PrismDB: Read-aware Log-structured Merge Trees for Heterogeneous Storage

Ashwini Raina
Princeton University

Asaf Cidon
Columbia University

Kyle Jamieson
Princeton University

Michael J. Freedman
Princeton University

Abstract

In recent years, emerging hardware storage technologies have focused on divergent goals: better performance or lower cost-per-bit of storage. Correspondingly, data systems that employ these new technologies are optimized either to be fast (but expensive) or cheap (but slow). We take a different approach: by combining a heterogeneous set of fast and low-cost storage technologies within the same system, we can achieve a Pareto-efficient balance between performance and cost-per-bit.

This paper presents the design and implementation of PrismDB, a novel log-structured merge tree key-value store that exploits the full spectrum of heterogeneous storage technologies simultaneously. We introduce the notion of "read-awareness" to log-structured merge trees, which allows hot objects to be pinned to faster storage, achieving better tiering and hot-cold separation of objects. Compared to Mutant, a prior key-value store for heterogeneous storage, and RocksDB, PrismDB can achieve up to 5.8× and 5.1× higher throughput (respectively), reduce read tail latency by 10× and 10.7×, and reduce update latency by 10.3× and 9×.

1 Introduction

Several new hardware storage technologies have emerged over the past few years, expressing the competing goals of improving the performance and reducing the cost of storage. On one side, high performance non-volatile memory (NVM) technologies, such as 3D XPoint [4, 65] and Z-NAND [8], provide latencies of 100s of nanoseconds to 10 μs, making them suitable as DRAM replacement for different workloads [23, 24, 58]. On the other end of the spectrum, cheap and dense storage such as QLC NAND [42, 44] enables applications to store vast amounts of data on flash at a low cost-per-bit. Yet with this lower cost, QLC has a higher latency and is significantly less reliable than less dense flash technology.

Table 1 compares the wide range of cost and performance across three representative storage technologies, showing the tradeoffs between their reliability (P/E cycle lifetime), normalized cost, and read and write latency.

In short, each of these emerging hardware technologies introduces its own set of benefits, trade-offs, and limitations related to performance, cost, and endurance. As an example, Table 1 illustrates the read latency of Optane (NVM), TLC, and QLC, using the fio [3] benchmark. There is a roughly 15× performance difference between Optane and QLC on random reads, and sequential reads show a similar trend (not shown here). Yet this performance comes at a steep cost: Optane costs nearly 10× per GB compared to QLC. Endurance also varies widely: dense flash technologies such as QLC NAND impose a limitation on how many times they can be written to before they get worn out [43].

It is hard for system architects to reason about these unique performance, cost, and endurance characteristics when developing software data systems, and many studies have shown that simply running existing software systems on new hardware storage technologies often leads to poor results [22–24, 40]. Therefore, significant recent effort has sought to build new databases, file systems, and other software storage systems that are architected specifically for these new technologies [22, 23, 33, 38, 40, 54, 71].

However, we argue that these new architectures do not go far enough in rethinking the use of emerging hardware technology, as they continue to take the legacy perspective of the storage substrate as a homogeneous and monolithic layer. They choose one point in the design space: very fast but expensive [23, 38, 71] (e.g., by using 3D XPoint), or very cheap but slow [54] (e.g., by using QLC NAND).

Conversely, this paper explores how a key-value store can simultaneously leverage multiple new storage technologies to realize more optimal trade-offs between performance, endurance and cost. In particular, we investigate combining heterogeneous storage technologies within a Log-structured Merge Tree [50] (LSM), a widely-used data structure that powers many modern flash-based databases and key-value stores (e.g., Google's BigTable [15] and LevelDB, Apache Cassandra [37], Facebook's RocksDB [21] and MySQL storage backend [41], MongoDB [45], and PebbleDB [51]).

At a high-level, LSM trees maintain high write rates by

	NVM	TLC	QLC
Lifetime (P/E cycles)	18,000	540	200
Cost ($/GB)	$1.3	$0.4	$0.1
Avg Read Latency (4KB)	26μs	195μs	391μs
Avg Write Latency (64MB)	121μs	216μs	456μs

Table 1: Comparison of representative lifetime, cost, and read and write latency of new storage technologies: Optane SSD (NVM), and two generations of flash (TLC and QLC NAND, which have three and four bits per cell, respectively). The cost numbers are based on the cost of consumer devices as of July 25, 2019 on Amazon (Intel's Optane SSD 900P, 750 Series and 760P), and the lifetime numbers are based on publicly available information [20, 31, 64]. Latency numbers are computed using Fio [3] benchmark.

buffering multiple updates in memory, then sorting the updates' keys before writing the new keys and values as a block to disk to the first level of the LSM tree. Multiple blocks from one level are then merged into lower levels, ensuring that blocks at each lower level are sorted and disjoint. As these sorted blocks can be written as large sequential writes, they are thought to be a good fit for flash-based storage, which requires large contiguous writes for maintaining performance and endurance.

Existing LSM tree-based databases assume all levels are stored on a homogeneous storage medium. However, the access patterns to these levels are not homogeneous. Objects stored in the higher levels of the LSM are read and updated much more often than objects stored at the bottom layers. Therefore, the upper levels of the LSM tree (e.g., L0, L1, L2) would benefit from using a high performance, high endurance (and more expensive) storage medium such as NVM. On the other hand, the lower levels (e.g., L3, L4, or the last couple of layers of the LSM tree), which store 90% or more of the data, can use a much cheaper form of storage such as QLC NAND. In addition, since they are updated much less frequently, these lower levels can meet the lower endurance requirements of cheaper flash storage (i.e., fewer P/E cycles). To summarize, this heterogeneous LSM tree design would allow an LSM tree to enjoy the performance benefit of using fast storage (e.g., Optane) to speed up frequently read objects from high levels, while maintaining a low cost-per-bit, since over 90% of data would be stored in the lower, less frequently accessed levels on cheaper storage (e.g., QLC).

Yet, we have found that this observation cannot be naively applied to LSM trees. We show that a straightforward heterogeneous implementation, where different LSM levels are mapped to different storage technologies, performs only marginally better than an LSM tree fully mapped on the slowest storage. We make the observation that LSM trees by default are "write-aware", i.e., the key layout in the tree is dictated by the order of the writes. This fundamentally restricts the ability of LSM implementations to fully exploit the performance benefits of heterogeneous storage.

In this paper, we introduce the notion of "read-awareness" to LSM trees. In ***read-aware LSM trees***, the key layout within the tree is influenced by both the write order as well as the read order of keys. Existing LSM implementations always compact all the keys down from upper to lower levels. We propose a new compaction algorithm called *pinned compaction*, in which keys that are read more often are retained in the same level. Unlike traditional compaction, our compaction algorithm also allows for keys to rise up the tree levels towards faster storage, while maintaining consistency.

We present the design and implementation of PrismDB, a key-value store built on top of RocksDB that implements read-awareness via pinned compactions. In order to add read awareness to the LSM tree, PrismDB needs to decide which objects to pin during compaction time. To this end, it uses a lightweight object popularity mechanism based on the CLOCK algorithm. In order to convert the CLOCK value to a pinning policy, PrismDB dynamically estimates the distribution of CLOCK values across different keys, and uses that distribution to determine which objects to pin during compaction.

Our paper makes the following contributions:

1. We introduce the first holistic evaluation of LSM trees on heterogeneous storage technologies taking cost, performance, as well as endurance into account.
2. We identify limitations of deploying existing LSM tree key-value stores on heterogeneous storage.
3. We propose a new LSM tree variant—the *read-aware LSM*—and a new compaction algorithm that unlocks the full benefit of heterogeneous storage.
4. We compare PrismDB to Mutant, a prior key-value store for heterogeneous storage, and RocksDB running on heterogeneous storage, and show that PrismDB achieves up to 5.8× and 5.1× higher throughput (respectively), reduces read tail latency by 10× and 10.7×, and reduces update latency by 10.3× and 9×.

2 Background

We provide a brief background on new storage technologies, as well as on log-structured merge (LSM) trees.

2.1 Trends in Storage

In recent years, storage devices have evolved in two orthogonal directions: faster (and more expensive) non-volatile memory and cheaper (and slower) dense flash. New fast and persistent memory technologies, such as 3D XPoint [4, 44] and Z-NAND [8], which we refer to collectively throughout the paper as Non-Volatile Memory (NVM), provide low random

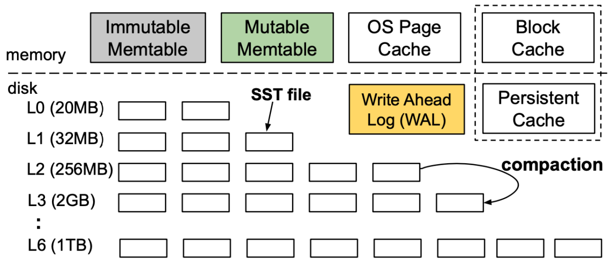


Figure 1: Elements of a Log Structured Merge Tree.

read and write latencies of 10s μs or less. Due to their low latency and high throughput, and since they are cheaper per bit than DRAM, they have the potential to serve as a replacement for DRAM in datacenter use cases [23, 24, 65, 66].

On the other end of the spectrum, flash technology has become ever more dense and cheap. Flash manufacturers have been able to pack more bits in each device, both by stacking cells vertically (3D flash), and by packing more bits per memory cell. However, making devices denser also causes their latency to increase and makes them less reliable [29, 44, 49, 54, 64]. The latest QLC technology, which packs 4 bits per memory cell, can only tolerate 100–200 write cycles before it becomes unusable [49, 55, 64]. For this reason, the main current use case for QLC is for applications that issue a small number of writes [54, 56].

2.2 Log-Structured Merge (LSM) Trees

In LSMs (Figure 1), data is written first into DRAM, where it is stored in Memtables, which are in-memory data structures based on Skiplists [7]. Once a Memtable becomes full, its data is written into a Sorted String Table (SST). An SST is a file on disk that contains sorted variable-sized key-value pairs that are partitioned into a sequence of data blocks. In addition to data blocks, the SST stores meta blocks, which contain Bloom filters and the metadata of the file (e.g., data size, index size, number of entries).

SST files are stored in flash in multiple levels, called L0, L1, and so forth. L0 contains files that were recently flushed from the Memtable. After L0, each level is comprised of SSTs that are disjoint (in keyspace) from other SSTs on the same level, and the LSM maintains a sort order over each level's SSTs. To find a key, a binary search is first done over the start key of all SST files to identify which file contains the key. Only then, an additional binary search is done inside the file (using its index block) to locate the exact position of the key.

Each level also has a target size that specifies the volume of data that should be stored in the level, typically with an exponentially increasing capacity (e.g., in Figure 1, the level size multiplier is 8). Once a level reaches its target size, a compaction is triggered. During compaction, at least one SST file is picked and merged with its overlapping key range in the next level.

The motivation for the original LSM Tree [50] was to design an indexing data structure for higher write throughput, targeting for storage devices that require large contiguous writes to exhibit good performance (initially magnetic disks to avoid seeks and later flash for full-page-sized writes). An LSM tree inherently trades off read performance for most efficient writes. Due to its relatively efficient handling of writes, even with the shift in workloads becoming more read-heavy—which has led to the introduction of some auxiliary data structures such as SST fence pointers, bloom filters, and read caches to improve read performance [5, 7, 21]—the core structure of the LSM tree has stayed largely unchanged over the past 25 years.

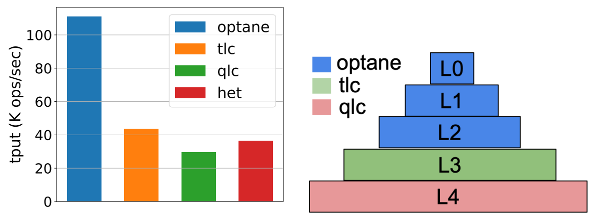
In short, the order of write operations dictates the organization of keys at the tree levels, and so we refer to LSM trees as "write aware". In particular, read operations do not change the key layout in the LSM tree, i.e., an LSM tree does change the placement of particular objects based on the frequency that they are read. As we shall see, however, this distinction becomes crucial for achieving better read performance when employing LSM over heterogeneous storage.

Read Optimizations in LSM Trees. As mentioned, LSM trees have introduced several auxiliary data structures to improve read performance. Recall that a key can be located in any level in the LSM tree (and in fact, in multiple levels simultaneously in the case of updates or deletions). To avoid reading a potential on-disk SSTable per level, LSM trees such as RocksDB [7] employ in-memory Bloom filters [1] per SST to check whether the key possibly exists in the SST before reading it from disk.

LSM trees additionally cache LSM blocks in memory for accelerating reads. The *block cache* caches entire SST file blocks using the least-recently-used (LRU) algorithm. In addition, the LSM trees may rely on the operating system's page cache, which also caches at the block level. These block and page caches operate at the block level because storage is typically block addressable and reads happen at a page granularity (commonly 4 KB). Since key-value objects can be as small as tens of bytes [14], caching at an individual object granularity would mean discarding the rest of the page data, which would reduce I/O utilization. This creates a mismatch between the granularity of caching (4 KB blocks) and object sizes (10s-100s of bytes), which we analyze in §3.3.

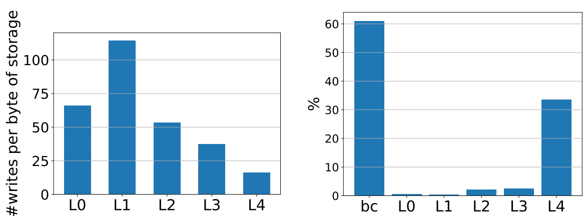
3 LSM Performance on Modern Storage

In this section, we present an evaluation of LSM tree performance on both homogeneous and heterogeneous storage, and highlight the shortcomings of existing LSM tree design when attempting to utilize heterogeneous storage. Throughout the



(a) (b)

Figure 2: RocksDB throughput utilizing homogeneous and heterogeneous storage (Fig. 2a). Throughput measured using the YCSB benchmark, with a 95:5 read-update ratio and Zipfian 0.99 distribution of requests to keys. Heterogenous storage is configured as Optane for L0-L2, TLC for L3, and QLC for L4 (Fig 2b).



(a) (b)

Figure 3: Comparing the distribution of writes and reads across LSM tree levels (L0-L4) and block cache (bc), using YCSB to generate workloads.

paper, we use RocksDB, an open-source key-value store used by Facebook [14] and others [46], as our baseline LSM tree implementation.

3.1 Homogeneous Storage

We first evaluate RocksDB on a homogeneous, single-disk setup using the YCSB benchmark. We compare its performance on three homogeneous configurations: Optane, TLC, and QLC. Figure 2a shows the throughput and read latency of running RocksDB on these varying technologies. RocksDB running on Optane has roughly 2.7× and 4× higher throughput compared to TLC and QLC, respectively.

We make the observation that while an LSM tree is a single logical data structure, each one of its levels has very different performance and endurance requirements, as depicted in Figure 12. Objects stored in the upper levels of the LSM are read and updated much more often than objects stored at the bottom layers. Therefore, the higher levels would benefit from using a high performance and high endurance (and more expensive) storage medium such as NVM. The lower levels on the other hand, which store 90% or more of the data, can use a much cheaper form of storage such as QLC NAND. In addition, since they are updated much less frequently, they may meet the endurance requirements of cheaper flash storage. On the other hand, QLC has much slower read latency, so query latency can suffer if a high fraction of reads are served from L4 (as in Figure 3b).

Benchmark	Memtable	L0	L1	L2	L3	L4
YCSB	25%	3%	2%	5%	16%	49%

Table 2: Distribution of point read queries across LSM levels with block cache disabled. Block cache is only effective at caching objects present in upper levels (Figure 3b).

3.2 Heterogeneous Storage

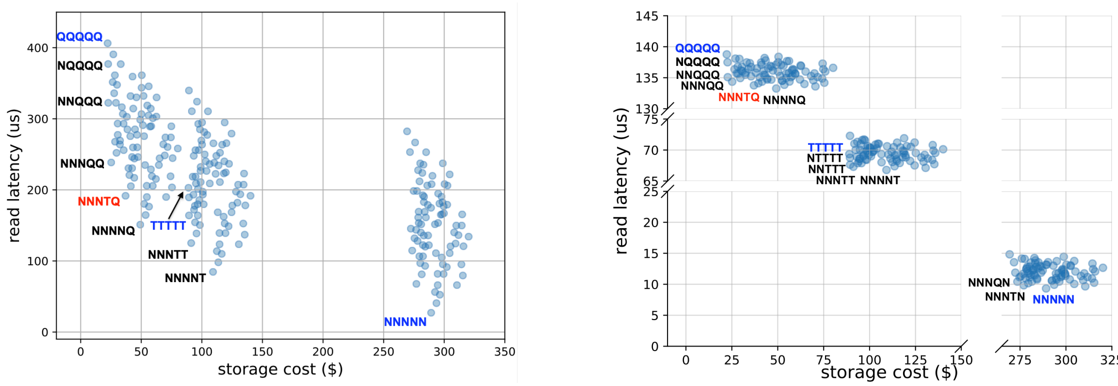
We next examine how well the LSM tree performs on a heterogeneous setup. The LSM tree running on heterogeneous storage is referred to as *LSM-het* for the remainder of the paper. We use a 5-level LSM tree, where each level is mapped to a different storage tier. In Figure 4, we simulate the cost vs. performance trade off of different configurations, by simulating the read latency and cost of each level, when they are placed on different storage technologies.

In order to simulate endurance constraints, we assume the storage devices need to last for at least 3 years, which is a typical lifetime for storage devices. Therefore, if a particular storage technology has been written to too much to last 3 years, we add additional spare storage capacity to that technology until we reach enough storage to achieve the 3 year limit. This method follows the same principle used by enterprise flash devices, which are provisioned with spare capacity to achieve higher endurance for write-heavy workloads.

The ideal points on the Pareto frontier of the curve, as expected, are those where upper levels use the same or a faster storage technology than the lower levels. In the paper, we examine one of the points on the Pareto frontier, $NNNTQ$, (depicted in Figure 2b), which maps LSM levels L0-L2 to NVM, L3 to TLC, and L4 to QLC, which contains the key space of the entire database.

We modified RocksDB to map different levels onto different storage types, and again evaluate its performance with the YCSB [18] benchmark. We observed that even though LSM-het has faster storage components, it performs only marginally better than an LSM that is fully mapped to the slowest storage QLC (Figure 2a), and it has about 3× worse throughput compared to the LSM running fully on Optane. In short: a heterogeneous RocksDB configuration pays the extra cost of faster storage but does not achieve any significant performance boost.

Why the lack of performance improvement? Table 2 shows that roughly 65% of read queries for specific keys are served from either L3 and L4 levels, which are mapped on slower storage. This completely diminishes the impact that the faster



(a) **Without DRAM Caching** (b) **With DRAM Caching**

Figure 4: Simulation of the trade-off between average read latency (in μs) and storage cost (cents per GB) when storing data in an LSM tree, with a minimum storage lifetime constraint of 3 years. Percentages of reads and writes across LSM levels are based on RocksDB production data [21] for a total database size of 223 GB. The result of each possible hardware configuration is shown in the figure. The hardware configuration is represented by a five-tuple, corresponding to each level of the LSM tree, where N is Optane SSD (NVM), T is TLC SSD, and Q is QLC SSD. Blue represents homogeneous configurations, and the red configuration ($NNNTQ$) is used as the default one in this paper.

storage tier would have on read latency. In particular, optimizing LSM trees for heterogeneous storage can be thought of a multi-tiered storage problem, where the first tier is DRAM, the second is NVM, and the remaining are one or more types of flash devices (or even HDDs). Yet we cannot rely on LSM's traditional design to organize the tree into an efficient multi-tiered storage system, since it does not try to keep popular objects in higher levels of the LSM tree. In §4, we show how we can transform the existing LSM data structure to account for the read performance of different storage technologies.

3.3 Caching Efficiency

Single and multi-disk experiments show that it is critical to consider LSM level dynamics while deploying LSM trees over heterogeneous storage. We now turn to another crucial aspect: the caching dynamics of LSM tree systems.

As discussed in §2.2, LSM trees cache at a block granularity (4 KB or more) to maximize I/O utilization. However, typical key-value pairs in production workloads are on the order of tens to hundreds of bytes [14, 21, 47]. Since the layout of keys within blocks is only dependent on writes, *SST blocks contain objects with different read popularity*.

Therefore, LSM tree block-level caching turns out to be less effective, as a significant percentage of the objects in the block cache may not be popular. This general limitation of LSM trees affects both the homogeneous and heterogeneous storage setups and is a key aspect of our design.

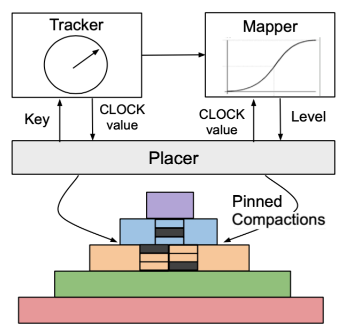


Figure 5: PrismDB system diagram.

4 PrismDB Design

In this section, we introduce PrismDB, a read-aware LSM tree. We first discuss design challenges, and then detail PrismDB's main design components.

PrismDB is comprised of three main components. The *tracker* is responsible for tracking which objects are popular. The *mapper* keeps track of the distribution of object popularity across the entire LSM tree and translates that distribution to an actionable algorithm for selecting which objects to pin, so that the *placer* can ensure that popular objects remain on higher levels of the LSM tree, where they will be stored on faster storage when employing heterogeneous hardware.

4.1 Tracker: Lightweight Tracking of Keys

The first component in PrismDB, the tracker, is responsible for tracking which objects are frequently read at a low overhead. There is a large body of work on how to track and estimate object popularity [12, 53, 57, 61, 62]. However, many of the existing mechanisms require a relatively large amount of data per object, in order to track various access statistics (number of prior accesses, frequency, relationships with other objects, etc.) [12, 53]. Given that key-value objects are often small (e.g., less than 1 KB [14, 16, 47], we need to limit the amount of metadata we use for tracking purposes per object, yet maintain an accurate prediction on how "hot" the object is, or how likely it is to be read or updated in the near future. In addition, LSM tree implementations support a high number of concurrent write and read operations to the database [5, 7]. This requires a high performing popularity tracking mechanism that can track millions of small objects at high throughput.

CLOCK [19] is a well-known classical approach that approximates least recently used (LRU) while offering better space efficiency and concurrency [22, 26]. Using a single CLOCK bit only captures recency, and employing multiple clock bits can be used to also track frequency. This makes CLOCK an attractive option for estimating the relative popularity of different keys, in order to group more popular keys into higher levels of the LSM tree.

PrismDB's tracker implements the multi-bit CLOCK algorithm for lightweight object tracking. The tracker uses a concurrent hash map that maps keys to their CLOCK bits. Each LSM read requires the tracker to update the CLOCK bits of the object that was read. Since setting the CLOCK bits is on the critical path of reads, the set operation needs to be lightweight. This leads us to make several performance optimizations in the tracker.

First, in order to save space, the tracker does not store CLOCK bits of all key-value pairs in the system, only the most recently read ones. Second, the tracker is optimized for concurrent key insertions and evictions. Traditional CLOCK implementations use a doubly linked list or a circular buffer that contains the CLOCK bits along with a hash map that contains the mapping from key to CLOCK bits, which requires that insertion and eviction operations be serialized. In contrast, PrismDB's tracker does not keep a separate buffer from the hash table. Therefore, it does not require any extra synchronization for those operations, and relies on the synchronization provided by the concurrent hash map. Third, the tracker conducts eviction offline, in a background process, so that when a key-value read causes a new key to be inserted into the tracker, that doesn't trigger eviction of an older key in the critical path.

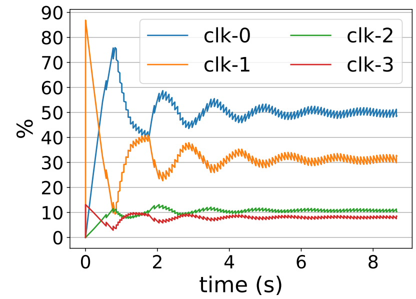


Figure 6: Clock value distributions of PrismDB using the YCSB working with a Zipfian 0.99 distribution.

4.2 Mapper: Enforcing the Pinning Threshold

Ideally, we would like to be able to set a "pinning" threshold on the popular keys; i.e., at each pass of the compactor, PrismDB should pin some percent (e.g., 10%) of the most popular objects that are being tracked by the tracker to the same level.

However, the actual relative popularity of the object depends on the CLOCK bit distribution. For example, if PrismDB wants to enforce a pinning threshold of 10%, and exactly 10% of keys have all their CLOCK bits set to 3 (using a two bit CLOCK), then PrismDB should pin all the items with a CLOCK value of 3. However, if 50% of the keys have a CLOCK value of 3, then PrismDB should not pin all items with the CLOCK value 3, otherwise it will significantly exceed the desired pinning threshold. To illustrate, Figure 6 shows an example of the CLOCK distribution over YCSB, using the default Zipfian distribution (0.99).

To this end, the mapper is responsible for keeping track of the CLOCK value distribution, and uses the distribution to enforce the pinning threshold. In order to maintain the CLOCK value distributions, the mapper maintains the aggregate number of keys that have a particular CLOCK value in an array. The array gets updated during insertion and eviction. During eviction, the tracker keeps count of the CLOCK values of all the items it decremented and the item it evicted and updates the mapper with the new distribution. It also updates the mapper when a new item is inserted into the hash map.

Pinning Threshold Algorithm. In order to enforce the pinning threshold, the mapper uses the following algorithm, which is best illustrated with an example. Suppose the CLOCK distribution is similar to the one depicted in Figure 6 (after the distribution stabilizes), where the percentage of keys with a CLOCK value of 3 is about 10%, the percentage of those with a value of 2 is 10%, the ones with a value of 3 is 30%, and the remaining 50% have a CLOCK value of 0. Suppose the desired pinning threshold is 15%. If the placer encounters an item with a CLOCK value of 3, it will always pin it. If it encounters an item with a CLOCK value of 2, it will flip a coin whether to keep it or not (in this example, with weight 0.5). If it encounters an item with CLOCK value of 1 or 0, or an item that is not currently being tracked (recall the tracker does not track all items in the database), that item will be compacted down.

To summarize, the mapper satisfies the pinning threshold using the highest-ranked CLOCK items by descending rank, and if need be, randomly samples objects that belong to the lowest CLOCK value that is needed to satisfy the threshold.

Note that as Figure 6 shows, it takes the CLOCK value distribution a few seconds to converge. The distribution initially fluctuates because there are a relatively small number of keys in the tracker, so each eviction cycle of the CLOCK hand changes the distribution. Thus, by default, PrismDB starts pinning objects only after the tracker is full.

4.3 Placer: Pinning Keys to Levels

The third component in our system, the *placer*, is responsible for pinning popular keys to LSM levels residing on different storage mediums.

Ideally, given our tiered storage design, we would like to keep all hot keys at the top of the LSM tree. However, we need to take into account LSM tree level sizing. Write amplification in LSM trees is minimized when the ratio between the sizes of each subsequent level is fixed across all levels, namely, that level L_i is k-times larger than the level L_{i-1}, where $k = 10$ typically [21,50]. This imposes a size constraint on each level, and means that we cannot indiscriminately pin all the hot objects to the top levels that reside on fast storage. Deviating from the LSM tree sizing rule can increase the overall write amplification and reduce overall performance, which we also confirmed experimentally.

Another design goal for the placer is that it must not violate the consistency guarantees of the LSM tree. LSM trees typically maintain consistency by always storing newer versions of objects in upper levels and older versions in the lower levels. Naively pinning hot keys to top levels can break the consistency guarantee of LSM trees. (We discuss this further in §4.4.)

The placer also needs to be lightweight, since it is competing with reads, writes, and background compaction jobs for resources. It should avoid adding any locks on the database since reads and writes are being served concurrently. The timing of when to trigger the pinning process is also important. Ideally, it should be done during periods of reduced database activity to avoid resource contention.

Pinned Compactions. To achieve the above stated goals, we combine the pinning logic with the compaction process of LSM. We introduce a new compaction algorithm called *pinned compaction*. Compaction by default moves the keys towards the bottom levels of the tree. Compaction does a merge sort on the SST files between two adjacent levels. In the process, it reads every single key from the SST files and writes them in the sorted order into the output level. We extend the new compaction process to not just retaining the popular keys, but also "pull up" popular keys stuck in the lower level, possibly on slower storage. We call this process, not found in traditional LSM trees, "up-compaction."

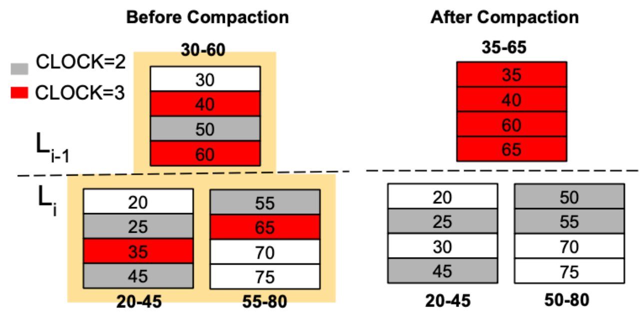


Figure 7: Example of pinned compaction

Figure 7 shows an example of how pinned compactions work on files under compaction. Keys shown in red and gray are present in the tracker; keys in white are not in tracker. Keys 40, 60, 35, and 64 (shown in red) are keys with clock value 3. Keys in gray have clock value 2. Assume that the pinning threshold pins all keys with clock value 3. During the merge process, instead of compacting everything down, keys 40 and 60 are retained in L_{i-1}. Also, popular keys, 35 and 65, from the lower level L_i are compacted up and stored in L_i.

Choosing Which SST File to Compact. When a compaction job is triggered, it first chooses a level and then an SST file in that level for compaction. With traditional LSM trees, picking the candidate SST file is based on some system level objective like reclaiming storage space, reducing write amplification, and so forth.

For PrismDB, we introduce a new SST file selection criteria, which better aligns with the high-level goal of pinning keys to appropriate levels. When the SST file is created, a popularity score is assigned to it, based on the CLOCK values of objects present in that file. Objects not present in the tracker get a default score of -1. Since by default CLOCK values range between [0,3], using them directly is problematic, since the difference between a highly popular key (CLOCK value 3) and an unpopular key (CLOCK value -1) is just 4, i.e., it only take 3 un-popular keys to negate the contribution of a highly popular key. We devise a workaround to this problem by using a weight parameter n to boost the contributions of popular keys towards the SST popularity score. The SST file popularity score is computed as follows:

$$score = \Sigma(object_i\, CLOCK\, value)^n$$

where i is the i^{th} key in the SST file. In our experiments, we use $n = 3$.

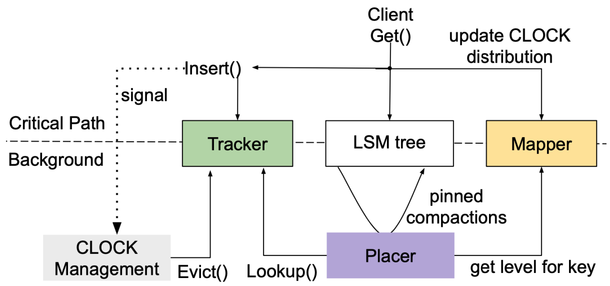


Figure 8: PrismDB component interaction

During compaction, each SST file in a level is sorted in the ascending order of popularity score, and the file with the lowest score is selected for compaction.

4.4 Maintaining Consistency

LSM-backed databases like RocksDB ensure that database clients will always read the latest version of the object committed to that database. LSMs ensure this behavior by versioning all writes (inserts, updates, or deletes) to a key, where newer versions of the key reside on upper levels of the LSM tree and older versions towards the tree's bottom. In this regard, when a read query traverses the tree from memtables, to L0, and then on downwards, the first version of the key it encounters is the latest (including a "tombstone" record for deletes). Even during compactions, this guarantee is maintained.

PrismDB's read-aware LSM tree maintains the same consistency guarantees. If a pinned compaction retains a set of objects in the same level, they will always be the more recent versions of objects than present in the lower levels. Even when objects rise up a level during a pinned compaction, they don't break the LSM guarantee: if the same key is encountered when merging SST files from levels L_{i-1} and L_i, the merge process preserves the (newer) key from level L_{i-1}. In doing so, read-aware LSM trees continue to ensure the ordering of key versions down the tree.

5 Implementation

We implemented PrismDB[1] on top of RocksDB v6.2.0, which is written in C++. The tracker is built on the concurrent hash map implementation from Intel's TBB library [30].

Intel's TBB concurrent hash map internally implements a hash table with chaining. Chained hashing offers constant time insert operations, however lookup and delete operations may be slower. In our system, inserts to the hash table happen on the read critical path and lookups and deletes happen in the background. Additionally, the tracker's workload is insert heavy; the ratio of inserts to lookup is 18:1, which makes hash chaining a reasonable choice for this workload.

The hash map index is the object key and each index stores a 1 byte value. The top two bits are used for storing the CLOCK value, while the bottom six bits store the last six bits of the hash of the key version. Multiple versions of the same client key can exist in the LSM. Only the latest version is useful for pinning objects. The tracker ideally should cache the most recent key version. LSM versions can be up to 8 bytes in size. Storing six bits of the hash value of the version provides sufficient resolution to distinguish between versions.

Figure 8 shows the interaction between PrismDB's different design components. When a database client calls the `get` API, the key is inserted into the tracker. Typically an insert operation also invokes an eviction. However, since the tracker `insert` is on the critical path of reads for PrismDB, we defer the evictions to a background thread. If the key is not present in tracker, `insert` inserts the key with an initial CLOCK value of 1. Setting the initial value to 3 can result in keys that are accessed only once to stay in tracker for a long time, since the CLOCK value needs to get decremented down to 0 before the key is evicted. For a key that is already present in tracker, if its version matches the tracker's version, then the CLOCK value is set to max value i.e. 3; otherwise it is treated as a new key and the CLOCK value is initialized to 1.

A background eviction thread implements the CLOCK hand movement using an iterator over the hash table. Note that new inserts into tracker can happen concurrently while the background iterator is running, by relying on TBB's concurrent hash map guarantee [32] that as long as erase operations are serialized, an iterator running in one thread does not conflict with insert operations happening in other threads. The limitation of running concurrently is that the iterator may either miss visiting some items or visit them more than once within a full iteration. This makes our implementation an approximation of the CLOCK algorithm. In the context of our system, such approximate behavior does not affect system performance. Finally, the CLOCK value is stored in the hash table as an atomic variable, thus `lookup` does not need to be serialized with eviction.

Each client `get` call updates the CLOCK value distribution in the mapper. Mapper is implemented as an array of four atomic integers; each keeps track of the number of keys with a particular CLOCK value.While running pinned compactions, placer calls the `lookup` API to check if the key is present in tracker. If so, it queries mapper to determine whether to pin the key to the current level.

6 Evaluation

We evaluate PrismDB by answering the following questions:

1. How does PrismDB on heterogeneous storage compare to the more common deployment of RocksDB on homo-

[1]GitHub link elided for anonymity purposes.

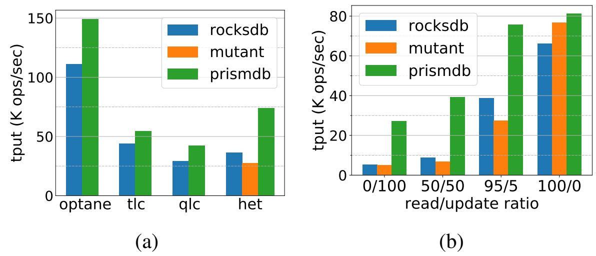


Figure 9: Throughput comparison between PrismDB, Mutant and RocksDB under YCSB zipfian-0.99 distribution.

Configuration	QQQQQ	NNNTQ	TTTTT	NNNNN
Storage Cost	$22	$37	$89	$289

Table 3: Estimates of cost in homogeneous configurations in comparison to our default heterogeneous configuration for a 223 GB database on a 5-level LSM tree, using the same notations and parameters from Figure 4.

geneous storage? What are the cost, performance and endurance trade-offs between the two?

2. What is the gap between PrismDB, RocksDB and Mutant running on a heterogeneous storage configuration?
3. Does the hot-cold object separation in PrismDB result in better block cache hit rate?
4. What is the impact of pinned compactions on read and write amplification for different LSM tree levels?
5. How effective is PrismDB in placing the right objects on different levels and how does pinning affect performance?

Configuration: We performed our experiments on a 32-core, 64 GB RAM machine running Ubuntu 18.04.2 LTS 32. Three heterogeneous storage devices were locally attached to this machine, all running over PCIe3.1: an Optane SSD (Intel 900p), which uses 3D XPoint, a TLC SSD (Intel 760p), and a QLC SSD (Intel 660p).

Baselines: We compare PrismDB against two baselines: a) RocksDB and b) Mutant. Mutant is a storage layer for LSM backed key-value stores that assigns SST files to heterogeneous storage based on the access frequency of SST files. It uses a background process called migration to move files between storage types. We implemented Mutant on top of RocksDB. We set Mutant's SST file cooling coefficient α to 0.999 and optimization epoch to 1 second. We did not implement Mutant's "migration resistance" optimization as it trades off storage space for lesser number of migrations. To do a fair comparison, we want the storage sizes to be fixed. For all systems, pending compaction byte limit was set to 0 so that LSM levels strictly adhere to their storage allocations.

Workload: We use the popular open source benchmark YCSB [18] to compare the performance across the PrismDB, RocksDB and Mutant. All experiments are run using 8 concurrent database clients, unless otherwise specified. By default, we use YCSB with 95% reads and 5% with a Zipfian parameter of 0.99. We load each one of the databases with 8 million keys and the benchmarks generate a total of 50 million requests. Unless otherwise specified, all the databases we compare in this section use a 1:10 ratio between DRAM and storage, where 20% of the DRAM is dedicated to block cache (a common production configuration [21]). In the heterogeneous configurations, the storage is divided roughly in a ratio of 1:9:90 between NVM, TLC, and QLC respectively, in the "NNNTQ" configuration as shown in Figure 2b. By default, PrismDB sets the tracker size to 10% of database key space and uses a pinning threshold of 10%.

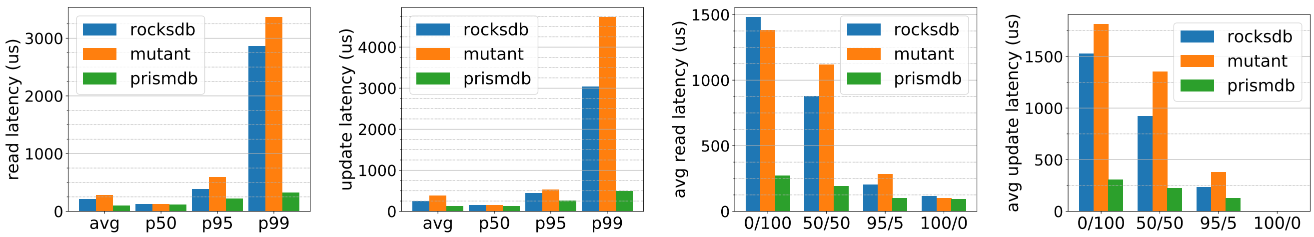
6.1 Homogeneous vs. Heterogeneous

Figure 9a compares the throughput of PrismDB with RocksDB and Mutant under four different configurations: three homogeneous configurations (NVM, TLC, and QLC) and one heterogeneous configuration, where we combine all three technologies. We make a few observations. First, as expected, NVM provides higher throughput than TLC, which is better than QLC. When we use vanilla RocksDB in the heterogeneous storage configuration, since its LSM tree is not read-aware, it provides only a small performance benefit over the pure QLC setup.

Second, PrismDB's heterogeneous configuration is able to outperform homogeneous configurations of TLC and QLC. PrismDB with heterogeneous storage exhibits 70% higher throughput than homogeneous TLC, which is the standard flash-based LSM tree setup used today. In Table 3 we compare the storage cost of all four configurations, using the same methodology we employed for the simulation in §3.2. Based on our simulation results, our heterogeneous configuration is almost 2.4× cheaper than a pure TLC setup, since about 90% of PrismDB's storage (the lowest level) is stored on QLC. To summarize, we believe there are many datacenter operators that would benefit from the cost-performance trade off offered by PrismDB's heterogeneous storage configuration.

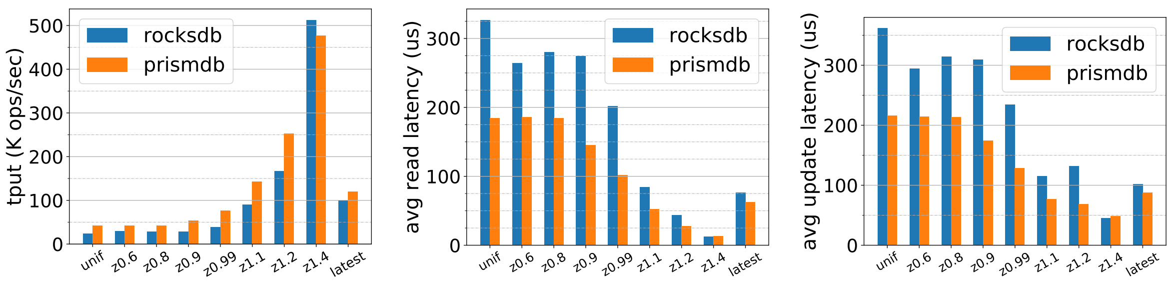
6.2 Heterogeneous Storage: PrismDB vs. RocksDB and Mutant

Figure 9b compares the throughput of the PrismDB to RocksDB and Mutant under the heterogeneous setup with different read/write ratios and shows that PrismDB significantly outperforms the other systems under any read-write ratio. Interestingly, PrismDB's benefit is lowest under a 100% read workload. The reason for this is that object pinning occurs during compaction, and a 100% read workload does not generate any compactions, which leads to suboptimal placement. Due to PrismDB's design, even a small amount of writes improves PrismDB's read performance.



(a) (b) (c) (d)

Figure 10: Average read and update latencies. Figures 10a and 10b use YCSB 95/5.



(a) (b) (c)

Figure 11: Performance with different YCSB distributions: "z" signifies Zipfian with its corresponding parameter, "latest" is the similar to Zipfian 0.99, where newly inserted objects are the most popular ones [18].

Figures 10a and 10b present the read and update latencies of PrismDB compared to RocksDB and Mutant. PrismDB improves the average, 95^{th} percentile (p95), and 99^{th} percentile (p99) read (update) latencies of RocksDB by 2.1× (1.9×), 1.7× (1.7×) and 8.8× (6.2×) respectively. However, it only improves the median read and update latency by 8% and 11%. The reason for this is that the median read and update are likelier to be cached both by RocksDB and by PrismDB in DRAM or in the NVM device. Therefore, unsurprisingly, PrismDB's performance improvements stem primarily from queries that would hit lower storage tiers under RocksDB, and are likelier to hit faster storage tiers with PrismDB.

The reason PrismDB is able to significantly outperform Mutant, is because Mutant does not take LSM tree level read and write dynamics into account [72]. It simply tries to migrate data on an SST file granularity based on access frequency. Therefore, it does not create hot-cold separation within blocks. In addition, its migrations induce I/O and require entire files to be locked, both of which degrade throughput and cause latency to spike, even when compared to RocksDB.

We also present the average latencies under varying read and update ratios in Figures 10c and 10d. Similar to the throughput results, PrismDB's read latency actually benefits from some level of writes to generate compaction.

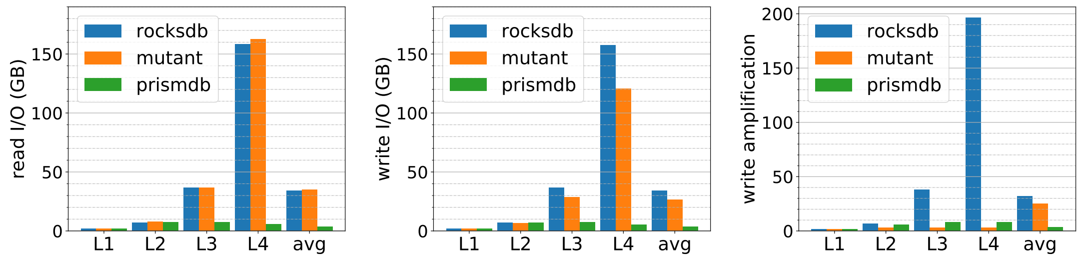
We compare PrismDB to RocksDB under different YCSB distributions in Figures 11a, 11b and 11c. PrismDB outperforms RocksDB in all distributions, except for Zipfian with a parameter of 1.4 (and above). The reason for this is that the higher the Zipfian parameter, the more skewed the workload. Once the workload becomes very skewed, almost all of its working set fits in RocksDB's block cache in DRAM. At that point, since PrismDB has a slightly higher overhead for point queries in DRAM due to its tracker, which needs to be updated at every read operation, RocksDB would slightly outperform PrismDB.

Config	RocksDB	Mutant	PrismDB	Overall Improvement	Data Block Improvement
Optane	54.1%	n/a	79.0%	1.45×	2.28×
TLC	60.1%	n/a	78.8%	1.31×	2.00×
QLC	49.7%	n/a	78.5%	1.58×	2.67×
Het	49.9%	49.6%	78.6%	1.57×	2.46×

Table 4: DRAM hit rate improvement over RocksDB. SST data, index and filter block accesses are counted in hit rate calculation.

6.3 Impact of Hot-Cold Separation

Since PrismDB pins hot objects to their levels, it naturally creates SST blocks that contain blocks with a similar level of



(a) (b) (c)

Figure 12: Comparing I/O usage of PrismDB to Mutant and RocksDB. PrismDB significantly reduces the amount of compaction I/O and as a result the write amplification.

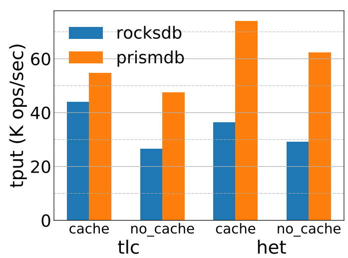


Figure 13: Throughput with DRAM caching disabled.

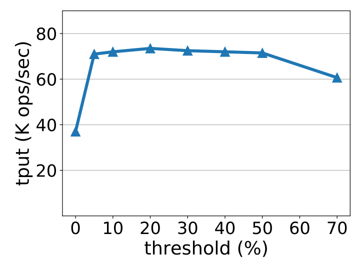


Figure 14: Effect of pinning threshold on throughput

read popularity. Since LSM trees cache data at the block level, this leads to a higher percentage of popular keys being read into DRAM, and therefore to higher hit rates. Hot-cold separation is the primary reason PrismDB outperforms RocksDB even under homogeneous setups, as shown in Figure 9a.

To demonstrate for the effect of PrismDB's hot-cold separation, Table 4 measures the hit rate of the block cache for data, index and filter blocks. We also show individual improvement of data block hit rates[2]. The table demonstrates that PrismDB significantly increases the data block hit rate for both homogeneous and heterogeneous configurations.

6.4 Impact on Read and Write Amplification

We analyze the impact of PrismDB on I/O usage in Figure 12. The subfigures show that PrismDB significantly reduces the amount of read and write I/O. In particular, in the YCSB trace, PrismDB conducts only 1135 total compactions, compared to RocksDB, which does 2602 compactions.

There are two primary reason for the reduced I/O. The first is due to PrismDB's improved hot-cold separation, which results in a higher DRAM block hit rate as we showed above. Therefore, reads as well as updates are more likely to be cached in DRAM, and less likely to be read and written from disk. The second reason is that by pinning more popular objects in upper levels of the LSM tree, and by prioritizing the compaction of SST files with fewer popular objects (as described in §4.3), PrismDB reduces read amplification. The reason for this is that when looking up an object in storage, LSM trees sequentially look up the object in level 0, then level 1, etc. Therefore, since PrismDB keeps more popular objects in upper levels, it reduces overall read amplification.

To demonstrate the interplay between these two behaviors, we compare PrismDB to RocksDB with and without DRAM caching. Figure 13 shows that, even when caching is disabled and even under a homogeneous setup (TLC), PrismDB has higher throughput than RocksDB, since it reduces read amplification by pinning popular objects in upper levels, which require a fewer number of reads to fetch.

6.5 Effectiveness of Pinned Compaction

We analyze the affect of setting different pinning thresholds on the overall throughput of PrismDB in Figure 14. The figure shows that setting the threshold too low or too high is detrimental to performance. If the threshold is set too low, very few objects will be pinned and PrismDB will converge to a similar throughput of RocksDB. If it is set too high, the compactor will be less effective in compacting SST files (since many objects will be pinned), leading to higher I/O consumption and

[2]Note that as mentioned in §2.2, LSM trees make extensive use of OS the page cache, for which it is difficult to measure the exact data block hit rate.

reduced throughput[3] We also microbenchmarked the tracker insertion, and found it took less than $2\mu s$ to insert a new key to the hash map.

7 Related Work

We split related work into three parts: recent prior work that incorporates emerging storage technologies into databases, key-value caches and file systems.

Databases and Key-value Stores. KVell [38], SLM-DB [34] and uDepot [35] are databases specifically designed for NVM. They observe that unlike flash, NVM supports fast writes to small amounts of data. Therefore, they can avoid costly compactions and do not need to sort keys on disk. However, since these databases are optimized for NVM, they would not perform well under a hetrogeneous storage setting that includes traditional flash, which requires large contiguous writes for performance and endurance.

Arulraj *et al*. have explored in several papers how to incorporate databases to use byte-addressable NVM [9, 10], and in particular show how to re-architect the WAL to more efficiently take advantage of NVM. Several other recent papers further explore logging and recovery for byte-addressable NVM [48, 59]. In the same vein, HiKV [69] and work by van Renen *et al*. [58] propose key-value stores that improves the latency of requests using persistent memory.

In addition, there is a very large body of work on improving the performance of LSM trees. For example, TRIAD [11] employs several strategies to reduce write amplification, such as keeping frequently updated objects in memory, and delaying compaction until the overlap between files is high. Other examples include PebblesDB [51], LSM-Trie [68] and WiscKey [39], each of which use different techniques to minimize compaction I/O and write amplification, and thereby significantly improve overall LSM tree performance. Other systems include EvenDB [27], which groups together key-value pairs that are likely to be accessed together, and CLSM [28] and LOCS [63] that improve the concurrency of LSM trees. The techniques employed by all of these systems are largely orthogonal to our work, and can be incorporated into PrismDB.

Unlike the systems mentioned above, Mutant [72] is a key-value store that tries to comprehensively explore the performance-cost trade-off by using hetrogeneous storage devices. Note that it does not consider storage endurance aspects in its design. Mutant ranks the LSM tree SST files based on their access frequencies and then places them on appropriate storage devices through a process called migration. However, as we showed in §6.2, Mutant performs poorly because migrations are expensive; during migration of SST files the read latencies can spike by an order of magnitude (as also reported in the Mutant paper). SST file migrations also increase background I/O significantly.

MyNVM [23] and Wu *et al*. [66] incorporate NVM into SQL databases as a first-level cache ahead of flash, but unlike PrismDB, do not integrate the heterogeneous storage technologies into the basic LSM structure. Since neither of these systems are open source, we cannot compare to them directly. However, since these systems cache at the block granularity, we expect that similar to Mutant, their performance would be lower than PrismDB since they do not do hot-cold separation of objects within the LSM levels.

Caches. Flashield [22] is a hybrid memory and flash key-value cache that uses a CLOCK-based algorithm to filter which objects should be stored in memory and which on flash. Similarly, RIPQ [57] uses segmented LRU to co-locate objects that have similar priorities. Fatcache [2], McDipper [6] and Tao [13] are all caches that use flash as a last level cache to replace DRAM. Persistent Memcached [40] is a key-value cache based on memcached, implemented fully in byte-addressable NVM. All of these systems use some combination of storage and memory, but do not take advantage of both fast and cheap storage technologies.

File Systems. There are several recent works on using NVM to increase up file system performance. Most prominently, ext4 DAX is Linux's officially supported file system extension for byte-addressable NVM. Other examples that either build on top of ext4 DAX or propose entirely new file systems include SplitFS [33], NOVA [71], Aerie [60], Quill [25], Flex [70], SCMFS [67] and BPFS [17]. The main research challenge tackled by these systems is how to accelerate the performance of the file system using NVM, while maintaining consistency after file system crashes. On the other end of the spectrum, DIRECT [54] enables the adoption of cheap and unreliable storage, such as QLC, in distributed file systems, by repairing local errors using remote replicas. While these file systems all utilize new storage technologies, such as NVM or QLC, they all have an assumption of a homogeneous storage medium and do not consider using the full spectrum of available hardware capabilities.

Strata [36] is a file system that uses different types of storage (NVM, normal SSD and HDD) to navigate the cost-performance trade off. For example, similar to our design data is asynchronously flushed down to cheaper storage devices. However, Strata focuses exclusively on the file system and its interactions with the kernel. As our results demonstrate, managing storage placement at the file level misses that a file may contain hundreds or thousands of objects.

[3] This effect is reminiscent of the trade off between disk capacity utilization and write costs in log-structured file systems [52].

8 Summary

By combining multiple storage storage technologies within the same system we can enable both *fast* and *cost-effective* data systems. In this paper, we demonstrate that by making LSM trees read-aware using pinned compactions, PrismDB is able to achieve the Pareto frontier of the performance and cost trade off of hetrogeneous storage. In addition, as a side effect of pinning popular objects to higher levels of the LSM tree, and achieving hot-cold object separation on disk, PrismDB can significantly outperform other LSM tree key-value stores even in homogeneous storage setups. To conclude, we believe that the concept of using a wide spectrum of storage technologies within the same system can also be applied in other contexts, and is an exciting area for future research.

References

[1] Bloom filter. https://en.wikipedia.org/wiki/Bloom_filter.

[2] Fatcache. https://www.github.com/twitter/fatcache.

[3] Flexible I/O tester. https://github.com/axboe/fio.

[4] Intel Optane memory. https://www.intel.com/content/www/us/en/architecture-and-technology/optane-memory.html.

[5] LevelDB. http://leveldb.org/.

[6] McDipper: A key-value cache for flash storage. https://www.facebook.com/notes/facebook-engineering/mcdipper-a-key-value-cache-for-flash-storage/10151347090423920/.

[7] RocksDB wiki. github.com/facebook/rocksdb/wiki//.

[8] Samsung Z-SSD redefining fast responsiveness. https://www.samsung.com/semiconductor/ssd/z-ssd/.

[9] Joy Arulraj and Andrew Pavlo. How to build a non-volatile memory database management system. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, pages 1753–1758, New York, NY, USA, 2017. ACM.

[10] Joy Arulraj, Matthew Perron, and Andrew Pavlo. Write-behind logging. *Proc. VLDB Endow.*, 10(4):337–348, November 2016.

[11] Oana Balmau, Diego Didona, Rachid Guerraoui, Willy Zwaenepoel, Huapeng Yuan, Aashray Arora, Karan Gupta, and Pavan Konka. TRIAD: Creating synergies between memory, disk and log in log structured key-value stores. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 363–375, Santa Clara, CA, July 2017. USENIX Association.

[12] Nathan Beckmann, Haoxian Chen, and Asaf Cidon. LHD: Improving cache hit rate by maximizing hit density. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 389–403, Renton, WA, April 2018. USENIX Association.

[13] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry Li, Mark Marchukov, Dmitri Petrov, Lovro Puzar, Yee Jiun Song, and Venkat Venkataramani. TAO: Facebook's Distributed Data Store for the Social Graph. In *Presented as part of the 2013 USENIX Annual Technical Conference (USENIX ATC 13)*, pages 49–60, San Jose, CA, 2013.

[14] Zhichao Cao, Siying Dong, Sagar Vemuri, and David H.C. Du. Characterizing, modeling, and benchmarking rocksdb key-value workloads at facebook. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, pages 209–223, Santa Clara, CA, February 2020. USENIX Association.

[15] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst.*, 26(2):4:1–4:26, June 2008.

[16] Asaf Cidon, Assaf Eisenman, Mohammad Alizadeh, and Sachin Katti. Cliffhanger: Scaling performance cliffs in web memory caches. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 379–392, Santa Clara, CA, March 2016.

[17] Jeremy Condit, Edmund B Nightingale, Christopher Frost, Engin Ipek, Benjamin Lee, Doug Burger, and Derrick Coetzee. Better i/o through byte-addressable, persistent memory. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 133–146, 2009.

[18] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, page 143–154, New York, NY, USA, 2010. Association for Computing Machinery.

[19] Fernando J Corbato. A paging experiment with the multics system. Technical report, DTIC Document, 1968.

[20] Disctech. Intel 760p. https://www.disctech.com/intel-760p-ssdpekkw128g8xt-128gb-pcie-nvme-solid-state-drive.

[21] Siying Dong, Mark Callaghan, Leonidas Galanis, Dhruba Borthakur, Tony Savor, and Michael Strum. Optimizing space amplification in rocksDB. In *CIDR 2017, 8th Biennial Conference on Innovative Data Systems Research, Chaminade, CA, USA, January 8-11, 2017, Online Proceedings*, 2017.

[22] Assaf Eisenman, Asaf Cidon, Evgenya Pergament, Or Haimovich, Ryan Stutsman, Mohammad Alizadeh, and Sachin Katti. Flashield: a hybrid key-value cache that controls flash write amplification. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 65–78, Boston, MA, February 2019. USENIX Association.

[23] Assaf Eisenman, Darryl Gardner, Islam AbdelRahman, Jens Axboe, Siying Dong, Kim Hazelwood, Chris Petersen, Asaf Cidon, and Sachin Katti. Reducing DRAM footprint with NVM in Facebook. In *Proceedings of the Thirteenth EuroSys Conference*, EuroSys '18, pages 42:1–42:13, New York, NY, USA, 2018. ACM.

[24] Assaf Eisenman, Maxim Naumov, Darryl Gardner, Misha Smelyanskiy, Sergey Pupyrev, Kim Hazelwood, Uladzimir Pashkevich, Asaf Cidon, and Sachin Katti. Bandana: Using non-volatile memory for storing deep learning models. In *SysML '19*, 2019.

[25] Louis Alex Eisner, Todor Mollov, and Steven J Swanson. *Quill: Exploiting fast non-volatile memory by transparently bypassing the file system*. Citeseer, 2013.

[26] Bin Fan, David G. Andersen, and Michael Kaminsky. MemC3: Compact and concurrent MemCache with dumber caching and smarter hashing. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, nsdi'13, pages 371–384, 2013.

[27] Eran Gilad, Edward Bortnikov, Anastasia Braginsky, Yonatan Gottesman, Eshcar Hillel, Idit Keidar, Nurit Moscovici, and Rana Shahout. EvenDB: Optimizing key-value storage for spatial locality. In *Proceedings of the Fifteenth European Conference on Computer Systems*, EuroSys '20, New York, NY, USA, 2020. Association for Computing Machinery.

[28] Guy Golan-Gueta, Edward Bortnikov, Eshcar Hillel, and Idit Keidar. Scaling concurrent log-structured data stores. In *Proceedings of the Tenth European Conference on Computer Systems*, EuroSys '15, New York, NY, USA, 2015. Association for Computing Machinery.

[29] Laura M. Grupp, John D. Davis, and Steven Swanson. The bleak future of NAND flash memory. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies*, pages 17–24, San Jose, CA, 2012.

[30] Intel. Intel thread building blocks (tbb) library. https://software.intel.com/content/www/us/en/develop/tools/threading-building-blocks.html.

[31] Intel. Product brief: Intel optane ssd-900p series pcie. https://www.intel.com/content/dam/www/public/us/en/documents/product-briefs/optane-ssd-900p-brief.pdf.

[32] Intel. Traversing concurrent hash map concurrently. https://software.intel.com/en-us/blogs/2010/05/14/traversing-concurrent_hash_map-concurrently.

[33] Rohan Kadekodi, Se Kwon Lee, Sanidhya Kashyap, Taesoo Kim, Aasheesh Kolli, and Vijay Chidambaram. SplitFS: Reducing software overhead in file systems for persistent memory. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, pages 494–508, New York, NY, USA, 2019. ACM.

[34] Olzhas Kaiyrakhmet, Songyi Lee, Beomseok Nam, Sam H Noh, and Young-Ri Choi. SLM-DB: single-level key-value store with persistent memory. In *17th {USENIX} Conference on File and Storage Technologies ({FAST} 19)*, pages 191–205, 2019.

[35] Kornilios Kourtis, Nikolas Ioannou, and Ioannis Koltsidas. Reaping the performance of fast NVM storage with uDepot. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*, pages 1–15, Boston, MA, February 2019. USENIX Association.

[36] Youngjin Kwon, Henrique Fingler, Tyler Hunt, Simon Peter, Emmett Witchel, and Thomas Anderson. Strata: A cross media file system. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 460–477, 2017.

[37] Avinash Lakshman and Prashant Malik. Cassandra: A decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44(2):35–40, April 2010.

[38] Baptiste Lepers, Oana Balmau, Karan Gupta, and Willy Zwaenepoel. KVell: The design and implementation of a fast persistent key-value store. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, New York, NY, USA, 2019. Association for Computing Machinery.

[39] Lanyue Lu, Thanumalayan Sankaranarayana Pillai, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. WiscKey: Separating keys from values in SSD-conscious storage. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*, pages 133–148, Santa Clara, CA, February 2016.

[40] Virendra J. Marathe, Margo Seltzer, Steve Byan, and Tim Harris. Persistent memcached: Bringing legacy code to byte-addressable persistent memory. In *Proceedings of the 9th USENIX Conference on Hot Topics in Storage and File Systems*, HotStorage'17, pages 4–4, Berkeley, CA, USA, 2017. USENIX Association.

[41] Yoshinori Matsunobu. Myrocks: A space and write-optimized MySQL database. code.facebook.com/posts/190251048047090/myrocks-a-space-and-write-optimized-mysql-database/.

[42] C Mellor. Toshiba flashes 100TB QLC flash drive, may go on sale within months. really. http://www.theregister.co.uk/2016/08/10/toshiba_100tb_qlc_ssd//.

[43] Micron. Comparing SSD and HDD endurance in the age of QLC SSDs. https://www.micron.com/-/media/client/global/documents/products/white-paper/5210_ssd_vs_hdd_endurance_white_paper.pdf.

[44] Micron. QLC NAND technology. https://www.micron.com/products/advanced-solutions/qlc-nand.

[45] MongoDB. WiredTiger. http://www.wiredtiger.com/.

[46] A. Narayan and P. Mattis. Why we built CockroachDB on top of RocksDB. https://www.cockroachlabs.com/blog/cockroachdb-on-rocksd/.

[47] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C. Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, David Stafford, Tony Tung, and Venkateshwaran Venkataramani. Scaling Memcache at Facebook. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 385–398, Lombard, IL, 2013.

[48] Matheus Almeida Ogleari, Ethan L Miller, and Jishen Zhao. Steal but no force: Efficient hardware undo+redo logging for persistent memory systems. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 336–349. IEEE, 2018.

[49] S Ohshima and Y Tanaka. New 3D flash technologies offer both low cost and low power solutions. https://www.flashmemorysummit.com/English/Conference/Keynotes.html.

[50] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. The log-structured merge-tree (LSM-tree). *Acta Informatica*, 33(4):351–385, 1996.

[51] Pandian Raju, Rohan Kadekodi, Vijay Chidambaram, and Ittai Abraham. PebblesDB: Building key-value stores using fragmented log-structured merge trees. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 497–514, New York, NY, USA, 2017. ACM.

[52] Mendel Rosenblum and John K Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems (TOCS)*, 10(1):26–52, 1992.

[53] Zhenyu Song, Daniel S. Berger, Kai Li, and Wyatt Lloyd. Learning relaxed belady for content distribution network caching. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 529–544, Santa Clara, CA, February 2020. USENIX Association.

[54] Amy Tai, Andrew Kryczka, Shobhit O. Kanaujia, Kyle Jamieson, Michael J. Freedman, and Asaf Cidon. Who's afraid of uncorrectable bit errors? online recovery of flash errors with distributed redundancy. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 977–992, Renton, WA, July 2019. USENIX Association.

[55] Billy Tallis. The crucial P1 1TB SSD review: The other consumer QLC SSD. https://www.anandtech.com/show/13512/the-crucial-p1-1tb-ssd-review.

[56] Billy Tallis. Intel details upcoming SSDs for datacenter including QLC NAND. https://www.anandtech.com/show/13157/intel-details-upcoming-ssds-for-datacenter-including-qlc-nand.

[57] Linpeng Tang, Qi Huang, Wyatt Lloyd, Sanjeev Kumar, and Kai Li. RIPQ: Advanced photo caching on flash for Facebook. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies*, pages 373–386, Santa Clara, CA, 2015.

[58] Alexander van Renen, Viktor Leis, Alfons Kemper, Thomas Neumann, Takushi Hashida, Kazuichi Oe, Yoshiyasu Doi, Lilian Harada, and Mitsuru Sato. Managing non-volatile memory in database systems. In *Proceedings of the 2018 International Conference on*

Management of Data, SIGMOD '18, New York, NY, USA, 2018. Association for Computing Machinery.

[59] Alexander van Renen, Lukas Vogel, Viktor Leis, Thomas Neumann, and Alfons Kemper. Persistent memory I/O primitives. In *Proceedings of the 15th International Workshop on Data Management on New Hardware*, DaMoN'19, New York, NY, USA, 2019. Association for Computing Machinery.

[60] Haris Volos, Sanketh Nalli, Sankarlingam Panneerselvam, Venkatanathan Varadarajan, Prashant Saxena, and Michael M Swift. Aerie: Flexible file-system interfaces to storage-class memory. In *Proceedings of the Ninth European Conference on Computer Systems*, pages 1–14, 2014.

[61] Carl Waldspurger, Trausti Saemundsson, Irfan Ahmad, and Nohhyun Park. Cache modeling and optimization using miniature simulations. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 487–498, Santa Clara, CA, 2017. USENIX Association.

[62] Carl A. Waldspurger, Nohhyun Park, Alexander Garthwaite, and Irfan Ahmad. Efficient MRC construction with SHARDS. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*, pages 95–110, Santa Clara, CA, 2015. USENIX Association.

[63] Peng Wang, Guangyu Sun, Song Jiang, Jian Ouyang, Shiding Lin, Chen Zhang, and Jason Cong. An efficient design and implementation of LSM-tree based key-value store on open-channel SSD. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, New York, NY, USA, 2014. Association for Computing Machinery.

[64] Sean Webster. Intel ssd 660p. https://www.tomshardware.com/reviews/intel-ssd-660p-qlc-nvme,5719.html.

[65] Kan Wu, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. Towards an unwritten contract of Intel Optane SSD. In *11th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 19)*, Renton, WA, July 2019. USENIX Association.

[66] Kan Wu, Andrea Arpaci-Dusseau, Remzi Arpaci-Dusseau, Rathijit Sen, and Kwanghyun Park. Exploiting Intel Optane SSD for Microsoft SQL Server. In *Proceedings of the 15th International Workshop on Data Management on New Hardware*, DaMoN'19, New York, NY, USA, 2019. Association for Computing Machinery.

[67] Xiaojian Wu, Sheng Qiu, and AL Narasimha Reddy. SCMFS: A file system for storage class memory and its extensions. *ACM Transactions on Storage (TOS)*, 9(3):1–23, 2013.

[68] Xingbo Wu, Yuehai Xu, Zili Shao, and Song Jiang. LSM-trie: An LSM-tree-based ultra-large key-value store for small data items. In *Proceedings of the 2015 USENIX Annual Technical Conference*, pages 71–82, Santa Clara, CA, 2015.

[69] Fei Xia, Dejun Jiang, Jin Xiong, and Ninghui Sun. HiKV: A hybrid index key-value store for DRAM-NVM memory systems. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 349–362, Santa Clara, CA, 2017. USENIX Association.

[70] Jian Xu, Juno Kim, Amirsaman Memaripour, and Steven Swanson. Finding and fixing performance pathologies in persistent memory software stacks. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 427–439, 2019.

[71] Jian Xu and Steven Swanson. NOVA: A log-structured file system for hybrid volatile/non-volatile main memories. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*, pages 323–338, Santa Clara, CA, 2016. USENIX Association.

[72] Hobin Yoon, Juncheng Yang, Sveinn Fannar Kristjansson, Steinn E. Sigurdarson, Ymir Vigfusson, and Ada Gavrilovska. Mutant: Balancing storage cost and latency in lsm-tree data stores. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '18, page 162–173, New York, NY, USA, 2018. Association for Computing Machinery.