GearDB: A GC-free Key-Value Store on HM-SMR Drives with Gear Compaction

Ting Yao[1,2], Jiguang Wan[1*], Ping Huang[2], Yiwen Zhang[1], Zhiwen Liu[1],
Changsheng Xie[1], and Xubin He[2]

[1]WNLO, School of Computer Science and Technology,
Huazhong University of Science and Technology, China
[1]Key Laboratory of Information Storage System, Ministry of Education of China
[2]Temple University, USA

Abstract

Host-managed shingled magnetic recording drives (HM-SMR) give a capacity advantage to harness the explosive growth of data. Applications where data is sequentially written and randomly read, such as key-value stores based on Log-Structured Merge Trees (LSM-trees), make the HM-SMR an ideal solution due to its capacity, predictable performance, and economical cost. However, building an LSM-tree based KV store on HM-SMR drives presents severe challenges in maintaining the performance and space efficiency due to the redundant cleaning processes for applications and storage devices (i.e., compaction and garbage collections). To eliminate the overhead of on-disk garbage collections (GC) and improve compaction efficiency, this paper presents *GearDB*, a GC-free KV store tailored for HM-SMR drives. GearDB proposes three new techniques: a new on-disk data layout, compaction windows, and a novel gear compaction algorithm. We implement and evaluate GearDB with LevelDB on a real HM-SMR drive. Our extensive experiments have shown that GearDB achieves both good performance and space efficiency, i.e., on average 1.71× faster than LevelDB in random write with a space efficiency of 89.9%.

1 Introduction

Shingled Magnetic Recording (SMR) [12] is a core technology driving disk areal density increases. With millions of SMR drives shipped by drive vendors [32, 17], SMR presents a compelling solution to the big data challenge in an era of explosive data growth. SMR achieves higher areal density within the same physical footprint as conventional hard disks by overlapping tracks, like shingles on a roof. SMR drives are divided into large multi-megabyte zones that must be written sequentially. Reads can be processed precisely from any uncovered portion of tracks, but random writes risk corrupting data on overlapped tracks, imposing random write complexities [10, 12, 3]. The sequential write restriction makes log-structured writes to shingled zones a common practice [38, 32, 33], creating a potential garbage collection (GC) problem. GC reclaims disk space by migrating valid data to produce empty zones for new writes. The data migration overhead of GC severely degrades system performance.

Among the three SMR types (i.e., drive-managed, host-managed, and host-aware), HM-SMR presents a preferred option due to its capacity, predictable performance, and low total cost of ownership (TCO). HM-SMR offers an ideal choice in data center environments that demand predictable performance and control of how data is handled [15], especially for domains where applications commonly write data sequentially and read data randomly, such as social media, cloud storage, online backup, life sciences as well as media and entertainment [15]. The key-value data store based on Log-Structured Merge trees (LSM-trees) [37] inherently creates that access pattern due to its batched sequential writes and thus becomes a desirable target application for HM-SMR.

LSM-tree based KV stores, such as Cassandra [28], RocksDB [9], LevelDB [11], and BigTable [5], have become the state-of-art persistent KV stores. They achieve high write throughput and fast range queries on hard disk drives and optimize for write-intensive workloads. The increasing demand on KV stores' capacities makes adopting HM-SMR drives an economical choice [24]. Researchers from both academia and industry have been attempting to build key-value data stores on HM-SMR drives by modifying applications to take advantage of the high capacity and predictable performance of HM-SMR, such as Kinetic from Seagate [41], SMR based key-value store from Huawei [31], SMORE form Netapp [32], and others [47, 38, 48].

However, building an LSM-tree based KV store on HM-SMR drives comes with a serious challenge: the redundant cleaning processes on both LSM-trees and HM-SMR drives harm performance. In an LSM-tree, the compaction processes are conducted throughout the lifetime to clean invalid data and keep data sorted in

*Corresponding author. Email: jgwan@hust.edu.cn

multiple levels. In an HM-SMR drive, the zones with a log-structured data layout are fragmented as a result of data being invalidated by applications (e.g., compactions from LSM-trees). Therefore, garbage collection must be executed to maintain sizeable free disk space for writing new data. Existing applications on HM-SMR drives either leave the garbage collection problem unsolved [33, 22] or use a simple greedy strategy to migrate live data from partially empty zones [32]. Redundant cleaning processes, the garbage collection for storage devices in particular, degrade system performance dramatically. To demonstrate the impact of on-disk garbage collection, we implement a cost-benefit and a greedy garbage collection strategy similar to the free space management in log-structured files system and SSDs [40, 2]. Evaluation results in Section 2.3 indicate that garbage collection not only causes expensive overheads on system latency but also hurts the space utilization of HM-SMR drives. Conventional KV stores on HM-SMR drives face a dilemma: either obtain high space efficiency with poor performance or take good performance with poor space utilization. The space utilization is defined as the ratio between the on-disk valid data volume and the allocated disk space.

To obtain both good performance and high space efficiency in building an LSM-tree based KV store on HM-SMR drives, we propose GearDB with three novel design strategies. **First**, we propose a new on-disk data layout, where a zone only stores SSTables from the same level of an LSM-tree, contrary to arbitrarily logging SSTables of multiple levels to a zone as with the conventional log layout. In this way, SSTables in a zone share the same compaction frequency, remedying dispersed fragments on disks. The new on-disk data layout manages SSTables to align with the underlying SMR zones at the application level. **Second**, we design a *compaction window* for each level of an LSM-tree, which is composed of $1/k$ zones of that level. Compaction windows help to limit compactions and the corresponding fragments to a confined region of the disk space. **Third**, based on the new data layout and compaction windows, we propose a new compaction algorithm called *Gear Compaction*. Gear compaction proceeds in compaction windows and descends level by level only if the newly generated data overlaps the compaction window of the next level. Gear compaction not only improves the compaction efficiency but also empties compaction windows automatically so that SMR zones can be reused without garbage collection. By applying these design techniques, we implement GearDB based on LevelDB, a state-of-art LSM-tree based KV store. Evaluating GearDB and LevelDB on a real HM-SMR drive, test results demonstrate that GearDB is 1.71× faster in random writes compared to LevelDB, and has an efficient space utilization of 89.9% in a bimodal distribution (i.e., zones are either nearly empty or nearly full).

2 Background and Motivation

In this section, we discuss HM-SMR and LSM-trees, as well as challenges and our motivation in building LSM-tree based KV stores on HM-SMR drives.

2.1 Shingled Magnetic Recording (SMR)

Shingled Magnetic Recording (SMR) techniques provide a substantial increase in disk areal density by overlapping adjacent tracks. SMR drives allow fast sequential writes and reads like any conventional HDDs, but have destructive random writes. SMR drives are classified into three types based on where the random write complexity is handled: in the drive, in the host, or co-operatively by both [17]. Drive-managed SMR (DM-SMR) implements a translation layer in firmware to accommodate both sequential and random writes. It acts as a drop-in replacement of existing HDDs but suffers highly unpredictable and inferior performance [1, 4]. Host-managed SMR (HM-SMR) requires host-software modifications to reap its advantages [33]. It accommodates only sequential writes and delivers predictable performance by exposing internal drive states. Host-aware SMR (HA-SMR) lies somewhere between HM-SMR and DM-SMR. However, it is the most complicated and obtains maximum benefit and predictability when it works as HM-SMR [45]. Research has demonstrated that SMR drives can fulfill modern storage needs without compromising performance [38, 32, 33, 41].

Like many production HA/HM-SMR drives [42, 18, 15], the drive used in this study is divided into 256 MB-sized zones. Each zone accommodates strict sequential writes by maintaining a write pointer to resume the subsequent write. A guard region separates two adjacent zones. A zone without valid data can be reused as an empty zone via resetting the zone's write pointer to the first block of that zone. All the intricacies of HM-SMR are exposed to the software by a new command set, the T10 Zone Block Commands [19]. To comply with the SMR sequential write restrictions, applications or operating systems are required to write data in a log-structured fashion [38, 32, 33, 4, 27]. However, the log-structured layout imposes additional overhead in the form of garbage collection (GC). GC blocks foreground requests and degrades system performance due to live data migration.

2.2 LSM-trees and Compaction

Log-Structured Merge trees (LSM-trees) [37] exploit the high-sequential write bandwidth of storage devices by writing sequentially [39]. Index changes are first deferred and buffered in memory, then cascaded to disks level by level via merging and sorting. The Stepped-Merge technique is a variant of LSM-trees [21], which changes a single index into k indexes at each level to reduce the cost of inserts.

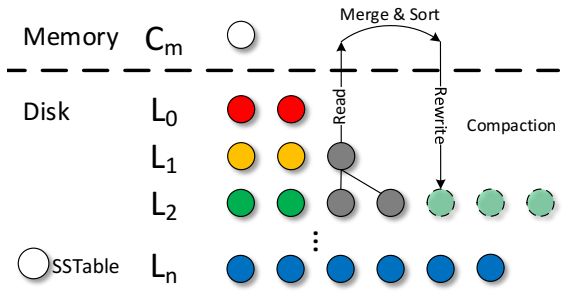


Figure 1: **A compaction process in an LSM-tree based KV store.** *This figure shows the LSM-tree data structure, which is composed of a memory component and a multi-leveled disk component. Compaction is conducted level by level to merge SSTables from the lower to higher levels.*

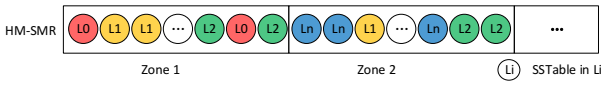


Figure 2: **Conventional on-disk data layout with log-structured writes.** *This figure shows that the conventional log write causes SSTables of different levels mixed in the zones on HM-SMR drives.*

Due to their high update throughput, LSM-trees and their variants have been widely used in KV stores [39, 8, 24, 5, 28]. LevelDB [11] is a popular key-value store based on LSM-trees. In LevelDB, the LSM-tree batches writes in memory first and then flushes batched data to storage as sorted tables (i.e., SSTable) when the memory buffer is full. SSTables on storage devices are sorted and stored in multiple levels and merged from lower levels to higher levels. Level sizes increase exponentially by an amplification factor (e.g., AF=10). The process of merging and cleaning SSTables is called compaction, and it is conducted throughout the lifetime of an LSM-tree to clean invalid/stale KV items and keep data sorted on each level for efficient reads [37, 43]. Figure 1 illustrates the compaction in an LSM-tree data structure (e.g., LevelDB). When the size limit of L_i is reached, a compaction starts merging SSTables from L_i to L_{i+1} and proceeds in the following steps. First, a victim SSTable in L_i is picked in a round-robin manner, along with any SSTables in L_{i+1} whose key range overlaps that of the victim SSTable. Second, these SSTables are fetched into memory, merged and resorted to generate new SSTables. Third, the new SSTables are written back to L_{i+1}. Those stale SSTables, including the victim SSTable and the overlapped SSTables, then become invalid, leaving dispersed garbage data in the disk space.

2.3 Motivation

The log-structured write fashion required by HM-SMR drives could lead to excessive disk fragments and therefore

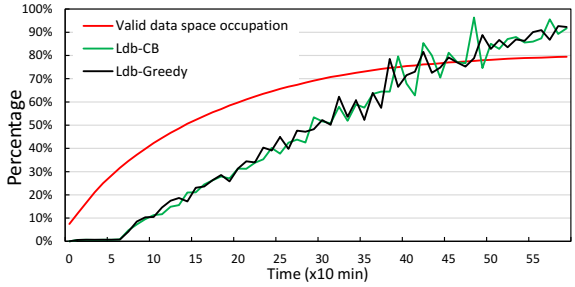


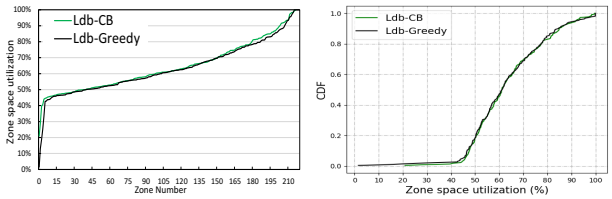
Figure 3: **Cost of garbage collections.** *The green line shows the ratio of valid data volumes to the test disk space; the other two lines show the ratio of time consumption of garbage collections in every ten minutes during the random loading.*

necessitates costly garbage collections to deal with them. Specifically, in an LSM-tree based KV store on HM-SMR drives, compaction cleans invalid KV items in LSM-trees but leaves invalid SSTables on the disk. Especially, the arbitrarily sequential writes of the conventional log result in SSTables from multiple levels that have different compaction frequency being mixed in the same zones, as shown in Figure 2. As compaction procedures constantly invalidate SSTables during the lifespan of LSM-trees, the fragments on HM-SMR drives become severely dispersed, necessitating many GCs. Due to the high write amplification of LSM-trees [29] (i.e., more than 12×) and the huge volume of dispersed fragments caused by compaction, passive GCs become inevitable. Passive GCs are triggered when the free disk space is under a threshold (i.e., 20% [36]) and clean zone space by migrating valid data from zones to zones.

To demonstrate the problems of garbage collections in the LSM-tree based KV store on HM-SMR drives, we implement LevelDB [11], a state-of-art LSM-tree based KV store, on a real HM-SMR drive using log-structured writes to zones. We implement both greedy and cost-benefit GC strategies [40, 2, 32] to manage the free space on HM-SMR drives. The greedy GC cleans the zone with the most invalid data by migrating its valid data to other zones. Cost-benefit GC selects a zone by considering the age and the space utilization of that zone (u) according to Equation 1 [40]. We define the age of a zone as the sum of SSTables' level ($\sum_0^n L_i$), based on the observation that SSTables in a higher level live longer and have a lower compaction frequency, where n is the number of SSTables in the zone and L_i is the level of an SSTable. The cost includes reading a zone and writing back u valid data.

$$\frac{benefit}{cost} = \frac{FreeSpaceGain \times ZoneAge}{cost} = \frac{(1-u)\times\sum_0^n L_i}{1+u} \quad (1)$$

With the parameters described in Section 5, we randomly load 20 million KV items to an HM-SMR drive using only 70



(a) Zone space utilization (b) CDF of zone space utilization

Figure 4: **Zone space utilization.** *Figure a shows the zone space utilization of each zone after random loading the first 40 GB database, plotting in the order of increasing space utilization. Figure b shows the CDF of the zone space utilization.*

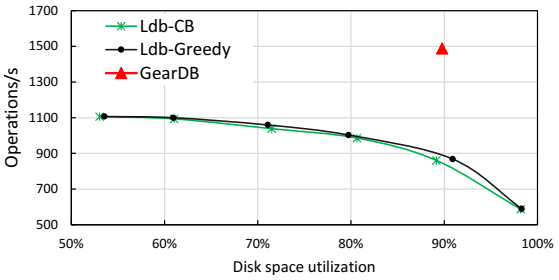


Figure 5: **The throughput vs. different space utilization.** *This figure shows that in the conventional KV store on an HM-SMR drive, the system performance decreases with the space utilization. It is unable to get both good performance and space efficiency simultaneously.*

GB disk space comprised of 280 shingled zones. The valid data volume of the workload is about 54 GB, approximating 80% of the disk space, due to duplicated and deleted KV entries. Through this experiment, we have made the following observations. First, we calculate the valid data volume and the GC time every ten minutes during the load process. As shown in Figure 3, the time consumption of GC grows with the valid data volume. When valid data grows to about 70% of the disk space, more than half of the time in that ten minutes is spent to perform GC. GC accounts for more than 80% of the execution time when valid data reaches 76% of the disk space. The test results demonstrate that garbage collection takes a substantial proportion of the total execution time, downgrading the system performance dramatically. Because SSTables from different levels in a zone are mixed, multiple zones show similar age in cost-benefit GC policy. The similar zone ages make greedy and cost-benefit policies present almost the same performance, as they both prefer reclaiming zones with the most invalid data.

Second, we record the space utilization of each occupied zone on the HM-SMR drive after loading 10 million KV items. Figure 4 (a) shows the percentage of valid data in each zone (in a sorted way), and Figure 4 (b) shows the cumulative distribution function (CDF) of the zone space utilization. Both greedy and cost-benefit have an unsatisfactory average space efficiency of 60%. More specifically, 85% zones have a space utilization ranging from 45% to 80%. We contend that this space utilization distribution results in the significant amount of time spent in doing GC, as discussed above. The more live data in zones that is migrated, the more disk bandwidth is needed for cleaning and not available for writing new data. A better and more friendly space utilization would be a bimodal distribution, where most of the zones are nearly full, a few are empty, and the cleaner can always work with the empty zones, eliminating the overhead of GC, i.e., valid data migration. In this way, we can achieve both high disk space utilization and eliminate on-disk garbage collection overheads. This forms the key objective of our GearDB design, as discussed in the next section.

Third, by changing the threshold of GC (from 100% to 50%) on the 110 GB restricted disk capacity, we test 6 groups of 80 GB random writes to show the performance variations with disk space utilization. **The disk space utilization, or space efficiency, is defined as the ratio of the on-disk valid data volume to the allocated disk space.** As shown in Figure 5, system performance decreases with space utilization. Running on an HM-SMR drive, LevelDB faces a dilemma where it only delivers a compromised trade-off between performance and space utilization. Our goal in designing GearDB is to achieve higher performance and better space efficiency simultaneously. The red triangle mark in Figure 5 denotes the measured performance and space efficiency of GearDB, i.e., 89.9% space efficiency and 1489 random load IOPS.

In summary, with log-structured writes, existing KV stores on HM-SMR suffer from redundant cleaning processes in both LSM-trees (i.e., compaction) and HM-SMR drives (i.e., garbage collection). The expensive GC degrades system performance, decreases space utilization, and creates a suboptimal trade-off between performance and space efficiency.

3 GearDB Design

In this section we present GearDB and three key techniques to eliminate the garbage collection and improve compaction efficiency. GearDB is an LSM-tree based KV store that achieves both high performance and space efficiency on an HM-SMR drive. Figure 6 shows the overall architecture of GearDB's design strategies. First, we propose a new on-disk data layout that provides application-specific data management for HM-SMR drives, where a zone only serves SSTables from one level to prevent data in different levels from being mixed and causing dispersed fragments. Second, based on the new on-disk data layout, we design a compaction window for each LSM-trees level. Compactions only proceed within compaction windows, which restricts frag-

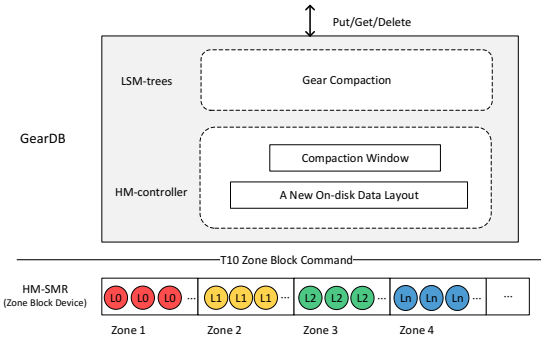


Figure 6: **The architecture of GearDB.** *This figure shows the overall structure of GearDB with three design strategies. GearDB accesses the HM-SMR drive directly via the T10 zone block command. For the data layout on an HM-SMR drive, SSTables form the same level are located in integral zones, and each zone only serves SSTables from the same level.* $\textcircled{L_i}$ *represents the SSTable from* L_i.

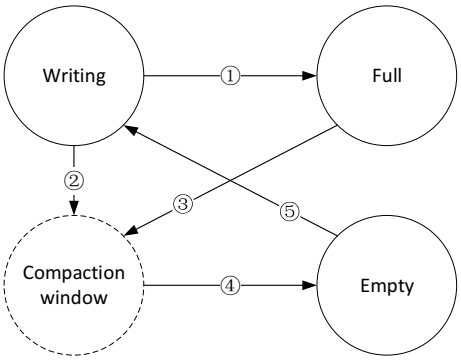


Figure 7: **Zone state transitions in GearDB.** *In GearDB, multiple zones are allocated to each level according to the data size of that level. These zones can be in three states during their lifetime, namely writing zone, full zone, and empty zone. Zones, including full zones and writing zones, rotate to construct a compaction window.*

ments in compaction windows. Third, we propose a novel compaction algorithm, called *gear compaction*, based on the new data layout and compaction windows. Gear compaction divides the merged data of each compaction into three portions and further compacts with the overlapped data in the compaction window of the next level. Gear compactions automatically empty SMR zones in compaction windows by invalidating all SSTables, so that zones can be reused without the need to garbage collection. We elaborate on the three strategies in the following subsections.

3.1 A New On-disk Data Layout

As discussed in Section 2.3, data fragments on HM-SMR drives are widely dispersed due to log-structured writes and compactions, which causes SSTables from different levels to be mixed within zones (Figure 2). To alleviate this problem, we propose a new data layout to manage HM-SMR drives in GearDB.

The key idea of the new data layout is that each zone only serves SSTables from one level, as shown in Figure 6. We dynamically assign zones to different levels of an LSM-tree. Initially, each level in use is attached to one zone. As the data volume of a level increases, additional zones are allocated to that level. Once a zone is assigned to Level L_i, it can only store sequentially written SSTables from L_i until it is released as an empty zone by GearDB. When an LSM-tree reaches a balanced state, each level is composed of multiple zones according to its size limit. Among the zones of each level, only one zone accepts incoming writes, named a *writing zone*. Sequential writes in each zone strictly respect the shingling constraints of HM-SMR drives.

Since SSTables in a zone belong to the same level, they share the same compaction frequency (or the same hotness). This data layout results in less fragmented disk space and offers convenience for the following design strategies, which potentially leads to the desired bimodal distribution and thus allows us to achieve high system performance at low cost. Additionally, sequential read performance is improved due to better spatial locality.

3.2 Compaction Windows

With our new data layout, each level in an LSM-tree has multiple zones corresponding to its data volume or size limit. To further address the dispersed fragments on HM-SMR drives based on the new data layout, we propose a compaction window for each level. A compaction window (*CW*) for a level is composed of a group of zones belonging to the level, which is used to limit compactions and fragments.

Specifically, GearDB presets a compaction window for each level of the LSM-tree. To construct a compaction window, a certain number of zones are picked from the zones belonging to that level in a rotating fashion. The compaction window size (S_{cwi}) of level L_i is given by Equation 2, where the compaction window size of each level is $1/k$ of the level size limit (L_{Li}). Hence, the compaction window size increases by the same amplification factor as the size for each level. By default, the compaction window size is $\frac{1}{4}$ of the corresponding level size. Note that the compaction window of L_0 and L_1 comprises the entire level since these two levels only take one zone in our study.

$$S_{cwi} = \frac{1}{k} \times L_{Li} \quad (1 \leq k \leq AF) \tag{2}$$

Compaction windows are not designed to directly improve the system performance. However, by limiting compactions within compaction windows, the corresponding fragments

are restricted to compaction windows instead of spanning over the entire disk space. Therefore, system performance benefits from gear compactions. Since a zone full of invalid data can be reused as an empty zone without data migration, compaction windows that filled with invalid SSTables can be released as a group of empty zones to serve future write requests. When zones of a compaction window are released, another group of zones of that level is selected to form a new compaction window. Different zones in a level rotate to constitute the compaction window, guaranteeing every SSTable gets involved in compactions evenly.

To facilitate the management of underlying SMR zones in GearDB, we divide zones into three states, namely writing zone, full zone, and empty zone. Each level only maintains a writing zone for sequentially appending newly arrived SSTables. Figure 7 shows the diagram of zone state transitions. ① A writing zone becomes a full zone once it is filled. ② ③ A writing zone or full zone can be added into a compaction window by rotation. ④ When all SSTables of a compaction window have been invalidated by gear compactions, the zones become empty and ⑤ ready to serve write requests without incurring device-level garbage collection.

3.3 Gear Compaction

Based on our new data layout and compaction windows, we develop a new compaction algorithm in this section. Gear compaction aims to automatically clean compaction windows during compactions and thus eliminate costly and redundant garbage collections.

Gear Compaction Algorithm. A gear compaction process starts by compacting L_0 and L_1, called active compaction. Active compaction triggers subsequent passive compactions, and compactions progress from lower levels to higher levels. For a conventional compaction between L_i and L_{i+1} in LevelDB, the merge-sorted data are directly written back to the next level (i.e., L_{i+1}). However, for a gear compaction between L_i and L_{i+1}, the merge-sorted data is divided into three parts according to its key range, including: out of L_{i+2}'s compaction window, out of L_{i+2}'s key range, and within L_{i+2}'s compaction window. These three parts of the merged data do not stay in memory. Instead, they are respectively 1) written to L_{i+1}, 2) dumped to L_{i+2}, or 3) processed to passive compactions (i.e., compacted with overlapped SSTables in the CW of L_{i+2}). The dump operation (i.e., step 2) helps to reduce the further write amplification of writing the data to L_{i+1} and dumping it to L_{i+2}. To avoid data being compacted to the highest level directly, L_{i+2} can only join the gear compaction if L_{i+1} reaches its size limit and L_{i+2} reaches the size of its compaction window. As a result, GearDB maintains the temporal locality of LSM-trees, where newer data resides in lower levels.

Figure 8 illustrates the gear compaction process. **Step 1**, the active compaction is performed between L_0 and L_1,

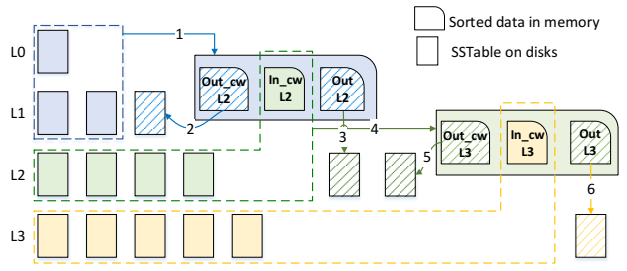


Figure 8: **Process of gear compaction.** *The active compaction of L_0 and L_1 drives passive compactions in higher level. The resultant data of each compaction is divided into three parts according to its key range, including out of L_i's compaction window (Out_cw L_i), in L_i's compaction window (In_cw L_i), and out of L_i's key range (Out L_i).*

and the resultant data in memory is divided into three parts according to their key range, i.e., out of L_2's compaction window, out of the next level, and within L_2's compaction window. **Step 2**, the data whose key range overlaps SSTables that is out of L_2's compaction window is written back to L_1. **Step 3**, the data whose key range does not overlap L_2's SSTables is dumped to L_2, avoiding further compaction and the associated write amplification. **Step 4**, the data whose key range overlaps with SSTables in L_2's compaction window remains in memory for further passive compaction with the overlapped SSTables in L_2' s compaction window. This gear compaction process proceeds recursively in compaction windows, level by level, until either the compaction reaches the highest level or the regenerated data does not overlap the compaction window of the next level. Thus, gear compaction only proceeds within compaction windows and therefore invalid SSTables only appear in compaction windows.

The gear compaction process is described in Algorithm 1. Lines 7-17 illustrate the key range division (detailed later in "**sorted data division**"). Active compaction starts from L_0 and L_1, and passive compaction continues level by level until the merge and sort results of L_i and L_{i+1} do not overlap L_{i+2}'s compaction window (Line 19 and 24). In addition, if the data volume written to L_{i+2} is less than the size of an SSTable (e.g., 4 MB), we write it back to L_{i+1} together with other data written to L_{i+1}. In this way, the size of each SSTable is kept at about 4 MB to ensure no small SSTable increases the overhead of metadata management.

Sorted data division. To divide the sorted data during gear compaction (e.g., L_i and L_{i+1}) into the above mentioned three categories, GearDB needs to compare the key range of the sorted data with the key ranges of SSTables in L_{i+2}. As in LevelDB, SSTables within a level do not overlap in GearDB. However, key ranges of some SSTables might not be successive. Key range gaps between SSTables complicate the division of the sorted data, and we need to compare the sorted datas keys with individual SSTables. Excessive com-

ALGORITHM 1: Gear Compaction Algorithm

Input: V_i: victim SSTable in L_i

```
do
    DoGearComp ← false;
    O_{i+1} ← GetOverlaps (V_i); /*O_{i+1}: overlapped SSTables in
    L_{i+1}'s compaction window*/
    result ← merge-sort(V_i, O_{i+1});
    iter.key ← MakeInputIterator(result);
    for iter.first to iter.end do
        if key In_CW L_{i+2} then
            write to buffer; /*wait in memory for the passive
            compaction*/
        else
            if key Out_CW L_{i+2} then
                write to L_{i+1};
            else
                if key Out L_{i+2} then
                    write to L_{i+2};
                end
            end
        end
    end
    if buffer != Null then
        i++;
        V_i ←GetVictims(buffer);
        DoGearComp ← true;
    end
while DoGearComp == true;
```

parisons can slow down the division and increase the cost of gear compaction and metadata management. To remedy this problem, we divide each level into large granularity key ranges. Specifically, for SSTables in CW and out of CW respectively, if the key range gap between SSTables does not overlap with other SSTables in that level, we combine the key ranges into a large consecutive range. As a result, the sorted data only needs to compare with the minimum and maximum keys of several key ranges to do the division. For example, suppose the compaction window of L_{i+2} has two SSTables with respective key ranges of $a-b$ and $c-d$. We check other SSTables in L_{i+2} to find if any SSTable overlaps the key range $b-c$. If not, we amend the key range of L_{i+2}'s compaction window as $a-d$ to reduce the key range comparison during division.

How compaction windows are reclaimed. As discussed above, gear compactions only proceed within compaction windows. Since a compaction window filled with invalid data can be simply released as empty zones, compaction windows are reclaimed automatically by gear compactions. As a result, redundant garbage collection that requires valid data migration is avoided. To invalidate all SSTables in the CW of L_{i+1}, the SSTables in L_i whose key ranges overlap with

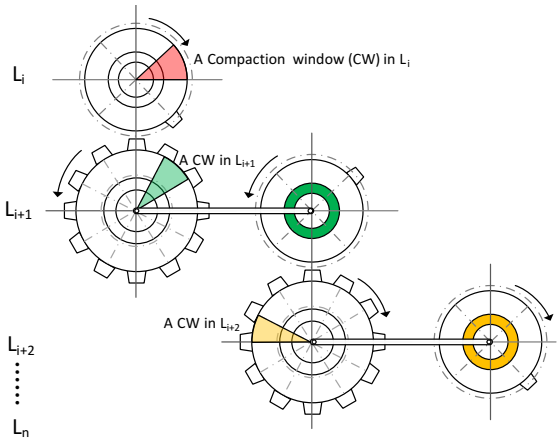


Figure 9: **Compaction windows are reclaimed in a gear fashion.** *The red, green, and yellow sectors represent the compaction windows of L_i, L_{i+1}, and L_{i+2}. Compaction windows are reclaimed by compaction like a group of gears. Reclaim k compaction windows (CW) in L_i mimics a full round moving of a gear, which leads to one move in the driven gear, that is cleaning one compaction window in L_{i+1}, and so on.*

L_{i+1}'s CW must be involved in gear compactions. Once all zones of L_i rotationally join the compaction window, we get these SSTables in L_i. In this fashion, when all compaction windows in L_i are reclaimed, the compaction window of L_{i+1} is reclaimed. As shown in Figure 9, when gear compactions clean k compaction windows in L_i, one compaction window in L_{i+1} is cleaned correspondingly; when gear compactions clean k compaction windows in L_{i+1}, one compaction window in L_{i+2} is cleaned; and so on. This process of releasing compaction windows works like a group of gears, where a complete rotation (i.e., k steps) in a driving gear (i.e., L_i) triggers one move in a driven gear (i.e., L_{i+1}), which gives our name as "gear compaction."

In summary, GearDB maintains the balance of LSM-trees by keeping the amplification factor of adjacent levels unchanged, keeping SSTables sorted and un-overlapped in each level, and rotating the compaction window at each level. The benefits of gear compaction include: 1) compactions and fragments are limited to the compaction window of each level; 2) compaction windows are reclaimed automatically during gear compactions, thereby eliminating the expensive on-disk garbage collections since compaction windows filled with invalid SSTables can be reused as free space; and 3) gear compaction compacts SSTables to a higher level with fewer reads and writes and no additional overhead.

4 Implementation

To verify the efficiency of GearDB's design strategies, we implement GearDB based on LevelDB 1.19 [11], a state-

of-art LSM-tree based KV store from Google. We use the libzbc [16] interface to access a 13 TB HM-SMR drive from Seagate. Libzbc allows applications to implement direct accesses to HM-SMR drives via T10/T13 ZBC/ZAC command set [20, 19], facilitating Linux application development.

As shown in Figure 6, GearDB maintains a standard interface for users, including GET/PUT/DELETE. The gear compaction is implemented on LSM-trees by modifying the conventional compaction processes of LevelDB. At the lowest level, we implement an HM-SMR controller to: 1) write sequentially in zones and manage per-zone write pointers using the new interface provided by libzbc; 2) map SSTables to dedicate zones and map zones to specific levels; and 3) manage the compaction window of each level. The mapping relationship is maintained by: 1) a Lbdfile structure denoting the indirection map of an SSTable and its zone location; 2) a zone_info structure recording all SSTables of a zone; and 3) a zone_info_list[L_i] structure containing all zones of Level L_i. Ldbfiles maintain the metadata of each SSTable. The size of the Ldbfile dominates the size of the metadata in the HM-SMR controller, and the other two structures only link Ldbfiles and zones with pointers (8 bytes). These data structures consume a negligible portion of memory, e.g., for an 80GB database, the overall metadata of the HM-SMR controller is less than 4 MB.

To keep metadata consistent, a conventional zone is allocated on HM-SMR drives to persist the metadata together with the version changes of the database after each compaction. In LevelDB, a manifest file is used to record the initial state of the database and the changes of each compaction. To recover from a system crash, the database starts from the initial state and replays the version changes. GearDB rebuilds the database in the same way.

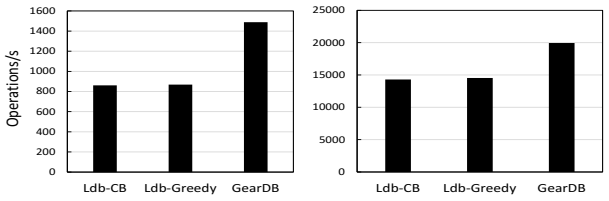
Other implementation details worth mentioning include: 1) for sequential write workloads that incur no compaction, GearDB dumps zones to the higher level by revising the zone_info_list[L_i] to avoid data migration. 2) To accelerate the compaction process in both GearDB and LevelDB, we fetch victim SSTables and overlapped SSTables into memory in the unit of SSTables instead of blocks. More details of the implementation can be found in our open source code with the link provided in Section 7.

5 Evaluation

GearDB is designed to deliver both high performance and space efficiency for building key-value stores on HM-SMR drives. In this section, we conduct extensive experiments to evaluate GearDB by focusing on answering the following questions: 1) what are the performance advantages of GearDB? (Section 5.1); 2) what factors contribute to these performance benefits? (Section 5.2); and 3) What space efficiency can GearDB achieve? (Section 5.3). In addition, we discuss the results of sensitivity studies of CW size, the performance of GearDB vs. SMRDB [38], and the performance of GearDB vs. LevelDB on HDDs(Section 5.4).

Table 1: System configuration for experiments

Linux	64-bit Linux 4.15.0-34-generic
CPU	8 * Intel(R) Core(TM) i7-6700 CPU @ 3.40GHz
Memory	32 GB
HM-SMR	13TB Seagate ST13125NM007 Random 4 KB request (IOPS): 163(R) Sequential (MB/s): 180(R), 178(W)



(a) Random load (b) Sequential load

Figure 10: **Load performance.** *GearDB shows its advantage in both random load and sequential load compared to LevelDB.*

We compare GearDB performance against LevelDB (version 1.19) [11] with greedy GC (Ldb-Greedy) and cost-benefit GC (Ldb-CB) policies. Our test environment is listed in Table 1. By default, we use 16-byte keys, 4 KB values, and 4 MB SSTables.

5.1 Performance Evaluation

In this section, we first evaluate the read and write performance of LevelDB and GearDB using the db_bench micro-benchmark released with LevelDB. Then, we evaluate performance using YCSB macro-benchmark suite [7].

5.1.1 Load Performance

We evaluate random load performance by inserting 20 million key-value items (i.e., 80 GB) in a uniformly distributed random order. Since the random load benchmark includes repetitive and deleted keys, the actual valid data volume of the database is around 54 GB. We restrict the capacity of our HM-SMR drive by using only the first 280 shingled zones (i.e., 70 GB). The final valid data takes up 77.14% of the usable disk space. The random write performance is shown in Figure 10 (a). GearDB outperforms Ldb-Greedy and Ldb-CB by 1.71× and 1.73× respectively. The two LevelDB solutions have lower throughput because of the time-consuming compaction and redundant GCs. Compaction in LevelDB produces write amplification and dispersed fragments on disk. Costly garbage collections clean disk space by migrating valid data, thus slowing down the random write performance. GearDB delivers better performance because

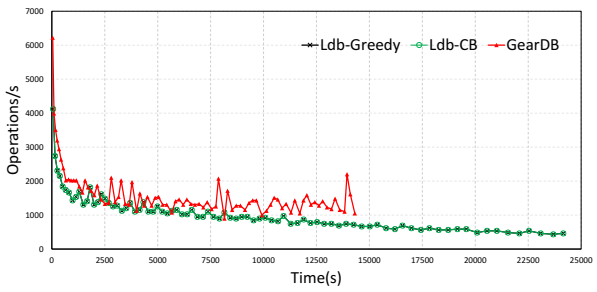


Figure 11: **Detail of random load.** *This figure shows the incremental performance of every 1 GB randomly loaded during the process of loading an 80 GB database. GearDB has higher throughput and a shorten execution time for loading the same sized database compared to LevelDB.*

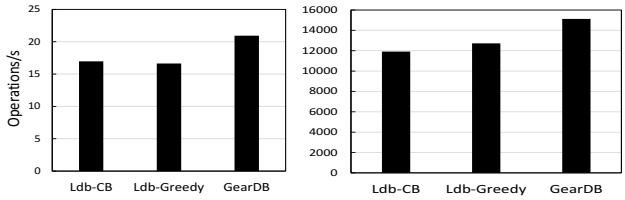
fragments are limited to compaction windows, garbage collections are eliminated by gear compactions, and compaction efficiency is improved. We further investigate detailed reasons for GearDB's performance improvements in Section 5.2.

Figure 11 shows the incremental throughput by recording the performance for every 1 GB of randomly loaded data (i.e., 250k KV entries). We make four observations from this figure. First, GearDB is faster than LevelDB for randomly loading the same volume of data. Second, GearDB achieves higher throughput than LevelDB, and the performance advantage becomes more pronounced as the volume of the database grows. Third, both LevelDB and GearDB's performance decrease with time, because the overhead of compaction and GCs (only LevelDB has GCs) increases with the data volume. Fourth, the performance variation of GearDB comes from the variation of the data volume in gear compactions. On the contrary, LevelDB shows less fluctuation in performance due to the relatively stable data volume involved in each compaction.

Similarly, we evaluate the sequential load performance by inserting 20 million KV items in sequential order. No compactions or garbage collections were incurred for sequential writes. Figure 10(b) shows that GearDB is 1.37× and 1.39× faster than Ldb-Greedy and Ldb-CB respectively. This performance gain is attributed to the more efficient dump strategy of GearDB as presented in Section 4. GearDB dumps SSTables to the next level by simply revising the metadata of zones.

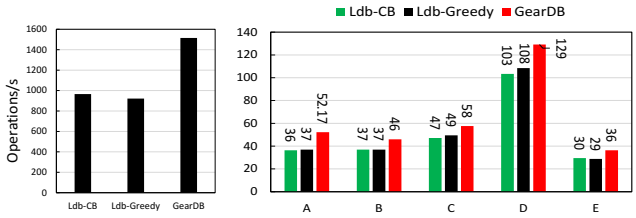
5.1.2 Read Performance

Read performance is evaluated by reading one million key-value items from the randomly loaded database. Figure 12 shows the results. The performance of sequential reads is much better than random reads due to the natural characteristics of disk drives. GearDB gets its performance advantage in both random and sequential reads because it consolidates



(a) Random read (b) Sequential read

Figure 12: **Read performance.**



(a) Workload Load (b) Workloads A-E

Figure 13: **Throughput on Macro-benchmarks.** *This figure shows the throughput of three key-value stores on load and other five workloads. In the left figure, the x-axis represents different workloads. The load workload corresponds to constructing an 80 GB database. Workload A is composed with 50% reads and 50% updates; Workload-B has 95% reads and 5% updates; Workload-C includes 100% reads; Workload-D has 95% reads and 5% latest keys insert; Workload-E has 95% range queries and 5% keys insert.*

SSTables of the same level. Our new data layout helps reduce the seek time of searching and locating SSTables by ensuring each zone stores SSTables from just one level.

5.1.3 Macro-benchmark

To evaluate performance with more realistic workloads, we run the YCSB benchmark [7] on GearDB and LevelDB. The YCSB benchmark is an industry standard macro-benchmark suite delivered by Yahoo!. Figure 13 shows the results of the macro-benchmark in load and five other representative workloads. GearDB is 1.56× and 1.64× faster than Ldb-CB and Ldb-Greedy on the load workload for the same reasons discussed in Section 5.1.1. Workloads A-E are evaluated based on the randomly loaded database. The performance gains of GearDB under workloads A-E are 1.44×, 1.24×, 1.22×, 1.25×, and 1.23× compared to LevelDB, which are consistent with the results of micro-benchmarks.

5.2 Performance Gain Analysis

In this section, we investigate GearDB's performance improvement when compared to LevelDB.

Operation Time Breakdown. To figure out the advan-

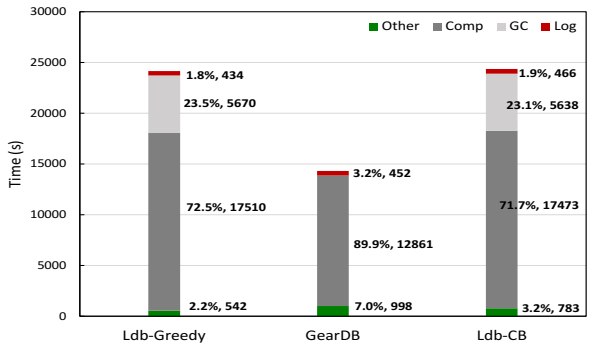


Figure 14: **Random load time breakdown.** *This figure shows the time spent (the y-axis) on different operations when we randomly load an 80 GB database. The numbers next to each bar show their time consumption and the ratio to the overall run time. For LevelDB, compaction and garbage collections take the most significant percentage of the overall runtime. GearDB eliminates the garbage collections and improves compaction efficiency.*

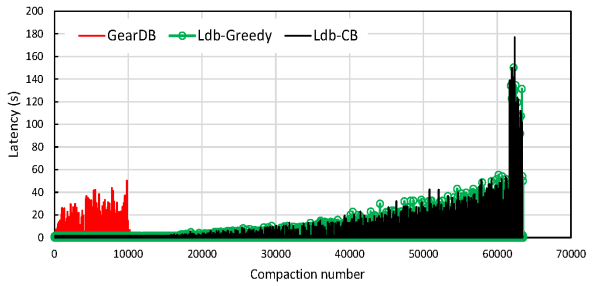


Figure 15: **Compaction analysis.** *This figure shows the latency of every individual compaction during the 80 GB random load.*

tages and disadvantages of GearDB and LevelDB, we break down the time of all KV store operations (i.e., log, compaction, garbage collection, and other write operations) for the random load process. As shown in Figure 14, we observe that compared to Ldb-Greedy and Ldb-CB: 1) GearDB adds no overhead to any operations; and 2) GearDB's performance advantage mainly comes from the more efficient compaction and eliminated garbage collection. LevelDB has a longer random load time because garbage collections take about a quarter of the overall runtime and the compaction is less efficient than GearDB. We record the detailed information of garbage collections for Ldb-CB and Ldb-Greedy as follows: 1) the overall garbage collection time is 5,638 s and 5,670 s, which account for 23.14% and 23.47% of the overall random load time (24,360 s and 24,156 s); and 2) the migration data volume in garbage collection is 417 GB and 430 GB, which is 25.53% and 25.77% of the overall disk writes.

Compaction Efficiency. To understand the compaction efficiency of the three key-value stores specifically, we

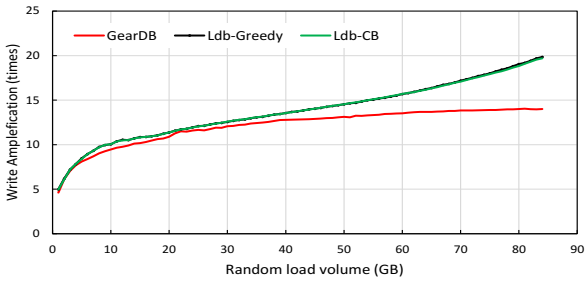


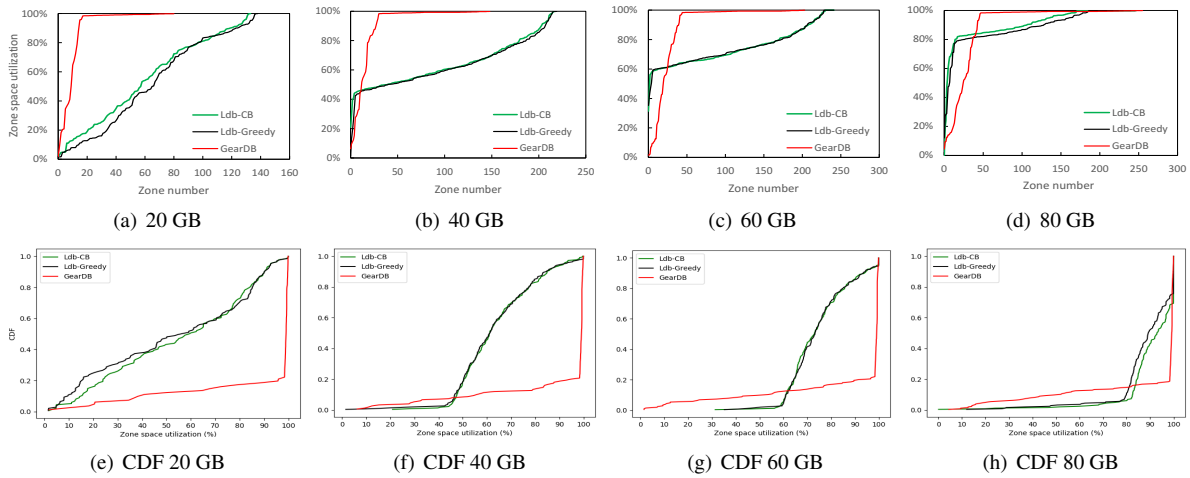
Figure 16: **Write amplification.** *This figure shows the write amplification factor of three KV systems when we load different sizes of the database (i.e., from 1 GB to 80 GB).*

record the compaction latency for every compaction during the random loading process, which is shown in Figure 15. From this figure, we make the following three observations. First, GearDB dramatically reduces the number of compactions (i.e., 53,311 and 53,203 less than Ldb-Greedy and Ldb-CB respectively). Since GearDB continues gear compaction to higher levels when key ranges overlap with the compaction window of the adjacent level, more data are compacted in one compaction process, reducing the number of compactions. Second, the average compaction latency of GearDB is higher than LevelDB because gear compaction involves more data in each compaction. Third, the overall compaction latency is 1.80× shorter in GearDB than LevelDB with greedy or cost-benefit strategies.

Write Amplification. Write amplification (WA) is an important factor in the performance of key-value stores. We calculate the write amplification factor by dividing the overall disk-write volume by the total volume of user data written. The WA of the three systems is shown in Figure 16. In both LevelDB and GearDB, the WA increases with the volume of the database as the compaction data volume increases. Ldb-Greedy and Ldb-CB have a similar large write amplification because they need to migrate data in both compactions and garbage collections. GearDB reduces the write amplification since it performs no on-disk garbage collections.

5.3 Space Efficiency Evaluation

Figure 17 shows a comparison of zone usage and zone space utilization after randomly loading 20, 40, 60, and 80 GB databases. From these results, we find GearDB occupies fewer zones than LevelDB after loading the same size database. For example, GearDB saves 71 zones (i.e., 17.75 GB) when storing a 40 GB database. Moreover, GearDB has higher zone space utilization than LevelDB, i.e., GearDB's average space utilization of loading the 80 GB database is 89.9%. We show the corresponding CDFs of zone space utilization in Figure 17. These results show that GearDB restricts fragments in a small portion of occupied zones (i.e.,



(a) 20 GB (b) 40 GB (c) 60 GB (d) 80 GB

(e) CDF 20 GB (f) CDF 40 GB (g) CDF 60 GB (h) CDF 80 GB

Figure 17: **Zone space utilization.** *Figures a-d show the zone space utilization of each occupied zone when we randomly load 20, 40, 60, and 80 GB database. Figures e-h show the corresponding CDF of the zone space utilization. The results show that our design consistently maintains a high space efficiency.*

compaction windows). GearDB achieves a bimodal zone space utilization, where most zones are nearly full, and a few zones are nearly empty since they are in compaction windows. This bimodal distribution not only improves space utilization but also wipes out garbage collections, since an empty zone can be reused by resetting write pointers without incurring data migration. LevelDB suffers from low space utilization, especially for smaller databases, since fewer GCs are triggered when the database is small. Once more garbage collections are triggered, LevelDB's space efficiency improves at the cost of system performance. However, GearDB achieves a high space utilization during its lifetime without sacrificing system performance. The overall performance and space efficiency gain of GearDB is denoted by the red triangle mark in Figure 5.

5.4 Extended Evaluations

Sensitivity Study of the Compaction Window Size. We evaluate GearDB with 5 different compaction window sizes by changing the k in Equation 2 (i.e., k=2, 4, 6, 8, 10). The experimental setup is the same as used for the micro-benchmarks. Consistently, GearDB maintains a random load throughput ranging from 1,314 to 1,470 operations/s, and a space utilization ranging from 86% to 91%. Since the compaction window size does not have a significant influence on system performance and space efficiency, we set $k = 4$ as the default CW size for GearDB.

GearDB vs. SMRDB. SMRDB [38] is an HM-SMR friendly KV store, which enlarges the SSTable to a zone size (e.g., 256 MB) to prevent overwriting and reduces the number of levels to two to reduce compactions. We implement and then evaluate SMRDB using db_bench. Test results show that SMRDB is slower than GearDB by 1.97× for random loading. SMRDB brings severe compaction latency due to the large data volume involved in each compaction. GearDB has similar sequential read performance, and 1.68× faster random read performance compared to SMRDB since the large SSTable increases the overhead of fetching KV items in SSTables.

GearDB on HM-SMR vs. LevelDB on HDD. To further demonstrate the potential of GearDB, we compare GearDB on HM-SMR to LevelDB on a Seagate hard disk drive (ST1000DM003). The original LevelDB uses the standard file system interface (i.e., Ext4 in our evaluation), and we call it Ldb-hdd. The basic performance evaluation on db_bench shows that GearDB on HM-SMR outperforms Ldb-hdd by 2.38× for randomly loading an 80 GB database. GearDB has higher sequential write performance and similar random read performance with Ldb-hdd. However, Ldb-hdd has the superiority on sequential read (i.e., 7.02× faster) due to the file system cache. Our GearDB bypasses the file system and thus does not benefit from the cache.

6 Related Work

GearDB is an LSM-tree based key-value store tailored for HM-SMR drives, which aims to realize both good performance and high space utilization. We first discuss existing works that exploit HM-SMR drives without compromising system performance. ZEA provides HM-SMR software abstractions that map zone block addresses to the logical block addresses of HDDs [33]. SMRDB [38] is an HM-SMR friendly KV store described and evaluated in Section 5.4. Caveat-Scriptor [23] and SEALDB [47] allow to write anywhere on HM-SMR drives by letting the host write beware.

Kinetic [41] provides KV Ethernet HM-SMR drives plus an open API to support object storage. Huawei's key-value store (KVS) [31] provides simple and redundant KV accesses on HM-SMR drives via a core design of a log-structured database. SMORE [32] is an object store on an array of HM-SMR drives, which also accesses disks in a log-structured approach. HiSMRfs [22] stores file metadata on SSDs and stores file data on SMR drives.

Next, we discuss research to enhance and improve LSM-trees by reducing the write amplification caused by compactions. Lwc-tree [48] performs lightweight compactions by appending data to SSTables and only merging metadata. PebblesDB [39] mitigates writes by using guards to maintain partially sorted levels. WiscKey [29] separates keys from values and compacts keys only, thus reducing compaction overhead by eliminating value migrations. VTtrees [44] use an extra layer of indirection to avoid reprocessing sorted data, at the cost of fragmentation. TRIAD [34] uses a holistic of three technologies on memory, disk, and log to reduce write amplification. Blsm [43] proposes a new merge scheduler to synchronize merge completions, and hence obviates upstream writes from waiting downstream merges. LSM-trie [46] de-amortizes compaction overhead with hash-range based compaction for better read performance. [25] and [24, 13] optimize LSM-trees tailored for specific storage devices and specific application scenarios. In contrast, GearDB improves system performance via providing a new data layout that facilitates the data fetching in compaction and eliminating write amplification from redundant GC.

Third, recent works have sought to optimize or manage SSDs at the application layer [30, 14, 35]. They aim to solve the double logging problem in both FTLs and append-only applications via new block I/O interfaces [30] or application-driven FTLs [14]. However, there still exists the need to employ GC policies for reclaiming flash segments in FTLs. By contrast, GearDB eliminates the overhead of disk space cleaning via three design strategies.

Finally, SSD streams [6, 26] associate data with similar update frequencies or lifetimes to the same stream and place it into the same unit for multi-stream SSD. The data layout of GearDB shares the initial consideration of separating data with similar lifetimes. However, the methodology is different entirely, e.g., [6] assigns write requests of multiple levels to dedicated streams.

7 Conclusion

In this paper, we present GearDB, an LSM-tree based key-value store tailored for HM-SMR drives. GearDB is designed to achieve both good performance and high space utilization with three techniques: a new data layout, compaction windows, and a novel gear compaction algorithm. We implement GearDB on a real HM-SMR drive. Experimental results show that GearDB improves the overall system performance and space utilization, i.e., $1.71\times$ faster than LevelDB in random write with a space efficiency of 89.9%. GearDB's performance gains mainly come from efficient gear compaction by eliminating garbage collections. The open source GearDB is available at *https://github.com/PDS-Lab/GearDB*.

8 Acknowledgement

We thank our shepherd Bill Jannen and the anonymous reviewers for their insightful comments and feedback. We also thank Seagate Technology LLC for providing the sample HM-SMR drive to run our experiments. 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 111 Project (No.B07038); the Director Fund of WNLO. The work performed at Temple was partially supported by the U.S. National Science Foundation grants CCF-1717660 and CNS-1702474.

References

[1] AGHAYEV, A., AND DESNOYERS, P. Skylight—a window on shingled disk operation. In *13th USENIX Conference on File and Storage Technologies (FAST 15)* (2015), pp. 135–149.

[2] AGRAWAL, N., PRABHAKARAN, V., WOBBER, T., DAVIS, J. D., MANASSE, M. S., AND PANIGRAHY, R. Design tradeoffs for SSD performance. In *USENIX Annual Technical Conference* (2008).

[3] AMER, A., LONG, D. D. E., MILLER, E. L., PARIS, J.-F., AND SCHWARZ, S. J. T. Design issues for a shingled write disk system. In *Proceedings of the 2010 IEEE 26th Symposium on MSST* (2010).

[4] CASSUTO, Y., SANVIDO, M. A. A., GUYOT, C., HALL, D. R., AND BANDIC, Z. Z. Indirection systems for shingled-recording disk drives. In *Proceedings of the 2010 IEEE 26th Symposium on MSST* (2010), pp. 1–14.

[5] CHANG, F., DEAN, J., GHEMAWAT, S., HSIEH, W. C., WALLACH, D. A., BURROWS, M., CHANDRA, T., FIKES, A., AND GRUBER, R. 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.

[6] CHOI, C. Increasing ssd performance and lifetime with multi-stream technology. `https://www.snia.org/sites/default/files/DSI/2016/presentations/sec/ChanghoChoi_Increasing_SSD_Performance-rev.pdf`, 2016.

[7] COOPER, B. F., SILBERSTEIN, A., TAM, E., RAMAKRISHNAN, R., AND SEARS, R. Benchmarking cloud serving systems with YCSB. In *Proceedings of the ACM Symposium on Cloud Computing (SOCC'10)* (2010).

[8] DAYAN, N., ATHANASSOULIS, M., AND IDREOS, S. Monkey: Optimal navigable key-value store. In *Proceedings of the 2017 ACM International Conference on Management of Data* (2017), ACM, pp. 79–94.

[9] FACEBOOK. RocksDB, a persistent key-value store for fast storage enviroments. `http://rocksdb.org/`.

[10] FELDMAN, T., AND GIBSON, G. Shingled magnetic recording: Areal density increase requires new data management. *USENIX; login: Magazine 38*, 3 (2013), 22–30.

[11] GHEMAWAT, S., AND DEAN, J. Leveldb. https://github.com/Level/leveldown/issues/298, 2016.

[12] GIBSON, G., AND GANGER, G. Principles of operation for shingled disk devices. *Carnegie Mellon University Parallel Data Lab Technical Report CMU-PDL-11-107* (2011).

[13] GOLAN-GUETA, G., BORTNIKOV, E., HILLEL, E., AND KEIDAR, I. Scaling concurrent log-structured data stores. In *Proceedings of the Tenth European Conference on Computer Systems (EuroSys)* (2015).

[14] GONZÁLEZ, J., BJØRLING, M., LEE, S., DONG, C., AND HUANG, Y. R. Application-driven flash translation layers on open-channel ssds. In *Nonvolatile Memory Workshop (NVMW)* (2014).

[15] HGST. Hgst delivers world's first 10tb enterprise hdd for active archive applications. http://investor.wdc.com/news-releases/news-release-details/hgst-delivers-worlds-first-10tb-enterprise-hdd-active-archive, 2015.

[16] HGST. Libzbc version 5.4.1. https://github.com/hgst/libzbc, 2017.

[17] HGST. Ultrastar Hs14 — 14tb 3.5 inch helium platform enterprise smr hard drive. https://www.hgst.com/products/hard-drives/ultrastar-hs14, 2017.

[18] HGST. Ultrastar dc hc600 smr series, 15TB. https://www.westerndigital.com/products/data-center-drives/ultrastar-dc-hc600-series-hdd, 2018.

[19] INCITS T10 TECHNICAL COMMITTEE. Information technology-zoned block commands (ZBC). draft standard t10/bsr INCITS 550, american national standards institute, inc. http://www.t10.org/drafts.htm, 2017.

[20] INCITS T13 TECHNICAL COMMITTEE. Zoned-device ata command set (ZAC) working draft.

[21] JAGADISH, H., NARAYAN, P., SESHADRI, S., SUDARSHAN, S., AND KANNEGANTI, R. Incremental organization for data recording and warehousing. In *VLDB* (1997), vol. 97, pp. 16–25.

[22] JIN, C., XI, W.-Y., CHING, Z.-Y., HUO, F., AND LIM, C.-T. HiSM-Rfs: A high performance file system for shingled storage array. In *Proceedings of the 2014 IEEE 30th Symposium on MSST* (2014), IEEE, pp. 1–6.

[23] KADEKODI, S., PIMPALE, S., AND GIBSON, G. A. Caveat-Scriptor: Write anywhere shingled disks. In *7th USENIX Workshop on HotStorage* (2015).

[24] KAI, R., QING, Z., JOY, A., AND GARTH, G. SlimDB—a space-efficient key-value storage engine for semi-sorted data. *Proceedings of the VLDB Endowment 10*, 13 (2017).

[25] KANNAN, S., BHAT, N., GAVRILOVSKA, A., ARPACI-DUSSEAU, A., AND ARPACI-DUSSEAU, R. Redesigning lsms for nonvolatile memory with novelsm. In *USENIX Annual Technical Conference* (2018), pp. 993–1005.

[26] KIM, T., HAHN, S. S., LEE, S., HWANG, J., LEE, J., AND KIM, J. Pcstream: Automatic stream allocation using program contexts. In *10th USENIX Workshop on HotStorage* (2018).

[27] KU, S. C.-Y., AND MORGAN, S. P. An smr-aware append-only file system. In *Storage Developer Conference* (2015).

[28] LAKSHMAN, A., AND MALIK, P. Cassandra: A decentralized structured storage system. In *The 3rd ACM SIGOPS International Workshop on Large Scale Distributed Systems and Middleware* (2009).

[29] LANYUE, L., SANKARANARAYANA, P. T., C, A.-D. A., AND H, A.-D. R. WiscKey: separating keys from values in ssd-conscious storage. In *14th USENIX Conference on File and Storage Technologies (FAST 16)* (2016), pp. 133–148.

[30] LEE, S., LIU, M., JUN, S. W., XU, S., KIM, J., AND ARVIND, A. Application-managed flash. In *14th USENIX Conference on File and Storage Technologies (FAST 16)* (2016), pp. 339–353.

[31] LUO, Q., AND ZHANG, L. Implement object storage with smr based key-value store. In *Storage Developer Conference* (2015).

[32] MACKO, P., GE, X., KELLEY, J., SLIK, D., ET AL. SMORE: A cold data object store for smr drives. In *Proceedings of the 2017 IEEE 33th Symposium on MSST* (2017).

[33] MANZANARES, A., WATKINS, N., GUYOT, C., LEMOAL, D., MALTZAHN, C., AND BANDIC, Z. ZEA, a data management approach for smr. In *8th USENIX Workshop on HotStorage* (2016).

[34] MARIA, B. O., DIEGO, D., RACHID, G., WILLY, Z., HUAPENG, Y., AASHRAY, A., KARAN, G., AND PAVAN, K. TRIAD: creating synergies between memory, disk and log in log structured key-value stores. In *USENIX Annual Technical Conference* (2017).

[35] MARMOL, L., SUNDARARAMAN, S., TALAGALA, N., RANGASWAMI, R., DEVENDRAPPA, S., RAMSUNDAR, B., AND GANESAN, S. NVMKV: A scalable and lightweight flash aware key-value store. In *6th USENIX Workshop on HotStorage* (2014).

[36] MARTIN, M., TIM, H., KRSTE, A., AND JOHN, K. Trash day: Coordinating garbage collection in distributed systems. In *HotOS* (2015).

[37] ONEIL, P., CHENG, E., GAWLICK, D., AND ONEIL, E. The log-structured merge-tree (lsm-tree). *Acta Informatica 33*, 4 (1996), 351–385.

[38] PITCHUMANI, R., HUGHES, J., AND MILLER, E. L. SMRDB: key-value data store for shingled magnetic recording disks. In *Proceedings of the 8th ACM International Systems and Storage Conference* (2015).

[39] RAJU, P., KADEKODI, R., CHIDAMBARAM, V., AND ABRAHAM, I. Pebblesdb: Building key-value stores using fragmented log-structured merge trees. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP)* (2017), ACM, pp. 497–514.

[40] ROSENBLUM, M., AND OUSTERHOUT, J. K. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems (TOCS) 10*, 1 (1992), 26–52.

[41] SEAGATE. The seagate kinetic open storage vision. https://www.seagate.com/tech-insights/kinetic-vision-how-seagate-new-developer-tools-meets-the-needs-of-cloud-storage-platforms-master-ti/.

[42] SEAGATE. Archive hdds from seagate. http://www.seagate.com/www-content/product-content/hdd-fam/seagate-archive-hdd/en-us/docs/100757960a.pdf, 2014.

[43] SEARS, R., AND RAMAKRISHNAN, R. bLSM: A general purpose log structured merge tree. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data (SIGMOD 12)* (2012).

[44] SHETTY, P., SPILLANE, R. P., MALPANI, R., ANDREWS, B., SEYSTER, J., AND ZADOK, E. Building workload-independent storage with VT-trees. In *11th USENIX Conference on File and Storage Technologies (FAST 13)* (2013), pp. 17–30.

[45] WU, F., YANG, M.-C., FAN, Z., ZHANG, B., GE, X., AND H.C.DU, D. Evaluating host aware smr drives. In *8th USENIX Workshop on HotStorage* (2016).

[46] WU, X., XU, Y., SHAO, Z., AND JIANG, S. LSM-trie: An lsm-tree-based ultra-large key- value store for small data. In *USENIX Annual Technical Conference* (2015).

[47] YAO, T., TAN, Z., WAN, J., HUANG, P., ZHANG, Y., XIE, C., AND HE, X. A set-aware key-value store on shingled magnetic recording drives with dynamic band. In *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)* (2018), IEEE, pp. 306–315.

[48] YAO, T., WAN, J., HUANG, P., HE, X., GUI, Q., WU, F., AND XIE, C. 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).