UNIVERSITY OF CALIFORNIA
SANTA CRUZ

DATA MANAGEMENT FOR SHINGLED MAGNETIC RECORDING DISKS

A dissertation submitted in partial satisfaction of the
requirements for the degree of

DOCTOR OF PHILOSOPHY

in

COMPUTER SCIENCE

by

Rekha Pitchumani

December 2015

The Dissertation of Rekha Pitchumani
is approved:

Ethan L. Miller, Chair

Darrell D. E. Long

Ahmed Amer

Tyrus Miller
Vice Provost and Dean of Graduate Studies

Table of Contents

List of Figures

List of Tables

Abstract

Data Management for Shingled Magnetic Recording Disks

by

Rekha Pitchumani

The areal density growth of hard disk drives is hindered by the limits imposed by the laws of physics and any further significant growth to the density demands some major changes to the currently employed recording techniques. Shingled Magnetic Recording (SMR) is leading next generation disk technology. 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. We hypothesize that using a smarter interface and an efficiently designed SMR-aware data management solution that overcomes the unique data management challenges faced by these disks, they could fit into more traditional roles. This thesis proposes a key-value object interface for SMR disks, and presents SMRDB, a Log-Structured Merge (LSM) tree based 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 (YCSB) results show that, despite being restricted to sequential writes, SMRDB *outperforms* LevelDB by 8.8–123.6%.

A workload generator that generates loads with realistic temporal characteristics is required to measure the realistic impact compaction has on incoming requests and to measure the effectiveness of compaction overhead mitigation techniques. Though YCSB has emerged as the standard benchmark for evaluating key-value systems, and provides a variety of options to generate realistic workloads, like most benchmarks it has ignored the temporal characteristics of generated workloads. YCSB's constant-rate request arrival process is unrealistic and fails to capture the real world arrival patterns. Existing workload studies on disk, filesystem, key-value system, network, and web traffic all show that they all exhibit some common temporal properties such as burstiness, self similarity, long range dependence, and diurnal activity. In the second part of this thesis, we show that the commonly observed traffic patterns can be modeled using the three categories of arrival processes: a)*Poisson*, b)*Self similar*, and c)*Envelope-guided* processes, and have incorporated all three models into YCSB.

Finally, we hypothesize that the negative impact that compaction would have on performance can be mitigated in a hybrid drive configuration, by *write-offloading* to the NVRAM component of the hybrid drive when compaction is in progress. In the final part of this thesis, we have created a version of SMRDB with design improvements made possible by write-offloading, and a compaction overhead more representative of other proposed SMR data management schemes, to evaluate the effectiveness of write-offloading. We show that write-offloading not only hides the compaction overhead and improves write performance, but also makes room for more compaction-induced data rearrangements.

To my mom, dad, husband and son for all their love and support.

Acknowledgments

A little more than five years back, I decided to move from India to the United States with my two year old son, to begin my graduate studies. It was a life-changing decision for me. I wouldn't have started grad school or successfully written this thesis without the encouragements and support from lots of people in my life. I thoroughly enjoyed the experience and I am thankful to everyone who were there for me and made this an enjoyable experience.

This work would not be possible without the guidance I received from my advisor, Ethan L. Miller. My discussions with him, be it regarding my research, or internships, or career decisions, or personal problems, have always been very helpful. Over the years, he has been behind the growth I see in myself. I am forever thankful for the high bar he sets for his students, and the support he provides us to rise up to the expectations. I also thank James L. Hughes, who was my mentor and manager at Seagate, for all the discussions that helped me shape my research. His tremendous support for my research made my internship experience at Seagate a memorable one. He gave me the opportunity to make a real world impact whilst pursuing my Ph.D.

I am very much grateful for the opportunity to work closely with Ahmed Amer when I started my first research project. His enthusiasm and passion for research is catching. The encouragement and help he offers to students who are in need of it, is very much appreciated by all of us and has inspired us to do the same for each other. I thank Darrell Long for all the pieces of wisdom he has offered over the years. He taught me to be both appreciative and critical of the research papers we read. Ahmed and Darrell has also teamed up with Ethan to be both on my thesis committee and my advancement committee and helped to turn my ideas into meaningful research. I also thank Erez Zadok for his helpful comments on my advancement proposal.

I have been lucky to receive funding through several different sources over the course of this work. I am thankful for the support received in part by the National Science Foundation under award IIP-1266400 and the industrial members of the Center for Research in Storage Systems. I am also thankful for the support received through my internship at Seagate.

I thank my other co-authors Yangwook Kang, Andy Hospodor, Thomas Marlette, Shayna Frank, Stephanie Jones, and Christina Strong; the experiences of working with all of them have been valuable for my growth as a researcher. I thank all my other colleagues at the Storage System Research Center and alumni of the lab for their sympathetic ears. I thank all the friends I made while at UCSC, especially Manikandan Punniyakotti, Rajan Vaish, and Preeti Gupta for being good friends. I also thank the wonderful family student housing community at UCSC and the childcare center for making my initial year at UCSC, the one that I spent far away from my husband and other family members with a toddler, a comfortable one. Without their support, I wouldn't have been able to complete my coursework and engage in research.

My parents have always believed in me, even when I have lost hope, and have guided me all through my life. I thank my dad for instilling in me an interest in research and encouraging me to become an independent thinker. I thank my mom for patiently listening to everything I say, and telling me that everything will fall in place if I just keep doing my work; she is always right. I am thankful to my loving husband, Pitchai, who quit a job that he liked and moved, to be where I wanted to be, to pursue my dreams. I am forever grateful for his understanding and support. I am also thankful to my son, Vidhush, for whom I am the world, for his smiles, giggles, and love. I truly started pushing my own boundaries only after giving birth to him. I also thank my brother for his unwavering support for all my endeavors.

I couldn't have done this without you. Thank you, all of you.

Chapter 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 [91] and the industry is eager for the introduction of new technologies to overcome the limit.

Recently, Coughlin and Grochowski [26] also noted that the period from 2000 to 2010 saw an average annual areal density growth of 40-50%, but the year 2011 saw only a 20% annual growth. The next major spurt in areal density growth is expected to occur with the introduction of Heat Assisted Magnetic Recording (HAMR) drives, but unfortunately HAMR requires huge changes in the manufacturing process and high investments for new manufacturing capital equipment [26]. Meanwhile, Shingled Magnetic Recording (SMR) [44] [111] is leading next generation disk technology, thanks to the technology's ability to increase density without any major changes to the underlying recording medium or the disk head design.

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.

Existing file systems designed for traditional disks cannot handle data management on SMR disks, as most of their data layout policies rely on random write capability to write new data, and adopt the update-in-place mechanism to write modified data. Even the solutions that update data out-of-place largely rely on the ability to randomly write to any desired location on disk, and would require a redesign to adapt to SMR disks when performance matters. For example, the Log structured File System (LFS) [98] 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.

For better manageability, SMR disks can be divided into multiple multi-track sequentially writable *bands* by sacrificing the capacity of some tracks and utilizing those tracks as special *guard tracks* between the bands. The LFS layout could be adapted to SMR disks by increasing the segment size to match the SMR band size, but the effects of such an increase has not been well evaluated. 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, even for original segment size [74]. Nevertheless, sequential read performance can be expected to be poor with a traditional LFS layout, without more aggressive data movement.

The data also need 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 regions. 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 index structures 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.

Further, SMR access restrictions can be handled in the drive, in the host, or co-operatively by both the host and the drive [37]. Standardization efforts to broadly categorize these disks into Autonomous (drive managed) and Host-Managed drives [19], and to bring about new standard command sets assisting the different categories are underway. The suitable interface for an SMR disk is also largely unknown, providing room for new interfaces such as that of Seagate's Ethernet key-value drives [101], thanks to the new shift towards distributed, highly scalable NoSQL data stores.

1.1 Thesis Contributions

In this thesis, we address this problem by presenting an SMR-friendly data management solution that is backward compatible with traditional hard disks, in that it will work on traditional disks, as it would on an SMR disk. This thesis examines the hypothesis that a high performance data management solution for SMR disks is possible and 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. This hypothesis is evaluated in the three contributions of this thesis:

1. 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 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. To aid in a more realistic evaluation of SMRDB's compaction overhead, we classify typically observed temporal patterns into three kinds of arrival processes, based on existing workload studies. We incorporated inter-arrival time generators based on all three temporal models into the Yahoo! Cloud Serving Benchmark's framework.
3. To mitigate the performance impact, we suggest a hybrid drive configuration and employ *write-offloading* to the NVRAM component of the hybrid drive, when and only when background compaction is in progress. We use our inter-arrival generators to not only show that the technique mitigates the compaction impact on foreground performance, but also to measure utilization of the NVRAM component under realistic loads.

1.2 Key-Value Data Management for SMR 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.

Both the industry and the research communities 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 [22, 31]. 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 [85] 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 [113, 94], and an entire user-level file system [105]. Thus, our solution enables easy adoption of SMR disks in a wide range of applications.

SMRDB manages the underlying disk in an 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 [92]. Therefore, we have evaluated SMRDB using a traditional disk written like an SMR disk. Our initial evaluation compares SMRDB against LevelDB, state-of-the-art LSM-tree based embeddable database engine and shows that SMRDB outperforms LevelDB in most cases.

1.3 Realistic Request Arrival Generation

Our initial evaluation uses the Yahoo! Cloud Serving Benchmark (YCSB) [25], which is the standard benchmark of choice for evaluating key-value systems. Though YCSB comes with a workload generator that is flexible in the selection of different mixes of operations and data sizes, and key selections based on different distributions, its framework does not come with the flexibility to generate realistic request arrivals. A trace based evaluation is also out of reach, as, to the best of our knowledge, no key-value workload trace is publicly available. The only published key-value workload study is that of an in-memory key-value caching layer, Facebook's Memcached deployment [13], and even if obtained it will be unsuitable to evaluate individual key-value storage nodes, such as SMRDB.

Although, workload studies that analyze traces collected from real systems aid in realistic synthetic workload generation and the only published key-value workload study, the one on Facebook's in-memory key-value caching layer [13], notes that the observed workloads are bursty, and diurnal with traffic spikes. The observed temporal patterns do not come as a surprise, as existing studies show that disk, filesystem, network, and web traffic all exhibit some common temporal properties such as burstiness, self similarity, long range dependence, and diurnal activity.

To aid in a more realistic evaluation of SMRDB, we classify typically observed temporal patterns into three kinds of arrival processes: a)*Poisson*, b)*Self similar*, and c)*Envelope-guided* processes. We have implemented inter-arrival time generation based on all three temporal models, and extended YCSB's framework to enable a workload executor that sends requests based on them. We have evaluated the generated requests and show that the statistical properties of the generated requests conform to that of the arrival process selected. Further, to demonstrate the usefulness of realistic arrivals, we used the modified code to evaluate a modern embeddable key-value store, LevelDB [39], one on which SMRDB is based upon.

1.4 Measuring Compaction Overhead and Its Mitigation

The background compaction (garbage collection and data reorganization) process is an added overhead and interferes with incoming requests. Background tasks in storage systems, such as scrubbing, cache de-staging and data migration across tiers, typically co-exist with foreground request processing. Most systems treat these tasks as lower priority tasks and schedule them to be performed with minimal interference to the incoming traffic, while also ensuring they do not starve. Such scheduling is not trivial in background activities such as compaction in SMRDB, as the reorganization is not a low priority task, but a higher priority activity required for good range read performance.

This thesis measures compaction's impact on foreground request processing under realistic loads generated using our work described in the previous section, and proposes a method to mitigate the impact. Caching improves read performance, and when employed would reduce the number of read requests reaching the drive. Even though the missed requests have to be read from the disk, to minimize interference from compaction, they can be served with a higher priority while holding off compaction. As caching mitigates the impact on read requests, and has been well studied, in this thesis, we focus on the write request handling.

This work mitigates the impact on incoming user write traffic when a small NVRAM component is added to the disk. We *offload* the writes to the NVRAM component, and show that the technique successfully hides the compaction activity in the background and improves the write performance tremendously. Given that the lifespan of some NVRAM devices are determined by the total bytes written to it, we offload the writes when and only when compaction is in progress (to reduce the total writes to NVRAM), and measure the total bytes written to NVRAM due to write-offloading under various realistic loads.

1.5 Organization

This thesis is organized as follows:

Background and related work: Chapter 2 describes the background on Shingled Magnetic Recording technology and its unique data management challenges. We discuss the interface options for SMR disks and the role the interface plays on the data management solutions's efficiency. We describe existing solutions and discuss why they are insufficient to solve SMR's challenges, and the need for our work.

SMRDB: Key-Value Data Store for SMR Disks: Chapter 3 presents the design and evaluation of SMRDB, our LSM-tree based key-value object management solution. We compare SMRDB against a state-of-the-art embeddable key-value store, using both a micro-benchmark and macro-benchmark, the Yahoo! Cloud Serving Benchmark, the standard benchmark for evaluating key-value systems, and present the results.

Realistic Request Arrival Generation: Chapter 4 describes our work enhancing the Yahoo! Cloud Serving Benchmark to incorporate realistic temporal characteristics into the generated workload. We describe the three temporal characteristic models, that we have determined to be a necessary and sufficient set of request arrival models that all storage benchmarks should provide.

Compaction Overhead and Mitigation: Chapter 5 presents write-offloading as a technique to mitigate the performance impact compaction has on incoming write requests and the results when tested under various realistic loads.

Conclusion: Chapter 6 discusses the future directions of this work, where we present few areas of extension for our work and new research possibilities that arise out of this thesis, and conclude this thesis.

Chapter 2

Background and Related Work

Disk drives are magnetic storage devices that record data bits on small grains on the recording media by the direction of the grain's magnetization. These grains need to be made smaller to pack more data and thus, to increase the areal density. The currently employed disk technologies are reaching the density limits imposed by the super-paramagnetic effect, which makes smaller grains thermally unstable. Further, popularly known as the media trilemma [91], the media signal-to-noise ratio, the write-ability of the media by a narrow track head, and the thermal stability of the media are all related in a way that improving one will deteriorate the others.

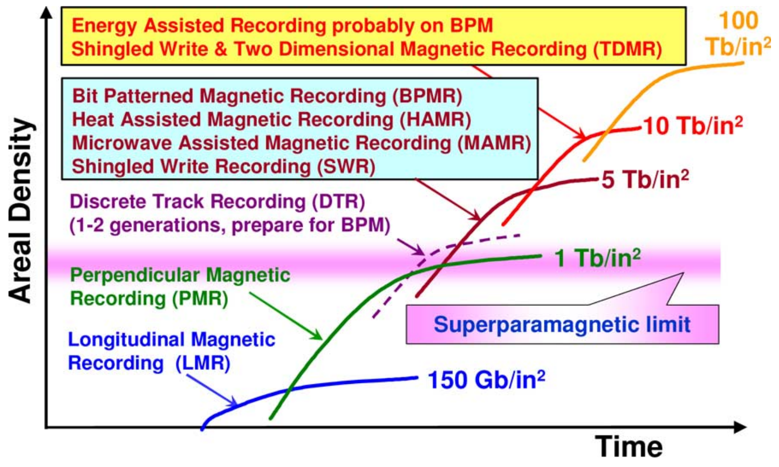


Figure 2.1: Future disk technology roadmap

Figure 2.1 from [108] gives the future disk technology roadmap and their expected areal density limits. The super-paramagnetic limit on the currently employed Perpendicular Magnetic Recording (PMR) technology is indicated by a pink line in the figure. Thermally assisted technologies [73] like Heat and Microwave Assisted Magnetic Recording (HAMR, MAMR) [99, 132] require thermal assistance to write data to a higher anisotropy recording media that is more thermally stable. Discrete Track Recording (DTR) and Bit Patterned Magnetic Recording (BPMR) [95] both work with patterned media involving advanced lithographic changes. BPMR's challenge is etching one island with well defined position and geometry per bit, and finding a way to write data to these islands.

While all the above technologies require major changes to the manufacturing processes of the current disk head, the recording media, or both, SMR and Two Dimensional Magnetic Recording (TDMR) [128] don't. SMR is the most easy to implement solution out of all the future technology solutions, but alters the traditional track writing mechanism, making random writes unsafe. SMR with regular reads is the next immediate technology to be adopted and will probably play a role on most of the other future disk technologies as well.

The rest of the chapter provides the necessary background information on Shingled Magnetic Recording technology and the associated data management challenges. This chapter compares SMR disks to traditional disks and NAND flash, to understand why SMR disks require a new data management solution, and describes the different SMR drive types, the drive interface choices, and other proposed alternate data management approaches in detail.

2.1 Shingled Magnetic Recording

Hard disks have a magnetic recording medium that is organized into 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.

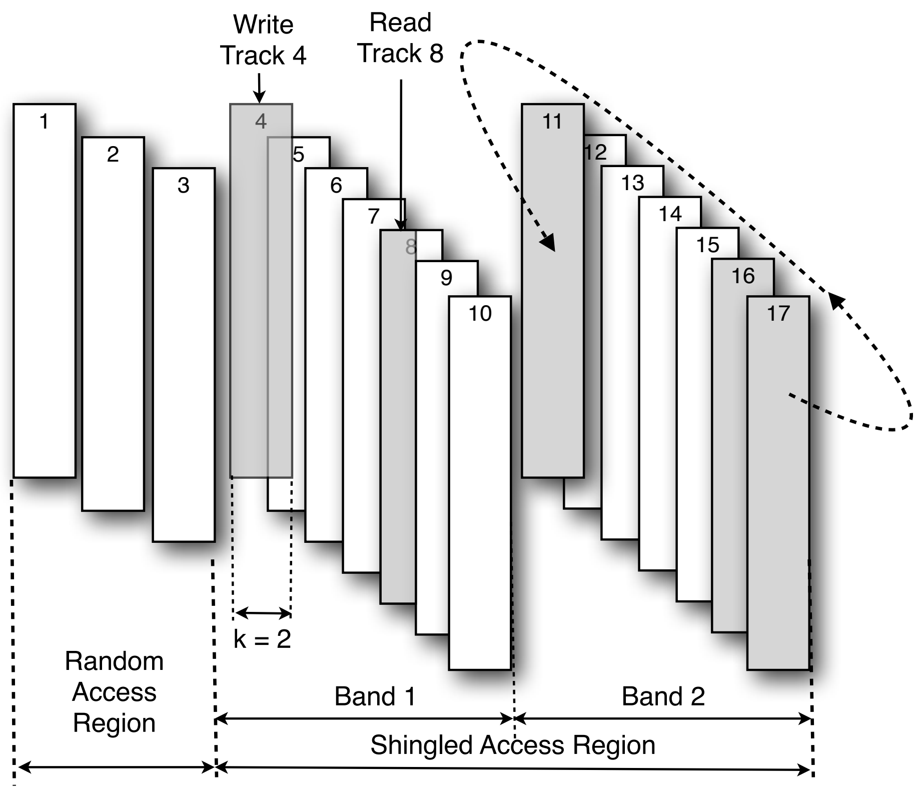


Figure 2.2: Shingled writing overwrites k read tracks. SMR disks can be divided into many append-only bands and a small random-access region, if desired.

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 2.2, 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 read tracks in the same area than traditional disks.

2.1.1 Role in Future Disks

Several upcoming disk technologies, all aimed to increase disk's areal density, also rely on SMR. Bit Patterned Magnetic Recording (BPMR) [108] works with patterned recording media, with distinct islands etched on the surface for each bit. BPMR's challenge lies in etching one island with well defined position and geometry per bit, and finding a way to write data to these islands. Greaves *et al.* have suggested adopting shingled writing to write data to BPMR's well placed islands [45].

Two Dimensional Magnetic Recording (TDMR) [108] also adapts shingled writing where the tracks are squeezed tighter together such that it requires the track's data to be decoded by reading multiple adjacent tracks using a not-yet-fully-developed two dimensional read-back process. Heat and Microwave Assisted Magnetic Recording (HAMR, MAMR) [108] require thermal assistance to write data to a higher anisotropy recording media that is more thermally stable. Shingled writing has a major role to play in almost all of the future disk technologies, and moving forward there is a high probability that it will be the *de facto* method to write data to disks.

2.1.2 Shingled Bands

The SMR disk can be divided into *bands*, each of which is a sequentially writable region made of consecutive tracks, to bound the region that needs to be written sequentially. As Figure 2.2 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 2.2, 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 [9]. If a band is used as a fixed log, the band is written 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 continue from the physical start of the band as shown in band 2 in the Figure 2.2. While writes can be directed to the head of the circular log, cleaning can move data from its tail. Circular bands 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.

2.1.3 Comparison to Similar Devices

With the SMR data management challenges in mind, this section presents a comparative device characteristic study with other well understood and well studied storage devices with similar challenges. This study will not only be helpful in understanding how unique the SMR challenges are, but will also serve as a guide in determining which other works are related to the problem in hand. Table 2.1 compares shingled disk with traditional hard disk and NAND flash.

The differences between traditional hard disks and SMR disks have been discussed in detail in previous sections. Both shingled disk and flash face update-in-place issues. To update a previously written page in flash, the entire erase-block containing the page has to undergo a read-erase-write cycle. Both performance and the lifespan of the device will get affected by such an operation. A block update in SMR will require a read-modify-write from the point of modification to either the physical or the logical (tail of the band's log) end of the band. Lifetime is assumed to be unaffected by reads and writes, as no literature on shingled writing informs us otherwise. But

	Traditional Disk	**NAND Flash**	**Shingled Disk**
Random Reads	Yes, but slow	Yes	Yes, but slow
Update-in-place	Yes	Can be done with a *read-erase-write* of an entire erase-block	Requires a *read-modify-write* from point of modification to either the physical or logical end of the band
Wear Levelling	Not required	Required	Assumed not required

Table 2.1: Comparison of device characteristics.

performance is affected by such an operation, though how much depends on how far from the end the point of modification is.

Writes to NAND flash write blocks are also sequential. Due to their similarities, it is convenient to view both NAND flash and SMR disks as devices with big sequentially writable blocks that need to be cleaned and freed before reuse. The key differences between these two devices that has often been ignored are: a) Random reads: SMR disks, unlike flash, faces penalties incurred by disk head arm movement and hence, random reads are slow, and, b) Wear leveling: is assumed to not be a concern for SMR disks, in accordance with existing works on shingled writing. SMR, thus shares some challenges with disk and some with flash and the overall data management challenge faced is unique from that of any other device.

2.1.4 Drive Types and Interfaces

The T10 committee has broadly classified SMR drives into Autonomous (Drive-Managed) and Host-Managed drives [19]. Feldman and Gibson [37] cover in-detail the three possible ways in which the SMR access restrictions could be handled: 1) a drive-managed SMR disk, one that will not require the host software to be changed, but will have performance degradation, 2) a host-managed SMR disk, one that will need new host solutions to manage a banded SMR disk, which will return an error if the shingled bands are not written to sequentially, and 3) a host-aware SMR disk, a combination of options 1 and 2, where new host solutions are needed, but the drive can handle occasional non-sequential band writes with internal remapping at some performance cost. Though standardization efforts are underway for all three types, it is not yet certain which of these drives would make it to the market.

We see three interface choices for SMR disks and Table 2.2 compares them to each other, and this section describes each of them in detail.

	Emulated Disk Block Interface	**Banded Shingled Device Interface**	**Object-based Interface**
Existing Interface	Yes	No	Yes
Existing File Systems	Yes	No	Yes
Layout Decisions	At firmware	At host	At firmware
Additional Resources	Yes	No	Yes
Aid for layout	Geometry knowledge	Object knowledge	Geometry and object knowledge
Device agnostic	No	No	Yes

Table 2.2: Comparison of SMR disk interface.

Emulated Disk Block Interface

In order to retain compatibility with host system software, new storage devices have always been pretending to be a traditional hard disk with traditional block interface. Modern hard disk drives with complex geometry have been reporting a fake default cylinder, head and sectors per track values like the old generation hard disks for the same reason. NAND flash devices also chose to emulate the existing hard disk interface for widespread adoption. In a similar trend, for easier device adoption, SMR disks can hide their differences through complex disk firmware and pretend to be a regular hard disk.

Data management approaches similar to SSDs can be implemented in the disk firmware to convert write requests into sequential writes. A translation layer can hide the SMR access restrictions and make existing file systems and applications that do not use a file system work without any change. The data layout process can make intelligent choices based on geometry knowledge. The additional reads and writes required for the background activities will cost less when dealt by the firmware than host induced reads and writes.

Sticking to very old interface for the sake of compatibility is restrictive. Even though the production cost would increase as more resources are required at the device to do the additional work, no additional features could be provided. And, though existing file systems and applications will work, their basic performance assumptions will no longer be true. Moreover, researchers have noted the shortcomings of the block interface for SSDs, and have suggested alternate interfaces for flash as well. Approaches such as *nameless writes* [131] try to reduce the cost of large indirection tables in SSDs by means of de-indirection. They propose a new interface that allows writing a data block without an address. The device can choose where to write the block and inform the address back to the host on a successful write. An object interface has also been suggested for NAND flash to provide both a more efficient design [60] and also to provide more advanced features [58, 59].

Banded Shingled Device Interface

The second approach is to design an interface tailored to shingled disks, one that will aid an SMR-aware file system to take care of data management at the host. Banding requires disk geometry knowledge, and hence, can only be done by the disk firmware. The banded disk interface may be designed to either just expose the predetermined band information, or provide commands to create and manage bands, and to read and write to the bands. The manufacturing cost will not raise, as no additional resources, such as more processing power and memory, are needed to support the above tasks.

Standardization of the new interface will take time and will be helpful only for SMR technology. Other upcoming disk technologies and storage devices will not benefit from the effort. SMR-aware file system development and stabilization is also a huge effort, further increasing the effort. Large enterprises that design their own file system best optimized for their own unique access characteristics [40, 14] are more likely to prefer this interface.

Object-based Interface

The object-based storage architecture [76] has been a topic of research for quite some time now. The motivation behind the architecture is to separate the device specific storage component of the file system from the user component and move it into the device instead. The interface must provide commands to create/manage variable sized objects, and read/write the objects from the Object Storage Device (OSD). The T10 OSD command set [127] was proposed as a standard OSD interface long back, but the interface hasn't been actively adopted and still has room for improvement. Several system software components like object-based file systems and device drivers are also available to support the interface.

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 object storage device with a Key-Value interface, Kinetic, the Ethernet key-value drives from Seagate [101]. The Seagate Kinetic Open Storage Platform has inspired Open Kinetic [86], an open source collaborative project under the Linux Foundation dedicated to create an open standard around Kinetic devices. The advantages of adopting the object-storage architecture for shingled disks are:

Hide access restrictions The shingled disk can hide the challenges associated with the technology and eliminate it from the host system perspective.

Efficient data management Object-based interface, being a richer interface, can aid the storage device to do better data management by providing more information about the stored data. Information that only the drive firmware possesses, such as the disk geometry knowledge could also be factored into the data layout decisions. If it were a hybrid drive, it can also utilize the available flash space efficiently say, to store all of its metadata. Disk idle time can be used effectively for background activities, such as garbage collection and data reorganization, without any unwanted data movement to and from the host.

Device agnostic The interface is device agnostic. The host no longer needs to know whether the device is a SMR disk-only drive or a hybrid drive or a flash device or a device with any new future technology.

Easier adoption The host operating system does not have to be changed every time the disk technology changes. Supporting all existing local and distributed object-based filesystems would be straight forward. Existing block-based filesystems can also be mounted on a pseudo block device that reroutes the block requests as object requests. In other words, the SMR OSD can act like a block-based translation layer too, if required.

Use in multi-tiered architectures Multi-tiered storage architecture, where storage devices with different performance and cost characteristics are grouped into a hierarchy of different tiers is becoming increasingly popular. An example multi-tiered system would consist of a performance tier made up of NAND flash devices and/or other NVRAM caches, a capacity tier made up of hard disk drives and a backup/archive tier comprised of hard disk drives and tapes. As an unified device agnostic interface can facilitate easy data movement across the tiers, shingled disks with an object interface can easily fit into these systems without any additional effort.

2.1.5 Alternate Data Management Approaches

In this section, we describe alternate data management approaches that have been proposed thus far, for SMR disks.

Read-Modify-Write

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 sectors 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 and rewrite all the sectors that were read in, as illustrated by Figure 2.3. The approach would work well for target workloads without lot of data modifications. For example, SFS [81] is an SMR-aware file system designed for personal video recorder systems/set-top-boxes that uses random access bands for storing file system metadata and shingled bands for storing file data. SFS does not expect a lot of data modifications in its target workload and takes the read-modify-write approach to perform modifications to the data. For other workloads, performance will obviously take a hit.

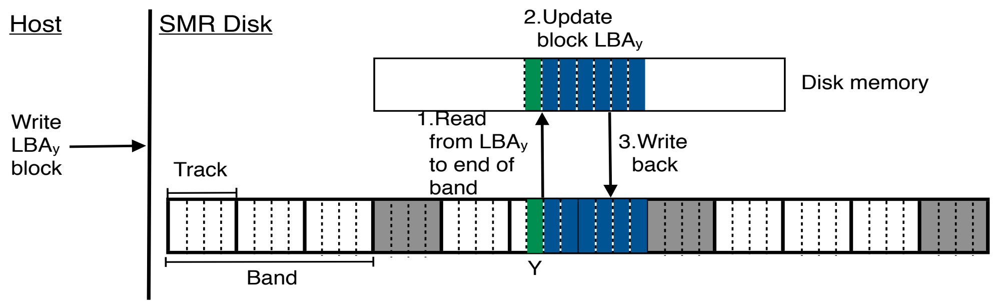


Figure 2.3: Read-Modify-Write increases the read and write amplification for every write operation.

Cassuto *et al.* [21] proposed indirection mechanisms using buffer bands, circular shingled bands and read-modify-write updates. As buffer bands are used to buffer the incoming writes, the data modifications can be handled in batches. The second architecture they propose divides the available LBA range into multiple subranges and assigns a small buffer band and an over provisioned circular shingled band for each subrange. They further divide the LBA subrange into read/write units called S-blocks and write the updated S-block to the tail of its circular shingled band. H-SWD [69] added hot and cold data separation to Cassuto *et al.*'s indirection scheme, but the effects on read performance wasn't measured. Their evaluation was thus incomplete, as only the write requests of block based traces were used for the evaluation, while all read requests were ignored.

Venkataraman *et al.* [117] suggested writing the tracks that are most likely to be updated (*e.g.,* the most recently updated track) in a shingled region at the end of shingled region, to reduce the number of tracks to be read and updated. Hall *et al.* [47] suggested dividing the disk into a buffer band and multiple circular shingled bands. They propose to apply the updates from the buffer bands by reading a track from the tail of the circular shingled band, performing the updates and writing the track to the head. He *et al.* [49] presented new static LBA-to-PBA address mappings to reduce the write amplification in read-modify-write in-place update SMR disks. They offer one-fifth of the capacity for smaller bands, and show that for about 50% space utilization of the remaining space, the write response time can be comparable to regular hard disk drives. But, they neither discuss the read-modify-write reliability copies, nor did they show the impact of mappings' impact on read performance.

Luo *et al.* [71] suggested a wave-like shingled recording to further increase the capacity and dividing the shingled bands further into segments of the same size in the radial direction, to achieve smaller updatable regions without sacrificing track space. Both the approaches are practically applicable only at the drive level, and the technological and performance implications of the approaches can be more accurately measured only when implemented at the drive level. The SSD in their hybrid SMR disk system has been used as a regular SSD read cache, aiding read performance as one would expect.

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 safely written to another location as well. However, this possibility has not been taken into account by most systems and has been ignored. Providing a consistent performance guarantee for different workloads is a challenge.

Skylight [5] combined software and hardware techniques to reverse engineer key properties of drive-managed SMR drives, and shed some light on the inner workings of the first generation drive-managed read-modify-write based SMR drive from Seagate.

SSD-like Approaches

SMR disk and NAND flash face similar issues with update-in-place. To update a previously written page in flash, the entire erase-block containing the page has to undergo a read-erase-write cycle, affecting both performance and the device's lifetime. Due to the above characteristics, it is convenient to view both NAND flash and SMR disks as devices with big sequentially writable blocks that need to be cleaned and freed before reuse.

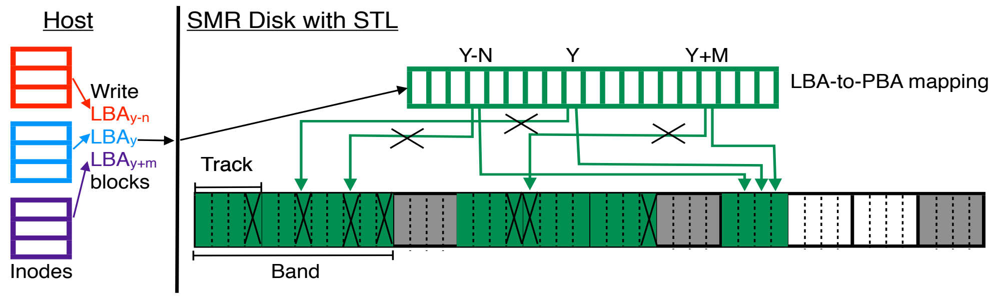


Figure 2.4: A shingled disk with a block translation layer in the disk.

Gibson *et al.* [41] proposed the use of a Shingle Translation Layer (STL) in shingled disk firmware similar to the Flash Translation Layer (FTL) in solid state storage devices, as shown in Figure 2.4. The log structured layout introduced by LFS has been adopted by NAND flash devices internally and has been suggested for SMR disks [84].

Amer *et al.* [10, 9] suggested various ways in which to adopt the layout for SMR disks, complete with garbage collection suggestions. But LFS layout has been more successful in flash than in disks and the adaptations suggested hasn't been evaluated for SMR disks yet. Jones *et al.* [55] proposed a band compaction algorithm that uses block write frequency to separate blocks to reduce data movement in an LFS-based SMR disk management layer, but has not evaluated the impact on read performance.

A key difference between these two devices is random read performance as shingled disks faces penalties incurred by disk head arm movement unlike NAND flash. A plain LFS-based block translation layer can change a sequential read into random reads, depending on the original order in which it was written. There will be a vast improvement in random write performance, since the random writes will be internally converted to sequential writes, but the improvements come at the price of potentially reducing read performance. For a workload with mixed reads and writes, there may not be overall benefit if the read performance is affected.

Matthews *et al.* [74] pointed out that read performance will be affected in LFS, since the reads were not being mostly served by caches as expected by the original assumption. They showed that data can easily be reorganized in the background along with garbage collection to improve the read performance. Both ALIS [52] and BORG [15] reorganized data at block level, based on workload detection. SDS [110] implemented additional functionalities at the disk by acquiring knowledge about the file system's on-disk data structures. These elaborate mechanisms will not be required in case of an object based system, but could be adapted in blind block-based indirection systems to improve read performance. Recently, Macko *et al.* [72] used data block back references to aid data reorganization in a write-anywhere file system.

Another major concern with the LFS layout has been garbage collection efficiency. The GC overhead is determined by how the data (both live and stale) gets distributed across bands. Seltzer [103, 104] showed that the cleaning overhead in LFS can significantly lower the system performance as cleaning can interfere with active I/O requests, making LFS unsuitable for workloads such as those dominated by random updates to a full disk with little idle time to clean.

Blackwell *et al.* [18] showed that simple heuristics based on the disk idle time can help perform LFS cleaning in the background, without affecting normal file access performance. Segregating hot and cold data in different segments has been illustrated to improve LFS cleaning performance by WOLF [123] and Hylog [126] and has been employed successfully in flash systems as well [51, 80]. Going one step further, PROFS [122] added disk characteristics based placement to LFS, all of which are applicable and can be adopted for SMR data placement and cleaning.

Update-out-of-place File Systems

Figure 2.5 shows an update-out-of-place file system that manages data on a banded SMR disk. Though there exists many update-out-of-place file systems, most of them rely on the ability to randomly write to any desired location on disk. For example, the Log-Structured File System 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. Though segments could be mapped to bands, all the issues with the LFS layout, discussed in the previous section, still apply. Adapting existing update-out-of-place filesystems is not straightforward, and requires a redesign to adapt to sequential writes to large regions on disk.

Suresh *et al.* [114] presented an SMR file system. Though they presented their solution as a file system, it was not designed to work as a general purpose file system. It is rather designed for use cases that writes to files only sequentially, never reopens a closed file for a write, and never rewrites a block in the file. HiSMRfs [53] is a general purpose filesystem that separates metadata from data, and stores the metadata in either a high performing non-shingled region, or, if available, in an SSD. The data blocks are appended in shingled regions and a garbage collector cleans dead data blocks. But the

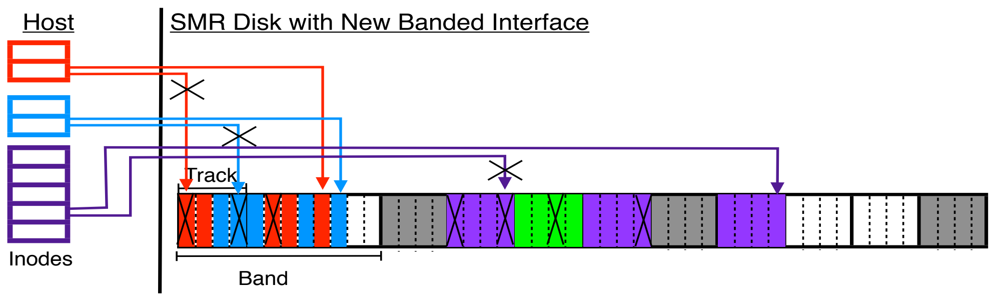


Figure 2.5: A new update-out-of-place file system for a shingled disk with a banded device interface.

evaluation is not extensive and does not highlight the tradeoffs and effects of their design choices, except the effects of storing metadata in SSDs.

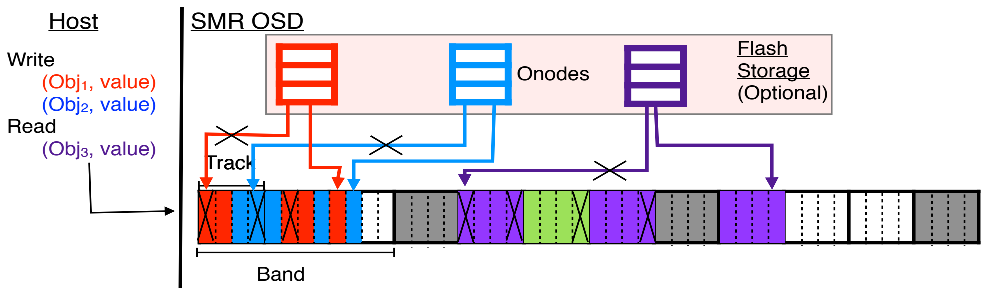


Figure 2.6: Shingled disk managed as an object storage device.

Shingled Disks as Object Storage Devices

The object-based storage architecture [76] has been a topic of research for quite some time now. The motivation behind the architecture is to separate the device specific storage component of the file system from the user component and move it into the device instead. Amer *et al.* [9] suggested the object storage model as an alternative for shingled disks, but did not expand further. A shingled disk managed as an object storage device is illustrated in Figure 2.6.

2.1.6 Metadata Structures

Traditionally, B-trees have been used by file systems to store file and directory metadata and the benefits of doing so are well known. B-trees will work well if stored in the random access bands. But if it is stored in shingled bands, the leaf node modifications will force modifications to all nodes till the root and traditional B-trees will no longer be ideal. Both index structures for storing the metadata and their on-disk placement in the shingled bands haven't been studied yet.

Ohad Rodeh [97] studied the issues in using B-trees in a Copy On Write (COW) system and proposed a COW B-tree variant. The COW B-tree can be placed in the shingled band, but has high write amplification and will pollute the band with stale data. Alternatively, the B-tree nodes can be stored using a translation table to tell where each node is physically located as is done by BFTL [129], but the nodes can get scattered across the disk. B^+-Tree (ST) [83] has two modes, a disk mode with a translation table and a log mode that organizes the index as transactional logs. At any time, parts of the tree are in disk mode and the rest in log mode and an online algorithm decides when to switch a node from one mode to the other based on the read/write cost.

μ-tree [57] is another flash tree structure that is similar to B^+-trees, where every leaf node is split into as many levels as the height of the tree and and all the nodes along the path from the root to leaf are packed in the leaf node itself. The goal is to reduce the number of nodes to be written on a leaf update. The number of read operations required during a lookup will be high since the tree's organization increases the height of the tree. Increasing the number of nodes to be read, while also scattering the nodes across the disk is highly unsuitable for SMR disks.

LSM-tree [85] contains a small in-memory B-tree and several append only B-trees on disk. It is highly write optimized, but the read amplification is unacceptably high. FD-tree [67] is similar to LSM-tree, except that a few levels of sorted runs of increasing sizes gets created when the head tree gets merged instead of the append only B-trees. Though they employ a fractional cascading mechanism to improve reads, performance will still be bad for disks as an index search has to go through multiple levels. LA-tree's [6] use of cascaded buffers to perform update operations in a lazy manner can be adopted for SMR disks. FlashB-tree [54] proposed mechanisms to avoid the high cost of B-tree node splitting, merging and rotation, and is also applicable for SMR disks.

2.2 Key-Value Storage Systems

Key-value stores have become a vital component of cloud computing applications and high performance web scale databases. The key-value interface, being a simple and versatile interface, is applicable to both the large distributed stores and the individual storage nodes that make up the large distributed stores. Being a device agnostic interface, it is suitable for all the different kinds of devices in the storage hierarchy, and as such serves as a unified model for all the layers in the hierarchy. Building distributed key-value stores that meet web scale demands and scale as demand rises is an active research area [22, 31, 65].

Here, we look only at key-value stores that follow a log-structured approach to writes, as it is a key requirement for SMR disks. FAWN-KV [12], FlashStore [29] and SkimpyStash [30] 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.

Disk-based key-value systems such as LogBase [118] that are built for write-heavy workloads and are not so much concerned about range query performance also adapt a LFS style approach. But systems that require both the write performance of logging systems and demand a decent range query performance typically go for the LSM approach. The LSM approach consists of a in-memory store that gets flushed to disk sequentially once full, resulting in multiple ordered (by key order or hash order as requirement demands) data stores. Periodically, in the background these stores get merged to form bigger, reordered data stores. bLSM [102] describes the problems associated with the LSM approach and proposes techniques to resolve them. bLSM serves as the backing store for the key-value systems PNUTS [24] and Walnut [23]. Other LSM based key-value systems include BigTable [22], SILT [68], Loris [116], BabuDB [113] and KVDB [105].

2.3 Motivation for Realistic Request Arrivals in Benchmarks

Many systems designed to solve real-world problems prove the worth of their solution through an evaluation framework that replays real-world workload traces. Though a good approach, such traces are not abundant and the ones that are available may not always be applicable. In such situations, benchmarks are used in the evaluation of different designs with the same goals, not just in academic research, but also in real world product promotions. Hence, standard benchmarks have to be representative of real world needs, modeled based on observed real-world workloads. Even though the workload's temporal characteristics are a big influencer on the system behavior, it has been mostly ignored by benchmarks.

Many works exist on building better key-value systems, but a trace-based evaluation is out of reach, as, to the best of our knowledge, no key-value workload trace is publicly available. The only published key-value workload study is that of an in-memory key-value caching layer, Facebook's Memcached deployment [13], and even if obtained will be unsuitable to evaluate individual key-value storage nodes, such as the Kinetic [101] disks, the ethernet key-value disks from Seagate.

In recent years, the Yahoo! Cloud Serving Benchmark (YCSB) [25] has emerged as the standard benchmark of choice for evaluating key-value systems. YCSB has been used both in the evaluation of large distributed key value stores [34, 35], and individual key-value storage nodes [68]. YCSB has also been used to generate representative data serving scale-out workload in the evaluation of modern processor limitations [38]. YCSB comes with a workload generator that is flexible in the selection of different mixes of operations and data sizes, and key selections based on different distributions. But its framework does not come with the flexibility to generate realistic request arrivals.

Storage devices have background tasks, such as scrubbing, cache de-staging, data migration across tiers and automatic backups, that need to co-exist with foreground request processing. Modern storage media, such as NAND Flash and Shingled Magnetic Recording disks, require log-structured data management approaches to overcome the media's inability to update data in-place. Such devices also come with background compaction (garbage collection and reorganization) processes that oftentimes interfere with incoming media access requests. Different designs handle the additional overhead in different ways, and are in need of an evaluation framework that does not bombard the system with requests continuously, but rather generates realistic arrivals with periods of both low and high activities.

2.3.1 State of the Practice

YCSB's constant-rate request arrival process is unrealistic and fails to capture real world arrival patterns. Even then, the state of the practice thus far has been to use YCSB in an unrealistic manner to evaluate many real world solutions that require realistic request arrivals. In this section, we will discuss a few of them, with the purpose to serve as motivation for realistic request arrivals in benchmarks.

ElastMan [8] is an elasticity controller for cloud-based elastic key-value stores, designed to automatically respond to changes in workload and respond to spikes and diurnal behavior. The controller automatically resizes an elastic service in response to changes in workload, in order to meet service level objectives at a reduced cost. But the evaluation is done with multiple YCSB clients, each generating a workload at a constant rate, and the variations including the spikes with various magnitudes and the diurnal behavior is provided by adding and removing YCSB clients.

Albatross [28] is a technique for live migration of the database cache and the state of active transactions in a multi-tenant database, designed to tolerate load variations while minimizing operating cost by low cost live migration of tenant databases. Zephyr [33] is also a technique to efficiently migrate a live database in a shared nothing transactional database architecture. Even though database systems serving cloud platforms must serve large numbers of varied applications (or tenants), both of them evaluate their techniques with YCSB with just a steady load increase.

Pisces [109] is a system for achieving datacenter-wide per-tenant performance isolation and fairness in a shared key-value storage system. The evolution is done by running multiple YCSB instances, one for each tenant, to mimic their expected workload. Most of the evaluation is done with constant demand tenants. A small dynamic workload section on mixing bursty demand and diurnal demand tenants with the constant demand tenants is presented, but doesn't elaborate on how these demands were generated.

All of the above systems would have benefitted tremendously if YCSB or a similar key-value benchmark had the ability to generate request arrivals similar to those observed in real world systems.

2.3.2 Related Work

Some simulation systems extract request inter-arrival time distributions from real-world systems or traces, and mimic the arrival pattern in the simulated traffic by sampling the inter arrivals from the configurable distribution parameter [32, 75, 4]. In TPC-W, a widely used traditional client-server benchmark, user arrivals are defined by a Poisson process. To rectify its lack of ability to produce burstiness, Mi *et al.* [77] injected burstiness into it, using a Markov-modulated process, based on the popular ON/OFF traffic models used in networking to create correlated inter-arrival times.

One of the methods we use to generate realistic arrivals, the b-model, a simple model to generate self similar, bursty traffic for a wide range of time scales, was first used by Wang *et al.*to generate self similar disk IO traces [124]. Hong and Madhyastha argued that there was no need to model self similarity at large time scales in disk traffic, as it is irrelevant for measuring disk response times and queuing behavior, and used the b-model to generate synthetic arrivals at short time scales [50]. But as we discussed earlier, realistic arrivals can be useful in evaluating many system functionalities that span across all storage media at different time scales, and it is important that benchmarks recognize the importance of the temporal characteristics of generated workloads.

YCSB++ [87] extended YCSB with a set of additional features that can be used in database advanced functionality performance testing and debugging, but does not address the lack of arrival variability. Features provided by YCSB++ and our work could be complementary to each other.

2.4 Mitigating Background Activity Overhead

Existing research has shown that idletime exists in storage devices and can be used to schedule general background traffic [42, 78, 130]. Idletime background work scheduling geared towards specific activities such as GC [18] and scrubbing [11] also exists. The goal of these systems is to minimize the impact on foreground traffic, while maximizing the background activity throughput. But, if the background activity has high priority and cannot wait for future (oftentimes inaccurately) predicted idletimes, the above methods are insufficient.

Freeblock scheduling [70, 115] predicted rotational latency delays between foreground disk accesses and tried to fill them with background tasks' disk accesses, to squeeze in background activities on disks with limited idletime. But used alone, such an opportunistic system can delay the movement of all live data in order from the segments being cleaned, thus delaying the completion and space reclamation of the current and future compaction runs. However, the approach can be complimentary to our mitigation technique and can be used while serving user read requests missed by an external cache.

Write-offloading has been used by researchers to create idle periods to spin-down disks, both by directing write requests to spun-down disks in a data center to other devices in the data center [82], and by redirecting I/O to and from a physically separate flash-based cache [17]. Gecko [107] is a recent log-structured design that chains together a small number of drives into a single log, and limits writes to one drive that is extensively cached to avoid any contention, while performing GC on other drives that are not being written to. Our work, instead, focuses on limiting the compaction contention on a single drive with limited resources through write-offloading.

2.5 Hybrid Storage Devices

Hybrid storage devices are devices that integrate a small amount of higher-cost media to a larger lower-cost media, with the intent of adding some of the characteristics of one media to the other, either to make up for the characteristic the other media might lack, or to simply provide overall enhanced efficiency. For example, HDDs offer high capacity at low cost-per-bit, has good sequential read and write throughput, but its random read and write performance does not meet the requirements of many applications. While NAND flash could offer the performance required, the high cost makes it an unsuitable alternative in many cases and in such cases, hybrid systems made of flash and HDDs become very attractive.

Hybrid storage devices, unlike hybrid storage arrays or hybrid tiered storage systems, are tightly coupled to each other and typically not addressable as two individual devices. The characteristics of these hybrid devices, such as read latency, write latency and power consumption, are determined by device's controller design. When managed by an external host, the host is either blind to the real nature of the device, optimizing with false assumptions, or use special commands like the ATA NVCache feature set [2], and do not make use the available resources to the fullest. Hybrid storage devices are best suited as self managing storage devices. For example, data management schemes for hybrid SMR disks could probably benefit from a tight integration with the in-built flash devices, that goes beyond just block caching, to reduce the compaction cost. But doing so requires modification of both the interface and the filesystem, to expose and utilize the internals, and such modifications every time a manufacturer decided to change the device's specification is very unrealistic.

Many systems use a small amount of NVRAM, such as NAND flash, MRAM, etc, to cache frequently accessed data stored primarily on a HDD [1, 79, 17]. HeRMES also maintains its write buffer and all of its metadata in MRAM, while also logging the metadata changes in the HDD [79]. Conquest stores all small files and metadata in battery backed DRAM and holds only the data content of large files on disk [121]. Flash-backed NVCache has also been used to backup queued I/O requests on flash until it can be scheduled at reach the disk at a later time, to reduce write latency [16], or to create artificial idle periods for a disk spin-down [17]. Combo drive [89] exposes both devices, by simply providing a single contiguous concatenated storage space, where the Flash memory is mapped to the beginning and the magnetic media is mapped to the end of that space.

Researchers have also proposed hybrid SSD devices to increase SSD efficiency. To improve performance, energy consumption, and flash lifetime of SSD cache devices, a hybrid architecture with a small-size Phase Change Memory (PCM) has been designed with the PCM serving as the write cache and the flash as the read cache [106]. NVMFS stores all hot data permanently on NVRAM without writing back to SSD, while also caching the relatively cold data temporarily in the NVRAM with another copy on SSD [93]. Kannan etal., propose using active NVRAM to enhance the memory capacities, to enable data processing at the storage nodes. [62], and to provide frequent, low overhead checkpoints in high-end machines [61]. FRASH uses byte-addressable NVRAM as both a storage, maintaining the metadata in NVRAM and a memory device, to harbor in-core data structures [56].

Chapter 3

SMRDB: Key-Value Data Store for SMR Disks

This chapter describes SMRDB, 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 works and enables high performance on traditional disks, as it would on a host-managed SMR disk.

3.1 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 a database engine that is designed to do its own data access and storage management, operating on a host-managed SMR drive without any drive remapping solution. SMRDB could also easily be adapted to run on the drive controller, and hence is also an object management layer supporting 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, as shown in Figure 3.1. Tablet servers, such as BigTable [22], HBase [48], and LogBase [118] belong to the first category and systems such as Dynamo [31] and Voldemort [119] are of the second. SMRDB could be used as a stand-alone 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.

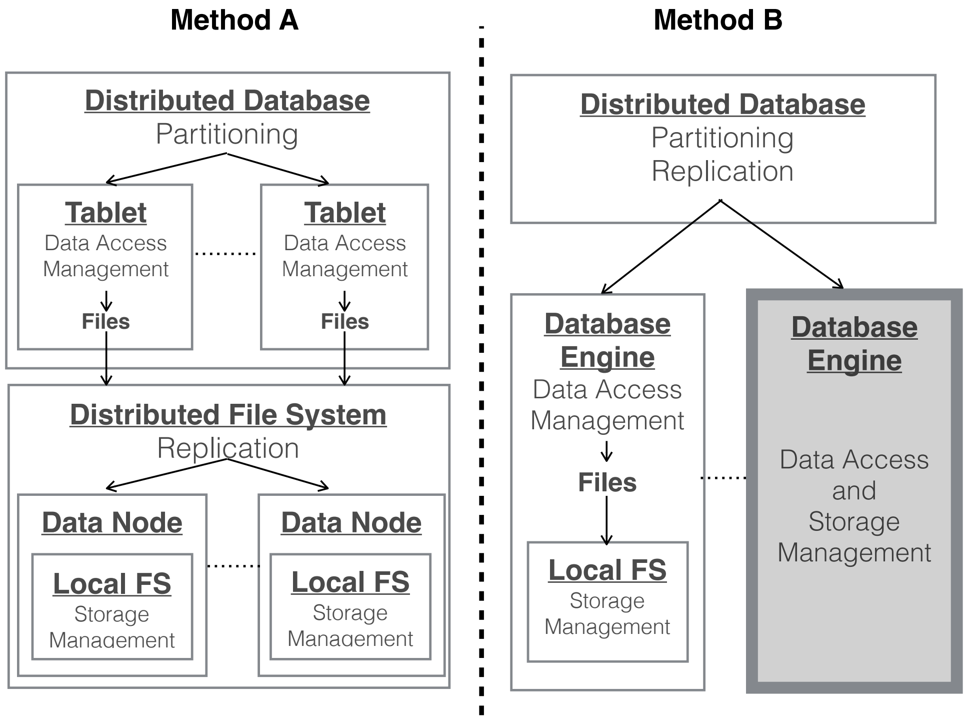


Figure 3.1: Distributed database architectures.

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 3.2. If presented with a traditional hard

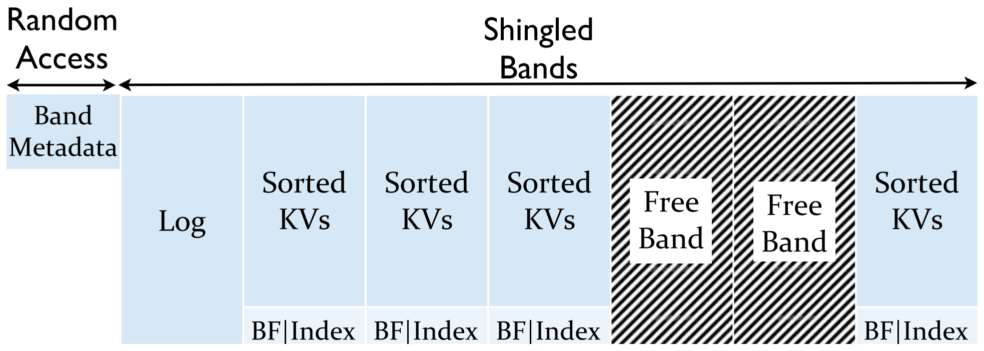


Figure 3.2: SMRDB's on-disk layout.

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 high level shingled band information, and not key-specific metadata. The KV pairs and related metadata are all stored in shingled bands.

3.2 Design

This section describes SMRDB's design in detail.

3.2.1 Log Structured Data Management

SMR sequential write restrictions call for log-structured writes to shingled bands. The Log-structured File System (LFS) [98] 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 [74]. 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.

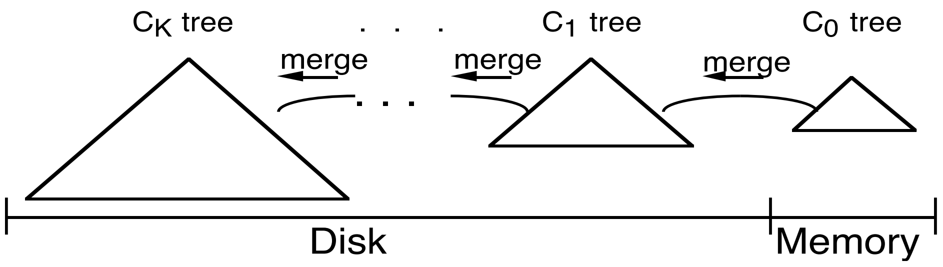


Figure 3.3: Log-Structured Merge tree.

Log Structured Merge trees [85] 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, as shown in Figure 3.3, 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. BigTable's [22] data access management architecture is LSM-tree based, as are systems such as HBase [48], Cassandra [65], and LevelDB [39].

To meet our scan performance goals, we will need aggressive data reorganization, as being done in LSM-trees. In the following sections, we present SMRDB's LSM-tree based design for SMR disks, based on an open source LSM tree based, embeddable KV database library, LevelDB [39]. LevelDB is an 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 [22] 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. Similarly, SMRDB maintains a memtable of size equivalent to an SMR band, and places each SSTable in an SMR band, as shown in the Figure 3.4.

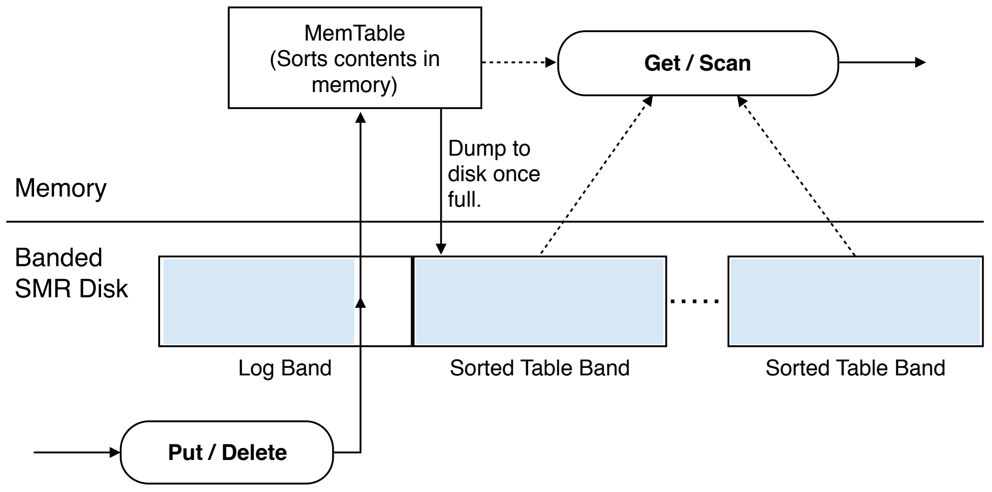


Figure 3.4: LevelDB based data access management.

3.2.2 Data Access Operations

SMRDB's primary data access operations are described in detail in this section.

PUT Newly inserted KV pairs are added to an in-memory *memtable*, which sorts them by lexicographic key order. When the size of the memtable reaches the capacity of a band, the memtable is flushed to an empty shingled band. All the keys in the band are added to a Bloom Filter (BF), and is stored, together with an index mapping the keys to their locations inside the band, 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, and is discussed further in Chapter 5. Bands are written-to only sequentially, and remain read-only until a background compaction process copies data out of a band and deallocates the band. Updating an already existing key is handled like new inserts and invalidates the previous entry for the key. 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. Tombstone entries are entries with markers denoting that the key has been deleted, thus invalidating the previous entries for the key. A read encountering a tombstone entry as the most recent entry for the key would report that the key was not found. The tombstone entries and the invalided entries are truly removed and the space freed when the bands containing them are compacted.

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 searches the index if found. The BFs filter out most of the bands and ais in search cost reduction. As BFs have a low memory footprint, most of them could be cached in memory and also reduce 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.

3.2.3 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 3.5. To achieve the ideal state, where the entire disk is ordered as in Figure 3.5, the bands are organized into levels and the background *compaction* process merges bands within or across levels.

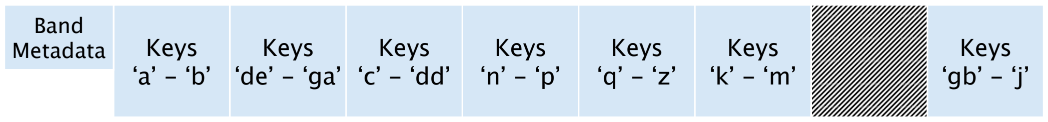


Figure 3.5: On-disk key ordering with physical ordering within a band and logical ordering across bands.

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.

LevelDB-style Organization and Compaction

Here, we describe an organization and compaction scheme for the bands similar to that of LevelDB.

Data Organization The bands are organized into multiple levels as shown in Figure 3.6. The level 0 (L0) bands are the results of *memtable* dumps, and the key range covered by each L0 band can overlap with each other. Each level has a size threshold and is increasingly bigger than the previous level. The bands in the other levels have a non-overlapping key range with respect to the other bands in the same level. Thus, when a key has to be read, the maximum number of bands that need to be searched is the number of L0 band + 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 an SMR band in our solution.

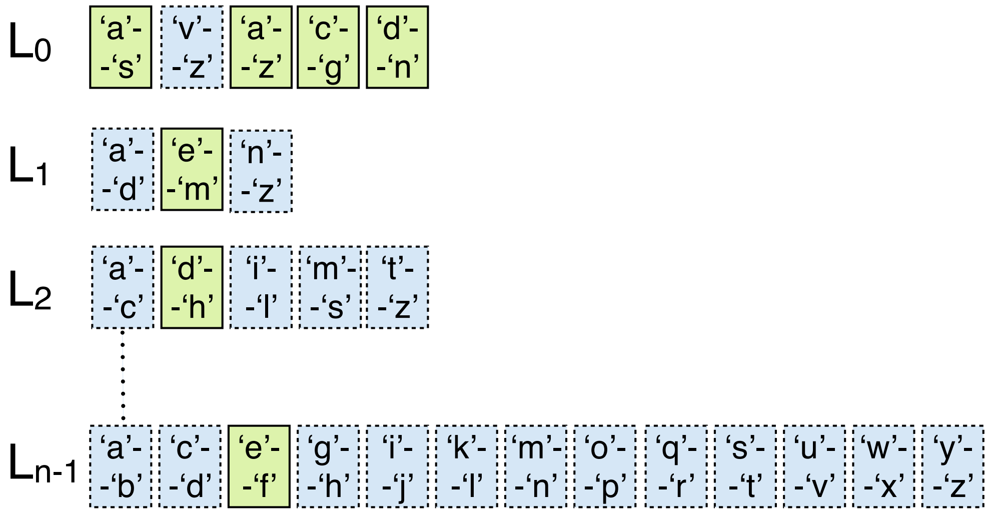


Figure 3.6: LevelDB style multi-level organization. A key starting with 'e' could be in any one of the 7 bands marked (in green) with a line border.

Compaction Periodically, selected bands in levels i and $i+1$ are *merged/compacted* to form new bands 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 bands in a level exceed the size threshold for the level, the level is chosen and a band 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 bands, as the bands chosen for compaction are read, reordered and rewritten entirely. When a band is chosen to be compacted, all the bands in the selected level and those in the next level, whose key ranges overlap the selected band's key range, are compacted. The strict non-overlap key range requirement reduces the number of bands that needs to be searched during a read, but increases the number of bands that get selected for compaction in the next level. Most of the time, this can result in copying bands without any major reordering of their contents. For example, if a band 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, though only one key was inserted to the $d - j$ range.

Recently, *stitching* has been proposed to solve the data movement problem in LSM trees [105]. 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 amount of data reads and writes that is required to keep them all ordered.

SMR-friendly Organization and Compaction

Our proposed SMR-friendly scheme organizes the bands into only two levels, a first buffering level which is the result of memtable dumps, and a second *mostly-ordered* level.

Band Organization As shown in Figure 3.7, the SMR friendly organization has only 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 3.5, 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.

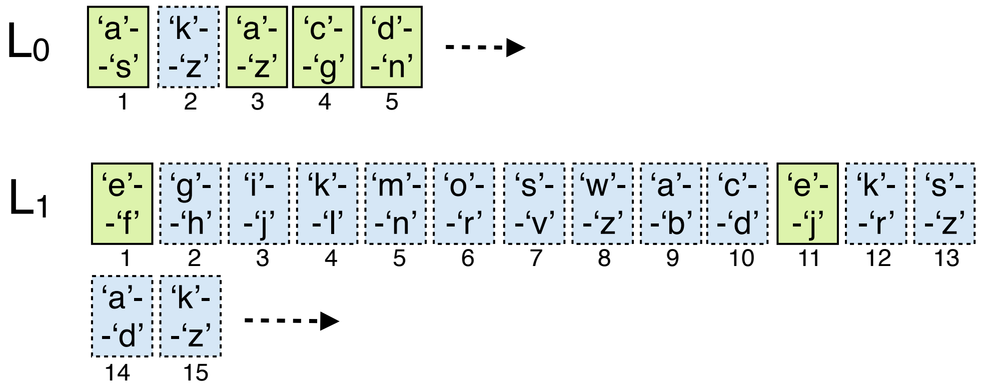


Figure 3.7: SMRDB's two-level organization. A key starting with 'e' could be in any one of the 6 marked bands.

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** [90] 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 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 3.7, 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 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 3.7, 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 an 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.

3.3 Evaluation

SMRDB's design is better evaluated on a raw banded SMR disk without any interference from a drive-managed SMR disk'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 [92], 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 an 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 an SMR disk.

3.3.1 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 3.8 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 written sequentially using direct IO to bypass the buffer cache, buffered IO using

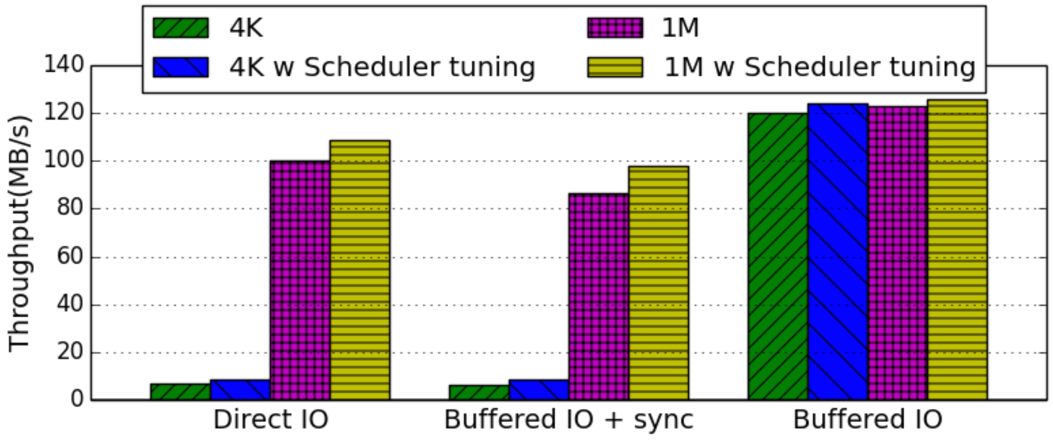


Figure 3.8: Raw sequential write performance of the test disk in the test environment.

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.

3.3.2 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.

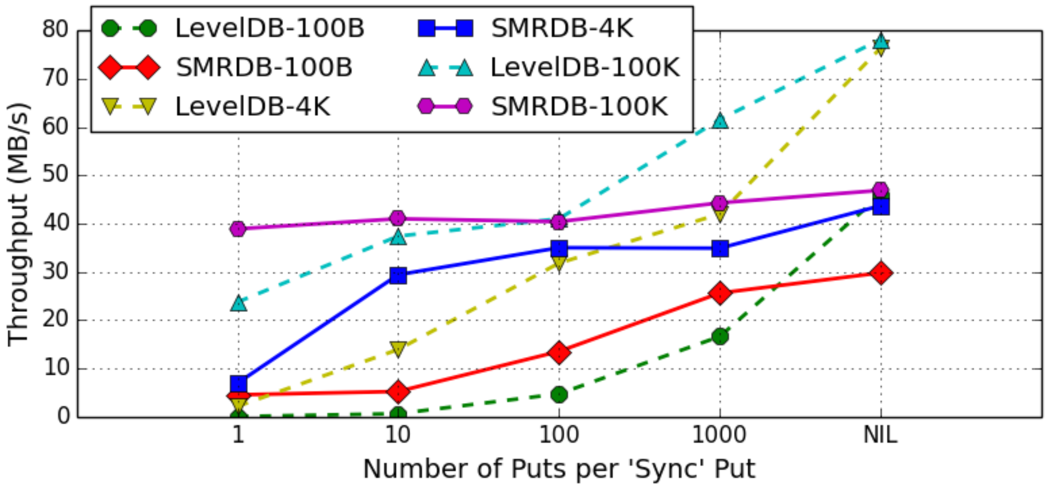


Figure 3.9: Sequential ‘Put’ performance for various value sizes (100 bytes, 4 KB, and 100 KB). Here, LevelDB-100B refers to LevelDB workload with 100 byte values, and so on. As SMRDB uses a small log buffer that gets flushed to disk more often, it is outperformed by LevelDB in cases with no or small amount of synced ’Puts’.

Sequential Writes

To measure the sequential write performances 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 were 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 3.9 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.

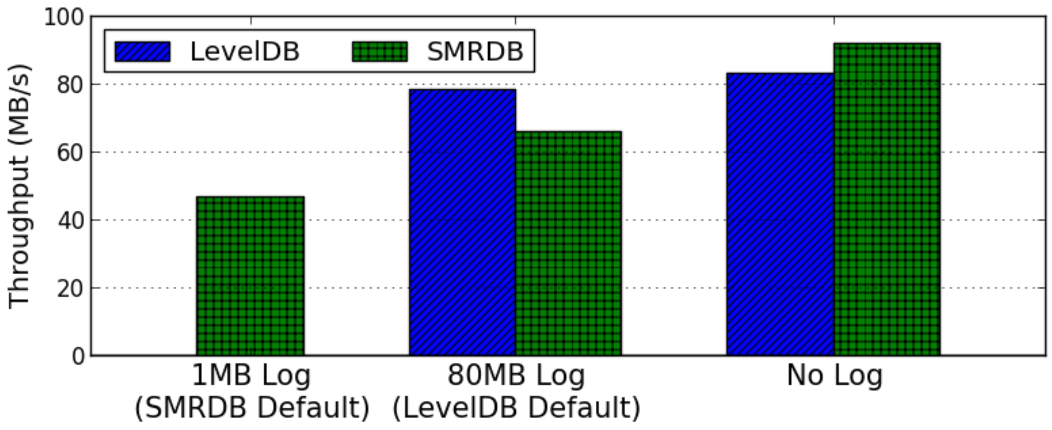


Figure 3.10: Logging writes everything twice and has a negative impact even for sequential writes.

LevelDB does not sync the log file if the user does not explicitly sync, and our smaller log buffer gets synced to disk more often. Our next experiment is designed to illustrate this fact, and the results of the experiment are shown in Figure 3.10. We took the case with the largest disparity, the no sync 100 KB value sequential writes, and performed the same test with a modified SMRDB that does not sync the log files, and a modified LevelDB and SMRDB that does not log. In LevelDB's case, there is only a small performance gap between the default and the no log write case. But with SMRDB, the performance doubles if no log is written, and the performance of the case where the log is written but not synced is between the two. Default LevelDB outperforms the no log sync option of SMRDB because the file system caching and disk buffering layers behave differently and the buffering pushes data to the disk more vigorously than the caching layer. SMRDB's performance can thus be greatly improved, if the log is moved to a separate lot device, such as an NVRAM device.

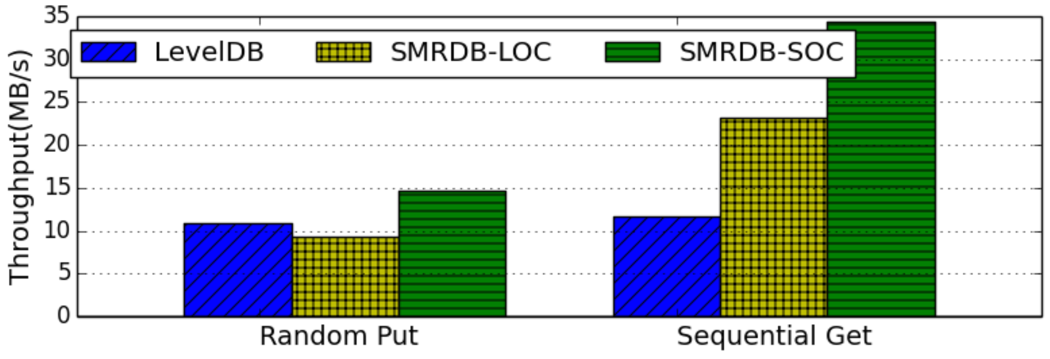


Figure 3.11: Performance of random 'Put' and sequential 'Get' after 'Put'.

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, SMRDB with LevelDB-style organization and compaction (denoted as SMRDB-LOC) and SMRDB with the SMR-friendly organization and compaction (denoted as SMRDB-SOC), both described in Section 3.2.3. Figures 3.11 and 3.12 show the overall performance of random inserts, compaction overhead and the sequential performance immediately after all the inserts have completed.

As seen in Figure 3.11, 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 3.12 shows the compaction overhead in greater detail. The graph on the left shows the time spent, in seconds, on compactions for all three cases, and the graph on the right shows how much data were read and written by the DBs (excluding the log write) for inserts that total

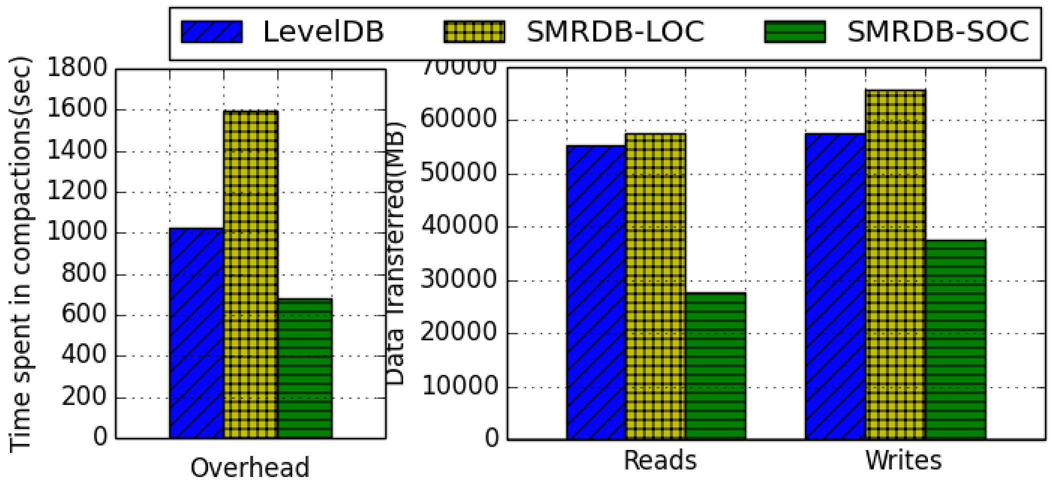


Figure 3.12: Background compaction overhead measured both in terms of amount of data read and re-written and time spent for writing 11 GB of randomly inserted data.

roughly 11 GB of data. The SMR friendly approach is clearly better than LevelDB style compaction. Further, as seen in Figure 3.11, 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.

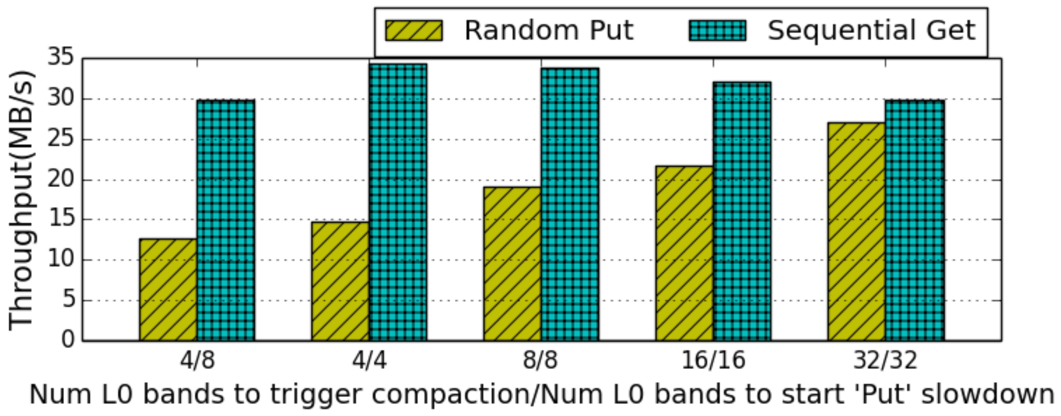


Figure 3.13: The insert performance increases when SMRDB's compaction is delayed.

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 3.13. 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.

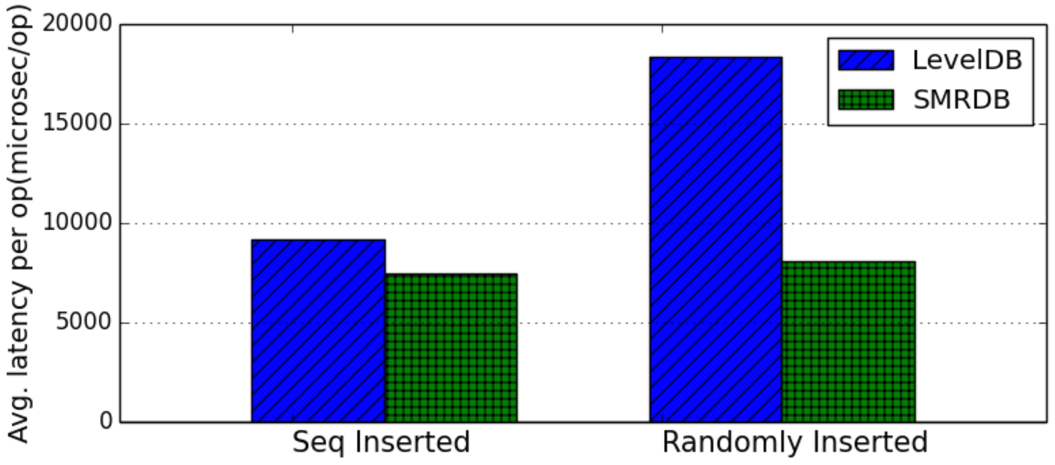


Figure 3.14: SMRDB has better random read performance, both after inserting the keys sequentially and randomly.

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 3.14. 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.

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.

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 3.15. 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.

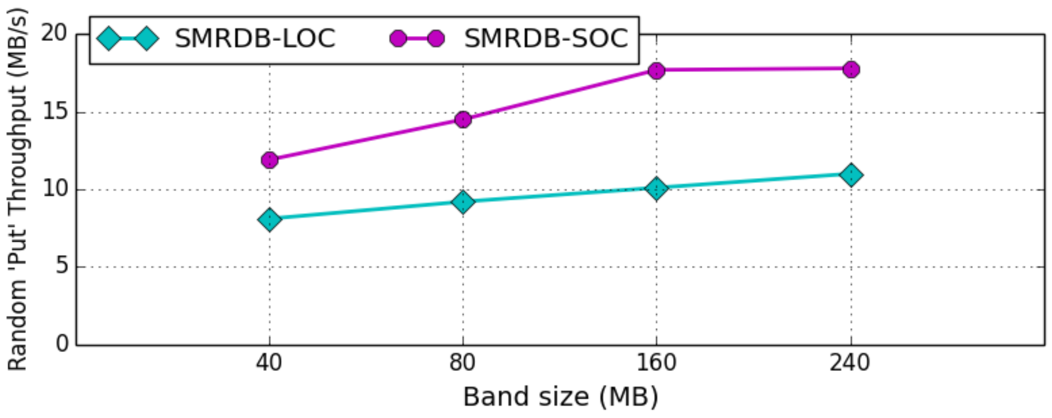
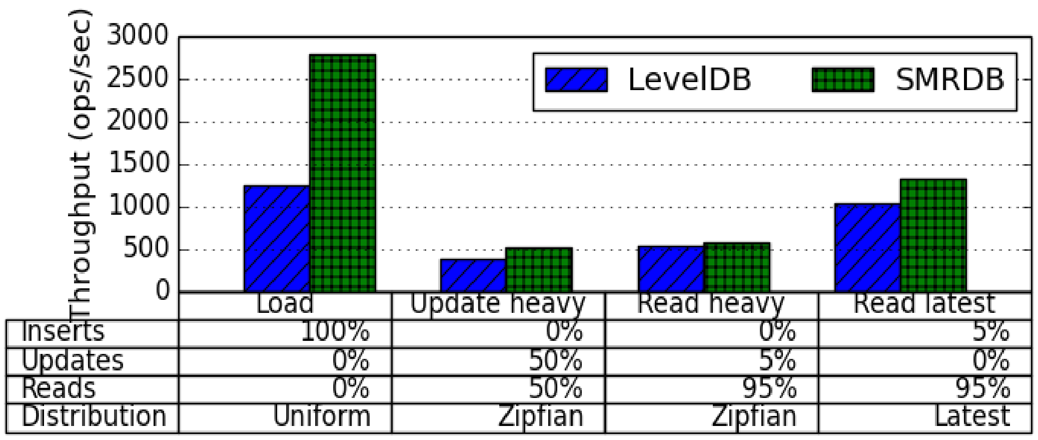


Figure 3.15: Bigger band sizes have better performance.

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.

3.3.3 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) [25], 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).



	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 3.16: SMRDB consistently performs better than LevelDB in various benchmarks, that are part of the the YCSB Suite.

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 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 performed 200,000 operations each to record the performance. Figure 3.16 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.

We expand on the load phase performance further, in Figure 3.17. 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 than 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

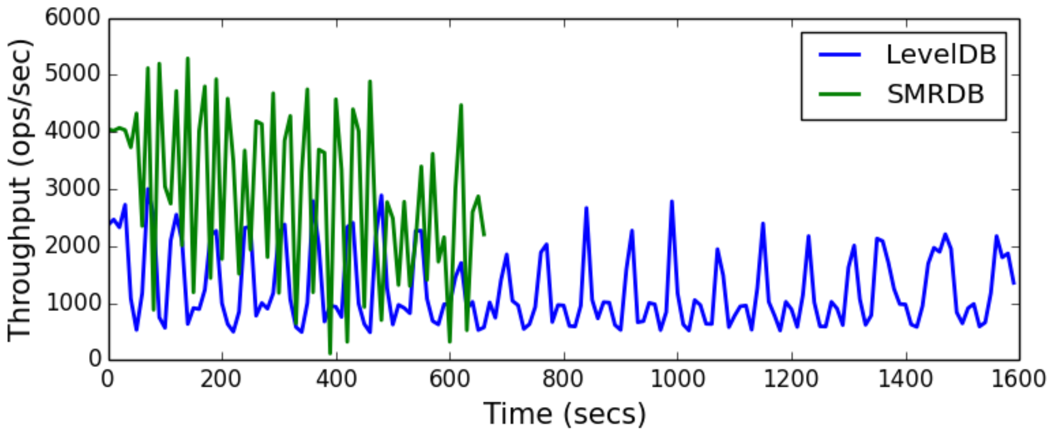


Figure 3.17: Performance of YCSB's load phase, which inserts uniformly distributed keys, shown over time.

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.

3.4 Summary

We presented SMRDB, a key-value database engine for SMR disks, in this chapter. 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.

Chapter 4

Realistic Request Arrival Generation

In this chapter, to aid in a more realistic evaluation of SMRDB, we classify temporal patterns based on existing workload studies on disk, filesystem, key-value system, network, and web traffic, since they all exhibit some common temporal properties such as burstiness, self similarity, long range dependence, and diurnal activity. We show that the commonly observed traffic patterns can be modeled using the three categories of arrival processes: a)Poisson, b)Self similar, and c)Envelope-guided processes. The three categories presented are a necessary and sufficient set of request arrival models that all storage benchmarks should provide.

The Yahoo! Cloud Serving Benchmark (YCSB) has emerged as the standard benchmark for evaluating key-value systems, and has been preferred by both the industry and academia. Though YCSB provides a variety of options to generate realistic workloads, like most benchmarks, it has ignored the temporal characteristics of generated workloads. YCSB's constant-rate request arrival process is unrealistic and fails to capture the real world arrival patterns. This chapter also describes our modified YCSB, one that generates workloads based on all three models, and shows the effect of realistic request arrivals through a vanilla LevelDB evaluation.

4.1 Request Arrival Process Models

Request arrival process is a stochastic process, and most of the time it is strongly correlated to itself. Autocorrelation is the cross-correlation of a time series with itself, and is a measure of whether a workload is correlated to itself or not. The autocorrelation function (ACF) can be used to measure the similarity between the original arrival time series and the same time series shifted by some time delay, as a function of the time lag between them [36]. In other words, the ACF shows whether the request interarrivals at any point of time is dependent on its previous values, or is independent. The ACF of a stochastic process $X = (X_1, X_2, X_3, ..)$ with mean μ and variance σ^2 at lag k is given by

$$r(k) = \frac{\frac{1}{n-k}\sum_{i=1}^{n-k}(X_i - \mu)(X_{i+k} - \mu)}{\sigma^2}$$

The above function is normalized and would result in values between $[-1, 1]$. When lag is 0, the series is compared to itself unmodified and the ACF will have the highest value 1. Positive ACF values mean that the random variable has a high probability to be followed by another variable of the same order of magnitude, while negative ACF implies the inverse.

Real requests do not arrive at a constant rate with a fixed time interval between them. It is important for a benchmark's workload generator to offer a variety of choices, that are both realistic and configurable, to match the workload a user has in mind. In this section, we categorize the request arrival process into three categories: a)*Poisson*, b)*Self similar*, and c)*Envelope-guided*, and argue based on evidence from existing web, disk, file and network IO studies that the categories presented are both necessary and sufficient to represent the real world needs.

4.1.1 Poisson Process

A Poisson process is a simple and widely used stochastic process for modeling arrival times. Requests can be modeled as a Poisson process if the request inter-arrival times are truly independent and exponentially distributed. The ACF of a poisson process is usually low and close to zero even at lag 1. Unless the inter arrivals are truly uncorrelated, the Poisson process is an unsuitable choice. Research shows that most arrivals are correlated, and cannot be modeled accurately by a Poisson process [88]. Nevertheless, when many different kinds of independent workloads are run on a system, the resulting traffic could look like a Poisson process.

Cao *et.al.* [20] studied the internet traffic and found that as the rate of new TCP connections increases, arrival processes (packet and connection) tend locally towards Poisson, and that time series variables (packet sizes, transferred file sizes, and connection round-trip times) tend locally towards independent. They concluded that the cause of the nonstationarity is superposition: the intermingling of sequences of connections between different source-destination pairs, and the intermingling of sequences of packets from different connections. Similar behavior can be expected in web scale cloud scale key-value workloads too. Hence, we have chosen the Poisson process as our first category.

4.1.2 Self Similar Process

Self similarity means the series look similar to itself at different time scales. Self similar workloads typically include bursts of increased activity, and similar looking bursts appear at many different time scales. A Poisson process too looks bursty at smaller time scales, as other processes following a long-tailed distribution do. But when aggregated and viewed at higher time scales, it gets smoothened, whereas aggregating streams of self similar traffic typically intensifies the self similarity instead of smoothening it. Long range dependence means the series is correlated to not just its immediate past, but also its distant past. So, the ACF of a long range dependent process decays slowly. Self similarity and long range dependence, though separate phenomena, are typically observed together.

Many real life observed workloads are both self similar and long range dependent. Self similarity has been observed in WWW traffic [27], Ethernet local area network (LAN) traffic [66], file-system traffic [46] and also in disk-level I/O traffic [43, 96]. Further, researchers investigated a number of wide-area TCP arrival processes [88], and concluded that even if the finite arrival process derived from a particular packet trace does not appear self similar, if it exhibits large-scale correlations suggestive of long-range dependence, then that process is almost certainly better approximated using a self similar process than using a Poisson process. Hence, we believe benchmarks should also provide the facility to model request arrivals based on a self-similar process.

Hurst parameter, H, is the exponent that describes the cumulative expected deviation from the mean after n steps in a random walk [36]. Higher values of H are the result of stronger long-range dependence. The Hurst parameter is often used to quantitatively measure the self similarity of a time series. H is equal to 0.5 for a Poisson process, and is in the range 0.5-1 for a self similar process. A variety of methods exists to estimate the value of H of a time series, and for a thorough description of the most popular methods we recommend referring to Feitelson's book on workload modeling [36]. We use Selfis [63] to compute the Hurst parameter of the generated inter-arrivals using five different estimation methods.

Feitelson also describes in detail a variety of methods to model self-similarity [36]. In our work, we generate self-similar traffic using the *b-model*, a simple model to generate self similar, bursty traffic for a wide range of time scales [124]. The model requires a single characteristic parameter, bias b. The idea is to split the entire amount of work recursively into two portions in a proportion determined by the bias b, similar to Figure 4.1. Thus, the total number of operations N is divided into bN and $(1-b)N$, and whether the first half of the divided time-period receives bN operations or $(1-b)N$ operations is determined randomly. Such recursive work division generates self similar traffic with high local irregularity, where b closer to 1 generates traffic with high

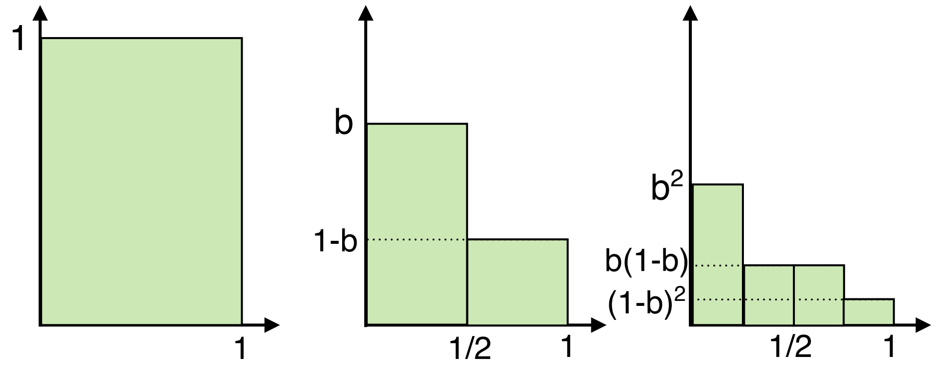


Figure 4.1: Multiplicative cascading generation of b-model.

irregularity and $b = 0.5$ results in uniform traffic.

4.1.3 Envelope-Guided Process

Gribble *et al.* showed that high-level file system events exhibit self similar behavior, but only for short-term time scales of approximately under a day [46]. By examining long-term file system trace data, they showed that high variability and self similar behavior does not persist across time scales of days, weeks and months, and concluded that the file system traffic is well represented by a self similar process for short time scales, but is unsuitable for longer time scales.

At longer time scales, many workloads exhibit a clear diurnal pattern [46, 13]. Karagiannis *et al.* show that periodicity can obscure the analysis of a signal giving partial evidence of long-range dependence [64]. Also, Akgul *et al.*showed that periodicity-based anomalies affect Hurst parameter estimation, causing unreliable H estimates, and if periodic anomalies exist they should be removed before estimation [7]. The presence of periodicity could have led to the conclusion by Gribble *et al.*that a self similar process is unsuitable for long time scales.

If the traffic is periodic and exhibits a pattern such as a daily/weekly activity cycle, then the ACF plot of the arrival process does not decay slowly as it does for a self similar process. Instead, the ACF oscillates between positive and negative values, corresponding to the periodicity of the original time series. The autocorrelation function can clearly extract and demonstrate periodicity even from much noisier data. As diurnal cycle is common in storage workloads, it is vital that benchmarks come with the option to generate such traffic.

The observed self similarity at smaller time scales can also be attributed to traffic conforming to heavy tailed distributions such as the Pareto distribution. Heavy tailed distributions can also result in larger H values similar to the long range dependent process. As summing heavy-tailed random variables do not average out, but rather lead to a heavy-tailed sum, when a process composed of heavy-tailed samples is aggregated, we will get a process with similar statistics. Paxson *et al.*showed that 'pseudo self similar' processes, arrival processes that appear to some extent self similar, could be produced by constructing arrivals using Pareto interarrivals, and that the generated traffic has large-scale correlations and the Òvisual self similarityÓ property, though the traffic generated is not actually long-range dependent (and thus not self similar) [88].

Roughan *et al.*noted that Internet backbone traffic has both daily and weekly periodic components, as well as a longer-term trend, and superimposed on top of these components are shorter time scale stochastic variations [100]. They modeled such traffic by segmenting the traffic into a regular, predictable component, and a stochastic component. Our final category, the envelope-guided process is similar to their approach. A predictable component such as a sine wave function determines the arrival rate per time interval, and the actual inter arrivals are generated from a secondary distribution such as the Pareto distribution. While the overall arrival rate is determined by an arrival rate function, the burstiness of the arrivals is determined by the secondary distribution selected.

4.2 YCSB Modifications

The Yahoo! Cloud Serving Benchmark is a client program, written in Java, that is designed to generate requests conforming to user-specified workload. YCSB client architecture, as shown in Figure 4.2, has a thread-safe workload generator that generates requests according to user specifications in a workload configuration file, a workload executor to execute the generated requests, a database interface layer to connect with and pass requests to the database, and a separate thread to periodically collect and report the status. By default, the workload executor runs a single thread, but is configurable and can be increased by the user. Each executor thread gets the request from the thread-safe workload generator and sends it to the database through its own instance of the database layer.

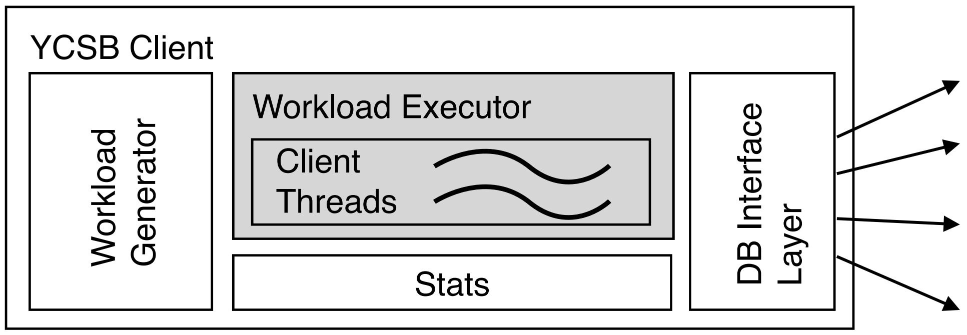
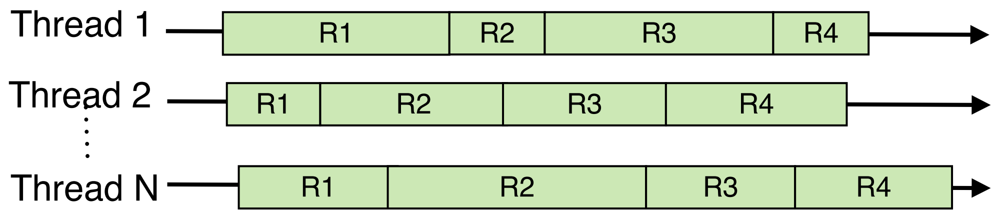


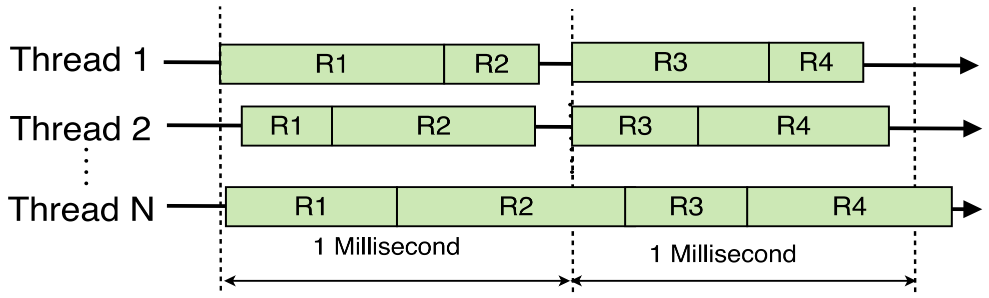
Figure 4.2: YCSB client architecture. The workload executor drives multiple threads to send requests generated by the workload generator to the database.

The threads perform back-to-back synchronous IO as shown in Figure 4.3a. Each thread sends a request to the database, waits for the response and immediately sends the next request once the previous response is received. When configured to run with multiple threads, the total number of operations to be performed is equally divided among the threads and each thread runs its share of operations in the same synchronous fashion, in parallel. As shown in Figure 4.3a, the threads start execution at slightly different times to avoid hitting the database at the same time, and after that the time requests are sent depends on previous completions.

To control the load offered to the database, the threads come with the ability to throttle the rate at which requests are generated. When a target arrival rate is specified, after every request the thread monitors the number of requests generated



(a) By default, each thread sends requests, waits for the completion of the sent request, and immediately sends the next request.



(b) If user specifies a target load resulting in say 2 requests per millisecond per thread, requests may be delayed depending on whether 2 request per millisecond has been completed.

Figure 4.3: Request execution process in YCSB.

until then and the elapsed time, and sleeps if necessary to maintain the target arrival rate. The resulting request execution process looks as in Figure 4.3b, in which the user specified target throughput results in 2 requests per millisecond per thread. If the target specified is low, the threads could all hit the database at the same time, but some may not due to timing inaccuracies resulting from sleep.

We have implemented three inter-arrival time generators as per the three models described in the previous section. Each YCSB client is designed to have its own inter-arrival time generator that determines the arrivals for the generated traffic. For a Poisson process, an exponential distribution is used to generate the inter-arrival times. For a self similar process, we implemented the b model and the bias b is user configurable. For the envelope-guided process, we have implemented a sine wave function, the shape of which is user configurable, which is used to determine the request arrivals

Table 4.1: Additional Configuration Parameters

Parameters	**Description**
arrivalgenerator	Specifies the generator to use. Accepts *constant*, *poisson*, *self similar*, and *diurnal.*
ss.bias	Specifies bias b in the b-model, used to generate self similar traffic.
diurnal.modulation	Specifies the diurnal cycle modulation.
diurnal.cyclelength	Specifies length of the diurnal cycle in minutes.
diurnal.distribution	Specifies the distribution to use for the stochastic variation.
diurnal.distribution.shape	Specifies the distribution's shape parameter.

per second. The actual inter-arrivals for the envelope-guided process are obtained from a secondary configurable distribution, such as the Pareto distribution. The envelope function could take a number of forms as per the need, and our work could be extended to include more patterns, as well as more secondary distribution choices.

We assume that the request sizes are independent of arrival times, as the only published key-value workload study found that the request arrivals are not correlated to the request sizes. Thus, no modifications to YCSB's workload generator was necessary. Our changes include new additions to the configuration parameters, to specify the choice of inter-arrival time generators, and shape parameters for specific generators, and modifications to the workload executor, to utilize the inter-arrival times generated by specified inter-arrival generators. The additional parameters we introduced and their descriptions are listed in Table 4.1

We noticed that, even with nanosleep and big-resolution timers, sleep does not always wake up as instructed and gives rise to lot of timing inaccuracies. Busy wait of all available threads is also out of the question, due to the high CPU overhead. We redesigned the workload executor, as shown in Figure 4.4, to facilitate generating bursts of IO activity at specified time intervals. We utilize Java's thread pool functionality to have a number of threads active at any given time. Though the thread pool provides

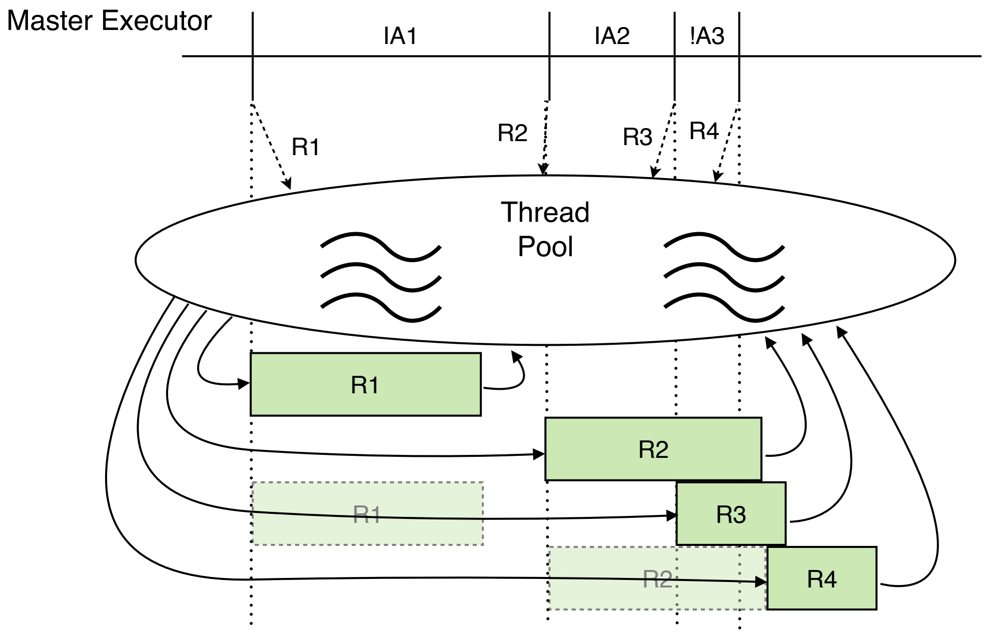


Figure 4.4: Our modifications to the workload executor to facilitate realistic arrivals. If threads in the pool are unavailable either due to configuration or slow response, there could be delay as in R4.

its own task queue and can pick threads once they are available to serve other tasks, we found the automatic detection of thread availability to be slower. So, we maintain our own thread queue, to which we add a thread once it is done servicing a request, and instruct the thread pool to execute the first available thread in our queue when needed. The master workload executor obtains the inter-arrival times from the inter-arrival generator, busy waits until the next request is due to be issued, and then issues the request using the first available thread.

4.3 Evaluation

In this section, we evaluate the accuracy and the effectiveness of our generators. After a brief description of our experimental setup, we generate traffic using a variety of configurations, and show that the generated traffic conforms to the specified arrival processes, both visually and empirically, *via* illustrations and statistical analysis of the generated traffic. Finally, we demonstrate the usefulness of realistic arrivals by evaluating a state-of-the-art key-value embeddable database library under all three models of request arrivals.

4.3.1 Experimental Setup

The evaluation was done on a Fedora 21 linux machine that has a quad-core 3.30 GHz Intel(R) Core(TM) i5-3550 processor with a 128 KB L1 cache, 1 MB L2 cache, and a 6 MB L3 cache, and 16 GB of RAM. For the application demonstration, the database evaluated was an embeddable database and to be able to connect and communicate with YCSB, LevelDB [39] was used with the MapKeeper [3] server. Both the YCSB client and Mapkeeper server were run on the same machine. The database was stored on a separate dedicated 160 GB single platter Seagate SATA disk drive running ext4 filesystem.

4.3.2 Realistic Arrival Visualization

For evaluating the arrival characteristics of the generated traffic, we ran the YCSB client against the YCSB's placeholder database, the *basic* database. The basic database receives all requests, does nothing, optionally injects delays, and reports a successful completion. We modified it slightly, to log the requests received with a timestamp. Throughout this subsection, all our experiments specified a target request rate of 10,000 operations per second, for better visual comparison of the different traffic generated.

Figure 4.5 shows YCSB's original behavior when executed with 1, 4, and 8 threads. We can see from the bottom graph that the arrival rate mostly remains between 9990-10010 operations per second, and the minor variations are typically a result of sleep inaccuracies. The top two graphs zoom in on a small interval of time, 60 seconds and 100 milliseconds. The arrivals at the millisecond interval, the topmost graph, shows the number of arrivals oscillating, and is particularly evident in the single threaded case.

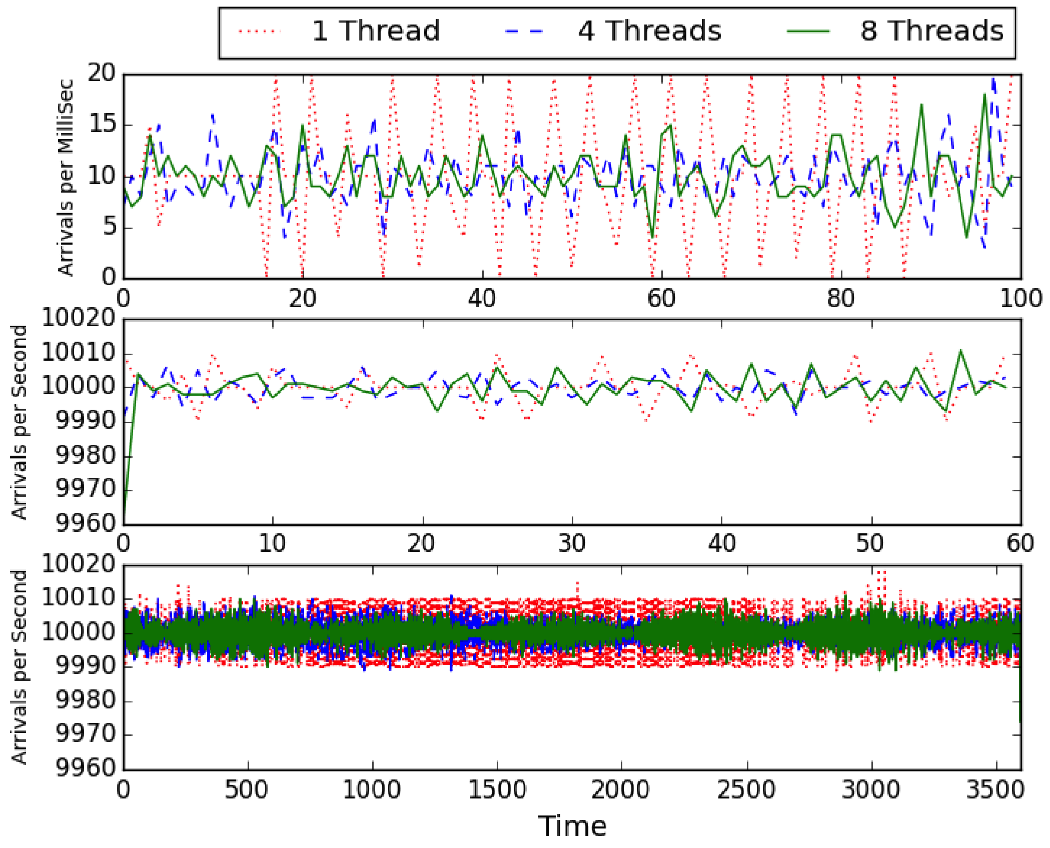


Figure 4.5: Original YCSB request arrival pattern for a target rate of 10,000 operations per second.

This is because the basic database does nothing and returns immediately and the client thread performs all IO at a time and then sleeps. But having multiple threads smooth them out as different threads are executing and sleeping at different times.

Figure 4.6 illustrates the traffic generated when configured with a constant arrival process and a Poisson arrival process by our modified workload executor. The constant arrival process illustrates how our framework is not subject to the sleep related inaccuracies seen in the original YCSB, and is able to send requests at generated intervals precisely. As discussed earlier, the Poisson process may look bursty at smaller time scales, but when aggregated gets smoothened.

Figure 4.7 shows self similar arrivals generated using the *b*-model. Traffic bursts can be clearly seen at all time scales, minutes, seconds, and milliseconds, and

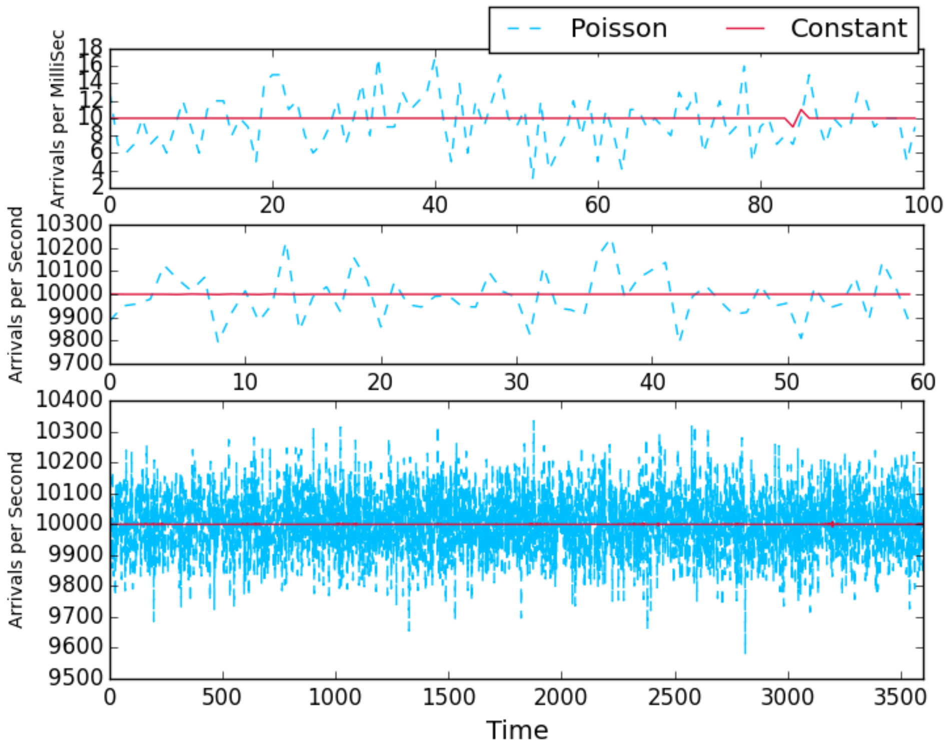


Figure 4.6: Traffic generated by our modified workload executor for a target rate of 10,000 operations per second, configured with a constant and Poisson request arrival process.

aggregation does not smooth the traffic as in the Poisson process, as described earlier. As noted before, the bias b, which is configurable, determines the burstiness of the generated traffic. A bias equal to 0.75 creates more bursts than does a 0.65 bias.

The generated envelope-guided arrivals can be seen in Figure 4.8. The envelope function here is a diurnal cycle with the stochastic variations provided *via* Pareto inter-arrivals. α is Pareto's shape parameter and determines the variations introduced in the traffic. As seen in the figure, $\alpha = 1.9$ generates bursts at smaller timescales and smoothes out when aggregated, similar to a Poisson process. But $\alpha = 1.1$ introduces lots of variations and generates bursts at different timescales. This traffic with periodicity is not really self similar, but behaves like a self similar process.

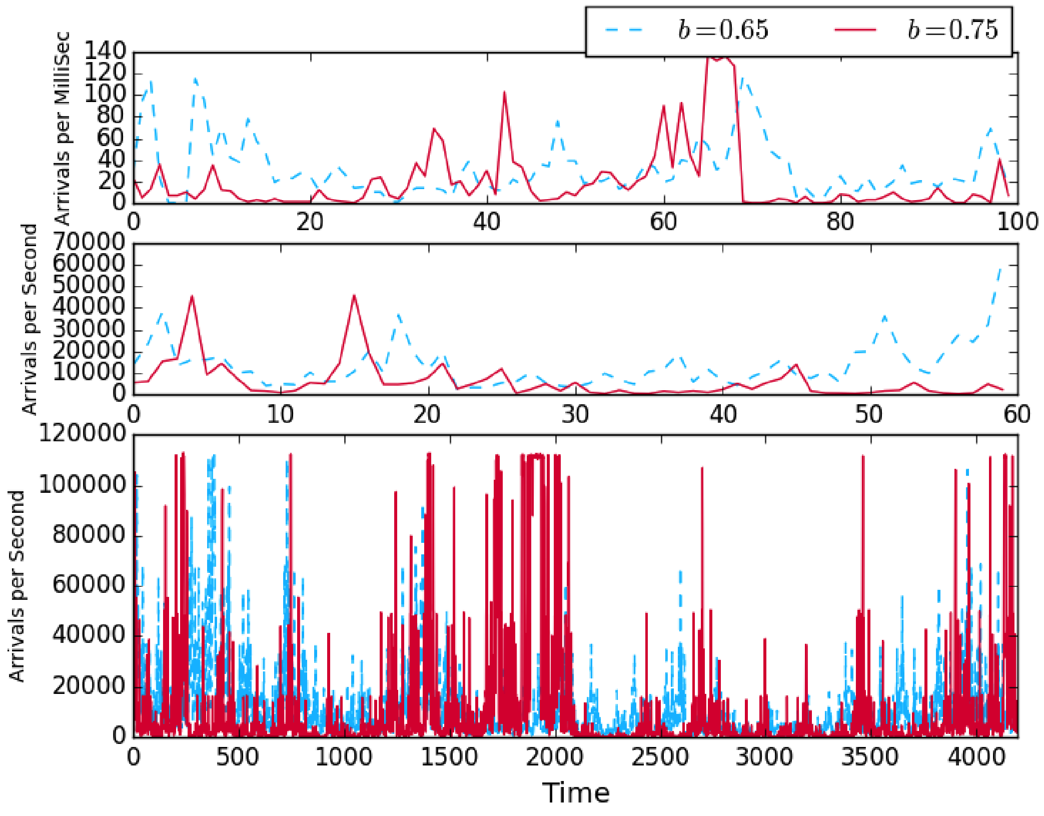


Figure 4.7: Self similar traffic generated by our modified workload executor, for a target rate of 10,000 operations per second, using the *b*-model configured with values 0.65 and 0.75.

4.3.3 Empirical Evaluation of Arrivals

We empirically evaluate the generated request arrival in this subsection to show that their statistic characteristics hold. We use both the Auto Correlation Function plot and Hurst parameter estimation for the evaluation. For better visualization, we present the ACF plot of the inter-arrivals generated for a short duration run, in Figure 4.9. It can be seen from the figure that, as described earlier, ACF of the inter-arrivals of the Poisson traffic quickly reaches near zero and remains close to zero throughout. But the ACF of the inter-arrivals of the self similar traffic with a bias 0.65 decays slowly to zero. The periodicity present in the envelope-guided diurnal traffic is also clearly visible, even with the presence of stochastic variations. The ACF plots of the arrivals generated per

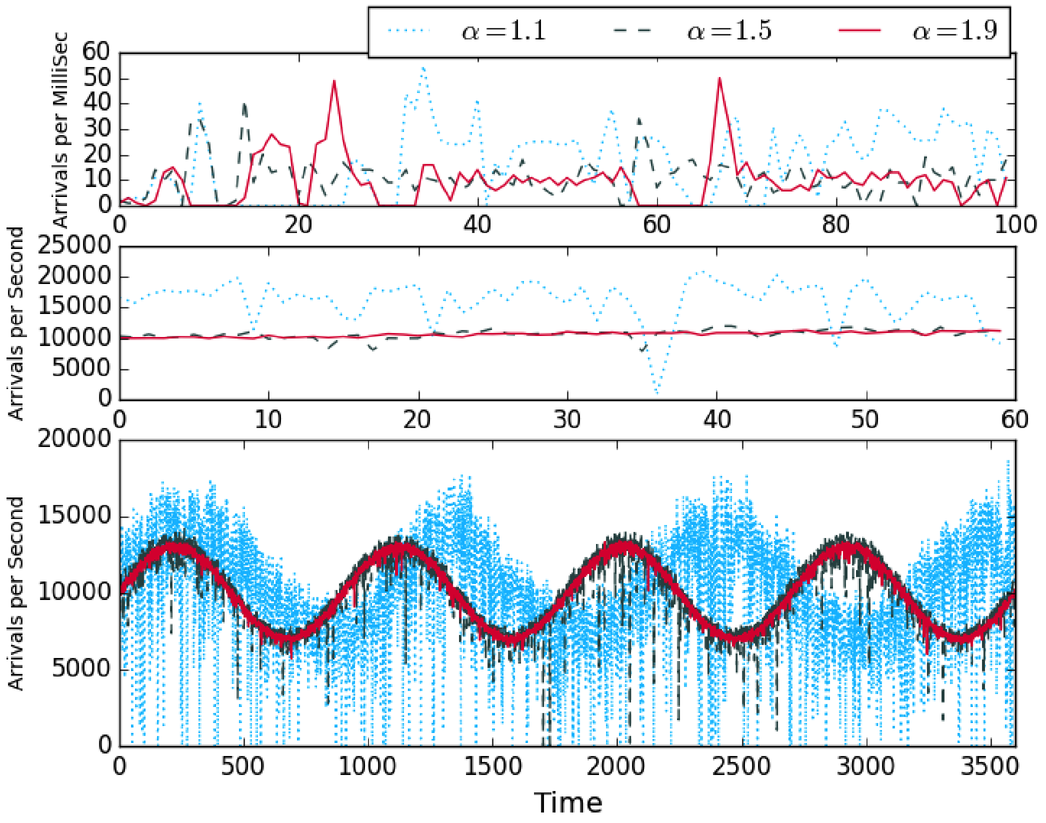


Figure 4.8: Envelope-guided arrivals, configured with a diurnal envelope combined with a Pareto stochastic variations, for a target rate of 10,000 operations per second.

second for the runs shown in the previous subsection was also similar.

Table 4.2 shows the results of the Hurst parameter estimation for the arrivals visualized in the previous subsection. We present the estimations for both a short duration inter-arrival time series picked from the beginning of the entire run, and the entire run's arrivals per second time series. As mentioned earlier, we use the tool Selfis [63] to estimate H using five different methods, namely the Aggregate Variance method, R/S plot, Periodogram, the Abry-Veitch Estimator, and the Whittle Estimator, and detailed descriptions of the various methods can be found in Feitelson's book on workload modeling [36]. As seen in the table, there are variations among the values estimated by the different methods, hence the approach of using different methods for the estimation is adopted.

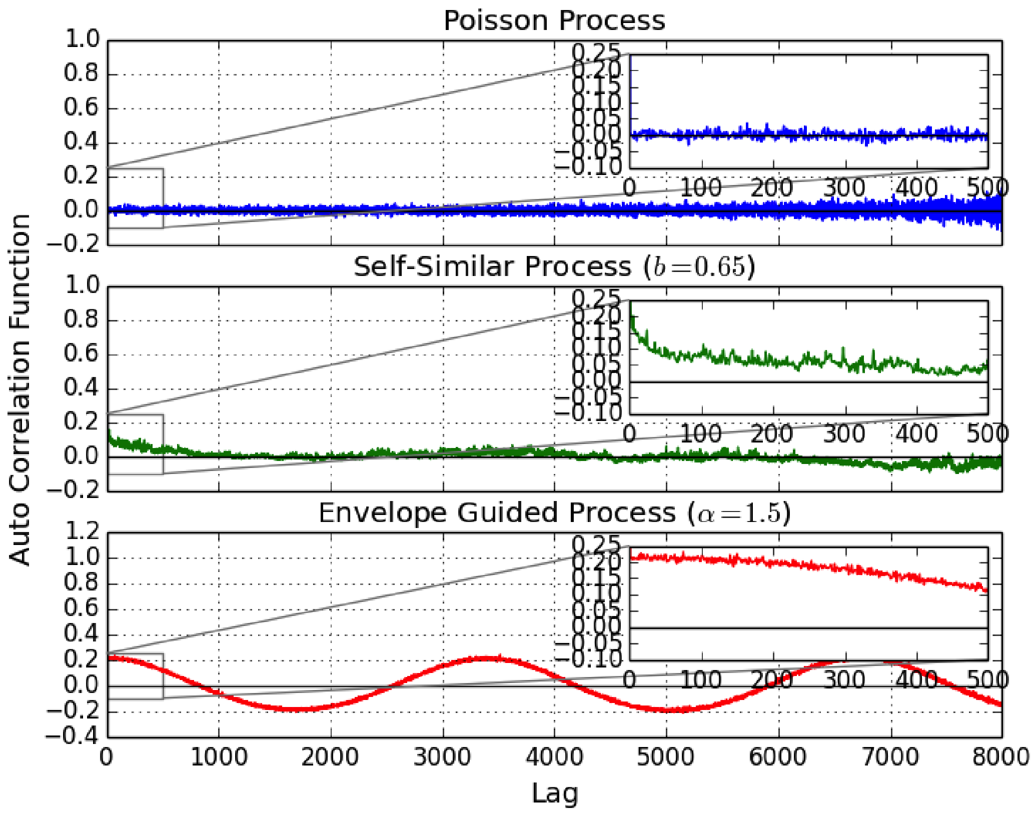


Figure 4.9: Auto Correlation Function plot for a sample short workload.

The general practice is to declare a process as self similar if most methods result in a Hurst estimation of above 0.5, and if it is close to 0.5, a Poisson process. The results show that the arrivals generated for both the Poisson and self similar processes are indeed Poisson and self similar, when seen at both scales. When seen as a whole, the diurnal process also results in higher Hurst estimates owing to high variability in the process.

4.3.4 Demonstration

In this subsection, we demonstrate the need for realistic arrivals by evaluating a state-of-the-art key-value database library using requests generated by all three arrival models we implemented in YCSB.

Table 4.2: Hurst Parameter Estimation

	Aggregate Variance	**R/S**	**Periodo-gram**	**Abry-Veitch Estimator**	**Whittle Estimator**
Small Part Of The Entire Inter-Arrival Time Series					
	Poisson				
	0.455	0.505	0.481	0.453	0.5
Bias	Self Similar				
0.728	0.698	0.626	0.695	0.509	0.5
0.75	0.761	0.636	0.757	0.656	0.559
alpha	Envelope-Guided				
1.1	0.465	0.501	0.492	0.559	0.5
1.5	0.447	0.491	0.49	0.547	0.5
1.9	0.499	0.482	0.504	0.526	0.5
Entire Arrivals Per Second Time Series					
	Poisson				
	0.447	0.002	0.469	0.529	0.5
Bias	Self Similar				
0.65	0.859	0.777	1.003	0.986	0.954
0.75	0.862	0.663	1.041	1.122	0.996
alpha	Envelope-Guided				
1.1	0.835	0.666	0.72	0.683	0.766
1.5	0.817	0.631	0.673	0.747	0.775
1.9	0.784	0.306	1.494	0.585	0.999

LevelDB

LevelDB is an open-source embeddable database library, that was written at Google and follows the same design as BigTable's [22] tablet. Like many modern key-value databases that strive to offer both good random insert and good sequential read performance, it follows a Log-Structured Merge (LSM) [85] tree based data management. An LSM-tree contains multiple ordered log-structured indexes, one in the memory and the others on disk, and when any index exceeds a per-determined size threshold, parts of it are compacted and merged with the index in the next level.

The compaction process, that both cleans and reorganizes data, is typically implemented as a background process that co-exists with foreground requests. Compaction is either done periodically, as in LevelDB or during administrator specified time window, as in HBase [48]. In essence, many systems similar to LevelDB exists, with background tasks that compete with foreground requests and affects performance. Some other systems such as Wang *et al.*'s work [125], extends LevelDB by implementing optimized scheduling and dispatching polices for concurrent I/O requests, to exploit the parallelism in open-channel SSDs. Proper evaluation and comparison of such systems is possible only with realistic arrivals that generates both periods of activity and downtime realistically.

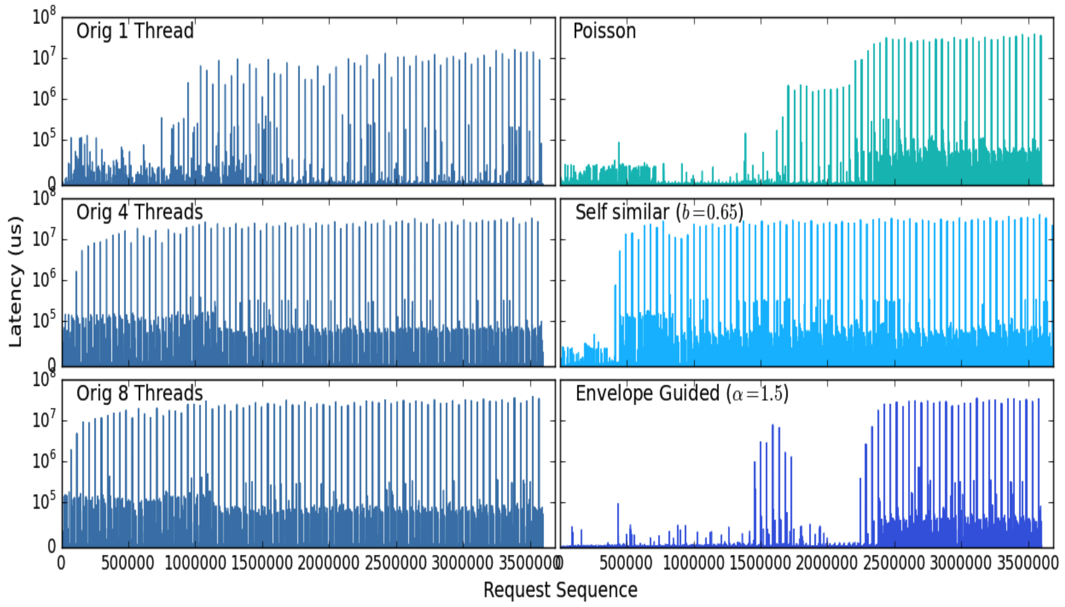
Hence, we have chosen LevelDB as the sample database to demonstrate the usefulness of our work. LevelDB is an user-level library that stores its data as files on the filesystem. Periodically, some of the files are to be compacted, and this happens along with serving incoming requests. As mentioned earlier, we run LevelDB on a dedicated 160GB hard disk drive, and storage management on the drive is done by an ext4 filesystem. LevelDB is an embeddable database and does not have a server communication component. Hence, as recommended by YCSB, we use MapKeeper server configured to use LevelDB as its datastore.

All experiments in this subsection are 100% random inserts with 16 byte keys and 1 KB values. We ran original YCSB against LevelDB without specifying any arrival rate throttling using a single thread, 4 threads and 8 threads. The overall throughput achieved by the continuous bombardment of requests were 1712.35, 1736.6 and 1733.75 operations per second respectively. We then ran the same workload using a Poisson arrival process, a self similar process with bias 0.65 and a envelope-guided diurnal process with $\alpha = 1.5$ Pareto inter-arrivals, with 1700 average target arrivals per second. The observed overall throughput of the Poisson, self similar and envelope-guided processes are 1663.32, 1690.94, 1655.59 operations per second. For very similar throughputs (in the range 1655–1690), we observed the per-request latencies varied tremendously.

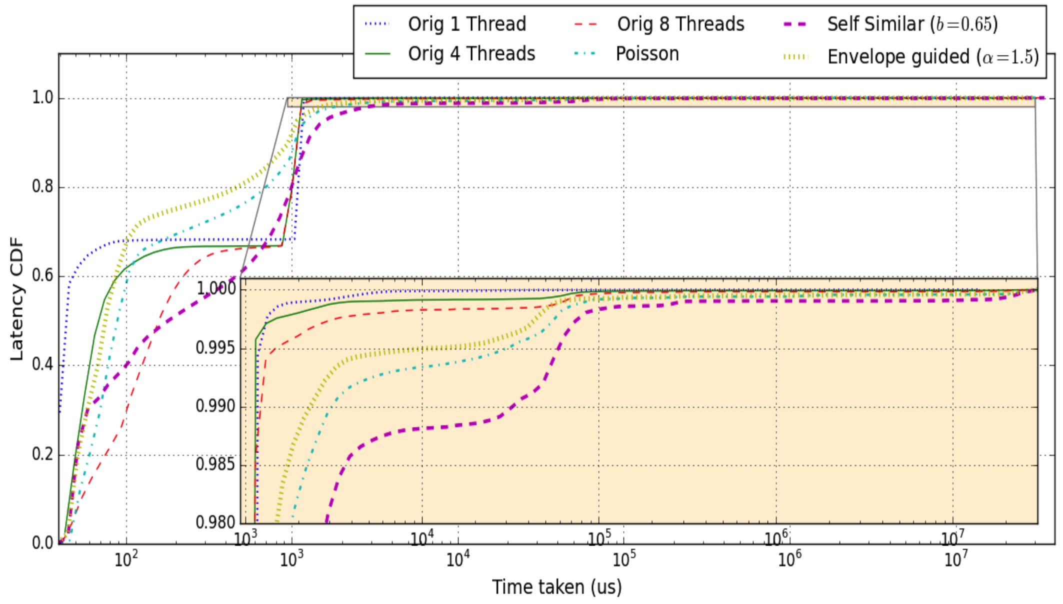
The results of the above experiment are shown in detail in Figure 4.10. Figure 4.10a is a semi-log plot of the observed per-request latencies over time, and Figure 4.10b is the normalized cumulative histogram on the latencies using logarithmic bins. Figure 4.10a is presented as a semi-log plot to highlight the requests that took really long time to complete. Without varying anything other than the arrival process, we could observe a high degree of variance in the observed latencies.

We can see that the peak delays appear consistently for three of the six runs, while they are absent for most of the time in both the Poisson traffic and the envelope-guided traffic. Determining the cause of these peak delays require an in-depth study of how LevelDB works. Suspecting background compactions, we measured the time spent in compaction during all these runs and found that the self similar traffic spent the least amount of time in compactions, and the single threaded original arrivals spent 141 seconds more than the self similar traffic on compaction. Even though self similar traffic produces the most bursty traffic, it also provides more downtime for the compaction to proceed uninterrupted, thus taking lesser time. This observation leads us to believe background compaction scheduling could bring the delays down, and our work could not only help identify such cases, but also aid in realistic evaluation of intelligent schemes such as compaction scheduling.

The latencies at the bottom in same graph appear much denser in cases where more requests bombard the database at any given time, that is in case of the multi-threaded original traffic and the self similar traffic. When the traffic is more smoother, more requests are completed sooner, as is evident from the results. This also indicates lack of better multi-request handling, while further investigation is required to confirm the same. The normalized cumulative histogram in Figure 4.10b also depicts how a workload's temporal characteristics affect the system behavior. The heavy tail highlighted in the figure shows the difference between the original back-to-back traffic and



(a) Semilog plot of observed latencies over the duration of the experiment.



(b) Normalized cumulative histogram of the observed latencies under various arrival patterns.

Figure 4.10: Latencies measured while running a 1 KB random insert workload against LevelDB, tested under various arrival models, demonstrates the effects of realistic request arrivals.

the realistic traffic we generate, and that systems which aim to bring down the heavy tail must most definitely be evaluated with realistic traffic.

4.4 Summary

Request arrival pattern can make a significant difference in the results of storage systems being evaluated, because performance observed under high yet steady client demand may actually be very different from that observed under bursty conditions. In this chapter, we categorized realistic request arrivals into three kinds, and implemented all three of them in the popular key-value storage benchmark, YCSB. We evaluated the arrivals we generated by showing that their statistical properties are both realistic and in line with commonly observed traffic patterns. We also demonstrated the effects of realistic arrivals on system behavior by evaluating a state-of-the-art key-value database, LevelDB, and conclude that to be representative of real world workloads, all storage benchmarks should provide the flexibility to generate realistic request arrivals.

Chapter 5

Compaction Overhead and Mitigation

In this chapter, we examine the compaction overhead in SMRDB in greater depth and suggest write-offloading to a NVRAM device when compaction is in progress to mitigate the impact compaction has on performance, and do so when and only when compaction is in progress to reduce writes to NVRAM device, as devices such as NAND flash have a lifespan dependency on writes. The chapter also describes the newly written version of SMRDB, adapted for write-offloading, *albeit* with LevelDB-style compaction that incurs overhead similar to the popular static LBA mapping schemes. We also evaluate write-offloading using realistic requests generated by our modified YCSB framework described in the previous chapter.

5.1 Compaction Overhead in SMRDB

Compaction in SMRDB not only frees up dead space, but also aggressively rearranges and moves data around to result in good read performance. This rearrangement generates additional read and writes requests to be serviced by the disk, an added overhead interfering with the service of user-initiated read and write requests. Even if the user requests were given a higher priority and background requests were asked to be served after servicing the user requests, the interference and performance degradation would come in the form of disk head arm movement.

Compaction overhead will be present in all SMR data management schemes, but is elevated in schemes that does more than just clean dead data, such as in SMRDB where compaction also reorganizes data. This is not only true for our LSM-tree based dynamic mapping scheme, but also for static LBA mapping schemes such as the ones described by Cassuto *et al.* [21] and the first generation drive-managed SMR drives from Seagate [5]. The static LBA mapping schemes do more reads and writes than needed while performing a read-modify-write to update a modified LBA, but maintain the LBA ordering, which in turn could maintain the read performance an user would expect from a disk, if not for the compaction interference.

The following are the concurrent I/O request threads that interfere and affect one another's performance in SMRDB:

1. *Sorted object writes:* are user-initiated writes. The user objects are sorted in memory and dumped to the disk by sequential writes to disk. The order of the writes determines when and how frequently background compaction happens, but otherwise results in good performance due to sequential nature of the writes.

2. *Log writes:* are by themselves sequential writes to disks, but, since every log write is also accompanied by another write to a separate location on disk (the sorted object writes) in parallel, affects incoming write performance. As the user objects are held in memory and sorted, the log writes protect the user data and are essential. As discussed in Chapter 3, if the logs were stored in a separate NVRAM device, performance would double, but a smaller NVRAM device with lifespan dependency on writes will wear out soon.

3. *Object reads:* are user-initiated reads. Read performance has been an important design goal and the compactions strive to provide a sequential read throughput

that one would expect out of a disk. Random reads on a disk wouldn't feel the interference from the compaction as the reads are themselves random and suffer from disk arm movement. Though sequential reads would feel the interference, compactions couldn't be given a lower priority as they are required for future sequential read performance.

But if desired, an NVRAM object cache can be employed to serve read requests, thus mitigating the interference with other reads and writes. As caching has been well studied, and read caches are not in need of any SMR specific handling, we do not study the behavior of caches in this work. We suggest a key-value object cache rather than a raw block cache as compactions keep changing the object location.

4. *Compaction-induced writes:* are sorted band writes running in the background, and are also sequential writes to disk. Though sequential, when in progress, it interferes with incoming writes (both log and user object writes).

5. *Compaction-induced reads:* are sequential reads to bands being compacted. The number of bands being compacted at a given time determines the level of interference caused by the compaction-induced reads.

5.1.1 User Write Schemes

One key difference in write behavior between SMRDB and most other SMR management schemes is that SMRDB writes every object twice to make every band sorted, while most other systems would only log the writes. While the write behavior increases the write amplification, the sorting helps not just the sequential reads from the user, but also the compaction-induced reads. As illustrated by the merging of bands in Figure 5.1, compaction reads request data from a band in sorted order, and an already sorted band is read sequentially by the compaction process. Even when multiple bands are compacted, data can be read sequentially in batches from the bands being compacted, as shown in the figure, making compaction runs faster and more efficient.

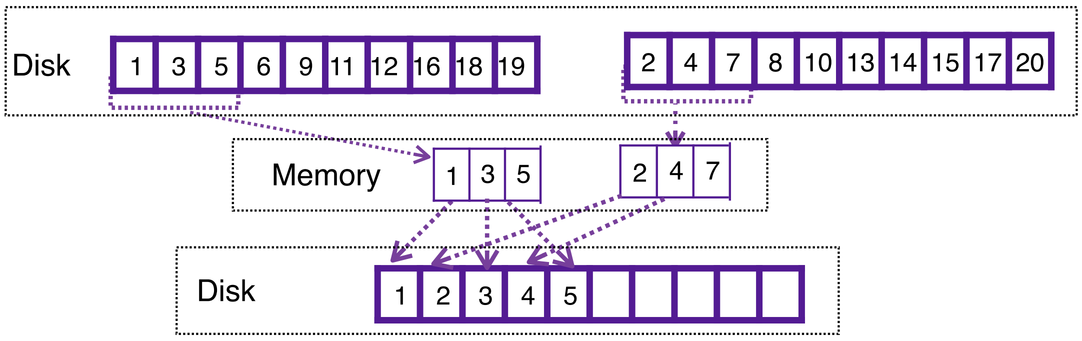


Figure 5.1: Merging pre-sorted bands reads data sequentially from disk, and can be improved further by batch reading parts of the bands being merged into memory.

One approach to improve the write performance, as discussed earlier, is to store the logs in a separate disk/NVRAM device/battery-backed RAM. However, the fact remains that every object is written twice and the user writes directed to disk would still face contention from background compaction.

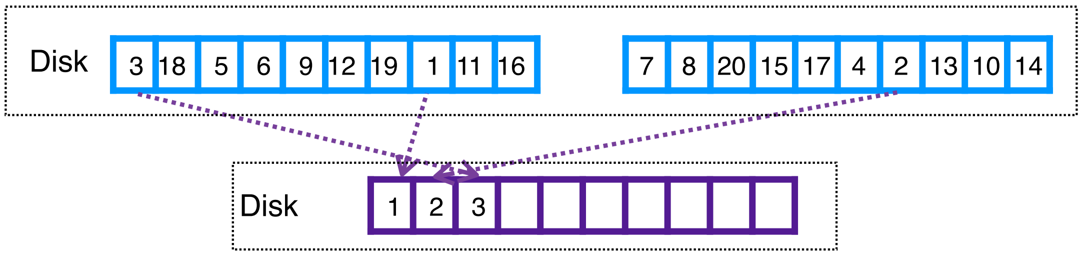


Figure 5.2: Merging a number of unsorted bands would result in random reads, lowering performance.

However, if the write order is not sequential and overlaps with data in other bands, the data in the band will be read and reordered again by the merger. Hence, the incoming object writes could just be logged and the sorting could be left to be done by the merger. The scheme would result in unsorted bands as those shown in Figure 5.2. The tradeoff in this alternate scheme is to give up sequential read performance on the bands that have not yet been compacted to gain some write performance. In this scheme, the compaction-induced reads would not only increase the interference on incoming write requests, but as the compaction-induce reads to unsorted bands are now random reads, compactions would be slower, resulting in fewer compaction runs than before. For the above reasons, SMRDB chose the first method of writing in Chapter 3.

5.2 Mitigation by Write-Offloading

Write-Offloading offloads the incoming user writes to another available resource. While caching can reduce the number of reads reaching the primary resource, write-offloading takes care of writes, and frees up the primary resource, and generates a pseudo downtime for the primary resource, which can be used for reasons such as power savings, or, as in our case, to perform background activities. In this section, we describe how and why write-offloading can be used to mitigate compaction overhead and increase performance.

Some systems prioritize writes and push off compaction to be done during periods of lesser usage [18]. Systems such a s HBase [48] split compaction into minor and major compactions and perform major compactions during user specified intervals, to be able to configure the runs to be performed during expected downtime such as nights. Others, such as LevelDB, prioritize scan performance at the cost of write performance and delay and even stop incoming writes, to make time for compaction to complete its job. In Chapter 3, we discussed smarter band organization and compaction process to achieve a balance between the two. We also showed how delaying the compaction process slightly and accumulating more incoming writes would benefit both read and write performance.

Here, we look at how write-offloading would help no matter what the compaction process does and how high the compaction overhead is without offloading. The basic idea is to offload the writes to a separate resource, when and only when compaction is in progress. Since writes happen sequentially to disk, write performance would not be a concern if not for the interference from compaction. When compaction is in progress, if the writes are offloaded, it would make more room for the compaction to run and achieve its intended goal without affecting the incoming writes, creating a win-win scenario.

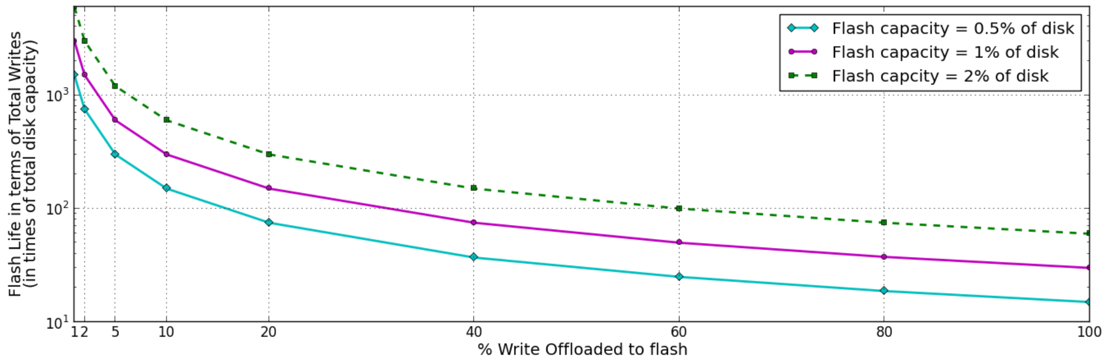


Figure 5.3: Expected flash lifespan, if flash wears out in 3,000 write cycles, for various degrees of write-offload.

If an NVRAM device with lifespan dependency on writes, such as NAND flash were to be used for offloading, logging everything on the device first will wear out the device sooner. For the very same reason, disk-based write caches have previously been employed to extend SSD lifetimes [112]. Whereas, offloading only when compaction is in progress, would reduce the number of writes to the NVRAM device and the device would live longer. Figure 5.3 presents the lifetime expectancy of the attached flash device in terms of total writes, measured in times of total disk capacity, for varying degrees of write-offload, if flash is expected to wear out in 3,000 write cycles. As per the figure, if 40 GB of flash is added to a 8 TB disk, and writes are offloaded 100% of the time, the attached flash would wear out when the total incoming writes reach 120 TB (15 full disk overwrites). On the other hand, if the writes were offloaded only 40% of the time, the attached flash will last until 300 TB (37.5 full disk overwrites).

A small NVRAM device attached to the drive would suffice for the offloaded data, as data doesn't remain in the offloaded device long and is moved to the banded disk by compaction. No matter how long or slow the compaction run is, it would not affect the incoming write performance and would result in predictable performance. The technique also paves way for elimination of the double writes in SMRDB, by making unsorted first level writes more attractive.

As discussed in the previous section, the problem with unsorted (or logged) first level writes instead of log+sorted writes is decreased read performance, both for incoming sequential reads from the user and compaction-induced reads. But with offloading, a) longer compaction runs wouldn't affect the incoming write performance, eliminating the concern over slow compaction-induced reads on unsorted disk bands, b) compaction-induced random reads on data offloaded to the NVRAM device would also not affect incoming write performance, as random reads on the NVRAM device are much faster than those on disk and can happen in parallel to the writes without much effect, and c) with more room for compaction runs, compaction would not leave many bands on the disk unsorted for long, so incoming sequential scans will also not be affected to a large extent.

5.3 Evaluation System

In this section, we describe the data management system used to evaluate write-offloading. Our goal in this work is not just to improve SMRDB, but to suggest write-offloading as a technique to mitigate compaction overhead for all SMR data management systems. We will not be recreating all popular data layouts proposed for SMR disks, but the compaction overhead in our system has to be representative of that expected in most systems.

Most SMR data management systems include buffer bands to log incoming writes, and move data from the buffer bands to their intended location in the background. SMRDB not only logged the writes in log bands, but also sorted them in memory and stored them in another band as and when data is written. But, as discussed in the previous section, with write-offloading to a NVRAM device, SMRDB would also benefit from only logging the writes instead of logging and sorting the incoming writes. The performance of a pure LFS layout, where the only goal of compaction is to clean dead data, has not yet been evaluated for SMR disks, and we believe poor read performance is the reason.

Most systems so far seem to prefer a static LBA mapping to the bands and are prepared to incur high read and write amplification for the sake of sequential read performance. SMRDB maps the keyspace dynamically to bands and reorganizes data only if the benefit of doing so is high. The improved compaction scheme in Chapter 3 thus decreases the read and write amplification expected in a static mapping scheme, or a strict LevelDB-style compaction with a strict no-overlap in levels policy.

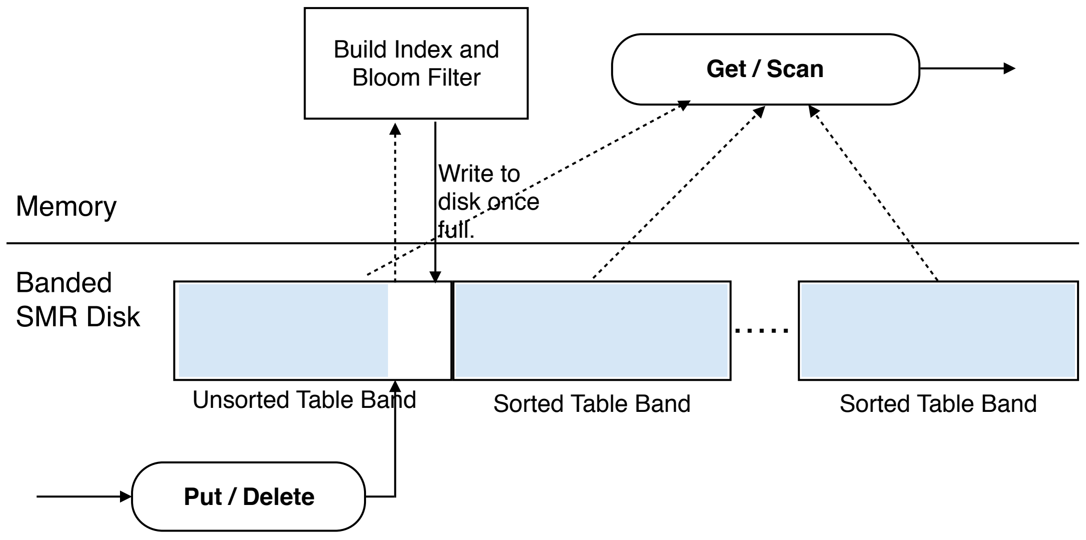


Figure 5.4: System design eliminating double writes.

The object management system in this work is a recreation of SMRDB with changes to accommodate first level logged/unsorted band writes, and without the improved compaction scheme to provide a more representative compaction overhead to evaluate write-offloading with. Figure 5.4 illustrates the read and write behavior in the implemented system when writes are not offloaded. Similar to SMRDB, the code is based on LevelDB and bands an underlying disk into equal sized bands. Skylight [5] reverse engineered drive-managed SMR drives and noted the band sizes to be 15–40 MB and SMRDB in Chapter 3 was evaluated with 80 MB bands. We did not observe a performance difference between 32 MB bands and 64 MB, and hence fixed our band sizes to be 64 MB. Our system has 3 levels of buffer bands before reaching the final level where all data are stored. The first two levels are made up of 10 bands each and the third level has 100 bands of data.

The compaction is aggressive. When data is actively inserted, compaction is triggered when the number of buffer of bands reach the above threshold, but incoming writes are not stopped if the threshold is met and new bands are allocated to serve as buffer bands. But when a periodic checker finds that compaction hasn't happened for a while, indicating fewer writes, buffer bands are merged more aggressively. Because, if the buffer bands are near full and objects in the buffer bands are not compacted and sent down, future write bursts could trigger compactions early. The offloaded writes are stored in files in the attached NVRAM device, using the default file store implementation through which LevelDB stores all its data.

5.4 Evaluation

In this section, we evaluate write-offloading using both a micro-benchmark, db_bench and a macro-benchmark, YCSB. Our experiments evaluate the system under various realistic arrivals using the framework described in the previous chapter. DB_bench and stock YCSB have been used in the evaluation of SMRDB and LevelDB in Chapter 3 and the modified YCSB has been used to evaluate LevelDB in Chapter 4. Although the experiments in all three chapters were performed in different machines, the use of the same benchmarks provides a rough comparison performance across the chapters.

5.4.1 Experimental Setup

The evaluation was done on a Ubuntu 14.04 LTS Linux machine that has a 8-core 3.50 GHz Intel® Xeon® E5-1620 v3 processor with a 256 KB L1 cache, 1 MB L2 cache, and a 10 MB L3 cache, and 16 GB of RAM. As our system is an embeddable database, MapKeeper [3] server was used to connect and communicate with YCSB.

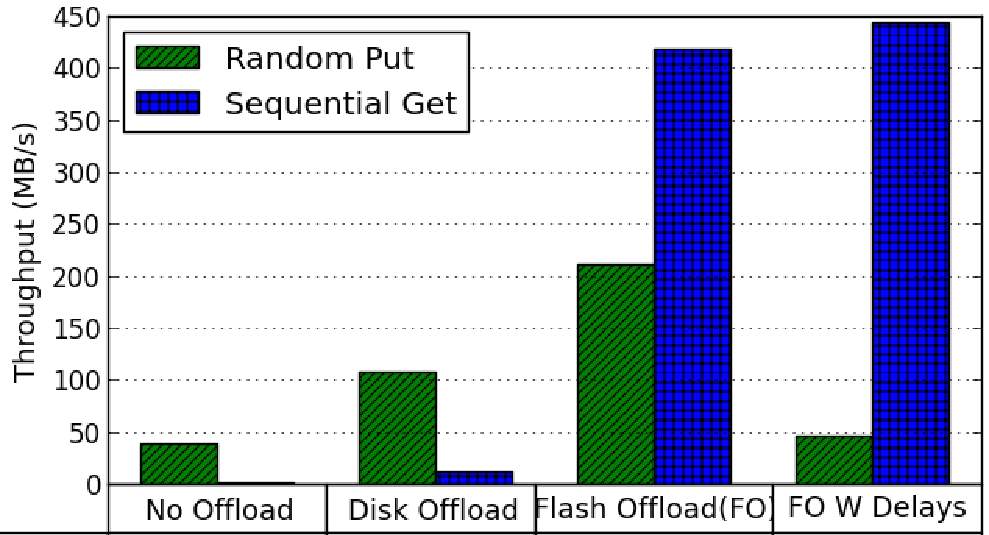
Both the YCSB client and Mapkeeper server were run on the same machine. A separate dedicated 160 GB single platter Seagate SATA disk drive was banded and written to using raw sector-level primitives, similar to SMRDB. The device used to offload writes is a 16 GB partition in a 120 GB SanDisk SSD formatted with ext4 filesystem, and the rest of the device remains unused.

5.4.2 Micro-Benchmark

We use the LevelDB's *db_bench* benchmark program to evaluate write-offloading under a random insert workload, which is immediately followed by a sequential scan load. The micro-benchmark run is designed to evaluate the technique's effect on performance during high compaction overhead, and hence, only tests random inserts which is bound to result in a high compaction overhead. The keys were 16 bytes long and the test inserts 2.5 million 4 KB values, and the subsequent test reads 10,000 sequential objects.

We present a performance comparison between the system without write-offloading, with write-offloading to a separate disk, with write-offloading to NAND flash, and finally, with write-offloading to flash with 10 microseconds induced sleep delays when too many files get accumulated in the flash, and the results are shown in Figure 5.5. The table in the figure presents how much data were read and rewritten by compaction, remains unsorted in the first level, and were offloaded to another device at the end the insert phase and right before the sequential read phase, presented as the percentage of data originally written by the benchmark.

The no offload results show that while eliminating double writes and writing the first level bands unsorted results in a decent random insert performance (compared to the results seen in Chapter 3), the sequential read performance takes a huge hit. That is because the compaction runs also take longer to merge unsorted bands, and 78.36% of the data written remain in the unsorted first level buffer. Offloading the writes to another separate disk more than doubles the performance while making room for more compaction runs, resulting in fewer data in the first level unsorted bands. But



	No Offload	Disk Offload	Flash Offload(FO)	FO W Delays
% Read	39.55	53.52	29.98	172.25
% Rewritten	37.44	49.14	27.47	159.1
% Unsorted	78.36	68.38	76.28	30.13
% Offloaded	-	96.13	92.26	89.1

Figure 5.5: Measured random insert and subsequent sequential read throughput, using LevelDB's benchmark program, db_bench. The table presents the system state after the inserts and before the sequential reads.

even then, 68.38% of the data remains unsorted. Though offloading to disk reduces the amount of unsorted data and results in better read performance than before, the performance is still low because of the huge percent of unsorted data on disks.

Offloading to flash, on the other hand, is way faster and the inserts are completed in a few compaction runs. And unsorted-ness is not a concern on flash as on disks due to faster random reads and the subsequent sequential read performance is also high, as most data ($\approx 76\%$) remain on flash. Compactions run more often when a per request 10 microsecond delay is introduced while offloading to flash, and only $\approx 30\%$ of the data remain unsorted. As the data on disk are mostly fully sorted, the sequential read performance is higher than the previous case even though less data remain on flash.

5.4.3 Macro-Benchmark

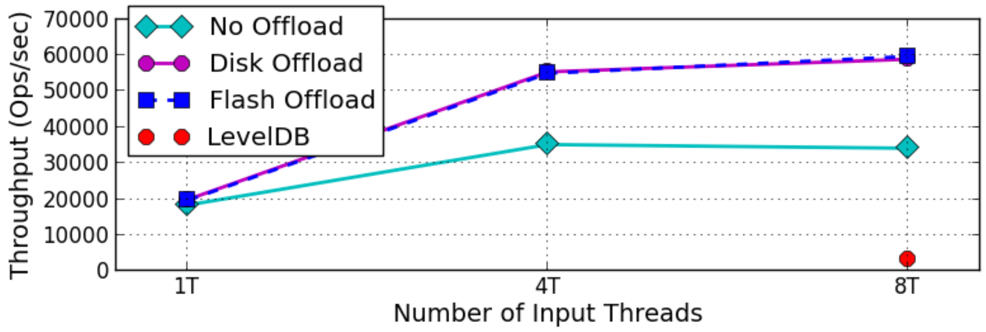
In this section, we use the Yahoo Cloud Serving Benchmark, with and without the realistic request arrival generation described in Chapter 4, to evaluate write-offloading. There are no per request delays as the one shown in the previous micro-benchmark in these experiments. The value size of the objects are 1 KB in all the experiments and the inserts are random, unless otherwise mentioned.

Back-to-Back Multi-threaded Arrivals

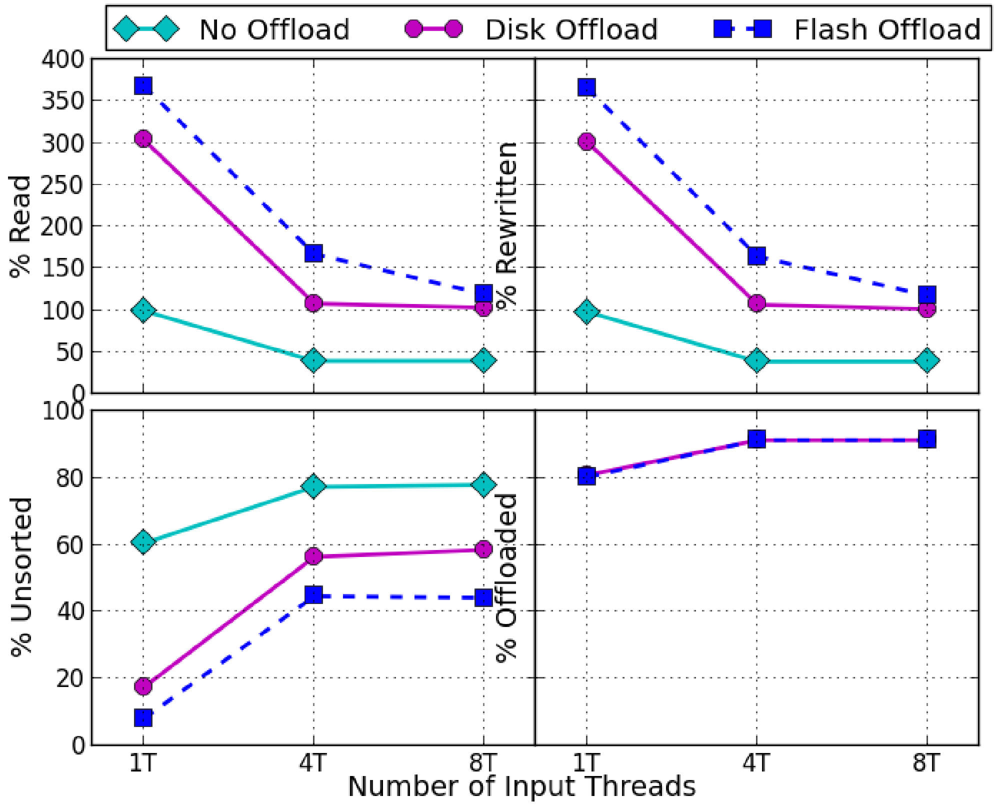
This experiment sends 10 million multi-threaded back-to-back requests to measure the maximum throughput in the YCSB mapkeeper framework, with and without write-offloading, and to evaluate the effectiveness of write-offloading in such a scenario, and is presented in Figure 5.6. We evaluated the system without offloads, with offloads to disk and offloads to flash with requests sent *via* a single thread (1T), 4 threads (4T) and 8 threads (8T). Figure 5.6a presents the write throughput and the Figure 5.6b presents the percentage of reads and writes by compaction, the percentage of data in unsorted bands and the percentage of data writes that were offloaded to flash.

For reference, we ran the same workload using 8 threads on LevelDB configured to store its files on an ext4 filesystem on the same test disk, and to use a 64 MB write buffer (same as in our system), and show the throughput in Figure 5.6a. As shown in the figure, our system far exceeds LevelDB's performance, but due to the added framework induced overhead all three systems achieve the same throughput in the single threaded case. Even though the throughput is the same, the write-offloaded cases do more compaction than the one without any offloads and result in fewer data in unsorted bands. And, offloading to flash paves way for more compaction runs, and results in fewer data in unsorted bands than when offloading to disk.

With multiple threads, in the no offload case, the throughput increases at the expense of compaction runs (fewer compaction induced reads and writes), leaving more data in unsorted bands than with a single thread. The multithreaded tests on offloaded



(a) Insert throughput of the system without offload, with offloads to disk and with offloads to flash, and that of stock LevelDB with same write buffer as our system.



(b) System state during and after the insert phase.

Figure 5.6: System performance measured with and without write-offloads with back-to-back multi-threaded random inserts using YCSB.

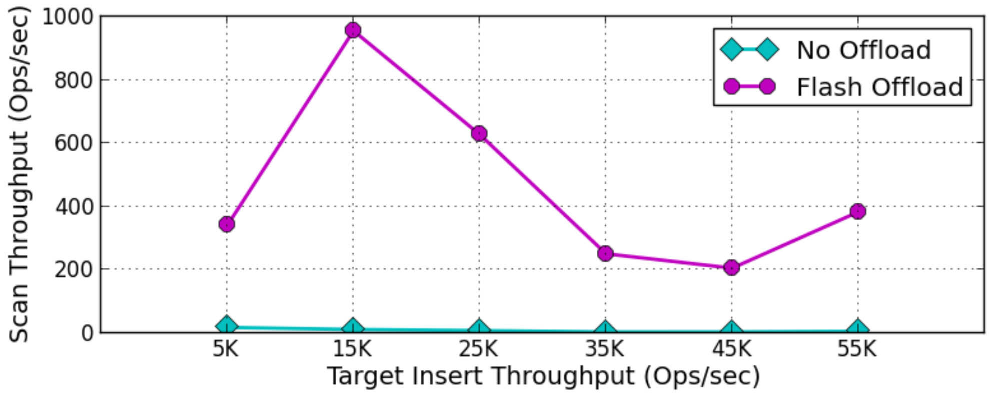
cases have higher throughput than the one without offloads, while also compacting more than the no offload case. While both flash and disk offloads have the same performance, flash offloads compact more than disk offloads do. The throughput does not scale when the threads are increased from 4 to 8, and could be due to either framework induced overhead or internal multithreaded engineering issues that we inherited from LevelDB, on which our code is based on.

Although we know and demonstrated the read related issues while offloading to disk in the micro-benchmark, we included the disk offloads in the current experiment to show that on a low insert rate as is common in most real world disk workloads, a disk offload might be sufficient to achieve result without the added expense of a NVRAM device. While not a suitable device to make a hybrid SMR drive, write-offloading to disk is a technique worth exploration in a multi-SMR-drive environment that work together. Though an interesting direction, as it is not the primary topic of this work, we will consider disk offloads no further.

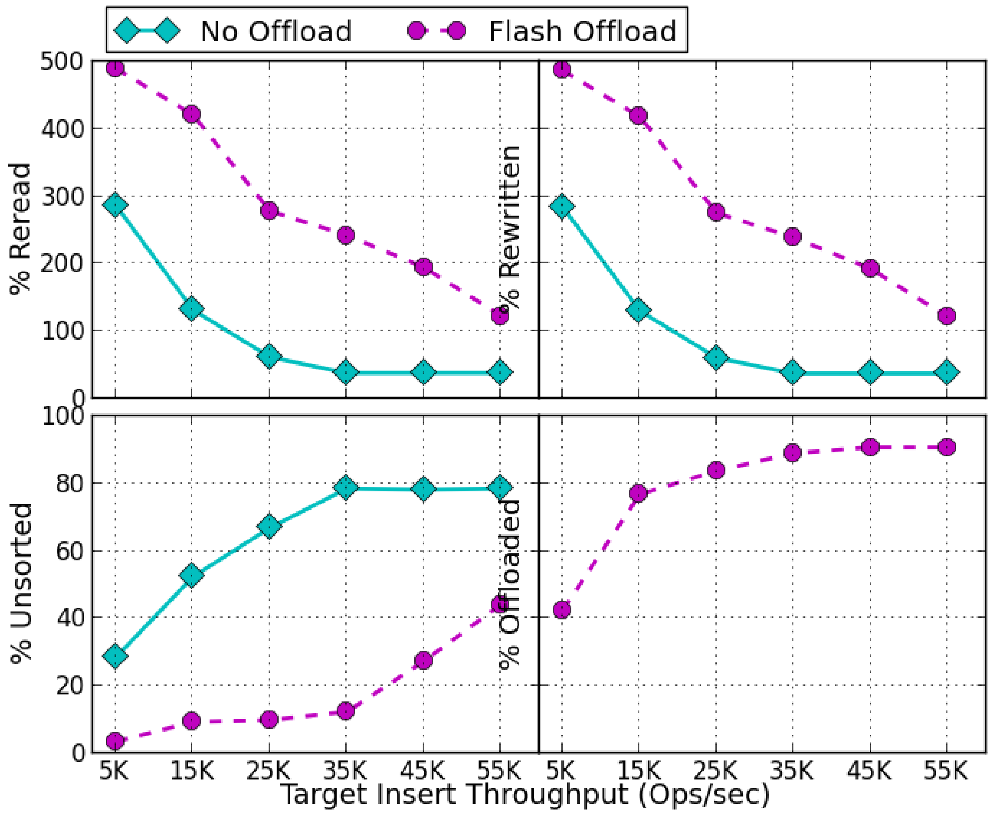
Poisson Arrivals

The following experiment loads the database with 10 million objects, inserted randomly using poisson arrivals with different target throughputs, and run a scan workload that randomly selects the start keys and scan lengths between 1 and 100 among the loaded keyspace. We varied the target throughput from 5000 operations per second to 55,000 operations per second in increments of 5000 operations per second. The compaction and offload statistics immediately after the load phase, and the resulting scan throughput are presented in Figure 5.7 for both with and without write-offloads to flash.

The compactions runs are happening in parallel to serving the scan requests in these tests. As seen in the Figure 5.7a, the scans perform poorly when writes are not offloaded. As seen in Figure 5.7b, when the target throughput is high, fewer compaction runs are completed, resulting in most data in unsorted bands. When the target is



(a) Read throughput of the system immediately after the insert phase is complete.



(b) Compaction and offload statistic comparison and the initial insert phase.

Figure 5.7: Performance comparison of the system with and without write-offloads to flash under a workload with Poisson arrivals, at various target rates.

low, more compactions happen and there are fewer unsorted bands. Even then, the scan performance is poor because compactions are not fully done and are happening in parallel to the scans, interfering with and lowering the scan performance.

If writes are offloaded to flash when compaction is in progress, compaction has moved more data around through more runs and decrease the percent of unsorted data by the end of the load phase. Lower the target throughput, higher the compaction runs and lesser are the number of unsorted bands. With lower target throughput, the data are also offloaded less often to flash, meaning that the compaction runs run quicker and more often than when writes are not offloaded. In other words, the system could serve inserts at a higher throughput than what LevelDB does, and rearranges most of the data in the right order, and requires only less than half the writes to be offloaded to flash.

The scan throughput of the offloaded case at 5K target throughput in Figure 5.7a is what is achieved from a mostly compacted disk, since the Figure 5.7b shows that very small percent of data remains in flash and compaction data movement is the largest in the set. When target throughput was 15K, compaction data movement is closer to that of the previous case, making the disk data largely sequential. Also, the percent of unsorted data is larger than the previous case and the high offloaded percent indicates that the unsorted might mostly be on flash. The combination of these two results in high scan performance. As we move from 15K to 45K, we note that compaction has not been run as much as before, causing compaction runs in parallel to the scan operations, decreasing the throughput. Even then, throughput is not as low as without offload, since the unsorted data resides mostly on flash. The 55K case performs better than the 45K case even as everything else follows the same pattern and is because a very large portion of data resides on flash.

Figure 5.8 compares the actual observed latencies of the system with and without offloads side-by-side, and Figure 5.9 presents histograms of the same. As seen in Figure 5.8, there are many inserts with high latencies when writes are not offloaded

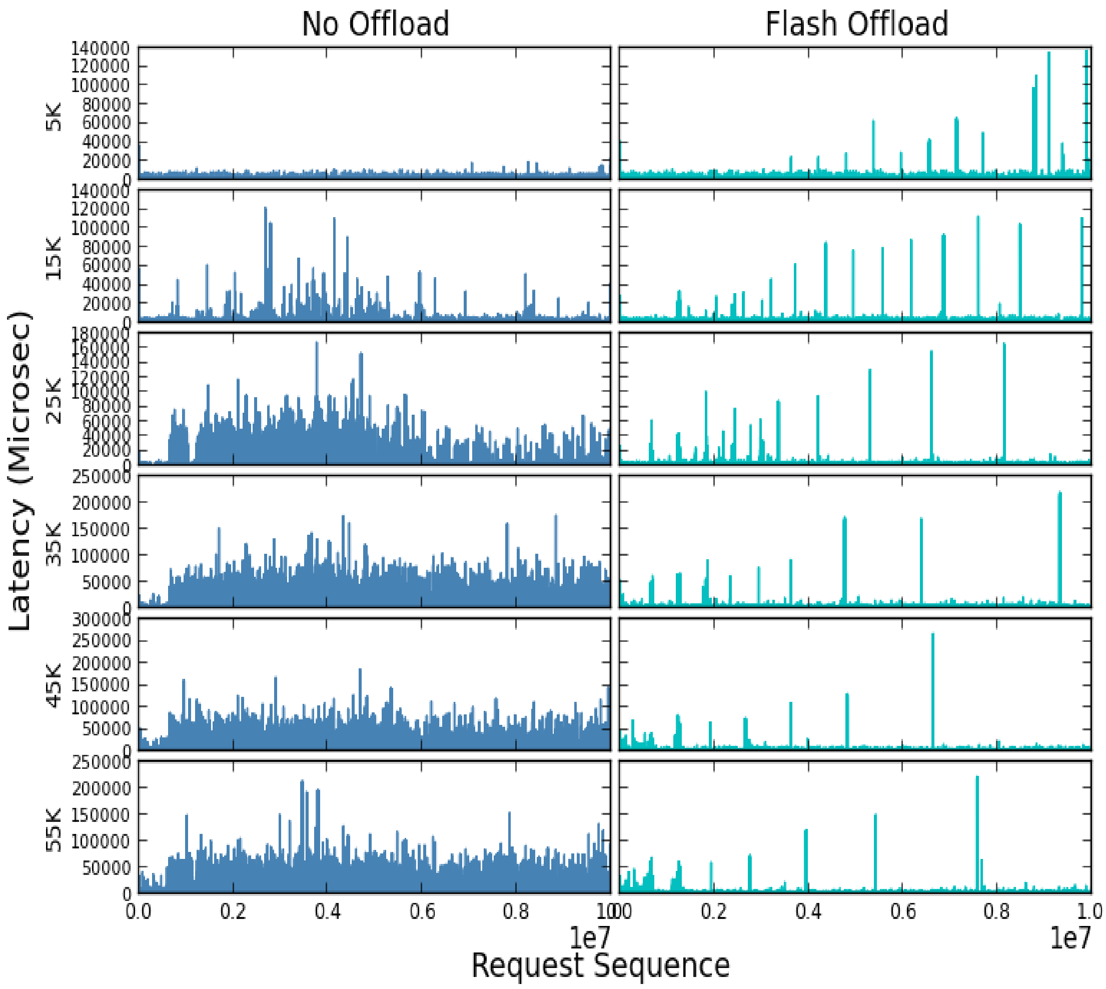
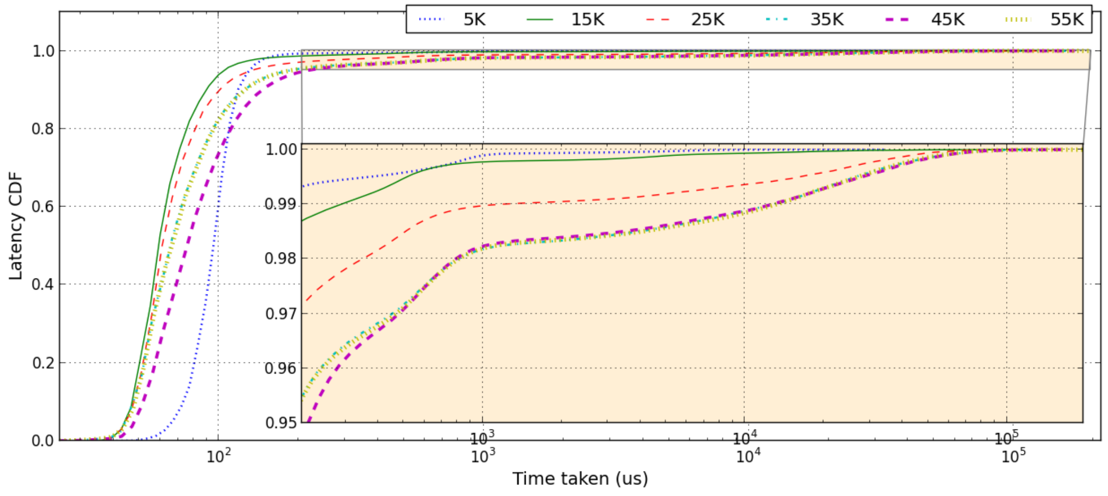
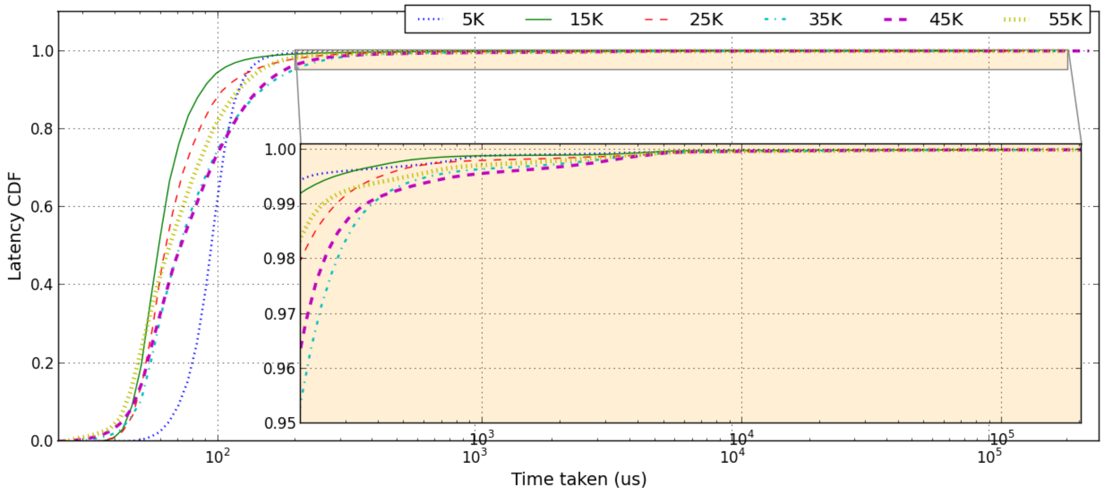


Figure 5.8: A comparison of observed latencies measured under Poisson arrivals.

than when they are. On further examination, the abnormally high latencies when writes are offloaded happen when the current memtable is full and the previous memtable is still being synced to the file stored on flash (there are only two outstanding memtables at any given time). There are no other evident cause for the delays introduced by the file system sync operations, and could be avoided if the flash was directly managed by the system similar to the SMR drive without depending on a file system. We consider such a direct management future work. As the data are stored on the write buffers with or without offloading, most requests have similar latencies as seen by the cumulative histogram in Figure 5.9. The high latencies are highlighted in both Figure 5.9a and



(a) If writes are not offloaded, higher target rates result in many inserts with high latencies



(b) When offloaded to flash, there are fewer high latency inserts

Figure 5.9: Normalized cumulative histogram of the observed latencies when the objects are inserted using YCSB Poisson arrivals, at various target rates.

Figure 5.9b and illustrate the difference between the two cases.

Self-Similar and Diurnal Arrivals

It has been noted that disks are idle greater than 90% of the time and that most of the time reads dominate writes. SMR disks are also more likely to be used in the capacity tier than the performance tier. And while it is important to be able to support high write rates, most of the time the write rate is going to be lower. In this final experiment, we evaluate our system with self-similar and diurnal arrivals at average target rates of 5000 operations per second or less.

Table 5.1 presents the compaction and offload statistics of running 8 workloads one after another in the order specified in the table on top of the same disk. The first two workloads are pure inserts and the rest of them are mixes of inserts, updates and scan operations. The inserts and updates together form 30% of the operations performed in the rest of the workloads. The key takeaways from this experiment are given below:

1. As bursty workloads write most of their data in bursts and trigger compactions during the bursts, writes are offloaded most of the time. Even at a rate of 100 operations per second, writes are offloaded 50-70% of the time.

2. Aggressive cleaning combined with a requirement to strictly enforce no-overlap of key range across bands results in a high read and write amplification. The amplifications are the worst when the rate of write is very low. As seen in the table, workload 6 writes in a low but relatively regular rate than self-similar workload, and aggressive cleaning moves data to the lowest level during periods of low usage repeatedly throughout the length of the test run causing the high amplification. Higher rates accumulate more data at upper levels before moving them to the lowest level where all data live.

3. While write-offloading is very useful in supporting high request service rate, low rate workloads would benefit from smarter I/O rate and type detection schemes

Table 5.1: Compaction and offload statistics of more realistic disk workloads run in the specified order.

Workload Order	**Written (MB)**	**% Read**	**% Rewritten**	**% Offloaded**
Self-Similar: Bias = 0.65				
1	Avg = 5000 ops/sec			
	22180	1500.75	1497.55	94.73
2	Avg = 5000 ops/sec			
	5607	1597.4	1539.77	99.98
3	Avg = 100 ops/sec			
	123	34556.1	34493.5	50.41
4	Avg = 100 ops/sec			
	431	17426.45	17330.39	71.46
Diurnal: Cycle length = 15 mins, Distribution = Pareto, Pareto-shape = 1.5				
5	Avg = 100 ops/sec, Modulation = 90 ops/sec			
	124	50995	50849	50
6	Avg = 500 ops/sec, Modulation = 490 ops/sec			
	123	20125.25	20026.02	40
7	Avg = 1000 ops/sec, Modulation = 950 ops/sec			
	1109	2925.79	2915.87	94.41
8	Avg = 5000 ops/sec Modulation = 4900 ops/sec			
	2896	2266.82	2132.35	97.86

to determine when to run compaction. If combined with a better compaction process such as our sequentiality metric-based compaction process, described and shown to reduce the read and write amplification in Chapter 3, the amplification could be reduced even further and the available resources could be used better. Further, detection does not have to be accurate, as write-offloading can take over and support unexpected changes in the workload.

5.5 Summary

In this chapter, we have proposed write-offloading to a NVRAM device when and only compaction is in progress, to mitigate the compaction induced interference on incoming object writes. We have implemented a version of SMRDB with logged first level band writes and LevelDB-style high overhead compaction to evaluate the effectiveness of write-offloading. Our detailed evaluation using a variety of request arrival process models shows that compaction overhead and compaction induced amplification can be very high if the compaction is aggressive, and that write-offloading can be used to hide the compaction overhead and achieve high throughput.

Chapter 6

Conclusion

Shingled Magnetic Recording demands writes to be largely sequential on disks, and new SMR-aware data management solutions. In Chapter 3, we proposed an SMR-aware Log-Structured Merge tree based key-value data management for SMR disks, and showed that despite being restricted to sequential writes, it outperforms LevelDB, a state-of-the-art LSM-based filesystem-dependent key-value store, in most cases. To measure compaction overhead and to evaluate mitigation techniques under realistic loads, we incorporated realistic request arrival generation in YCSB in Chapter 4. Finally, in Chapter 5, we proposed an improved data layout for a hybrid SMR drive that includes write-offloading to the NVRAM component, and showed under various realistic loads that the layout and the technique hides the compaction overhead effectively and increases performance. The work described in this thesis is only a beginning. In this chapter, we conclude by discussing possible future directions and the implications of this work.

6.1 Future Directions

This section describes a number of ways in which this work can be extended in the future, either to improve data management on SMR disks, or to enhance key-value benchmarks with a few more desirable and essential features.

6.1.1 Data Management

Bands Allocations and Band Boundaries

This thesis adopted a fixed-sized band model, with entire band allocations and deallocations. Design possibilities using variable sized bands with partial band allocations and deallocations are promising areas for future explorations. An example data layout with variable sized bands would assign larger bands to hold KVs that does not get modified often, and smaller bands to house the KVs that are modified more often, to improve space utilization while reducing the read and write amplification. Another layout could assign multiple tables to a physical band instead of mapping a table to band entirely. The table sizes could now be smaller and tables with overlapping key ranges, as those described in Chapter 3, could be allocated to the same band. Such an allocation would allow for more overlapping tables, and a delayed and more efficient compaction, as the tables to be merged are physically close to each other.

A mechanism for the disk to decide band boundaries based on some measure of distance between keys would go hand in hand with the different layouts and help improve system performance. For example, if an user stores keys in the space between A-D and starts storing few keys starting with S, if the disk identifies the distance between the keys and start a new band for storing the keys starting with S instead of storing them in the same band that contains the keys starting with D, compaction induced read and write amplification could be reduced.

In a LSM layout, bands in levels other than the bottommost level act as multiple levels of buffer bands. Incorporating hot and cold key identification at the top level would allow for better separation and different handling of the keys. The hot KV pairs could then be stored on bands on a hot level and cold KV pairs on a cold level. Another direction for future work is to look at the physical placement of the buffer bands. For example, Cassuto *et al.* [21] suggested placing a second level of circular buffer bands for each LBA range close to bands storing the same LBA range. Unlike a static mapping and range splits, our dynamic key range mapping would ensure all buffer bands are fully utilized, but our work so far hasn't looked at the physical placement of the buffer bands on disk.

Compactions

In Chapter 3, we showed that the compaction overhead can be reduced by allowing for few overlapping key ranges. In Chapter 5, we showed that write-offloading to NVRAM would effectively hide even a high compaction overhead. We also showed that bursty workloads could trigger compaction immediately and cause a lot of writes to be offloaded even if the system is going to be idle most of the time. Our work could be extended by incorporating workload detection and different levels of sequentiality threshold assignment at different load levels. In other words, compaction could be triggered at various degrees at different load levels.

File System for the Object Disks

Previous research has shown that storing and managing file system metadata and small files on LSM-tree based KV stores work better than traditional file system techniques [113, 94, 116]. Shetty *et al.* built an user-level file system that stores all data blocks in an LSM-tree [105], and BlueSky [120] is a network file system that stores all data in a cloud KV store. These works show that storing the file system data and metadata in an LSM-tree based KV store can be efficient.

Our work raises a new question: Given a KV disk with variable length keys and values that reorganizes itself according to the lexicographic key-order, what is the best mechanism to split file data and how best to name the keys? Key-space assignment would control the data movement on the disk. As an example, say a file A's content is stored in keys A1, A2, and A3, inserting some contents in between A2 and A3 could be as simple as inserting a key A21 and the disk would, at a later point of time, put them all in lexicographic key order for faster access. Files that are expected to be accessed together can be assigned keys that are closer to each other, and assigning a key-space that does not touch any other keys stored on the disk would ensure less data movement.

Multi-Disk Management

It is a common practice to use multiple disks together in a RAID-like setup for reliability purposes. We showed in Chapter 5 that offloading to a disk would also mitigate compaction's impact on performance. In the future, an object-based RAID-like module could extend our work and enable the disks to work with each other to schedule compactions and offload writes to each other in such a way that hides the compaction overhead.

6.1.2 Benchmarking

Scaled-up Realistic Request Arrivals

In Chapter 4, we implemented realistic arrivals in a single YCSB client. To be able to generate requests at an arrival rate high enough for modern high performance storage systems, a single client is not sufficient even with a high number of threads. The approach recommended by YCSB is to use multiple clients at the same time for higher loads.

To generate higher loads in a realistic fashion, our work could be used in a user generative model, where each client chooses a model representative of a distinctive user who shares the underlying storage system with other users. For example, to generate realistic requests in a multi-tenant cloud storage system, each client would model a distinct tenant's access pattern. Our work could be extended in the future to perform multi-client co-ordinated request arrivals, where multiple clients together generate requests conforming to a single model.

Content Generation

Many modern high performance storage systems also perform additional functionalities such as compression and de-duplication to increase the overall throughput of the system. To compare and contrast these functionalities, it would be immensely helpful if standard benchmarks also come with realistic content generation with configurable levels of duplicity and compressibility. For example, YCSB generates values with random bytes and is unsuitable to evaluate such functionalities.

Realistic content generation is not just applicable to stored values, but also the keys in the recently popular variable-length key-value storage systems. The key-value stores, such as LevelDB and SMRDB, strive to order the keys and values as per the lexicographical order of the keys. The throughput of these systems are thus largely dependent on key content, as it can trigger different amounts of background compaction activity, and thus, it is important that benchmarks offer both realistic and configurable key content generation.

Correlation In Request Sizes

Chapter 4 assumed that the request size distribution is independent of the request arrival rate, in line with the observation by Facebook's key-value workload study. But the study may not be representative of all use-cases of the key-value model. More real world workload studies on different kinds of key-value workloads are required to validate the assumption, and if a correlation is found, it has to be incorporated into workload generation.

6.2 Final Thoughts

The primary concern this thesis addresses is that the sequential write requirement imposed by Shingled Magnetic Recording Technology would severely limit performance. This thesis has shown that with a richer interface that eliminates layers of indirection and an efficient SMR-aware object management layer, end user performance need no longer be a concern. Our work has assumed that SMR drives would enforce sequential writes to bands, but its characteristics would otherwise be very similar to existing hard disk drives. But raw banded SMR disks are not yet on the market, and the practical implications of the technology is not yet publicly known. Though our work has taken the first step and eliminated the concerns over sequential writes, it is yet to be seen if there are other practical concerns over the technology that needs to be tackled.

Since our work is backward compatible with and enables high end user performance on existing hard drives, our work is immediately applicable. Further, given that the recent storage shifts have given rise to the release of new generation of object storage drives with a key-value interface, our work is highly relevant, and opens up a new set of possibilities.

Bibliography

[1] Adaptive Memory Technology in Solid State Hybrid Drives. http://www.seagate.com/tech-insights/adaptive-memory-in-sshd-master-ti/.

[2] At attachment 8 - ata/atapi command set. http://t13.org/Documents/UploadedDocuments/docs2008/D1699r6a-ATA8-ACS.pdf.

[3] MapKeeper. `https://github.com/m1ch1/mapkeeper/`.

[4] Cristina L. Abad, Huong Luu, Nathan Roberts, Kihwal Lee, Yi Lu, and Roy H. Campbell. Metadata Traces and Workload Models for Evaluating Big Storage Systems. In *Proceedings of the 2012 IEEE/ACM Fifth International Conference on Utility and Cloud Computing*, UCC '12.

[5] Abutalib Aghayev and Peter Desnoyers. Skylight: A Window on Shingled Disk Operation. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies*, FAST'15, 2015.

[6] Devesh Agrawal, Deepak Ganesan, Ramesh Sitaraman, Yanlei Diao, and Shashi Singh. Lazy-Adaptive Tree: An Optimized Index Structure for Flash Devices. In *Proceedings of the 35th Conference on Very Large Databases*, VLDB '09, August 2009.

[7] Tayfun Akgul, S. Baykut, M. Erol-Kantarci, and S.F. Oktug. Periodicity-Based Anomalies in Self-Similar Network Traffic Flow Measurements. *IEEE Transactions on Instrumentation and Measurement*, 60(4):1358–1366, April 2011.

[8] Ahmad Al-Shishtawy and Vladimir Vlassov. ElastMan: Elasticity Manager for Elastic Key-value Stores in the Cloud. In *Proceedings of the 2013 ACM Cloud and Autonomic Computing Conference*, CAC '13, pages 7:1–7:10, 2013.

[9] Ahmed Amer, JoAnne Holliday, Darrell D. E. Long, Ethan L. Miller, Jehan-François Pâris, and Thomas Schwarz. Data Management and Layout for Shingled Magnetic Recording. *IEEE Transactions on Magnetics*, 47(10):3691–3697, October 2011.

[10] Ahmed Amer, Darrell D. E. Long, Ethan L. Miller, Jehan-Francois Paris, and S.J. Thomas Schwarz. Design Issues for a Shingled Write Disk System. In *Proceedings*

of the 26th IEEE Conference on Mass Storage Systems and Technologies, MSST '10, May 2010.

[11] George Amvrosiadis, Alina Oprea, and Bianca Schroeder. Practical scrubbing: Getting to the bad sector at the right time. In *IEEE/IFIP International Conference on Dependable Systems and Networks*, DSN, pages 1–12, 2012.

[12] David G. Andersen, Jason Franklin, Michael Kaminsky, Amar Phanishayee, Lawrence Tan, and Vijay Vasudevan. FAWN: A Fast Array of Wimpy Nodes. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles*, SOSP '09, October 2009.

[13] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload Analysis of a Large-Scale Key-Value Store. *ACM SIGMETRICS Performance Evaluation Review*, 40(1):53–64, June 2012.

[14] Doug Beaver, Sanjeev Kumar, Harry C. Li, Jason Sobel, and Peter Vajgel. Finding a needle in Haystack: Facebook's photo storage. In *Proceedings of the 9th Symposium on Operating Systems Design and Implementation*, OSDI, October 2010.

[15] Medha Bhadkamkar, Jorge Guerra, Luis Useche, Sam Burnett, Jason Liptak, Raju Rangaswami, and Vagelis Hristidis. BORG: Block-reORGanization for Self-optimizing Storage Systems. In *Proceedings of the 7th USENIX conference on File and Storage Technologies*, FAST '09, 2009.

[16] Timothy Bisson and Scott A. Brandt. Reducing Hybrid Disk Write Latency with Flash-Backed I/O Requests. In *Proceedings of the 2007 15th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, MASCOTS '07, pages 402–409, 2007.

[17] Timothy Bisson, Scott A. Brandt, and Darrell D. E. Long. A Hybrid Disk-Aware Spin-Down Algorithm with I/O Subsystem Support. In *Proceedings of the 26th IEEE International Performance Conference on Computers and Communication*, IPCCC '07, 2007.

[18] Trevor Blackwell, Jeffrey Harris, , and Margo Seltzer. Heuristic Cleaning Algorithms in Log-Structured File Systems. In *Proceedings of the Winter 1995 USENIX Technical Conference*, pages 277–288. USENIX, January 1995.

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

[20] Jin Cao, William S. Cleveland, Dong Lin, and Don X. Sun. On the Nonstationarity of Internet Traffic. In *Proceedings of the 2001 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '01, pages 102–112, 2001.

[21] Yuval Cassuto, Marco A.A. Sanvido, Cyril Guyot, David R. Hall, and Zvonimir Z. Bandic. Indirection Systems for Shingled-Recording Disk Drives. In *Proceedings of the 26th IEEE Conference on Mass Storage Systems and Technologies*, MSST '10, May 2010.

[22] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert 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.

[23] Jianjun Chen, Chris Douglas, Michi Mutsuzaki, Patrick Quaid, Raghu Ramakrishnan, Sriram Rao, and Russell Sears. Walnut: a unified cloud object store. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, pages 743–754, 2012.

[24] Brian F. Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, Hans-Arno Jacobsen, Nick Puz, Daniel Weaver, and Ramana Yerneni. PNUTS: Yahoo!'s hosted data serving platform. In *Proceedings of the 34th Conference on Very Large Databases*, VLDB '08, August 2008.

[25] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, 2010.

[26] Tom Coughlin and Ed Grochowski. 2012 hard disk drive capital equipment market & technology report.

[27] Mark E. Crovella and Azer Bestavros. Self-similarity in World Wide Web Traffic: Evidence and Possible Causes. *IEEE/ACM Transactions on Networking*, 5(6):835–846, December 1997.

[28] Sudipto Das, Shoji Nishimura, Divyakant Agrawal, and Amr El Abbadi. Albatross: Lightweight Elasticity in Shared Storage Databases for the Cloud Using Live Data Migration. *Proceedings of the VLDB Endowment*, 4(8):494–505, May 2011.

[29] Biplob Debnath, Sudipta Sengupta, and Jin Li. FlashStore: High Throughput Persistent Key-Value Store. In *Proceedings of the 36th Conference on Very Large Databases*, VLDB '10, September 2010.

[30] Biplob Debnath, Sudipta Sengupta, and Jin 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.

[31] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's Highly Available Key-value Store. In

Proceedings of the 21st ACM Symposium on Operating Systems Principles, SOSP '07, October 2007.

[32] Christina Delimitrou, Sriram Sankar, Kushagra Vaid, and Christos Kozyrakis. Decoupling Datacenter Studies from Access to Large-scale Applications: A Modeling Approach for Storage Workloads. In *Proceedings of the 2011 IEEE International Symposium on Workload Characterization*, IISWC '11, pages 51–60, 2011.

[33] Aaron J. Elmore, Sudipto Das, Divyakant Agrawal, and Amr El Abbadi. Zephyr: Live Migration in Shared Nothing Databases for Elastic Cloud Platforms. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, SIGMOD '11, pages 301–312, 2011.

[34] Robert Escriva, Bernard Wong, and Emin Gün Sirer. HyperDex: A Distributed, Searchable Key-value Store. In *Proceedings of the ACM SIGCOMM 2012 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '12, pages 25–36, 2012.

[35] Bin Fan, David G. Andersen, and Michael Kaminsky. MemC3: Compact and Concurrent MemCache with Dumber Caching and Smarter Hashing. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, NSDI '13, pages 371–384, 2013.

[36] Dror G. Feitelson. Workload Modeling for Computer Systems Performance Evaluation. This book is being published by Cambridge University Press. `http://www.cs.huji.ac.il/~feit/wlmod/wlmod.pdf`, 2014.

[37] Tim Feldman and Garth Gibson. Shingled Magnetic Recording: Areal Density Increase Requires New Data Management. *;login:*, 38(3), June 2013.

[38] Michael Ferdman, Almutaz Adileh, Onur Kocberber, Stavros Volos, Mohammad Alisafaee, Djordje Jevdjic, Cansu Kaynak, Adrian Daniel Popescu, Anastasia Ailamaki, and Babak Falsafi. Clearing the Clouds: A Study of Emerging Scale-out Workloads on Modern Hardware. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, pages 37–48, 2012.

[39] Sanjay Ghemawat and Jeff Dean. LevelDB. `https://github.com/google/leveldb`, 2015.

[40] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google File System. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03)*, October 2003.

[41] Garth Gibson and Greg Ganger. Principles of Operation for Shingled Disk Devices. Technical Report CMU-PDL-11-107, Carnegie Mellon University, 2011.

[42] Richard Golding, Peter Bosch, Carl Staelin, Tim Sullivan, and John Wilkes. Idleness is not sloth. In *Proceedings of the Winter 1995 USENIX Technical Conference*, pages 201–212, January 1995.

[43] Maria E. Gomez and Vicente Santonja. Self-Similarity in I/O Workload: Analysis and Modeling. In *Proceedings of the Workload Characterization: Methodology and Case Studies*, WWC '98, 1998.

[44] Simon Greaves, Yasushi Kanai, and Hiroaki Muraoka. Shingled Recording for 2–3 Tbit/in^2. *IEEE Transactions on Magnetics*, 45(10):3823–3829, Oct. 2009.

[45] Simon Greaves, Yasushi Kanai, and Hiroaki Muraoka. Shingled Magnetic Recording on Bit Patterned Media. *IEEE Transactions on Magnetics*, 46(6):1460–1463, June 2010.

[46] Steven D. Gribble, Gurmeet Singh Manku, Drew Roselli, Eric A. Brewer, Timothy J. Gibson, and Ethan L. Miller. Self-similarity in File Systems. In *Proceedings of the 1998 ACM SIGMETRICS Joint International Conference on Measurement and Modeling of Computer Systems*, pages 141–150, 1998.

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

[48] HBase. `https://hbase.apache.org/`, 2015.

[49] Weiping He and David H. C. Du. Novel Address Mappings for Shingled Write Disks. In *Proceedings of the 6th USENIX Conference on Hot Topics in Storage and File Systems*, HotStorage'14, 2014.

[50] Bo Hong and Tara M. Madhyastha. The Relevance of Long-Range Dependence in Disk Traffic and Implications for Trace Synthesis. In *Proceedings of the 22Nd IEEE / 13th NASA Goddard Conference on Mass Storage Systems and Technologies (MSST '05)*, pages 316–326, 2005.

[51] Jen-Wei Hsieh, Tei-Wei Kuo, and Li-Pin Chang. Efficient identification of hot data for flash memory storage systems. *ACM Transactions on Storage*, 2(1):22–40, February 2006.

[52] Windsor W. Hsu, Alan Jay Smith, and Honesty C. Young. The Automatic Improvement of Locality in Storage Systems. *ACM Transactions on Computer Systems*, 23(4):424–473, November 2005.

[53] Chao Jin, Wei-Ya Xi, Zhi-Yong Ching, Feng Huo, and Chun-Teck Lim. Hismrfs: A high performance file system for shingled storage array. In *2014 30th Symposium on Mass Storage Systems and Technologies*, MSST '14, 2014.

[54] Rize Jin, Se Jin Kwon, and Tae-Sun Chung. FlashB-tree: a novel B-tree index scheme for solid state drives. In *Proceedings of the 2011 ACM Symposium on Research in Applied Computation*, RACS '11, pages 50–55, 2011.

[55] Stephanie Jones, Ahmed Amer, Ethan L. Miller, Darrell D. E. Long, Rekha Pitchumani, and Christina Strong. Classifying Data to Reduce Long Term Data

Movement in Shingled Write Disks. In *Proceedings of the 31st International Conference on Massive Storage Systems and Technology*, MSST 2015, June 2015.

[56] Jaemin Jung, Youjip Won, Eunki Kim, Hyungjong Shin, and Byeonggil Jeon. FRASH: Exploiting Storage Class Memory in Hybrid File System for Hierarchical Storage. *ACM Transactions on Storage*, 6(1):3:1–3:25, April 2010.

[57] Dongwon Kang, Dawoon Jung, Jeong-Uk Kang, and Jin-Soo Kim. μ-Tree : An Ordered Index Structure for NAND Flash Memory. In *7th ACM & IEEE Conference on Embedded Software*, EMSOFT '07, pages 144–153, 2007.

[58] Yangwook Kang, Thomas Marlette, Ethan L. Miller, and Rekha Pitchumani. Muninn: a Versioning Key-Value Store using Object-based Storage Model. In *Proceedings of the 7th International Systems and Storage Conference*, SYSTOR Õ14, June 2014.

[59] Yangwook Kang, Yang suk Kee, Ethan L. Miller, and Chanik Park. Enabling Cost-effective Data Processing with Smart SSD. In *the 29th IEEE Symposium on Massive Storage Systems and Technologies*, MSST '13, May 2013.

[60] Yangwook Kang, Jingpei Yang, and Ethan L. Miller. Object-based SCM: An Efficient Interface for Storage Class Memories. In *Proceedings of the 27th IEEE Conference on Mass Storage Systems and Technologies*, MSST '11, May 2011.

[61] Sudarsun Kannan, Ada Gavrilovska, Karsten Schwan, and Dejan Milojicic. Optimizing Checkpoints Using NVM as Virtual Memory. In *IEEE 27th International Symposium on Parallel Distributed Processing*, IPDPS, pages 29–40, May 2013.

[62] Sudarsun Kannan, Ada Gavrilovska, Karsten Schwan, Dejan Milojicic, and Vanish Talwar. Using Active NVRAM for I/O Staging. In *Proceedings of the 2nd International Workshop on Petascal Data Analytics: Challenges and Opportunities*, PDAC '11, pages 15–22, 2011.

[63] Thomas Karagiannis, Michalis Faloutsos, and Mart Molle. A User-friendly Self-similarity Analysis Tool. *ACM SIGCOMM Computer Communication Review*, 33(3):81–93, July 2003.

[64] Thomas Karagiannis, Michalis Faloutsos, and Rudolff H. Riedi. Long-range dependence: now you see it, now you don't! In *Global Telecommunications Conference*, volume 3 of *GLOBECOM '02*, pages 2165–2169, Nov 2002.

[65] Avinash Lakshman and Prashant Malik. Cassandra: A Decentralized Structured Storage System. *ACM SIGOPS Operating Systems Review*, 44(2):35–40, April 2010.

[66] Will E. Leland, Murad S. Taqqu, Walter Willinger, and Daniel V. Wilson. On the Self-similar Nature of Ethernet Traffic (Extended Version). *IEEE/ACM Transactions on Networking*, 2(1):1–15, February 1994.

[67] Yinan Li, Bingsheng He, Robin Jun Yang, Qiong Luo, and Ke Yi. Tree indexing on solid state drives. *Proceedings of the VLDB Endowment*, 3(1-2):1195–1206, September 2010.

[68] Hyeontaek Lim, Bin Fan, David G. Andersen, and Michael Kaminsky. SILT: A Memory-Efficient, High-Performance Key-Value Store. In *Proceedings of the 23rd ACM Symposium on Operating System Principles*, SOSP, October 2011.

[69] Chung-I Lin, Dongchul Park, Weiping He, and David Du. H-SWD: Incorporating Hot Data Identification into Shingled Write Disks. In *Proceedings of the 20th IEEE International Symposium on Modeling, Analysis and Simulations of Computer and Telecommunication Systems*, MASCOTS '12, pages 248 – 255, August 2012.

[70] Christopher R. Lumb, Jiri Schindler, Gregory R. Ganger, David F. Nagle, and Erik Riedel. Towards Higher Disk Head Utilization: Extracting Free Bandwidth from Busy Disk Drives. In *Proceedings of the 4th Conference on Symposium on Operating System Design & Implementation - Volume 4*, OSDI '00, 2000.

[71] Dan Luo, Jiguang Wan, Yifeng Zhu, Nannan Zhao, Feng Li, and Changsheng Xie. Design and implementation of a hybrid shingled write disk system. *Parallel and Distributed Systems, IEEE Transactions on*, 2015.

[72] Peter Macko, Margo Seltzer, and Keith A. Smith. Tracking Back References in a Write-Anywhere File System. In *Proceedings of the 8th USENIX conference on File and Storage Technologies*, FAST '10, 2010.

[73] Koji Matsumoto, Akihiro Inomata, and Shinya Hasegawa. Thermally Assisted Magnetic Recording. *Fujitsu Scientific and Technical Journal*, 42(1):158–167, Jan. 2006.

[74] Jeanna Neefe Matthews, Drew Roselli, Adam M. Costello, Randolph Y. Wang, and Thomas E. Anderson. Improving the Performance of Log-Structured File Systems with Adaptive Methods. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, SOSP '97, pages 238–251, October 1997.

[75] David Meisner, Junjie Wu, and Thomas F. Wenisch. BigHouse: A Simulation Infrastructure for Data Center Systems. In *IEEE International Symposium on Performance Analysis of Systems and Software*, ISPASS '12, pages 35–45, 2012.

[76] Mike Mesnier, Gregory R. Ganger, and Erik Riedel. Object-Based Storage. *IEEE Communications Magazine*, 41(8), August 2003.

[77] Ningfang Mi, Giuliano Casale, Ludmila Cherkasova, and Evgenia Smirni. Injecting Realistic Burstiness to a Traditional Client-server Benchmark. In *Proceedings of the 6th International Conference on Autonomic Computing*, ICAC '09, pages 149–158, 2009.

[78] Ningfang Mi, Alma Riska, Qi Zhang, Evgenia Smirni, and Erik Riedel. Efficient Management of Idleness in Storage Systems. *ACM Transactions on Storage*, 5(2):4:1–4:25, June 2009.

[79] Ethan L. Miller, Scott A. Brandt, and Darrell D. E. Long. HeRMES: High-Performance Reliable MRAM-Enabled Storage. In *Proceedings of the 8th IEEE Workshop on Hot Topics in Operating Systems*, HotOS-VIII, pages 83–87, Schloss Elmau, Germany, May 2001.

[80] Changwoo Mina, Kangnyeon Kimb, Hyunjin Choc, Sang-Won Leed, and Young Ik Eome. SFS: Random Write Considered Harmful in Solid State Drives. In *Proceedings of the 10th USENIX conference on File and Storage Technologies*, FAST '12, 2012.

[81] Damien Le Moal, Zvonimir Bandic, and Cyril Guyot. Shingled File System Host-Side Management of Shingled Magnetic Recording Disks. In *IEEE International Conference on Consumer Electronics*, 2012.

[82] Dushyanth Narayanan, Austin Donnelly, and Antony Rowstron. Write Off-Loading: Practical Power Management for Enterprise Storage. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies*, FAST '08, pages 253–267, February 2008.

[83] Suman Nath and Aman Kansal. FlashDB: dynamic self-tuning database for NAND flash. In *Proceedings of the 6th international conference on Information processing in sensor networks*, IPSN '07, pages 410–419, 2007.

[84] Richard M. H. New and Mason Lamar Williams. Log-Structured File System for Disk Drives with Shingled Writing. United States Patent 7996645.

[85] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. The Log-Structured Merge-Tree (LSM-Tree). *Acta Informatica*, 33:351–385, 1996.

[86] Open-Kinetic. The Kinetic Open Storage Project. http://www.openkinetic.org/index.php?title=Kinetic_Open_Storage_Group, 2015.

[87] Swapnil Patil, Milo Polte, Kai Ren, Wittawat Tantisiriroj, Lin Xiao, Julio López, Garth Gibson, Adam Fuchs, and Billie Rinaldi. YCSB++: Benchmarking and Performance Debugging Advanced Features in Scalable Table Stores. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, SOCC '11, pages 9:1–9:14, 2011.

[88] Vern Paxson and Sally Floyd. Wide Area Traffic: The Failure of Poisson Modeling. *IEEE/ACM Transactions on Networking*, 3(3):226–244, June 1995.

[89] Hannes Payer, Marco A.A. Sanvido, Zvonimir Z. Bandic, and Christoph M. Kirsch. Combo Drive: Optimizing Cost and Performance in a Heterogeneous Storage Device. In *Proceedings of the Workshop on Integrating Solid-state Memory into the Storage Hierarchy (WISH), co-located with ASPLOS*, 2009.

[90] Gregory Piatetsky-Shapiro and Charles 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.

[91] S.N. Piramanayagam and K. Srinivasan. Recording media research for future hard disk drives. *Journal of Magnetism and Magnetic Materials*, 321(6):485 – 494, 2009.

[92] Rekha Pitchumani, Andy Hospodor, Ahmed Amer, Yangwook Kang, Ethan L. Miller, and Darrell D. E. Long. Emulating a Shingled Write Disk. In *Proceedings of the 20th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, MASCOTS '12, August 2012.

[93] Sheng Qiu and A. L. Narasimha Reddy. NVMFS: A Hybrid File System for Improving Random Write in NAND-flash SSD. *IEEE / NASA Goddard Conference on Mass Storage Systems and Technologies*, 2013.

[94] Kai Ren and Garth Gibson. TABLEFS: Embedding a NoSQL Database Inside the Local File System. In *IEEE Asia-Pacific Magnetic Recording Conference*, APMRC 2012, 2012.

[95] H.J. Richter, A.Y. Dobin, O. Heinonen, K.Z. Gao, R.J.Mvd. Veerdonk, R.T. Lynch, J. Xue, D. Weller, P. Asselin, M.F. Erden, and R.M. Brockie. Recording on Bit-Patterned Media at Densities of 1 Tb/in^2 and Beyond. *IEEE Transactions on Magnetics*, 42(10):2255–2260, Oct. 2006.

[96] Alma Riska and Erik Riedel. Long-Range Dependence at the Disk Drive Level. In *Proceedings of the 3rd International Conference on the Quantitative Evaluation of Systems*, QEST '06, pages 41–50, 2006.

[97] Ohad Rodeh. B-trees, Shadowing, and Clones. *ACM Transactions on Storage*, 3(4):15:1–15:27, February 2008.

[98] Mendel Rosenblum and John K. Ousterhout. The Design and Implementation of a Log-Structured File System. *ACM Transactions on Computer Systems*, 10(1):26–52, February 1992.

[99] Robert E. Rottmeyer, Sharat Batra, Dorothea Buechel, William A. Challener, Julius Hohlfeld, Yukiko Kubota, Lei Li, Bin Lu, Cristophe Mihalcea, Keith Mountfiled, Kalman Pelhos, Peng Chubing, Tim Rausch, Michael A. Seigler, Dieter Weller, and Yang Xiaomin. Heat-Assisted Magnetic Recording. *IEEE Transactions on Magnetics*, 42(10):2417–2421, Oct. 2006.

[100] Matthew Roughan, Albert Greenberg, Charles Kalmanek, Michael Rumsewicz, Jennifer Yates, and Yin Zhang. Experience in Measuring Backbone Traffic Variability: Models, Metrics, Measurements and Meaning. In *Proceedings of the 2nd ACM SIGCOMM Workshop on Internet Measurement*, IMW '02, pages 91–92, 2002.

[101] 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.

[102] Russell Sears and Raghu 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.

[103] Margo Seltzer, Keith Bostic, M. Kirk McKusick, and Carl Staelin. An implementation of a log-structured file system for UNIX. In *Proceedings of the Winter 1993 USENIX Technical Conference*, pages 307–326, January 1993.

[104] Margo Seltzer, Keith A. Smith, Hari Balakrishnan, Jacqueline Chang, Sara McMains, and Venkata Padmanabhan. File System Logging versus Clustering: A Performance Comparison. In *Proceedings of the Winter 1995 USENIX Technical Conference*, pages 249–264, 1995.

[105] Pradeep Shetty, Richard Spillane, Ravikant Malpani, Binesh Andrews, Justin Seyster, and Erez Zadok. Building Workload-Independent Storage with VT-Trees. In *Proceedings of the 11th USENIX Conference on File and Storage Technologies*, FAST'13, 2013.

[106] Liang Shi, Jianhua Li, Chun Jason Xue, and Xuehai Zhou. Hybrid Nonvolatile Disk Cache for Energy-efficient and High-performance Systems. *ACM Transactions on Design Automation of Electronic Systems*, pages 8:1–8:23, Jan. 2013.

[107] Ji Yong Shin, Mahesh Balakrishnan, Tudor Marian, and Hakim Weatherspoon. Gecko: Contention-Oblivious Disk Arrays for Cloud Storage. In *Proceedings of the 11th USENIX Conference on File and Storage Technologies*, FAST '13, 2013.

[108] Y. Shiroishi, K. Fukuda, I. Tagawa, S. Takenoiri, H. Tanaka, and N. Yoshikawa. Future options for HDD storage. *IEEE Transactions on Magnetics*, 45(10), Oct. 2009.

[109] David Shue, Michael J. Freedman, and Anees Shaikh. Performance Isolation and Fairness for Multi-tenant Cloud Storage. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation (OSDI '12)*, pages 349–362, 2012.

[110] Muthian Sivathanu, Vijayan Prabhakaran, Florentina I. Popovici, Timothy E. Denehy, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dussea. Semantically-Smart Disk Systems. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies*, FAST '03, 2003.

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

[112] Gokul Soundararajan, Vijayan Prabhakaran, Mahesh Balakrishnan, and Ted Wobber. Extending SSD Lifetimes with Disk-based Write Caches. In *Proceedings of the 8th USENIX Conference on File and Storage Technologies*, FAST'10, 2010.

[113] Jan Stender, Björn Kolbeck, Mikael Högqvist, and Felix 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.

[114] Anand Suresh, Garth Gibson, and Greg Ganger. Shingled Magnetic Recording for Big Data Applications. Technical Report CMU-PDL-12-105, Carnegie Mellon University, May 2012.

[115] Eno Thereska, Jiri Schindler, John Bucy, Brandon Salmon, Christopher R. Lumb, and Gregory R. Ganger. A Framework for Building Unobtrusive Disk Maintenance Applications. In *Proceedings of the 3rd USENIX Conference on File and Storage Technologies*, FAST '04, 2004.

[116] Richard van Heuven van Staereling, Raja Appuswamy, David C. van Moolenbroek, and Andrew S. Tanenbaum. Efficient, Modular Metadata Management with Loris. In *Networking, Architecture and Storage, 2011 6th IEEE International Conference on*, NAS, 2011.

[117] Kalyana Sundaram Venkataraman, Guiqiang Dong, and Tong Zhang. Techniques Mitigating Update-Induced Latency Overhead in Shingled Magnetic Recording. *IEEE Transactions on Magnetics*, 48(5):1899–1905, May 2012.

[118] Hoang Tam Vo, Sheng Wang, Divyakant Agrawal, Gang Chen, and Beng Chin Ooi. LogBase: A Scalable Log-structured Database System in the Cloud. *Proceedings of the VLDB Endowment*, 5(10):1004–1015, June 2012.

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

[120] Michael Vrable, Stefan Savage, and Geoffrey M. Voelker. BlueSky: A Cloud-Backed File System for the Enterprise. In *Proceedings of the 10th USENIX conference on File and Storage Technologies*, FAST'12, 2012.

[121] An-I Andy Wang, Geoff Kuenning, Peter Reiher, and Gerald Popek. The Conquest File System: Better Performance Through a Disk/Persistent-RAM Hybrid Design. *ACM Transactions on Storage*, 2(3):309–348, 2006.

[122] Jun Wang and Yiming Hu. PROFS-Performance-Oriented Data Reorganization for Log-structured File System on Multi-Zone Disks. In *Proceedings of the 9th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, MASCOTS '01.

[123] Jun Wang and Yiming Hu. WOLF—A Novel Reordering Write Buffer to Boost the Performance of Log-Structured File Systems. In *Proceedings of the Conference on File and Storage Technologies*, FAST, 2002.

[124] Mengzhi Wang, Ngai Hang Chan, Spiros Papadimitriou, Christos Faloutsos, and Tara Madhyastha. Data Mining Meets Performance Evaluation: Fast Algorithms for Modeling Bursty Traffic. In *Proceedings of the 18th International Conference on Data Engineering*, ICDE '02, 2002.

[125] Peng Wang, Guangyu Sun, Song Jiang, Jian Ouyang, Shiding Lin, Chen Zhang, and Jason Cong. An Efficient Design and Implementation of LSM-tree Based Key-value Store on Open-channel SSD. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, 2014.

[126] Wenguang Wang, Yanping Zhao, and Rick Bunt. HyLog: A High Performance Approach to Managing Disk Layout. In *Proceedings of the 3rd USENIX Conference on File and Storage Technologies*, FAST '04, 2004.

[127] Ralph O. Weber. Information Technology—SCSI Object-Based Storage Device Commands (OSD). Technical Council Proposal Document T10/1355-D, Technical Committee T10, August 2002.

[128] Roger Wood, Mason Williams, Aleksandar Kavcic, and Jim Miles. The Feasibility of Magnetic Recording at 10 Terabits Per Square Inch on Conventional Media. *IEEE Transactions on Magnetics*, 45(2):917 –923, Feb. 2009.

[129] Chin-Hsien Wu, Tei-Wei Kuo, and Li Ping Chang. An Efficient B-Tree Layer Implementation for Flash-Memory Storage Systems . *ACM Transactions on Embedded Computer Systems*, 6(3), July 2007.

[130] Feng Yan, Alma Riska, and Evgenia Smirni. Busy Bee: How to Use Traffic Information for Better Scheduling of Background Tasks. In *Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering*, ICPE '12, 2012.

[131] Yiying Zhang, Leo Prasath Arulraj, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. De-indirection for flash-based SSDs with nameless writes. In *Proceedings of the 10th USENIX conference on File and Storage Technologies*, FAST'12, 2012.

[132] Jian-Gang Zhu, Xiaochun Zhu, and Yuhui Tang. Microwave Assisted Magnetic Recording. *IEEE Transactions on Magnetics*, 44:125–131, Jan. 2008.