OPTIMIZING KEY-VALUE STORES FOR HYBRID STORAGE ENVIRONMENTS

by

Prashanth Menon

A thesis submitted in conformity with the requirements for the degree of Master of Applied Science
Graduate Department of Electrical and Computer Engineering
University of Toronto

© Copyright 2015 by Prashanth Menon
Abstract

Optimizing Key-Value Stores For Hybrid Storage Environments

Prashanth Menon
Master of Applied Science
Graduate Department of Electrical and Computer Engineering
University of Toronto
2015

Modern write-intensive key-value stores have emerged as the prevailing data storage system for many big applications. However, these systems often sacrifice their read performance to cope with high data ingestion rates. Solid-state drives (SSD) can lend their help, but the volume of data and the peculiar characteristics of SSDs make their exclusive use uneconomical. Hence, hybrid storage environments with both SSDs and hard-disk drives (HDD) present interesting research opportunities for optimization.

This thesis investigates how to design and optimize key-value stores for hybrid storage environments. We first modify and extend an existing key-value store to leverage the SSD as a cache. We next formulate an analytical cost model to predict the performance of a generic log-structured hybrid system. Finally, we use insights from our model to design and implement a new hybrid-optimized key-value store, LogStore. We demonstrate that LogStore is up to 7x faster than LevelDB, a state-of-the-art key-value store.
Acknowledgements

I first and foremost extend my sincerest gratitude to my supervisor, Prof. Hans-Arno Jacobsen. Arno always provided thoughtful advice and guidance, and was a voice of reason. Arno had a way to cut straight to the point, to ask the right question to elicit the precise answer he sought. Ever supportive and always of the mind that all my work can be completed in a matter of hours, Arno kept me motivated entirely by his belief in my success. Arno has equipped me with the tools to conduct successful research.

I would like to thank several members of the Middleware Systems Research Group (MSRG) with whom I’ve had nonsensical, comical, thoughtful and philosophical discussions that enriched my graduate life. I would like to thank Tilmann Rabl, Kaiwen Zhang, Mo Sadoghi, Kianoosh Mokhtarian and each of the various summer interns that have passed through our doors. I would like to especially thank Tilmann and Mo whom have been my primary collaborators, and with whom I’ve learned the most about conducting novel and fruitful research. MSRG is a close-knit group; we have travelled together and that inevitably brings people very close. I will cherish the memories of our fantastical retreats.

Lastly, I dedicate this manuscript to my parents. It is through their sacrifice that I was afforded the privilege to pursue graduate school, and it is because of their unconditional love and unending encouragement that I was able to complete this first step.
Contents

1 Introduction ........................................... 1
   1.1 Problem Statement .................................. 3
   1.2 Contributions ....................................... 3
   1.3 Thesis Structure ..................................... 4

2 Background ........................................... 5
   2.1 LSM Trees ........................................... 5
   2.2 SSD Characteristics .................................. 7
   2.3 SSD Price Performance Ratio ......................... 8

3 Related Work ......................................... 10

4 Optimizing Cassandra For Hybrid Storage .......... 13
   4.1 Read and Write Path ................................ 14
   4.2 Extending the Row Cache on SSDs .................... 15
   4.3 Dynamic Schema Catalog .............................. 17
   4.4 Evaluation ........................................... 20
      4.4.1 SSD Row Cache Performance ....................... 25
      4.4.2 Dynamic Schema Catalog Performance ........... 26

5 Log-Structured Hybrid Storage Cost Model .......... 29
   5.1 Hybrid Storage Performance Analysis ............... 29
5.2 Hybrid Storage Cost Model ........................................... 31
  5.2.1 Read-Only Model ............................................. 32
  5.2.2 Write-Only Model ............................................ 34
  5.2.3 Read-Write Model ............................................ 36
  5.2.4 Discussion of the model .................................... 37

6 LogStore .......................................................... 39
  6.1 LogStore Architecture ........................................ 39
    6.1.1 Informed Compaction ...................................... 42
    6.1.2 Reverse Compaction ....................................... 44
    6.1.3 Write Path Optimization .................................. 46
    6.1.4 Staging Compactions ...................................... 47
  6.2 Evaluation .................................................... 48
    6.2.1 Read-Only Performance .................................... 50
    6.2.2 Write-Only Performance .................................. 56
    6.2.3 Mixed Read-Write Performance ............................ 61

7 Conclusions and Future Work .................................. 69
  7.1 Conclusions .................................................. 69
  7.2 Future Work ................................................ 70

Bibliography ....................................................... 72

Appendices ........................................................ 77

A Optimizing LevelDB for SSD ................................... 78
List of Tables

4.1 Experiment configurations .......................................................... 20

6.1 Workload specifications .............................................................. 49

6.2 Analytical and experimental throughput for read-write workload for LDB-HDD, LDB-SSD and LogStore ........................................... 67
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Log-structured storage</td>
<td>6</td>
</tr>
<tr>
<td>4.1</td>
<td>Row cache read and write path (extensions highlighted)</td>
<td>15</td>
</tr>
<tr>
<td>4.2</td>
<td>Redundant vs. dynamic schema</td>
<td>18</td>
</tr>
<tr>
<td>4.3</td>
<td>HDD vs SDD throughput results</td>
<td>22</td>
</tr>
<tr>
<td>4.4</td>
<td>HDD vs SDD latency results</td>
<td>22</td>
</tr>
<tr>
<td>4.5</td>
<td>HDD vs SDD throughput for 99%-filled disks</td>
<td>23</td>
</tr>
<tr>
<td>4.6</td>
<td>HDD vs SDD latency for 99%-filled disks</td>
<td>23</td>
</tr>
<tr>
<td>4.7</td>
<td>Throughput of row cache across workloads</td>
<td>25</td>
</tr>
<tr>
<td>4.8</td>
<td>Latency of row cache across workloads</td>
<td>26</td>
</tr>
<tr>
<td>4.9</td>
<td>Throughput of dynamic schema across workloads</td>
<td>27</td>
</tr>
<tr>
<td>4.10</td>
<td>Latency of dynamic schema across workloads</td>
<td>28</td>
</tr>
<tr>
<td>5.1</td>
<td>Zipfian vs. uniform distribution</td>
<td>30</td>
</tr>
<tr>
<td>5.2</td>
<td>Data placement on SSD and HDD</td>
<td>30</td>
</tr>
<tr>
<td>6.1</td>
<td>LogStore architecture</td>
<td>40</td>
</tr>
<tr>
<td>6.2</td>
<td>LogStore informed compaction</td>
<td>42</td>
</tr>
<tr>
<td>6.3</td>
<td>Throughput for read-only workload with varying RAM sizes</td>
<td>50</td>
</tr>
<tr>
<td>6.4</td>
<td>Latency for read-only workload with varying RAM sizes</td>
<td>51</td>
</tr>
<tr>
<td>6.5</td>
<td>Throughput of LogStore in read-only workload with varying SSD usage</td>
<td>51</td>
</tr>
<tr>
<td>6.6</td>
<td>Throughput of each system over time</td>
<td>54</td>
</tr>
</tbody>
</table>
6.7 Analytical and experimental throughput for read-only workload with 8GB RAM ......................................................... 55
6.8 Throughput for write-only workload ........................................ 57
6.9 Latency for write-only workload ............................................ 57
6.10 Throughput of write-only workload over time ....................... 59
6.11 Throughput for mixed read-write workload ......................... 61
6.12 Latency for mixed read-write workload ............................. 62
6.13 Percentage of reads serviced by each level in LogStore .......... 64

A.1 Number of performed compactions in LevelDB and our optimized LevelDB in a mixed read-write workload ....................... 79
A.2 Distribution of read accesses across levels in LevelDB and our optimized LevelDB in a mixed read-write workload ................... 80
A.3 Throughput of LevelDB and our optimized LevelDB in a mixed read-write workload ...................................................... 81
Chapter 1

Introduction

Modern big data applications are characterized by the large volume of data they need to store, the high velocity at which this data is produced and received, and the unstructured nature of the data itself [21]. Moreover, the data needs to be readily available for continuous, efficient and performant query processing. Examples of such applications include traffic monitoring [30, 28], smart grid applications [19], and application performance management [23, 17, 29]. The common requirement among these applications is the need for a write-intensive data store (e.g., [12, 20, 16, 32]) that can also sustain fast querying traffic. Current approaches to this problem either try to exploit the speed of memory-based solutions, thus, drastically limiting the amount of data that can be stored (in order to remain cost-effective) or rely on disk-based approaches and, thus, only store a small sample of the required data to cope with the high data velocity and volume [29].

While traditional relational databases have been the main workhorse behind many data-intensive applications, the recent data explosion has made it increasingly challenging to deploy them in a cost-effective way so as to sustain the required performance. As a result, a new array of lightweight and economical alternatives have emerged, such as the performant key-value stores [31, 26, 5, 12, 20, 16, 32]. These key-value stores are able to offer high write rates primarily by two means: partitioning and choice of data-
structure. Key-value stores often partition the entire data-set across a cluster of nodes, where each node handles only a subset of the overall traffic. Within a single node, write-intensive data structures are employed that often avoid random disk access when writing a record in favour of fast sequential accesses. This design choice sacrifices read performance by having to perform multiple random disk accesses to read a record. For big data applications that require fast query performance, this presents a problem. A common, albeit naive, solution to address the poor read performance of write-intensive key-value stores is to replace very slow conventional hard-disk drives (HDD) with much faster solid-state drives (SSD) that offer dramatically improved random read speeds. Existing consumer-grade SSDs deliver between 30x-40x faster random read speeds versus their magnetic HDD counterparts. The difference is even more pronounced when using faster PCIe-based SSDs.

SSDs have experienced a coming of age in the enterprise market due to their low energy consumption, smaller size, and their significantly improved performance in comparison to traditional hardware [35, 22]. SSDs are frequently used as a drop-in replacement for HDDs, but this approach will often yield suboptimal results from both the software and hardware. Classical data-structures and algorithms that work on HDDs should not be directly applied to SSDs due to the different storage characteristics of the device. For example, modern SSDs have asymmetric read/write performance, where random reads are much faster than random writes. Moreover, the current market price of SSD technology and low bit-density makes their exclusive use uneconomical for most big data applications. This has prompted both industry and academia to explore new hybrid storage approaches (e.g., [7, 10, 16]), in which modern and traditional technologies are colocated to form a storage memory hierarchy. This new storage hierarchy, along with the requirement for ever increasing performance from key-value stores, presents a very interesting research opportunity, on which this thesis focuses on.
1.1 Problem Statement

The goal of this thesis is to explore how key-value stores can be designed, implemented and optimized for hybrid storage environments that include both SSDs and HDDs. Key-value stores play a pivotal role as the main storage system in many big data applications, but were largely designed for homogenous storage. With the advent of SSDs, built on the philosophy of no moving parts, hybrid storage environments are increasingly prevalent (e.g., [7, 10, 16]). This new paradigm requires a more hardware-conscious redesign of storage system software that can efficiently take advantage of the characteristics of each storage device to provide optimal performance. This thesis focuses on two general approaches to optimizing key-value stores for hybrid storage environments: using available SSDs as an extension to memory to cache data and to place SSDs as part of the storage hierarchy where data is stored on exactly one storage device with no cached copies (except in main memory).

1.2 Contributions

To address the aforementioned problem, this thesis makes the following contributions:

1. In our first contribution, CaSSanDra, we propose two extensions to a popular class of key-value stores that cache certain data and schema information on an SSD device. We describe the implementation of each technique and evaluate the modified system across a variety of representative workloads to demonstrate significant improvement in read throughput and reduction in total space usage.

2. Our second contribution is an analytical cost model to estimate the performance of generic log-structured hybrid storage systems. The model accounts for access pattern, data layout and specific system characteristics to accurately predict (with a lower bound) the performance that can be expected from a log-structured hybrid
Chapter 1. Introduction

storage system. Additionally, the analytical cost model provides insight that help
guide the design of LogStore and reveals crucial bottlenecks in LevelDB.

3. Finally, our last contribution is LogStore, a novel, cost-based log-structured storage
system designed and optimized for hybrid storage environments. In LogStore recent
changes are first stored on SSDs, and as the data ages, it is pushed to HDD, while
minimizing the read and write amplification for merging and compaction of data
from SSDs and HDDs. We also ensure that all writes on both SSD and HDD
are sequential in large block sizes. Furthermore, we develop a holistic approach
to improve both read and write performance by dynamically optimizing the data
layout based on the observed access patterns.

The first contribution can be found in [24] and the latter two contributions can be
found in [25].

1.3 Thesis Structure

The remainder of this thesis is divided into seven chapters. Chapter 2 provides back-
ground on the characteristics of the SSD devices and on the architecture used by most
key-value stores. Chapter 3 examines some of the related work in the area. Chapter 4
explores two techniques for optimizing extensible row-stores within a hybrid storage en-
vironment and conducts an evaluation of the techniques. Chapter 5 formulates an ana-
lytical cost model to estimate the performance of a generic log-structured hybrid storage
system. Chapter 6 presents the architecture of LogStore along with a thorough evaluation
across a variety of workloads. In Chapter 7, we draw our conclusions and present an
outlook on future work.
Chapter 2

Background

We begin this chapter by describing the Log-Structured Merge Tree (LSM-tree) data-structure that underpins many of the most popular key-value stores currently available (e.g., [2], [1], [3], [4], [12], [20], [16] and [34]). We follow this discussion with a description of the specific characteristics of modern SSD devices and why LSM-trees operate well on them. We stress that though SSDs have attractive performance numbers, SSD-only installations will provide only suboptimal performance unless its specific characteristics are taken advantage of. We finalize this chapter by providing a cost analysis demonstrating that SSD-only installations often yield suboptimal price-performance numbers as well. This leads to the adoption of hybrid storage environments.

2.1 LSM Trees

The LSM-tree, introduced in [26], is one of the most well-know write-intensive data-structures. An LSM-tree is actually a collection of exponentially growing B-Trees, the smallest of which is always memory-resident. Insertions are batched and bulk loaded into the smallest B-Tree in memory; as the smaller tree fill up, the data is bulk loaded into the next level, larger B-Tree in a purely sequential manner. This rolling merge process continues as each component B-Tree reaches its threshold size. In this way, an LSM-
tree gradually spread insertions into the structure across this set of component trees. Unlike the conventional B-Tree that must perform random I/O to update index pages, the LSM-tree requires only bulk sequential I/O and random reads.

Unlike the LSM-tree, which makes use of multiple layers of B-Trees, modern key-value stores (e.g., [12, 20, 2]) rely on *sorted string tables* (SST). SSTs are small, fixed sized, immutable files that store key and value pairs in a sorted fashion. SSTs are organized in levels and at each level, SSTs are non-overlapping. The size of the levels (i.e., the number of SSTs per level) grows exponentially. In systems like BigTable, Cassandra and LevelDB the levels typically grow by a factor of 10. In Figure 2.1, a schematic of the exponential structure can be seen.

Data is initially inserted into the *Memtable*, an in-memory data structure that supports in-place updates, and appended to a durable *commit log* ([12, 20, 2]). As soon as the Memtable reaches a configurable threshold size, it is converted into an SST and is pushed to the next level. On each level, a record is stored in exactly one SST, but different versions of a record may be present at different levels. This means that when an SST is pushed down, it has to be merged with other SSTs on the next level. Since SSTs are immutable, all overlapping SSTs are loaded into memory and reorganized into a new set of non-overlapping SSTs. This process can cascade through multiple levels if multiple level overflow after an insert or update. However, due to the exponential growth of the levels, the update frequency in a higher level is only logarithmic to the frequency in the previous level. An SST that is merged to the next level, on average it overlaps with 10
SSTs on the next level (given that the levels cover the same key space and the updates are uniform).

To retrieve a record, the latest version has to be retrieved. In key-value stores such as LevelDB where new updates invalidate and overwrite all previous updates, the latest version is always found on the lowest level. In Cassandra and other extensible key-value stores, all versions have to be read because different columns of a record may be added or deleted individually. To find a record, all levels have to be traversed in order from the Memtable to Level-N. In each level, at most one SST has to be read; finding the appropriate SST to read is quick since the key ranges of all SSTs are stored in memory. To improve read speed further, bloom filters can be used to quickly determine probabilistically if an SST contains a record or not [20]. In order to reduce the number of levels visited during a read operation, levels are compacted and merged regularly. For example, frequently read SSTs are merged to higher levels to reduce the number of versions of a record and to improve the data layout.

2.2 SSD Characteristics

Because most SSD devices today are built from NAND flash memory, the specific characteristics of this type of chip have to be considered in order to get the best performance and durability [8]. In the following discussion, we use Flash and NAND interchangeably and will only consider NAND-based SSDs. NAND memory has an asymmetric read and write performance, which is due to the more complex implementation of write operations. In contrast to magnetic disk, NAND memory cells cannot easily be overwritten, but have to go through a slow erase cycle before accepting new writes. While read and write operations are performed on 4 KB - 8 KB pages, erase operations are done in groups of up to 256 pages (erase blocks). To make things more complicated, cheap high density NAND chips, i.e., multilevel cell chips (MLC), have a low number of write-erase cycles,
which is in the order of 2000 to 3000 cycles per block and these get slower as they get older because of higher retry rates on reads. At the same time, flash cells lose data over time, which means that data has to be rewritten in order not to get lost.

Due to these issues, an SSD’s internal controller uses advanced algorithms and data structures to implement wear-leveling, a technique that evenly spreads the write load across all chips to limit the number of erase cycles in total [14]. Wear-leveling is critical to extending the lifetime of the SSD device. These algorithms are hidden from the operating system in the Flash Translation Layer (FTL), embedded in the device’s internal controller. Although modern systems continually improve the performance of the FTL (e.g., [33]), the difference in overhead of write and read operations is still significant. For small random write operations, this effect is more pronounced, since even small changes can result in series of write and erase operations, known as write amplification [8].

For the reasons we discussed earlier, it is of utmost importance that storage systems be designed to make conscious use of the SSD in order to ease the pressure on the FTL controller, extend the lifetime of the SSD and extract the highest performance from the device. This entails minimizing random writes and heavily leveraging large, block-aligned sequential writes combined with page-size random reads. The LSM-tree described in the previous section matches these requirements since it manages data in relatively large data blocks and only performs sequential writes. This eliminates the effect of write amplification at the FTL level (by always overwriting full erase blocks) and ensures flash cells eventually see an equal write workload. The LSM-tree was designed for write-intensive workloads on HDD, but naturally also fits SSD-based systems.

2.3 SSD Price Performance Ratio

In the previous section we showed that not all data structures work well on SSDs, and, hence, an SSD-only storage system will yield suboptimal results. In this section, we show
sub-optimality from a price-performance perspective.

Although more and more installations are built exclusively with SSD storage, the overall price performance of these installations is typically still worse than HDD-based solutions. This is due to the higher price, lower life time, and lower capacity. Below, we discuss the price performance of SSD vs. HDD. Our estimations are based on current market prices for SSDs and HDDs. While prices of SSDs currently are in the order of $0.50 per GB, HDD prices are in the order of $0.03 per GB.

In our analysis, we assume a data intensive scenario using large data sets. For small data sets, SSDs are always economical, because they are still sold with much smaller capacities. For example, a 240 GB SSD is about as expensive as a 3 TB HDD and if 240 GB of storage is enough, the throughput is greatly improved by switching to SSDs (we present extensive performance numbers in Chapter 6.2). If the data size is on the order of HDD capacities, using SSDs increases the storage price. This increase varies, based on the amount of SSD required. As an example, consider a server for USD $1000, adding an extra SSD for USD $120 increases the price by 12%. Consequently, for the upgrade to be economical, the performance improvement should be at least 12%, otherwise the same improvement can be gained by scale out. For larger amounts of data per node, the price increase is more dramatic, for example, replacing 3 TB of HDD by SDD in a USD $3000 server, increases the price by 50% and thus should yield a performance increase of 50%.

---

1. All prices were taken from amazon.com as of June 2015
2. E.g., Intel SSD 530 Series 240 GB
3. E.g., Seagate Barracuda 3TB
Chapter 3

Related Work

One of the most well-known write-intensive structure is a log-structured merge-tree (LSM-Tree) introduced in [26]. Although, this structure supports fast insertions through in-memory batching and bulk loading, it suffers from unexpected performance spikes due to merging the content of smaller trees into larger trees, thus, forcing the data to travel across all levels. The strength of the LSM-tree is its ability to convert random writes into large batched sequential writes. However, it achieves this property at the expense of read performance. The LSM-tree has a poor read performance because every read request must be sent to all levels incurring many random I/O operations. We also note that the LSM-tree was initially designed to operate in a homogenous storage environment composed of HDDs only, rather than a hybrid environment composed of different storage devices with different performance characteristics.

In order to alleviate the LSM-tree’s read problem, the fractal cascading structure is proposed in [5]. However, the fractal tree only improves the asymptotic bounds and assumes that index pages of the B-Tree are disk-resident, while in practice, B-Tree index pages are assumed to be cached. But more importantly, the fractal cascading structure is still subject to sporadic spikes due to frequent merging. The techniques we present in LogStore seek to minimize the need for compaction altogether by taking advantage of the
Chapter 3. Related Work

performance that SSDs provide. Recently, a variation of the LSM-tree has been proposed, known as bLSM [32], which schedules the merging process in regular intervals to avoid spikes. However, bLSM cannot cope with skewed writes (i.e., non-uniform insertions) and does not address the unnecessary read and write amplification of merging a smaller and a larger structure. Neither the fractal tree or the bLSM tree are designed for hybrid storage, where data can be stored on both SSD and HDD; both assume a homogenous storage environment.

Recently, there has been a renewed interest in improving log-structured storage using modern hardware such as SSDs. In [35], the authors proposed to build LevelDB entirely on flash and to exploit the internals of flash using open-channel SSDs [27]. Both [35] and [27] work by exposing the multiple parallel channels of the SSD to the kernel as individual block devices, therefore allowing higher-level storage applications concurrent access to the SSD. [35] modifies LevelDB to support multi-threaded I/O that directly exploits the available in-device parallelism on SSDs. Specifically, the authors optimize scheduling and dispatching of concurrent I/O to ensure maximal utilization of the available parallel I/O channels to the SSD device. Similarly, in [22], the authors demonstrate how to leverage SSD internals, such as the strong consistency and atomicity offered by the Flash Transition Layer (FTL), to implement transactional support in key-value stores. Approaches such as [35, 22, 27] that attempt to utilize the SSDs internal are entirely orthogonal and complementary to the contributions in this thesis. Moreover, each of the techniques can be incorporated in CaSSanDra and LogStore to maximize the usage of the SSD device. Finally, both CaSSanDra and LogStore were designed for hybrid storage environments with both SSDs and HDDs. [35], [27] and [22] all focus on the design of storage systems that are optimized only for SSD and not for hybrid environments.

Lastly, there is a recent trend in the database community to also exploit key SSD characteristics, such as fast random reads that is orders of magnitude faster than magnetic physical drives (e.g., [9, 6, 10, 7]). One way to exploit SSDs is to introduce a storage
hierarchy in which SSDs are placed as a cache between main memory and disks, thereby extending the database bufferpool to span over both main memory and SSDs. A novel temperature-based bufferpool replacement policy was introduced in [10], which substantially improved both transactional and analytical query processing on IBM DB2. In our work on CaSSanDra, we go beyond a simple extension of the bufferpool to the SSD and developing specialized bufferpool enhancements that target the slow read path problem (incurring many random I/Os in order to consolidate row fragments across many SSTables) of extensible row stores. This problem is present in all extensible row stores, but our work focuses on Cassandra. Additionally, the bufferpool inside traditional databases is tightly integrated into other parts of the system such as the lock manager, log manager and the recovery subsystem to ensure the ACID properties are maintained. Our cache in CaSSanDra is lightweight and requires very minimal work to integrate into the key-value store. Lastly, in [15] similar to our cache in CaSSanDra, the use of SSDs as cache was also explored in a proof-of-concept key-value store prototype. In contrast, we introduce the storage hierarchy and our SSD caching techniques within a commercialized key-value store. Furthermore, while [15] focused on using the SSD device as a cache, CaSSanDra identified new avenues for exploiting the use of SSDs within key-value stores, namely, our dynamic schema catalog technique that stores internal system metadata to reduce overall space usage. Most importantly, in LogStore, our goal is to fundamentally redesign write-intensive data structures for hybrid storage environments by introducing SSD-enabled staging of log-structured data stores. This is in stark contrast to existing techniques that adapt and tune the existing data management systems to take advantage of SSD read performance (e.g., [9, 6, 10, 7]).
Chapter 4

Optimizing Cassandra For Hybrid Storage

Apache Cassandra is a popular key-value storage solution that has proven to be highly performant under different workloads [29]. Initially developed at Facebook, Cassandra was designed to handle large volumes of data spread across many commodity servers. In this chapter, we focus specifically on the architecture of Cassandra within a single node and explore how it can be optimized in a hybrid storage environment.

Cassandra can be classified as an extensible row-store since it can store a variable number of attributes per row [11]. Each row is accessible through a globally unique key. Although columns can differ per row, columns are grouped into more static column families. These are treated like tables in a relational database. Each column family will be stored in separate files. In order to allow flexibility of a different schema per row, Cassandra stores metadata with each value. More specifically, this feature is enabled by serializing and colocating a row’s schema information and metadata alongside the row’s data. Metadata information also includes a timestamp per column to enable versioning of column data.
Chapter 4. Optimizing Cassandra For Hybrid Storage

4.1 Read and Write Path

An overview of the read and write path in Cassandra is given in Figure 4.1.

Cassandra’s storage model is based on BigTable, which in turn is based on the LSM-tree. As such, the write path is similar to the LSM-tree as described in Chapter 2.1: writes are buffered in an in-memory Memtable and appended to a durable commit log. If and when the Memtable exceeds its threshold size, it is converted to an SST and flushed to disk. Updates (including blind writes) are handled in the same manner as new record insertions, while deletions are handled by inserting a record tombstone. In Cassandra, rows can have arbitrary schemas and new attributes can be added at any time, which means that parts of a single row can be stored across multiple SSTs. A read request for a given row may, therefore, incur multiple random I/Os to read each fragment of the row stored in different SSTs. For this reason, Cassandra provides an optional row cache and key cache (not shown in the figure). The row cache stores the consolidated, most recent version of a row, while the key cache acts as an index to SSTs on disk. If row or key caches are enabled, write operations need to keep them updated. When a write enters Cassandra and the row cache is enabled, the write goes into the Memtable, is appended to the commit log, and is merged with the latest version of the row in the row cache, if a version exists in the cache. This means that the hottest rows will be duplicately stored in main memory.

The read path starts by looking into the caches, first the row cache and then the key cache. Rows that reside in the row cache can be completely served from memory since they are always complete and up-to-date. Rows are added to the row cache whenever there is a cache miss; rows are replaced in LRU fashion. For workloads with small sets of hot data that are frequently accessed, the row cache can improve the system’s performance considerably. Rows that do not reside in the row cache may be distributed across multiple SSTs and Memtables, thus creating multiple disk accesses for a single read. In fact, all versions of a row have to be considered because of the flexible schema.
Fighting back: Using observability tools to improve the DBMS (not just diagnose it)

This is because an older version might have a requested column that does not exist in later versions. To reduce the number of SSTs that have to be considered to retrieve the full contents of a row, a periodic compaction process is run to merge multiple SSTs. In this process multiple versions of a row are consolidated and rows that are marked as deleted are completely removed.

4.2 Extending the Row Cache on SSDs

Cassandra’s row cache stores fully compacted rows in memory. This improves performance dramatically if the workload features a hot data set that fits into memory. If the hot set does not fit in memory the benefit of the row cache is quickly consumed by the cost of much slower disk accesses. Storing only consolidated rows on disk would significantly impact write performance. This is because a random I/O would be required per key during a Memtable flush to read the latest version from disk, perform the merge operation in memory, and flush out the newest merged version back out. The benefit of
batching updates in a Memtable is that the cost of the eventual I/O is amortized across multiple updates. Additionally, the I/O required to flush a Memtable is only sequential, requiring no random accesses to disk, and is very fast. In order to introduce SSDs into Cassandra’s storage hierarchy, we extend the row cache beyond memory onto the SSD.

The row cache in Cassandra has been modified to chain together the in-memory cache and the SSD cache as shown in Figure 4.1. Read requests against the row cache flow along the chain, first querying the in-memory cache then the SSD cache. A read request stops along the chain at the first point its request is successful since each cache always stores the most recent version of the compacted row. The in-memory row cache is implemented as a size-bounded LRU cache, where the oldest and largest entries are ejected when a capacity threshold is reached. With the modified row cache chain, entries that are evicted from the in-memory cache cascade into the SSD cache asynchronously, thereby incurring zero additional latency during cache insertions or updates.

The extended row cache is modelled as an append-only cache file with an associated hash-based index that is memory resident. The index maps row keys to 64 bit offsets into a logical cache file. A query into the SSD cache is executed by first consulting the index to determine the position of the compacted row in the cache file on SSD, followed by a single SSD seek to retrieve the row. An important observation here is that concurrent reads never block one another except inevitably in I/O access to the SSD itself. Append-only operation is achieved by enforcing a sequential order to write requests. We eliminate the potential bottleneck of using a single-writer policy by buffering writes into a 16KB in-memory buffer that is flushed to SSD when full. Having a write buffer also affords us the opportunity to optimize certain read queries by directly accessing the buffer rather than reaching into the SSD. An insertion into the SSD cache begins by serializing the row into the write-buffer and updating the hash-index of the row’s key with the updated offset. Deletes from the row cache are handled by simply removing the row’s key from the hash-index. A background garbage-collection task is executed periodically to remove
deleted entries from the SSD cache files. This design takes advantage of the peculiar behavior of SSDs discussed above. In this way, because writes are buffered in memory and flushed at page boundaries sequentially, we suffer no write amplification or wear levelling.

The logical cache file described above is actually composed of a collection of contiguous segment files, each of a configurable size and preallocated on creation. The most recently created segment is termed active and is the only segment that accepts writes, whereas any segment can service reads. Segment files are created as part of a write if the write operation overflows the currently active segment. A 64-bit offset in this model is analogous to a virtual address; given a 64-bit offset, we use the the higher-order 32 bits to determine which segment file the row’s data resides and the lower-order 32 bits as the offset into the segment. We perform this segmentation for practical purposes, but more importantly because it makes the task of garbage-collecting deleted rows a relatively simple one. We track the amount of garbage in each segment file, where garbage is defined as the percentage of the segment file that contains deleted rows. The garbage-collection task uses this metric to select segments to compact that will yield the largest reclamation of space.

### 4.3 Dynamic Schema Catalog

Every row in the data component of Cassandra’s SST is made up of a row key, an optional column index, and a series of column name, column value and timestamp tuples. This provides a schema-less data model that is flexible and easy to work with. For many big data use cases, however, schemas are only slowly evolving, meaning that column names are frequently duplicated unnecessarily. This duplication not only increases latency at read time, since more data needs to be read from disk, but also wastes storage space. This overhead can be reduced by extracting the schema from the rows and storing them
Figure 4.2: Redundant vs. dynamic schema

separately in a schema catalog. However, removing schema information altogether and reverting to the relational model where a single schema is defined per table is not an option. We want to retain the flexibility of a schema-less design, but reduce the unnecessary duplication and storage of schema information. If we persist the dynamic schema catalog on disk, however, it will introduce additional random I/Os and, thus, deteriorate performance. Storing the catalog in memory is not an option because of the potential size of the catalog and the issue of data loss in cases of failures. Since SSDs provide fast random access and stable storage, it is the ideal medium to store frequently used and updated metadata. We can therefore maintain the flexibility of a schema-less data model with the benefits of a schema-ed one, along with reduced disk footprint.

The schema catalog is a global data structure within a single running Cassandra instance. Conceptually, the catalog maintains a mapping between schema identifiers (IDs) and schema definitions where IDs are monotonically increasing 32-bit unsigned
integer values and a schema definition is simply a sorted set of column names each a string value. This is illustrated in Figure 4.2. The implementation of the schema catalog resembles what was described for our SSD row cache in the previous section. That is, the physical layout of the schema file consist of a collection of segment files, each storing a contiguous subset of schema ID and schema tuples, along with an associated memory-resident hash-index.

The dynamic schema catalog exposes operations for insertion and retrieval of schema definitions. When Cassandra is preparing to serialize a row to disk, it first consults the schema catalog to obtain a schema ID for the row by supplying the sorted set of columns within the row. If the catalog determines that the schema definition of the row has already been assigned an ID, this ID is returned immediately. Otherwise, a new schema ID is generated, the schema ID and definition are persisted to SSD and the ID is returned to the caller. The order of these operations is critical: the schema ID and its definition must be persisted safely before the ID generation call can return to ensure the data isn’t corrupt. Cassandra will serialize the schema ID locally with the row and will exclude serialization of the column index and any of the column names. During a read operation, the schema ID will be deserialized first at which point the schema catalog will be queried to efficiently retrieve the schema definition from SSD. With the schema definition available prior to deserialization of the columns, Cassandra knows precisely how the column values are laid out on disk and can choose to retrieve either all or a subset of the columns depending on the type of read operation.

If a table’s rows are expected to be largely homogeneous, meaning that rows differ very little in the columns they use, then the proposed schema extraction technique can yield substantial savings in space and reduce the latency of read operations. If, however, the table’s data is predicted to be highly variable in the columns that are inserted, extracting the schema from the table can be detrimental. For example, a table with 10 columns can have up to $2^{10} - 1$ possible unique schemas. It is obvious that multiple high-variability
tables all using a schema catalog can quickly exhaust available schema IDs. Note that the current implementation makes no attempt to reclaim schema IDs if their associated schemas are no longer in use, though this is a work in progress.

Having the schema available before reading a row’s columns allows Cassandra to perform sliced-name read queries particularly well. A sliced-name query is one where only a subset of all the row’s columns are requested. Historically, Cassandra executes sliced-name queries with the assistance of a column index that is serialized along with the row just before the column names, values and timestamps appear. This index is only a sample of the columns, meaning it provides only an approximation of a column’s position in the row but provides no guarantee that the column is actually present. This can lead to potentially very inefficient and unnecessary reads that add latency to the request. Having a dynamic schema catalog available allows Cassandra to efficiently and quickly retrieve the schema from SSD to determine both the existence of a column in the row and its precise position through simple calculation, making it a favourable option for slice-name queries.

### 4.4 Evaluation

For our evaluation, we used the Yahoo! Cloud Serving Benchmark (YCSB) suite. YCSB is composed of a data generation component and a workload generation component. For data generation, YCSB functions on records which are a collection of columns
where each column has a name and value of fixed size. Each record is indexed by a 25-byte key and the user can configure the number of columns in the record along with the size of each column. The data loading phase will insert a configurable number of records into the data store. In the transactional phase, YCSB provides parameters to control concurrency, maximum execution time, statistical distributions of accessed keys and distributions for operations. YCSB offers standard CRUD (create, read, update delete) operations that makes it amenable to key-value stores.

In this section, we will compare the performance of a Cassandra installation that uses HDD versus an installation with SSD. For these experiments, we used a system that was equipped with an Intel Xeon X5450 eight-core CPU running at 3GHz, 16GB of RAM, four 1TB hard drives configured to create two RAID0 2TB hard drives and two 240GB Intel X520 SSDs. Since Cassandra uses two different basic disk-stored entities (SSTs and the commit log), there are 4 different configurations with HDDs and SSDs. We tested all configurations in a read-mostly environment (95% read, 5% write). The dataset consisted of 100 million rows, totalling 50GB data. We used a distribution where the last inserted row is accessed most frequently and all keys are accessed according to a zipfian distribution; we shall henceforth refer to this type of request distribution as a latest distribution.

In Figure 4.3, the throughput of all configurations outlined in Table 4.1 can be seen. As expected, putting the data on SSD has a dramatic performance benefit. The same trend can be seen for the latency, as show in Figure 4.4. It is important to note that configurations C1 through C4 use a YCSB client setup that is suboptimal for SSDs. These configurations were chosen because they saturate I/O access to our HDDs. Configurations C5 and C6 are optimized to operate against the SSD by using more clients each running a higher number of threads to achieve a much higher throughput and a latency that is average for the class of SSDs we used.

Figures 4.3 and 4.4 show that placing everything on SSD does not make sense for
Figure 4.3: HDD vs SDD throughput results

Figure 4.4: HDD vs SDD latency results
Figure 4.5: HDD vs SDD throughput for 99%-filled disks

Figure 4.6: HDD vs SDD latency for 99%-filled disks
Cassandra. Storing the commit log to SSD (C2) offers a minimal boost to throughput and latency versus writing the data to SSD (C3). Since not all data is accessed with the same frequency, it is more efficient to selectively store frequently accessed data on SSDs and rely on HDD for the bulk of data that is infrequently accessed. Another reason to do this is the fact that SSD performance degrades with higher fill ratios. As seen in Figure 4.5, the performance of a highly filled SSD degrades much worse than the performance of a highly filled disk. It has to be noted that the workload in this case is still read heavy, for write heavy workloads even worse degradations will be experienced.

When evaluating our extended SSD row cache, the size of the data set was 100 million records, where each record had five columns having a size of 75 bytes. The total size of the data on disk after load averaged 50GB. Our evaluation process was broken down into four phases: data loading, data fragmentation, Memtable flush, bufferpool warmup, and transactional workload phases. The fragmentation phase attempts to spread the columns of a row across multiple SSTs to illustrate the effect of read amplification on LSM-based storage systems. In the fragmentation phase, we used a latest request distribution with 10% of operations being read and the remaining 90% of operations updating anywhere between one and all five columns. The warming phase also used a latest request distribution with read operations accounting for 99% of all operations. The warmup phase was run until either the cache was full or stored at most 10% of the total dataset. The transactional phase was run with a latest distribution (a zipfian distribution where the most recently entered keys are favoured). These experiments all used configuration C5 (refer to Table 4.1), the optimal configuration for HDDs to provide a balanced evaluation.

When evaluating our dynamic schema model, we used a dataset consisting of 40 million records where each record consisted of between 5 and 10 columns, of 10 bytes. By default, YCSB does not vary the number of columns in a record during the loading phase. We modified YCSB to create a new varying-size record generator, which we plugged into the default data generator. Each run of the experiment created a different
4.4.1 SSD Row Cache Performance

In Figure 4.7, the throughput of the two Cassandra instances can be seen for the three different workloads that were tested. For the 95% read-heavy workload, we see that the SSD-enabled row-cache provides an 85% improvement in throughput growing from 384 reads/sec to 710 reads/sec. This is because a larger portion of the hot data is cached on the SSD; in fact, our configuration enabled storing more than twice the amount of data than when using an in-memory cache alone, achieving a cache-hit ratio of more than 85%. When a read operation reaches the server for a row that does not reside in the off-heap memory cache, only a single SSD seek is required to fulfill the request. In addition, cached data is pre-compacted, meaning that at most one seek is required to fetch the row. We see the same effect in the remaining two workloads despite a lower proportion of reads. Cassandra is a write-optimized system meaning that in write-heavy scenarios,
the efficacy of a cache is reduced. This is evidenced by the reduction in the cache-hit ratio from 72% in the workload with 85% reads to 60% in the 75%-read workload.

As seen in Figure 4.8, in the 95% read workload, the SSD-enabled row cache averaged a latency of 3ms while the in-memory cache managed a read latency of 5.6ms, a 46% improvement. As the proportion of reads is reduced from 85% to 75%, the latency when using an SSD for the row-cache remains roughly the same. This is because the latest request distribution gives us a high probability that read operations for can be served directly from Cassandra’s Memtable, which effectively acts as a write-back cache.

### 4.4.2 Dynamic Schema Catalog Performance

Next, we illustrate that by extracting the metadata (i.e., schema) from the data on-disk we suffer no perceivable performance penalty. The column names in our test were fixed at 5 bytes and the number of columns varied between 5 and 10. This accounts to a minimum saving of 25 bytes from being written on a per-row basis. Cassandra, not uncommon from many commercial databases, performs buffered I/O; all reads and writes are executed in 16 KB pages. In our experiment configuration, one row fits well
within a single Cassandra page. This means that reading a row will incur no additional overhead since the total size of a row with a colocated schema is larger than a modified row with the schema extracted out. When we extract out the metadata, we expected no degradation in performance or latency and the results in Figure 4.9 and Figure 4.10 confirm our assertion. Specifically, we conclude that in the 95% and 50%-read workloads, the latency and throughput were comparable with any difference being attributed to the environment.

Throughput and latency are not major motivations for implementing the dynamic schema. Fairly significant space savings can be obtained by extracting redundant schema information and we find this to be much more compelling. In normal operation, data sizes averaged 6.8GB compressed after the initial load of 40 million keys. With a modified Cassandra, data sizes averaged at 6.01GB of data, a savings of roughly 10%. This value will grow as the number of columns in the table grow and as column names grow in length.

Another potential benefit for dynamic schema model (omitted in the interest of space), is in executing column-slice queries. When performing a read from Cassandra, it is
possible to read a slice of the row by specifying which columns to read. Though Cassandra has an index per-row, it is only a sample; not every column has an appropriate index entry. If we have a schema on hand, we know precisely the layout of the row on disk which we can use to optimize the read process and avoid cache pollution.

Finally, it is important to note that we are not using high-end enterprise PCIe-bus SSDs (e.g., FusionIO), yet we are getting a substantial performance improvement. Therefore, we conclude that even with inexpensive commodity SSDs, a considerable throughput and latency improvement is achieved.
Chapter 5

Log-Structured Hybrid Storage Cost Model

In this chapter, we construct a rigorous analytical cost model to predict the performance of a generic log-structured hybrid storage system. In this setting, there is both an SSD and an HDD, though both can be replaced with any storage device. Each device is part of the hierarchy and none is used as a cache. Only a single copy of data exists in one of the devices that constitute the hierarchy.

We begin this chapter first by providing an example performance analysis of a hybrid storage system. We do this to provide intuition for how to analyze such a system under two differing workloads. We use this intuition in the sections that follow when we formalize the analytical model and conclude with a brief discussion of the model.

5.1 Hybrid Storage Performance Analysis

Any hybrid storage system must be able to exploit the individual characteristics of each storage device. The most important dimension in this context is performance, since SSDs are many times faster in read and write throughput than HDD. To get an improved performance, most of the I/O workload should be sent to the SSD. For example, in
our experiments, the SSD is up to an order of 40 times faster than disk for random access. Thus, random reads should always go to SSD. Writes are faster on SSD, but small random writes result in a high write amplification on SSD and thus decrease the SSD performance.

We examine two kinds of access distributions, uniform and Zipfian. In the uniform case, all records in the data set are accessed with the same probability. With a Zipfian distribution, the probability of an element is inversely proportional to its rank, it is typically chosen to represent skewed data access [18, 13]. In Figure 5.1, both distributions can be seen for 10000 elements, ordered by access frequency (the Zipf parameter in all examples is 1). An optimal data placement for read accesses can be seen in Figure 5.2. In this example, 50% of the data are stored on HDD and 50% are stored on SSD. 1% of the data is also cached in memory. In the uniform case, all data has the same access rate, thus approximately 50% of the accesses go to SSD and 50% to HDD, 1% of the accesses go to...
data cached in RAM. Under the assumption that there is no additional data movement or management overhead, this setup results in a throughput, which is approximately twice as high as a system, which only uses HDD. The cache in RAM has very limited effect on the performance, since only 1% of the accesses are served from RAM. In case of a Zipfian access distribution, a good data placement serves the 1% most frequently accessed data from RAM and places the 50% most accessed data on SSD. This is depicted in Figure 5.2. For the example, the 1% most accessed records get approximately accessed in 53% of the cases, while the top 50% most accessed data gets 93% of the accesses. As a result, the SSD sees roughly 40% of the total data accesses and 7% go to the HDD. Given that HDD still limits the throughput, we expect a performance improvement by a factor of approximately 6.5 over an HDD-only storage layout.

5.2 Hybrid Storage Cost Model

Based on the above example, we can develop a general model for hybrid storage performance. We can use this model to predict the performance of a generic log-structured hybrid storage system given a workload, device, and system characteristics. In addition to its predictive capabilities, the model also provides insight on how RAM and SSD sizes, the number of levels, and the degree of skew in the workload interact with one another to affect the overall performance that can be achieved in a log-structured hybrid system.

Since performance varies by type of access, we will discuss read-only, write-only, and mixed workloads separately. In a hybrid setting, some levels of the data store will be stored on the SSD and some will be stored on the HDD. In general, the data store will have a total of $L = L_{SSD} + L_{HDD}$ levels, where we indicate the number of levels stored on the SSD and HDD by $L_{SSD}$ and $L_{HDD}$, respectively. We also make two assumptions that serve to simplify the analysis. The first assumption is that only a single client with a single thread is accessing the system and issues read and write operations sequentially.
The second assumption is that both read and write operations select keys from the same Zipfian distribution with the same skew.

### 5.2.1 Read-Only Model

In the following, we indicate uniform and Zipfian distributions with a superscript where a distinction is necessary (e.g., $R_{HDD}^u$ for the access rate to HDD in a uniform distribution). In formulas that are applicable for both distributions, we omit the superscript. If we assume a uniform distribution of read accesses, all data has the same probability of getting accessed. Thus, SSD and HDD are accessed according to the amount of data they store. Let $|SDD|$ be the relative amount of data on SSD, let $|HDD|$ be the relative amount of data on disk, and let $|RAM|$ be the amount of data cached in RAM. For the read-only case, we only have to discuss one operation, which is a single record access. This is not necessarily a single operation on either SSD or HDD, since finding a record requires identifying the correct SST, looking up the position in the file’s internal index, and reading the record. In the cost model, this is abstracted as a high-level read operation. Since RAM access is much faster than SSD or HDD, we consider access to RAM as free and, thus, set the cost of the according amount of read accesses ($R_{RAM}^u$) as 0 (and the throughput as $\infty$). Since SSDs and HDDs vary considerably in performance, the performance model is relative to the performance of SSD ($TP_{SSD}$) and HDD ($TP_{HDD}$) high-level read operations. In the uniform case, the amount of accesses HDD directly correlates to the amount of data stored on HDD reduced by the amount of that data cached in RAM:

$$R_{HDD}^u = (1 - |RAM|) \times |HDD|$$

$$= (1 - |RAM|) \times (1 - |SSD|) \quad (5.1)$$
The SSD access rate \( R_{SSD}^u \) is defined analogously. The total throughput for a read-only, HDD limited setup \( TP'_R \) can be estimated using the following equation:

\[
TP'_R = \frac{TP_{HDD}}{L_{HDD} \times R_{HDD}}
\]  

(5.2)

Since throughput is limited by HDD, SSD speed does not influence the final throughput. The relative amount of workload seen by HDD is the percentage of data stored on disk minus the amount of accesses served from RAM. Since we assume a uniform distribution, all data has the same probability of being cached and thus the amount of cached accesses for disk is relative to the amount of data on disk. The formula for an SSD limited case is analogous. The tipping point can be estimated by the throughput vs. access rate. If the relative access rate is higher than the relative throughput of either SSD or HDD, the respective storage device is the bottleneck:

\[
TP_R = \begin{cases} 
\frac{TP_{HDD}}{L_{HDD}R_{HDD}}, & \text{if } \frac{TP_{HDD}}{L_{HDD}R_{HDD}} < \frac{TP_{SSD}}{L_{SSD}R_{SSD}} \\
\frac{TP_{SSD}}{L_{SSD}R_{SSD}}, & \text{else}
\end{cases}
\]  

(5.3)

If we assume a Zipfian access probability, the access distribution is different and not equal to the relative data sizes on the respective devices. As depicted in Figure 5.2, we assume the more frequent accessed data to be placed on SSD and the top accessed data of that to be cached in RAM. The amount of accesses to RAM \( R_{RAM}^z \) can be estimated using the formula for the Zipfian distribution:

\[
R_{RAM}^z = \frac{\sum_{n=1}^{\text{|RAM|}} \frac{1}{n^s}}{\sum_{n=1}^{N} \frac{1}{n^s}}
\]  

(5.4)

Where \( N \) is the total amount of data (in number of records) and \( s \) is the Zipf parameter or skew factor. We can use the same formula to estimate the amount of accesses to SSD and HDD.
The total read throughput in the Zipfian case ($TP_R$) again is dependent on the throughput and access rate of the limiting device and can be estimated using Equation 5.3. If we assume an HDD limited setup, we can see in Equation 5.6, that the throughput is only depending on the access rate that hits the HDD. Since the RAM caches only data from SSD in the model, the amount of data cached in RAM does not change the throughput.

### 5.2.2 Write-Only Model

Intuitively, the cost of an individual insert into the LSM-tree is equal to the total I/O cost of propagating the record through each of the $L$ levels of the LSM-tree. If we let $M = \frac{\text{size of } L_{i+1}}{\text{size of } L_i}$ be the growth factor between the levels of the tree, and let $S_{SST}$ be the size of an SST, then each compaction reads $(M + 1) \times S_{SST}$ bytes of data and subsequently writes an equivalent amount of data back out. We need to amortize this cost across the number of records in an individual SST, $S_{SST}/r$, where $r$ is the size of an individual record. Therefore, the total I/O required per-record during compaction is $2 \times (M + 1) \times r$. This record will undergo, at most, $L - 1$ compactions. Since compactions (and therefore writes) are always performed in a sequential manner, if we let $R$ and $W$ be the sequential read speed and the sequential write speed of the device (in bytes/sec), respectively, then we can estimate the overall write throughput of the LSM-tree to be:

$$TP_W = \frac{\min(W, R)}{2(M + 1)(L - 1)r}$$

(5.7)
Equation 5.7 is flexible enough to model storage devices that have asymmetric sequential read and write speeds. This is because compaction never brings all input data into memory, but works on blocks of data. Therefore, compaction proceeds in a series of read, merge, write operations. If sequential read speed is slower than sequential write speed, then the in-memory merge and subsequent serialization operations are always blocked awaiting new data as input. Similarly, if sequential write speed of the device is slower than sequential read speed, the read step and the in-memory merge step is blocked awaiting for the serialization of the previous block of sorted data to be written out to disk. In essence, the speed at which compaction can proceed is bottlenecked by the slower operation, either sequential read or sequential write. However, in many common storage devices, these speeds are equal and balanced.

In a hybrid log-structured storage system, a fraction of the levels will be stored on SSD and the remaining fraction will be stored on HDD. Writes are initially buffered in the Memtable (in RAM) and eventually flush to the SSD in the form of an SST. Data travels through the levels on the SSD before migrating out to the HDD through compaction as the SSD reaches its capacity. In the stable state, the SSD is always full, meaning that as new data enters the system, data must be migrated out to the HDD. The write-throughput, therefore, not only depends on the sequential read/write speed of each device, but also on the number of levels writes travel through on each device. If we let $R_{HDD}$ and $W_{HDD}$ be the sequential read speed and sequential write speed of HDD, respectively, and if we let $R_{SSD}$ and $W_{SSD}$ be the sequential read speed and sequential write speed of SSD, respectively, then we can use Equation 5.7 as a starting point to derive an inequality that determines which device will be the bottleneck. The write-only throughput performance of a hybrid setting is then captured with:
Equation 5.8 provides a lower bound on the expected write throughput in a hybrid environment. It has to be noted that the difference in throughput for SSD and HDD on serial writes is much less pronounced than in the random read case (in our experiments, we see a difference of factor 2x-3x). Additionally, while we heretofore did not distinguish between new insertions and updates, they do offer interesting cases to consider. An ordered insert workload does not benefit from a cache and will experience very simple and cheap compactions since there is no overlapping key ranges between levels. In an update workload using Zipfian key distribution, the in-memory Memtable absorbs some of the updates, but compactions will be frequent and costly. Finally, an unordered insert workload and an update workload are essentially indistinguishable for the purposes of our cost model.

5.2.3 Read-Write Model

To simplify the analysis in the mixed read-write case, we assume that operations are performed sequentially, one after another, by a single client. Then, the total throughput in a mixed workload is dependent on the ratio of reads \((r)\) vs. writes \((w)\) that the storage system sees, along with the read and write throughput the system is capable of. We can use Equation 5.3 from Chapter 5.2.1 to calculate the read throughput, and use Equation 5.8 from Chapter 5.2.2 to calculate the write throughput. Since each of these formulas account for the bottlenecking device, the number of levels and the effects of cache and compaction, we can estimate the total throughput of mixed read-write workload by weighing each contributing component by the proportion of reads and
writes. The resulting formula is provided in Equation 5.9.

\[
TP_{RW} = \frac{1}{\frac{r}{TP_r} + \frac{w}{TP_w}}
\]  
(5.9)

It should be noted that if a workload includes a considerable amount of updates, the cache becomes less effective. The reasoning is that new and updated data is frequently compacted and during this procedure pollute the cache. As a result, we see a significantly reduced cache hit ratio for read operations in the Zipfian case. The loss of cache efficiency is especially bad when an HDD is present (as in a hybrid case) since a larger percentage of read operations will require a costly seek latency that is often the dominant factor in performance. For this reason, any hybrid storage architecture should strive to enable the SSD to absorb the majority of read requests, while maintaining a small size overall.

### 5.2.4 Discussion of the model

One subtle effect that is not captured by the model is the increasing access latency for HDD if less data is stored in a skewed workload. Since more frequently accessed data is stored on SSD, accesses on HDD have less locality and thus require more seek time on average. In our experiments, we see up to 20% loss in throughput on HDD if 50% of the data is stored on SSD. Additionally, our model allocates all available RAM as a cache that only caches the hottest data in the workload. Our experiments show that not all RAM is available and not only hot data is in the cache - some cold data from the HDD will be cache-resident.

The model we present here is applicable to both hybrid (heterogenous) and homogenous storage environments. We evaluate the model’s accuracy in Chapter 6.2 by comparing its performance predictions with results we acquire experimentally of both a hybrid log-structured storage system and a conventional LSM-tree storage system. The model’s generality helps us uncover a significant performance bottleneck hidden within LevelDB
when operating on SSD. We diagnose this specific problem, optimize and solve the issue and evaluate our optimization in Appendix A.
Chapter 6

LogStore

The analytical model presented in Chapter 5 clearly establishes a connection between the number of levels on each device, the access rates to each device and the throughput that can be expected of a hybrid storage system. Specifically, in a hybrid system that is bottlenecked by the HDD (as will certainly be the case when including RAM and storage-class memory devices like SSD), both read and write throughput can be improved by storing at most one level on HDD. In this way, read operations require at most one seek on HDD, and updates require at most one compaction to HDD. Read throughput can be further improved by minimizing the access rate to the level stored on the HDD. In a uniform key request distribution, the ratio of access to HDD is directly proportional to the size of the SSD. In a skewed request distribution, the size of the SSD plays less of a role as the degree of skew itself. Ideally, we can leverage the skew in the distribution to select an SSD size such that both the SSD and the HDD are fully utilized. We directly use these insights to inform the design of LogStore.

6.1 LogStore Architecture

The architecture of LogStore, shown in Figure 6.1, resembles other log-structured data management systems as described earlier in Chapter 2. Writes in LogStore are buffered
LogStore Architecture

- Resembles LSM-tree + BigTable
- SSTable storage + Memtable in-memory format
- Write and read path essentially the same
- Organize SSTables into three levels
- Youngest levels on SSD, oldest on HDD
- SSD stores half of total data (across two levels)
- Maintain an in-memory histogram of accesses per-level
- Buckets for histogram = SSTs in level
- Read path now modifies histograms on successful read request

Figure 6.1: LogStore architecture
in a Memtable and written out to a commit log. When the Memtable has reached a configurable size, it is converted into a read-only Immutable Memtable. When this occurs, a new Memtable is created to handle new writes while the Immutable Memtable is simultaneously flushed to the first level as an SST. LogStore structures all SSTs into a series of three levels, the first two of which are on SSD while the last is on an HDD. Additionally, LogStore stores metadata about all SSTs — including the level they belong to, their size, creation times and the minimum and maximum keys they contain — in memory.

SSTs within a level are disjoint in the keys they store while SSTs across levels may overlap in key ranges, and often do in skewed workloads. LogStore does not size the levels such that they grow exponentially, but rather arranges the levels so that the total amount of data stored on the SSD (combined between Level-0 and Level-1) is a configurable fraction of the total amount of data. In most of our experiments, the SSD stores 50% of the total data.

LogStore maintains one histogram per-level in memory that tracks accesses to keys stored in the associated level. If an SST on level $L$ successfully services a read operation for key $k$, LogStore requests level $L$’s histogram to increment the count of the bucket into which $k$ falls. LogStore’s histograms keep buckets sorted by smallest key in the bucket’s interval, which makes finding the bucket for a given key a logarithmic operation in the number of buckets. Additionally, the buckets use atomic counters and so can be safely and efficiently mutated in a multithreaded setting. The histogram for a level is static in the sense that it is neither equi-depth or equi-width, nor does it split or merge buckets at run-time. There is a one-to-one mapping between SSTs in a level and the buckets in the histogram; the range of keys covered by a bucket match the range of keys stored in the matching SST within the level. When an SST is added or removed from a level (through compaction), LogStore first clones the original histogram, adds or removes the appropriate buckets, and adjusts the counts to reflect the change. It is important that
the access history for key ranges is not lost between such modifications so that LogStore remains sensitive to changing workload characteristics. If a new bucket is created to account for the addition of a new SST, its count is calculated by finding the count for the range of keys the SST covers using the source level’s histogram. Constructing static histograms in this way can result in the existence of gaps in key-ranges between buckets, but this is not a problem since it is guaranteed that a level’s histogram will only receive increment requests for keys that fall into a valid bucket.

The LogStore architecture has three main goals:

1. Store the hottest data on the SSD while evicting the coldest data to the HDD.

2. Perform as much of the I/O-intensive, preparatory work on the SSD as possible.

3. Ensure at most one seek for reads on HDD.

Each of the optimizations that follow in this section strive to achieve one of the above listed goals.

### 6.1.1 Informed Compaction

Like all log-structured database systems, LogStore performs compaction to merge multiple versions of key-value pairs and remove deleted keys from the store. However, the choice of
which SST (i.e., key-range) to compact from level $L$ to level $L+1$ is especially important in a hybrid storage environment. It would be undesirable to merge the contents of a frequently accessed SST from a very fast SSD to a much slower HDD since this would result in significantly reduced performance. Traditionally, LSM-tree implementations employ a round-robin merging strategy where the choice of which SST to compact on a given level is made by rotating through the key space of that level. In the beginning, the SST storing the smallest range of keys is chosen and each subsequent compaction selects SSTs storing increasing key values. When the SST storing the largest keys in the level has completed compaction, the process wraps around to start from the smallest keys yet again. Though simple and intuitive, this strategy does not work in a hybrid storage environment. Since every SST has an equal probability of being selected for compaction, a round-robin strategy will unwittingly merge the contents a frequently accessed SST from a fast SSD to a much slower HDD and considerably hurt performance. We believe a temperature-based SST selection strategy must be used in a hybrid setting in recognition of the dramatic difference in performance between SSDs and HDDs.

LogStore strives to retain only the most frequently accessed SSTs on the SSD, where access latencies are lowest, and ensures that the coldest SSTs are evicted to the HDD. LogStore achieves this goal by implementing a workload-aware, temperature-based SST selection strategy that intelligently chooses which SSTs should be compacted, and leverages the compaction process as a data-movement mechanism. Figure 6.2 illustrates the general process of compaction in LogStore. When a compaction is triggered on Level-1, the thread performing the compaction first consults the level’s histogram to determine the coldest key-range, then finds all SSTs that store keys which fall into the chosen range. By definition, these SSTs are accessed the least frequently on the level and naturally make the best candidates for eviction from the SSD. As before, LogStore finds all SSTs on the last level that overlap in their key range with the input SSTs and these collectively form the input tables to the compaction process. Once the compaction completes, we also need
to ensure that access counts for the input key-ranges are transferred to the histogram on the last level. Retaining this access history is necessary to handle the case where the range becomes hot at a later point in time, as we will discuss shortly.

Informed compaction ensures that the coldest data is migrated out to the HDD, and therefore, only the hottest data is retained on the SSD. In this way, informed compaction achieves the first goal set out in the design of LogStore.

### 6.1.2 Reverse Compaction

Compactions have the effect of pushing data from younger levels down towards older levels. In LogStore, this means that data moves towards the HDD. While informed compactions from Chapter 6.1.1 evicts the coldest data to the HDD, it may be the case that data on HDD becomes hot, possibly even hotter than some data on the SSD itself. In this scenario, it may be cost-beneficial to migrate data from an old level on HDD to a younger level on the SSD in recognition of increasing access frequency. LogStore achieves this by implementing reverse compactions. Reverse compactions use the exact same compaction process as discussed previously, but with a few alterations. The choice of which SST to compact now becomes the most frequently accessed SST on Level-2 (on the HDD). LogStore then finds all overlapping SSTs on Level-1 and collectively forms the input tables to the compaction process. In a reverse compaction, the source is Level-2 and the target is Level-1.

The decision for when it is most opportune to perform a reverse compaction is based on two conditions. The first condition requires that the hottest SST on Level-2 is accessed more frequently that the least 10% frequently accessed SSTs on Level-1. Though the optimal strategy is to employ a true Least Frequently Used (LFU) cache on the SSD, LogStore chooses to be conservative to prevent costly thrashing between SSD and HDD. The second condition that must be met is determined through an intuitive cost analysis: a reverse compaction is scheduled when the cost of compaction is less than the aggregate
cost of reading the SST on HDD the last \( n \) times. If we let \( S \) be the size of each SST (for generality, assume SSTs have equal size), let \( R_h \) and \( R_s \) be the random read speed of HDD and SSD, respectively, let \( W_h \) and \( W_s \) be the sequential read/write speed of HDD and SSD, respectively, and let \( T \) be the total number of SSTs involved, then the total cost of compaction is as shown in Equation 6.1.

\[
C_{\text{comp}} = (T - 1)R_s + R_h + S\left(\frac{2T - 1}{W_s} + \frac{1}{W_h}\right)
\]  

(6.1)

Equation 6.1 can be decomposed into two parts. The first two terms account for the \( T \) required seeks to the beginning of each input SST, \( T - 1 \) of which take \( R_s \) time since they exist on the SSD. The last term accounts for the total sequential I/O required to read \( T \) SSTs into memory, execute a merge and write \( T \) SSTs onto the SSD where each SST is \( S \) megabytes in size.

Since we keep a histogram of access counts for SSTs, we can calculate the total accrued I/O cost for accessing a key in the SST. If we let the number of times the SST has been read be \( N \), then the minimum total cost of I/O to access the SST is as shown in Equation 6.2. Reading an SST involves, at most, two seeks. The first seek is to read the index for the SST located at the end of the SST file. The second seek is driven by the results of the index probe that tells us the offset into the file that contains a block of key-value pairs that is searched sequentially. It is important to note that the cost equation in Equation 6.2 is the \textit{minimum} cost incurred by requesting a key that belongs to a table on the last level. Since SSTs across levels are not disjoint, it is possible for SSTs in Level 0 and Level 1 to potentially contain the requested key-value pair, paying additional I/O. This additional cost is captured in \texttt{LogStore} by tracking failed SSTs lookups at runtime.

\[
C_{\text{read}} = 2NR_h
\]  

(6.2)

\texttt{LogStore} triggers a reverse compaction when the compaction cost from Equation 6.1
is less than the total accrued read cost for the SST in Equation 6.2. In this way, both normal and reverse compactions are used as a means to arrange the hottest data to be stored on the SSD and have the HDD only store the coldest data.

Reverse and informed compactions together achieve the first goal set out by LogStore: to ensure the hottest tables eventually reside on the SSD, and ensure the coldest tables are migrated to the HDD.

6.1.3 Write Path Optimization

During periods of prolonged data insertion, conventional LSM-tree based storage systems continuously schedule and execute compactions to maintain the rigid size invariants for its levels. By design, log-structured stores have a write-path that is computationally simple and performs I/O in large sequential batches. This results in SSTs accumulating very quickly on the youngest level, which also happens to be the smallest level. In contrast, compaction is more complex computationally — merging $k$ sorted lists each of size $n$ requires $O(kn \log k)$ time to perform — and involves significantly more I/O as Equation 6.1 reflects. Writes into the store outpace the compactions required to distribute data from young and small levels where SSTs rapidly collect, to older and larger levels. Practical log-structured storage systems (e.g. [2], [1], [3]) exacerbate the problem by throttling incoming writes if the system detects that compactions are lagging, even going so far as to stall writes altogether in the extreme case.

It is important to observe that while the function of compaction is to maintain the rigid leveled structure to provide bounded read latency, it is unnecessary if there is no read traffic to reap the benefits. Instead, LogStore optimizes the write-heavy case by relaxing the size constraints of the levels, relaxing the requirement that SSTs are disjoint within levels stored on the SSD and deferring compaction to the point where its avoidance begins to impact incoming read requests. As before, when a Memtable fills up it is immediately flushed to the youngest level on the SSD, but no compaction is between Level-0 and
Level-1. LogStore allows SSTs on SSD-resident levels to overlap, but keeps SSTs on the HDD level disjoint to ensure at most one disk seek on HDD and abide by the third goal of LogStore’s design. In a sense, LogStore views the SSD as a very large buffer that collects SSTs during write-heavy workloads. While SSTs are now allowed to overlap when stored on the SSD, we still maintain disjoint and sorted buckets in the level’s histogram.

Deferring compaction enables extremely fast insertion speed since we execute fewer compactions overall during heavy write traffic. Any compactions that do occur are triggered when the total data size of SSD-resident levels has exceeded a threshold, at which point a compaction is required to reclaim space. In the next section, we apply an optimization to reduce the need to compact to HDD in the presence of a skewed workload, further improving write speeds. It should be noted that though the write optimization allows writes to proceed unimpeded, LogStore is still responsive and will compact frequently read overlapping SSTs on the SSD if it is cost-beneficial to do so. We apply the same logic as reflected in Equation 6.1 and 6.2 but adjust for the degree of overlap. If reads are sufficient to warrant the compaction, we issue one on the SSD whose output is retained on the SSD.

6.1.4 Staging Compactions

The write-path optimization described in Chapter 6.1.3 creates two peculiarities that we describe and solve below.

Relaxing the disjointedness condition for SSTs within a level complicates the read path. Where previously LogStore guaranteed at most one SST per-level could contain a given key, this is no longer the case on the SSD — LogStore still guarantees at most one candidate SST on the last level on HDD. This problem is ameliorated to a degree by the fact that any redundant SST lookups are always performed on the SSD, but we cannot let it go unchecked. For this reason, LogStore tracks access history using its histograms to detect when excessive overlap is impacting read performance. If and when reads are
negatively impacted, LogStore schedules a staging compaction. A staging compaction is a regular compaction with the caveat that the source and target level are the same and is always on the SSD. Staging compactions solve the read-path problem by converting multiple overlapping SSTs into disjoint SSTs on the SSD, thereby reverting to the single-SST-per-level guarantee for read requests.

The criteria for when staging compactions are executed follows logic very similar to that of reverse compactions described in Chapter 6.1.2: if the total I/O cost needed to access an SST over the previous $N$ times exceeds the I/O cost required to compact the $T$ overlapping SSTs on the SSD, LogStore will schedule a staging compaction of these SSTs.

Staging compactions execute very quickly since they run entirely on the SSD. Additionally, they have the added benefit of teasing apart hot and cold data stored together in wide SSTs into separate SSTs that are treated and tracked independently. This is important to ensure that LogStore never evicts warm data from the SSD simply because it is stored together with colder data. LogStore executes staging compactions if the Level-1 SSTs chosen for compaction to HDD overlaps more than a threshold number of Level-2 SSTs. Staging compactions help to ensure hot and warm data is retained on the SSD and helps to perform the majority of the I/O intensive preparatory work on the fast high-bandwidth SSD device rather then the HDD. Staging compactions help to achieve the first two design goals in LogStore.

### 6.2 Evaluation

In this section, we evaluate LogStore under a variety of different workloads and system parameters. We implemented LogStore on top of LevelDB [2], a popular embedded key-value store and an open source implementation of the LSM-tree data structure. We choose LevelDB as our base primarily due to its well-documented, small and modular codebase. LevelDB has served as a base for several new key-value stores, including [1]
and [3], that each implement orthogonal optimizations easily portable to LogStore.

LevelDB implements a custom benchmarking utility that is capable of executing many common key-value workloads. However, the utility does not include a reliable skewed request distribution such as a Zipfian. To solve this problem we use the popular Yahoo! Cloud Serving Benchmark (YCSB) [13] to generate workload traces, which we replay through a modified version of LevelDB’s benchmarking tool with almost no overhead.

Our evaluation considers three workload groups: read-only, mixed read-write, and write-only. We further decompose the mixed read-write group into three types: read-heavy, balanced and write-heavy. The workload specifications are listed in Table 6.1.

<table>
<thead>
<tr>
<th>Workload</th>
<th>% Read</th>
<th>% Write</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read-Only</td>
<td>100%</td>
<td>0%</td>
</tr>
<tr>
<td>Read-Heavy</td>
<td>90%</td>
<td>10%</td>
</tr>
<tr>
<td>Balanced</td>
<td>50%</td>
<td>50%</td>
</tr>
<tr>
<td>Write-Heavy</td>
<td>10%</td>
<td>90%</td>
</tr>
<tr>
<td>Write-Only</td>
<td>0%</td>
<td>100%</td>
</tr>
</tbody>
</table>

Table 6.1: Workload specifications

We execute the experiments in four phases: cleaning, data loading, warming and the workload phase. The cleaning phase removes any data from previous experiments, cleaning devices and flushing operating system caches. In the data loading phase, a single thread sequentially inserts new key-value pairs into the store. When the data loading is complete, the benchmark waits for any running compactions to complete and for the system to quiesce. In the warming phase, we run a simple sequential read workload to allow LogStore and operating system caches to warm up. Finally, the target workload is run against a primed store where we collect metrics. Each reported metric is the result of an average over three executions of the experiment.

We run all our experiments on an Intel Xeon X5450 quad-core system operating at 3GHz. The system has six DIMM slots and can support up to 64GB of RAM. Since many of our experiments vary the amount of installed RAM, we clearly state the RAM
configuration at the beginning of each section. The system contains a single 4TB 7200rpm
hard-disk drive and a single 240GB Intel X520 SSD that are both connected over a SATA
3.0 600 MB/s interconnect. Our dataset has 100 million key-value pairs where each key
is 16 bytes and each value is 1024 bytes, totalling in roughly 100 GB of data. We disable
compression (of values) to ensure the experiments focus purely on I/O. YCSB’s Zipfian
request distribution has a configurable skew parameter, $s > 0$. The skew observed in the
workload is directly proportional to $s$; our experiments use a skew parameter $s = 0.9$,
which is sufficiently skewed to make use of each storage device. In the experiments
that follow, we compare our LogStore system against LevelDB version 1.17 running either
together entirely on HDD or entirely on SSD with no further modifications. For convenience,
we henceforth refer to these two database configurations as LDB-HDD and LDB-SSD,
respectively.

6.2.1 Read-Only Performance

In this section, we observe how each system behaves when we run a read-only workload
and we vary the amount of memory available to the store. The keys are chosen from a
Zipfian distribution. For these experiments, we benchmark LDB-HDD, LDB-SSD and
LogStore, but also benchmark what we called LogStore-Unoptimized. LogStore-Unoptimized is
our very own LogStore system, but we disable all the optimizations outlined in this work. LogStore-Unoptimized randomly chooses which SSTs to store on levels residing on the SSD and which will be migrated to the HDD.

LevelDB and other LSM-tree based storage systems do little to account for skew in a workload aside from implementing custom caches. LogStore is designed to recognize inherent skew in a workload and adapt by arranging for the hottest data to eventually migrate onto the SSD and slowly evict the coldest data to the HDD. The expectation in a skewed read-only workload is that there is a time after which LogStore has identified the hottest data and optimized data layout so as to achieve much of the performance
of LDB-SSD, but with a fraction of the requisite SSD capacity. Our expectations are validated in Figures 6.3 and 6.4, which show throughput and latency results, respectively. In the figures, we see that LDB-SSD offers the highest throughput across all of the RAM configurations we use. However, we also see that with small amounts of RAM in comparison to the total data size (in this case 2%), LogStore is able to achieve almost half the throughput of LDB-SSD and is almost five times faster than LDB-HDD. This is because LogStore is able to identify the frequently accessed data, which in these experiments is initially stored on the HDD, and migrates this data to the SSD to reduce read latency. Additionally, we see that LogStore-Unoptimized is not even twice as fast as LDB-HDD. This is because it is entirely unaware of the workload characteristics; any performance improvements LogStore-Unoptimized sees over LDB-HDD is only because of the the SSD. LogStore-Unoptimized clearly illustrates that arbitrarily storing half your data on an SSD (and not using the optimizations we offer in this work) yields suboptimal performance compared to LogStore.

The first two levels of LogStore receive 92% of the accesses, with the remaining 8% going to the last level on the HDD. Even with this tremendous skew towards the SSD, LogStore remains bottlenecked by the HDD since the HDD is more than 40 times slower than the SSD. This is expected based on the analysis we derived in Chapter 5.2.1. Moreover, as we increase the amount of RAM in the system, the throughput of LevelDB rises quicker than that of LogStore. The reason is because RAM is actually absorbing many of the accesses that would normally be destined for the SSD, while the HDD sees the same access frequency, and therefore remains the bottleneck. We further investigate this peculiarity in the following sections.

An interesting point to look at is when moving from 2GB of RAM to 4GB, the jump in performance for LDB-SSD is more than double. This is because LevelDB relies heavily on memory-mapped I/O that results in significant page-fault rates if the VM space exceeds available memory. This is no problem on HDD because the physical disk I/O dominates
the time taken to handle the fault itself. This is not the case when operating on a fast SSD as the page-fault machinery actually has an impact on the I/O path to the SSD. To validate our hypothesis, we ran the same experiment on LDB-SSD, but disabled all memory mapped I/O. The result was near 7K reads/sec with almost no page-faulting. LogStore does not suffer from this problem because it does not perform memory-mapped I/O.

**Effect of SSD size on LogStore**

In this section, we look at how LogStore performs as we vary the size of the SSD LogStore is allowed to use. The intuition is that overall the performance would rise as SSD sizes increase, but of particular interest is the proportional performance improvement and whether it makes economic sense. A Zipfian distribution has a long tail that suggests there may be a point beyond which additional investment in larger SSDs will yield diminishing returns. To investigate this we execute the same read-only workload, but we vary the size of SSD available to LogStore while keeping both the total data size and amount of available memory constant. The results, plotted in Figure 6.5, measures both the overall performance of LogStore in each configuration and the hit rate on the SSD, i.e., the percentage of accesses that logically hit levels stored on the SSD. As shown in the figure, while the overall hit rate on the SSD is trending up, the efficiency-per-byte is decreasing, clearly indicating diminishing returns. However, because the SSD is more than an order of magnitude faster to randomly access than the HDD, the ability to divert even the smallest amount of traffic away from the HDD to the SSD (or into caches) yields significant performance gains. This is one of LogStore’s primary goals and one of the reasons it is able to perform almost five times faster than LDB-HDD.
Adaptability of LogStore

Figure 6.6 shows the performance of LDB-HDD, LDB-SSD and LogStore as a time-series graph over the duration of a read-only workload after we completely switch the request key distribution from a latest distribution to a Zipfian key distribution. Before the experiment began, LogStore had optimized data placement to maximize performance for the latest distribution it had previously observed. The experiment demonstrates how adaptive each system is to changing skewed key accesses.

Unsurprisingly, LDB-SSD performs the best with the highest throughput and shortest run-time. There is a very brief warm-up time where caches are filled, but after this period LDB-SSD provides a stable and fast throughput for the remainder of the workload. Similarly in LDB-HDD, we see a stable throughput over the workload, but at a significantly lower throughput - roughly 240 operations per second. The reason LevelDB offers such a stable throughput is because it does no data layout optimization during the workload, meaning no compactions or data movement is done at all. Therefore, all of the device bandwidth is used for application traffic and the caches are fully utilized. In contrast, LogStore identifies the skewed read pattern almost immediately and begins optimizing the data layout by scheduling reverse compactions to migrate the hottest data to the SSD. Reverse compactions place non-negligible demands on the I/O system, all the more
so when using a slow storage media like the HDD. This is why the read throughput of LogStore is initially lower than that of LDB-HDD since reads are competing with compactions for limited I/O to the HDD. As LogStore begins to fill its youngest levels with the hottest SSTs, the read throughput climbs steadily and eventually matches LDB-HDD. When the SSD has reached capacity housing only the hottest data, compactions subside and SST evictions to the HDD are no longer required. At this point, LogStore’s read throughput jumps up to more than 6x that of LDB-HDD and stabilizes for the remainder of the workload. If there was a change in the request access pattern, LogStore repeats the process of identifying the hot spots in the accesses and optimizing the data layout by bringing hot data to the SSD when it is cost-beneficial to do so.

**Effectiveness of read cost-model**

In this section, we investigate the accuracy of the read-only cost model developed in Chapter 5.2.1. We use the model to analytically predict the expected throughput of each system and compare these results with results of the experiments in Chapter 6.2.1. We focus on a system with 8GB of RAM; an HDD with 10ms random seek latency (100 seeks/sec) and 150MB/s of sequential read and write speed; and an SSD with 0.25ms random seek latency (4000 seeks/sec) and 240MB/s of sequential read and write speed.
Additionally, we use a Zipfian skew of $s = 0.9$.

Figure 6.7 graphs the predicted and experimental read throughput of each system for a read-only workload. We observe that the predictions are close, but not equal. The largest deviation from the mode’s prediction is in LDB-HDD of about 27%. We believe this is because the LDB-HDD system does not run sufficiently long enough to ensure the hottest data is brought into the cache. LDB-SSD has roughly a 15% difference between what the model predicts and what the experiments show. This is because the I/O latency in LDB-SSD is not dominant in the read-path as it is in LDB-HDD. Acquiring locks, updating shared data structures and system calls add non-negligible overhead that is not captured in the model. LogStore has minimal variation between the predicted and actual numbers, roughly 8%, which is attributable to the imperfect nature of real-world caches. Our model expects only the hottest data will be stored in memory, while in actuality there is a portion of RAM that will store cold keys from HDD. As such, the model, to a degree, overestimates the efficiency of the cache.

The results of this chapter offer validation to the analytical cost model we constructed in Chapter 5.2 and show its applicability to both hybrid and homogenous storage environments. Though we focused on a specific system and environment configuration for the results here, we have verified the results for each of the RAM configurations we used in the previous read-only experiments section.

6.2.2 Write-Only Performance

We now look at how LogStore performs when we execute a write-only workload. This workload updates keys in the store with new values equivalent in size to the original. Like before, the keys are selected from a Zipfian distribution and we configure the system with 8GB of RAM.

LSM-trees are write-optimized primarily because the data structure is able to convert all write traffic into purely sequential I/O and the in-memory component buffers multiple
updates, and therefore amortize the cost of flushing the component to persistent storage. We expect both LevelDB and LogStore to offer very high write-throughput, but we need to demonstrate that the optimizations we made to LogStore outlined in Chapter 6.1.3, outperforms LevelDB. The results of our write experiments in Figures 6.8 and 6.9 indeed show that LogStore is able to outperform both LDB-HDD and LDB-SSD. The reason LevelDB’s write throughput is lower than LogStore is because it spends the majority of its execution time performing compactions. LevelDB’s implementation is such that reads never block each other, reads never block writes and vice versa. However, compactions may block writes if the system detects that compactions are lagging. In our experi-
ments, the client is inserting and updating data at a speed much higher than LevelDB compactions are able to execute, and hence, LevelDB throttles incoming writes to cope.

LogStore is able to handle a much larger volume of writes before triggering any compaction, owing to the write optimization described earlier in Chapter 6.1.3. In fact, many of the compactions that are triggered to move data from the SSD to the HDD are converted to staging compactions that operate on the SSD alone. In a skewed write-heavy workload, many of the overlapping SSTs on the youngest levels contain overwritten, and hence, useless key-value pairs. Staging compactions easily and quickly remove overwritten values and can significantly reduce the size of the levels on the SSD; staging compactions often obviate the need for a subsequent compaction to HDD, saving on costly I/O. This is why LogStore’s write throughput is higher than LevelDB. Additionally, since LogStore only uses three levels where LevelDB uses seven, the degree of write amplification is much lower. Deferring and staging compactions allow the SSD to absorb updates to the most frequently accessed keys, as the write cost model in Chapter 5.2.2 predicts. LevelDB, on the other hand, will unnecessarily propagate older versions of the hottest keys through the older levels.

Write-only performance over time

In this section, we investigate how each of the systems perform over time as we run a write-only workload. Figures 6.10(a), 6.10(b) and 6.10(c) graph the throughput of LogStore, LDB-HDD and LDB-SSD, respectively, over the duration of a write-only workload with a Zipfian key distribution. LogStore’s throughput has the interesting characteristic in that it is periodic. When write throughput is highest, around 30K writes/sec, any and all compactions that execute, run on the SSD itself with no I/O going to the HDD — these are staging compactions. When write throughput is at its lowest, around 800 writes/sec, compaction is occurring from SSD to HDD and writes slow down to ensure SSD usage is maintained below maximum capacity. When LevelDB is running on SSD as in Figure
Figure 6.10: Throughput of write-only workload over time
6.10(c), the throughput averages roughly 4000 ops/sec, but there is variability as writes are throttled due to compaction. LevelDB running on HDD has an even lower average throughput with even higher variability. Though compactions involve only sequential I/O to run, their initialization involves a number of random seeks equivalent to the number of SSTs involved. Though LevelDB tries to limit the sizes of compactions, the random I/O cost of the HDD compared to the SSD is non-negligible. Since compactions run faster on an SSD than on the HDD, a lower majority of writes are throttled in LDB-SSD than in LDB-HDD.

Effectiveness of write cost-model

We now look at the accuracy of the write cost-model we presented in Chapter 5.2.2. We use the same device characteristics as in the read cost-model evaluation in Chapter 6.2.1, and in addition assume the record size is 1 KB.

If we use Equation 5.7, the expected throughput of LDB-HDD and LDB-SSD is 1163 ops/sec and 2790 ops/sec, respectively. It is vital to note that our model provides a lower bound. The reason for the difference is because the model expects all inserted data to be compacted \( L - 1 \) times. In practice, the number of levels a record will go through in a finite workload is a function of the amount of data that is inserted. Specifically, if we insert \( D \) MB of data, every record will be compacted a total \( \log_M(D) - 1 \) times. If we use this observation, the model correctly predicts the throughput we observe in the experiments.

Using Equation 5.8 on LogStore, the model predicts a throughput of roughly 4876 ops/sec. The difference between the analytical model and the results of the experimentation is attributed to the effect that staging compactions have. In essence, not every record will migrate to the HDD. In fact, in a write-only workload, only half of the total inserted data set will arrive on the HDD. The model is more abstract and does not capture this very specific optimization.
6.2.3 Mixed Read-Write Performance

In this section, we evaluate how each system performs when we run each of the three mixed read-write workloads from Table 6.1: read-heavy, balanced and write-heavy. We also benchmark a modified version of LevelDB that leverages a custom SSD row cache. The design and implementation of the row cache is identical to that of the Cassandra row cache described in Chapter 4.2. A small (64K) in-memory block buffers insertions into the SSD cache before being appended to an append-only cache file on the SSD. A compact in-memory index is used to efficiently find the offset of a given key into the cache file. The write path of LevelDB has been modified to insert keys into the cache if an older version of the key exists and every read always first checks the SSD cache for the latest version of the key. We configure the systems with 8GB of RAM and use a Zipfian key distribution with a skew parameter, $s = 0.9$. For LevelDB running with the SSD cache, all data is stored on the HDD and the SSD cache is configured to be 50% of the total size of the dataset. The results of the experiment are shown in Figures 6.11 and 6.12, with the former graphing average throughput and the latter graphing average latency.
Performance under a read-heavy workload

In the read-heavy experiment, we observe that LogStore is more than 7x faster than LDB-HDD and is only 30% slower than LDB-SSD, all while only storing a fraction of all data on SSD. We note first that with the addition of 10% writes to the workload, all three systems have a reduced throughput in comparison to the read-only workload from Chapter 6.2.1. This drop occurs for two reasons: by the inherent nature of LSM-trees operate and due to the compaction process. Since LSM-trees are log-structured, the hottest keys will frequently be flushed from the Memtable to multiple SSTs. While the Memtable can serve some reads for hot keys, it is limited in size which means the majority of read traffic will need to traverse the various persistent tree components. In a read-only workload, the system reaches a steady state where the operating system page cache is populated with only the hottest keys. However, in a mixed workload with writes, compactions have to be run. Compactions heavily pollute the page cache each time they execute, while having the added effect of also moving hot keys between new SSTs. The efficacy of the page cache in a mixed workload is dramatically reduced in a mixed workload in comparison to the read-only workload.

As described in Chapter 2, many modern LSM-tree systems, including LevelDB, implement an optimization whereby frequently accessed SSTs across levels that overlap in
key ranges are compacted into a single level. This optimization, which we term read-triggered compaction, is meant to reduce read latency by reducing the number of SSTs to check for hot keys by one. LevelDB executes read-triggered compactions very often in read-heavy workloads. These compactions ruin the page cache and steal limited I/O capacity from client traffic. It is for this reason that LDB-SSD experiences such a precipitous drop in throughput of almost an order of magnitude from the read-only case. Since compactions complete quicker on SSD than HDD, they run more often through each of the seven levels, each time polluting the cache. Compactions are also a very I/O-intensive process and contend for bandwidth to the device with application traffic. LDB-HDD is especially affected by the loss of cache efficiency since this results in a larger percentage of read operations having to pay the cost of a very slow seek on HDD, hence the low throughput we see in LDB-HDD.

LevelDB with an SSD cache does not suffer from the caching problem because it maintains its own application cache. However, the benefit of the cache is not very pronounced primarily because the HDD device is the bottleneck in many ways. In the read-heavy case, the SSD cache is able to achieve almost 90% hit rate for read traffic. However, the 10% of reads that need to traverse the levels of the tree still need to compete against compaction traffic for I/O bandwidth to the disk. This result stresses just how slow the disk is and how much of a bottleneck it can be. We also note that write latency in LevelDB when the SSD row cache is enabled is higher than when it is disabled due to the overhead incurred to dual-write insertions into both the Memtable and the SSD row cache.

LogStore does not escape the problem of cache inefficiency since it is an LSM-tree, but the effect is not nearly as severe as in LevelDB. This is because LogStore executes significantly fewer compactions over the course of the workload due to its write optimization and cost-based analysis. While LogStore does issue read-triggered compactions, they are always run at cost-opportune times based on the observed workload. The remaining
compactions are performed only to preserve space on the SSD as writes are accepted into the store. Fewer compactions overall results in less churn in the application and operating system page cache and more I/O and CPU resource availability of the system to serve application reads and writes. As seen in Figure 6.13, LogStore is also able to drive the majority of read requests to the SSD; 12% of read requests are served from the Memtable in memory, almost 80% is served from levels stored on the SSD, while the remaining 9% have to touch the HDD but only require one seek.

**Performance under a balanced workload**

In the balanced workload experiment, both LDB-HDD and LDB-SSD have improved performance over the read-heavy workload primarily because writes in any LSM-tree based system are fast. However, both systems execute a large number of compactions (anywhere from 3000-5000 compactions per 2 million keys) that throttle incoming write traffic. LogStore executes almost half as many compactions, roughly 1900 compactions per 2 million keys. This results in more device bandwidth availability for application traffic and less cache churn. The performance improvement of LogStore over the read-heavy workload is not as pronounced because LogStore allows overlapping SSTs on the SSD, and hence, will require more SST probes on average than in LevelDB. However,
these SST lookups only occur on the SSD and complete very quickly, especially when table indexes can be maintained in memory. LevelDB with an SSD row cache achieves very good performance because slower reads occur less frequently in a balanced workload. Even in this workload, the SSD row cache is able to achieve almost a 90% hit rate for reads. This means that 45% of all operations incur the latency of an SSD seek, 5% require a full HDD seek and the remaining 50% of operations require the amortized cost of a sequential Memtable flush. However, LevelDB does not implement any of the write-patch optimizations that LogStore does and does not match LogStore’s performance in the balanced workload.

As in the read-heavy case, LogStore is able to utilize the SSD by ensuring the majority of accesses (almost 80% as seen in Figure 6.13) of read traffic is served from the SSD. It should be noted that since there is no locality in the key accesses that hit the HDD, the average seek time to HDD is especially bad. Our cost model predicts this and we observe that long HDD seek latency heavily weighs the average read latency despite skew to the SSD. Finally, in the balanced workload with sufficient writes, LogStore needs to compact some of the writes to the HDD to maintain space on the SSD. This compaction process, as in LDB-HDD, is very slow on HDD.

**Performance under a write-heavy workload**

All of the systems have increased performance over their balanced workload numbers primarily because writes are faster than reads in an LSM-tree. In each of the LevelDB-based systems, the reduction in read percentage results in fewer read-triggered compactions. This means that, on average, a larger number of SSTs across the levels have to be probed to service a read request. In these experiments, LDB-HDD and LDB-SSD require 7 SST/read on average to satisfy a read operation. Since compactions run slower on HDD, the probes happen faster since the hot pages have a higher chance of being in cache. This is not a problem for LevelDB with the SSD row cache since compactions operate on a
different device than cache lookups and the page cache is not as advantageous since SSD seeks are fast. The SSD row cache continues to provide anywhere from 85% to 90% hit rates for reads, but also add nonnegligible latency to write traffic that does not exist in the unmodified LevelDB with the SSD cache disabled.

In LogStore the majority of compactions that execute are staging compactions on the SSD followed by informed compactions to the HDD. These staging compactions are meant to break apart wide SSTs in an effort to ensure warm data is retained on the SSD, control the degree of overlap between SSTs on SSD and HDD and to provide a final sorted order to a key range prior to a subsequent data migration to the HDD. As in the write-only experiments, as the ratio of reads falls, fewer read-triggered compactions are run and the SSD begins to simply buffer SSTs. LogStore is able to defer compaction to the point where data reclamation is required to free space on the SSD. Moreover, since we use a skewed distribution, fewer compactions to the HDD are required since our staging compactions can remove heavily overwritten data. Having overlapping SSTs on the SSD does mean that LogStore probes more SSTs on average than LevelDB, but these probes are always performed on the SSD. From Figure 6.13, we still see that LogStore is able to ensure 75% of read accesses go to the SSD (with 12% being served from memory).

Effectiveness of read-write cost-model

In this section we evaluate the effectiveness of the read-write cost model presented in Chapter 5.2.3. As before, we assume the same device and system characteristics as in Chapter 6.2.1 and Chapter 6.2.2. Table 6.2 lists the results of applying the read-write model to each system for each of the three mixed read-write workloads and the experimental results from this section.

Our model predicts a throughput for LDB-SSD in a read-heavy workload that is more than 5x higher than what we observe through the experiments. The reason for this, as mentioned previously, is primarily due to an optimization within LevelDB that seeks to
Table 6.2: Analytical and experimental throughput for read-write workload for LDB-HDD, LDB-SSD and LogStore

<table>
<thead>
<tr>
<th>Workload</th>
<th>System</th>
<th>Analytical (ops/sec)</th>
<th>Experimental (ops/sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read-Heavy</td>
<td>LDB-HDD</td>
<td>111</td>
<td>97</td>
</tr>
<tr>
<td></td>
<td>LogStore</td>
<td>1082</td>
<td>725</td>
</tr>
<tr>
<td></td>
<td>LDB-SSD</td>
<td>5618</td>
<td>1095</td>
</tr>
<tr>
<td>Balanced</td>
<td>LDB-HDD</td>
<td>192</td>
<td>201</td>
</tr>
<tr>
<td></td>
<td>LogStore</td>
<td>1612</td>
<td>733</td>
</tr>
<tr>
<td></td>
<td>LDB-SSD</td>
<td>4424</td>
<td>2544</td>
</tr>
<tr>
<td>Write-Heavy</td>
<td>LDB-HDD</td>
<td>721</td>
<td>683</td>
</tr>
<tr>
<td></td>
<td>LogStore</td>
<td>3184</td>
<td>2050</td>
</tr>
<tr>
<td></td>
<td>LDB-SSD</td>
<td>5010</td>
<td>4310</td>
</tr>
</tbody>
</table>

compact frequently accessed key ranges that overlap between levels. This compaction process is not only very I/O intensive, and hence reduces bandwidth for regular application traffic, but it also means that the hottest keys remain in caches less often since they are continuously read and re-written in different SSTs. We fix this very problem of running LevelDB on SSD in Appendix A. This problem doesn’t exist when LevelDB is run on HDD since compactions run much slower, meaning that read operations for hot keys have a higher probability of being served directly from memory. The model’s prediction and the experimental results converge as the write ratio increases because the optimization isn’t triggered as often due to a lower read percentage.

The model’s prediction for LogStore is accurate across the workloads with the exception of the balanced workload. This is because the model does not capture the effect of LogStore’s write optimization. In a balanced workload, LogStore sees a large number of writes and simply allows the SSTs to flush onto the SSD unimpeded. However, since there is an equivalent number of reads to the same key ranges, LogStore must continuously perform compaction so as not to impact the read latency. Though these compactions are triggered at optimal times (when the cost of compaction and the I/O required to read a set of SSTs is equal), compactions steal bandwidth from the SSD and negatively impact
overall throughput. In the write-heavy case, the HDD becomes severely bottlenecked by
having to serve up to 10% of read traffic at very high latencies. These reads must contend
with informed compactions to ensure the SSD has sufficient room to accept new writes.
Chapter 7

Conclusions and Future Work

7.1 Conclusions

This thesis explored how modern key-value stores can be designed, implemented and optimized for hybrid storage environments. We began by exploring how to use an SSD as a cache to store a subset of data and internal schema information in a popular extensible row-store. The two techniques we implemented and evaluated offered improved read throughput – a common shortcoming of key-value stores – and reduced overall space usage by removing redundant data.

We constructed a formal analytical cost model to estimate the performance of a generic log-structured hybrid storage system. The model was parameterized to account for the characteristics of the workload, the specific performance characteristics of each available device in the hierarchy, and the distribution of data across each device. The model is also flexible enough to predict the performance of a log-structured system when stored on either a homogenous or heterogenous storage environment.

Finally, we designed, implemented and evaluated LogStore, a completely re-architected key-value store specifically for hybrid storage environments. The design of LogStore was heavily influenced by insights from the analytical model we constructed and is highly
optimized to deliver both the highest read and write throughput across a range of workloads. In contrast to using SSDs as a cache, LogStore uses SSDs as directly part of the storage hierarchy, either storing data on SSD or HDD, not both. Through extensive evaluation, we demonstrated that our analytical model is accurate in its prediction, and that LogStore outperforms LevelDB, another modern key-value store.

7.2 Future Work

There is extensive opportunity for future work in each of the three contributions we make in this thesis.

One of the conveniences provided by Cassandra is the ability to manually move and load SSTs across physical instances. This is possible because SSTs contain all the necessary information to represent a partition of the table. This capability, however, is not possible for tables that employ the schema extraction technique. Schema IDs are non-replicated identifiers that are local to a single node. Moving an SST encoded with schema IDs from one node to another may cause irreconcilable conflicts since schema definitions may be missing or different altogether. We can solve this problem by making the schema catalog a first-class citizen at the cluster-level. Such a modification will require replicating schema IDs across the cluster as they are created. This can be implemented by piggy-back on the replication machinery already present in Cassandra for regular data, but here we use it for metadata.

One of the limitations of LogStore is that it works at the granularity of SSTs. The histograms that LogStore maintains along with each form of compaction that we designed for hybrid storage operate at the level of SSTs. This is not a problem in a skewed workload that has locality or clusters. In such a case, an SST is likely to capture and store this locality and the histograms will be useful. For sufficiently large workloads that are both skewed and do not have locality, operation at the SST level may be too coarse grained.
Moving an SST with only a handful of keys that are very frequently accessed is not optimal in such a case. Instead, it would be ideal to employ a dynamic histogram within SSTs that grows finer as access frequency of keys grows. With this in place, LogStore is better suited to identify the hot data within an SST and tease it apart from the colder portions. In this way, LogStore can operate at a small granularity than an SST. The same cost benefit analysis we construct in this thesis still applies, but the size and width of SSTs is dynamic.
Bibliography


Appendices
Appendix A

Optimizing LevelDB for SSD

LevelDB includes several optimizations that aim to improve read and write performance. One such optimization strives to collapse and merge overlapping ranges of keys between levels into a single level if the key range is read frequently enough. This optimization is borne in recognition of the fact that an optimal layout for an LSM-tree is one where only a single on-disk tree component exists. Such an organization minimizes read latency by only having to perform one I/O. LevelDB’s implementation of this technique is sub-optimal when operating on an SSD; LevelDB is too aggressive in issuing read-triggered compactions. Since compactions are I/O-intensive, executing them frequently limits the available bandwidth for the system to support regular read/write client traffic. Additionally, excessive compactions have a negative impact on the effectiveness of the cache, as we saw previously. All of this results in suboptimal performance. We observed this effect in Chapter 6.2.3, especially in the read-heavy case where LDB-SSD’s performance dropped from almost 11000 ops/sec in the read-only case to 1000 ops/sec, with the only difference being the addition of 10% write traffic. Moreover, our very own cost model predicted a throughput in the read-heavy case for LDB-SSD of roughly 5000 ops/sec, almost 5x faster than what we observe.

The problem stems from cost logic that we discovered inside LevelDB. Though Lev-
Appendix A. Optimizing LevelDB for SSD

Figure A.1: Number of performed compactions in LevelDB and our optimized LevelDB in a mixed read-write workload

LevelDB’s cost-analysis resembles our formulations in Equation 6.1 and Equation 6.2, it is not optimized for SSD. To fix the issue and optimize LevelDB for SSD, we apply cost logic taken from LogStore and leverage current SSD device parameters. The new model chooses to delay read-triggered compactions so long as it is more cost beneficial not to do so. However, when the cost of compacting an SST (with overlapping SSTs on the next level) is cheaper than the cost of probing the SST for the \( n \)th time, we issue a read-triggered compaction. The goal is to reduce the frequency of read-triggered compactions based solely on the speed of the SSD in order to improve overall throughput over a variety of mixed read-write workloads. Reducing compaction also means that data is pushed to larger levels much more slowly, which is not a concern in a homogenous storage environment.

To validate that our adjusted LevelDB cost model performs fewer read-triggered compactions, we benchmark both LevelDB and the optimized LevelDB on our SSD in the same configuration as described in Chapter 6.2. The two systems are identical in every regard with the exception of the adjustment we made to the cost model. We run the read-heavy, balanced and write-heavy workloads on both systems since these are the only configurations in which read-triggered compactions will occur.

Figure A.1 shows that, indeed, the optimized LevelDB does execute fewer compactions
Figure A.2: Distribution of read accesses across levels in LevelDB and our optimized LevelDB in a mixed read-write workload.

Overall. In the read-heavy case, LevelDB executes almost 4200 compactions for every set of 2 million operations, whereas the optimized LevelDB executes only 1300. The reduction in compactions by a factor of 3x is entirely due to the reduction in read-triggered compactions. The trend of reduced compactions in the optimized LevelDB continues (though reduced) as the write percentage increases and finally converges when there are no reads. When the write percentage is at 100%, no read-triggered compactions run at all since there are no reads in the system to trigger them.

As we have seen, the optimized LevelDB executes fewer compactions due to read requests. This optimization should result in a larger percentage of reads being serviced from lower (younger) levels in the tree since it takes a longer time for this data to be pushed to the larger (older) levels. We see this very effect in Figure A.2, which shows the percentage of read requests served by each level in both LevelDB and the optimized LevelDB. In the read-heavy case, almost 70% of reads are served by the last level in LevelDB, while only roughly 48% reaches the last level in the optimized version. This is because data moves through the levels more slowly since the optimization issues fewer compactions by design. This is shown clearly since Level 3 in LevelDB services 0.8% of
reads while it services almost 20% of reads in the optimization. This data would have already been pushed to the last level in LevelDB through compaction, but the model decided to wait. This trend continues as the write percentage increases and eventually disappears as fewer read-triggered compactions execute overall.

As a result of fewer compactions, the expectation would be that the unused bandwidth can be used to serve normal application traffic, yielding higher overall throughput. Figure A.3 verifies our expectation, clearly showing that the optimized LevelDB provides a 3x increase in throughput versus its unoptimized counterpart in the read-heavy workload. This increase in throughput is directly proportional to the reduction in the number of compactions triggered, as seen previously. As the write percentage increases, fewer read-triggered compactions execute, and the throughput of the two systems converge.