LogStore: A Workload-aware, Adaptable Key-Value Store on Hybrid Storage Systems

Prashanth Menon*, Thamir M. Qadah◇, Tilmann Rabl†, Mohammad Sadoghi‡, Hans-Arno Jacobsen#
*School of Computer Science, Carnegie Mellon University
◇Umm Al-Qura University and Purdue University
†Database Systems and Information Management Group, TU Berlin
‡University of California, Davis
#Middleware Systems Research Group, University of Toronto

Abstract—**Due to recent explosion of data volume and velocity, a new array of lightweight key-value stores have emerged to serve as alternatives to traditional databases. The majority of these storage engines, however, sacrifice their read performance in order to cope with write throughput by avoiding random disk access when writing a record in favor of fast sequential accesses. But, the boundary between sequential vs. random access is becoming blurred with the advent of solid-state drives (SSDs).**

In this work, we propose our new key-value store, LogStore, optimized for hybrid storage architectures. Additionally, introduce a novel cost-based data staging model based on log-structured storage, in which recent changes are first stored on SSDs, and pushed to HDD as it ages, while minimizing the read/write amplification for merging data from SSDs and HDDs. Furthermore, we take a holistic approach in improving both the read and write performance by dynamically optimizing the data layout, such as deferring and reversing the compaction process, and developing an access strategy to leverage the strengths of each available medium in our storage hierarchy. Lastly, in our extensive evaluation, we demonstrate that LogStore achieves up to 6x improvement in throughput/latency over LevelDB, a state-of-the-art key-value store.

I. Introduction

Big data challenges are not characterized only by the large volume of data that has to be processed, but also by a high rate of data production and consumption i.e., high-velocity [30], [45], [36], [37], [44]. Explosion in data volume and velocity is commonplace in a wide range of monitoring applications. In modern monitoring applications, many thousands of sensors continuously produce a multitude of readings that have to be stored at a high pace, but have to also be readily available for continuous query processing. Examples of such applications include traffic monitoring [41], [39], smart grid applications [25], and application performance management [32], [22], [40].

Due to the recent data explosion, it has been increasingly challenging to rely on traditional database technology to offer a cost-effective solution to sustain the required performance. As a result, a new array of distributed and light-weight key-values stores have emerged to fulfil this need [43], [34], [7], [14], [26], [21], [46], [33], [42], [20]. Many of these key-value stores are designed to scale-out by incrementally adding nodes to the cluster. Typically, each individual node employs a custom (often embedded) storage engine to service local data requests, building the distribution fabric atop this federated storage. However, these storage engines tend to sacrifice their read performance in order to cope with the data velocity (i.e., write throughput) by avoiding random disk access when writing a record in favor of fast sequential accesses, thereby further increasing the number of random accesses required for reading a record. Fortunately the gap between sequential versus random access is disappearing with the advent of storage-class memory, e.g., solid-state drives (SSDs), that is built upon the philosophy of no moving parts

Building on the success of key-value stores and emerging storage-class memory, we present *LogStore*, a novel optimized storage approach that serves as a key building block for distributed key-value stores in order to sustain high-velocity and high-volume data. The intuition behind *LogStore* is the efficient use of modern hardware, especially modern storage technology such as SSDs in hybrid storage architectures. These technologies have significantly improved performance in comparison to traditional hardware [33], [50], [31]. However, classical data structures and algorithms cannot directly be applied due to the different characteristics of SSD devices. Also, the high cost of the new technology makes their exclusive use uneconomical in many cases. Therefore, hybrid storage approaches are being explored in both industry and academia (e.g., [9], [12], [21]), in which modern and traditional technologies are co-allocated to form a storage memory hierarchy. In contrast, our proposed technique is a redesign of key-value store technology in the light of new storage memory opportunities. While *LogStore* seeks to improve the performance of a single multicore machine with a hybrid storage hierarchy, it can easily be used as a building block in distributed key-value stores (e.g., [1], [14], [17], [16]).

In *LogStore*, we introduce a database staging mechanism using a novel, cost-based, log-structured storage system such that 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 contributions of *LogStore* are as follows: (1) An analytical cost model to estimate the performance of log-structured hybrid storage systems. The model accounts for access pattern and specific system characteristics, provides insights that guide the design of *LogStore* and reveals bottlenecks in LevelDB. (2) A new statistics-driven compaction process that retains only the hottest data on the SSD and evicts cold data to the HDD to achieve maximum throughput. (3) A new reversed compaction process that identifies hot data stored on HDD and migrates this data to the SSD through compaction. This technique also leverages statistics to remain adaptive to shifting workloads. (4) An optimization that enables faster write throughput by selectively deferring compactions based on access frequency. Reducing compaction execution offers faster overall throughput , and (5) a new compaction process that operates within a single level (termed staging compaction). This compaction process reduces the impact of having overlapping ranges of SSTs (which is unique to *LogStore*) on I/O performance of read operations.

The rest of this paper is structured as follows. In Section II, we present related work and a brief overview of data management inside a conventional log-structured storage system. Section IV develops an analytical model to estimate the performance of hybrid storage. In Section V, the *LogStore* architecture is presented. Section VI shows the experimental analysis of *LogStore*. In Section VII, we draw our conclusions and present an outlook on future work.

II. Background & Related Work

A well-known write-intensive structure is a log-structured merge-tree (LSM-Tree) introduced in [34]. The idea is to spread inserts across a set of B-Trees that are exponentially increasing in size. As new data is inserted, the inserts are batched and bulk loaded (i.e., sequential writes) first into smaller B-Trees; as the smaller trees fill up, the data is bulk loaded into the next level, larger B-Trees. Although, this structure supports fast insertions through bulk loading, it suffers from unexpected performance spikes due to merging the content of smaller to larger trees, thus, forcing the data to travel across all levels. In addition, the LSM-Tree has a poor read performance because every read request must be sent to all levels incurring many random I/O operations.

Unlike the LSM-tree, which makes use of multiple layers of B-Trees, modern key-value (KV) stores (e.g., [14], [26], [3]) 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.

Data is initially inserted into the Memtable, an in-memory data structure that supports in-place updates ([14], [26], [3]). 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 levels 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.

There has been a renewed interest in improving LSM-Tree based database systems. Luo and Carey provide a good survey of recent advances in this research area [29]. From the RUM conjecture [6] perspective, *LogStore* is write-optimized like other LSM-Tree based systems. However, it also improves read/write performance by exploiting modern storage devices and by using various novel compaction techniques and policies that are workload-aware. Therefore, we focus our discussion of related work to these aspects.

Modern Hardware. SSD characteristics has been studied well in the literature. Chen et al. [15], performed an extensive experimental study of SSDs. They have confirmed the exceptional performance of random read operations, and identified issues related to the performance of write operations. With respect to utilizing SSD in log-structured databases, in [50], the authors proposed to build *LevelDB* entirely on flash and to exploit the internals of flash using open-channel SSDs [35]. In particular, they modified *LevelDB* to support multi-threaded I/O that directly exploits the available in-device parallelism on SSDs [50]. Similarly, in [31], 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 [50], [31] that attempt to utilize the SSDs internal are complementary to our proposal, and can also be exploited in *LogStore* to maximize the usage of the SSD device.

In the same spirit as *LogStore*, SSDs were introduced in the storage memory hierarchy of Cassandra [33]. However, unlike *LogStore*, they used SSDs only to store meta-data information (such as record-level schema information) and used SSDs as a data cache (i.e., database bufferpool) to provide fast access to redundant copies of the data. In contrast, *LogStore* uses SSDs as a staging area and not a cache. The data is either placed on SSDs or HDDs and not both. Moreover, *LogStore* remains performant over a range of differing workload types, while a cache (either in memory or on SSD) is only beneficial for read-heavy workloads.

WiscKey [28] proposed an LSM-tree-based design that separates keys form values which improves compactions of SSTs because keys tend to be much smaller than values in size. HashKV [13] improves upon this idea and reduces the impact of garbage collection. Alternative to LSM-tree, index structures based on fractional cascading that exploit SSD characteristics has been proposed (e.g., FD-tree[27] and FD+tree [49]). We believe that the ideas of separating keys form values and fractional cascading based data structures can be used to in conjunction with *LogStore*'s techniques to further exploit SSD characteristics.

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., [11], [8], [12], [9]). 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 [12], which substantially improved both transactional and analytical query processing on IBM DB2. Although our vision is aligned with these promising results, our goal is to fundamentally redesign write-intensive data structures (introducing SSD-enabled staging of log-structured data stores) as opposed to adapting and tuning the existing data management systems to take advantage of SSD read performance (e.g., [11], [8], [12], [9]).

Compaction Management. Due to the frequency of compaction tasks, traditional log-structured storage systems may not efficiently ultilize modern hardware such SSD. In DirectLoad, Qin et al. [38] improve the synergy by using aligned block-sizes and a two-level architecture composed of a memory component and append-only files on SSD to eliminate the overhead of frequent compactions. However, the drawback of their approach is that files need to be garbage-collected to reclaim unused storage space, which can interfere with write-operations. *LogStore* achieves the same write-throughput improvement as DirectLoad by mitigating the impact of compaction tasks for write-only workloads but by using drastically different techniques, and does not require garbage collection.

Compactions can also have a negative impact on the system's buffer-cache, Ahmad and Kemme [5] proposed offloading the compactions functionality to dedicated compaction servers to reduce buffer-cache invalidations in distributed key-value storage systems. Teng et al.[48] proposes techniques to reduce contention between newly compacted pages and existing hot pages. Leaper [51] uses machine learning to predict and prefetch hot records to reduce invalidations. These ideas are orthogonal because they are concerned with the buffer-cache while in this work we are concerned with data placement on storage devices. Hence, they can be used by *LogStore* to improve further read performance.

While the compaction process runs in the background, lagging compactions can stall write operations. bLSM [46], schedules the merging process in regular intervals to avoid stalls. 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. In contrast, *LogStore*, optimizes for highly skewed and prolonged insertions by using the proposed techniques of deferred and staging compactions.

Yoon et al. [52] propose Mutant which uses a temprature-based approach to organize SSTs similar to *LogStore*. Unlike *LogStore*, however, Mutant focuses on the read path and does not optimize the write path. Therefore, it is expected to perform worse than *LogStore* with write-only workloads.

III. A Case for Log-Structured Hybrid Storage

Many distributed key-value stores employ an LSM-tree storage architecture for local storage. The LSM-tree is designed to optimize writes to the HDD; since data is always written sequentially in large files (blocks), this write access pattern also naturally fits the use of SSDs. In the following, we discuss the specific characteristics of typical SSDs and why log-structured data structures work well on them. We continue by

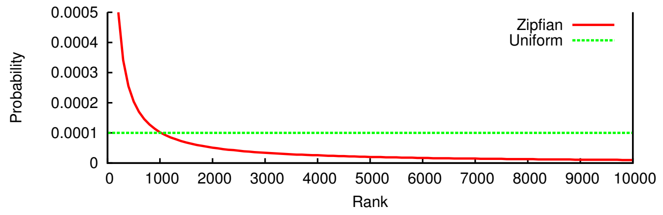

Fig. 1. Zipfian vs. uniform distribution

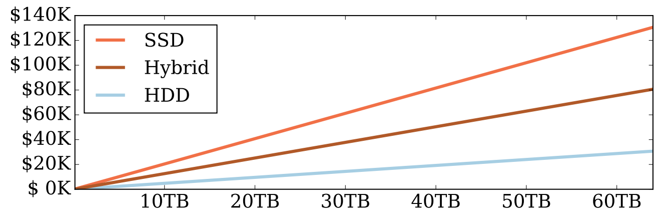

Fig. 2. Annual storage costs (USD) based on Google Cloud prices

demonstrating that SSD-only installations often do not provide the best price-performance, and motivate the use of hybrid setups using both SSD and HDD. We also provide an intuitive example to predict the performance of a log-structured data structure in a hybrid setup.

A. *SSD Characteristics*

Because most SSDs 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 [10]. NAND memory has an asymmetric read and write performance. This is due to the fact that 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.

Due to these issues, an SSD's internal controller uses advanced algorithms and data structures to evenly wear out all chips and limit the number of erase cycles in total [19]. These algorithms are hidden from the operating system in the FTL. Although modern systems continually improve the performance of the FTL (e.g., [47]), 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, know as write amplification [10].

Because of the log-structured data management in LevelDB and similar systems, and the management of relatively large data blocks (typically 2 MB), write amplification does effectively not occur on the FTL level, since only full erase blocks are overwritten. Thus, no data has to be copied to empty cells.

B. *SSD Price Performance Ratio*

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. Figure 2, shows the annual cost for provisioning storage up to 64TB[1]. As we can clearly see, a hybrid-based provisioning using 50% of the required storage capacity using SSD significantly reduces the cost (e.g., for 32TB, it can save about $25K annually). In a cloud-based environment, the cost of using SSD is about 4.25 more than HDD. Below, we discuss the price performance of SSD vs. HDD.

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 performance numbers in Section VI). 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%.

Other than replacing HDD completely by SSD, it is also possible to use it as an additional cache layer [23], [33]. This enables a hybrid architecture, where some data resides on HDD and some on SSD. The benefit of this solution is a more flexible trade-off between performance and cost, and an architecture that can take advantage of the different characteristics of HDD and SSD. In the following, we present a model to estimate the performance improvement that can be achieved by using a hybrid storage architecture.

C. Hybrid Storage Performance Analysis

Unlike previous approaches to hybrid storage systems (e.g., [9], [12], [23]), the approach presented in this work does not consider SSD as a buffer to the HDD, but as an independent storage device that complements the HDD. This means that from a system point of view, data is either stored on SSD or on HDD but not on both.

The basic idea is 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 60 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 [24], [18]. In Figure 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 3. 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 3. 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.

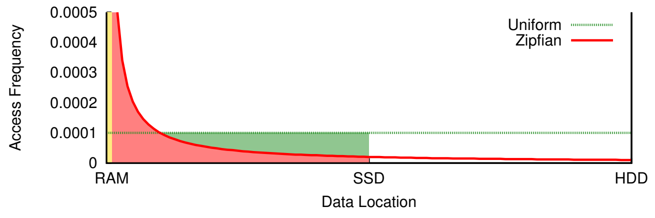

Fig. 3. Data placement on SSD and HDD

IV. Hybrid Storage Cost Model

In this section, we develop a general cost 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, and consider Zipfian and uniform key access distributions that are common in production workloads. 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.

[1]https://cloud.google.com/storage/pricing

A. Read-Only Model

In the following, we indicate uniform and Zipfian distributions with a superscript where a distinction is necessary (e.g., R^{u}_{HDD} 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^{u}_{RAM}) as 0 (and the throughput as 1). 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:

$$\begin{aligned} R^{u}_{HDD} &= (1 \quad |RAM|) \quad |HDD| \\ &= (1 \quad |RAM|) \quad (1 \quad |SSD|) \end{aligned} \tag{1}$$

The SSD access rate (R^{u}_{SSD}) is defined analogously. The total throughput for a read-only, HDD limited setup (TP'_{R}) can be estimated using the following equation:

$$TP'_{R} = \frac{TP_{HDD}}{L_{HDD} \quad R_{HDD}} \tag{2}$$

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:

$$TP_{R} = \begin{cases} \dfrac{TP_{HDD}}{L_{HDD}R_{HDD}}; & \text{if } \dfrac{TP_{HDD}}{L_{HDD}R_{HDD}} < \dfrac{TP_{SSD}}{L_{SSD}R_{SSD}} \\ \dfrac{TP_{SSD}}{L_{SSD}R_{SSD}}; & \text{else} \end{cases} \tag{3}$$

If we assume a Zipfian access probability, the access distribution is different and not equal to the relative data sizes on the respective devices. 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^{z}_{RAM} can be estimated using the formula for the Zipfian distribution:

$$R^{z}_{RAM} = \frac{\sum_{n=1}^{|RAM|} \frac{1}{n^{s}}}{\sum_{n=1}^{N} \frac{1}{n^{s}}} \tag{4}$$

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.

$$R^{z}_{SSD} = \begin{cases} \dfrac{\sum_{n=1}^{|SSD|} \frac{1}{n^{s}}}{\sum_{n=1}^{N} \frac{1}{n^{s}}} \quad R^{z}_{RAM}; & \text{if } |SSD| > |RAM| \\ 0; & \text{else} \end{cases} \tag{5}$$

$$R^{z}_{HDD} = 1 \quad R^{z}_{SSD} \quad R^{z}_{RAM} \tag{6}$$

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 3. If we assume an HDD limited setup, we can see in Equation 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.

B. 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 $(M + 1) \quad S_{SST}$ 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, $S_{SST}=e$, where e is the size of an individual record. Therefore, the total I/O required per-record during compaction is $2 \quad (M + 1) \quad e$. This record will undergo, at most, $L \quad 1$ compactions. Since compactions (and therefore writes) are always performed in a sequential manner, if we let W be the sequential read/write speed of the device (in bytes/sec), we can estimate the overall write throughput of the LSM-tree to be:

$$TP_{W} = \frac{W}{2(M + 1)(L \quad 1)e} \tag{7}$$

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 W_{HDD} and W_{SSD} be the sequential read/write speed of HDD and SSD, respectively, then we can use Equation 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:

$$TP_W = \begin{cases} \frac{W_{HDD}}{2(M+1)L_{HDD}e'}; & \text{if } \frac{W_{HDD}}{L_{HDD}} < \frac{W_{SSD}}{L_{SSD} \quad 1} \\ \frac{W_{SSD}}{2(M+1)(L_{SSD} \quad 1)e'}; & \text{else} \end{cases} \tag{8}$$

Equation 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. 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.

C. *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 3 from Section IV-A to calculate the read throughput, and use Equation 8 from Section IV-B 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 9.

$$TP_{RW} = \frac{1}{\frac{r}{TP_R} + \frac{w}{TP_W}} \tag{9}$$

D. *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.

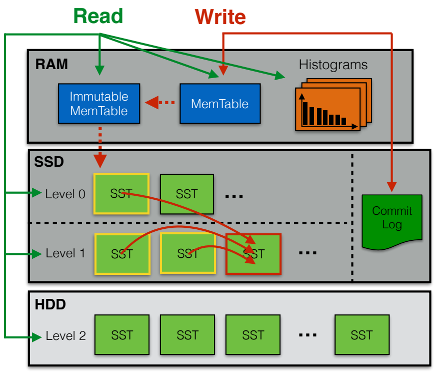

Fig. 4. *LogStore* architecture

V. LogStore Architecture

The model presented previously in Section IV 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*.

The architecture of *LogStore*, shown in Figure 4, resembles other log-structured data management systems as described earlier in Section II. Writes in *LogStore* are buffered 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* relies on the compaction process to achieve that threshold. When *LogStore* decides on the next compaction, it estimates the amount of data residing on SSD relative to the total expected database size after the

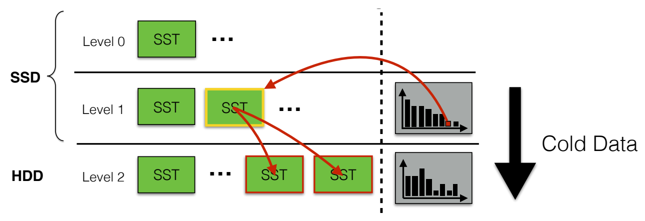

Fig. 5. *LogStore* informed compaction

compaction. Based on these estimations, it schedules the next compaction such that the threshold is maintained.

LogStore maintains one histogram per-level in memory that tracks accesses to keys stored in the associated level. Histograms are maintained by an in-memory shared data structure that is updated on processing read/write requests. The data structure uses atomic counters for thread-safety, and min/max heaps to ensure fast access to statistics. Each level is associated with its own histogram instance, and each bucket in the histogram correspond to an SST. When a read operation for a record with key k is processed, the counter (of the bucket) associated with k's SST is incremented. *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. 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. 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. It is important to note that *LogStore* does not cache data on SSD - there is no duplication of data. *LogStore* provides a series of techniques to ensure the hottest data is migrated to SSD through active workload observance and online adaptation.

A. *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 5 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*.

B. *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 Section V-A 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 10.

$$C_{comp} = (T \quad 1)R_s + R_h + S\left(\frac{2T \quad 1}{W_s} + \frac{1}{W_h}\right) \tag{10}$$

Equation 10 can be decomposed into two parts. The first two terms account for the T required seeks to the beginning of each input SST, $T \quad 1$ 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 11. 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 11 is the *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 *LogStore* by tracking failed SSTs lookups at runtime.

$$C_{read} = 2NR_h \tag{11}$$

LogStore triggers a reverse compaction when the compaction cost from Equation 10 is less than the total accrued read cost for the SST in Equation 11. 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.

C. *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 10 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. [3], [2], [4]) 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 10 and 11 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.

D. Staging Compactions

The write-path optimization described in Section V-C 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 Section V-B: 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*.

VI. Evaluation

In this section, we evaluate *LogStore* under a variety of different workloads and system parameters. We implemented *LogStore* on top of *LevelDB* [3], 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, and because it (and its derivatives) serve as the storage engine for several distributed key-value stores (e.g., [16], [1])

In our evaluation, we consider workloads based on the popular Yahoo! Cloud Serving Benchmark (YCSB) [18]. The workload specifications are listed in Table I. We execute the experiments in four phases: cleaning, data loading, warming and the workload phase.

We run all experiments on Google Cloud Platform using N2 instance types. We configure each instance with 8GB of RAM,

Workload	Description
A	A balanced workload (i.e., 50% Read operations, and 50% Write operations).[2]
A-RMW	A balanced workload (i.e., 50% Read operations and 50% Read-Modify-Write operation which updates parts of the record.
C	A Read-only workload (i.e., 100% read operations) .
B-90	A variant of Workload-B from YCSB (Read-heavy). This workload contains 90% Read operations and 10% Write operations.
B-10	Another variate of Workload-B (Write-heavy). This workload contains 10% Read operations and 90% Write operations.
E	Performs scan operations of short ranges of records. The start of the range is based on a Zipfian distribution while the length of the range follows a uniform distribution $\in [1, 100]$. This workload contains 95% Scan operations and 5% Insert operations which insert new keys.
Write-Only	A workload containing Write-only operations.

TABLE I
Workload descriptions

Type	Read %	Read	Write
NVMe	100%	180K	N/A
NVMe	90%	150K	16.7K
NVMe	50%	64.2K	64.2K
NVMe	10%	10.5K	94.1K
NVMe	0%	N/A	99.9K
SSD	100%	15K	N/A
SSD	90%	13.6K	1.5K
SSD	50%	7.5K	7.5K
SSD	10%	1.5K	13.6K
SSD	0%	N/A	15K
HDD	100%	3K	N/A
HDD	90%	2.8K	0.3K
HDD	50%	2K	2K
HDD	10%	0.5K	4.9K
HDD	0%	N/A	6K

TABLE II
Measured Storage IOPS Limits on Google Cloud Platform

unless stated otherwise, and 4 vCPUs. Furthermore, instances are configured with a local NVMe storage of size 375GB, 1TB of SSD, and 4TB standard disk drive which correspond to the typical hard-disk. Table II shows the estimated performance of the storage virtual devices used in our experiments[3]. 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. The Zipfian request distribution has a configurable skew parameter, $s > 0$. The skew observed in the workload is directly proportional to s. In our experiments, we 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 multiple configurations of *LogStore* against *LevelDB* version 1.17. The all systems configurations are summarized in Table III. We note that *LDB-NVMe-$* is based on [33], which uses SSD as a cache layer to improve the read performance.

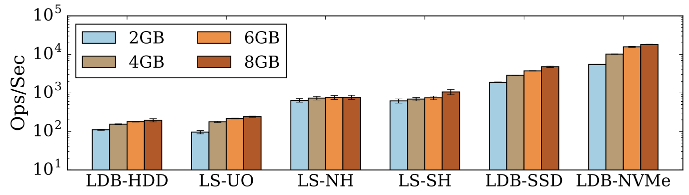

Fig. 6. Throughput for Workload-C with varying RAM sizes

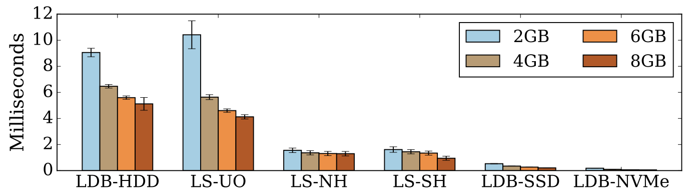

Fig. 7. Latency for Workload-C with varying RAM sizes

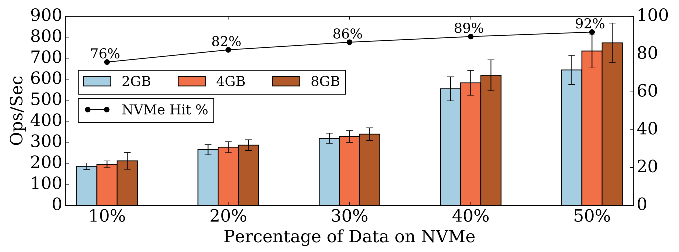

Fig. 8. Throughput of *LogStore* in Workload-C with varying SSD usage

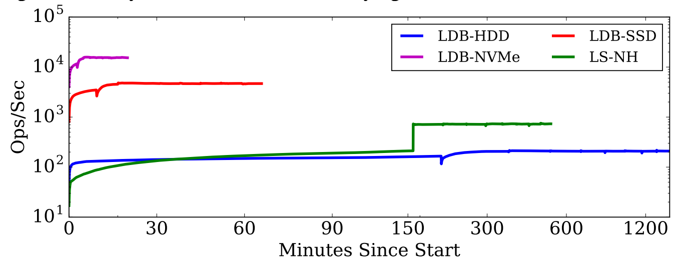

Fig. 9. Throughput of each system over time (Workload-C)

Name	$	L0 + L1	L2	L3+
LDB-NVMe	N/A	NVMe	NVMe	NVMe
LDB-SSD	N/A	SSD	SSD	SSD
LDB-HDD	N/A	HDD	HDD	HDD
LDB-NVMe-$	NVMe	HDD	HDD	HDD
LS-NH	N/A	NVMe	HDD	N/A
LS-NS	N/A	NVMe	SSD	N/A
LS-SH	N/A	SSD	HDD	N/A
LS-UO	N/A	NVMe	HDD	N/A

TABLE III
EVALUATED SYSTEMS AND STORAGE DEVICE CONFIGURATIONS

A. *Read-Only Performance*

In this section, we observe how each system behaves when we run YCSB's Wokload-C, and we vary the amount of memory available to the system (i.e., either 2GB, 4GB, 6GB, or 8GB). The keys are chosen from a Zipfian distribution. For these experiments, we benchmark *LDB-HDD*, *LDB-NVMe*, *LS-NH*, and *LS-UO*. *LS-UO* is uses the same configuration as *LS-NH* but all optimizations (i.e., informed compactions and reverse-compactions) are disabled. In particular, *LS-UO* uses an NVMe device to store SSTs for L0 and L1 but randomly selects which SSTs to migrate from NVMe to HDD. Using *LS-UO*, we demonstrate the effectiveness of our compaction techniques, and how these techniques contribute to the performance of *LogStore*.

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-NVMe*, but with a fraction of the requisite SSD capacity. Our expectations are validated in Figures 6 and 7, which show throughput and latency results, respectively. In the figures, we see that *LDB-NVMe* 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%), *LS-NH* outperforms *LDB-HDD* by 4-6 . This is because *LogStore* identifies the frequently accessed data, initially stored on the HDD in this experiment, and migrates this data to the NVMe. Additionally, we see that *LS-UO* is not even twice as fast as *LDB-HDD* because it is entirely unaware of key access patterns in the workload. The limited performance improvements *LS-UO* sees over *LDB-HDD* is entirely some operations go to the NVMe. *LS-UO* clearly illustrates that arbitrarily storing half your data on an SSD/NVMe (and not using the optimizations we offer in this work) yields suboptimal performance compared to *LS-NH*, which outperforms *LS-UO* by up to 6 .

The first two levels of *LS-NH* receive 92% of the accesses, with the remaining 8% going to the last level on the HDD. Even with this tremendous skew towards the NVMe, *LS-NH* and *LS-SH* remain bottlenecked by the HDD since the HDD used in this evaluation is 5-60 slower than SSD/NVMe (See Table II). This is expected based on the analysis we derived in Section IV-A. Note that *LS-UO*'s first two levels receive only 7%, which explains its low performance despite having NVMe because it does not the informed and reverse compaction techniques.

Moreover, as we increase the amount of RAM in the system, the throughput of *LevelDB* rises quicker than that of *LS-NH*. This is because the RAM is absorbing accesses that would normally be destined for the NVMe, while the HDD sees the same access frequency as before. This means that adding RAM to a hybrid configuration is not offering help because the HDD is the primary bottleneck. We investigate this peculiarity further in the following sections.

1) *Effect of SSD size on LogStore:*

In this section, we look at how *LS-NH* performs as we vary the size of the NVMe *LS-NH* is allowed to use. The intuition is that the performance would rise as the utilization of NVMe increases. 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 *LS-NH* while keeping both the total data size and amount of available memory constant.

The results, plotted in Figure 8, measures both the overall performance of *LS-NH* in each configuration and the hit rate on the NVMe, 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 NVMe is trending up, the efficiency-per-byte is decreasing, clearly indicating diminishing returns.

[3]Measured using *https://fio.readthedocs.io/*

System	Analytical	Experimental (ops/sec)	% Diff. (ops/sec)
LDB-HDD	200	$196 \pm 9{:}6$	+2.0%
LS-NH	822	$773 \pm 47{:}9$	+6.3%
LDB-SSD	$5; 000$	$4; 785 \pm 96{:}0$	+4.4%
LDB-NVMe	$20; 000$	$18; 029 \pm 186{:}8$	+11%

TABLE IV
ANALYTICAL AND EXPERIMENTAL THROUGHPUT FOR WORKLOAD-C (READ-ONLY)

However, because the NVMe 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 6 faster than *LDB-HDD*.

2) Adaptability of LogStore: Figure 9 shows the performance of *LDB-HDD*, *LDB-NVMe*, *LDB-SSD* and *LS-NH* 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. This experiment demonstrates how adaptive each system is as key-value access patterns evolve.

Unsurprisingly, *LDB-NVMe* and *LDB-SSD* are much better than *LDB-HDD* and *LS-NH* because they use fast storage devices. There is a very brief warm-up time where caches are filled, but after this period both *LDB-NVMe* and *LDB-SSD*provide a stable and fast throughput. Similarly in *LDB-HDD*, we see a stable throughput over the workload, but at a significantly lower throughput - roughly 208 ops/sec. The reason *LevelDB* offers such a stable throughput is because it does no data layout optimization during the workload. In contrast, *LS-NH* identifies the skewed read pattern immediately and begins optimizing the data layout by scheduling reverse compactions to migrate the hottest data to the NVMe. As *LS-NH* begins to fill its youngest levels with the hottest SSTs, the read throughput climbs steadily and eventually matches and exceeds the performance of *LDB-HDD* (See Figure 9, Minute 45 and beyond). When the NVMe has reached capacity storing 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.

3) Effectiveness of read cost-model: In this section, we use our model developed in Section IV-A to analytically predict the expected throughput of each system using the experiment configuration and setup used in Section VI-A. For the analytical model, we use 8GB of RAM for the system, Zipfian skew of $s = 0{:}9$, $R^z_{RAM} = 0{:}7$, and $R^z_{SSD=NVMe} = 0{:}227$. Additionally, we use random seek throughput values of 60, 1500, and 6000 seeks/sec for HDD, SSD, and NVMe, respectively. We summarize the results in Table IV. Notably, for Workload-C, the predicted throughput is within 11% of the observed values. For *LS-NH*, the difference is attributed to the imperfect nature of real-world caches. Our model expects only the hottest data to be stored in main memory, while in actuality there is a portion of RAM that stores cold keys from HDD. As such, the model, to a degree, overestimates the efficiency of the cache. We can solve this problem by disabling file system caching of SST data that resides on HDD. We intend to explore this optimization in future work.

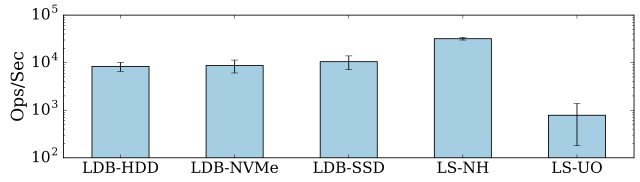

Fig. 10. Throughput for write-only workload

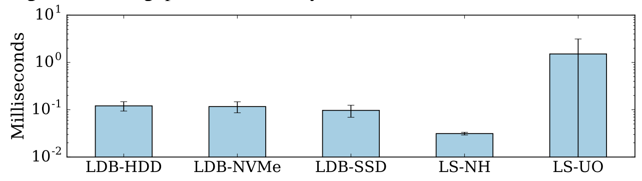

Fig. 11. Latency for write-only workload

The results of this section offer validation of our analytical cost model 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.

B. Write-Only Performance

We now look at how *LS-NH* 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.

In a write-heavy workload, we expect *LevelDB*-based systems and *LogStore*-based systems to offer very high write-throughput, but we want to know the impact of the optimizations used by *LS-NH* outlined in Section V-C. Hence we also include *LS-UO* that does not use deferred and staging compaction and rely on the same heuristics used by the original *LevelDB* implementation. The results of our write experiments in Figures 10 and 11 indeed show that *LS-NH* is able to outperform *LevelDB*-based systems by 3-$3{:}8$. The performance of *LevelDB*-based systems is lower than *LS-NH* is because they spends the majority of the 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 experiments, the client is inserting and updating data at a rate much higher than *LevelDB* compactions are able to execute. Hence, *LevelDB* throttles incoming writes to cope.

LS-NH is able to handle a much larger volume of writes before triggering any compaction, owing to the write-path optimization described in Section V-C. Many of the compactions that are triggered to purge data from NVMe are converted to staging compactions that operate on the NVMe alone. In a skewed write-heavy workload, many of the overlapping SSTs on the youngest levels contain overwritten key-value pairs. As a result, staging compactions quickly remove overwritten values, thereby reducing the size of the levels on the NVMe; staging compactions often obviate the need for a subsequent compaction to HDD. Additionally, since *LS-NH* only uses three levels whereas *LevelDB* uses seven, the degree of write amplification is much lower. However, reducing the number of levels used by *LevelDB* systems, is not enough to the improve performance as there are other contributing factors. We run experiments with *LDB-NVMe* with three levels instead of seven, and we observe that the throughput is reduced by 94% in our experiments. Deferring and staging compactions allow the SSD to absorb updates to the most frequently accessed keys, as the write cost model in Section IV-B predicts. *LevelDB*, on the other hand, will unnecessarily propagate older versions of the hottest keys through the older levels. Finally, it is important to note that the write-only experiments characterize the compaction process of these systems. As we can observe from Figure 10 and Figure 11, despite that *LDB-NVMe* storage device is faster than *LDB-SSD*, they have comparable performance which is because there is a large number of compaction tasks in both systems, and these tasks interfere with write operations. With respect to *LogStore*-based systems, the number of compactions performed by *LS-NH* is very small compared to *LS-UO*, and the performance to drop in *LS-UO* by nearly 98% compared to *LS-NH*. The variance in the performance results is also much higher in *LS-UO*, which is also attributed the compaction process.

1) Effectiveness of write cost-model: We now look at the accuracy of the write cost-model we presented in Section IV-B. We use sequential write throughput values of 150, 240, and 480 MB/s, for HDD, SSD, and NVMe, respectively. In addition, we assume the record size is 1KB.

Using Equation 7, the expected throughput of *LDB-HDD*, *LDB-SSD* and *LDB-NVMe* are 1,110, 1,776, and 3,552 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 $L \quad 1$ 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 $log_M(D) \quad 1$ times. If we use this observation, the model correctly predicts the throughput we observe in the experiments.

Using Equation 8 on *LogStore*, the model predicts a throughput of roughly 6,658 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.

C. Mixed Read-Write Performance

In this section, we evaluate how each system performs when we run four mixed read-write workloads from Table I based on YCSB [18]: Workload-A (balanced), Workload-A-RMW (balanced), Workload-B-90 (read-heavy), and Workload-B-10 (write-heavy). We configure all systems with 8GB of RAM and use a Zipfian key distribution with a skew parameter, $s = 0.9$ for both reads and writes. In this section, we include 3 configurations of *LevelDB* (*LDB-HDD*, *LDB-SSD*, and *LDB-NVMe*), and 3 configurations of *LogStore* (*LS-NH*, *LS-SH*, and *LS-NS*). The systems are summarized in Table III.

The results of the experiment are shown in Figures 12 and 13, with the former graphing average throughput and the latter graphing average latency. In the remaining of this section, we focus on *LS-NH* as the main point of comparison against *LevelDB*-based systems. However, it is worth noting that *LogStore* can take advantage of faster storage at the last level as in *LS-NS*. As shown in the figures, using regular SSD as the last level as in *LS-NS*, can improve the system's average throughput by up to 5 across various mixed workloads over *LS-NH*.

1) Performance under a Workload-B-10 (read-heavy): In the read-heavy experiment, we observe that *LS-NH* is more than 3.5 faster than *LDB-HDD* while only storing a fraction of all data on SSD. Interestingly, with the addition of just 10% writes to the workload, all three systems have a reduced throughput in comparison to the read-only workload as seen in Section VI-A. This occurs partly due to the nature of how LSM-trees function, but mostly due to the compaction process. First, the most frequently written keys will often end up in different SSTs through the Memtable flushing process. Compactions exacerbate this by merging data into new SSTs. This data movement of hot keys reduces the efficiency of any available caches. As described in Section II, many modern LSM-tree systems, *LevelDB* included, implement an optimization whereby frequently accessed SSTs that overlap in key ranges between levels are compacted into a single level. This optimization, which we term read-triggered compaction, exists to reduce the read latency for these frequently accessed keys by potentially having to perform fewer I/Os *LevelDB* executes read-triggered compactions very often in read-heavy workloads, and we see that this optimization does not work since they steal limited I/O capacity from the client. It is for this reason that *LDB-SSD* experiences a drop in throughput by 25% from the read-only case. Since compactions complete quicker on NVMe/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*.

LogStore-based systems do 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. In our experiments, on average *LS-NH* execute only 6% of the average number of compactions executed by *LDB-HDD*. 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 14, *LS-NH* 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

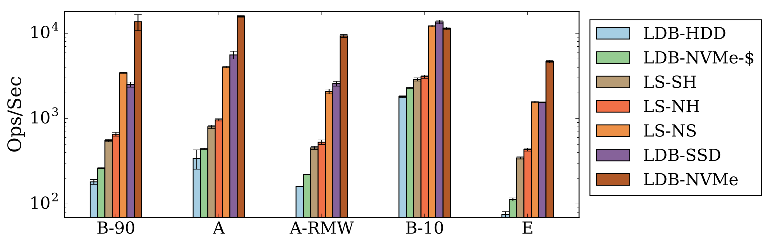

Fig. 12. Throughput for mixed read-write workloads

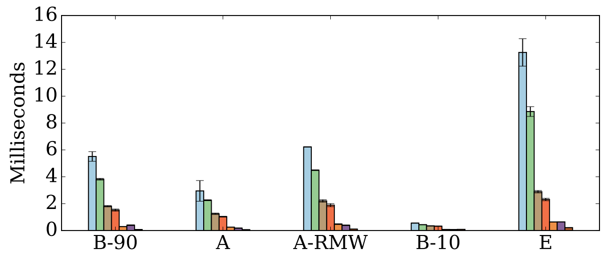

Fig. 13. Latency for mixed read-write workloads

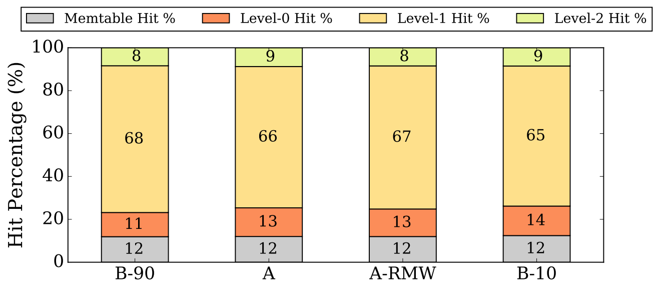

Fig. 14. Percentage of reads serviced by each level in *LS-NH*

from levels stored on the SSD, while the remaining 8% have to touch the HDD but only require one seek.

2) Performance under Workload-A (Balanced): In the balanced workload experiment using YCSB Workload-A, all systems have improved performance over the Workload-B-90. This is primarily because writes in any LSM-tree based system are fast and because *LevelDB* executes fewer read-triggered compactions with fewer reads. However, relatively, *LevelDB*-based systems execute significantly more compactions than *LogStore*-based systems. For example, *LDB-NVMe-$*, and *LDB-HDD* execute 5823 and 7042, respectively, while *LS-NH* executes 1202 compactions per workload run. This results in more device bandwidth availability for application traffic and less cache churn. As in the read-heavy case, *LS-NH* is able to utilize the NVMe by ensuring the majority of accesses (80% as seen in Figure 14) of read traffic is served from the NVMe. 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, *LS-NH* needs to compact some of the writes to the HDD to maintain space on NVMe. This compaction process, as in *LDB-HDD*, is very slow on HDD.

As in the original YCSB Workload-A, update operations are blind-writes, which updates a key with a random string of size 1024 bytes. We also run experiments to evaluate *LogStore* using Workload-A-RMW that performs Read-Modify-Write (RMW) operations instead blind-write operations. A RMW operation modifies part of the value (i.e., a field in a record). Because the operations are more intensive, all systems show reduced performance but the performance trends are similar to Workload-A. This similarity is because both *LevelDB* and *LogStore* do not perform in-place updates and the reduction comes from processing the additional read operation that precedes the update operation in each request.

3) Performance under Workload-B-10 (write-heavy): Again, this is primarily because the LSM-tree is well-suited for write-heavy workloads. In *LDB-HDD* and *LDB-SSD*, the reduction in read percentage over the balanced case results in fewer read-triggered compactions. Though delaying compactions can compromise read latency by having to perform most SST lookups on average, the effect is not pronounced since the workload makeup has fewer reads overall. In these experiments, *LDB-HDD* and *LDB-SSD* require seven SST probes on average to satisfy a read operation. Interestingly, since compactions run slower on HDD, these SST probes execute faster since the hot pages have a higher chance of being in cache. Finally, we observe that the NVMe cache offers very minimal benefit to a write-heavy workload in our implementation. Since the NVMe cache is append-only, it suffers from cache churn for write-heavy workloads.

In *LS-NH*, the majority of compactions that execute are staging compactions on the NVMe 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 NVMe, control the degree of overlap between SSTs on NVMe 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. *LS-NH* is able to defer compaction to the point where data reclamation is required to free space on NVMe. 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 NVMe does mean that *LS-NH* probes more SSTs on average than *LevelDB*, but these probes are always performed on the SSD. From Figure 14, we still see that *LogStore* is able to ensure 80% of read accesses go to the NVMe (with 12% being served from memory).

Moreover, we run experiments using Workload-E. This workload performs more reads per-key (50 on overage) than Workload-B because the requested key constitute the start of the range to be scanned. With *LDB-HDD*, the cost of scans are amplified when accessing ranges on HDD. In contrast, because *LS-NH* keeps the hottest records on NVMe and the high chance that scanned ranges overlap due to the Zipfian distribution, *LS-NH* improves upon *LDB-HDD* by nearly 6 . On one hand, using faster storage for L2 (*LS-NS*) improves the average throughput significantly by 3:6 over *LS-NH*. On

Workload	System	Analytical (ops/sec)	Experimental (ops/sec)	% Diff.
B-90	*LDB-HDD*	180	181 ± 11:9	-1.0%
	LDB-SSD	2; 962	2; 507 ± 174:5	+18.2%
	LDB-NVMe	15; 586	13; 641 ± 2902:4	+14.3%
	LS-NH	745	656 ± 34:1	+13.5%
A	*LDB-HDD*	312	342:9 ± 87:6	-9.9%
	LDB-SSD	6; 148	5; 575 ± 577:5	+10.3%
	LDB-NVMe	14; 940	15; 764 ± 340:9	-5.5%
	LS-NH	1; 215	967 ± 30:1	+24.3%
B-10	*LDB-HDD*	1; 441	1; 809 ± 43:0	-25.6%
	LDB-SSD	10; 441	13; 591 ± 577:5	-26.8%
	LDB-NVMe	16; 459	11; 412 ± 340:9	+44.2%
	LS-NH	3; 753	3; 095 ± 30:1	+20.6%

TABLE V
ANALYTICAL AND EXPERIMENTAL THROUGHPUT FOR READ-WRITE WORKLOAD FOR *LDB-HDD*, *LDB-SSD*, *LDB-NVMe* AND *LS-NH*

the other hand, using slower SSD storage in place of NVMe (*LS-SH*) reduces the throughput by only 20%.

4) Effectiveness of read-write cost-model: In this section we evaluate the effectiveness of the read-write cost model presented in Section IV-C. As before, we assume the same device and system characteristics as in Section VI-A3 and Section VI-B1. Table V 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. Overall, the model predictions are mostly within 25% of the observed values.

In a read-heavy workload, our model predicts a throughput for *LDB-SSD* that is about 18% higher. The reason for this is due to an optimization within *LevelDB* that seeks to compact frequently accessed key ranges that overlap between levels. Though compactions are performed in the background, they directly interfere with application I/O and pollute the file system cache. This problem less pronounced in *LDB-HDD* since compactions run much slower, meaning that read operations for hot keys have a higher probability of being served directly from memory. The predicted and experimental results converge as the write ratio increases because the optimization is triggered less often.

We also observe that in the write-heavy model, the prediction for *LDB-NVMe* is higher by 44%. Our model does not capture that throttling mechanism used by *LevelDB* to slow write operations. In our experiments, we observed that *LDB-NVMe* slows down significantly more write operations than *LDB-SSD*. This phenomena occurs because compactions are fast and thus it keeps the system at the same threshold that triggers the slowdown. The model's prediction for *LS-NH* is accurate across the workloads with the exception of the balanced workload, which is 25% higher than the observed figure. 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 NVMe 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.

VII. CONCLUSIONS

This paper presented *LogStore*, a new key-value store architecture that is workload-aware, dynamic, and designed to operate in a hybrid storage environment. Unlike previous works that use the SSDs only to extend in-memory caches or buffer pools, *LogStore* uses SSDs to complement HDDs; data is either stored on the SSD or on the HDD, never both. Being an embedded storage system optimized for multicore hybrid machines, *LogStore* can be incorporated as a building block for distributed key-value stores.

LogStore implements informed, reverse, deferred, and staging compactions that are each driven by low-overhead statistics and a cost-benefit run-time analysis. *LogStore* ensures that the hottest data is migrated to the SSD, performs the majority of I/O-intensive preparatory work on the SSD, and guarantees at most on seek on HDD. This work also presented an analytical cost model to predict the throughput of a generic log-structured hybrid storage system. We used insights provided through the model in the design of *LogStore* and we evaluated the model's accuracy through experimentation.

Using workloads based on YCSB, we evaluate multiple configurations based on *LogStore*, and demonstrate that they achieve 6 better throughput than *LevelDB* running on HDD in read-only workloads, and up to 3:6 better throughput than *LevelDB* running on SSD/NVMe when executing a write-only workload. Across mixed read-write workloads, *LogStore* has 1:5-5:7 better throughput than *LevelDB* on HDD.

REFERENCES

[1] CockroachDB, the scalable, survivable, strongly consistent, sql database. https://www.cockroachlabs.com.

[2] HyperLevelDB. http://hyperdex.org/performance/leveldb/.

[3] LevelDB, a fast key-value storage library by google. https://code.google.com/p/leveldb/.

[4] RocksDB, an embeddable persistent key-value store for fast storage by facebook. http://rocksdb.org/.

[5] M. Y. Ahmad and B. Kemme. Compaction management in distributed key-value datastores. *Proc. VLDB Endow.*, 8(8):850–861, 2015.

[6] M. Athanassoulis, M. S. Kester, L. M. Maas, R. Stoica, S. Idreos, A. Ailamaki, and M. Callaghan. Designing access methods: The RUM conjecture. In E. Pitoura, S. Maabout, G. Koutrika, A. Marian, L. Tanca, I. Manolescu, and K. Stefanidis, editors, *Proceedings of the 19th International Conference on Extending Database Technology, EDBT 2016, Bordeaux, France, March 15-16, 2016, Bordeaux, France, March 15-16, 2016*, pages 461–466. OpenProceedings.org, 2016.

[7] M. A. Bender, M. Farach-Colton, J. T. Fineman, Y. R. Fogel, B. C. Kuszmaul, and J. Nelson. Cache-oblivious streaming b-trees. In *Proceedings of the nineteenth annual ACM symposium on Parallel algorithms and architectures*, SPAA '07, pages 81–92, New York, NY, USA, 2007. ACM.

[8] B. Bhattacharjee, M. Canim, C. A. Lang, G. A. Mihaila, and K. A. Ross. Storage class memory aware data management. *IEEE Data Eng. Bull.*, 33(4):35–40, 2010.

[9] B. Bhattacharjee, K. A. Ross, C. A. Lang, G. A. Mihaila, and M. Banikazemi. Enhancing recovery using an SSD buffer pool extension. In *DaMoN*, pages 10–16, 2011.

[10] L. Bouganim, B. ??r J?nsson, and P. Bonnet. uFLIP: Understanding Flash IO Patterns. In *CIDR '09: Fourth Biennial Conference on Innovative Data Systems Research*.

[11] M. Canim, B. Bhattacharjee, G. A. Mihaila, C. A. Lang, and K. A. Ross. An object placement advisor for DB2 using solid state storage. *PVLDB*, 2(2):1318–1329, 2009.

[12] M. Canim, G. A. Mihaila, B. Bhattacharjee, K. A. Ross, and C. A. Lang. SSD bufferpool extensions for database systems. *PVLDB*, 3(2):1435–1446, 2010.

[13] H. H. W. Chan, Y. Li, P. P. C. Lee, and Y. Xu. Hashkv: Enabling efficient updates in KV storage via hashing. In H. S. Gunawi and B. Reed, editors, *2018 USENIX Annual Technical Conference, USENIX ATC 2018, Boston, MA, USA, July 11-13, 2018*, pages 1007–1019. USENIX Association, 2018.

[14] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. In *OSDI*, pages 205–218, 2006.

[15] F. Chen, D. A. Koufaty, and X. Zhang. Understanding intrinsic characteristics and system implications of flash memory based solid state drives. In J. R. Douceur, A. G. Greenberg, T. Bonald, and J. Nieh, editors, *Proceedings of the Eleventh International Joint Conference on Measurement and Modeling of Computer Systems, SIGMETRICS/Performance 2009, Seattle, WA, USA, June 15-19, 2009*, pages 181–192. ACM, 2009.

[16] J. Chen, C. Douglas, M. Mutsuzaki, P. Quaid, R. Ramakrishnan, S. Rao, and R. Sears. Walnut: A unified cloud object store. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, pages 743–754, New York, NY, USA, 2012. ACM.

[17] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H. Jacobsen, N. Puz, D. Weaver, and R. Yerneni. PNUTS: yahoo!'s hosted data serving platform. 2008.

[18] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking Cloud Serving Systems with YCSB. In *SoCC '10*, pages 143–154, 2010.

[19] M. Cornwell. Anatomy of a Solid-State Drive. *Communications of the ACM*, 2012.

[20] N. Dayan, M. Athanassoulis, and S. Idreos. Monkey: Optimal navigable key-value store. In S. Salihoglu, W. Zhou, R. Chirkova, J. Yang, and D. Suciu, editors, *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*, pages 79–94. ACM, 2017.

[21] J. Do, D. Zhang, J. M. Patel, D. J. DeWitt, J. F. Naughton, and A. Halverson. Turbocharging DBMS buffer pool using SSDs. In *SIGMOD*, SIGMOD '11, pages 1113–1124, New York, NY, USA, 2011. ACM.

[22] C. Gaspar. Deploying nagios in a large enterprise environment. In *LISA*, 2007.

[23] G. Graefe. The Five-Minute Rule 20 Years Later: and How Flash Memory Changes the Rules . *Communications of the ACM*, 2009.

[24] J. Gray, P. Sundaresan, S. Englert, K. Baclawski, and P. J. Weinberger. Quickly Generating Billion-Record Synthetic Databases. In *SIGMOD '94*, pages 243–252, 1994.

[25] S. Hu, W. Liu, T. Rabl, S. Huang, Y. Liang, Z. Xiao, H.-A. Jacobsen, X. Pei, and J. Wang. DualTable: A Hybrid Storage Model for Update Optimization in Hive. In *Proceedings of the 31st International Conference on Data Engineering*, 2015.

[26] A. Lakshman and P. Malik. Cassandra: a decentralized structured storage system. *SIGOPS Operating Systems Review*, 44(2):35–40, 2010.

[27] Y. Li, B. He, J. Yang, Q. Luo, and K. Yi. Tree indexing on solid state drives. *Proc. VLDB Endow.*, 3(1):1195–1206, 2010.

[28] L. Lu, T. S. Pillai, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Wisckey: Separating keys from values in ssd-conscious storage. In A. D. Brown and F. I. Popovici, editors, *14th USENIX Conference on File and Storage Technologies, FAST 2016, Santa Clara, CA, USA, February 22-25, 2016*, pages 133–148. USENIX Association, 2016.

[29] C. Luo and M. J. Carey. Lsm-based storage techniques: a survey. *VLDB J.*, 29(1):393–418, 2020.

[30] J. Manyika, M. Chui, B. Brown, J. Bughin, R. Dobbs, C. Roxburgh, and A. H. Byers. Big data: The Next Frontier for Innovation, Competition, and Productivity. Technical report, McKinsey Global Institute, 2011. http://www.mckinsey.com/insights/mgi/research/technology_and_innovation/big_data_the_next_frontier_for_innovation.

[31] L. Marmol, S. Sundararaman, N. Talagala, R. Rangaswami, S. Devendrappa, B. Ramsundar, and S. Ganesan. Nvmkv: A scalable and lightweight flash aware key-value store. In *HotStorage*, 2014.

[32] M. L. Massie, B. N. Chun, and D. E. Culler. The Ganglia Distributed Monitoring System: Design, Implementation, and Experience. *Parallel Computing*, 30(7):817–840, 2004.

[33] P. Menon, T. Rabl, M. Sadoghi, and H.-A. Jacobsen. CaSSanDra: An SSD Boosted Key-Value Store. In *ICDE*, 2014.

[34] P. E. O'Neil, E. Cheng, D. Gawlick, and E. J. O'Neil. The log-structured merge-tree (LSM-Tree). *Acta Inf.*, 33(4):351–385, 1996.

[35] J. Ouyang, S. Lin, S. Jiang, Z. Hou, Y. Wang, and Y. Wang. Sdf: Software-defined flash for web-scale internet storage systems. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, pages 471–484, New York, NY, USA, 2014. ACM.

[36] T. Qadah, S. Gupta, and M. Sadoghi. Q-store: Distributed, multi-partition transactions via queue-oriented execution and communication. In A. Bonifati, Y. Zhou, M. A. V. Salles, A. Böhm, D. Olteanu, G. H. L. Fletcher, A. Khan, and B. Yang, editors, *Proceedings of the 23rd International Conference on Extending Database Technology, EDBT 2020, Copenhagen, Denmark, March 30 - April 02, 2020*, pages 73–84. OpenProceedings.org, 2020.

[37] T. M. Qadah and M. Sadoghi. Quecc: A queue-oriented, control-free concurrency architecture. In P. Ferreira and L. Shrira, editors, *Proceedings of the 19th International Middleware Conference, Middleware 2018, Rennes, France, December 10-14, 2018*, pages 13–25. ACM, 2018.

[38] A. Qin, M. Xiao, J. Ma, D. Tan, R. Lee, and X. Zhang. Directload: A fast web-scale index system across large regional centers. In *35th IEEE International Conference on Data Engineering, ICDE 2019, Macao, China, April 8-11, 2019*, pages 1790–1801. IEEE, 2019.

[39] D. Qiu, K. Zhang, and H.-A. Jacobsen. Smart Phone Application for Connected Vehicles and Smart Transportation . In *Middleware Posters*, 2013.

[40] T. Rabl, M. Sadoghi, H.-A. Jacobsen, S. Gómez-Villamor, V. Muntés-Mulero, and S. Mankowskii. Solving Big Data Challenges for Enterprise Application Performance Management. *PVLDB*, 2012.

[41] T. Rabl, M. Sadoghi, K. Zhang, and H.-A. Jacobsen. MADES - A Multi-Layered, Adaptive, Distributed Event Store. In *Proceedings of the 7th ACM International Conference on Distributed Event-Based Systems*, 2013.

[42] P. Raju, R. Kadekodi, V. Chidambaram, and I. Abraham. Pebblesdb: Building key-value stores using fragmented log-structured merge trees. In *Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai, China, October 28-31, 2017*, pages 497–514. ACM, 2017.

[43] M. Rosenblum and J. K. Ousterhout. The design and implementation of a log-structured file system. *ACM Trans. Comput. Syst.*, 10(1):26–52, Feb. 1992.

[44] M. Sadoghi, S. Bhattacherjee, B. Bhattacharjee, and M. Canim. L-store: A real-time OLTP and OLAP system. In M. H. Böhlen, R. Pichler, N. May, E. Rahm, S. Wu, and K. Hose, editors, *Proceedings of the 21st International Conference on Extending Database Technology, EDBT 2018, Vienna, Austria, March 26-29, 2018*, pages 540–551. OpenProceedings.org, 2018.

[45] M. Sadoghi and S. Blanas. *Transaction Processing on Modern Hardware*. Synthesis Lectures on Data Management. Morgan & Claypool Publishers, 2019.

[46] R. Sears and R. Ramakrishnan. bLSM: a general purpose log structured merge tree. In *SIGMOD Conference*, pages 217–228, 2012.

[47] A. Soga, C. Sun, and K. Takeuchi. NAND Flash Aware Data Management System for High-Speed SSDs by Garbage Collection Overhead Suppression. In *6th International Memory Workshop*, 2014.

[48] D. Teng, L. Guo, R. Lee, F. Chen, S. Ma, Y. Zhang, and X. Zhang. Lsbm-tree: Re-enabling buffer caching in data management for mixed reads and writes. In K. Lee and L. Liu, editors, *37th IEEE International Conference on Distributed Computing Systems, ICDCS 2017, Atlanta, GA, USA, June 5-8, 2017*, pages 68–79. IEEE Computer Society, 2017.

[49] R. Thonangi, S. Babu, and J. Yang. A practical concurrent index for solid-state drives. In X. Chen, G. Lebanon, H. Wang, and M. J. Zaki, editors, *21st ACM International Conference on Information and Knowledge Management, CIKM'12, Maui, HI, USA, October 29 - November 02, 2012*, pages 1332–1341. ACM, 2012.

[50] P. Wang, G. Sun, S. Jiang, J. Ouyang, S. Lin, C. Zhang, and J. Cong. An efficient design and implementation of lsm-tree based key-value store on open-channel ssd. In *EuroSys*, 2014.

[51] L. Yang, H. Wu, T. Zhang, X. Cheng, F. Li, L. Zou, Y. Wang, R. Chen, J. Wang, and G. Huang. Leaper: A learned prefetcher for cache invalidation in lsm-tree based storage engines. *Proc. VLDB Endow.*, 13(11):1976–1989, 2020.

[52] H. Yoon, J. Yang, S. F. Kristjansson, S. E. Sigurdarson, Y. Vigfusson, and A. Gavrilovska. Mutant: Balancing storage cost and latency in lsm-tree data stores. In *Proceedings of the ACM Symposium on Cloud Computing, SoCC 2018, Carlsbad, CA, USA, October 11-13, 2018*, pages 162–173. ACM, 2018.

Prashanth Menon is third year Ph.D. student in the Database Group at Carnegie Mellon University. He received his M.A.Sc. from the University of Toronto in 2015, where is research was on designing key-value stores for multi-tiered storage systems. His research interests are broadly across databases, new storage technology, and distributed systems. More recently, he is interested in improving the performance of hybrid transaction processing (HTAP) databases using a blend of vectorized execution and just-in-time compilation.

Thamir M. Qadah is a Ph.D. candidate in the School of Electrical and Computer Engineering at Purdue University, West Lafayette, Indiana, and a lecturer at Umm Al-Qura University, Makkah, Saudi Arabia. He is a member of the ExpoLab research group, and is co-advised by Prof. Mohammad Sadoghi and Prof. Arif Ghafoor. His research interests are in the design and implementation of secure, dependable, and high-performance software systems that exploit modern hardware technologies. His research on queue-oriented transaction processing is recognized by the Best Paper Award in Middleware'18.

Tilmann Rabl is a visiting professor at the Database Systems and Information Management (DIMA) group at the University of Technology Berlin and deputy director of the Intelligent Analytics for Massive Data department at the German Research Institute for Artificial Intelligence. His research is in the area of big data and streaming systems and benchmarking. He has published more than 70 papers. At DIMA he is research director and technical coordinator of the Berlin Big Data Center (BBDC). He received his PhD at the University of Passau. In his PhD thesis, he invented the Parallel Data Generation Framework, for which he received the Transaction Performance Processing Councils Technical Contribution Award. He is a professional affiliate of the TPC and co-founder and chair of the SPEC Research working group on big data. He is also CEO and cofounder of the startup bankmark.

Mohammad Sadoghi is an Assistant Professor in the Computer Science Department at the University of California, Davis. Formerly, he was an Assistant Professor at Purdue University and Research Staff Member at IBM T.J. Watson Research Center. He received his Ph.D. from the University of Toronto in 2013. He leads the ExpoLab research group with the aim to pioneer a distributed ledger that unifies secure transactional and real-time analytical processing (L-Store), all centered around a democratic and decentralized computational model (ResilientDB). He has co-founded a blockchain company called Moka Blox LLC, the ResilientDB spinoff. He has over 80 publications in leading database conferences/journals and 34 filed U.S. patents. He served as the Area Editor for Transaction Processing in Encyclopedia of Big Data Technologies by Springer. He has co-authored the book "Transaction Processing on Modern Hardware", Morgan & Claypool Synthesis Lectures on Data Management, and currently co-authoring a book entitled "Fault-tolerant Distributed Transactions on Blockchain" also as part of Morgan & Claypool series.

Hans-Arno Jacobsen is pioneering research that lies at the interface between computer science, computer engineering and information systems. He holds numerous patents and was involved in important industrial developments with partners like Bell Canada, Computer Associates, IBM, Yahoo! and Sun Microsystems. His principal areas of research include the design and the development of middleware systems, event processing, service computing and applications in enterprise data processing. His applied research is focused on ICT for energy management and energy efficiency. He also explores the integration of modern hardware components, such as FPGAs (Field Programmable Gate Arrays), into middleware architectures. After studying and completing his doctorate in Germany, France and the USA, Prof. Jacobsen engaged in post-doctoral research at INRIA near Paris before moving to the University of Toronto in 2001. There, he worked as a professor in the Department of Electrical and Computer Engineering and the Department of Computer Science. After being awarded the prestigious Alexander von Humboldt-Professorship, he joined TUM in October 2011.