Efficient Compactions between Storage Tiers with PrismDB

Ashwini Raina
Princeton University
United States
araina@cs.princeton.edu

Jianan Lu
Princeton University
United States
jiananl@princeton.edu

Asaf Cidon
Columbia University
United States
asaf.cidon@columbia.edu

Michael J. Freedman
Princeton University
United States
mfreed@cs.princeton.edu

ABSTRACT

In recent years, emerging storage hardware technologies have focused on divergent goals: better performance or lower cost-per-bit. Correspondingly, data systems that employ these technologies are typically optimized either to be fast (but expensive) or cheap (but slow). We take a different approach: by architecting a storage engine to natively utilize two tiers of fast and low-cost storage technologies, we can achieve a Pareto efficient balance between performance and cost-per-bit.

This paper presents the design and implementation of PrismDB, a novel key-value store that exploits two extreme ends of the spectrum of modern NVMe storage technologies (3D XPoint and QLC NAND) simultaneously. Our key contribution is how to efficiently migrate and compact data between two different storage tiers. Inspired by the classic cost-benefit analysis of log cleaning, we develop a new algorithm for multi-tiered storage compaction that balances the benefit of reclaiming space for hot objects in fast storage with the cost of compaction I/O in slow storage. Compared to the standard use of RocksDB on flash in datacenters today, PrismDB's average throughput on tiered storage is 3.3× faster, its read tail latency is 2× better, and it is 5× more durable using equivalently-priced hardware.

CCS CONCEPTS

• **Information systems** → **Data layout**; • **Computer systems organization** → *Secondary storage organization.*

KEYWORDS

PrismDB, key-value store, tiered, storage, compaction

ACM Reference Format:
Ashwini Raina, Jianan Lu, Asaf Cidon, and Michael J. Freedman. 2023. Efficient Compactions between Storage Tiers with PrismDB. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3 (ASPLOS '23), March 25–29, 2023, Vancouver, BC, Canada.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3582016.3582052

1 INTRODUCTION

Several new NVMe storage technologies have recently emerged, expressing the competing goals of improving performance and reducing storage costs. On one side, high performance non-volatile memory (NVM[1]) technologies, such as Optane SSD [30, 72] and Z-NAND [59], provide single-digit μs latencies. On the other end of the spectrum, cheap and dense storage such as QLC NAND [45, 47] 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 less reliable than less dense flash technology (e.g., TLC NAND).

Table 1: Comparing NVM (Optane SSD) and dense flash (QLC). Cost taken from cdw.com for Intel's Optane SSD 5800x and Intel 660p, and lifetime is based on publicly available information [29, 70]. Latency of 4 KB random read is computed with Fio [4].

	NVM	QLC
Lifetime (DWPD)	200	0.1
Cost ($/GB)	$2.5	$0.1
Avg Read Latency (4 KB)	6μs	391μs

Table 1 compares the large variation in endurance, cost and performance across two representative storage technologies. For example, we observe that there is a roughly 65× performance difference between Optane SSD (NVM) and QLC on random reads, and sequential reads show a similar trend (not shown). However, Optane SSD costs more than 20× per GB compared to QLC. Endurance also varies widely: QLC NAND can only sustain a relatively small number of writes before exhibiting errors [46].

Many studies have shown that simply running existing software systems on new hardware storage technologies often leads to poor results [13, 16, 21, 22, 39]. Therefore, significant recent effort has sought to build new software storage systems that are architected specifically for these new technologies [16, 21, 22, 39, 61, 63]. They typically choose one point in the design space: fast but expensive [16, 22, 39] (e.g., using Optane SSD or Z-NAND), or cheap but slower [21, 61, 63] (e.g., using dense flash). However, these systems do not exploit the cost-performance benefits of using multiple storage tiers.

While a few recent key-value stores [13, 73, 74, 79] combine fast and cheap storage technologies together, they reuse the same monolithic data structure used for flash (typically log-structured merge trees) and use naive techniques for compacting data across tiers (e.g., by simply copying entire files from one tier to the other). As we demonstrate experimentally in §3, these existing approaches are ill-equipped for a multi-tiered use case, because they inefficiently use fast NVM devices. In fact, we spent the first year of this project trying to retrofit a log-structured merge (LSM) tree to use multiple storage tiers, and take advantage of fast NVM devices by retaining more frequently-accessed data on NVM. However, we were not able to show any performance improvement because doing so led to excessive compactions, hurting overall performance.

[1] In the paper, "NVM" refers to fast block devices (e.g., Intel Optane SSD, Samsung Z-NAND, Kioxia FL6). We do not focus on persistent memory.

ASPLOS '23, March 25–29, 2023, Vancouver, BC, Canada

ACM ISBN 978-1-4503-9918-0/23/03.
https://doi.org/10.1145/3582016.3582052

This led us to fundamentally rethink key-value store data structures and compaction mechanisms to fully exploit fast and slow storage tiers, in order to realize more optimal trade-offs between performance, endurance, and cost. We design a new key-value store, PrismDB, which assumes a setting where a large percentage (e.g., 75-95%) of the storage capacity sits on dense flash (e.g., QLC), and the rest on NVM (e.g., Optane SSD). In this setting, given the far inferior performance and endurance of flash compared to NVM, PrismDB's primary design goals are to maximize read and write performance, while minimizing the amount of I/O (and especially writes) issued to flash.

To achieve this goal, PrismDB uses a hybrid data layout, where hot objects are stored in slab-based files on NVM (where random access is fast), and cold objects are stored in a sorted log on flash (where sequential writes are prioritized). Since NVM does not suffer from write amplification and supports fast random writes, it can efficiently perform in-place updates and fresh inserts of objects directly into slabs. PrismDB tries to store frequently read or updated data in NVM, and cold, immutable data on flash. To estimate the access frequency of objects, PrismDB uses a lightweight object popularity mechanism based on the clock algorithm. Yet since key popularity distributions vary across workloads, PrismDB records clock value distributions and uses them to determine which objects are colder and should be compacted to QLC. As most requests are served from either DRAM or NVM, the bottleneck shifts from I/O to CPU. PrismDB employs a partitioned, shared-nothing architecture to minimize the amount of synchronization between threads.

Since space in the faster tier (NVM) quickly fills up, PrismDB needs to decide which objects to migrate from NVM to flash. This leads to a fundamental trade-off: keeping a large number of popular objects in NVM ensures a higher proportion of accesses are served from NVM, but it comes at the expense of migration or compaction efficiency. Retaining more objects in NVM means the system needs to work harder to find less-popular objects, and thus has to merge with a wider range of keys in flash, thereby increasing expensive flash write I/O. In addition, when deciding which keys to compact, PrismDB needs to consider other typical factors [52, 57] that impact compaction performance: the overlap between the merged key ranges and amount of stale data to be cleaned.

Our primary contribution is multi-tiered storage compaction (MSC), a novel compaction mechanism that captures the relationship between key popularity and write amplification on multi-tiered storage. We design a new metric by adapting the classic cost-benefit model of the log-structured file system (LFS) [57] to a multi-tiered setting and assign a score to each range of keys that measures whether they are good candidates for compaction. The score is higher when ranges have more cold data that can be compacted from the faster storage tier, and is lower when it incurs a large amount of flash I/O per compacted object. Since computing this metric precisely is expensive, PrismDB introduces an approximation algorithm that performs well in practice (§5.3). PrismDB does not only migrate data from the faster tier to the lower tier, but also, under read-heavy workloads, it can promote objects from the slower tier to the faster one in response to changing object popularity.

We implement PrismDB and compare it to a multi-tiered version of RocksDB [23], a version of RocksDB that uses NVM as a second-level (L2) cache, a version of RocksDB that pins hot objects to levels mapped on NVM, as well as to two academic systems - Mutant [79] and SpanDB [13]. PrismDB significantly outperforms all the baselines under all YCSB [17] workloads that issue point queries (puts/gets/updates) and under Twitter production traces [76] that are insert or read heavy.

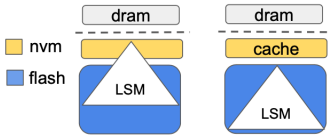


Figure 1: Tiered storage designs: embedded within a single-tier data structure (left) and extra cache capacity (right).

Our paper makes the following primary contributions:

(1) **Architecture.** A hybrid architecture that exploits NVM in a multi-tiered storage setup while minimizing flash writes.

(2) **Multi-tiered compactions.** A model and algorithm for efficient compactions for multi-tiered storage that balances data placement with compaction I/O in the slow tier.

(3) **Popularity scoring.** A novel algorithm for popularity-based object placement decisions on multi-tiered storage.

(4) **Performance and cost-efficiency.** Compared to multi-tiered RocksDB, PrismDB's more efficient data structures and compaction mechanism improve the throughput and average latency by 2.4× and 2.7× on write-dominated workloads, and by 2.5× and 2× on read-heavy workloads. PrismDB achieves 3× higher throughput than SpanDB on fsync enabled workloads. PrismDB is 5× more durable compared to single-tier RocksDB on TLC flash, which is the standard deployment in datacenters today.

2 BACKGROUND AND DIRECTLY RELATED WORK

We provide a brief background on new storage technologies, and survey systems that use multiple NVMe storage tiers.

Trends in storage. In recent years, NVMe storage devices have evolved in two orthogonal directions: faster (and more expensive) non-volatile memory and cheaper (and slower) dense flash. New fast NVM technologies, such as 3D XPoint [30, 47] and Z-NAND [59], which we refer to collectively throughout the paper as Non-Volatile Memory (NVM), provide random read and write latencies of 10μs or less.

On the other end of the spectrum, NAND or 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 [28, 32, 47, 51, 61, 70]. The latest QLC NAND technology, which packs 4 bits per memory cell, can only tolerate 100–300 write cycles before it becomes unusable [51, 62, 70]. Future dense flash technologies, such as the recently announced PLC NAND (5 bits per cell), will exacerbate this trade-off [32].

Existing tiered storage designs. With the emergence of fast NVMe storage devices, several recent storage system architectures

Table 2: Comparing single-tier NVM and QLC with multi-tier (89% QLC, 11% NVM, labeled "het") on Zipf 0.8 workload.

	RocksDB			PrismDB
	NVM	QLC	het	het
Throughput (Kops/sec)	121	54	93	184
Cost ($/GB)	$2.5	$0.1	$0.3	$0.3

were proposed for tiered storage. These designs typically try to augment an existing key-value store, which was typically originally designed for flash, with a small amount of fast storage (e.g., Optane SSD). We can broadly group the architectures into two categories, depicted by Figure 1:

(1) **Embedded in data structure.** NVM is incoporated into an existing flash-based data structure. This approach is used by Mutant [79] and SpanDB [13], both of which employ log-structured merge (LSM) trees. Mutant migrates cold LSM files to slow storage, while SpanDB stores the tree's top levels on NVM, and the bottom ones on flash.

(2) **Extra cache.** NVM is simply treated as extra cache space for objects that are stored in DRAM. Example of such systems are: MyNVM [22], SQL Server [73] and Orthus-KV [74]. MyNVM and SQL Server simply treat NVM as an L2 cache, while Orthus-KV dynamically uses NVM as an L2 cache or as auxiliary storage tier that provides extra storage bandwidth.

In both of these approaches, a traditional single-tier data structure, which is optimized for flash (typically an LSM-tree), is retrofitted to use NVM. While these existing approaches have the advantage of relatively easy integration with existing flash-based key-value stores, they do not take full advantage of NVM. In the next section, we analyze why.

3 LSM PERFORMANCE ON EMERGING STORAGE

Before describing why existing multi-tiered designs fail to achieve the full potential of NVM, we outline the desired properties of such a system. For a persistent key-value store to be high-performance and affordable, it needs to satisfy the following design goals.

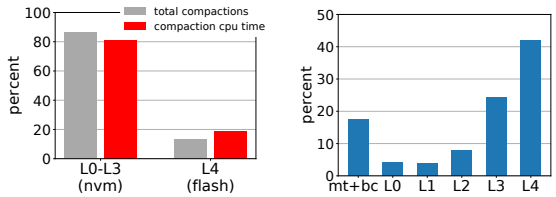
(1) **Navigate cost-performance trade-off.** The system should provide high performance (throughput, read/write latency), while most of its data (i.e., 75-95%) is on low-cost storage.

(2) **Support small objects.** The system needs to support datacenter key-value workloads that consist of small objects (i.e., 1 KB or smaller [11, 76]). Thus, we cannot assume the database's index entirely fits in DRAM [16, 22].

(3) **Minimize flash I/O.** Since flash is slow, the system should minimize flash reads. It should also reduce flash writes to maximize system lifetime given flash's limited write cycles.

We now evaluate these design goals on an LSM-based key-value store. We use RocksDB, a popular open-source key-value store [11]. Throughout the paper, by default we use YCSB-A [17] (50% reads, 50% updates), on a 100 GB database with 1 KB object sizes on hardware described in §7.

Single-tier storage. We first analyze how a single-tier storage setup performs using NVM and QLC, the two extremes of the cost-performance trade-off. Table 2 compares the throughput and cost of



(a) (b)

Figure 2: (a) Percentage of compactions in NVM and QLC in a multi-tier RocksDB. (b) Distribution of reads across LSM levels (L0-L4), memtable (mt) and block cache (bc).

running RocksDB on the two storage configurations. We note that even though raw NVM performance is far superior to QLC (e.g., 65× lower average 4 KB random read latency), the observed overall system throughput in Table 2 is only 2.2× better than QLC. On QLC, RocksDB is I/O bound, and spends 36% of its CPU time in iowait because QLC I/O is slow. On NVM, RocksDB is bottlenecked by CPU, since the system spends less time waiting for I/O completions. This leads us to ask: would a multi-tiered storage setup be bottlenecked by QLC I/O, CPU contention or both?

Multi-tier storage. We next examine how an LSM-tree performs on a multi-tiered setup, similar to the one used by SpanDB [13]. We refer to the LSM multi-tiered storage setup as *het* (i.e., heterogeneous). We use a 5-level LSM-tree, in which levels L0-L3 are mapped to NVM, and L4 to QLC, where L4 is 89% of the storage capacity in the database.

By storing only 11% of the database on NVM, *het*'s cost-per-bit ($0.34/GB) is close to standard single-tier TLC flash setups used today in datacenters ($0.31/GB). At the same time, it achieves a throughput that is only 23% less than the throughput of RocksDB running only NVM. Therefore, a multi-tier RocksDB configuration pays a small extra cost for faster storage but achieves a significant performance boost.

Nevertheless, the performance of RocksDB in this configuration is still far from optimal. We make two observations. First, as shown by others [5, 20], RocksDB spends significant CPU time (54%) on background compactions. However, in our experiments with RocksDB in the tiered setting (Figure 2a), more than 80% of compaction time is spent sorting data in the NVM tier. Since NVM supports fast in-place updates, spending CPU cycles sorting objects is unnecessary.

Second, significant CPU time (23%) is still spent on I/O wait. Our results show (Figure 2b), that even though RocksDB uses NVM for the upper levels of its tree, many reads are still served from flash (42%), which is 65× slower than NVM, contributing to high I/O wait time. This negates the full performance benefit of using NVM as a fast storage tier. The reason more reads are not served from NVM is that LSM-tree organization is purely write-driven; LSM-trees do not try to cache frequently-read items on the upper levels.

To add read-awareness to the LSM-tree, we spent the first full year of this project building a prototype system, Rocksdb-RA, that stores more frequently-read objects on upper levels of the tree (L0-L3) on NVM using *pinned compactions*. Unlike traditional LSM compaction that compacts all the objects down to the lower level,

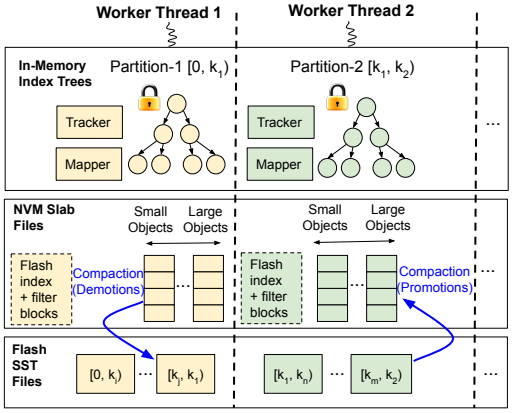


Figure 3: PrismDB system diagram.

RocksDB-RA "pins" some percentage of the popular objects, retaining them in the NVM tier.

However, due to limited capacity of a level, retaining objects results in lesser free space and triggers more compactions on that level. We found that even though Rocksdb-RA was able to serve 27% more client reads from NVM compared to flash, the total number of compactions increased by 2.3×, which resulted in an overall degradation of the system's throughput and latency compared to RocksDB.

This leads us to conclude there exists a fundamental tension between object pinning and compaction efficiency, which we explore in this paper. Unfortunately, the salient features of an LSM tree, including its multiple levels and the need to use large sorted files, make compactions particularly expensive. As object pinning leads to more compactions, it typically degrades the performance in LSM trees. This motivates our decision to architect a new data layout and compaction mechanism that is tailored to multi-tiered storage.

4 PRISMDB'S DATA LAYOUT

In this section, we introduce PrismDB's system components.

4.1 Design Overview

Figure 3 depicts the architecture of PrismDB on two-tiered storage (NVM and flash). Since NVM is low latency, in order to reduce synchronization on shared data structures, PrismDB employs a partitioned, shared-nothing architecture. Each partition consists of a subset of the key space and runs a dedicated worker thread to handle client requests one at a time. This partitioned approach is extended across all storage tiers; i.e., each partition handles its own data structures on DRAM, NVM, and flash. The partitioning algorithm can be range-based for scan heavy workloads or hash-based for workloads that exhibit load skew/imbalance.

PrismDB uses a hybrid data layout, optimized for both tiers. At a high level, metadata (e.g., indices and bloom filters) is stored on DRAM and NVM for fast lookups. DRAM and NVM also store recently-accessed objects. In order to support fast writes and reduce flash wear, all newly-written data (including updates) are written to NVM. Therefore, to optimize for fast random writes and in-place updates, the NVM data layout uses unsorted slabs. On the other hand, the data layout on flash is optimized for sequential write access in order to minimize write amplification, and is thus based on a sorted log.

Each partition uses a *tracker* and a *mapper* that are stored in DRAM. The tracker estimates key popularity using the clock algorithm. The mapper maintains the clock value distributions and uses those distributions to enforce the placement of objects on storage. It uses a parameter, called the pinning threshold, to determine whether an object is "popular" enough for NVM. Each partition runs a background compaction thread to free up space on NVM by moving the colder objects to flash while retaining the popular ones on NVM. Key ranges are selected for compaction based on a new compaction algorithm, called multi-tiered storage compaction (MSC), which we describe in §5. We now provide more detail on the data structures on DRAM, NVM, and flash.

In-memory data structures. We use a B-tree index to locate NVM objects, which are not sorted. Since we need to support a large database of small objects, an index over all keys in the database will not fit in memory. Hence, only the index for objects on NVM is stored in memory. Each index entry in the B-tree stores the key and its NVM address (i.e., 1-byte slab ID plus 4-byte page offset). Each partition also stores the clock-based tracker in memory (see §4.3). Each tracker entry includes the key and 1-byte clock metadata. PrismDB does not use a userspace DRAM cache for caching frequently-read objects and instead relies on the OS page cache.

Data layout on NVM. All new data is written to NVM, and it also serves as a second-level read cache after DRAM for hot objects. NVM uses a slab-based data layout to support fast random inserts and updates of small objects. PrismDB uses a set of slab files each of which is dedicated to a specific object size range (e.g., 100 B, 200 B, …, 1 KB). Objects of similar size are inserted into available fixed-size slots in the right slab file, along with a metadata header containing version number (implemented as logical timestamp) and object size information. If an object is deleted, its slot is freed for new data. An in-place update can take place on the original slot if the object doesn't change its size range, and if it does, the object needs to be deleted and moved to another slab file.

The index for objects on flash is stored in NVM rather than DRAM, because reading them from flash (i.e., hundreds of microseconds) amortizes the cost of the index lookup. Similarly, PrismDB stores on NVM a bloom filter [8] for each file on flash to prevent issuing expensive flash I/Os for non-existent objects. The combined size of the flash index and filters can range from 100s of MBs to a few GBs, depending on the object size. For a small-object database, the flash index and filters reside entirely on NVM.

Data layout on flash. To support large sequential writes, data on flash uses Sorted String Table (SST) files (similar to LSM trees [23, 27]), which are stored in a log. Each SST file has an index that points to all the file's data blocks, and a bloom filter for the file's objects. SST files store disjoint key ranges, which makes searching an object fast. When the percentage of NVM in PrismDB is 10% or higher, by default we store all the flash data in a single-level log. When the percentage is lower, PrismDB by can store the flash data in a multi-level log, similar to an LSM-tree. This choice is based on the fanout of objects stored on NVM and flash.

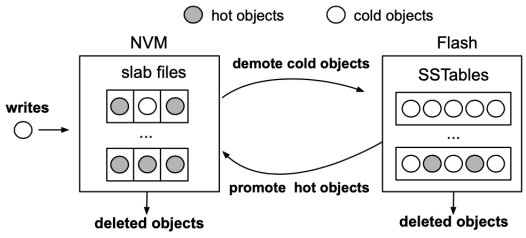


Figure 4: Lifecycle of a written object in PrismDB.

4.2 Lifecycle of a Written Object

Figure 4 illustrates the lifecycle of an object written to PrismDB. Objects are always synchronously written to NVM because writes need to be persisted for crash recovery. When NVM's used capacity hits a high watermark (by default 98%), the partition triggers a background compaction job to demote colder objects to flash until it frees up enough space on NVM (by default when NVM usage reduces to 95%). In the meantime, incoming writes are rate-limited to ensure NVM does not exceed its capacity. The job selects "cold" NVM objects, by filtering the hot ones using the mapper, and *demotes* them to flash. At the same time, given that PrismDB is already incurring the cost of writing a new SST file, if the compaction job finds hot flash objects, it may *promote* them to NVM during the compaction. In read-heavy workloads, where write-triggered compactions are rare, PrismDB proactively triggers compactions when it detects that too many objects are accessed from flash, with the goal of promoting hot objects to NVM. Since NVM stores more recent data, obsolete versions of objects on flash are deleted when merging with new versions from NVM.

4.3 Popularity Tracking

We now discuss in more detail how PrismDB tracks popular objects, and designates them as hot (or not).

Tracker: lightweight tracking of objects. The tracker estimates object access popularity, while incurring a minimal overhead. There is a large body of work on how to track and estimate object popularity [7, 60, 63, 67, 68]. However, many existing mechanisms require a relatively large amount of data per object, and computation per access. Given that key-value objects are often small (e.g., less than 1 KB [11, 14, 49]), we need to limit the amount of metadata we use for tracking purposes per object. We also need to be able to track millions of objects at a high throughput.

We turn to *clock* [18], a classic approach to approximate the least recently used (LRU) eviction policy while offering better space efficiency and concurrency [21, 24]. PrismDB's tracker uses the multi-bit clock algorithm for object tracking. The tracker uses a concurrent hash map that maps object keys to their clock bits. The object values are not stored in the tracker. Each client read or update operation requires the tracker to update the clock bits of the object that was accessed. Once the tracker becomes full, it uses the clock algorithm to decrement clock values of objects and then evicts the object with value 0.

Since setting the clock bits is on the critical path of reads and updates, the tracker is optimized for concurrent key insertions, evictions, and lookups. Further, to save space, the tracker does not store clock bits of all key-value pairs in the database, only the most recently accessed. In our evaluation, tracker size is set to 10% of the total database keys; see §6 for more details.

Mapper: enforcing the pinning threshold. Ideally, PrismDB needs a threshold to determine which objects are hot, which we call the *pinning threshold*. At each pass of the compaction job, PrismDB should pin on faster storage some percent (e.g., 10%) of the most popular objects that are being tracked.

However, enforcing this threshold depends on the clock bit distribution, which vary as a function of the workload (Figure 12 in appendix). Consider a two-bit clock, with 3 being most popular and 0 being least popular. If PrismDB wants to enforce a pinning threshold of 10%, and exactly 10% of keys have all their clock bits set to 3, then PrismDB should pin all the objects with a clock value of 3. However, if 50% of the keys have a clock value of 3, then PrismDB cannot pin all objects a value of 3, otherwise it will exceed the desired pinning threshold. To this end, the mapper is responsible for keeping track of the clock value distribution, and uses that distribution to enforce the pinning threshold.

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 such that 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 1 is 30%, and the remaining 50% have a clock value of 0. Now, suppose the desired threshold is 15% (i.e., the most popular 15% of objects are stored on NVM, and the rest on flash). If the compaction job encounters an object with a clock value of 3, it will always pin it. If it encounters an object with a clock value of 2, it will randomly choose whether to keep it or not (in this example, with a probability of 0.5). If it encounters either an object with clock value of 1 or 0, or an object that is not currently being tracked (recall the tracker does not track all objects in the database), that object will be demoted to flash.

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

5 MULTI-TIERED STORAGE COMPACTION

In this section we describe PrismDB's compaction mechanism. We discuss the trade-off between garbage collection and data pinning (§5.1). We then present an analytical model for multi-tiered compaction efficiency (§5.2). Then we present our multi-tiered storage compaction algorithm (MSC) (§5.3).

5.1 Performance Trade-Off

PrismDB's compaction serves two purposes. First, it needs to reclaim enough space in NVM to absorb new incoming writes. Compaction frees up space in NVM when it is close to running out of capacity, and removes stale values from flash that have been updated more recently in NVM. Second, it needs to prioritize limited NVM space for hot data. Compaction demotes cold objects to flash and promotes hot objects to NVM.

However, these two functions present a fundamental trade-off. On one hand, if PrismDB were to move objects from NVM to flash

without considering their popularity, then NVM will not serve as an effective second-level cache. On the other hand, if PrismDB were to pin a high percentage of objects that it encounters, it will take it much longer (and require more CPU) to free up enough space in NVM. This will not only delay incoming client writes, but also incur higher flash write amplification. This is because the objects in NVM selected to be demoted will have a "sparser" key range, and require the compaction job to merge the NVM objects with a larger number of SST files on flash.

5.2 Modeling Compaction Cost and Benefit

Since data on NVM is organized as individual key-value pairs and unpopular objects can be scattered across the entire key space, there are many possible choices for selecting objects for compaction. To reduce the search space and bound the flash I/O caused by each compaction, PrismDB divides the contiguous NVM key space into smaller key ranges based on existing SST file bounds on flash. We define a compaction key range, i, as the key ranges of i consecutive SST files. The value of i is tunable in PrismDB, and by default is set to 1. A higher value of i is more suitable for workloads with where popular objects are evenly distributed across the key space.

Analytical model. We design a novel compaction policy, which is inspired by the classic log cleaning cost-benefit analysis [57]. In traditional log cleaning or compaction (e.g., LFS [57] and other related systems [15, 58]), the system tries to select the optimal contiguous segment of data to compact. It tries to choose the segment that offers the highest benefit (free up the most space on disk, and keep it free for a long time), at the lowest cost (I/O incurred by the garbage collection). We adopt a similar approach, but we model the benefit and cost differently, adapting them to the multi-tiered setting.

Benefit. We model the benefit as demoting as many "cold" objects to the slow storage tier as possible. Cold objects are ones that haven't been recently accessed (read or written). We consider writes, because objects that are not frequently updated are likely to stay "stable" in the future, which avoids moving them back and forth across tiers thereby saving costly flash disk bandwidth. Thus, compaction offers greater benefit to the system if it can move more cold data to the slower tier.

We assign every object a *coldness* score between (0, 1] (where 1 is cold and 0 is hot). The coldness of object j is the inverse of its clock value incremented by 1, $coldness(j) = \frac{1}{clock_j+1}$. We increment the clock value by 1 to avoid dividing by zero. If the object doesn't appear in the tracker, we assume its clock value is 0 and its coldness score is therefore 1.

We define the multi-tiered compaction *benefit* of a key range as the the sum of coldness values of the objects in that range: benefit $= \sum_{j=1}^{t_n} coldness(j)$, where t_n is the number of objects in the key range. Table 3 lists the notations and their meanings.

Cost. Log-structured systems like LFS typically consider the cleaning cost as the extra I/O incurred on the same disk where space is freed. In our multi-tiered disk setup, however, I/O on the slower tier is an order of magnitude costlier (in terms of bandwidth, latency and endurance) than I/O on the faster tier. Thus, for simplicity, we model the cost of compaction as total flash I/O incurred per migrated byte from NVM. During compaction, older versions of objects on flash will first be read and later deleted when merging with more recent NVM data. Thus, compacting an NVM key range involves reading SST file objects from flash, and then writing unpopular NVM objects and non-overlapping SST file objects back to flash.

Table 3: Notations.

Symbol	Description
$clock_j$	Clock value of a NVM object j.
t_n	Total number of objects in selected NVM key range.
t_f	Total number of objects in SST file before merging.
F	The fanout ratio $\frac{t_f}{t_n}$ between key range on flash and NVM.
p	Fraction of popular objects in candidate NVM key range.
o	Fraction of overlapping objects between NVM range and flash.

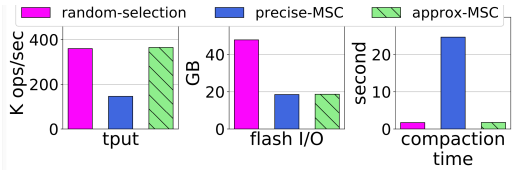


Figure 5: Comparison of throughput, flash write I/O and average compaction time for precise-MSC and approx-MSC metric, and random-selection policy on YCSB-A Zipf 0.99.

We initially assume object sizes to be equal, and define notations in terms of "number of objects", which directly equates to their size. For an NVM key range where p is the ratio of popular objects, the number of unpopular objects is $(1-p)\cdot t_n$, where t_n is the number of objects in that candidate NVM key range. We define o as the fraction of objects in the SST file that also appear in the NVM key range. Then the number of non-overlapping objects in the SST file becomes $(1-o)\cdot t_f$, where t_f is the total number of objects in the flash SST file. Thus compacting $(1-p)\cdot t_n$ objects on NVM incurs t_f read I/O from flash and $(1-p)\cdot t_n + (1-o)\cdot t_f$ write I/O to flash. The flash I/O per migrated object is: $\frac{t_f+(1-p)\cdot t_n+(1-o)\cdot t_f}{(1-p)\cdot t_n} = \frac{(2-o)\cdot t_f}{(1-p)\cdot t_n} + 1$ We define $\frac{t_f}{t_n}$ as the fanout, F, which represents the size ratio of key ranges on NVM and flash. Therefore, the flash I/O cost is reduced to $F \cdot \frac{(2-o)}{(1-p)} + 1$.

Finally, we define our multi-tiered storage compaction (MSC) metric as the ratio of benefit to cost.

$$\text{MSC} = \frac{\text{benefit}}{\text{cost}} = \frac{\sum_{j=1}^{t_n} coldness(j)}{F \cdot \frac{(2-o)}{(1-p)} + 1} \quad (1)$$

The metric's score is higher for key ranges that contain more colder objects and which incur lower I/O overhead per migrated object to flash. Given a list of candidate key ranges, PrismDB selects the range with the highest score for compaction. For workloads with variable-sized objects, p, o and F are normalized by size (in bytes) to compute MSC precisely.

5.3 MSC Algorithm

Precisely computing the compaction metric. To test the MSC metric, we first implement a policy that precisely scores all candidate key ranges and selects the one with highest MSC score. We

also introduce a strawman policy, random-selection, that randomly selects a candidate key range and moves its cold objects to flash. Figure 5 shows the comparison of precise metric with the strawman.

Since the random-selection policy is unaware of compaction benefit or cost, it can choose key ranges that contain fewer cold objects and incur higher write amplification on flash. We observe that the precise-MSC metric can decrease flash write I/O by more than 2.5× compared to the random-selection policy. However, it has worse overall throughput. This is because computing MSC precisely in Equation 1 requires checking the popularity of each object in the mapper, and navigating the indices of all items in the candidate range both in the DRAM B-tree and in the SST file indices, to check for overlaps. This is CPU intensive and leads to long compaction time (25 seconds, compared to the random-selection policy's 1.7 seconds), during which PrismDB has to rate-limit foreground client writes until it frees enough space on NVM.

Approximating compaction metric. Therefore, we propose a light-weight metric called approx-MSC that efficiently approximates the value of MSC. Instead of computing the statistics for each individual object in the candidate key range, approx-MSC metric breaks each partition's key space into smaller, fixed-sized ranges, which we call buckets. It then keeps tracks of approximate values related to p, o, and F for each bucket. approx-MSC can be computed using a weighted sum of the parameters of each bucket that overlaps with the candidate key range. We describe the implementation of buckets in §6.

Figure 5 shows that approx-MSC achieves high throughput while keeping total flash write I/O low (i.e., nearly same as precise-MSC's). Since its approximation takes less time and fewer CPU resources, it reduces average compaction time from 25 seconds to 1.7 seconds, close to the random-selection policy. Therefore, we use approx-MSC in PrismDB, and henceforth in the paper when we refer to MSC it refers to approx-MSC.

Key range selection. Another important design question is which NVM key ranges should be considered as candidates for compaction. One simple approach is to enumerate over all possible NVM key ranges to decide which to compact, but this is impractical for large databases. We use power-of-k choices [48] to select a subset of compaction key ranges as candidates. We empirically use $k = 8$, as it provides a good trade-off between throughput and flash I/O.

Object promotions under dynamic workloads. During the compaction process, objects can be both promoted and demoted. Demotions have an obvious benefit: they free up space in NVM. Promotions, on the other hand, are more expensive, since they take up space in NVM for an object that was previously stored in flash. However, sometimes objects also need to be promoted from flash to NVM, in order to enable fast reads for popular objects, which at some point got demoted. For example, this would occur when a large burst of newly-written objects fills up NVM. Such bursts have been observed in production workloads [11, 14, 76].

Read-triggered Compactions. By default, compactions are triggered when NVM fills up. However, in read-heavy workloads, NVM will only slowly fill up, and the write-triggered compaction process may not be called frequently enough to keep up with changing read popularity distributions. Therefore, PrismDB also employs

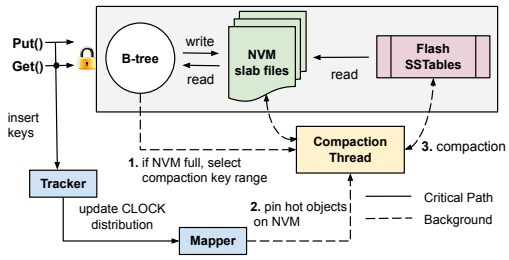


Figure 6: PrismDB's system components.

read-triggered compactions for moving objects between NVM and flash.

The main goal of read-triggered compactions is to improve the ratio of reads served from NVM. Read-triggered compactions have three stages: detection, invocation, and monitoring. In the detection stage, PrismDB checks if the write-triggered compactions aren't keeping up with popularity distributions, by detecting read-dominated workloads where a large proportion of the keys in the tracker are stored on flash. If so, it triggers compactions for an epoch (by default 1 M client operations). At the end of each epoch, the monitoring stage tracks the ratio of reads served from NVM vs. flash. If the compactions triggered in the previous epoch improved that ratio above a threshold (default 1%), it continues the compactions for next epoch. If not, it enters a cool-down period (default 10 M operations). At the end of that period, read-triggered compactions resume again from the detection stage.

6 IMPLEMENTATION

In this section, we describe PrismDB's implementation. PrismDB is written in C++. It is built on top of Google's B-tree [26], the SST file format from LevelDB [27] and the slab implementation for NVM from KVell [39]. Figure 6 depicts PrismDB's principal components. Every partition in PrismDB runs two threads, both of which synchronize using a single partition lock. A worker thread (depicted as a straight line) handles foreground client operations, including looking up, reading objects and writing data to NVM. It is also responsible for tracking object popularity and its distribution in the tracker and mapper. A compaction thread (depicted as a dashed line) runs in the background and is triggered intermittently to free space on NVM and re-balance data popularity so hot objects are always stored on NVM.

Interface. PrismDB supports 4 types of operations: `Put(k,v)`, `Get(k)`, `Delete(k)` and `Scan(k,n)`. Client requests are forwarded to the appropriate partition worker based on the operation key. The worker thread always acquires the partition lock before processing the request and releases it at the end.

`Put(k,v)` writes the key k with value v to NVM. The worker thread checks if the partition's B-tree index has that key. If the key is not present, it selects the slab file based on the object size, and inserts the object to a free slot location within the slab. Then it inserts the key and its disk location in the partition's B-tree index. If the key is present, the worker thread checks if the new object size still fits within the original slab size range. If it does, it performs an in-place update to the same disk location. Otherwise, it deletes the

object from the current slab and performs a fresh insert to the new slab. Then it updates the object's B-tree index entry with new disk location. Lastly, the worker thread updates the key popularity in the tracker.

`Get(k)` returns the most recent version of the key if found. The worker thread checks if the key is in the B-tree index. It the key is present, it reads the object from the slab file on NVM. Otherwise, it checks the index and filter blocks of the SST files and reads the object from flash. Lastly, the worker thread updates the key popularity in the tracker.

`Delete(k)` first looks up the key. If the key exists in NVM, it is deleted from the B-tree index and its slab slot is reclaimed. If the key is present on flash, the worker thread performs a fresh insert to NVM with a special delete tombstone entry. Eventually, both the NVM tombstone and the flash object will be deleted when compaction merges them.

`Scan(k,n)` fetches the next n objects with keys equal to or greater than k. The worker thread runs a two-level iterator, one on NVM objects and the other on flash objects. In every iteration, it compares the key values of objects pointed by the two iterators and selects the smaller object, and then moves the corresponding iterator pointer forward. For range queries that span multiple partitions, PrismDB locks and scans one partition at a time.

Tracker and mapper. The tracker is built on the concurrent hash map implementation from Intel's TBB library [31]. The hash-map index is the object key and each index stores a 1-byte value - two bits for clock and one bit for object location (NVM or flash). When a database client calls `Get` or `Put`, the worker thread inserts the key into the tracker. An insert operation can also invoke an eviction. Keys are initially inserted into the tracker with a value of 0 (min popularity), and keys accessed afterwards have their value set to 3 (max popularity). During eviction, clock values get decremented as per the clock algorithm, and keys with clock values of 0 are evicted. The clock value is stored as an atomic variable. Thus, looking up a clock value does not need to be serialized with eviction. The mapper is implemented as an array of four atomic integers; each tracks the number of keys with a particular clock value.

Compaction thread. Once triggered, the compaction thread first acquires the partition lock. It uses MSC to select a compaction key range and only picks unpopular objects for compaction. Next, the compaction thread reads these objects from NVM and the overlapped SST file(s) to memory, merge-sorts them and rewrites live objects as new SST file(s) back to flash. This stage consumes the most time in the compaction (seconds). Therefore, the compaction thread releases the lock before merging the files. We use a reference counting scheme, similar to RocksDB [64] to track live SST files in flash. This guarantees that compaction doesn't delete a SST file that is being used by a concurrent `Get` or `Scan` iterator. Concurrent client writes can update a compacted object with a more recent version. The compaction thread ensures data correctness by reacquiring the partition lock to check if the version on NVM has changed, and if it has, it skips deleting that item. To do so, PrismDB uses a lightweight compaction bitmap to track whether a object has changed its value since compaction.

MSC metric. By default, the approx-MSC metric uses a bucket size of 64K keys, which is equal to the average number of keys in an SST file. The global key space is divided into consecutive buckets. Every bucket contains four fields: `num_nvm_keys` (a counter of the number of keys present on NVM), `pop_bitmap` (a bitmap of key popularity), `nvm_bitmap` (a bitmap of keys on NVM), and `flash_bitmap` (a bitmap of keys on flash). Access to these fields is protected by the partition lock.

Table 4: YCSB workload description.

Type	Description (read%, update%)
write heavy	A (50%, 50%), F (50%, 50% read-modify-writes)
read heavy	B (95%, 5%), C (100%, 0%), D (latest; 95%, 5%)
scan heavy	E (95% scans, 5% updates)

`Puts` increment `num_nvm_keys` in the corresponding bucket. Compaction decrements it for each overlapped bucket since it knows exactly which keys to remove from NVM after identifying all unpopular keys in the selected compaction key range. `pop_bitmap` provides much faster access than looking up objects in the B-tree and the mapper. `Gets` set the key's bitmap value to 1, while evictions from the tracker set the value to 0. Keys with value 0 are treated as cold objects and have a coldness value of 1. This way, `pop_bitmap` approximates a key's popularity and coldness score without retrieving the accurate clock value from the mapper. `nvm_bitmap` tracks keys present on NVM. `Puts` set the key's bit to 1, while compactions set it to 0. `flash_bitmap` tracks if a key has any version (latest or older) present on flash. Compactions set the key's bit to 1, while `Deletes` set it to 0. `nvm_bitmap` and `flash_bitmap` piece together information about overlapped keys. AND-ing the two bitmaps and counting the number of bit 1 gives the total keys that exist on both NVM and flash.

Given a compaction key range, we sum over weighted parameters from each overlapped bucket to estimate the values of p, o, F, and coldness, and to compute an MSC score for the compaction key range. The weight of each bucket equals to the ratio of the overlapped region to bucket size. A concrete example of how MSC metric is computed is provided in §B.1.

Isolation and crash consistency PrismDB guarantees atomicity of individual writes and provides read-committed isolation (default in RocksDB, PostgreSQL). The current version doesn't support atomic batched writes, transactions and snapshots. We leave that for future work. PrismDB does not use a write-ahead log for crash recovery. Instead, client writes are committed synchronously to their NVM slab locations. PrismDB ensures crash consistency on NVM using a logical timestamp entry. It is part of the object metadata and is synchronously written to disk along with the object. See §B.2 for details.

7 EVALUATION

We evaluate PrismDB by answering the following questions:

(1) How does PrismDB compare to baselines (§7.1)?
(2) Which workloads does PrismDB benefit (§7.2, §7.3)?
(3) How do compactions impact performance (§7.4)?
(4) How to set the pinning threshold (§7.4)?

Configuration. We performed our experiments on a 32-core, 64 GB RAM machine running Ubuntu 18.04. Intel Optane SSD P5800X (NVM), Intel 760p (TLC NAND) and Intel 660p (QLC NAND) are

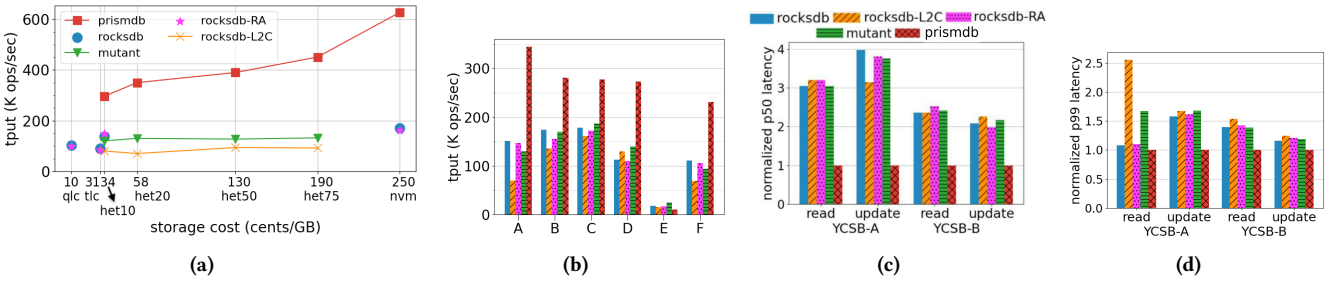


Figure 7: (a) Throughput vs. storage cost under YCSB-A. "hetX" means heterogeneous or multi-tier setup with X% on NVM, (b) YCSB Throughput, (c) YCSB normalized median latency, and (d) YCSB normalized p99 latency.

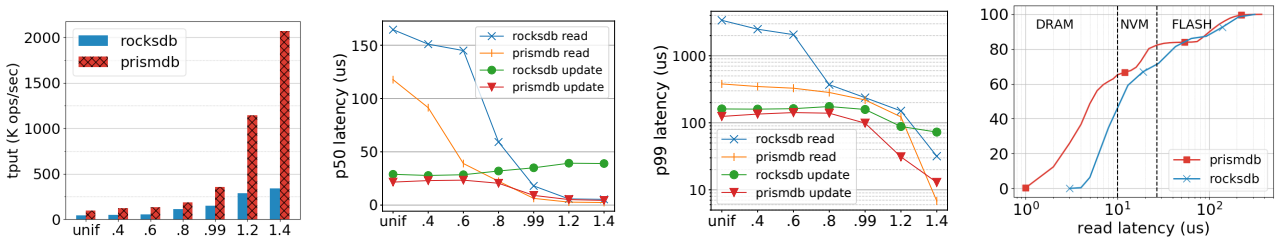


Figure 8: (a-c) YCSB-A performance with different Zipfian parameters, and (d) Read latency cdf on YCSB-B

locally attached to this machine. All baselines are evaluated using 8 clients. PrismDB uses 8 partitions (except Figure 11c). Each partition uses 2 threads, a foreground thread to handle reads/writes and clock-related operations and a background thread for compaction (Figure 6). The key space is range-partitioned into equal partitions. LSM based baselines are configured with 8 background threads.

We use a CPU cgroup of 10 cores in our experiments and set the memory cgroup to one-tenth of database size. 20% of the memory is used as block cache for LSM systems (as is standard [20]). By default, we use only 10 CPU cores, because PrismDB is able to fully utilize Optane SSD's bandwidth with only 10 cores. However, LSM-based baselines bottleneck on CPU and underutilize Optane SSD's I/O even with only 10 cores. For PrismDB, we set the tracker size to 20% of the total key space and pinning threshold to 70%. Other settings are default ones used in RocksDB.

Workloads. We run YCSB (Table 4) and three Twitter production workloads [76]. For YCSB, by default we use YCSB-A with Zipfian 0.99 distribution, 100 M key dataset and a fixed 1 KB object size (total database size of 100 GB). For the update-heavy workloads (A and F) the first half of the trace is used as a warm-up period. For read-heavy workloads (B, C and D), we run 300 M requests with a longer warm-up (80%) to allow the promotion-based compactions to take effect. The scan workload is run with 10 M requests and a 50% warm-up.

We choose three representative Twitter traces [76] with varying read to write ratios and distributions. Write heavy trace (cluster39) has 6:94 read write ratio (uniform writes). Mixed trace (cluster19) has 75:25 read write ratio (zipfian reads, uniform writes). Read intensive trace (cluster51) has 90:10 read write ratio (zipfian reads and writes).

Baselines. We compare PrismDB against five baselines: RocksDB (v6.2.0), RocksDB that uses NVM as an L2 read cache (labeled rocksdb-l2c), our initial read-aware LSM prototype from §3 (labeled as rocksdb-RA), and two academic LSM KV stores, Mutant [79] and SpanDB [13]. Mutant is a storage layer for LSMs that tracks access popularity of SST files and places them across heterogeneous storage accordingly. SpanDB uses SPDK [69] on the Optane drive to bypass the kernel's I/O overheads. Since SpanDB by design bypasses the page cache and persists all WAL writes to disk synchronously, we evaluate it separately by enabling fsync mode in RocksDB and PrismDB to persist all writes synchronously (RocksDB by default does not fsync the WAL). All figures in evaluation section, except Figure 9, use the default RocksDB setting of asynchronous WAL writes.

We considered comparing against PebblesDB [55], an LSM-based KV store that trades read performance for write throughput. However, in our experiments, RocksDB consistently outperformed a tuned PebblesDB on all YCSB workloads (also observed by others [25]). Also, PebblesDB does not have support for tiered storage, so we do not use it as a baseline. All multi-tier experiments use 20% NVM allocation (except in Section 7.1).

7.1 Single-tier vs. Multi-tier

Figure 7a compares the average throughput and storage cost between PrismDB and baseline systems under seven configurations: three single-tier configurations (NVM, TLC, and QLC) and four multi-tier configurations (het). Note that since default RocksDB places data in different storage types on a level granularity, we cannot create a configuration that will match every point on the X axis with a fixed LSM tree shape.

As expected, for regular RocksDB, NVM outperforms single-tier TLC NAND and QLC NAND setup, which uses denser and slower flash. Surprisingly, the QLC setup slightly outperforms TLC, which we attribute to the internal cache on the newer QLC device. Across

the multi-tier setups, the higher the proportion of NVM, the better the performance. Interestingly, the L2 cache configuration of RocksDB consistently underperforms the level-by-level configuration. The reason for this is that the L2 cache serves as a read cache, but not a write cache. All writes go to flash, which is slow. Additionally, the L2 cache needs to load frequently updated objects from flash to NVM which degrades system throughput.

PrismDB significantly outperforms the baselines on all multi-tier storage setups. Notably, the *het10* configuration with PrismDB has 3.3× better throughput and 2× lower tail latency than RocksDB with pure TLC, which is the default way RocksDB is deployed in datacenters today, while costing almost the same ($0.34/GB vs. $0.31/GB). Note these prices fluctuate on a daily basis. Mutant's performance is equivalent to RocksDB's, because it maps popular files to NVM, this comes at the cost of triggering more compactions, especially for write-dominated workloads like YCSB-A. For example, Mutant incurs 59% more background compactions on this workload. In addition, Mutant's mapping is coarse grained, since it makes placement decisions on a file-by-file basis, and a single file may have objects with varying popularity. Read aware RocksDB suffers from the trade-off between pinning and compaction efficiency. Performance gains from serving more reads from NVM are negated by increased compactions, thus offering little to no throughput or latency advantage.

7.2 Multi-tier Storage

YCSB Sweep. Figure 7 (b-d) compares the throughput, median and 99th percentile latencies across the different YCSB workloads. Mutant slightly outperforms RocksDB on read-friendly workloads (YCSB-C and YCSB-D) because it can map popular SST files to NVM better under low compaction churn. RocksDB's L2 cache configuration performs better than RocksDB in limited workloads (YCSB-D only). PrismDB consistently outperforms all baselines for point queries, in terms of throughput and tail latency, due to its more efficient data layout and compaction algorithm. Notably, it is able to outperform the baselines even for YCSB-B and YCSB-C, which are read-heavy and read-only, respectively. In these workloads, PrismDB's promotions get triggered to migrate hot data that may have been compacted in the past to flash. However, PrismDB offers the biggest performance improvement for workloads that include a significant percentage of writes (e.g., YCSB-A). For that workload, PrismDB saves more than 1.9× CPU and does more useful work compared to LSM-based stores that still need to run traditional compactions on NVM.

The only YCSB workload where PrismDB does not outperform the baselines is the scan workload. RocksDB in particular is optimized for scans. It uses a sophisticated prefetcher that proactively fetches blocks, which greatly improves its performance for predictable scan patterns. When we disable RocksDB's prefetcher, we find both systems exhibit comparable performance under scans. We leave implementing a prefetcher for PrismDB as future work.

PrismDB uses clock based LRU for the tracker, as it is a common caching policy used by many systems e.g., RocksDB's block cache. However, clock based LRU policy is not ideal for every workload. Large workloads like scans that access many objects in the database

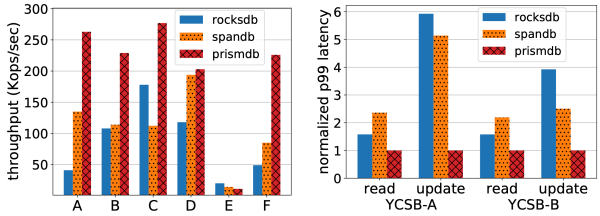


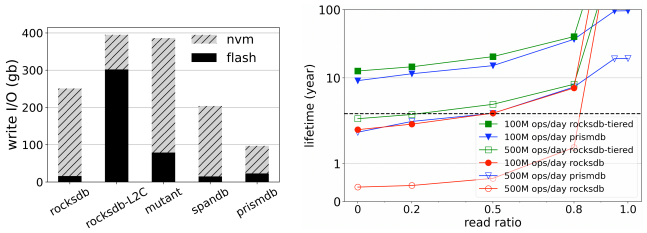
Figure 9: Performance with fsync enabled.

can cause tracker to evict keys often. Since PrismDB's scan implementation locks a partition and then reads the objects, if an object in the scan range was evicted by the tracker, that object won't be migrated to flash till the scan query on that partition completes. Therefore, tracker evictions have no impact on concurrent scan queries. However, future queries may be impacted by such evictions. Prior work [42, 80, 81] has explored techniques specific to improving scan performance. PrismDB by default is optimized for point query lookups. We leave the optimization of pinning popular scan ranges on NVM for future work, and we plan to explore some of the existing techniques, such as REMIX [81] which implements a cache optimized for range queries.

Data skewness. Next, we evaluate how data skewness impacts PrismDB. Figure 8 presents the results of a key distribution sweep using YCSB-A. PrismDB is able to provide a throughput benefit under all distributions. For highly skewed workloads, due to popularity tracking and data pinning, PrismDB serves more reads from DRAM or NVM and absorbs more writes as in-place updates in NVM, delivering superior performance. For uniform workloads, due to PrismDB's partitioned design and NVM data layout, it has much lower tail latency compared to RocksDB. RocksDB does excessive compactions across levels which are known to severely impact tail latency [6].

To better understand why PrismDB provides better read latency, we plot a CDF comparing the latencies of PrismDB to RocksDB on read-heavy YCSB-B (Figure 8d). As expected, the figure shows that PrismDB serves a lower percentage of its requests from flash compared to RocksDB due to better data placement. Interestingly, PrismDB also more efficiently utilizes DRAM. Over time, NVM blocks in PrismDB become more densely packed with popular objects, thus improving the hit rate of the OS page cache. In contrast, RocksDB caches data blocks in DRAM with mixed popularity and further "pollutes" much of its DRAM cache with compactions.

Performance with fsync. SpanDB's design enables faster synchronous WAL logging that benefits write heavy workloads. PrismDB outperforms RocksDB by upto 6.4 × and SpanDB by upto 2.5 × respectively (Figure 9a) on write heavy workloads (YCSB-A and YCSB-F). RocksDB's group commit protocol creates a bottleneck for logging on Optane, which SpanDB alleviates by using SPDK for logging. However, PrismDB's partitioned approach completely removes the centralized logging bottleneck. Further, like RocksDB, SpanDB spends significant CPU time on compactions to keep data sorted on LSM levels resident on Optane. Due to its use of SPDK, SpanDB requires the application to dedicate cores to busy-poll I/O, which is inefficient when CPU is a bottleneck. On YCSB-D, SpanDB



(a) (b)

Figure 10: (a) Total write I/O of different systems under YCSB-A. (b) PrismDB has similar lifetime to Rocksdb while offering higher performance.

Table 5: Performance on Twitter workloads.

Trace	Throughput (ops/s) RocksDB	Throughput (ops/s) PrismDB	Avg put latency RocksDB	Avg put latency PrismDB
write-heavy (cluster39)	236 K	337 K	35.3 μs	24.6 μs
mixed (cluster19)	245 K	248 K	25.4 μs	15.0 μs
read-heavy (cluster51)	798 K	2620 K	22.9 μs	5.4 μs

has comparable performance to PrismDB due to its use of large memtables (1 GB), which aids in fast access to objects with latest distribution. SpanDB also performs slightly better on the scan workload (YCSB-E) compared to PrismDB since it leverages the same sophisticated prefetcher as used in RocksDB.

QLC Endurance. So far, we have presented storage costs in terms of $/GB of capacity. But for lower endurance dense flash (e.g., QLC NAND), the application write-rate is another factor that determines the total cost of ownership (TCO), in case the drive wears out and needs to be replaced before its intended lifetime (typically 3 years). Thus, PrismDB's lifetime heavily depends on how quickly its flash component wears out.

Overall, PrismDB has the lowest total write I/O compared to all baselines (Figure 10a). Its write I/O on flash is slightly higher than RocksDB, which is due to the trade-off of pinning objects on NVM. Figure 10b evaluates the lifetime of three systems with similar storage costs: PrismDB, tiered RocksDB, and homogeneous RocksDB running on TLC SSD (standard datacenter deployment configuration today) under different workload settings. We assume a reasonable DB size seen in real deployment [20], 600 GB. We use two extremes of system load numbers reported from real production systems [11], 100 M ops/day from UP2X and 500 M ops/day from UDB, and compute expected lifetime of each system over a range of read/write ratios under such loads. Most real-world workloads are read-dominated (e.g., 99.8% requests in TAO are reads) [10, 11, 19]. PrismDB with QLC flash is a good fit for such systems. Even for update-heavy workloads with read ratio as low as 0.5, PrismDB exceeds the 3 year lifetime period. Update-mostly workloads with high request rate, on the other hand, will wear out the QLC flash before the 3 year period, and thus incur a higher TCO. For such workloads, system administrators can opt for higher endurance flash like TLC NAND.

7.3 Twitter Production Workloads

Table 5 compares PrismDB with RocksDB on three Twitter production traces, that correspond to three different workload patterns. PrismDB has 43% higher throughput for the insert-heavy workload (cluster39), due to its optimized writes and compactions. On the mixed workload (cluster19) it does not show any improvements. We identify two reasons for this. First, cluster19 workload has highly cacheable reads that fit in memory, so PrismDB's read optimizations don't have any impact. Second, this trace contains tiny objects (102 B avg size) which creates an interesting performance challenge, because of PrismDB's reliance on the OS page cache, which reads from or writes to NVM at a 4 KB granularity. We therefore added a small optimization that sorts free NVM slab slots by their disk locations, which ensures consecutive writes go to the same OS page. However compacting tiny objects still incurs 500 K random NVM reads which make compactions slower, resulting in the overall throughput of PrismDB to be similar to RocksDB's. PrismDB can be further optimized for tiny objects (sizes less than 128B) by using direct IO on Optane and bypassing the OS page cache. We leave the optimization for tiny objects as future work. On read heavy workload (cluster51), PrismDB exhibits a 3.2× improvement in throughput due to larger objects (370 B avg size) and Zipfian reads and writes, for which PrismDB's read and write optimizations are important.

7.4 Evaluating System Components

We now evaluate different components of PrismDB. Figure 11a shows the effect of promotions (moving data from QLC to NVM) under a read-only workload (YCSB-C). The figure shows that promotions significantly increase the ratio of reads from NVM, resulting in higher overall throughput. Figure 11b shows the throughput of three YCSB workloads for varying pinning threshold. Pinning threshold is calculated as a percentage of the tracker size. As seen in the figure, ideal pinning threshold is different for each workload. For a read-heavy workload, a higher threshold is better, since it is more important to retain popular keys even at the expense of less-efficient garbage collection. As expected, the reverse is true for write-heavy workloads. Automatically choosing the right pinning threshold for dynamic workloads can be done via a hill climbing approach, which we leave for future work. Figure 11c shows how PrismDB scales as a function of partitions (or clients). PrismDB continues to scale until it exceeds 10 partitions, which is equal to the number of cores in our experiment.

Figure 11d shows the effect of background compactions on throughput. Periods of high compaction activity (e.g., at 10, 30 and 60 seconds) correlates with a dip in throughput. When compaction activity is more spread out (100-150 seconds), throughput dips are less severe. As a future optimization, PrismDB can use a compaction scheduler (similar to [5, 6]) to spread compaction work over time such that applications using PrismDB can achieve more predictable performance.

7.5 Evaluation Summary

Our evaluation demonstrates the benefits of using multiple tiers of storage, and demonstrates that PrismDB's design, which is tailored to multiple storage tiers, is superior to existing LSM-based systems. PrismDB outperforms other baselines across most workloads, both synthetic and real-world, under both relaxed and strict durability requirements (i.e., with fsync enabled). In particular,

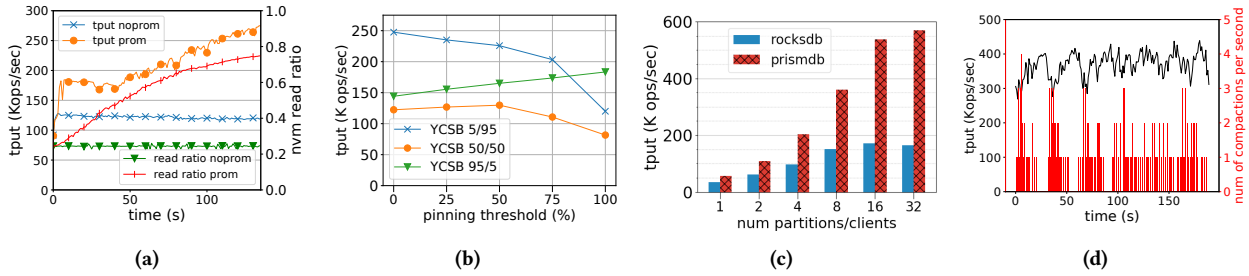


Figure 11: (a) Impact of promotions on read-only YCSB-C. (b) Impact of pinning threshold. (c) Throughput as function of number partitions/clients. (d) Effect of compactions on throughput (YCSB-A).

PrismDB's techniques for popularity tracking and object pinning improve both read and write performance in Zipfian workloads. In addition, PrismDB's novel cost-benefit compaction metric (MSC) achieves better hot-cold separation across storage tiers without incurring high write amplification on flash. We show that even a low-endurance flash drive like QLC NAND, when used in a multi-tier configuration under PrismDB, can meet the 3 year lifetime requirement for many common datacenter workloads. PrismDB is currently designed for two (logical) tiers: a fast random-access tier, and a fast sequential-access tier. These two logical tiers could contain multiple physical storage tiers. For example, for even higher endurance, the QLC tier of PrismDB could be replaced by a smaller upper level of TLC and a bottom level of QLC.

Limitations. PrismDB has a few limitations: (1) Write heavy workloads with high request loads are not suitable for the lower-endurance QLC NAND. For such workloads, PrismDB should be provisioned with higher endurance flash; (2) The current implementation of PrismDB has sub-optimal scan performance than other baselines due to lack of a sophisticated prefetcher; (3) Tiny objects (sizes less than 128 B) incur high read amplification during compactions since compacting a single NVM object requires reading a 4 KB page. Compacting a large number of tiny objects pollutes the OS page cache and degrades overall system performance. One possible solution is to implement an application-level page cache and bypass the OS page cache (not yet implemented in PrismDB); (4) PrismDB lacks transactional support. Adding transactional support to a tiered system like PrismDB is an interesting future direction.

8 OTHER RELATED WORK

We reviewed closely related work in §2. We now describe other related work. SplinterDB [16], KVell [39], and uDepot [35] are databases designed for fast NVMe devices (e.g., Optane SSD). Since they are optimized for NVM, they can avoid the buffering and large compactions employed by LSMs. However, they would not perform well in a tiered setting that includes traditional flash, which requires large contiguous writes.

Past work has explored using different tiers of memory in storage systems [9, 34, 41, 53, 54, 56, 78]. In addition, some key-value stores use persistent memory as a second byte-addressable memory tier [1–3, 33, 40, 44, 50, 65, 66, 75, 77, 82]. Compared to these works, PrismDB explores the fundamental tradeoff between object pinning and compaction overhead when using multiple storage tiers. The biggest delta of our work is a novel compaction algorithm that balances the benefit of retaining frequently-accessed objects on fast storage and the compaction cost incurred on the slow disk with low endurance. To the best of our knowledge, none of the prior work has explored this direction. We focus on tiered block-addressable storage compaction, and our techniques are complementary to a multi-tiered memory setting. Strata [36] is a file system for multi-tiered storage that places data at a file granularity. As our comparison with Mutant demonstrates, managing data placement at the file level is not suitable for datacenter key-value stores, in which each file may contain thousands of small objects of varying popularity. Somewhat related to our work, prior research has explored hot-cold separation of data at the flash translation layer within the SSD's firmware [12, 37, 38, 43, 71].

9 CONCLUSION

By combining multiple storage technologies within the same key-value store we can enable both *fast* and *affordable* storage engines. To achieve these goals, we demonstrated the importance of designing new hybrid data structures and compaction mechanisms, which are tailored for multi-tier storage setup. We believe that the general approach of simultaneously employing a mixture of storage technologies will likely prove useful for other areas of systems.

ACKNOWLEDGMENTS

We thank our shepherd Changwoo Min and the anonymous reviewers for their valuable feedback and suggestions. This research was supported by NSF awards CNS #2106530 and CSR #1763546, ARO award W911NF-21-1-0078, and an Intel equipment donation.

A ARTIFACT APPENDIX

A.1 Abstract

This artifact contains the source code for PrismDB and the necessary scripts to regenerate results in Section 7. It does not contain source code for RocksDB and SpanDB baselines as they are already publicly available on github. The artifact requires a specific storage hardware setup consisting of NVM and NAND flash SSDs. The artifact contains the configuration scripts, to setup the hardware and software environment for running YCSB benchmarks and Twitter workloads on PrismDB, and related output scripts for extracting the results detailed in Section 7.

A.2 Artifact Check-List (Meta-Information)

- **Compilation:** C/C++ code compiled using python scripts.
- **Run-time environment:** Linux - Ubuntu 18.04.6 LTS or higher.
- **Hardware:** NVM SSD (e.g., Intel Optane P5800X), TLC NAND flash SSD (e.g., Intel 760p) and QLC NAND flash SSD (e.g., Intel 660p).
- **Metrics:** Throughput and latency (median and tail).
- **Output:** Stored in output log folder.
- **Experiments:** YCSB benchmark and Twitter production traces.
- **How much disk space required (approximately)?:** 100GB
- **How much time is needed to prepare workflow (approximately)?:** A few minutes to pull the code and setup the environment.
- **How much time is needed to complete experiments (approximately)?:** Each YCSB experiment takes less than an hour.
- **Publicly available?:** Yes
- **Code licenses (if publicly available)?:** Creative Commons Attribution 4.0 International
- **Archived (provide DOI)?:** 10.5281/zenodo.7668541

A.3 Description

A.3.1 How to access. The artifact can be downloaded from https://doi.org/10.5281/zenodo.7668541

A.3.2 Hardware dependencies. An X86 machine with access to NVM SSD (e.g., Intel Optane P5800X), TLC NAND flash SSD (e.g., Intel 760p) and QLC NAND flash SSD (e.g., Intel 660p).

A.3.3 Software dependencies. Standard linux libraries and google-perftools.

A.4 Installation

Download and unzip the source code. Install the dependencies listed in the README.md file.

A.5 Experiment Workflow

(1) Configure the experiment parameters in scripts/config_test_example.yml file
(2) Under scripts folder, execute run.py with desired arguments
(3) Output is stored in scripts/logs/ycsb_log/output.log. Throughput and latency results can be extracted from the output.log

For more details, see README.md file.

A.6 Evaluation and Expected Results

The code and instructions can be used to regenerate Section 7 results for PrismDB system. For baselines RocksDB and SpanDB, similar steps can be run on their respective code repositories which are publicly available.

B APPENDIX

B.1 Calculating MSC Metric

Figure 13 shows a toy example with bucket size of 100 keys and a candidate compaction key range that overlaps with two buckets. Bucket 1 has 75% of its key space overlapping with the compaction key range ([25, 99] shown in grey). Bucket 2 has a 25% overlap ([100,125] shown in yellow). Thus, the weight is 0.75 for Bucket 1, and 0.25 for Bucket 2. Suppose the `num_uniq_keys` in Bucket 1 is 400 and in Bucket 2 is 100, then the estimated number of unique keys on NVM for key range [25, 99] will be 300 (0.75×400) and for key range [100, 125] will be 25 (0.25×100). The total number of NVM keys for the compaction key range is the sum of these weighted numbers from each overlapped bucket, which equals to 325 keys. Other compaction key range parameters are computed in the same manner.

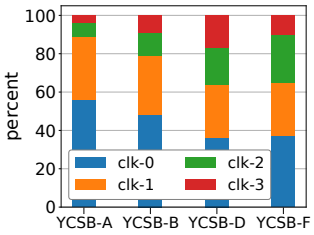


Figure 12: Clock value distributions under different workloads.

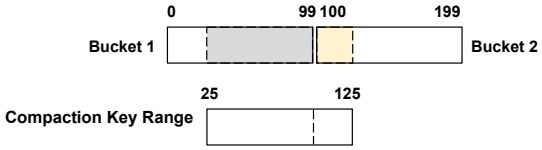


Figure 13: Buckets and compaction key ranges.

B.2 Crash Consistency and Recovery

PrismDB does not use a write-ahead log for crash recovery. Instead, client writes are committed synchronously to their slab locations on NVM. Current implementation of PrismDB only supports objects upto 4 KB in size. Objects are always stored within a single page. Since Optane drives can write 4 KB pages atomically even in case of power failures, PrismDB leverages this capability to perform atomic updates for these sub-page sized objects. For drives that don't support atomic writes to pages, PrismDB can be modified to write updated data in new slab locations. Old slot entries can be reclaimed as soon as the write to the new location succeeds. PrismDB can employ this same technique to support objects larger than 4 KB.

PrismDB ensures crash consistency on NVM using a logical timestamp entry. It is part of the object metadata and is written to disk synchronously along with the object. Each partition maintains its own logical timestamp, which is a counter that is incremented by every `Put` or `Delete` operation for that partition. During recovery, PrismDB scans over all the NVM slabs, and skips key entries with older timestamps when reconstructing the B-tree index in memory. Similar to RocksDB [23], PrismDB keeps an on-disk manifest file that stores the active list of partition SST files for generating a consistent view of the flash database. For keys that exist on both NVM and flash, PrismDB treats the NVM version as the latest version. Finally, partitions run recovery in parallel since they contain disjoint key spaces and don't require any coordination. This further speeds up recovery.

REFERENCES

[1] Shoaib Akram. 2021. Exploiting Intel optane persistent memory for full text search. In *Proceedings of the 2021 ACM SIGPLAN International Symposium on Memory Management*. 80–93.

[2] Joy Arulraj and Andrew Pavlo. 2017. How to Build a Non-Volatile Memory Database Management System. In *Proc. Intl. Conference on Management of Data (SIGMOD)* (Chicago, Illinois, USA). 6 pages. https://doi.org/10.1145/3035918.3054780

[3] Joy Arulraj, Matthew Perron, and Andrew Pavlo. 2016. Write-behind Logging. *Proc. VLDB Endowment* 10, 4 (Nov. 2016), 12 pages. https://doi.org/10.14778/3025111.3025116

[4] Jens Axboe. 2017. *Flexible I/O Tester*. https://github.com/axboe/fio

[5] Oana Balmau, Diego Didona, Rachid Guerraoui, Willy Zwaenepoel, Huapeng Yuan, Aashray Arora, Karan Gupta, and Pavan Konka. 2017. TRIAD: Creating Synergies Between Memory, Disk and Log in Log Structured Key-Value Stores. In *Proc. USENIX Annual Technical Conference (ATC)*. https://www.usenix.org/conference/atc17/technical-sessions/presentation/balmau

[6] Oana Balmau, Florin Dinu, Willy Zwaenepoel, Karan Gupta, Ravishankar Chandhiramoorthi, and Diego Didona. 2019. SILK: Preventing Latency Spikes in Log-Structured Merge Key-Value Stores. In *Proc. USENIX Annual Technical Conference (ATC)*. https://www.usenix.org/conference/atc19/presentation/balmau

[7] Nathan Beckmann, Haoxian Chen, and Asaf Cidon. 2018. LHD: Improving Cache Hit Rate by Maximizing Hit Density. In *Proc. USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. https://www.usenix.org/conference/nsdi18/presentation/beckmann

[8] Burton H. Bloom. 1970. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM* (1970).

[9] Santiago Bock, Bruce R. Childers, Rami Melhem, and Daniel Mossé. 2016. Concurrent Migration of Multiple Pages in software-managed hybrid main memory. In *2016 IEEE 34th International Conference on Computer Design (ICCD)*. 420–423. https://doi.org/10.1109/ICCD.2016.7753318

[10] 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. 2013. TAO: Facebook's Distributed Data Store for the Social Graph. In *Proc. USENIX Annual Technical Conference (ATC)*.

[11] Zhichao Cao, Siying Dong, Sagar Vemuri, and David H.C. Du. 2020. Characterizing, Modeling, and Benchmarking RocksDB Key-Value Workloads at Facebook. In *Proc. USENIX Conference on File and Storage Technologies (FAST)*. https://www.usenix.org/conference/fast20/presentation/cao-zhichao

[12] Li-Pin Chang and Tei-Wei Kuo. 2002. An adaptive striping architecture for flash memory storage systems of embedded systems. In *Proceedings. Eighth IEEE Real-Time and Embedded Technology and Applications Symposium*. 187–196. https://doi.org/10.1109/RTTAS.2002.1137393

[13] Hao Chen, Chaoyi Ruan, Cheng Li, Xiaosong Ma, and Yinlong Xu. 2021. SpanDB: A Fast, Cost-Effective LSM-tree Based KV Store on Hybrid Storage. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*. 17–32.

[14] Asaf Cidon, Assaf Eisenman, Mohammad Alizadeh, and Sachin Katti. 2016. Cliffhanger: Scaling Performance Cliffs in Web Memory Caches. In *Proc. USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. https://www.usenix.org/conference/nsdi16/technical-sessions/presentation/cidon

[15] Asaf Cidon, Daniel Rushton, Stephen M. Rumble, and Ryan Stutsman. 2017. Memshare: a Dynamic Multi-tenant Key-value Cache. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. USENIX Association, Santa Clara, CA, 321–334. https://www.usenix.org/conference/atc17/technical-sessions/presentation/cidon

[16] Alexander Conway, Abhishek Gupta, Vijay Chidambaram, Martin Farach-Colton, Richard Spillane, Amy Tai, and Rob Johnson. 2020. SplinterDB: Closing the Bandwidth Gap for NVMe Key-Value Stores. In *Proc. USENIX Annual Technical Conference (ATC)*. https://www.usenix.org/conference/atc20/presentation/conway

[17] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking Cloud Serving Systems with YCSB. In *Proc. Symposium on Cloud Computing (SoCC)* (Indianapolis, Indiana, USA). 12 pages. https://doi.org/10.1145/1807128.1807152

[18] Fernando J Corbato. 1968. *A paging experiment with the multics system*. Technical Report. DTIC Document.

[19] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, JJ Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. 2012. Spanner: Google's Globally-Distributed Database. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*. Hollywood, CA, 261–264. https://www.usenix.org/conference/osdi12/technical-sessions/presentation/corbett

[20] Siying Dong, Mark Callaghan, Leonidas Galanis, Dhruba Borthakur, Tony Savor, and Michael Strum. 2017. Optimizing Space Amplification in RocksDB. In *Proc. Biennial Conference on Innovative Data Systems Research (CIDR)*. http://cidrdb.org/cidr2017/papers/p82-dong-cidr17.pdf

[21] Assaf Eisenman, Asaf Cidon, Evgenya Pergament, Or Haimovich, Ryan Stutsman, Mohammad Alizadeh, and Sachin Katti. 2019. Flashield: a Hybrid Key-value Cache that Controls Flash Write Amplification. In *Proc. USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. https://www.usenix.org/conference/nsdi19/presentation/eisenman

[22] Assaf Eisenman, Darryl Gardner, Islam AbdelRahman, Jens Axboe, Siying Dong, Kim Hazelwood, Chris Petersen, Asaf Cidon, and Sachin Katti. 2018. Reducing DRAM Footprint with NVM in Facebook. In *Proc. EuroSys Conference* (Porto, Portugal). Article 42, 13 pages. https://doi.org/10.1145/3190508.3190524

[23] Facebook. 2014. *RocksDB*. https://rocksdb.org

[24] Bin Fan, David G. Andersen, and Michael Kaminsky. 2013. MemC3: Compact and Concurrent MemCache with Dumber Caching and Smarter Hashing. In *Proc. USENIX Symposium on Networked Systems Design and Implementation (NSDI)* (Lombard, IL). 14 pages. http://dl.acm.org/citation.cfm?id=2482626.2482662

[25] Eran Gilad, Edward Bortnikov, Anastasia Braginsky, Yonatan Gottesman, Eshcar Hillel, Idit Keidar, Nurit Moscovici, and Rana Shahout. 2020. EvenDB: Optimizing Key-Value Storage for Spatial Locality. In *Proc. European Conference on Computer Systems (EuroSys)* (Heraklion, Greece). Article 27, 16 pages. https://doi.org/10.1145/3342195.3387523

[26] Google. 2011. *Google b-tree implementation*. https://code.google.com/archive/p/cpp-btree/

[27] Google. 2011. *LevelDB*. http://leveldb.org/

[28] Laura M. Grupp, John D. Davis, and Steven Swanson. 2012. The Bleak Future of NAND Flash Memory. In *Proc. USENIX Conference on File and Storage Technologies (FAST)*. 1 pages. http://dl.acm.org/citation.cfm?id=2208461.2208463

[29] Intel. [n. d.]. *Intel Optane SSD DC P5800X Series*. https://ark.intel.com/content/www/us/en/ark/products/201859/intel-optane-ssd-dc-p5800x-series-1-6tb-2-5in-pcie-x4-3d-xpoint.html

[30] Intel. 2019. *Intel Optane memory*. https://www.intel.com/content/www/us/en/architecture-and-technology/optane-memory.html

[31] Intel. 2019. *Intel Thread Building Blocks (TBB) library*. https://software.intel.com/content/www/us/en/develop/tools/threading-building-blocks.html

[32] Shehbaz Jaffer, Kaveh Mahdaviani, and Bianca Schroeder. 2020. Rethinking WOM Codes to Enhance the Lifetime in New SSD Generations. In *Proc. USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage)*.

[33] Olzhas Kaiyrakhmet, Songyi Lee, Beomseok Nam, Sam H Noh, and Young-Ri Choi. 2019. SLM-DB: single-level key-value store with persistent memory. In *Proc. USENIX Conference on File and Storage Technologies (FAST)*.

[34] Jagadish B. Kotra, Haibo Zhang, Alaa R. Alameldeen, Chris Wilkerson, and Mahmut T. Kandemir. 2018. CHAMELEON: A Dynamically Reconfigurable Heterogeneous Memory System. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 533–545. https://doi.org/10.1109/MICRO.2018.00050

[35] Kornilios Kourtis, Nikolas Ioannou, and Ioannis Koltsidas. 2019. Reaping the performance of fast NVM storage with uDepot. In *Proc. USENIX Conference on File and Storage Technologies (FAST)*. https://www.usenix.org/conference/fast19/presentation/kourtis

[36] Youngjin Kwon, Henrique Fingler, Tyler Hunt, Simon Peter, Emmett Witchel, and Thomas Anderson. 2017. Strata: A cross media file system. In *Proc. Symposium on Operating Systems Principles (SOSP)*.

[37] Hyun-Seob Lee, Hyun-Sik Yun, and Dong-Ho Lee. 2009. HFTL: hybrid flash translation layer based on hot data identification for flash memory. *IEEE Transactions on Consumer Electronics* 55, 4 (2009), 2005–2011. https://doi.org/10.1109/TCE.2009.5373762

[38] Sungjin Lee, Dongkun Shin, Young-Jin Kim, and Jihong Kim. 2008. LAST: Locality-Aware Sector Translation for NAND Flash Memory-Based Storage Systems. *SIGOPS Oper. Syst. Rev.* 42, 6 (oct 2008), 36–42. https://doi.org/10.1145/1453775.1453783

[39] Baptiste Lepers, Oana Balmau, Karan Gupta, and Willy Zwaenepoel. 2019. KVell: The Design and Implementation of a Fast Persistent Key-Value Store. In *Proc. Symposium on Operating Systems Principles (SOSP)* (Huntsville, Ontario, Canada). 15 pages. https://doi.org/10.1145/3341301.3359628

[40] Wenjie Li, Dejun Jiang, Jin Xiong, and Yungang Bao. 2020. HiLSM: An LSM-Based Key-Value Store for Hybrid NVM-SSD Storage Systems. In *Proceedings of the 17th ACM International Conference on Computing Frontiers* (Catania, Sicily, Italy) *(CF '20)*. Association for Computing Machinery, New York, NY, USA, 208–216. https://doi.org/10.1145/3387902.3392621

[41] Yang Li, Saugata Ghose, Jongmoo Choi, Jin Sun, Hui Wang, and Onur Mutlu. 2017. Utility-Based Hybrid Memory Management. In *2017 IEEE International Conference on Cluster Computing (CLUSTER)*. 152–165. https://doi.org/10.1109/CLUSTER.2017.130

[42] Siqiang Luo, Subarna Chatterjee, Rafael Ketsetsidis, Niv Dayan, Wilson Qin, and Stratos Idreos. 2020. Rosetta: A Robust Space-Time Optimized Range Filter for Key-Value Stores. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (Portland, OR, USA) *(SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 2071–2086. https://doi.org/10.1145/3318464.3389731

[43] Yixin Luo, Yu Cai, Saugata Ghose, Jongmoo Choi, and Onur Mutlu. 2015. WARM: Improving NAND flash memory lifetime with write-hotness aware retention management. In *2015 31st Symposium on Mass Storage Systems and Technologies (MSST)*. 1–14. https://doi.org/10.1109/MSST.2015.7208284

[44] Virendra J. Marathe, Margo Seltzer, Steve Byan, and Tim Harris. 2017. Persistent Memcached: Bringing Legacy Code to Byte-addressable Persistent Memory. In *Proc. USENIX Conference on Hot Topics in Storage and File Systems (HotStorage)* (Santa Clara, CA). 1 pages. http://dl.acm.org/citation.cfm?id=3154601.3154605

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

[46] Micron. 2018. *The Great Endurance Race: SSDs in One Lane, HDDs in the Other.*

[47] Micron. 2019. *QLC NAND Technology.* https://www.micron.com/products/nand-flash/qlc-nand

[48] Michael Mitzenmacher. 2001. The power of two choices in randomized load balancing. *IEEE Transactions on Parallel and Distributed Systems* 12, 10 (2001), 1094–1104.

[49] 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. 2013. Scaling Memcache at Facebook. In *Proc. USENIX Symposium on Networked Systems Design and Implementation (NSDI)*.

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

[51] S Ohshima and Y Tanaka. [n. d.]. New 3D Flash Technologies Offer Both Low Cost and Low Power Solutions. https://www.flashmemorysummit.com/English/Conference/Keynotes.html.

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

[53] Ivy B. Peng and Jeffrey S. Vetter. 2018. Siena: Exploring the Design Space of Heterogeneous Memory Systems. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*. 427–440. https://doi.org/10.1109/SC.2018.00036

[54] Moinuddin K. Qureshi, Vijayalakshmi Srinivasan, and Jude A. Rivers. 2009. Scalable High Performance Main Memory System Using Phase-Change Memory Technology. *SIGARCH Comput. Archit. News* 37, 3 (jun 2009), 24–33. https://doi.org/10.1145/1555815.1555760

[55] Pandian Raju, Rohan Kadekodi, Vijay Chidambaram, and Ittai Abraham. 2017. PebblesDB: Building Key-Value Stores Using Fragmented Log-Structured Merge Trees. In *Proc. Symposium on Operating Systems Principles (SOSP)* (Shanghai, China). 18 pages. https://doi.org/10.1145/3132747.3132765

[56] Luiz E. Ramos, Eugene Gorbatov, and Ricardo Bianchini. 2011. Page Placement in Hybrid Memory Systems. In *Proceedings of the International Conference on Supercomputing* (Tucson, Arizona, USA) *(ICS '11)*. Association for Computing Machinery, New York, NY, USA, 85–95. https://doi.org/10.1145/1995896.1995911

[57] Mendel Rosenblum and John K Ousterhout. 1992. The design and implementation of a log-structured file system. *ACM Trans. Computer Systems* 10, 1 (1992).

[58] Stephen M. Rumble, Ankita Kejriwal, and John Ousterhout. 2014. Log-structured Memory for DRAM-based Storage. In *12th USENIX Conference on File and Storage Technologies (FAST 14)*. USENIX Association, Santa Clara, CA, 1–16. https://www.usenix.org/conference/fast14/technical-sessions/presentation/rumble

[59] Samsung. 2019. *Samsung Z-SSD Redefining fast responsiveness.* https://www.samsung.com/semiconductor/ssd/z-ssd/

[60] Zhenyu Song, Daniel S. Berger, Kai Li, and Wyatt Lloyd. 2020. Learning Relaxed Belady for Content Distribution Network Caching. In *Proc. USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. https://www.usenix.org/conference/nsdi20/presentation/song

[61] Amy Tai, Andrew Kryczka, Shobhit O. Kanaujia, Kyle Jamieson, Michael J. Freedman, and Asaf Cidon. 2019. Who's Afraid of Uncorrectable Bit Errors? Online Recovery of Flash Errors with Distributed Redundancy. In *Proc. USENIX Annual Technical Conference (ATC)*. https://www.usenix.org/conference/atc19/presentation/tai

[62] Billy Tallis. [n. d.]. The Crucial P1 1TB SSD Review: The Other Consumer QLC SSD. https://www.anandtech.com/show/13512/the-crucial-p1-1tb-ssd-review.

[63] Linpeng Tang, Qi Huang, Wyatt Lloyd, Sanjeev Kumar, and Kai Li. 2015. RIPQ: Advanced Photo Caching on Flash for Facebook. In *Proc. USENIX Conference on File and Storage Technologies (FAST)*. https://www.usenix.org/conference/fast15/technical-sessions/presentation/tang

[64] Tracking live sst files. [n. d.]. https://github.com/facebook/rocksdb/wiki/How-we-keep-track-of-live-SST-files. https://github.com/facebook/rocksdb/wiki/How-we-keep-track-of-live-SST-files

[65] Alexander van Renen, Viktor Leis, Alfons Kemper, Thomas Neumann, Takushi Hashida, Kazuichi Oe, Yoshiyasu Doi, Lilian Harada, and Mitsuru Sato. 2018. Managing Non-Volatile Memory in Database Systems. In *Proc. Intl. Conference on Management of Data (SIGMOD)* (Houston, TX, USA). 15 pages. https://doi.org/10.1145/3183713.3196897

[66] Alexander van Renen, Lukas Vogel, Viktor Leis, Thomas Neumann, and Alfons Kemper. 2019. Persistent Memory I/O Primitives. In *Proc. Intl. Workshop on Data Management on New Hardware (DaMoN)* (Amsterdam, Netherlands). Article 12, 7 pages. https://doi.org/10.1145/3329785.3329930

[67] Carl Waldspurger, Trausti Saemundsson, Irfan Ahmad, and Nohhyun Park. 2017. Cache Modeling and Optimization using Miniature Simulations. In *Proc. USENIX Annual Technical Conference (ATC)*. https://www.usenix.org/conference/atc17/technical-sessions/presentation/waldspurger

[68] Carl A. Waldspurger, Nohhyun Park, Alexander Garthwaite, and Irfan Ahmad. 2015. Efficient MRC Construction with SHARDS. In *Proc. USENIX Conference on File and Storage Technologies (FAST)*. https://www.usenix.org/conference/fast15/technical-sessions/presentation/waldspurger

[69] Benjamin Walker. 2016. SPDK: Building blocks for scalable, high performance storage applications. In *Storage Developer Conference. SNIA*.

[70] Sean Webster. [n. d.]. *Intel SSD 660p.* https://www.tomshardware.com/reviews/intel-ssd-660p-qlc-nvme,5719.html

[71] Chin-hsien Wu and Tei-wei Kuo. 2006. An Adaptive Two-Level Management for the Flash Translation Layer in Embedded Systems. In *2006 IEEE/ACM International Conference on Computer Aided Design*. 601–606. https://doi.org/10.1109/ICCAD.2006.320107

[72] Kan Wu, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. 2019. Towards an Unwritten Contract of Intel Optane SSD. In *Proc. USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage)*. https://www.usenix.org/conference/hotstorage19/presentation/wu-kan

[73] Kan Wu, Andrea Arpaci-Dusseau, Remzi Arpaci-Dusseau, Rathijit Sen, and Kwanghyun Park. 2019. Exploiting Intel Optane SSD for Microsoft SQL Server. In *Proc. Intl. Workshop on Data Management on New Hardware (DaMoN)* (Amsterdam, Netherlands). Article 15, 3 pages. https://doi.org/10.1145/3329785.3329916

[74] Kan Wu, Zhihan Guo, Guanzhou Hu, Kaiwei Tu, Ramnatthan Alagappan, Rathijit Sen, Kwanghyun Park, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2021. The Storage Hierarchy is Not a Hierarchy: Optimizing Caching on Modern Storage Devices with Orthus. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*. USENIX Association, 307–323. https://www.usenix.org/conference/fast21/presentation/wu-kan

[75] Fei Xia, Dejun Jiang, Jin Xiong, and Ninghui Sun. 2017. HiKV: A Hybrid Index Key-Value Store for DRAM-NVM Memory Systems. In *Proc. USENIX Annual Technical Conference (ATC)*. https://www.usenix.org/conference/atc17/technical-sessions/presentation/xia

[76] Juncheng Yang, Yao Yue, and K. V. Rashmi. 2020. A large scale analysis of hundreds of in-memory cache clusters at Twitter. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 191–208. https://www.usenix.org/conference/osdi20/presentation/yang

[77] Ting Yao, Yiwen Zhang, Jiguang Wan, Qiu Cui, Liu Tang, Hong Jiang, Changsheng Xie, and Xubin He. 2020. MatrixKV: Reducing Write Stalls and Write Amplification in LSM-tree Based KV Stores with Matrix Container in NVM. In *Proc. USENIX Annual Technical Conference (ATC)*. https://www.usenix.org/conference/atc20/presentation/yao

[78] HanBin Yoon, Justin Meza, Rachata Ausavarungnirun, Rachael A. Harding, and Onur Mutlu. 2012. Row buffer locality aware caching policies for hybrid memories. In *2012 IEEE 30th International Conference on Computer Design (ICCD)*. 337–344. https://doi.org/10.1109/ICCD.2012.6378661

[79] Hobin Yoon, Juncheng Yang, Sveinn Fannar Kristjansson, Steinn E. Sigurdarson, Ymir Vigfusson, and Ada Gavrilovska. 2018. Mutant: Balancing Storage Cost and Latency in LSM-Tree Data Stores. In *Proc. Symposium on Cloud Computing (SoCC)* (Carlsbad, CA, USA). 12 pages. https://doi.org/10.1145/3267809.3267846

[80] Huanchen Zhang, Hyeontaek Lim, Viktor Leis, David G. Andersen, Michael Kaminsky, Kimberly Keeton, and Andrew Pavlo. 2018. SuRF: Practical Range Query Filtering with Fast Succinct Tries. In *Proceedings of the 2018 International Conference on Management of Data* (Houston, TX, USA) *(SIGMOD '18)*. Association for Computing Machinery, New York, NY, USA, 323–336. https://doi.org/10.1145/3183713.3196931

[81] Wenshao Zhong, Chen Chen, Xingbo Wu, and Song Jiang. 2021. REMIX: Efficient Range Query for LSM-trees. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*. USENIX Association, 51–64. https://www.usenix.org/conference/fast21/presentation/zhong

[82] Xinjing Zhou, Joy Arulraj, Andrew Pavlo, and David Cohen. 2021. *Spitfire: A Three-Tier Buffer Manager for Volatile and Non-Volatile Memory.* Association for Computing Machinery, New York, NY, USA, 2195–2207. https://doi.org/10.1145/3448016.3452819

Received 2022-10-20; accepted 2023-01-19