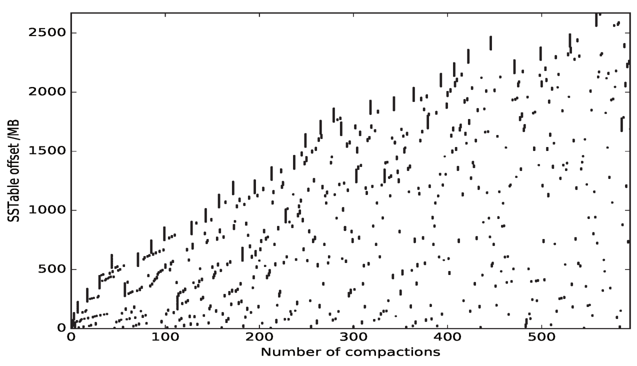
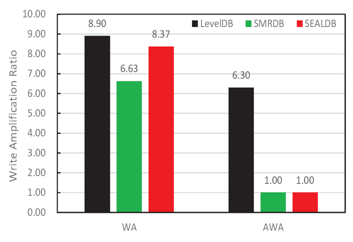
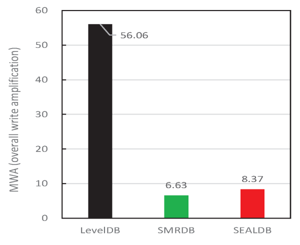


Fig. 11. **Data layout of sets in each compaction.** *This figure shows sets' distributions in the emulated SMR disk space for each compaction when randomly loading the first 10 GB database.*



(a) WA and AWA



(b) MWA

Fig. 12. **Write amplification.** *This figure shows write amplifications in KV stores (WA), SMR drives (AWA), and overall storage systems (MWA) of three competitors.*

of each competitor are aligned with what we find in micro-benchmark, which verifies the advantages of SEALDB. It should be noted that workloads in YCSB follow zipfian distribution with skewed requests, except workload-E with the latest distribution. Comparing to the uniform distribution workloads in the micro-benchmark, space locality is more prominent in zipfian distribution, which explains why SEALDB and SMRDB obtains a larger performance improvement in YCSB.

B. Detailed Analysis

Generally, the advantage of SEALDB can be attributed to three parts. First, the more efficient compactions; Second, the LSM-tree aware and SMR friendly data layout; Third, the eliminated auxiliary write amplification on SMR drives. We evaluate them in three subsections respectively, to carefully examine the details of our design and demonstrate why SEALDB delivers its performance improvement.

1) Compaction analysis: We record the detailed compaction information (i.e., latency and average data size) of three competitors when randomly loading the first 40 GB database. In Figure 10(a), the x-axis represents the order of compactions, and the y-axis shows the latency of each compaction. First, we find SEALDB and LevelDB share similar number of compactions, but SEALDB beats LevelDB with lower overall compaction latency (4.30×). The higher compaction efficiency of SEALDB comes from sets, which turn multiple scattered random accesses in a compaction into a large sequential one. Second, SMRDB experiences fewer compactions but the average compaction latency reaches 701.3 seconds, which brings 1.89× overall compaction latency comparing to SEALDB. Figure 10(b) shows the reason, where SMRDB's average amount of compaction data reaches 900 MB. This enormous data size comes from SMRDB's large SSTable and the two-level LSM-tree structure. It demonstrates that simply enlarging SSTables to fit SMR bands even produces a counter effect. More importantly, in this figure, we observe the average set size of SEALDB is 27.48 MB. As sets accumulate SSTables in each compaction, the average set size is equivalent to the average compaction data size.

2) Data layout: The performance improvement of SEALDB also takes advantage of dynamic bands. Figure 11 shows the data layout on SMR drives by tracing the physical address of each SSTable in every compaction, when randomly loading the first 10 GB database. In this figure, we observe 600 compactions, and for each compaction SSTables are written to a continuous physical address space. The consecutive offset of SSTables in each compaction represents a set. Sets fill the first 2.7 GB disk space gradually, which demonstrates the benefit of dynamic band management (i.e., writing sets sequentially and allowing inserts without overlapping valid data). Comparing this figure with the data layout of traditional LevelDB (in Figure 2), we can observe the space efficiency of SEALDB, which saves 6.3 GB disk space for storing the same 10 GB database.

3) Write amplification: Write amplification is another aspect reflecting the reason why SEALDB improves performance. In the 100 GB random load database, we evaluate write amplification from LSM-trees (WA), the auxiliary write amplification from SMR drives (AWA), and the overall multiplicative write amplification (MWA) respectively to illustrate

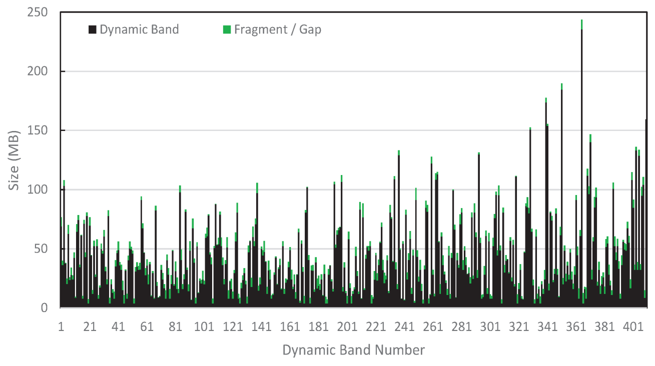


Fig. 13. **Data layout of dynamic bands and fragments.** *This figure shows the data layout of dynamic bands after randomly loading a 40 GB database. The fragments only takes a small amount of disk space.*

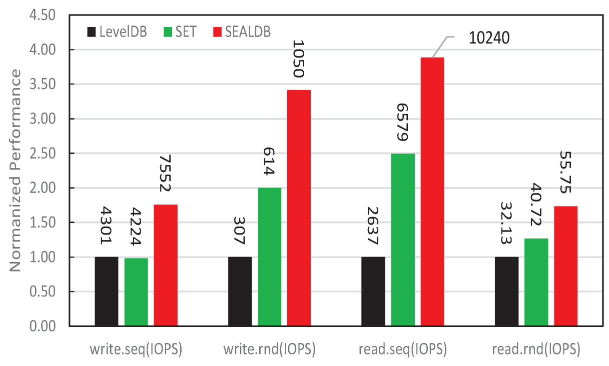


Fig. 14. **Contribution analysis of set and dynamic band.** *This figure shows the basic performance comparison of LevelDB, LevelDB with sets, and SEALDB with sets and dynamic bands.*

which part takes advantage of our design strategies.

Figure 12(a) shows that SEALDB does not mitigate the write amplification of LSM-trees by aligning semantic related SSTables into sets. However, Section IV-B1 has demonstrated the design of set contributes to more efficient data accesses in compactions. For SMRDB, it reduces write amplification of LSM-trees by the two-level structure, as it avoids KV items from constantly compacting from level 0 to level 6.

Figure 12(a) shows that both SMRDB and SEALDB are able to eliminate the auxiliary write amplification. For SMRDB, it dispels AWA by enlarging SSTable and putting each SSTable to a dedicated band, where writing an entire band does not induce auxiliary write amplification in SMR drives. For SEALDB, it eliminates AWA by taking sets as the compaction unit and adopting dynamic bands to store it, as dynamic bands fully comply with the constraints of SMR drives and never overlap valid data.

Figure 12(b) shows the overall efficacy of reducing the multiplicative write amplification. SEALDB mitigates MWA by 6.70× compared to LevelDB. The high MWA of LevelDB is produced by compactions in LSM-trees and multiplied by the AWA of SMR drives, since the unfriendly data layout of LevelDB makes SSTables in a compaction span in a large disk space.

C. Cost analysis

In this section, we conclude the overhead and contribution of each design strategy (i.e., set and dynamic band).

Set gathers data in each compaction to improve compaction efficiency. However, its combination with dynamic bands induces some overhead. Although set provides a large management granularity, the exceptional small sets can result in small fragments which is hard to reuse as free space, impairing the space utilization of SEALDB.

The benefits of dynamic band mainly own to large granularity sets, as large units reduce the software overhead of management and produce fewer small fragments. The complexity of free space list in dynamic band management is only $O(logn)$ when using the binary search in a sorted array. Considering the space utilization in dynamic bands, free space that cannot accommodate a set plus a guard region will be wasted.

To evaluate the space utilization, we randomly load a 40 GB database and record the layout of dynamic bands and fragments, ignoring free space regions that larger than the average set size (27.48 MB). As shown in Figure 13, each dynamic band is followed by a fragment or gap. The overall fragments size 1.7 GB, takes only 9.32% of the occupied space. However, these small fragments are quite difficult to be leveraged, thus SEALDB needs alternative garbage collection policies as a supplement. We leave it for our future work to further optimize SEALDB.

Finally, the contribution of *set* and *dynamic band* are analyzed respectively by evaluating the performance of LevelDB, LevelDB with *sets*, and SEALDB. From Figure 14, we find sets mainly contribute to improve read and random write performance, which accounts for 50% and 41% of the overall performance improvement. However, sequential write performance is only improved by dynamic band, since the benefit of sets comes from gathering data in compactions while no compaction happens in sequential write. Dynamic band makes improvement in every workload for its sequential dominant access pattern. With the combination of sets and dynamic bands, SEALDB achieves a decent degree of performance promotion in all workloads.

V. Related Work

Key-value stores have become important in supporting a huge variety of modern data center applications and SMR devices are gaining popularity in storage systems due to their density advantage. Aghayev and Desnoyers [3] carry out extensive experiments to understand the internal details of drive-managed SMR devices. Wu et al. [27] study the performance and characteristics of host-aware SMR devices. He et al. [26], Manzanares et al. [16], and Kadekodi et al. [29] have proposed several novel data management policies on SMR drives. These research consolidate our understanding of SMR technologies.

Leveraging SMR drives in the storage system imposes challenges. Relevant research address the challenge of building KV stores on SMR drives as follow: Skip-tree [31] compacts KV items by skipping some internal levels to reduce level by level compactions. Yao et al. [25] proposes a light-weight compaction tree to reduce I/O amplification in compactions. Kinetic [34] is an implementation of a KV store on SMR media released by Seagate. SMRDB [24] is what we compared in this study, whose key techniques consist of using the two level LSM-tree and enlarging SSTable to match the size of an SMR band. By contrast, SEALDB employs *set* and *dynamic band*, which provides a flexibility in managing disk space and performs better than SMRDB, as verified by our experiments.

VI. Conclusion

In this paper, we present *SEALDB* to address the multiplicative I/O amplification problem when running LSM-tree based key-value stores on SMR drives. First, SEALDB groups SSTables involved in each compaction into *sets* to guarantee more efficient compactions. Second, SEALDB proposes *dynamic band* on HM-SMR drives to accommodate the storage of sets and eliminate auxiliary I/O amplifications. Extensive experimental results have shown that *SEALDB* outperforms other counterparts (i.e., LevelDB and SMRDB) in various aspects.

VII. Acknowledgement

We thank the anonymous reviewers for their constructive comments and Billy Luqiu for his help to proofread the draft. Zhihu Tan and Jiguang Wan are corresponding authors of this paper. 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; the U.S. National Science Foundation grants CCF-1717660 and CNS-1702474.

References

[1] 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 OSDI*, 2006, pp. 205–218.

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

[3] A. Aghayev and P. Desnoyers, "Skylight—a window on shingled disk operation," in *Proceedings of the 13th USENIX Conference on FAST*, 2015, pp. 135–149.

[4] 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.

[5] Seagate. (2014) Archive hdds from seagate. [Online]. Available: http://www.seagate.com/www-content/product-content/hdd-fam/seagate-archive-hdd/en-us/docs/100757960a.pdf

[6] HGST. (2017) Western digitals worlds first 14tb enterprise hard disk drives. [Online]. Available: https://www.hgst.com/products/hard-drives/ultrastar-hs14

[7] 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 MSST*, 2010.

[8] A. Aghayev, T. Tso, G. Gibson, and P. Desnoyers, "Evolving ext4 for shingled disks," in *Proceedings of the 15th USENIX Conference on FAST*, vol. 1, 2017, p. 105.

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

[10] E. Brewer, L. Ying, L. Greenfield, R. Cypher, and T. T'so, "Disks for data centers," Google, Tech. Rep., 2016.

[11] S. Boyd-Wickizer, A. T. Clements, Y. Mao, A. Pesterev, M. F. Kaashoek, R. Morris, N. Zeldovich *et al.*, "An analysis of linux scalability to many cores," in *Proceedings of the 9th Symposium on OSDI*, vol. 10, 2010, pp. 86–93.

[12] Y. Cassuto, M. A. A. Sanvido, C. Guyot, D. R. Hall, and Z. Z. Bandic, "Indirection systems for shingled-recording disk drives," in *Proceedings of the 2010 IEEE 26th Symposium on MSST*, 2010, pp. 1–14.

[13] Jones, S. N, Amer, Ahmed, Miller, E. L, Long, D. DE, Pitchumani, Rekha, Strong, and C. R, "Classifying data to reduce long-term data movement in shingled write disks," *ACM Transactions on Storage (TOS)*, vol. 12, no. 1, p. 2, 2016.

[14] Seagate. Smrffs-ext4. [Online]. Available: https://github.com/Seagate/SMR_FS-EXT4

[15] C. Jin, W.-Y. Xi, Z.-Y. Ching, F. Huo, and C.-T. Lim, "Hismrfs: A high performance file system for shingled storage array," in *Proceedings of the 2014 IEEE 30th Symposium on MSST*, 2014, pp. 1–6.

[16] 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 HotStorage*, 2016.

[17] S. Ghemawat and J. Dean. (2016) Leveldb. [Online]. Available: https://github.com/Level/leveldown/issues/298

[18] T. G. Armstrong, V. Ponnekanti, D. Borthakur, and M. Callaghan, "Linkbench: A database benchmark based on the facebook social graph," in *Proceedings of the 2013 ACM SIGMOD*, 2013.

[19] R. Sumbaly, J. Kreps, L. Gao, A. Feinberg, C. Soman, and S. Shah, "Serving large-scale batch computed data with project voldemort," in *Proceedings of the 10th USENIX Symposium on FAST*, 2012.

[20] Facebook. Rocksdb, a persistent key-value store for fast storage enviroments. [Online]. Available: http://rocksdb.org/

[21] D. Beaver, S. Kumar, H. C. Li, J. Sobel, and P. Vajgel, "Finding a needle in haystack: Facebooks photo storage," in *Proceedings of the 9th Symposium on OSDI*, 2010.

[22] 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.

[23] 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.

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

[25] T. Yao, J. Wan, P. Huang, X. He, Q. Gui, F. Wu, and C. Xie, "A light-weight compaction tree to reduce i/o amplification toward efficient key-value stores," in *Proceedings of the 2017 IEEE 33th Symposium on MSST*, 2017.

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

[27] F. Wu, M.-C. Yang, Z. Fan, B. Zhang, X. Ge, and D. H.C.Du, "Evaluating host aware smr drives," in *8th USENIX Workshop on HotStorage*. Denver, CO: USENIX Association, 2016.

[28] I. T. T. Committee, "Information technology-zoned block commands (zbc). draft standard t10/bsr incits 550, american national standards institute, inc." Draft Standard, 2017. [Online]. Available: http://www.t10.org/drafts.htm

[29] S. Kadekodi, S. Pimpale, and G. A. Gibson, "Caveat-scriptor: Write anywhere shingled disks," in *7th USENIX Workshop on HotStorage*. Santa Clara, CA: USENIX Association, 2015.

[30] L. Lu, T. S. Pillai, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Wisckey: separating keys from values in ssd-conscious storage," in *Proceedings of the 14th USENIX Conference on FAST*, 2016, pp. 133–148.

[31] 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.

[32] A. K. KV, M. Cao, J. R. Santos, and A. Dilger, "Ext4 block and inode allocator improvements," in *Linux Symposium*, vol. 1.

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

[34] Kinetic open storage. [Online]. Available: https://www.openkinetic.org/