# SMRDB: Key-Value Data Store for Shingled Magnetic Recording Disks

Rekha Pitchumani
UC Santa Cruz
rekhap@soe.ucsc.edu

James Hughes
Seagate Technology
james.hughes@seagate.com

Ethan L. Miller
UC Santa Cruz
elm@soe.ucsc.edu

## Abstract

Shingled Magnetic Recording (SMR) disks employ a shingled write process that overlaps the data tracks on the disk surface like the shingles on a roof, thereby increasing disk areal density with minimal manufacturing changes. While these disks have the same read behavior as current disks, random writes and in-place data updates are no longer possible, since a write to a track must overwrite and destroy data on all tracks that it overlaps.

Given this change in write behavior, we argue that the best way to utilize these disks is not by masquerading them as traditional disks, but by using approaches that leverage their proclivity for sequential writes. To address this need, we developed SMRDB, a key-value data store for SMR disks, demonstrating that SMR disks can be effectively used to replace conventional disks for many applications. We evaluate SMRDB against a state-of-the-art LSM-tree based key-value database engine, LevelDB, on conventional disks. Our Yahoo! Cloud Serving Benchmark results show that, despite being restricted to sequential writes, SMRDB *outperforms* LevelDB by 8.8–123.6%.

***Categories and Subject Descriptors*** D.4.2 [*Operating Systems*]: Storage Management

***Keywords*** Shingled Magnetic Recording, Key-Value Storage Systems

## 1. Introduction

Hard disk drives have played a major role in creating today's data driven world by making digital storage cheap. The drive's areal density (the number of bits stored per unit area) growth has always been hindered by limits imposed by the laws of physics, but has steadily increased due to the introduction of new recording technologies. The currently employed Perpendicular Magnetic Recording is about to reach its density limit and the industry is eager for the introduction of new technology to overcome the limit.

Shingled Magnetic Recording (SMR) (Greaves et al. 2009) (SMR-8TB-HDD 2014) is leading next generation disk technology. SMR requires minimal manufacturing changes because it retains the use of existing disk head and media technologies, achieving its areal density gain by overlapping tracks on one another like shingles on a roof. Traditionally, user data stored on disks has been managed by block-based file systems and databases with an underlying assumption that these blocks are independently updatable units. However, because overlapping tracks result in destructive random writes and in-place updates, SMR disks demand new data management solutions.

SMR access restrictions can be handled in the drive, in the host, or co-operatively by both the host and the drive (Feldman and Gibson 2013). Standardization efforts to broadly categorize these disks into Autonomous (drive managed) and Host-Managed drives (Campello 2013), and to bring about new standard commands sets assisting the different categories are underway. In recent years, data utilization demands have driven a storage shift towards distributed, highly scalable NoSQL data stores. This shift has encouraged a new generation of disks, Ethernet key-value drives (Seagate-Kinetic 2014), presenting another suitable SMR drive interface contender that hasn't been explored yet.

We believe that SMR drives can be utilized to their fullest by accepting the sequential write nature of these drives and adapting to it, rather than masquerading as traditional disks. To demonstrate that SMR disks can fulfill modern storage needs and that the SMR capacity gain doesn't have to come with lower performance, we designed SMRDB, a Key-Value (KV) database engine for SMR disks. SMRDB is designed to run directly on top of a host-managed SMR disk exposing the SMR functionality and does not depend on a file system, eliminating the need for a block-level drive managed SMR solution or a new SMR aware file system.



*SYSTOR 2015*, May 26–28, 2015, Haifa, Israel.

ACM 978-1-4503-3607-9/15/05…$15.00.
http://dx.doi.org/10.1145/2757667.2757680

Both the industry and the research community have come up with a plethora of applications that use the KV data access model. Many of the data storage and management systems in the cloud have adopted the KV access model (Chang et al. 2006; DeCandia et al. 2007). A disk running SMRDB, being an embeddable database engine, can easily be plugged into these systems, thus replacing traditional disks with the new SMR disks with ease. Log-Structured Merge (LSM) tree (O'Neil et al. 1996) based KV stores have also been demonstrated to work better than traditional file system techniques for storing and managing file system metadata and small files (Stender et al. 2010; Ren and Gibson 2012), and an entire user-level file system (Shetty et al. 2013). Thus, our solution enables easy adoption of SMR disks in a wide range of applications.

SMRDB manages the underlying disk in a SMR friendly manner, without the need for a filesystem, using raw sector-level primitives and sequential writes. SMRDB stores its data in sequential disk regions and periodically merges the KVs in selected regions to free the dead space and reorder the KVs in those regions. SMRDB is designed to work with raw SMR disks without any drive level block remappers, but such a device is not yet available in the market for evaluation. However, since SMR only alters the track layout and rules about track overwrite, but is otherwise similar to existing Perpendicular Magnetic Recording technology, performance measurements from traditional disk used with sequential write restrictions suffice (Pitchumani et al. 2012). Therefore, we evaluate SMRDB using a traditional disk written like a SMR disk. Our evaluation compares SMRDB against LevelDB, state-of-the-art LSM-tree based embeddable database engine and shows that SMRDB outperforms LevelDB in most cases. Our work is the first to adapt an LSM tree based data layout scheme for SMR disks, and demonstrates that the scheme results in both good write performance, and good read performance, including range reads.

## 2. Shingled Magnetic Recording

Hard disks have a magnetic recording medium that is organized into groups of concentric circular tracks that are laid out with a guard gap between them. A magnetic recording head containing a separate write head element for writing and a read head element for reading is positioned above the recording medium. A successful write requires a higher magnetic field from the write element and the writing pole has to be big enough to obtain the required write field strength. The number of tracks that can be packed per inch without any overlap has thus been limited by the width of the write head.

Shingled writing takes advantage of the fact that the magnetic field required for a read is smaller than that required for a write. Hence, the track width required for reading can be smaller than that required for writing. A data track is written by partially overlapping the preceding track, with enough room to read the preceding track's data by a narrower read head. As shown in Figure 1, the result of the process is narrow read tracks and wide write tracks that overwrite $k$ (the value of which is decided by the manufacturer; different disks may have different values) such read tracks. Shingled writing achieves a higher areal density by packing more tracks in the same area than traditional disks.

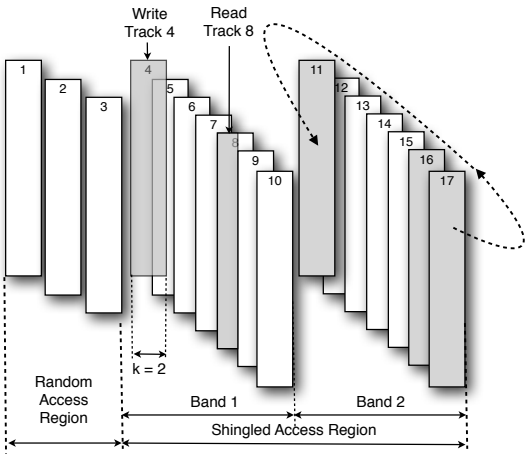



**Figure 1.** Shingled writing overwrites $k$ read tracks. SMR disks can be divided into many append-only bands and a small random-access region, if desired.

### 2.1 Shingled Bands

The SMR disk can be divided into *bands*, sequentially writable region made of consecutive tracks, to bound the region that needs to be written sequentially. As Figure 1 illustrates, fixed banding could be provided at the time of manufacture, by not shingling the first tracks of a band, here tracks 4 and 11. A band can also be formed by sacrificing the space of $k - 1$ (the number of tracks whose data gets destroyed by a write to the last track in the band) tracks to serve as a guard band between two neighbor bands. For example, in Figure 1, band 1 could be split into 2 bands (4-6 and 8-10) by using 1 track (7) as guard space.

A small random-access region might be available either from the disk manufacturers or by creating one from the shingled region with the help of guard tracks for every track. The rest of the disk contains multiple shingled bands, each of which can be thought of as containing a sequentially writable log, either a fixed log or a circular log (Amer et al. 2011). If a band is used as a fixed log, the band is allocated and deallocated as one large unit and is written to sequentially from the physical start to physical end. But if used as a circular log, writes wrap around the physical end of the band and restart from the physical start of the band as shown using band 2 in Figure 1; while data is written to from one end of the circular log, data could be freed from the other end. If used as circular bands it will cost an additional $k - 1$ tracks per band to serve as a non-stationary intra-band gap between

the head and tail of the circular band. The challenge lies in finding a way to efficiently store, retrieve and manage the data in these large sequentially writable bands without wasting a lot of space.

### 2.2 Log Structured Data Management

SMR sequential write restrictions call for log-structured writes to shingled bands. The Log-structured File System (LFS) (Rosenblum and Ousterhout 1992) organizes the disk into a segmented, append-only log and batches writes to the end of the log, but even LFS writes data sequentially only to a small unit called the *segment* and over time the segments available for writing get scattered randomly all over the disk. The LFS layout could be adapted to SMR disks by increasing the segment size to match the SMR band size. Researchers have pointed out that read performance will be affected in LFS, and have suggested data reorganization in the background when data is being moved by the garbage collector to improve the read performance (Matthews et al. 1997). Nevertheless, scan performance can be expected to be poor with a traditional LFS layout, without more aggressive data movement.

The data also needs to be indexed for efficient retrieval. Traditionally, B-trees have been the index structure of choice for disk-based systems, and would serve if stored in random-access bands. Depending on the size of the keys and values being stored, the index can grow very large and it will not be feasible to store the index in the random-access region in entirety. If stored in shingled bands, the leaf node modifications will force modifications to all nodes till the root, polluting the log with more dead data. Many copy-on-write trees have been designed for NAND flash, all aiming to reduce the number of nodes to be written on a leaf update. But the problem of their on-disk placement in large shingled bands, and their cleaning still remains.

Log Structured Merge trees (O'Neil et al. 1996) offer an alternative to LFS layout. Systems that require both the write performance of logging systems and demand a decent range read (read all keys falling within a given range) performance typically adopt the LSM tree based approach. An LSM-tree contains multiple ordered log-structured indexes, one in the memory and the others on disk. When any index exceeds a per-determined size threshold, parts of it are merged with the index in the next level. LSM-trees perform all disk writes in a log-structured manner, but sacrifice some of the write performance for additional merge operations, to offer good range read performance, and do not need a separate index management as required by the LFS layout.

## 3. LSM-Tree Based SMR Solution

To meet our scan performance goals, we will need aggressive data reorganization, as being done in LSM-trees. In this section, we present a simple LSM-tree based solution for SMR disks, based on an open source LSM tree based, embeddable KV database library, LevelDB (Ghemawat and Dean 2015).

LevelDB is a user-level library that uses a filesystem to read and write the key-value pairs to the underlying disk. It follows the same design as the BigTable (Chang et al. 2006) tablet. Every key-value pair is first written to a log file, and then added to an in-memory memtable. The memtable keeps its contents sorted, and when full, writes them to the disk as an SSTable (sorted string table) file. An additional metadata file stores the list of files and high level information about the files. Our simple solution is to maintain a memtable of size equivalent to an SMR band, and place each SSTable in a SMR band, as shown in the Figure 2.

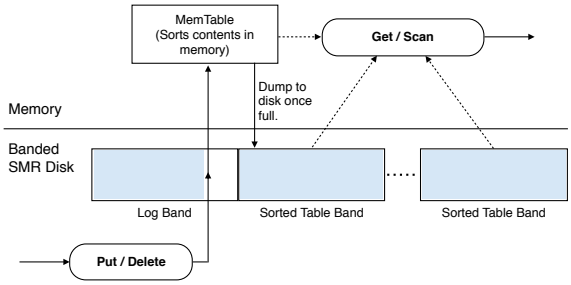



**Figure 2.** LevelDB based data access management.

***Data Organization*** The SSTables are organized into multiple levels as shown in Figure 3. The level 0 (L0) SSTables are the results of *memtable* dumps, and the key range covered by each L0 SSTable can overlap with each other. Each level has a size threshold and is increasingly bigger than the previous level. The SSTables in the other levels have a non-overlapping key range with respect to the other SSTables in the same level. Thus, when a key has to be read, the maximum number of SSTables that need to be searched is the number of L0 SSTables + the number of non-zero levels. The defaults in LevelDB are 7 levels, 4 MB *memtable* (all level 0 files are thus 4 MB long), and each file in a non-zero level is 2 MB long, whereas all SSTables will be as big as the SMR band in our solution.

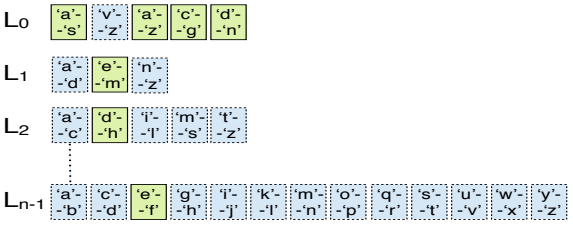



**Figure 3.** LevelDB style multi-level organization. A key starting with 'e' could be in any one of the 7 bands marked (in green) with a line border.

***Compaction*** Periodically, selected SSTables in levels $i$ and $i+1$ are *merged/compacted* to form new SSTables in level $i+1$, to rearrange the KV pairs in lexicographic order and to

free up the space used by dead (deleted/updated) KV pairs. If the total size of all the SSTables in a level exceed the size threshold for the level, the level is chosen and an SSTable in the chosen level is selected in a round robin manner.

***Problems*** When compactions are in progress, the increased read and write amplification, caused by I/O in the background, affects the incoming read/write performance. The scheme's compaction costs are larger for larger SSTables, as the SSTables chosen for compaction are read, reordered and rewritten entirely. When an SSTable is chosen to be compacted, all the SSTables in the selected level and those in the next level, whose key ranges overlap the selected SSTable's key range, are compacted. The strict non-overlap key range requirement reduces the number of SSTables that needs to be searched during a read, but increases the number of SSTables that get selected for compaction in the next level. Most of the time, this can result in copying SSTables without any major reordering of their contents. For example, if an SSTable selected for compaction in a level covered the key range $a - k$, but had only 1 key in the range $d - j$, all bands in the next level that overlap the range $a - k$, including the band with the $d - j$ range would be selected for compaction and read and rewritten, though only one key was inserted to the $d - j$ range.

Recently, *stitching* has been proposed to solve this problem (Shetty et al. 2013). Parts of the level's components were copied out to new locations and parts which would not be affected by the merge were not actually copied, but instead logically stitched with the portions that were newly written. They showed that an increased random insert performance could be achieved by sacrificing scan performance, and preference for either could be given using a stitching threshold. But they chose to do offline cleaning of invalidated data, and their results did not include online space reclamation of the portions that were invalidated. Hence, it is not clear how effective the scheme would be when the background compactions are done along with background cleaning for space reclamation and reuse.

LevelDB's multi-level organization also pushes older data down, decreasing the amount of data that gets selected for a compaction run at any given level. But if a new version of data that is currently in a lower level is inserted, it has to be copied multiple times and has to travel down each level through multiple compaction runs to finally free up the dead space. Though the amount of data that the initial compaction runs have to read and write might be lowered, it ultimately increases the the amount of data reads and writes that is required to keep them all ordered.

# 4. SMRDB

SMRDB is a variable-length KV database engine for SMR disks, which strives to keep the KV pairs on the disk physically ordered by lexicographical key order. While most embeddable database engines depend on an underlying local filesystem to manage the disk, SMRDB is a filesystem free, direct-on-disk solution that manages the underlying disk in an SMR-friendly manner. SMRDB is backward compatible with traditional hard disks, in that it will work and enable high performance on traditional disks, as it would on a host-managed SMR disk. This section describes SMRDB's design in detail.

## 4.1 Data Access Model and Management

Our goal is to demonstrate that SMR disks are capable of meeting modern storage needs in spite of the sequential write restrictions. To facilitate adoption, SMRDB is designed as a database engine that does its own data access and storage management, operating on a host-managed SMR drive without any drive remapping solutions. SMRDB supports the GET / PUT / DELETE / SCAN data access methods, in line with the successful KV data access model used in recent cloud storage systems.

Distributed databases either do their own key-space partitioning and KV data access management and offload storage management and replication to a distributed file system (DFS), or do their own partitioning and replication, offloading the data access management to external database engines. Tablet servers, such as BigTable (Chang et al. 2006), HBase (HBase 2015), and LogBase (Vo et al. 2012) belong to the first category and systems such as Dynamo (DeCandia et al. 2007) and Voldemort (Voldemort 2015) are of the second. SMRDB could be used as a standalone database engine, or existing file systems can use it to store blocks as fixed-sized KVs (LBAs as keys and block data as values) in SMR disks.

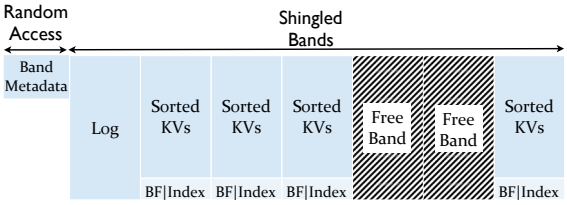



**Figure 4.** SMRDB's on-disk layout.

SMRDB does not require any data management by the drive firmware, only that the drive bands the disk into a small random-access region and fixed-sized shingled bands of requested size, as shown in Figure 4. If presented with a traditional hard disk, SMRDB splits most of the disk space into fixed-sized shingled bands and uses a small amount of random-access space. The random-access region is used to store only the high level shingled band information, and not key-specific metadata. The KV pairs and related metadata are all stored in shingled bands.

### 4.2 Data access operations

SMRDB's primary data access operations are similar to the simple solution we described in Section 3, and are described in detail here.

***PUT*** Newly inserted KV pairs are added to an in-memory *memtable*, which sorts them in lexicographic key order. When the size of the memtable reaches the capacity of a band, it is flushed to an empty shingled band. A Bloom Filter (BF) storing all the keys in the band, and an index mapping the keys to their locations inside the band are also stored in the band after the KV pairs. The in-memory KV pairs are also added to a much smaller log buffer and persisted in a separate log band. Systems requiring an increased write throughput could store the log on a NVRAM device instead. Bands are written to only sequentially, and remain read-only until a background *merger* copies data out of a band and deallocates it. Updating an already existing key invalidates the previous entry for the key and is handled like new inserts. During reads, only the most recent entry is retrieved. The invalidated entries are removed from the system by the background band merger.

***DELETE*** KV deletes are handled by inserting tombstone entries for the deleted keys, thus invalidating the previous entry for the key. The invalided entries are truly removed and the space freed when the bands are cleaned by the *merger*.

***GET*** GET first checks the in-memory table, and then the bands for the key. All the bands whose key ranges indicate that they might contain the key have to be searched, starting from the most recent band to the oldest. A key search first looks in the BF for the key, skips the band if not found, and looks in the index if found. The Bloom filters filter most of the bands, reducing the search cost, and with a low memory footprint, most of them could be cached in memory, reducing metadata disk accesses.

***SCAN*** Indexes of bands with overlapping key ranges are merged in memory and consulted to retrieve the range keys and their values. Since each band is ordered, and the keys reside physically close together on the disk due to their placement in large sequential bands, the only hindrance to near-optimal SCAN performance is the number of bands with overlapping key ranges. The background merger strives to keep this number low.

### 4.3 Background operations

Background compactions clean invalidated data, and strive to keep the entire disk's contents ordered, albeit split into multiple bands, with physical ordering within a band and logical ordering across bands, as shown in Figure 5. Compactions enable higher scan performance, but affect insert performance. SMRDB can be tuned for either higher random insert performance by triggering compactions less often, or higher scan performance by triggering compactions more often. SMRDB introduces an artificial slowdown factor, by which the inserts are slowed down if it determines that a compaction needs to be scheduled, to give the compactions more time to complete. The slowdown factor could also be tuned to give preference to either background jobs or incoming writes.

Ideally, the entire disk is ordered, as seen in Figure 5. To achieve this ideal state, the bands are organized into two levels, a first buffering level which is the result of memtable dumps, and a second mostly-ordered level.

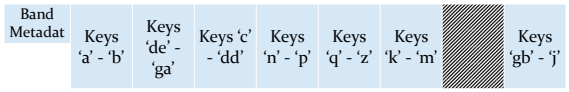



**Figure 5.** On-disk key ordering with physical ordering within a band and logical ordering across bands.

***Band Organization*** As shown in Figure 6, the SMR friendly organization only has two levels (0 and 1) and the bands in both levels can have overlapping key ranges. An effort to keep all the L1 bands ordered with no overlapping key ranges, to reach a state similar to the one shown in Figure 5, is made, but not strictly enforced. By removing the strict no-overlap rule for L1, we can select bands for compaction based on the cost it would incur vs the benefit the selection provides. We make sure the decision doesn't affect range reads, by assigning sequential access based 'benefit' points for the bands, as explained in the following section. Though we remove the strict upper bound on the number of BFs that need to be searched to read a value, our compaction scheme strives to keep the number low.

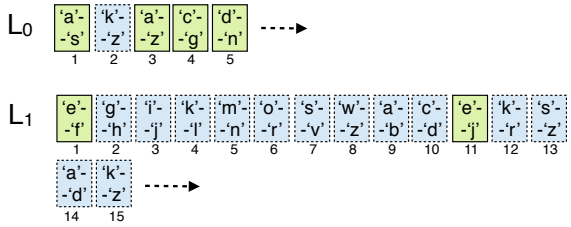



**Figure 6.** SMRDB's two-level organization. A key starting with 'e' could be in any one of the 6 marked bands.

***Compactions*** A user initiated manual compaction run will result in total cleanup and complete re-ordering of KV pairs, without any overlapping key range across bands. Regular background compactions select all overlapping bands in a selected key range and *prune* them to result in a smaller set of bands to merge, even if the pruning results in multiple bands with overlapping key ranges. To aid pruning, we define a *sequentiality* metric for each band. The metric measures how ordered a band already is, with respect to all the KV pairs stored in the entire database. Say all the KV pairs in the entire database were reordered, and the reordering did

not result in any KV pairs from any of the other bands to be inserted into the given bands' contents, then the band has the highest *sequentiality* score.

To estimate the *sequentiality* of a band, SMRDB builds an ***equi-depth*** **histogram** (Piatetsky-Shapiro and Connell 1984) for each band. In contrast to regular histograms with fixed bucket boundaries, an equi-depth histogram determines the bucket boundaries by keeping the number of values in each bucket equal, and has traditionally been used in database systems to perform query size estimation. The purpose is to specifically measure which sub-ranges hold the most data, and which don't, instead of just relying on the end values of the entire range. An equi-depth histogram based merely on the number of KV pairs in a sub range will not take into account the size of the KV pairs. Since we wish to avoid unnecessary reads and write, the histogram is built based on the data size, and determines the key sub-ranges, while keeping the byte size count equal in each sub-range. The chosen byte size determines the size of histogram metadata. Smaller sizes would result in more metadata and better estimation, but would require more memory utilization. SMRDB currently reuses the index information to build a fixed 4 KB histogram, which is used in all the experiments.

***Level-0 Table/Band selection:*** If a newer L0 band is selected for compaction, all older bands with overlapping key ranges in the level have to be chosen as well. For example, in Figure 6, the bands in a level are ordered by write time, and band 5 is newer than 4, which is newer than 3, etc.,. If band 5 is selected, then band 4 has to be selected as well, as a read expects the most recent value to be in the higher level and within a level, in the most recent band. Therefore, band 5 cannot be chosen, without choosing 4, 3, 2, and 1, while band 1 can be chosen without the rest. But we don't want to copy out older data to a new level, when a newer value exists. So, SMRDB chooses the oldest band and moves to newer bands, accumulating those that overlap (at least partially) with the oldest band, until we reach a threshold number of bands. If the selection resulted in only one L0 band and the L1 selection also turned out to be empty, the Table is just converted into a L1 Table without an actual copy.

***Level-1 Table/Band selection:*** The L1 band selection has to minimize the number of bands with overlapping key ranges, but should not trigger too many unnecessary band reads and writes. For a L0-to-L1 compaction, SMRDB first selects all L1 bands that overlap the selected L0 bands. For a L1-to-L1 compaction, triggered by too many bands with overlapping key ranges in the level, the L1 band that has the most overlaps is selected, as well as all the bands it overlaps. SMRDB, then *prunes* the selected bands and determines the band that requires the most reordering (in other words, is the least sequential) among them. The least sequential band and all L1 bands, that it overlaps, and are newer than it are selected for the compaction run, as it is safe to select newer bands in L1. For example, in Figure 6, it is safe to select band 11 and not band 1 in L1, but not vice-versa.

***Hot/Cold data separation*** Multi-level organization is believed to provide some amount of hot and cold data separation, where the upper levels contain hot data and the lower levels contain cold data. The general assumption is that hot data in an upper level will be cleaned out in the upper levels, and will not travel down to lower levels. But the order in which compactions take place is unpredictable, and hot data in a level could very easily travel down to the lower level, even when it has been already invalidated in a upper level. Multiple levels can also easily split sequential data across multiple bands. A better way to provide hot and cold data separation with less overhead would be to delay compaction at L0.

Hotness estimations provide more value in systems where the key space is limited and the users are forced to use/reuse the limited keys. But in a variable key length system, the users can avoid lots of data movement themselves, by simply making better use of the available flexible key space. We did not attempt to do any predictive hot data separation in this work. But our work could be extended to add one or more hot/warm data levels between L0 and L1, with actual KV hotness prediction and hot/warm KV movement between these levels.

## 5. Evaluation

SMRDB's design is better evaluated on a raw banded SMR disk without any interference from a drive-managed SMR disks's internal remappings, but such a disk is not yet publicly available to evaluate with. Hence, similar to the assumption made by the SMR emulator (Pitchumani et al. 2012), we assume that the performance of a raw SMR disk will be similar to today's PMR (standard) disk, and evaluate the performance of SMRDB using a regular hard disk, by banding it like a SMR disk. For our evaluation, SMRDB splits the available LBA range into fixed-sized bands, and reads/writes to the bands with SMR like restrictions, emulating how one would read/write to a SMR disk.

### 5.1 Experimental Setup

The evaluations were done on a VMware Linux guest running in a Macbook Pro host laptop. The host machine has a 2.7 GHz Intel Core i7 quad-core processor with a L2 cache of 256 KB per core, 6 MB L3 cache, and 16 GB RAM made up of 2 8 GB DDR3 1600 MHz cards. The guest machine is configured to use 2 cores and 8 GB RAM, and runs Fedora 18. The hard disk used for the tests is a Seagate Barracuda SATA 3 TB 7200 rpm disk. The disk is connected to the laptop via a high speed USB3 connection using a SATA to USB3 converter.

Figure 7 shows the raw sequential write performance of the disk in the setup. The disk performance was measured using the IO benchmark tool *fio*. 10 GB of data was writ-

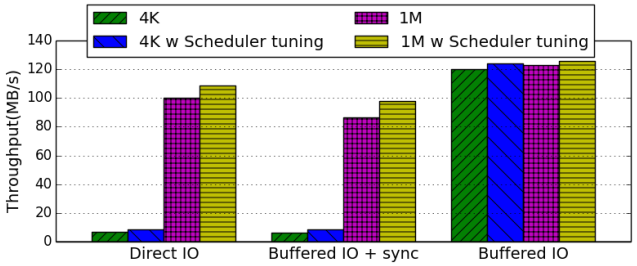



**Figure 7.** Raw sequential write performance of the test disk in the test environment.

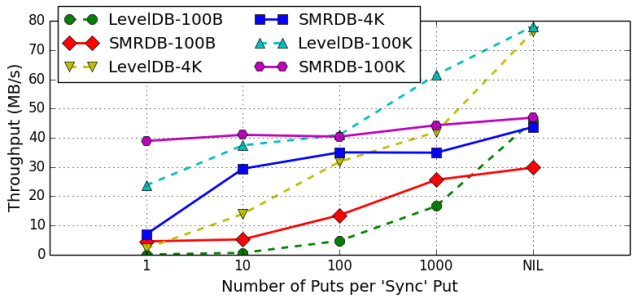



**Figure 8.** Sequential 'Put' performance for various value sizes (100 bytes, 4 KB, and 100 KB). Here, LevelDB-100B refers to LevelDB workload with 100 byte values, and so on. As SMRDB uses a small log buffer that gets flushed to disk more often, it is outperformed by LevelDB in cases with no or small amount of synced 'Puts'.

ten sequentially using direct IO to bypass the buffer cache, buffered IO using the buffer cache and buffered IO that syncs every IO to the disk. Two block sizes (4 KB and 1 MB) were used, each once with the default IO scheduler options and once with IO scheduler tuning that is described below. The goal was to measure the raw sequential write performance in the test environment to put subsequent results in perspective, and to demonstrate the effect of IO scheduler tuning.

As shown in the figure, buffered IO outperforms direct IO. But even though buffered IO with a sync for every IO has to write to disk the same as direct IO, it is slower than direct IO due to the buffering layer overhead. We switched from the default completely fair queueing IO scheduler to the noop scheduler and kept the queue length small. We also set the IO merging option to perform only simple one-hit merges instead of the default IO merging with complex lookups. The goal was to keep the work at the scheduling layer to the minimum, as they are not required by sequential writes. We verified that the tuned scheduler yielded better results for both LevelDB and SMRDB, and retained the tuning for all the experiments below.

### 5.2 Micro-Benchmarks

In this section, we micro benchmark SMRDB against LevelDB, using the *dbbench* benchmark that is shipped with LevelDB, and present the results here. LevelDB is run on the disk described in the previous section with ext4 filesystem in default configuration. SMRDB was also run on the same disk without any filesystem. To level the field, we chose the same memory buffer (*memtable*) size, 80 MB, for both LevelDB and SMRDB. LevelDB chooses the same amount of log buffer (80 MB) as memory buffer, giving it an unfair advantage over SMRDB, which uses a 1 MB log buffer.

#### 5.2.1 Sequential Writes

To measure the sequential write performance of SMRDB and LevelDB, we inserted key-value pairs with 16 byte keys in sequence and ran the tests for 3 value sizes: 100 bytes, 4 KB and 100 KB. 10 GB of data was inserted during all tests, and tests were identical for SMRDB and LevelDB. Both LevelDB and SMRDB handle an insert operation that requests a 'sync' by syncing the log. We did an insert with 'sync' for every 1, 10, 100 or 1000 inserts, and also ran one with no 'sync' inserts, and measured the throughputs. Figure 8 shows the results of the above experiment. As expected, inserting pairs with larger value sizes resulted in greater throughput. SMRDB outperforms LevelDB when there are many *synced* writes. But in all no sync write cases and 1 sync per 1000 insert for 4 KB and 100 KB values, even though both systems were configured with the same memory buffer, LevelDB outperforms SMRDB, as LevelDB chooses a log buffer as big as the memory buffer. But if logging is disabled in both SMRDB and LevelDB, SMRDB's performance doubles its default case, and outperforms LevelDB. Performance can thus be greatly improved, if the log is moved to another location, such as an NVRAM device.

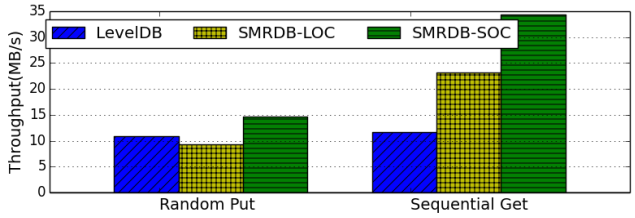



**Figure 9.** Performance of random 'Put' and sequential 'Get' after 'Put'.

#### 5.2.2 Random Writes and Compaction

Sequential inserts do not trigger any compactions, so performance is, as expected, very good. To illustrate the effects of compaction to the fullest, out next test performs uniformly distributed random key inserts. All inserts total roughly 11 GB of data. All the tests here use 16 byte keys and 4 KB values and have no user syncs. We compare 3 cases: default LevelDB which has the log buffer advantage over SMRDB, the LevelDB-based SMR solution described in Section 3 (denoted as SMRDB-LOC) and SMRDB with the SMR friendly organization and compaction described in

Section 4 (denoted as SMRDB-SOC). Figures 9 and 10 show the overall performance of random inserts, compaction overhead and the sequential performance immediately after all the inserts have completed.

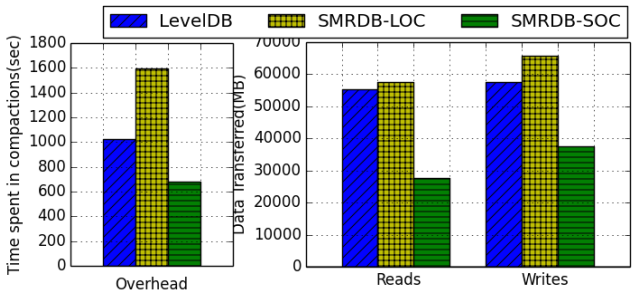



**Figure 10.** Background compaction overhead measured both in terms of amount of data read and re-written and time spent for writing 11 GB of randomly inserted data.

As seen in Figure 9, the insert throughput of SMRDB with LevelDB style compactions is slightly less than LevelDB, but with our SMR friendly approach, it outperforms LevelDB, even with LevelDB's log buffer advantage. Figure 10 shows the compaction overhead in greater detail. The graph on the left shows the time spent, in seconds, doing compactions for all three cases, and the graph on the right shows how much data was read and written by the DBs (excluding the log write) for inserts that total roughly 11 GB of data. The SMR friendly approach is clearly better than LevelDB style compaction. Further, as seen in Figure 9, SMRDB's sequential read throughput is higher than LevelDB, because sequential placement in bands ensures physical KV proximity, and the SMR friendly compaction is even better for sequential reads than the LevelDB style compaction.

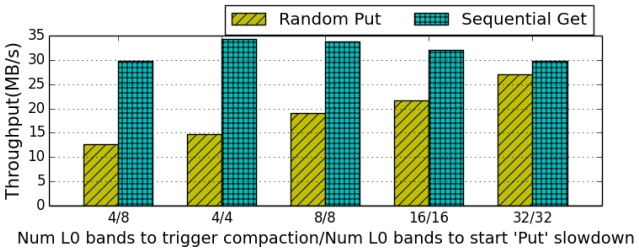



**Figure 11.** The insert performance increases when SMRDB's compaction is delayed.

The random write performance is much less than the sequential write performance on all setups due to compactions. The results depict the worst case scenario, and in a natural workload, where the inserts aren't completely random and don't affect all L1 bands equally, the performance would be much better. Nevertheless, better random insert performance would be desirable. As mentioned in the previous section, a better random write performance could be achieved by delaying L0 band compactions, and in turn sacrificing the sequential read performance.

In LevelDB, the default number of files to trigger a compaction is 4 and the default number of files to start slowing down the incoming writes is 8. We retained the numbers for SMRDB's L0 bands in the previous experiments. In this experiment, we varied these numbers in SMRDB and show the results in Figure 11. As seen in the figure, the random insert performance increases as the number of bands to trigger compaction increases. As the compactions get delayed, the number of times the same data gets read and re-written decreases, improving the insert performance. We also measured the sequential read performance after the inserts. The results in the figure are best case results, as every time the previous compaction would have completed and the resulting number of L0 files after the inserts were complete were always less than 6.

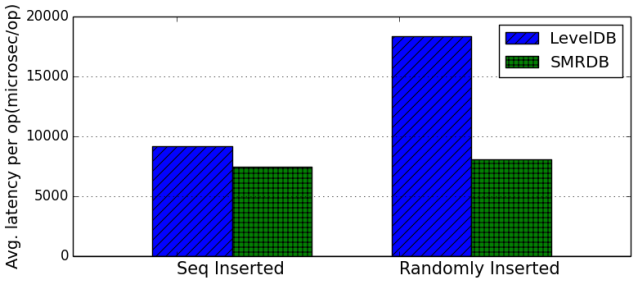



**Figure 12.** SMRDB has better random read performance, both after inserting the keys sequentially and randomly.

### 5.2.3 Reads

The sequential read performance was illustrated in the previous results. The kernel read ahead was not tuned in any way in the above tests. Increasing the read-ahead will not only increase the sequential read performance, but will also improve the compaction runtime and random insert performance, but would affect the random read performance. Hence, we did not do any tests with read-ahead variations; users can tune it to their liking based on the expected workload. The random read results are shown in Figure 12. We measured average per read latency for both LevelDB and SMRDB, once after inserting the key-value pairs sequentially, and again after inserting them randomly. SMRDB performs much better than LevelDB, especially after random inserts, as SMRDB's organization and compaction mechanism results in fewer SSTables to be searched.

### 5.2.4 Band Size

We had fixed the band size to 80 MB in the previous experiments, and also used the same memory buffer size for both LevelDB and SMRDB. The chosen band size had to be big enough to demonstrate the effect of big band sizes in SMR

disks, but not too big, to ensure a fair comparison (since LevelDB's default behavior was to use a log buffer the same size as memory buffer, very large size would not be fair to SMRDB), and our choice was 80 MB.

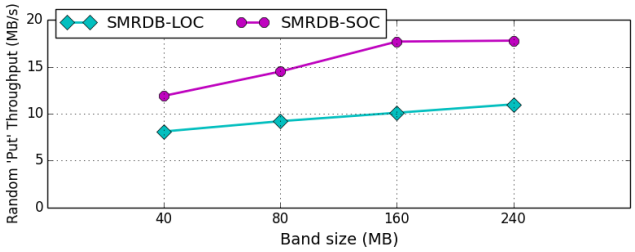



**Figure 13.** Bigger band sizes have better performance.

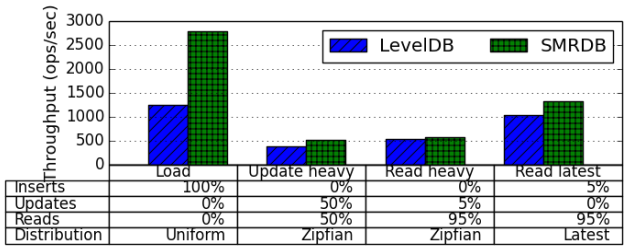
To illustrate the band size effect, we measured the random write performance of SMRDB, with LevelDB style compaction, and SMR friendly compaction, while varying the band size, shown in Figure 13. Sequential writes are not affected by varying the band sizes, and bigger bands are better for random reads, as the number of bands to search decreases. Bigger bands have larger sequential regions and are also good for sequential reads, and the sequential read performance after random inserts were similar to that of previous results. Since the main concern is the data movement for merges, we present only the random write performance.

We did not change the number of L0 bands that trigger compaction, as we varied the band sizes. This resulted in more data accumulation in L0, as the band size increased and slightly delayed compactions. The accumulation was justified as each band is ordered and larger band sizes ensure more ordering. The slight delay in compaction resulted in better performance as band size increased even for old LevelDB style compaction. SMRDB with SMR friendly compactions performed much better as band size increased. Bigger band sizes not only saves the space wasted for banding, but also improves performance in SMRDB.

### 5.3 Macro-Benchmarks

In addition to the above micro-benchmarks, we evaluated SMRDB against LevelDB, using a macro-benchmark suite, to gauge its performance on application level workloads. We use Yahoo! Cloud Serving Benchmark (YCSB) (Cooper et al. 2010), which has become a standard for cloud storage systems and key-value systems. As both LevelDB and SMRDB are embeddable databases, to be able to connect and communicate with YCSB, we used the MapKeeper server. We set up both the LevelDB-based and SMRDB-based MapKeeper servers to 'sync' every KV write, allocate the same amount of write buffer as previous experiments, and same default 8 MB cache (as we wanted to measure only the disk read/write performance).

YCSB was configured to use 4 KB values, and both systems were first loaded with 2 million entries. We chose 3 workloads: an update heavy workload and a read heavy




| | Load | Update heavy | Read heavy | Read latest |
|---|---|---|---|---|
| Inserts | 100% | 0% | 0% | 5% |
| Updates | 0% | 50% | 5% | 0% |
| Reads | 0% | 50% | 95% | 95% |
| Distribution | Uniform | Zipfian | Zipfian | Latest |

**Figure 14.** SMRDB consistently performs better than LevelDB in various benchmarks, that are part of the the YCSB Suite.

workload, with keys selected from a Zipfian distribution, and a read latest workload, that inserts keys and reads those keys that were recently inserted. The workloads each performed 200,000 operations to record the performance. Figure 14 describes the nature of the workloads, and compares the performance of the two systems, during the load phase, and under all 3 workloads. SMRDB clearly performs better than LevelDB in all cases. Further, SMRDB's performance in the update heavy workload, that follows a Zipfian distribution, disproves the theory that LevelDB style multi-level data organization is better suited to handle hot data.

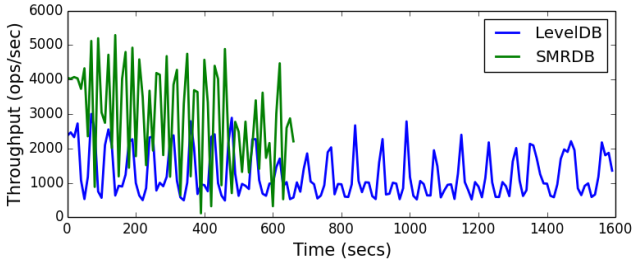



**Figure 15.** Performance of YCSB's load phase, which inserts uniformly distributed keys, shown over time.

We expand on the load phase performance further, in Figure 15. The figure shows the throughput of the system, measured in number of operations per second, over time. Though SMRDB delivers much higher throughput than LevelDB most of the time, and completes in less that half the time it takes LevelDB to load all the KV pairs, it does not deliver a consistent throughput. Similar to LevelDB, there are periodic drops in throughput, owing to background compactions. We can give a higher preference to random inserts than to range reads, and remove a percentage of the throughput drops by slowing down the compaction rate. But fluctuations in performance, similar to the ones seen in the figure, will still be there, eventually whenever compaction is being run. bLSM (Sears and Ramakrishnan 2012) presented a new merge scheduler to mitigate the throughput impact in multi-level LSM trees. We consider techniques to mitigate the in-

evitable performance degradation in SMRDB to be future work.

## 6. Related Work

In this section, we present previous research related to our work, covering both alternate data management approaches for SMR disks and log-structured key-value stores designed for other existing storage devices.

***SMR Data Management*** The LFS organizes the disk into a segmented, append-only log and batches writes to the end of the log. Various ways to adopt the LFS layout for SMR disks, complete with garbage collection suggestions, has been proposed (Amer et al. 2011). NAND flash devices also face update-in-place issues like SMR disks, and employ a LFS-based indirection system in the device firmware to hide the access restrictions and present a traditional disk like block interface. Following suit, the use of a Shingle Translation Layer (STL) in shingled disk firmware similar to the Flash Translation Layer (FTL) in solid state storage devices has been proposed (Gibson and Ganger 2011). But a key difference between SMR disks and NAND flash is random read performance: shingled disks face penalties incurred by disk head arm movement unlike NAND flash, and is one of the reasons why LFS layout has been more successful in flash than in disks. And, the LFS layout with segments as large as bands in SMR disks, which are expected to be in the order of 100 MB, hasn't been evaluated yet.

The simplest and most straight-forward solution is to assign fixed logical block addresses to the physical locations of the shingled bands and perform updates to a sector in the shingled band by reading all the sectors from the sector to be updated to the end of the band, modify the sector content in-memory and rewrite all the sectors that were read in. This approach would work well for target workloads without lot of data modifications. For example, SFS (Moal et al. 2012), a SMR aware file system for video recorders/set-top boxes takes this approach. For other workloads, performance will obviously take a hit. A block-based indirection system using a buffer band to buffer incoming writes and read-modify-write based updates to the shingled bands has been proposed (Cassuto et al. 2010). Intelligent schemes that combine read-modify-write with track level indirection and a buffer band have also been proposed (Hall et al. 2012; Venkataraman et al. 2012).

However, read-modify-write will result in data loss if there was a failure after the modified write started, unless the big chunk that was read into memory was backed up elsewhere, or the modified version is written to free band. This possibility has not been taken into account by most systems and has often been ignored. The indirection-based schemes' success will also be heavily dependent on the workload. If implemented as a drive-managed solution, it will be difficult for the disk to provide a consistent performance guarantee and though existing file systems and applications will work, their basic performance assumptions will no longer be true. Suresh *et al.* (Suresh et al. 2012) present a SMR file system for big data applications that writes to files only sequentially, never reopen a closed file for a write, and never rewrite a block in the file.

***Log structured Key-Value Stores*** Key-value storage systems are abundant; we look only at those that follow a log-structured approach to writes, because it is a key requirement for SMR disks. FAWN-KV (Andersen et al. 2009), FlashStore (Debnath et al. 2010) and SkimpyStash (Debnath et al. 2011) are all key-value stores for NAND flash that adopt a log based storage combined with some sort of hash based in-memory indexing. Since flash does not pay a random-access penalty, but pays for write amplification with its lifetime, these systems mostly stick to the LFS style, where data is sequentially appended once and not rewritten until the time to *clean* dead data, with improvements on indexes. Key-value databases that adopt an LSM-tree based layout include Bigtable (Chang et al. 2006), HBase (HBase 2015), LevelDB (Ghemawat and Dean 2015), LogBase (Vo et al. 2012), KVDB (Shetty et al. 2013).

## 7. Conclusion

Sustaining hard disk areal density growth requires a technology transition, and Shingled Magnetic Recording is the most likely next generation disk technology. SMR squeezes more data tracks on the existing surfaces by overlapping the tracks like the shingles on a roof. Since updating a data track overwrites data written on subsequent tracks, random writes and update-in-place data management techniques are destructive and cannot be used on SMR disks. As a result, SMR disks require new SMR-aware data management solutions.

We presented SMRDB, a key-value database engine for SMR disks, in this work. We are the first to suggest, optimize and evaluate an LSM tree based data layout and management for SMR disks. We evaluated its performance against LevelDB and showed that SMRDB outperforms LevelDB in most cases. Our work proves that SMR disks are capable of replacing traditional disks in a variety of applications. Our design could be adopted either as a drive-managed solution, or a host-managed solution and our work enables the easy adoption of SMR disks. In this work, SMRDB adopts a fixed-sized band model, with entire band allocation/deallocation. In the future, we intend to work on more data layout policies to further improve SMRDB. Design possibilities using variable sized bands with partial band allocations/deallocations are yet to be explored.

## Acknowledgments

The authors would like to thank Seagate Technology for supporting this work. This research was also supported in part by the National Science Foundation under award IIP-1266400 and the industrial members of the Center for Research in Storage Systems.

## References


A. Amer, J. Holliday, D. D. E. Long, E. L. Miller, J.-F. Pâris, and T. Schwarz. Data Management and Layout for Shingled Magnetic Recording. *IEEE Transactions on Magnetics*, 47(10): 3691–3697, Oct. 2011.

D. G. Andersen, J. Franklin, M. Kaminsky, A. Phanishayee, L. Tan, and V. Vasudevan. FAWN: A Fast Array of Wimpy Nodes. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles*, SOSP '09, Oct. 2009.

J. Campello. Shingled Magnetic Recording (SMR) Introduction. `http://www.t10.org/cgi-bin/ac.pl?t=d&f=13-148r0.pdf`, 2013.

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 Mass Storage Systems and Technologies*, MSST '10, 2010.

F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A Distributed Storage System for Structured Data. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '06, 2006.

B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, 2010.

B. Debnath, S. Sengupta, and J. Li. FlashStore: High Throughput Persistent Key-Value Store. In *Proceedings of the 36th Conference on Very Large Databases*, VLDB '10, Sept. 2010.

B. Debnath, S. Sengupta, and J. Li. SkimpyStash: RAM space skimpy key-value store on flash-based storage. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, SIGMOD '11, 2011.

G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's Highly Available Key-value Store. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles*, SOSP '07, Oct. 2007.

T. Feldman and G. Gibson. Shingled Magnetic Recording: Areal Density Increase Requires New Data Management. *;login:*, 38 (3), June 2013.

S. Ghemawat and J. Dean. LevelDB. `https://github.com/google/leveldb`, 2015.

G. Gibson and G. Ganger. Principles of Operation for Shingled Disk Devices. Technical Report CMU-PDL-11-107, Carnegie Mellon University, 2011.

S. Greaves, Y. Kanai, and H. Muraoka. Shingled Recording for 2–3 Tbit/in$^2$. *IEEE Transactions on Magnetics*, 45(10):3823–3829, Oct. 2009.

D. Hall, J. H. Marcos, and J. D. Coker. Data Handling Algorithms For Autonomous Shingled Magnetic Recording HDDs. *IEEE Transactions on Magnetics*, 48(5):1777–1781, May 2012.

HBase. `https://hbase.apache.org/`, 2015.

J. N. Matthews, D. Roselli, A. M. Costello, R. Y. Wang, and T. E. Anderson. Improving the Performance of Log-structured File Systems with Adaptive Methods. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, SOSP '97, 1997.

D. L. Moal, Z. Bandic, and C. Guyot. Shingled File System Host-Side Management of Shingled Magnetic Recording Disks. In *IEEE International Conference on Consumer Electronics*, 2012.

P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil. The Log-Structured Merge-Tree (LSM-Tree). *Acta Informatica*, 33:351–385, 1996.

G. Piatetsky-Shapiro and C. Connell. Accurate Estimation of the Number of Tuples Satisfying a Condition. In *Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data*, SIGMOD '84, 1984.

R. Pitchumani, A. Hospodor, A. Amer, Y. Kang, E. L. Miller, and D. D. E. Long. Emulating a Shingled Write Disk. In *Proceedings of the 2012 IEEE 20th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, MASCOTS '12, 2012.

K. Ren and G. Gibson. TABLEFS: Embedding a NoSQL Database Inside the Local File System. In *IEEE Asia-Pacific Magnetic Recording Conference*, APMRC 2012, 2012.

M. Rosenblum and J. K. Ousterhout. The Design and Implementation of a Log-Structured File System. *ACM Transactions on Computer Systems*, 10(1):26–52, Feb. 1992.

Seagate-Kinetic. The Seagate Kinetic Open Storage Vision. http://www.seagate.com/tech-insights/kinetic-vision-how-seagate-new-developer-tools-meets-the-needs-of-cloud-storage-platforms-master-ti/, 2014.

R. Sears and R. Ramakrishnan. bLSM: A General Purpose Log Structured Merge Tree. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, 2012.

P. Shetty, R. Spillane, R. Malpani, B. Andrews, J. Seyster, and E. Zadok. Building Workload-Independent Storage with VT-Trees. In *Proceedings of the 11th USENIX Conference on File and Storage Technologies*, FAST'13, 2013.

SMR-8TB-HDD. Seagate 8TB SMR HDD. `http://www.seagate.com/products/enterprise-servers-storage/nearline-storage/archive-hdd/`, 2014.

J. Stender, B. Kolbeck, M. Högqvist, and F. Hupfeld. BabuDB: Fast and Efficient File System Metadata Storage. In *Proceedings of the 2010 International Workshop on Storage Network Architecture and Parallel I/Os*, SNAPI '10, 2010.

A. Suresh, G. Gibson, and G. Ganger. Shingled Magnetic Recording for Big Data Applications. Technical Report CMU-PDL-12-105, Carnegie Mellon University, May 2012.

K. S. Venkataraman, G. Dong, and T. Zhang. Techniques Mitigating Update-Induced Latency Overhead in Shingled Magnetic Recording. *IEEE Transactions on Magnetics*, 48(5):1899–1905, May 2012.

H. T. Vo, S. Wang, D. Agrawal, G. Chen, and B. C. Ooi. LogBase: A Scalable Log-structured Database System in the Cloud. *Proceedings of the VLDB Endowment*, 5(10):1004–1015, June 2012.

Voldemort. `http://www.project-voldemort.com/voldemort/`, 2015.