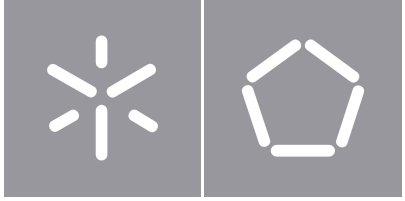


**University of Minho**
School of Engineering

Rúben Daniel Almeida Adão

# Co-designing Log-Structured Merge Key-Value Stores with a Non-Volatile Storage Hierarchy

december 2023



**University of Minho**
School of Engineering

Rúben Daniel Almeida Adão

# Co-designing Log-Structured Merge Key-Value Stores with a Non-Volatile Storage Hierarchy

Masters Dissertation
Master's in Informatics Engineering

Dissertation supervised by
**João Tiago Medeiros Paulo**
**Ricardo Gonçalves Macedo**

december 2023

# Copyright and Terms of Use for Third Party Work

This dissertation reports on academic work that can be used by third parties as long as the internationally accepted standards and good practices are respected concerning copyright and related rights.

This work can thereafter be used under the terms established in the license below.

Readers needing authorization conditions not provided for in the indicated licensing should contact the author through the RepositóriUM of the University of Minho.

## License granted to users of this work:



**CC BY 4.0**

https://creativecommons.org/licenses/by/4.0/

# Acknowledgements

My years of dedicated effort at University of Minho find their conclusion in this dissertation, a testament to the persistent assistance provided by those who have supported me throughout this academic journey.

My heartfelt gratitude to my esteemed supervisors, João Paulo and Ricardo Macedo, whose unwavering support and guidance have been invaluable throughout this academic journey. Many times I have tested their patience and yet they've endured my wild experiments in research and questionable hypotheses with good humor. Their expertise played a pivotal role in shaping the outcome of this dissertation, and I am truly appreciative of the countless hours they dedicated to helping me navigate the complexities of my research.

I would like to thank professor Oana Balmau at McGill University, her profound knowledge and insights have been instrumental in shaping my understanding of the subject of this dissertation. I also extend my gratitude to Changjun Zhou and Zhongjie Wu for their invaluable assistance in this work.

A special note of thanks goes to my family, whose enduring encouragement and belief in my capabilities have been a constant source of motivation. Their unwavering support has been the bedrock upon which I built my academic pursuits, and for that, I am profoundly grateful.

I am also indebted to my friends and roommates, who shared in the highs and lows of my college experience. Their camaraderie created a conducive environment for learning and personal growth. The memories we've created together will forever hold a special place in my heart.

To my colleagues at HASLab, for giving me a taste of the research world and for putting up with my quirks. I am sincerely grateful for the collaborative environment you've created.

This work is financed by National Funds through the Portuguese funding agency, FCT - Fundação para a Ciência e a Tecnologia, within projects LA/P/0063/2020 and UIDP/50014/2020.

This work is co-financed by Component 5 - Capitalization and Business Innovation, integrated in the Resilience Dimension of the Recovery and Resilience Plan within the scope of the Recovery and Resilience Mechanism (MRR) of the European Union (EU), framed in the Next Generation EU, for the period 2021 - 2026, within project ATE, with reference 56.

# Statement of Integrity

I hereby declare having conducted this academic work with integrity.

I confirm that I have not used plagiarism or any form of undue use of information or falsification of results along the process leading to its elaboration.

I further declare that I have fully acknowledged the Code of Ethical Conduct of the University of Minho.

University of Minho, Braga, december 2023



Rúben Daniel Almeida Adão

# Abstract

The trend of increasing size of datasets in storage-based applications has promoted the research of new methods and technologies for efficiently storing, processing, and analyzing large amounts of data. As a result, **Log Structured Merge** (**LSM**) **Key-Value Stores** (**KVSs**) have been highly adopted since their design allows high write throughput and enforces sequential disk access patterns. Additionally, with the advent of **Non-Volatile Main Memory** (**NVMM**), new storage technologies have emerged that offer faster access times compared to traditional block-based storage devices, thus accelerating **KVSs**.

However, while **NVMM** devices offer faster access to data, they are typically limited in capacity and are often more expensive. To address this trade-off, contemporary storage solutions harness the capabilities of heterogeneous storage devices in two fundamental manners: caching and tiering. In this dissertation, we show that, on one hand, read-dominated workloads benefit from a caching approach, but their performance degrades under tiering. On the other hand, for write-dominated workloads, the tiering approach presents better performance, while storing the entire dataset on **NVMM** actually degrades performance.

To overcome these challenges, this dissertation proposes KEIGO, a novel storage middleware that allows **LSM**-based **KVS** to efficiently use storage hierarchies composed of **NVMM** and block-based devices. KEIGO is aware of the different I/O operations done by the **KVS** (*e.g.,* foreground requests, and background flushes and compactions) and the characteristics of the underlying devices (*e.g.,* concurrency, read/write asymmetry). This knowledge serves as a pivotal factor in optimizing KEIGO's performance in the face of dynamic and mixed production workloads such as those observed in Nutanix and Meta. Moreover, KEIGO requires minimal code modifications to integrate into production-ready **LSM KVSs**.

Conducted experiments show that KEIGO significantly enhances the throughput of **LSM KVS** solutions, including RocksDB, Speedb, and LevelDB, by as much as 12.4×. Furthermore, it substantially reduces tail latency by up to 21.3× over both general-purpose storage solutions and **LSM KVSs** built from the ground up for hierarchical storage.

**Keywords** Key-value store, NVMM, persistent memory, hierarchical storage

# Resumo

O aumento do volume de dados gerido por aplicações orientadas ao armazenamento tem promovido a investigação de novos métodos e tecnologias para os armazenar, processar e analisar eficientemente. Como resultado, os sistemas de armazenamento chave-valor (**KVS**) baseados em **Log Structured Merge** (**LSM**) têm sido amplamente adotados, já que o seu desenho permite um elevado débito de escritas e impõe padrões de acesso sequenciais. Além disso, com o desenvolvimento de dispositivos de memória não volátil (**NVMM**), surgiram novas técnicas de armazenamento que oferecem tempos de acesso mais rápidos comparativamente aos discos tradicionais baseados em blocos, acelerando assim o desempenho das **KVSs**.

Contudo, embora os dispositivos **NVMM** ofereçam melhor desempenho, são normalmente limitados em termos de capacidade e mais caros. Para resolver este problema, as soluções de armazenamento atuais tiram partido das capacidades de armazenamento heterogéneo de duas formas: *caching* e *tiering*. Nesta dissertação demonstramos que, por um lado, as cargas de trabalho dominadas por leituras beneficiam de *caching*, mas o seu desempenho degrada com as soluções de tiering. Para as cargas de trabalho dominadas por escritas, a abordagem de *tiering* é mais benéfica, tendo em conta que o armazenamento de todo o conjunto de dados em **NVMM** degrada o desempenho.

De forma a resolver estes problemas, esta dissertação propõe um novo *middleware* de armazenamento, KEIGO, que ajuda **KVSs** a utilizar eficientemente armazenamento hierárquico composto por **NVMM** e outros dispositivos orientados ao bloco. O KEIGO tem conhecimento das diferentes operações de I/O efectuadas pela KVS (*p.ex.,* pedidos de leitura e escrita, *flushes* e compactações) e das características dos dispositivos subjacentes (*p.ex.,* concorrência, assimetria de leitura/escrita), permitindo otimizar o desempenho face a cargas de trabalho de produção dinâmicas e mistas, como as da Nutanix e da Meta.

Através de uma avaliação compreensiva, mostramos que o KEIGO melhora o débito de **LSM KVSs**, incluíndo RocksDB, Speedb e LevelDB, em até 12.4×. Reduz ainda a latência em até 21.3× em relação a soluções genéricas e a **KVSs** criadas especificamente para armazenamento hierárquico.

**Palavras-chave** Armazenamento chave-valor, NVMM, memória persistente, armazenamento hierárquico

# Contents


**1 Introduction** **1**

1.1 Problem . . . 2

1.2 Objectives . . . 3

1.3 Contributions . . . 4

1.4 Document Structure . . . 4

**2 State of the art** **6**

2.1 Background . . . 6

2.1.1 Key-Value Stores . . . 6

2.1.2 The Log-Structured Merge tree . . . 7

2.1.3 Persistent Memory (NVMM) . . . 13

2.1.4 Heterogeneous Storage . . . 19

2.2 Related work . . . 20

**3 Motivational Experiments** **25**

3.1 Placement of LSM components . . . 26

3.2 Popularity of LSM levels . . . 27

3.3 Concurrent compactions . . . 28

3.4 Lessons learned . . . 29

**4 KEIGO: LSM-aware hierarchical storage** **31**

4.1 KEIGO design overview . . . 32

4.2 Optimizing writes . . . 34

4.3 Optimizing reads . . . 35

4.4 Context-based I/O differentiation . . . 37

4.5 Implementation . . . 38

**5 Evaluation** **43**

5.1 Microbenchmarks: YCSB . . . . . 44

5.2 Production workloads . . . . . 48

5.3 KEIGO portability . . . . . 49

5.4 Evaluation Summary . . . . . 50

**6 Conclusion** **51**

6.1 Future Work . . . . . 52

# List of Figures


1 Examples of in-place and out-of-place update structures: each entry contains a key (*K*) and a value (*V*). . . . . 8

2 Original LSM-tree design. . . . . 9

3 Partitioned leveling merge policy. . . . . 10

4 Persistent memory as block storage. . . . . 15

5 NVMM-aware file system. . . . . 16

6 I/O workflow using PMDK. . . . . 17

7 Caching and Tiering. Caching copies data items from the capacity device (CD) to the performance device (PD). Tiering splits access to each layer and migrates items in the background. . . . . 19

8 **Performance of placing LSM components across different storage devices.** *Experiments conducted over RocksDB with both write-intensive (YCSB A) and read-intensive workloads (YCSB C). In ext4 and ext4-DAX, the dataset is fully stored in NVMe* **SSD** *and* **NVMM** *devices, respectively. The remainder setups are conducted with a combination of* **NVMM** *and NVMe* **SSD**. . . . . 26

9 **Number of foreground reads performed across LSM levels.** *Experiments were conducted using YCSB C over a 200M key-value pairs dataset under zipf 0.99, zipf 0.80, and uniform.* . . . . . 28

10 **Concurrent compactions under different placement schemes.** *Experiments were conducted with a fixed number of client (8) and flushing (1) threads and varying compactions threads (1–16). The title of each plot depicts the components that are placed on* **NVMM**. *The remainder levels are placed on the NVMe* **SSD**. . . . . 29

11 **KEIGO architecture.** KEIGO *follows a multi-tier storage structure driven by the design and workload properties of LSM KVS. The performance device handles* **NVMM** *operations (*e.g., `pmem_map_file()`, `pmem_mem_copy()`, `pmem_flush()`*, while the capacity device handle regular* **POSIX** *operations (*e.g., `open()`, `read()`, `fsync()`*).* 32

12 **Single-threaded YCSB workloads.** *We compare RocksDB over ext4-DAX, OpenCAS, and* KEIGO*, with the standalone NoveLSM and MatrixKV.* . . . . . 45

13 **Multi-threaded (8 client threads) YCSB workloads.** *We compare RocksDB over ext4-DAX, OpenCAS, and* KEIGO*, with the standalone NoveLSM and MatrixKV.* . . . . . 45

14 **Durable $C_{log}$ writes.** *We compare RocksDB over ext4, ext4-DAX, OpenCAS, and* KEIGO*, with MatrixKV under YCSB A.* . . . . . 46

15 **200 GB dataset (8 client threads) YCSB workloads.** *We compare RocksDB over ext4, OpenCAS, and* KEIGO*, under NVMM+NVMe* **SSD** *and NVMM+SATA* **SSD** *storage setups.* . . . . . 47

16 **400 GB dataset (8 client threads) YCSB workloads.** *We compare RocksDB over ext4, OpenCAS, and* KEIGO*, under NVMM+NVMe* **SSD** *and NVMM+SATA* **SSD** *storage setups.* . . . . . 47

17 **Performance breakdown.** KEIGO *over RocksDB performance breakdown under YCSB A and C with a $zipf_{80}$ distribution.* . . . . . 47

18 **Meta production workload.** *We compare RocksDB over ext4, ext4-DAX, OpenCAS, and* KEIGO*, under NVMM+NVMe SSD and NVMM+SATA SSD storage setups.* . . . . . 48

19 **Nutanix production workload.** *We compare RocksDB over ext4, OpenCAS, and* KEIGO*, under NVMM+NVMe SSD and NVMM+SATA SSD storage setups.* . . . . . 48

20 **Speedb integration with KEIGO.** *We compare Speedb over ext4, and* KEIGO *for YCSB A and YCSB C.* . . . . . 49

21 **LevelDB integration with KEIGO.** *We compare LevelDB over ext4, and* KEIGO *for YCSB A and YCSB C.* . . . . . 49

# List of Tables


1 **Performance characteristics and cost among different storage devices.** *Numbers estimated based on [64, 2, 48, 49, 75].* . . . . . 14

2 **Performance of durable $C_{log}$ writes.** *Experiments were conducted over RocksDB with fillrandom over NVMe and SATA* **SSD** *devices. In full all components reside on* **SSD**; *in wal-tier, $C_{log}$ is placed on* **NVMM** *and LSM levels on* **SSD**. . . . . 27

3 *Lines of code added to RocksDB, Speedb, and LevelDB.* . . . . . 42

4 *YCSB core workloads description.* . . . . . 44

# Acronyms

**DAX** Direct Access.

**DCPMM** DC Persistent Memory.

**DRAM** Dynamic Random Acess Memory.

**HDD** Hard Disk Drive.

**KVS** Key-Value Store.

**LSM** Log Structured Merge.

**NUMA** Non-uniform Memory Access.

**NVDIMM** Non-volatile Dual In-line Memory Module.

**NVMe** Non-Volatile Memory Express.

**NVMM** Non-Volatile Main Memory.

**OS** Operating System.

**PCM** Phase Change Memory.

**PM** Persistent Memory.

**PMDK** Persistent Memory Development Kit.

**POSIX** Portable Operating System Interface.

**SATA** Serial Advanced Technology Attachment.

**SPDK** Storage Performance Development Kit.

**SSD** Solid-State Drive.

**SST** Sorted String Table.

**VFS** Virtual File System.

**WAL** Write-ahead Log.

# Chapter 1

# Introduction

In recent years, data has been growing at an exponential rate. The total amount of stored data around the globe is projected to exceed 200 zettabytes by 2025 [56]. This trend translates to a growing size of datasets in various fields such as science, engineering, and social media. As a result, this has led to the research and development of new systems for efficiently storing, processing, and analyzing large datasets, including user-level file systems, object stores, and key-value stores.

In particular, **Key-Value Stores (KVSs)** are a widespread storage backend used in a variety of data-intensive applications, including caching [25, 55], messaging [52], streaming [4], metadata management [62], among others. In detail, **KVSs** are storage systems where data is stored in a key-value pair format. They benefit from an extremely simple interface exposed to the client, such as `Put(key,value)`, `Get(key)`, and `Delete(key)` for inserting/updating, reading, and deleting data, respectively. This has made them an attractive storage backend for this wide range of applications. Internally, **KVSs** commonly follow a **Log Structured Merge (LSM)** design, buffering (potentially random) writes in memory and eventually performing sequential writes to disk, where data is organized in levels. It is a fundamental design principle employed in these modern **KVS** systems to optimize write and read operations, especially for scenarios with write-intensive workloads. The **LSM** design is used by LevelDB [28] and BigTable [11] at Google, RocksDB [22] at Meta and Nutanix, Cassandra [39] at Apache, WiredTiger [23] at MongoDB, SplinterDB [14] at VMware, and startups such as Speedb [60].

Originally, **LSM** trees were designed with a focus on optimizing storage performance for traditional **Hard Disk Drives (HDDs)** [50, 57]. However, the storage landscape has undergone a big transformation in the past decade, introducing new technologies and mediums that have redefined the boundaries of storage capabilities. One of the most notable advancements in this evolution is the emergence of **Non-Volatile Main Memory (NVMM)** storage devices (also referred to as **Persistent Memory (PM)**), including 3D XPoint [30], **Phase Change Memory (PCM)** [35], and SST-RAM [26]. **NVMM** represents a groundbreaking shift by delivering performance levels that closely rival those of volatile **DRAM**

while providing the crucial benefit of data persistence. **NVMM** can be 2.1× and 1.4× faster than an **Non-Volatile Memory Express (NVMe) Solid-State Drive (SSD)** for reads and writes, respectively, achieving an order of magnitude lower latency. This development has shattered the conventional barriers between volatile and non-volatile storage, raising the question of how **LSM** trees can harness the unprecedented performance capabilities of **NVMM**. However, while they offer faster access to data than traditional block-based storage devices (*e.g.*, **HDD**, **SSD**), they have lower storage capacity and higher cost/GB. For example, **NVMM** can be 15× more expensive that an **SSD** with the same capacity [38].

As such, designing a **KVS** that provides a good trade-off between performance and cost is challenging. On one hand, **KVSs** must capitalize on the high bandwidth and minimal latency that **NVMM** and **NVMe SSD** drives can provide, enabling them to handle critical data with the utmost speed and responsiveness. On the other hand, **LSMs** should also remain adaptable to cost-effective storage options, including cheaper and slower storage media, which can be used to accommodate the remaining bulk of the data.

## 1.1 Problem

Contemporary storage solutions harness the capabilities of heterogeneous storage devices (*i.e.*, combine multiple storage devices, typically with different capacity and performance, to store data) in two fundamental manners: caching [51, 43, 58, 63, 64] and tiering [70, 75, 12, 71]. These strategies are pivotal in optimizing data access and management within modern storage architectures.

Caching predominantly revolves around the use of a high-performance storage device, such as **NVMM** or **NVMe SSDs**, to expedite access to data residing on slower, higher-capacity storage mediums, like **SATA SSDs** or traditional hard disk drives. The performance acceleration provided by this technique is achieved by copying frequently accessed data items onto the high-performance device. In essence, the high-performance device functions as a faster access point to popular data, even though the complete dataset is retained on the slower storage device.

Tiering, on the other hand, is centered around the concept of sharding data across various layers within the storage hierarchy, according to a predetermined placement scheme, without necessarily copying the data across these layers. The decision of where to place data items within the hierarchy is made based on criteria such as temporal locality and the likelihood of future access. For example, recently written data that has higher probability of being accessed in a near future can be positioned on the high-performance storage devices. This dynamic allocation of data ensures that frequently utilized and performance-sensitive information resides on the faster storage media, while less frequently accessed data is delegated to slower

tiers, thus optimizing both storage space and access time.

This dissertation shows that on one hand, read-dominated workloads benefit from a caching approach, but their performance degrades under tiering (*i.e,* when most of the accesses are made over the slower device). On the other hand, for write-dominated workloads, the tiering approach presents better performance, while storing the entire dataset on **NVMM** actually degrades performance. Therefore, existing solutions still can not provide a hierarchical storage scheme that benefits both reads and writes.

Another problem has to do with how the storage hierarchy logic is implemented. Numerous established systems have put forth the idea of building novel **LSM KVSs** from the ground up to fully leverage the benefits of a storage hierarchy [12, 53, 59, 70, 71, 73]. Although these custom **LSM** solutions can offer commendable performance gains, integrating them into existing production systems presents serious challenges. In many instances, **LSMs** are deeply ingrained within the system's core infrastructure and with various other system components. This tight coupling makes it difficult to replace one **LSM** implementation with another, demanding significant development effort and meticulous planning. The portability and maintainability of these solutions is very limited, and makes it harder for developers to take advantage of the benefits of **NVMM** in their own **KVS** systems.

A second approach is using general-purpose tiered file systems [38, 75, 76] or caching systems [43, 42, 58, 63, 64] that manage the storage hierarchy, together with the **LSM**. These systems are designed to be generally applicable, serving the needs of multiple applications with varying I/O workloads and storage requirements, as these do not require any code changes to the applications. This transparency however, means that all requests are treated in the same manner, which prevents the application to fully leverage the capabilities of the different storage devices. And since all requests have the same weight, the data placement might just not be optimal for the specific application. Another issue is that the migration policies, or the rules for moving data between storage tiers are often based on generic metrics such as access frequency or data age, which might not be relevant for certain types of data or workloads.

## 1.2 Objectives

Taking into consideration the aforementioned problems, this dissertation has two main objectives. First, it aims at co-designing a storage hierarchy that enables the combination of heterogeneous devices (including **NVMM**) with the design properties and internal workflows of **LSM** key-value stores. With this, **LSM**-based **KVS** will be able to achieve good performance under both read- and write-oriented workloads.

Second, to ensure the storage hierarchy and corresponding optimizations are applicable across differ-

ent **KVS** systems, it is important to provide the correct abstraction of which the storage hierarchy operates, achieving a balanced trade-off between general-purposed storage systems and specifically designed **KVS**.

## 1.3 Contributions

To achieve the aforementioned objectives, this thesis makes three main contributions:

- As a first contribution, we present an experimental study that demonstrates the impact of a **NVMM**-enabled storage hierarchy over **LSM**-based **KVSs**. Specifically, this was done to better understand the impact of caching and tiering in different scenarios, including the type of operations (read vs write), the access pattern (zipfian vs uniform), persistence guarantees (fsync after write) and device parallelism. We conclude that tiering is the recommended storage hierarchy strategy to improve write performance and caching is recommended for improving read performance. These lessons motivate the design of the final solution.

- As a second contribution, we present the design and implementation of KEIGO, a new storage middleware that enables **LSM**-based **KVS** efficiently use storage hierarchies composed of **NVMM** and block-based devices. KEIGO is aware of the different I/O operations done by the **KVS** and adjusts the data placement and caching policies accordingly, focusing on tiering for write-intensive workloads and caching for read-intensive workloads. KEIGO is also designed to be portable, requiring minimal modifications to the codebase of the **KVS**.

- As a third contribution, we present a comprehensive experimental evaluation that showcases the performance benefits of KEIGO over a number of storage scenarios. We tested KEIGO with a variety of YCSB workloads and real production workloads, and compared it against other state-of-the-art storage solutions. We show that KEIGO is able to enhance **LSM KVSs**' throughput by as much as 12.4× and reduces tail latency by up to 21.3× over both general-purpose storage solutions and **LSM KVSs** built from the ground up.

## 1.4 Document Structure

This document has the following organization:

Chapter 2 presents a study on **LSM**-Tree **Key-Value Stores (KVSs)**, **NVMM** technology and hierarchical storage. We also discuss the existing solutions for storage hierarchy systems, highlighting their

design and key features, and discussing their main limitations. In Chapter 3 we show the impact of a **NVMM**-enabled storage hierarchy over **LSM**-based **KVSs** with different setups. In Chapter 4 we introduce KEIGO, our system. We offer a comprehensive overview of its architecture, design principles, and describe implementation details. In Chapter 5 we present the results of our experiments and assessments of KEIGO's performance and effectiveness under different storage scenarios. In Chapter 6 we summarize our findings, reflect on the implications of our research, and discuss its broader significance in the field. We also outline potential avenues for future research and offer closing remarks on the contributions and limitations of our work.

# Chapter 2

# State of the art

Since this dissertation presents a solution for improving the performance of **LSM**-based **KVSs** in the presence of non-volatile heterogeneous storage, this chapter firstly provides essential background on the composition of **KVSs** and the **LSM** structure in particular. It then addresses **Non-Volatile Main Memory (NVMM)** technology, heterogeneous storage and discusses classic strategies to manage such hierarchies. Finally, we showcase current state-of-the-art systems that provide hierarchical storage solutions and **NVMM**-based storage solutions, whether in a general-purpose manner or through a custom **KVS**.

## 2.1 Background

This section discusses the organization of **LSM**-based **KVS**, **NVMM** technology, heterogeneous storage in **LSM** and presents classic strategies to manage such hierarchies.

### 2.1.1 Key-Value Stores

**KVSs** are a means of storing data that diverge from the conventional relational databases [31] in their data storage approach. In **KVSs**, data is organized as a singular collection, where each key is distinctive and corresponds to one or more values. Basic operations like `get`, `put`, `delete`, and `scan` enable the retrieval, insertion, removal, and exploration of data, respectively, using these unique keys. The `put` operation establishes an association between a key and its corresponding value, while the `get` operation retrieves the value if it exists. Deletion can be accomplished either through a dedicated `delete` action or by using a `put` operation with an empty value. Notably, **KVSs** typically feature a `scan` operation, facilitating the retrieval of a range of key-value pairs. The `scan` operation proves advantageous by eliminating the necessity for multiple `get` requests when obtaining a range of keys.

Over several years, the main storage medium used to store **KVSs** was the **Hard Disk Drive (HDD)** [16]. Most data structures and algorithms were thus optimized around the physical properties of these devices.

These were among others, high latencies, symmetric read and write speeds, slow random access and an infinite number of reads and writes for each block on the **HDD**. This led to specific optimizations geared towards **HDDs**, such as prioritizing sequential write and read operations.

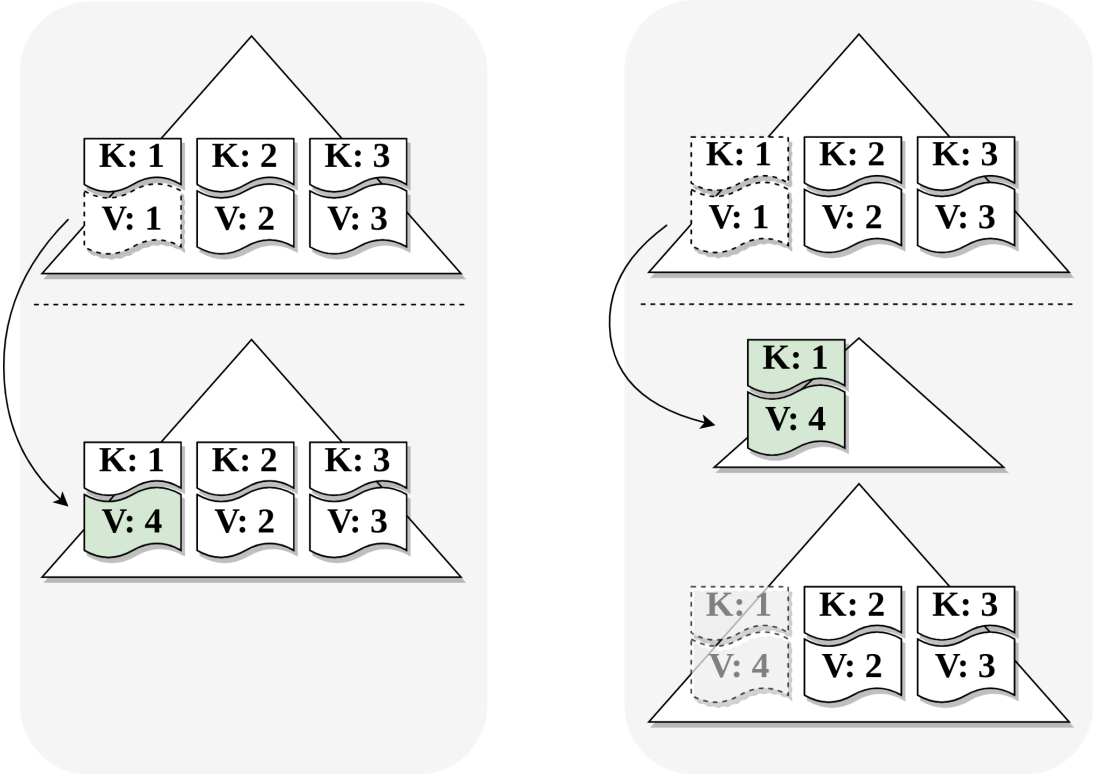
However, with the declining cost of flash devices, both data centers and consumers have increasingly shifted towards utilizing flash storage [3, 40]. This shift required that applications not only became compatible with flash devices but were also optimized for their unique attributes. Many of the assumptions and properties applicable to **HDDs** do not carry over to flash devices. Flash devices exhibit lower latencies, asymmetric read and write speeds, introduce a novel erase operation, are indifferent to random reads at the cell level, and have individual cells with a finite life cycle. Because of such idiosyncrasies, a properly and efficiently usage of these devices required a rethinking of the data structures and algorithms used to store data in **KVSs**.

Key-value stores can be designed in many ways because they do not have strict rules. In **KVSs**, the main structure is used for indexing. Indexing provides a way to find a reference to a given value. Index structures would be best if kept in memory while the actual data is on disk. This however, would require a large amount of memory, increasing with the dataset size. As such, it is common to move part of the index to storage. However, this comes with a challenge. It means more reading and writing to storage for each **KVS** action because now the index needs to be read and written as well. The extra reading and writing for each action is called I/O amplification. Index structures have to make a choice between using memory and dealing with I/O amplification. This choice also depends significantly on the storage type.

On flash storage writes are slower than reads, which makes it attractive to look for a data structure that is optimized for writes instead of reads. An example of such a data structure is the **Log Structured Merge (LSM)**-tree[50]. A big part of the modern flash-optimized **KVSs** make use of **LSM**-trees including LevelDB[28], RocksDB[22] and Cassandra[39]

### 2.1.2 The Log-Structured Merge tree

In general, when it comes to handling updates, index structures can follow one of two strategies: in-place updates and out-of-place updates.




(a) In-place update structure.

(b) Out-of-place update structure.

Figure 1: Examples of in-place and out-of-place update structures: each entry contains a key (*K*) and a value (*V*).

An in-place update structure directly replaces old records with new updates. For example, in Figure 1a, if we want to update the value linked with key `k1` from `v1` to `v4`, the index entry `(k1, v1)` is modified directly to reflect this update. These structures are usually optimized for reading since only the latest version of each record is kept. However, this design sacrifices writing performance because updates involve random I/O operations. Additionally, updates and deletions can lead to fragmentation of index pages, reducing space efficiency.

On the other hand, an out-of-place update structure, such as an **LSM**-tree, always stores updates in new locations without overwriting old entries. For instance, in Figure 1b, the update `(k1, v4)` is stored in a new location instead of directly updating the old entry `(k1, v1)`. This design enhances writing performance by utilizing sequential I/O for writes. It also simplifies the recovery process by not overwriting old data. However, the drawback is that reading performance is compromised since a record may be stored in multiple locations. Moreover, these structures typically need a separate data reorganization process to continuously enhance storage and query efficiency.

Before the introduction of the **LSM**-tree, earlier log-structured storage methods faced several signifi-

cant issues. A primary concern was the use of append-only logs, leading to poor query performance as related records were scattered throughout the log. Another challenge was the inefficient use of space due to the presence of outdated records that had not been removed. Although various processes for reorganizing data were developed, there was no systematic cost model to assess the trade-offs between write cost, read cost, and space utilization. This lack of a model made tuning early log-structured storage difficult, with data reorganization potentially causing performance bottlenecks.

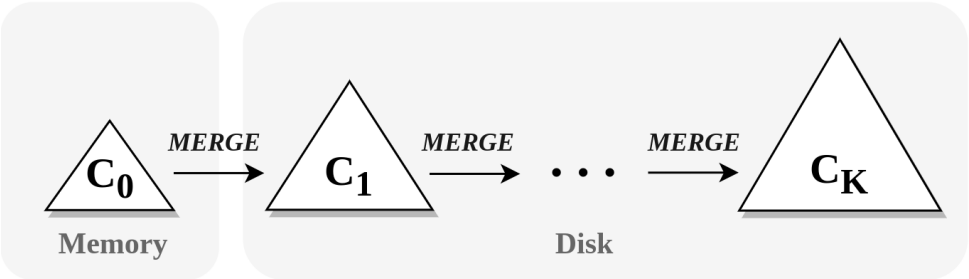



Figure 2: Original LSM-tree design.

The **LSM**-tree design proposed in 1996 [50], tackled these issues by incorporating a merge process directly into its structure. This design aimed to achieve high write performance while maintaining bounded query performance and space utilization. The original **LSM**-tree, as depicted in Figure 2, consists of a series of components ($C_0$, $C_1$, ..., $C_k$), each structured like a $B^+$-tree. $C_0$ operates in memory, handling incoming writes, while the remaining components ($C_1$, ..., $C_k$) reside on disk. When $C_i$ becomes full, a rolling merge process is activated to merge a range of leaf pages from $C_i$ into $C_i + 1$. This design is commonly known as the leveling merge policy [17, 18].

Today's **LSM**-tree implementations still apply updates out-of-place to reduce random I/O. All incoming writes are appended into a memory component. An insert or update operation simply adds a new entry, while a delete operation adds an tombstone entry indicating that a key has been deleted. However, today's **LSM**-tree implementations commonly exploit the immutability of disk components to simplify concurrency control and recovery. Multiple disk components are merged together into a new one without modifying existing components. This is different from the rolling merge process proposed by the original **LSM**-tree. A commonly adopted optimization in pair with component-immutability is to range-partition the disk components of LSM-trees into multiple (usually fixed-size) small partitions. Following the terminology from LevelDB, these partitions are known as **Sorted String Tables (SSTs)**.

Partitioning provides several advantages. Firstly, it divides a substantial component merge operation into multiple smaller ones, limiting the processing time and temporary disk space required for each merge operation. Additionally, partitioning can enhance performance for workloads involving sequentially cre-

ated keys or skewed updates by merging only those components with overlapping key ranges. In cases of sequentially created keys, almost no merge operation is executed as there are no components with overlapping key ranges.

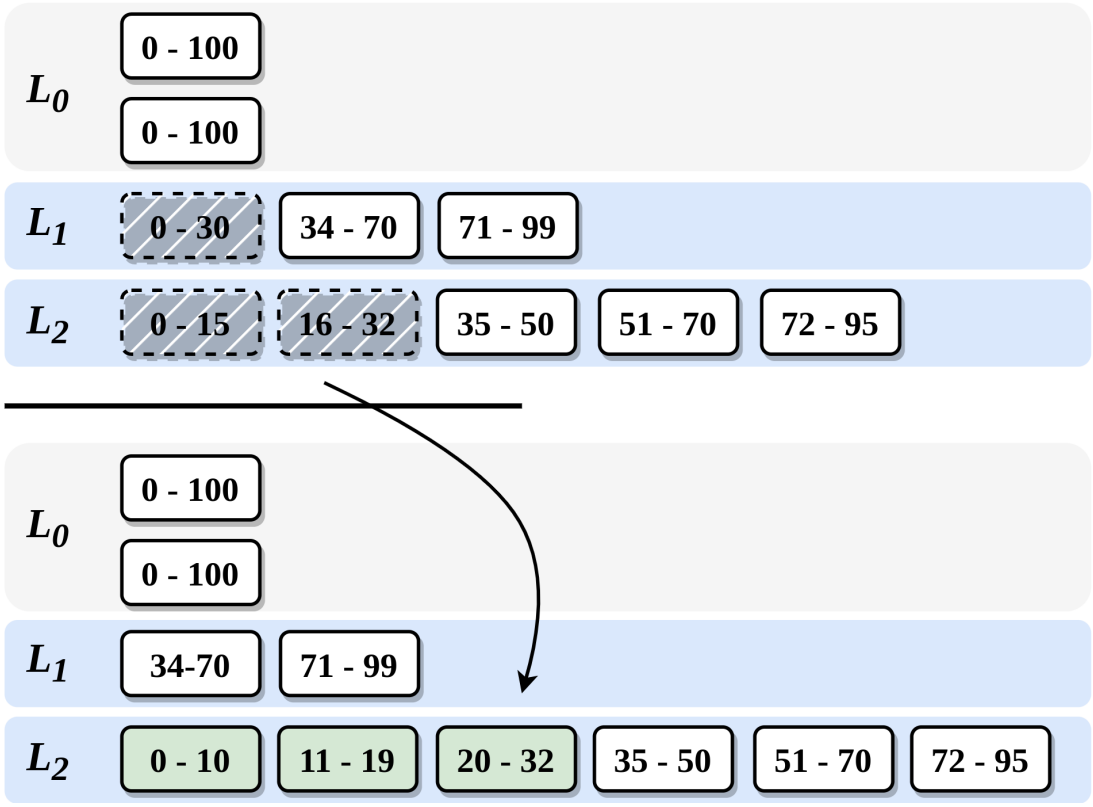



Figure 3: Partitioned leveling merge policy.

The partitioned leveling merge policy, originally introduced by LevelDB, involves dividing the disk component at each level into multiple fixed-size **SSTs**, as illustrated in Figure 3. Each **SST** is identified by its key range in the figure. It is important to note that disk components at level 0 are not partitioned, as they are directly flushed from memory. This design aids the system in handling write bursts by tolerating multiple unpartitioned components at level 0.

When merging an **SST** from level L into level L + 1, all overlapping **SSTs** at level L + 1 are chosen. Those selected are then merged with the **SST** from level L to generate new **SSTs** that remain at level L + 1. For instance, in the figure, the **SST** labeled $0 - 30$ at level 1 is merged with one labeled $0 - 5$ and $16 - 32$ at level 2. This merging process results in new **SSTs** labeled $0 - 10$, $11 - 19$, and $20 - 32$ at level 2, and the old **SSTs** are subsequently garbage-collected.

The **LSM**-tree design grew from a relatively simple model has since been heavily optimized/modified, and it is this modified design that we actually see back in modern literature for flash key-value stores. The new *de facto* design was popularized in LevelDB [28], being organized in four main components: a sorted

in-memory table, a sorted immutable in-memory table, a **Write-ahead Log (WAL)** on disk and multiple **SSTs** that are stored on disk as well.

On an insert or update, every key-value pair is first inserted into the **WAL**, which is an append-only log. This functions similar to journaling in traditional databases and guarantees persistence and consistency. On recovery, the **WAL** can then be replayed as it contains all changes in order. After the write to the **WAL** is complete, the same data is written again, but then to the in-memory table (called MemTable by LevelDB), which is typically small (few MBs). The in-memory table functions as a buffer that can be used to avoid small writes to disk. When the in-memory table reaches a certain size, it is converted to an immutable in-memory table. These immutable tables should eventually be written to storage. At the same time, a new mutable memtable is created or an old memtable is reused. The immutable in-memory tables are used to allow writes to continue to memory without a prolonged wait. Otherwise, the **KVS** has to wait for the entire I/O operation to complete. New data is written to the mutable memtable, while the old immutable table is written to storage. The immutable in-memory table is written to storage as an **SST**. Such an operation is known as a flush. The **SSTs** are stored on storage in a series of levels. Reaching from level 0, commonly referred to as $L_0$, up to level N, commonly referred to as $L_N$. The number of levels depends on the design and can be tweaked. Each level can only hold a certain amount of **SSTs**. This amount increases with each level, so higher levels can hold more data. Some **KVSs** maintain a growth factor between levels of 10x by default. For correctness, only 2 levels would be enough, but adding more levels helps with amortising step. Data can be moved to a higher level, with a process known as compaction. Compaction moves a number of **SSTs** from one level to the next level. This organization is commonly followed by several **KVSs**. In particular, one of the first practical **LSM**-based **KVS** was LevelDB [28], an open-source, high-performance key-value storage library developed by Google.

These common operations in **KVS** can be grouped into two categories: foreground tasks and background tasks. Foreground tasks are operations that are triggered by user commands, such as `get` and `put` requests. Background tasks are operations that are triggered by internal mechanisms, such as compaction and flushes.

### Foreground Tasks

Foregound tasks are **KVS** workflows typically triggered by specific user commands. In particular, a `put` will generate a write operation, while a `get` or a `scan` will generate a read request.

As stated previously when a write operation is received by an **LSM**-tree based key-value store, the first step in ensuring data consistency and avoiding data loss is to record the updates (along with any

associated metadata) in a persistent storage. This is done by appending the updates to a **WAL** ($C_{log}$) file. The $C_{log}$ file serves as an important mechanism for recovery in the event of system failures or crashes, by allowing the replaying of updates recorded in the $C_{log}$ file to restore the data to its most recent state.

After the updates have been recorded in the $C_{log}$ file, the corresponding changes are applied to the active MemTable. The active MemTable is used to serve read requests and it can be efficiently updated with the changes.

When it comes to read requests, these will traverse the **LSM**-tree levels in descending order until the targeted key is found (retrieving the corresponding value), or it is determined that the key does not exist. To improve read performance, many production **KVSs** employ aggressive in-memory caching techniques. However, disk I/O is still necessary, especially with larger databases or lower access locality. The accesses to slower storage can heavily contribute to tail latency, which is the delay experienced by a small fraction of requests that take significantly longer to complete than the average request.

To address this problem, **KVSs** employ background tasks (*i.e.,* flush, compaction) for internal management of the LSM structure and to efficiently access persistent data..

### Background Tasks

**LSM**-tree based **KVSs** employ several background operations to manage the data on disk. These operations are critical for maintaining the efficiency of the system, but they can also have an impact on the performance of foreground request processing. Two of the most important background operations are flush and compaction.

The Flush operation is used to transfer data from the immutable MemTable to the first persistent level of the **LSM** ($L_0$). During this process, the MemTable is written to disk as an **SST** file, often making $L_0$ temporarily larger than $L_1$. The flush operation is triggered when the number of immutable MemTables reaches a limit, flushing the oldest one.

The Compaction operation is used to merge **SST** files from a level $L_i$ with **SSTs** of overlapping key ranges at level $L_i + 1$ and remove invalid key-value pairs (*i.e.,* values that were updated or deleted). This operation is necessary to keep the levels of the **LSM**-tree organized and to prevent the system from becoming too fragmented. The compaction operation is triggered when a level becomes full.

Both flush and compaction operations involve large, sequential I/O, which can cause I/O contention and write stalls when user writes need to wait for flushes to empty MemTable space [8, 46]. This is because the foreground request processing that handles user requests is competing with the background operations for access to disk resources. To mitigate this, some systems use more advanced background

threads management strategy like rate-limiting, priority queuing and more [7, 6, 8, 46].

Additionally, it is also important to note that the strategies used to manage these background flows are very **KVS**-specific. For example, LevelDB [28] uses a single background thread to manage both flush and compaction operations, while RocksDB [22] uses multiple background threads to manage these operations separately.

#### RocksDB's design

RocksDB was initially a fork of LevelDB created by Meta in 2012. Since then, RocksDB has added a large number of new features. Due to its high performance and flexibility, RocksDB has successfully been used in various systems both inside and outside of Meta. According to Meta, a major motivation of for their adoption of LSM-based storage was its good space utilization. With the default size ratio of 10, RocksDB's leveling implementation has about 90% percent of the total data at the largest level, ensuring that at most 10% of the total storage space can be wasted for storing obsolete entries. One of the significant distinctions between RocksDB and LevelDB lies in their data storage mechanisms. RocksDB supports a more flexible column-family based data model, allowing users to organize data into separate column families with different configuration options. This enables better management and optimization of data based on its nature and access patterns. LevelDB, while efficient for simpler use cases, lacks the flexibility of column families, making it less adaptable to diverse data organization requirements.

RocksDB also introduces concurrent compaction as a fundamental feature, allowing multiple compaction processes to operate concurrently. This is achieved through the use of background threads and a multi-level compaction mechanism. Concurrent compactions in RocksDB aim to minimize write amplification and enhance overall system throughput.

### 2.1.3 Persistent Memory (NVMM)

Persistent memory or **NVMM** is a new class of non-volatile memory technology that offers a combination of high capacity and byte-addressability, similar to that of random access memory (**DRAM**), while also providing non-volatility, similar to that of storage devices such as **HDDs** and **SSDs**. This unique combination of properties makes persistent memory a versatile storage solution that can be used in a wide range of data-intensive applications.

One of the main characteristics of persistent memory is its high write throughput as shown in Table 1, which allows for rapid data transfer and processing. Additionally, it has low read latency, which improves the speed at which data can be accessed and processed. Furthermore, because it is byte-addressable,

| Device | Latency | Read | Write | Cost ($/GB) |
|---|---|---|---|---|
| DRAM | 80 ns | 25GB/s | 25GB/s | ≈35 [2] |
| NVMM | 300 ns | 6.8GB/s | 2.3GB/s | ≈5 [2] |
| NVMe SSD | 80 $\mu$s | 3.2GB/s | 1.6GB/s | ≈0.93 [48] |
| SATA SSD | 180 $\mu$s | 0.5GB/s | 0.5GB/s | ≈0.16 [49] |

Table 1: **Performance characteristics and cost among different storage devices.** *Numbers estimated based on [64, 2, 48, 49, 75].*

persistent memory can be accessed by processors directly, which further enhances its performance.

Intel Optane **DC Persistent Memory (DCPMM)** is the first commercially available persistent byte-addressable memory to hit the market. As expected, Optane Pmem boasts a significantly high write throughput and low read latencies. The emergence of this available technology has opened up a plethora of possibilities for database systems and other storage software.

Developers can leverage **DCPMM** in various modes to optimize their applications for performance, capacity, and data persistence. Here are the primary modes in which developers can use Intel **DCPMM**:

**Memory Mode** In Memory Mode, **DCPMM** behaves primarily as traditional volatile memory (**DRAM**). Applications see **DCPMM** as an extension of **DRAM**, offering large memory capacities at a lower cost compared to traditional **DRAM**. This mode provides increased memory capacity for applications without requiring major code modifications, making it an attractive option for workloads that benefit from large in-memory datasets.

**App Direct Mode** App Direct allows developers to access **DCPMM** as a persistent device, providing byte-addressable, non-volatile storage. Applications can use **DCPMM** to store and retrieve data persistently, surviving reboots and power cycles. Developers need to modify their applications to take advantage of the persistence features, making it suitable for scenarios where data durability is critical, such as in databases and file systems.

### NVMM As Block Storage

The initial extension to the operating system for **NVMM** involves identifying the presence of **NVMM** modules and loading a device driver into the operating system's I/O subsystem. This **Non-volatile Dual In-line Memory Module (NVDIMM)** driver serves two crucial roles. Firstly, it establishes an interface for management and system administrator utilities to configure and monitor the state of the **NVMM**

hardware. Secondly, it operates in a manner similar to storage device drivers.

The **NVDIMM** driver presents **NVMM** to applications and operating system modules as a high-speed block storage device (Figure 4). This means that applications, file systems, volume managers, and other storage middleware layers can utilize **NVMM** in the same way they currently use regular block-based storage devices, without requiring modifications.

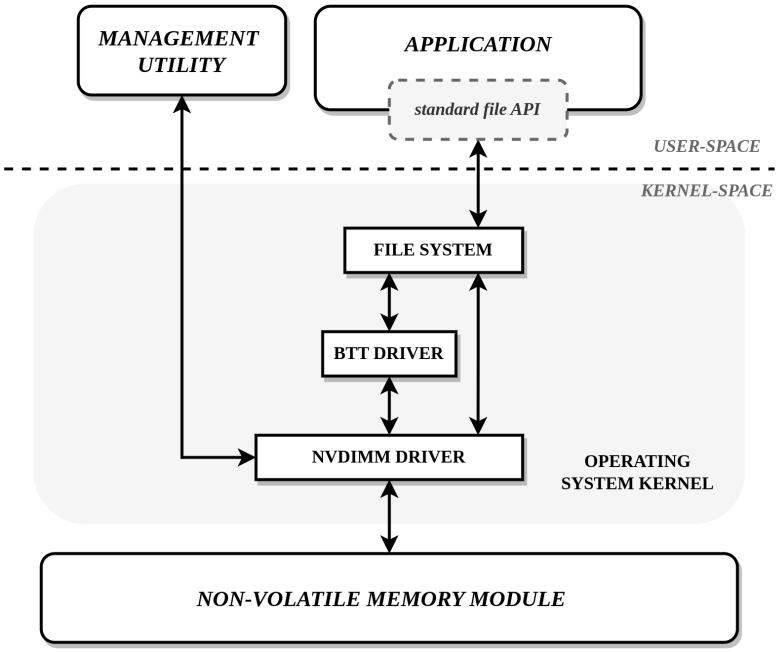



Figure 4: Persistent memory as block storage.

### NVMM-Aware File Systems

The next improvement to the operating system involves making the file system aware of and optimized for **NVMM**. File systems adapted for **NVMM** include Linux ext4 and XFS, as well as Microsoft Windows NTFS. These file systems can choose to use the block driver in the I/O subsystem (as explained earlier) or take a more direct approach by bypassing the I/O subsystem, often called direct-access mode (**DAX**) (Figure 5). In **DAX** persistent memory is used as byte-addressable load/store memory, providing the fastest and most direct route to access data stored in **NVMM**. Besides removing the need for I/O operations, this approach speeds up small data writes compared to traditional block storage devices. Traditional devices require the file system to read the device's native block size, modify the block, and then write the entire block back to the device.

These file systems, designed for **NVMM**, maintain the presentation of familiar, standard file APIs to applications. These APIs include operations like open, close, read, and write system calls. This consistency allows applications to stick with the familiar file APIs while taking advantage of the improved performance

offered by **NVMM**.

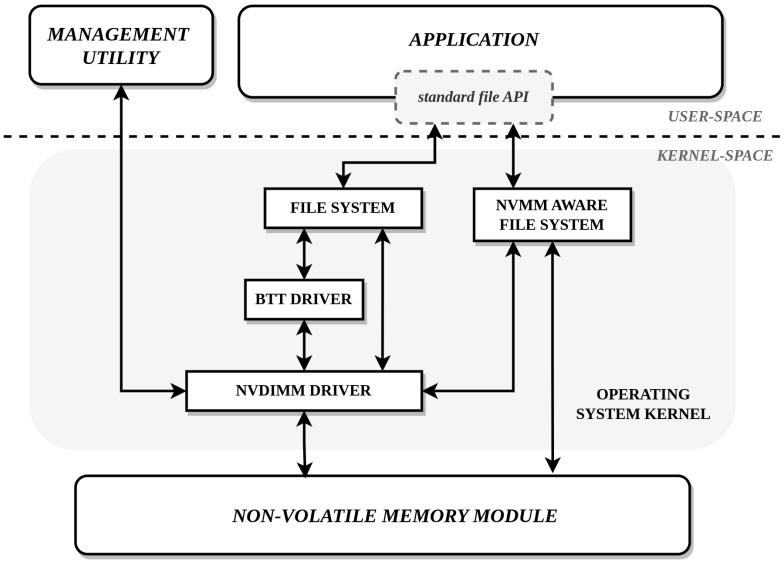



Figure 5: NVMM-aware file system.

## PMDK

The **Persistent Memory Development Kit (PMDK)** is a collection of open-source libraries and tools that enables developers to exploit the full potential of persistent memory. It is designed to simplify the process of creating applications that can take advantage of the unique properties of persistent memory, such as byte-addressability and non-volatility.

The several libraries and tools that cover a wide range of functionality, including memory allocation, transaction management, and data management. The memory allocation library allows for the creation of persistent memory pools (Figure 6), which can be used to allocate and deallocate persistent memory objects. The transaction management library provides a transactional interface that enables developers to create atomic and durable operations on persistent memory. The data management library provides data structures and algorithms that are optimized for use with persistent memory.

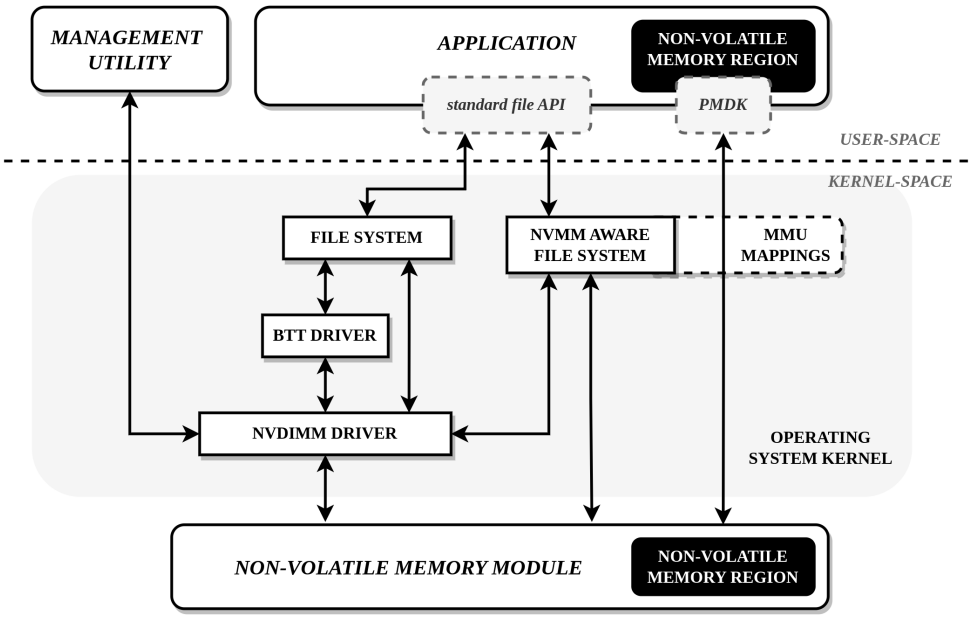



Figure 6: I/O workflow using PMDK.

This allows application developers to directly access persistent memory without going through the kernel. This is known as "user-space persistence" and it allows for faster and more efficient access to **NVMM** compared to traditional storage methods that rely on kernel-level support.

**PMDK** provides a set of libraries that support a variety of operations for reading and writing data to **NVMM**. One of the most use libraries is libpmem, which exposes a set of low-level operations for the application to interact with the pmem device. Specifically, libpmem exposes the following operations for storing and accessing data:

- **pmem_map**: This function facilitates the memory mapping of a specified file, typically residing on a persistent memory device, into the virtual address space of a process. The parameters of pmem_map_file allow developers to finely control the characteristics of the mapping, including the file path, desired length, access permissions, and creation mode. When invoked, the function creates a memory-mapped region, treating the file as a memory-mapped file. If the specified file does not exist, pmem_map_file will create it, and if the file is smaller than the requested length, it will be extended accordingly. The flags parameter enables the definition of access permissions (read-only, read-write, etc.).

- **pmem_memcpy**: This function is designed to efficiently copy data between two memory regions (either both persistent or one volatile and one persistent). Unlike traditional memory copy functions, pmem_memcpy is tailored to take advantage of the unique characteristics of persistent memory, offering optimizations that enhance performance and durability. It is therefore specially useful as a

writing mechanism analogue to **POSIX**'s `write`. To read, the application can simply use **POSIX**'s `memcpy`. The function signature typically includes source and destination pointers, the length of the data to be copied, and flags to specify the desired behavior, such as whether the operation should be performed atomically. Importantly, pmem_memcpy ensures the durability of data by utilizing appropriate memory barriers and persistence mechanisms inherent in persistent memory architectures.

- **pmem_persist**: Its primary purpose is to ensure the durability of modified data in persistent memory by explicitly flushing changes to the underlying storage medium. In persistent memory systems, data modifications may be cached in the CPU's cache or in device-level caches (*e.g.,* XPBuffer), and without proper flushing mechanisms, there is a risk of data loss in the event of power failure or system crash. This Flushing to persistence involves two steps:
  - Flushing the CPU caches or bypass them entirely as explained in the previous example.
  - Waiting for any hardware buffers to drain, to ensure writes have reached the media.

  These steps are performed together when pmem_persist is called.
- **pmem_drain**: This function only executes the second step of the flushing process, waiting for hardware buffers to drain. For example, a program that needs to flush several discontiguous ranges can call pmem_flush for each range and then follow up by calling pmem_drain once.

One of the key features of the **PMDK** library is its compatibility with existing software frameworks and libraries, such as the popular Memory Allocator Interface (malloc) and the C++ Standard Template Library (STL). This allows developers to leverage the benefits of persistent memory without having to completely rewrite existing code.

**Discontinuation of Intel Optane** In Q2 2022, Intel decided to discontinue the production of Optane **DCPMM** Our prototype is tested that device. Despite Intel discontinuing its Optane Memory product, it showed the potential of the technology, which motivated the research and development of other **NVMM** devices (*e.g.,* Phase Change Memory [35], Resistive RAM [1, 26]). Although this dissertation makes use of Optane Memory, KEIGO is extensible to other **NVMM** technologies (which we plan to support once these become mature and publicly available) and we believe our performance gains remain valid for those devices. Moreover, recent studies indicate that emerging persistent memory technologies are likely to have similar characteristics to those of Intel Optane **NVMM** devices [29]. It is therefore likely that the strategies proposed by KEIGO will remain relevant.

### 2.1.4 Heterogeneous Storage

Using a single device means that the user has to make a trade-off between performance, cost, and capacity. To address this, heterogeneous storage systems are designed to leverage multiple devices with diverse characteristics. Hierarchies are the most common way to accommodate multiple heterogeneous storage devices, each offering unique trade-offs of latency, throughput, and cost (shown in Table 1). In this work, we consider a hierarchy of byte-addressable persistent memory (**NVMM**) such as 3D XPoint [30], block-addressable NVMe solid state drives (**SSDs**), and **SATA SSDs**.

To manage the storage hierarchy, systems mainly follow two strategies: *caching* and *tiering*. For simplicity, we assume a two-tier hierarchy, made of a *performance device* that is faster, smaller, and expensive (*i.e.,* **NVMM**), and a *capacity device* that provides cheaper and larger capacity storage (*i.e.,* **NVMe**/**SATA SSD**, or a storage array).

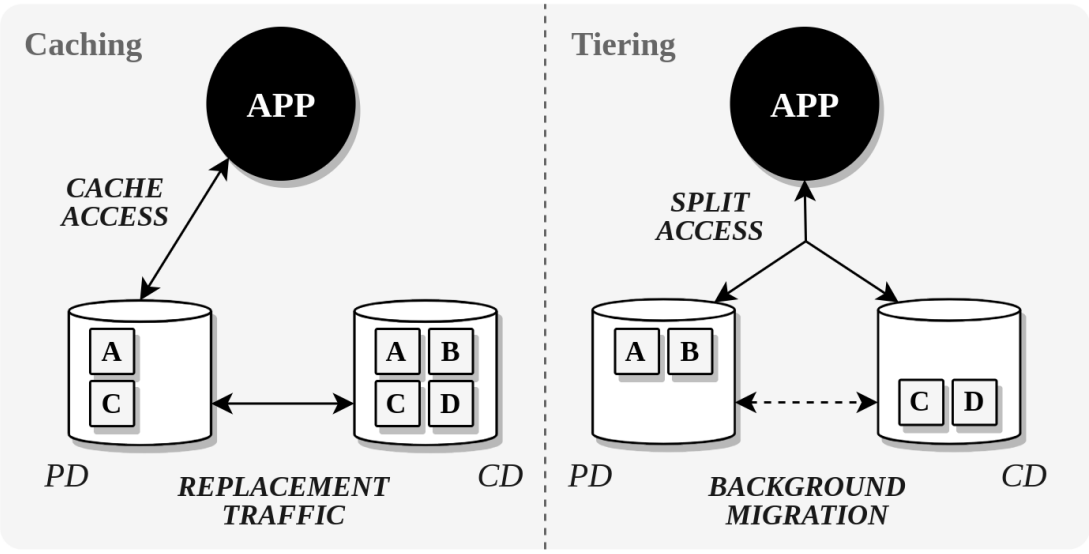



Figure 7: Caching and Tiering. Caching copies data items from the capacity device (CD) to the performance device (PD). Tiering splits access to each layer and migrates items in the background.

**Caching** In caching, the *performance device* is used as a persistent cache to accelerate the *capacity device* [63, 51, 43, 42, 58]. Similarly to the **OS** page cache, hot data items are held in the *performance device*, ensuring low latency access and high throughput under skewed, read-intensive workloads. Upon a *cache miss*, the request is serviced by the *capacity device*. Promotion and eviction of items are determined by the deployed caching scheme (*e.g.,* LRU, LFU). Depending on the writing policy (*e.g.,* write-back), the *performance device* can absorb write operations, flushing dirty data upon explicit synchronization [51] or periodically by a dedicated background process [43, 63].

**Tiering** In tiering, items are split across both devices according to a specific placement scheme, which is often driven by the popularity of items (*e.g.,* hot data is placed on the *performance device*) but can also be based on the item size, consistency guarantees, etc [75, 70, 38, 12]. Contrary to caching, items only reside in one of the devices and are not constantly promoted/evicted to/from the *performance device*.

## 2.2 Related work

In this section of the document we provide an overview of the current state of the art in hierarchical storage and hierarchical storage based **KVSs**. We present systems that provide General-purpose hierarchical storage solutions, **KVSs** with hierarchical storage support and **NVMM**-based **KVSs**. We list the main contributions and key features of some of the most widely used and widely cited systems in this field, considering the problems they tackle, the solutions they propose and the results they achieve.

**General-purpose hierarchical storage** A commonly used approach is to use hybrid storage systems, which are generally applicable and do not require **POSIX**-compliant applications to be modified in any way to obtain performance improvements. The storage system determines which device services each I/O request based on the data access pattern (*e.g.,* write followed by fsync [43, 63], small *vs.* large writes [75, 76]). Some of these general-purpose systems are cache-based, including OpenCAS [51], FirstResponder [58], P2Cache [43] and SPFS [63].

OpenCAS [51] is a generic block-layer caching mechanism that enables using a fast device as a cache (for both reads and writes) of a slower device, and in particular it can be configured so that an **NVMM** acts as a block cache for an **SSD**. OpenCAS is installed into the operating system as a kernel module, therefore this integration provides a cache solution that is transparent to users, applications and the existing storage. OpenCAS allows for several caching modes: write-back, write-through, write-around or pass-through (disabled). In write-through mode, all data is written synchronously to both the backend storage and the cache. In write-back mode, all data is written to the cache then eventually flushed to the backend storage. In write-around mode, non-cached writes are only written to the backend storage and reads bring the data to cache.

P2Cache, FirstResponder and SPFS are in-kernel caching mechanisms that enhance legacy file systems by using **NVMM** to absorb frequent writes.

FirstResponder addresses the integration of **NVMM** into existing file systems, emphasizing its functionality as a buffer cache. It operates as a union of storage and a cache, providing immediately durable writes and seamlessly coexisting with traditional file systems like ext4. FirstResponder is designed with

low management overhead, thereby optimizing the integration of **NVMM** without extensive modifications to mature file systems. FirstResponder is implemented in the **Virtual File System (VFS)** layer, which allows it to be used over multiple different filesystems.

P2Cache, on the other hand, distinguishes itself by implementing a tiered system that combines **NVMM** and **DRAM**. The system introduces a persistent cache below the **VFS** layer, ensuring instant data durability, with a big focus on strong consistency without requiring modifications to applications. This innovative approach allows P2Cache to efficiently handle write requests using **NVMM** for instant data durability, while using **DRAM** for most read I/O operations to enhance overall I/O performance (caching both reads and writes).

Stackable Persistent memory File System (SPFS) takes a modular approach to enhance I/O performance by acting as a lightweight stackable file system on top of disk-optimized file systems. SPFS specifically manages **NVMM** and leaves the disk and **VFS** cache to the underlying file system. SPFS's stackable nature allows it to seamlessly integrate with diverse block device-optimized file systems, providing a flexible and practical solution for leveraging **NVMM** in conjunction with traditional storage devices.

Another class of general-purpose heterogeneous storage systems relies on tiering, such as Orthus [64], Strata [38], Ziggurat [75], and TPFS [76].

Orthus aims to solve one problem presented by the current landscape of storage devices, namely their overlapping performance characteristics. They argue that under certain situations (*e.g.,* high concurrency), modern devices do not benefit from using traditional storage hierarchy systems like caching and tiering. Orthus introduces a non-hierarchical caching strategy that, when under excessive I/O load, redirects requests to less stressed devices. This strategy not only helps delivering additional performance from the capacity device but also avoids data movement between the devices when this movement does not improve performance. Their implementation is based on two systems: Orthus-CAS, a generic block-layer caching kernel module, and Orthus-KV, a user-level caching layer for an LSM-tree key-value store.

Strata, Ziggurat, and TPFS are file systems that tier data across **DRAM**, **NVMM**, and SSD according to the applications' access patterns and consistency requirements.

In particular, Strata is implemented with a **LSM** tree architecture, where data is initially synchronously written to an operation log in **NVMM**. This approach allows for fast, random, and small writes. Periodic digestion of logs into a read-optimized tree format helps manage write amplification and ensures efficient data search and retrieval. Strata supports high concurrency through atomic operations and provides a unified interface to the entire storage hierarchy, maintaining backward compatibility with **POSIX** applica-

tions.

Ziggurat, on the other hand, utilizes predictors to intelligently steer incoming writes to either **NVMM** or disks based on the application behaviors. It features an adaptive migration mechanism that efficiently manages data migration between storage tiers. The implementation involves profiling file data temperature, coalescing adjacent data blocks, and adjusting migration policies based on application access patterns. Evaluation results demonstrate that even with a small amount of **NVMM** and a large **SSD**, Ziggurat achieves substantial throughput improvements compared to traditional file systems running on **SSD** alone. As the amount of **NVMM** increases, Ziggurat's performance approaches that of an **NVMM**-only file system, showcasing its effectiveness in optimizing storage performance across different storage technologies. TPFS is simply an extension for Ziggurat, which supports **NVMM** explicitly.

These systems, however, are agnostic of applications' internal I/O logic. For **LSM KVSs**, this means that requests with different priorities and storage performance costs are treated in the same manner across the storage hierarchy, impacting the KVS' end performance as observed in Chapter 3. For example, while **KVS** compactions exhibit similar access pattern – sequentially read SST files from disk, merge sort in memory, and sequentially write new SST files to disk – they incur different performance costs (*e.g.,* low-level *vs.* high-level compactions), as observed in [8, 72]. As expected, the performance of **KVSs** ran on top of these systems is suboptimal, as compared to the performance of **KVSs** that are aware of the storage hierarchy and can leverage it to optimize their internal I/O logic.

**KVS with hierarchical storage support** Alternatively, prior work proposes building entirely new **LSM KVSs** for heterogeneous storage. These new **LSM** designs consider the trade-offs of each storage device with careful placement of the **LSM** components [34, 33, 59], employ new compaction schemes [70, 53], and redesign the write operation flow [12, 34].

Mutant [71] is described as a storage layer for **LSM**-tree data stores, designed to address the challenge of seamlessly balancing cost and performance in cloud databases. The primary problem it aims to solve is the tedious and error-prone process of manually trading off database cost and latency, especially when adapting to changing workloads or budget constraints. It is implemented as an extension of RocksDB and dynamically organizes **SSTs** based on access frequencies, leveraging the temporal locality of modern workloads. By continuously migrating older **SSTs** and colder data to slower and cheaper storage devices, Mutant automatically maintains a cost budget while minimizing latency. The implementation involves lightweight tracking of access patterns and a computationally-efficient algorithm to organize **SSTs**.

SpanDB [12] is a **KVS** system designed to optimize the performance on **NVMe SSDs**. The primary

problem it aims to solve is the underutilization of **NVMe SSDs** in existing **LSM**-tree **KVS**, as demonstrated by limited throughput improvements when deploying systems like RocksDB on such devices. To address this, SpanDB incorporates a hybrid storage approach, utilizing a small, fast-speed disk for recent data ($C_{log}$ and lower levels of the **LSM** tree) while storing the majority on larger, cheaper capacity disks. It employs the **Storage Performance Development Kit (SPDK)** library for fast, parallel accesses, introduces an asynchronous request processing pipeline suitable for polling-based I/O, and strategically partitions data based on actual workload characteristics.

PrismDB [53] is also a novel key-value store designed to achieve efficient balance between performance and cost-per-bit in storage systems. It simultaneously utilizes the fast and low-cost storage capabilities of **NVMM** and QLC NAND, respectively. The implementation involves a unique hybrid data layout, storing frequently accessed data in slab-based files on the fast storage (**NVMM**) and less frequently accessed or immutable data in a sorted log on slower, denser storage (QLC NAND). PrismDB introduces a multi-tiered storage compaction mechanism that efficiently migrates and compacts data between the two storage tiers based on object popularity and access frequency.

MatrixKV [70] proposes a new matrix-like data structure that manages $L_0$ and is placed on **NVMM**. The key problems targeted by MatrixKV are write stalls, which cause periodic drops in application throughput due to large data processing during $L_0$-$L_1$ compaction, and write amplification, associated with the depth of **LSM**-trees. MatrixKV leverages **NVMM** in **DRAM**-**NVMM**-**SSD** storage systems to construct cheaper and finer granularity compaction for $L_0$ and $L_1$, simultaneously reducing the depth of **LSM**-trees to mitigate write amplification.

Prism [59] is also a **KVS** built from the ground up to support heterogeneous storage. Prism's architecture is based on placing each one of its components (*i.e.,* Persistent Key Index, Persistent Write Buffer, Value Storage, Scan-aware Value Cache, and Heterogeneous Storage Index Table) on the storage device that best satisfies its requirements. In more detail, it follows a key-value separation design, where keys are placed on **NVMM** and values are placed on **SSD** (being first absorbed by **NVMM** and later migrated to **SSD**). This way, Prism is able to maximize the advantages of individual storage devices and suppress their disadvantages. They also introduce cross-media crash consistency and concurrency control techniques. Prism is implemented with **DRAM**, **NVMM**, and flash **SSDs**.

All of these systems are designed to leverage heterogeneous storage devices, but they are also redesigning the **LSM** architecture, requiring significant code changes to the **KVS**. Replacing existing, well-tested LSMs used in production with these systems is not trivial, posing significant implementation efforts. This problem is further accentuated with the release of new storage devices.

**NVMM-based KVS and Memory tiering** Recent work has also explored the design of new KVS that exclusively use **NVMM** as main storage [66, 34, 33, 13, 74, 73, 61, 37] and other research lines explore **NVMM** as a cheaper alternative/extension to **DRAM** [54, 24, 36, 21, 41]. These strategies are orthogonal to the work presented in this dissertation, as they not consider devices that reside at deeper levels of the storage hierarchy. However, they are still relevant, and could be combined with our system to further increase the performance of **LSM**-based **KVSs**.

# Chapter 3

# Motivational Experiments

We conducted an extensive experimental study to understand how traditional storage hierarchy techniques (*i.e.,* caching and tiering) combining **NVMM** and block-based storage devices (*e.g.,* **NVMe** and **SATA SSDs**) impact the performance of **LSM KVS**. Our observations complement previous studies that comprehensively explore the performance characteristics and idiosyncrasies of **NVMM** devices [68, 29, 67, 24], by pinpointing their impact in **LSM**-based **KVSs**. Unless stated otherwise, experiments were conducted with the following testing environment.

**Hardware and OS configurations** Experiments were conducted in a server configured with two 18-core Intel Xeon processors, 192GB of memory, a 128GB Intel Optane DC Persistent Memory, a 1.6TB Dell PM1725b **NVMe SSD**, and two 480GB Intel S4610 **SATA SSDs** configured with RAID-0. Software-wise, we used Ubuntu Server 20.04 with kernel 5.4.0. We restricted main memory to 16GB using Linux control groups to ensure experiments execute over datasets at least 3× larger than **DRAM**, and used a single socket to prevent **NUMA** effects. **NVMM** was configured under AppDirect mode (*i.e.,* **NVMM** is used has a persistent device and not a **DRAM** expansion).

**Experimental environment** Experiments were conducted over RocksDB configured with two $C_m$ components of 128 MB each, a thread pool of four threads for internal operations (including a flushing thread), and a 1 GB block cache. We evaluate RocksDB under a multi-threaded setting (8 client threads) using YCSB [15] update-intensive (YCSB A, 50% reads and 50% writes) and read-only (YCSB C) workloads with a uniform distribution. Before each experiment, RocksDB was pre-loaded with 50M key-value pairs in a uniform key distribution.

**Setups** *ext4* corresponds to an ext4 file system mounted over an **NVMe SSD** device. *ext4-DAX* corresponds to a **NVMM** device backed by ext4 with **DAX** enabled. Furthermore, we implemented a custom storage backend that combines **NVMM** and block-based storage. **NVMM** is backed by a **PMDK** driver

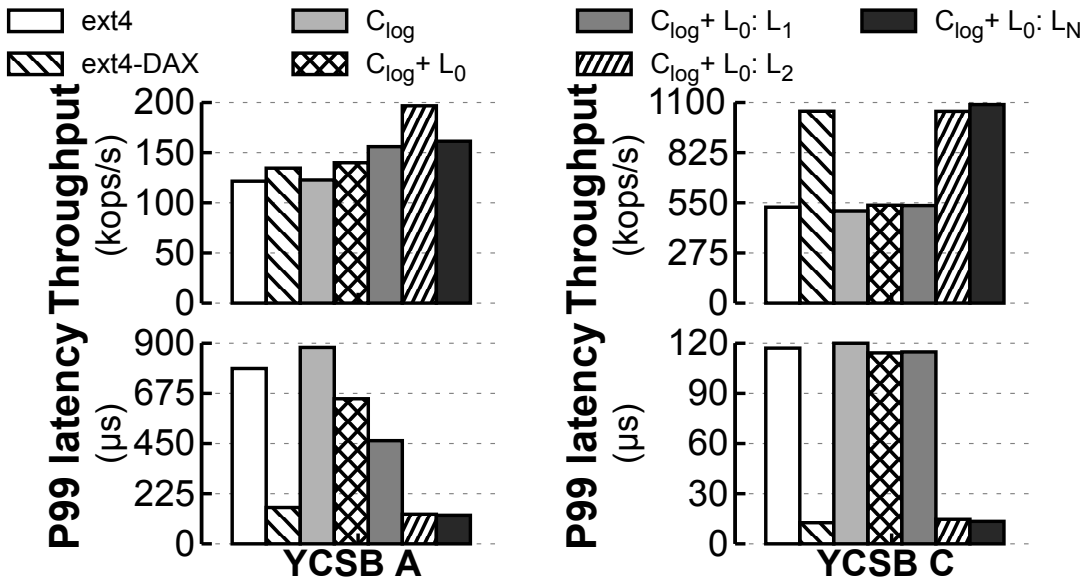



Figure 8: **Performance of placing LSM components across different storage devices.** *Experiments conducted over RocksDB with both write-intensive (YCSB A) and read-intensive workloads (YCSB C). In ext4 and ext4-DAX, the dataset is fully stored in NVMe* **SSD** *and* **NVMM** *devices, respectively. The remainder setups are conducted with a combination of* **NVMM** *and NVMe* **SSD**.

directly implemented over RocksDB (further details in Section 4.5), while the **SSDs** use a regular ext4 file system.

## 3.1 Placement of LSM components

We first show the performance impact of placing different components of the **LSM KVSs** (excluding $C_m$) across the storage hierarchy. To this end, we compare RocksDB throughput and tail latency performance under ext4, ext4-**DAX**, and the custom storage backend configured with different placement schemes. We start by placing $C_{log}$ on **NVMM** and keep the remainder components on the **NVMe SSD**. We then incrementally place more components on **NVMM** ($C_{log}$+ $L_0$, $C_{log}$+ $L_0$:$L_1$, ...), until the full dataset is serviced from this device. Figure 8 shows the performance (*i.e.,* throughput and 99th percentile latency) of RocksDB under write and read-intensive workloads for all storage backends.

**Write-intensive workloads** As expected, placing all components on **NVMM** (namely, ext4-**DAX** and $C_{log}$ + $L_0$:$L_N$) increases RocksDB throughput up to 34% when compared to ext4. Also, the custom storage backend improves throughput by 20% over ext4-**DAX** due to its **PMDK** driver, which accesses data items directly from user-space.

However, on closer inspection, we observe that placing the entire dataset on **NVMM** is actual detri-

| | NVMe SSD | | SATA SSD | |
|---|---|---|---|---|
| | **full** | **wal-tier** | **full** | **wal-tier** |
| **thr.** (kops/s) | 6.99 | 89.6 (+12.8×) | 1.98 | 51.7 (+26.1×) |
| **lat. 99**$^{th}$ ($\mu$s) | 368.7 | 14.9 (-24.7×) | 722.5 | 43.1 (-16.7×) |

Table 2: **Performance of durable $C_{log}$ writes.** *Experiments were conducted over RocksDB with fillrandom over NVMe and SATA SSD devices. In full all components reside on SSD; in wal-tier, $C_{log}$ is placed on NVMM and LSM levels on SSD.*

mental, degrading throughput up to 35% when compared to just placing a small subset of data items ($C_{log}$ + $L_0$:$L_2$). The reason behind this is due to NVMM's poor scalability under concurrent writes that is caused by contention in the XPBuffer (*i.e.,* increased evictions and write backs to the memory media) and the integrated memory controller (*i.e.,* limited queue capacity when multiple cores target a single DIMM), as observed in [68, 29]. As such, placing more components on NVMM will result in more I/O workflows concurrently writing to it, namely foreground writes to $C_{log}$, flushing $C_m$ to $L_0$, and parallel compactions occurring over different levels.

**Read-intensive workloads** Contrary to write-intensive workloads, RocksDB performance improves with the number of components placed on NVMM. This is because while NVMM's writes do not scale well, reads can scale up to the number of physical cores [68].

**Durable $C_{log}$ writes** We conducted an additional set of experiments to understand the performance impact of durable $C_{log}$ writes under different storage mediums, where each `write()` to $C_{log}$ is followed by an `fsync()`. We executed a single-threaded random write workload over RocksDB under two storage settings: in *full* all LSM components reside on SSD while in *wal-tier*, $C_{log}$ is placed on NVMM and the LSM levels on SSD. As shown in Table 2, just by placing the $C_{log}$ on NVMM enables up to 26.1× better throughput and 24.7× lower tail latency then when the full dataset is stored in SSD. For these experiments an ext4-DAX file system was mounted on NVMM, along with a regular ext4 file system on the slower devices.

## 3.2 Popularity of LSM levels

We now investigate how different data access distributions can impact the storage hierarchy. We analyze the number of accesses made over time across the LSM (levels and block cache) under different data

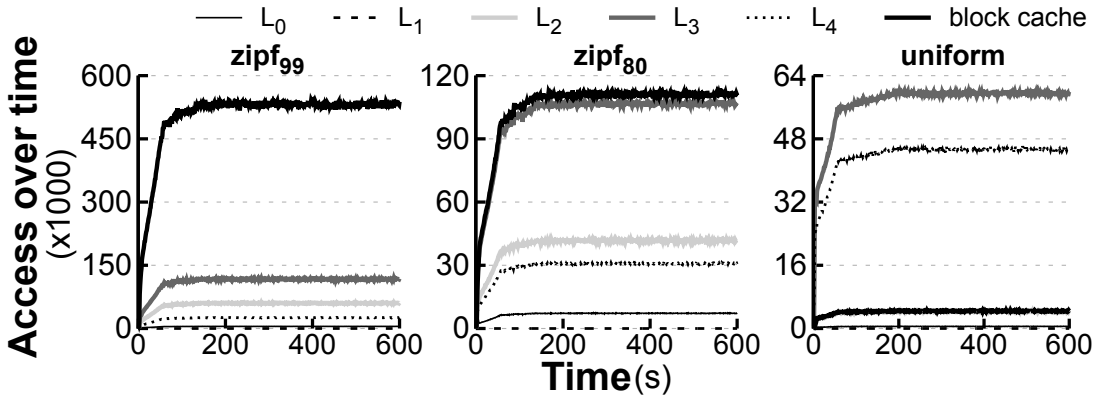



Figure 9: **Number of foreground reads performed across LSM levels.** *Experiments were conducted using YCSB C over a 200M key-value pairs dataset under zipf 0.99, zipf 0.80, and uniform.*

distributions, including zipfian 0.99 (high skew), zipfian 0.80 (medium skew), and uniform.

**Read workflows** Figure 9 depicts the number of foreground reads observed at each **LSM** level for a read-only workload, conducted over a dataset of 200M key-value pairs. Across all distributions, we observe that $L_3$ is the most accessed level. While $L_0$ to $L_2$ have a combined size of approximately 3 GB, $L_3$ holds 25 GB of the dataset. On the other hand, even though $L_4$ can accommodate 10× more data than $L_3$, it holds the older and colder portion of the dataset [44, 71]. We also observe that the block cache places an important role in highly skewed workloads, being able to service a large amount of reads, but its impact fades as the distribution becomes less skewed. This means that for low skew, read-intensive workloads, levels higher than $L_2$ need to be stored in **NVMM** as well; otherwise, a significant portion of requests will be made over the **SSD**.

**Write workflows** Due to the design of **LSM**-based **KVSs**, the write data path follows the same workflow regardless of the data distribution. As such, components that handle the main write path (*i.e.,* $C_{log}$ and $L_0$) should be placed in **NVMM**, while the placement of other levels must be carefully chosen to prevent write contention.

## 3.3 Concurrent compactions

Complementing the observations made in Section 3.1, we now analyze how increasing both the number of writers and the components stored in **NVMM** impacts the **KVS** performance. We fixed the number of client and flushing threads, but varied the number of compaction threads (1 to 16).

Figure 10 shows the average throughput over time across all tested configurations under YCSB A.

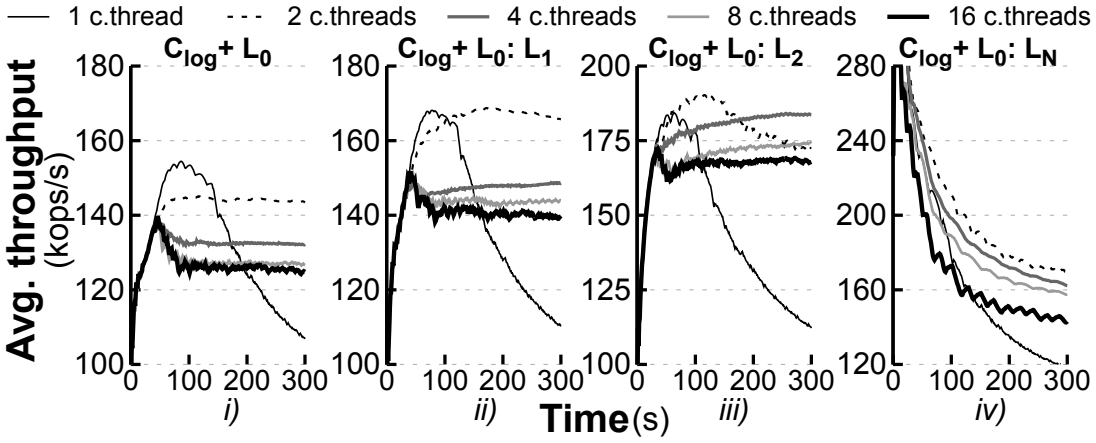



Figure 10: **Concurrent compactions under different placement schemes.** *Experiments were conducted with a fixed number of client (8) and flushing (1) threads and varying compactions threads (1–16). The title of each plot depicts the components that are placed on* **NVMM**. *The remainder levels are placed on the NVMe* **SSD**.

When a small amount of data is persisted in **NVMM** (*i.e., i)* and *ii)*), RocksDB's performance is suboptimal since a large amount of requests are serviced by **SSD**. Increasing the number of threads does not improve performance since the problem lies in the placement scheme. Similarly to Section 3.1, RocksDB achieves the best throughput performance under the $C_{log}$+ $L_0$:$L_2$ configuration, when using 4 compactions threads (*iii)*). With a low number of threads (*i.e.,* up to 2), RocksDB experiences write stalls, which occur when flushes cannot proceed either because flushes and low-level compactions ($L_0$-$L_1$) are slow or on hold [8, 72]. On the other hand, too many compaction threads (*i.e.,* 8 or 16) increases the number of writers contending the **NVMM**. These effect becomes further accentuated when all components reside on **NVMM** (*iv*). As such, given that compactions play an important role in **LSMs**, the storage hierarchy must be aware of them.

## 3.4 Lessons learned

Based on the conducted experimental study, we gain three main insights:

**Lesson 1** *Tiering* is the recommended storage hierarchy strategy to *improve write performance* in **LSM**-based **KVSs**. **LSM** components that are closer to $C_m$ – namely, $C_{log}$ and $L_0$ to $L_2$ – must be placed on **NVMM**. Not only because these components handle the critical write data path, but also because placing more levels degrades **KVSs**' end performance due to **NVMM**'s poor write concurrency.

**Lesson 2** *Caching* is the recommended storage hierarchy strategy to *improve read performance* in **LSM**-based **KVSs**. Leveraging the fact that **NVMM** offers excellent read bandwidth and scalability, one should strive to maximize **NVMM**'s hit ratio. This is especially important for workloads with medium and low skewed data access distributions.

**Lesson 3** Under mixed and dynamic workloads, where the proportion of read and write requests, as well as the skewness of data objects vary over time (as observed in production environments [47, 8, 10]), *there is not a winning placement strategy*. Instead, one must carefully combine both strategies to ensure **LSM**-based **KVSs** can simultaneously achieve good performance for read-intensive and write-intensive workloads.

# Chapter 4

# KEIGO: LSM-aware hierarchical storage

Based on the above lessons, we introduce KEIGO, a new **LSM**-aware storage system that accelerates the performance of **LSM**-based **KVS** using a hierarchy of storage devices. Through its design, KEIGO automatically combines devices' strengths, compensates for their weaknesses, all while requiring minimal changes to the **KVS**. KEIGO presents a co-design between the intrinsic properties and workload patterns of **LSM KVSs** (*i.e.,* write-oriented workflows, latency-critical components, strong temporal locality) with the characteristics of both **NVMM** (*e.g.,* byte-addressability, instant data durability) and block-based storage mediums (*e.g.,* large capacity, low cost), which combined, provide a high-performance, cost-effective, large capacity storage system. KEIGO's design is built under four objectives.

**Fast performance-critical operations** KEIGO must ensure that all operations done on the critical data path that are latency-sensitive and prone to write stalls, namely $C_{log}$, flush, and $L_0$ to $L_1$ compactions, are handled by the **NVMM** device.

**Maximize NVMM hit ratio** KEIGO aims to exploit the **NVMM**'s read bandwidth and scalability, maximizing the number of requests served from it.

**Low intrusiveness** KEIGO requires minimal code changes to **LSM**-based **KVSs**, thus minimizing the work needed to maintain and port it to new systems. We demonstrate this by integrating KEIGO with three popular key-value stores (i.e., RocksDB, LevelDB, Speedb) with less than ≈80 LoC.

**Flexibility** KEIGO can support different storage drivers and enables using different data placement strategies allowing system designers to adjust the system to their use case, without changing the **KVS**. We refer to the term *storage driver* as the component that implements the operation logic for interacting with specific storage devices.

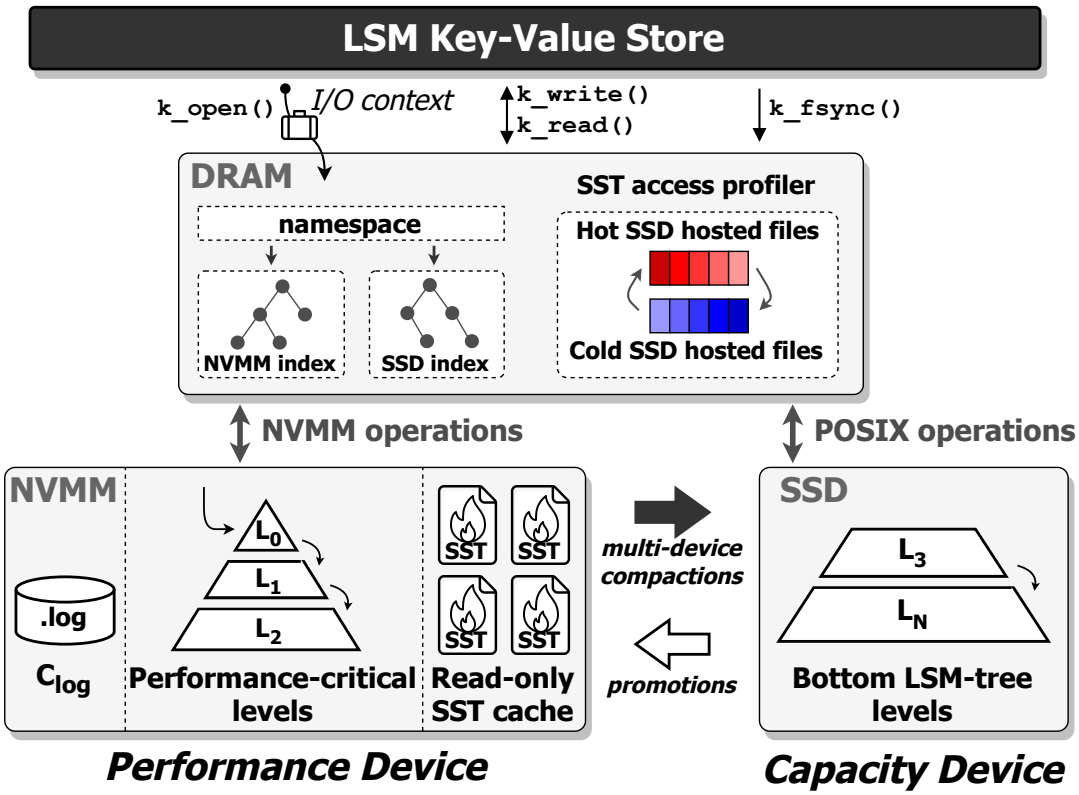



Figure 11: **KEIGO architecture.** KEIGO *follows a multi-tier storage structure driven by the design and workload properties of LSM KVS. The performance device handles* **NVMM** *operations (*e.g., `pmem_map_file()`, `pmem_mem_copy()`, `pmem_flush()`*, while the capacity device handle regular* **POSIX** *operations (*e.g., `open()`, `read()`, `fsync()`*).*

## 4.1 KEIGO design overview

Figure 11 provides a high-level view of KEIGO's architecture. It is provisioned as a user-space storage library that follows a multi-tier structure split across **DRAM**, **NVMM**, and **SSD** (*i.e.,* **SATA** or **NVMe SSDs**). **NVMM** acts as a *performance device*, which provides high performance but low capacity storage. It handles all data requests directly in user-space via memory mapping (*i.e.,* `mmap()`, `memcpy()`) and routes metadata operations to an in-kernel file system. The **SSD** acts as a *capacity device*, which provides cheaper and larger capacity storage, and handles I/O requests through standard **POSIX** system calls (*e.g.,* `read()`, `write()`). Within **DRAM**, KEIGO retains runtime system information, such as the namespace that maps **KVS** files (*e.g.,* **SST**, $C_{log}$) to their corresponding physical location and file temperature profiling. The data managed by each of these components is driven by three main techniques.

*Performance-aware data placement (Section 4.2).* On-disk **LSM** components are hierarchically split between *performance* and *capacity devices*. Performance-sensitive components whose operations directly

impact the throughput and tail latency perceived by clients, namely the $C_{log}$ and levels of the tree closer to $C_m$, are placed on **NVMM**. The capacity device on the other hand, stores the bottom **LSM**-tree levels which accommodate the colder majority of data and is where non-critical work is conducted (*i.e.,* high-level compactions).

*Persistent, read-only cache (Section 4.3).* While KEIGO's data placement scheme favors write-heavy and highly skewed workloads, read-heavy workloads can experience low performance if a substantial amount of requests are handled by the capacity device (Section 3.2). As such, KEIGO integrates a persistent, read-only cache that tracks and copies hot **SSD**-hosted **SST** files to the performance device, maximizing **NVMM**'s hit ratio and improving the **KVS** performance under low skewed, read-intensive workloads.

*Context-based I/O differentiation (Section 4.4).* To minimize intrusiveness when porting existing **KVSs** to use our solution, KEIGO exposes a **POSIX**-like interface where system calls are passed with an additional *context* field. The *context* defines the internal **KVS** operation that originated a given **POSIX** request (*e.g.,* `flush`, $C_{log}$, `compaction_Ln_Ln+1`, `foreground`), and is used to determine the device that will handle it. This is achieved by combining ideas from *context propagation* [45, 46], enabling **KVS**-level information to be propagated to KEIGO with minor code changes (see Table 3).

**Operation workflow** Figure 11 illustrates how KEIGO handles **KVS** I/O workflows. **KVS** files that contain clients' data are mainly created when flushing $C_m$ to disk (`flush`), under $C_{log}$, and during *compactions*. To create these files, the **KVS** interacts with KEIGO through `k_open()`, which combines the arguments of a regular **POSIX** `open()` call with the *context* that has triggered such operation at the **KVS** level, namely $C_{log}$, `flush`, or `compaction` (and respective levels involved). Based on the data placement scheme, KEIGO will then create the file at the corresponding device, invoking `mmap()` in the case of the performance device or `open()` in the case of the capacity device. To transparently handle subsequent I/O operations to that file, `k_open()` returns a *logical* file descriptor that can be mapped to the original file identifier, which is stored at KEIGO's namespace.

I/O operations submitted to **NVMM** are converted into their corresponding memory mapped version, servicing reads (`k_read()`) via `memcpy()` and writes (`k_write()`) through non-temporal store instructions, while operations submitted to the capacity device follow a standard **POSIX** semantics. Writes submitted to the critical data path, coming from $C_{log}$, flush, and low-level compactions, are handled by the performance device, while foreground reads are serviced from the tier where the corresponding files are cached/stored. Further, data can be directly transferred between devices when compactions that involve levels placed at different tiers are triggered (*e.g.,* $L_2$ to $L_3$, trivial moves), and when populating **NVMM**'s

cache with hot **SST** files stored in the **SSD**.

## 4.2 Optimizing writes

KEIGO's data placement mechanism aims at maximizing the performance of **LSM KVSs** by combining their distinctive access patterns with the characteristics of **NVMM** and block-based storage devices. The data managed by each device is defined by three main policies.

**1) Performance-critical components must be placed on NVMM** KEIGO integrates Lesson 1 in its design by placing the $C_{log}$ and lower levels of the tree, namely $L_0$ to $L_2$, on the performance device, ensuring that the critical data path is handled by **NVMM**. Our reasoning for this is threefold. First, writes to $C_{log}$ need to be fast, since they incur significant overhead to the critical data path, especially under crash-consistency guarantees where the $C_{log}$ is flushed after every client write to ensure data durability and ordering (Section 3.1). Second, modern workloads exhibit strong temporal data access locality, where the popularity of objects fades over time [71, 10]. This means that the most accessed keys will be stored in the most recent **SST** files, which are placed at the lower **LSM** levels. Third, background tasks that involve $C_m$ and the lower levels of the tree are prone to write stalls, especially under write-intensive workloads. Write stalls occur when flushes cannot proceed, either because flushes and low-level compactions are slow or on hold, resulting in high latency spikes and low throughput [8]. While placing these components on **NVMM** does not avoid write stalls, their duration and performance degradation is significantly hampered [72].

**2) Minimize concurrency on NVMM** Since **NVMM** provides better performance than block-based storage devices, placing as much levels on performance device as possible to maximize the **KVS** performance could be a valid assumption. However, as discussed in Chapter 3, the performance of the **KVS** under write-intensive workloads is significantly impacted with number of concurrent threads writing to **NVMM**, which can simultaneously come from writes to $C_{log}$, flushes, and compactions from different levels. This is particularly detrimental when higher levels of the **LSM** are placed on **NVMM**, which perform larger compactions and contend the device for longer periods. Moreover, read performance is also severely impacted when concurrent writes are issued to the device, being exacerbated with increasing number of writers [65].

As such, in KEIGO, the maximum number of levels placed in the performance device is not determined by the space left on the device, but rather, by the number of writer threads that can potentially work in parallel. This means that under write-intensive workloads, by storing a smaller subset of the overall

**KVS** dataset on the performance device, KEIGO can reduce **NVMM** write contention and achieve better performance.

**3) Non-critical work is offloaded to SSD** The remainder levels of the **LSM** tree (*i.e.,* $L_3$ to $L_N$) are placed on the capacity device. As new objects are inserted or updated in the **KVS**, older values are pushed down the stack and stored at the bottom levels, thus accommodating the main bulk and colder portion of the dataset. With this partitioning, KEIGO offloads non-critical compactions to the capacity device (*i.e.,* compactions that occur at the higher levels of the tree and do not directly impact clients' performance [8]), reducing write contention on **NVMM**.

## 4.3 Optimizing reads

KEIGO's data placement scheme favors write-intensive and highly skewed workloads, whose workflows are mainly serviced from the performance device. However, as observed in Section 3.2, under read-heavy workloads that exhibit medium skew or uniform distributions, the majority of requests will be made over the capacity device. While placing more levels on performance device improves the performance of read-dominated workloads, it would severely impact the performance of write-heavy ones (Lessons 1 and 3).

KEIGO overcomes this challenge by implementing a persistent cache that continuously tracks and copies hot **SSD**-hosted **SST** files to **NVMM**. Targeted files will temporarily reside in **NVMM** as a *read-only copy*, while the *original* version is still persisted on **SSD**. Copied files are created in read-only mode since in traditional **LSM KVSs** (*e.g.,* RocksDB) all files become immutable after being fully written [19]. Files are removed from the cache either when the *original* file is deleted (*e.g.,* due to compactions) or when a hotter **SST** file needs to be copied and there is no more space left in the device. Further, while read performance is severely degraded under concurrent **NVMM** writes, write performance exhibits minimal performance variation under multiple concurrent reads over **NVMM** [65, 9].

KEIGO's caching process is unrolled in three main phases (as depicted in Algorithm 1), which answer three important questions: *(1)* what files should be copied; *(2)* when should the files be copied; and *(3)* how should the actual process of copying files be made in the presence of workloads with different read-write proportions.

**1) File temperature profiling** KEIGO determines which files should be copied to the persistent cache by keeping track of the number of accesses to **SSTs**. Access counters are maintained at KEIGO's names-

pace and are updated by the **KVS** foreground threads (2–3). To prevent copying stale data, which can happen with changes in the data access pattern or during compactions, we take into account the *age* of each **SST** file and decrease the counter by following an exponential back-off approach every time a file is copied (11–12).

**Algorithm 1** Pseudo-code of the caching process' workflow.

```
1  Client thread: //foreground reads
2    namespace[logicalfd].access++
3    inc(nvmm_ctr); inc(total)
4  Hit ratio thread:
5    while(true)
6      if((nvmm_ctr / total) < ε)
7        //select hottest SST file from CD
8        hot_sst = pick_sst()
9        queue.push(hot_sst)
10       //decrease counters to prevent staleness
11       foreach (sst in namespace)
12         sst.dec()
13     else sleep(t)
14 Writer threads: //flush and compactions over NVMM
15   if WRITE inc_writers()
16   if CLOSE dec_writers()
17 Caching threads:
18   while(true)
19     writers = get_writers()
20     while(queue.empty() OR writers > σ)
21       wait()
22     hot_sst = queue.pop()
23     //create SST copy at NVMM
24     create_nvmm_copy(hot_sst)
```

**2) NVMM hit ratio maximization** KEIGO determines when **SST** files should be copied to the cache by monitoring the hit ratio of **NVMM**'s foreground reads. To achieve this, KEIGO uses a dedicated background thread that continuously computes **NVMM**'s read hit ratio and triggers a copy when the value is below a

certain threshold (given by $\varepsilon$). When a copy is triggered, the thread picks the most accessed **SSD**-hosted **SST** file and inserts it in an internal queue (5–9). The monitoring granularity and the hit ratio threshold are system parameters that can be configured by users.

**3) Concurrency-aware copying** KEIGO uses a dedicated thread pool to cache multiple files in parallel. However, this operation must be done carefully to avoid exacerbating **NVMM**'s write concurrency problem, and significantly impact the **KVS** performance. Instead, KEIGO keeps track on the number of active writers that are in the system at any given time, including $C_{log}$, flush, and compactions that target **NVMM**-hosted levels (15). Based on this, KEIGO automatically adjusts the number of caching threads to ensure that the number of total threads writing to **NVMM** (*i.e.,* $C_{log}$ + flush + compaction + caching threads) does not exceed $\sigma$ (18–24). Based on the results obtained in Section 3.3, we set $\sigma$ to 5. The number of writers is decreased when the corresponding background tasks finish (*i.e.,* `close()`) (16). Currently, KEIGO does not preempt ongoing file copies, which can sporadically lead to the number of active threads exceed $\sigma$ for a short period of time (few milliseconds). We defer the support of preempting ongoing copies to future work.

Moreover, we chose not to implement the read-only cache in main memory. Our reasoning is that existing **DRAM** space (*i.e.,* **OS** page cache) can be used to accommodate other sporadic requests made to data that resides in the capacity device.

## 4.4 Context-based I/O differentiation

KEIGO strikes a balance between being too high-level (like general-purpose storage systems) or too specific (like **LSM KVSs** built specifically for certain hardware). The origin of **KVS** operations (*e.g.,* $C_{log}$, flush, $L_0$ to $L_1$ compaction) is propagated to the storage, allowing the same level of control and performance as **LSM**-specific optimizations while imposing minimal code changes. KEIGO takes ideas from *context propagation*, a commonly used technique that enables a system to pass additional request information along its execution path [46, 45, 69], and applies them to determine which device should handle each I/O request. The process for differentiating requests in KEIGO is twofold.

**Instrumentation** First, KEIGO needs minimal instrumentation on the data path of **LSM** foreground and background work, including the $C_{log}$, flushes, and compactions at different levels. Whenever these operations are triggered, a tag (or *context*) associated with the corresponding operation type (*e.g.,* $C_{log}$, `flush`, `comp_l0_l1`) is registered at the process's address space through thread-local variables.

**Execution** During execution, when a new file is created, a `k_open()` call is sent to KEIGO, with the original arguments of `open()` and the *context* that originated the request. This *context* determines the device that will persist the file, according to the data placement scheme. Subsequent file requests (*e.g.,* `k_read()`, `k_write()`, `k_close()`) do not need to pass the operation context, as they can access the file through KEIGO's namespace.

## 4.5 Implementation

We implemented a prototype of KEIGO in 3K LoC, in C++. KEIGO is provided as a user-space library, extending 25 **POSIX** calls, namely, `unlink()`, `open()`, `close()`, `mmap()`, `lseek()`, `read()`, `pread()`, `write()`, `fsync()`, `fdatasync()`, `readahead()`, `fcntl()`, `posix_fadvise()`, `pwrite()`, `ftruncate()`, `fallocate()`, `sync_file_range()`, `fstat()`, `fread_unlocked()`, `fseek()`, `fclose()`, `fdopen()`, `feof()` and `clearerr()`. We found that supporting this set of calls is sufficient to enable KEIGO over **LSM**-based **KVSs** (see Section 5.3).

By expanding/reimplementing these **POSIX** calls logic to include a data placement mechanism, we are able to place the **KVS**'s files in the most apropriate device in a non-intrusive and portable way. The **KVS** is unaware of the changes, while our middleware is doing the work behind the scenes.

Orthogonally, one could use LD_PRELOAD mechanism to reimplement the aformentioned **POSIX** system calls. However, this mechanisms is ineffective for the goal we aim to achieve. Specifically, `LD_PRELOAD` is an OS primitive that enables assigning a custom library to be loaded before any other library that is linked to an executable (including the default C runtime libraries). The symbols of the preloaded library are bound first. This way developers can override the default behavior of stock unix functions and replace them with their own.

Additionally, this method is forced to respect the exact **POSIX** interface, and will not allow to pass aditional parameters as context to the library. However, in order to better assess where the data should be placed (*e.g.,* when opening a file) it is necessary to know some context bound to the **KVS**'s internal logic (*e.g.,* is this `open` being called during a flush, compaction, etc.).
We now present other details of KEIGO's implementation.

**NVMM driver** The **NVMM** driver is implemented using Intel **Persistent Memory Development Kit (PMDK)** (libpmem v1.11) [32]. **POSIX** operations handled by **NVMM** are converted into their corresponding memory mapped version. Specifically, `open()` and `close()` system calls are handled using `pmem_map_file()` and `pmem_unmap()` routines, respectively. Reads are serviced via `memcpy()`,

while writes are made with `pmem_memcpy()` using `ntstore` instructions to avoid polluting the processor cache. Files are mapped to memory with a fixed but configurable size. $C_{log}$ files are recycled for future usage. Requests submitted to **SSD** follow a standard **POSIX** semantics.

**Handling `open`** Since files are managed with different identifiers (namely, memory addresses for **NVMM** and file descriptor for **SSD**), `k_open()` returns a *logical* file descriptor to the **KVS**, which is then used for the subsequent system calls. While files stored in **NVMM** and **SSD** are managed with different types of file identifiers (namely, memory address and file descriptor), `k_open()` returns a *logical* file descriptor to the **KVS**, which will be agnostically used by any subsequent system call. KEIGO maintains a thread-safe queue populated with integers in incremental order that dequeues an element on `k_open()` and enqueues it on `k_close()`.

Whenever a open is issued to the library, KEIGO performs the following steps:

- Firstly, the library checks whether the file already exists in the namespace or not. To check for file existance and its redirection, KEIGO mantains an hashmap that maps every **SST** file number to its corresponding device. If the entry does not exist, KEIGO analyses the the context passed through the executing thread and determines in which device it has to be created (*e.g.,* a flush thread indicates that the file is going for $L_0$, a compaction thread for the levels above). If the file does exist, the context passed through the system call itself also hints to the type of accesses that will be made to the file (*e.g.,* read-only, write-only, etc).
- A logical file descriptor is returned upon the `k_open()` call, even if the file is mantained in **NVMM**, which manages files with memory addresses. This is done to ensure that the **KVS** can agnostically use the same file descriptor for any subsequent system call. One could argue that the logical file descriptors should only be issued by the `k_open()` if the file is going to **NVMM**, since an **SSD**-stored file will behave in the default **POSIX** norm. However, this would require a more complex consistency control between the file descriptors issued by the **OS** and our logical ones, as this could lead to duplicated file descriptors — the logical ones maintained for **NVMM**-hosted files, and those generated from the original **POSIX** open() call for files stored in **SSD**. This design also makes possible using other interfaces apart from **POSIX** on the **SSD** device, since it allows for custom logic to be added to KEIGO for dealing with **SSD**-stored files (*e.g.,* **Storage Performance Development Kit (SPDK)**).
- KEIGO associates the logical file descriptor with a `File Object`. There are different versions of this `File Object`, each one associated with a way of handling a file (*e.g.,* one associated with the $C_{log}$ file on **NVMM**, one with **SST** files on **NVMM**, on with **SST** files on **SSD**, etc). This objects contain

the custom behavior that will be executed for subsequent system calls targeting that file descriptor. They also store some important data, such as the original file identifier (*e.g.,* memory address, file descriptor), the pathname, the context that originated the file creation, and the file's caching status.

This steps ensure that when subsequent system calls are issued to the library, the `File Object` associated with the file descriptor is retrieved and the custom behavior is executed.

**Handling `fsync`** KEIGO provides similar consistency guarantees as **POSIX**, making data durable upon explicit synchronization. Specifically, it handles `fsync()` and `fdatasync()` calls through the combination of `pmem_flush()` and `pmem_drain()` instructions, flushing dirty data to **NVMM** and blocking until it is effectively persisted.

**In-memory structures** KEIGO's namespace is implemented in a concurrent hashmap that provides a mapping between the *logical* file descriptor and an object that maintains information about the actual file (*e.g.,* original file descriptor or memory address, pathname, file profiling). Entries are atomically updated at file creation (*i.e.,* flush, compactions, $C_{log}$), when data is transferred between devices (*i.e.,* trivial moves and caching), and file removal.

Additionally, in order to support system calls that do not use a file descriptor (*i.e.,* `unlink()`) a mapping between the **SST** numbers and the corresponding device also has to be mantained. For `unlinks`, the **KVS** will always issue the removal of the file's original path, unaware that the file might actually be stored on **NVMM**. This mapping is used to redirect the unlink to the correct device and avoiding it from issuing an error.

**Context propagation** To propagate the internal **KVS** operation that originated a given **POSIX** request to KEIGO, we use the **OS**'s thread-local storage mechanism [20] by storing this information in a variable residing in the local memory of each thread, eliminating contention or potential race conditions.

**NVMM cache** The cache uses a dedicated thread pool that copies **SSTs** in background and a separate thread that tracks **NVMM**'s read hit ratio. The hit ratio threshold ($\varepsilon$) is defined by a fixed, configurable parameter. On file copying, KEIGO prefetches targeted **SSD** files to the page cache using `fadvise()` and `readahead()`. After its completion, KEIGO hints the **OS** to evict the pages that hold prefetched data.

The caching process only starts after the hit ratio falls below the threshold ($\varepsilon$), meaning the number

of accesses on **NVMM** is lower than what we believe is the most suitable for the current workload. Each **SST**-associated `File Object` keeps track of the number of times the `k_read()` is issued for its corresponding file. Periodically, a custom thread sorts the files by the number of accesses and selects the hottest one. As expected this is not the quickest method to pinpoint a hot file, despite its extreme accuracy. Other algorithms exist that perform the selection in a quicker faction by sacrificing some of the accuracy when searching for the "hottest" file (*i.e.,* might not be the hottest file, but for a small margin). Our preliminary experiments showed that by giving emphasys to accuracy the throughput actually accelerates quicker than by selecting files quicker but with less hotness sensitivity, specially when the workloads are highly skewed. This makes sense since a **KVS** can manage thousands of files but the hotness can be extremely concentrated.

To ensure that hotness does not accumulate over time and that hot files can eventually become cold has the workload develops, the sorting thread also applies a decay factor to the number of accesses of each file everytime it goes through an iteration. This way, files that were hot in the past but are not being accessed anymore will eventually become cold and might be evicted from the cache.

After being selected, the file is placed in a queue where it will be eventually picked by a thread amongst the thread pool. These threads are the ones responsible for copying the file to **NVMM** and updating the metadata of the library (*i.e.,* ensuring new read accesses are issued to the file in cache, and not to the original file in **SSD**). This decision in implementation follows the lessons learned in Chapter 3, where we observed that the performance of the **KVS** is severely impacted when multiple threads are writing to **NVMM**. Therefore, by using a thread pool we can control the number of concurrent writes to **NVMM**. During certain periods, a large number of files can be selected for caching, but the copying threads will be effectively rate-limiting the number of writes.

This asynchronicity between choosing a file to cache and actually copying it, adds additional responsibility to the copying threads:

- Before copying a file there's a possibility of the original file having been deleted. This can happen when compactions or trivial moves occur during the copying process. Therefore, the copying thread must check whether the file still exists or not. If it does not, the thread removes the file from the queue and continues with the next one.
- At the end of the copy, the thread will also check if the original file is missing, due to the same reasons as above. If it is, the thread will also remove the file from the **NVMM**.
- During the copy process, the threads do not need to check for the file existance since the **OS** will not allow the deletion of a file that is being used by another process. However, a future improvement to the

| | Lines added/changed | | |
|---|---|---|---|
| | RocksDB | Speedb | LevelDB |
| Targeted code base size | ≈583K [22] | ≈545K [60] | ≈23K [28] |
| Initialize/destroy KEIGO | 2 | 2 | 2 |
| Register threads | 2 | 2 | 2 |
| Propagate *context* | 2 | 2 | 1 |
| Replace system calls | 44 | 68 | 35 |
| Trivial moves | 7 | 7 | – |
| **Total** | **57** | **81** | **40** |

Table 3: *Lines of code added to RocksDB, Speedb, and LevelDB.*

library could be made, detecting if the **KVS** has issued an `unlink()` for a file that has an ongoing copy.

After the copy is finished, the thread will associate the logical file descriptor of the file to a new **NVMM**-aware *File Object*. This ensures that from now on all of this file's behavior will be handled by the **NVMM** driver.

**Porting LSM-based KVS** We integrate KEIGO over three **LSM**-based **KVS**: *RocksDB* [22], *Speedb* [60], and *LevelDB* [28]. Table 3 summarizes the code changes required to port the three systems, which include initializing and destroying KEIGO, registering thread IDs and updating the thread-local variable for context propagation, and replacing **POSIX** system calls with those supported by KEIGO.

**Trivial moves** While trivial moves in RocksDB (and Speedb) are logic, files need to be physically moved if the end level is placed on a different device. We instrument RocksDB's data path responsible for trivial moves; whenever a trivial move is triggered, the file is placed in a queue where a dedicated thread pool performs the migration. The namespace is updated with the new location and file identifier of the file, and the copy held in **NVMM** is removed.

**Tunable parameters** KEIGO provides a number of tunable parameters, including the data placement scheme ($C_{log}$ and $L_0$-$L_2$ on **NVMM**), `mmap()` size (same as **SST** size), read hit ratio ($\varepsilon = 0.7$), size of the caching thread pool (2), etc. Our experiments show that the default values work well in the conducted testing scenarios, presenting a good balance between read and write performance. System designers can change these according to their use case.

# Chapter 5

# Evaluation

Our evaluation sets out to answer the following questions:

- What is the performance of a KEIGO-enabled RocksDB under different storage scenarios (Section 5.1)?
- How does KEIGO compare to other hierarchical storage systems (Section 5.1)?
- How do the different techniques employed in KEIGO contribute to the overall performance (Section 5.1)?
- What is the performance of a KEIGO-enabled RocksDB under production workloads (Section 5.2)?
- Can KEIGO be used to improve the performance of other LSM KVS (Section 5.3)?

**Experimental setting** The hardware, OS, and experimental environment configurations are the same as in Chapter 3.

**Baselines** We evaluate and compare two groups of systems: (1) general-purpose solutions, including KEIGO, used together with RocksDB, and (2) state-of-the-art LSMs built from the ground up for heterogeneous storage. One of our evaluation goals was to compare KEIGO against state-of-the-art general-purpose tiered file systems, such as Ziggurat [75], TPFS [76], or Strata [38]. Unfortunately, the code for Ziggurat and TPFS was not publicly available at the time of this writing. Furthermore, our experiments using Strata, P2Cache [43], and SPFS [63] crashed after a few operations, not allowing us to obtain meaningful results. We integrate RocksDB with the following general-purpose systems:

- **ext4** corresponds to an ext4 file system mounted over an **NVMe** or **SATA SSD** (RocksDB default setup [22]).
- **ext4-DAX** corresponds to an **NVMM** device accessed by ext4, with direct access (DAX) enabled. This is the most straightforward way in which an LSM might use **NVMM**.
- **OpenCAS** [51] is a kernel module for Linux that provides transparent and persistent caching to applications. We configured OpenCAS to use NVMe or SATA **SSDs** as the capacity devices, and **NVMM** as persistent cache. To ensure low client-side write latency, we use OpenCAS in *write-back* mode, mean-

| Workload | Description |
|---|---|
| A | *Write-intensive:* 50% updates, 50% reads |
| B | *Read-intensive:* 5% updates, 95% reads |
| C | *Read-only:* 100% reads |
| D | *Read-latest:* 5% updates, 95% reads |
| E | *Scan-intensive:* 5% updates, 95% scans |
| F | 50% read-modify-writes, 50% reads |

Table 4: *YCSB core workloads description.*

ing the system writes the data first to the cache and acknowledges to the application that the write is completed before the data is written to **SSD**. Periodically, those writes are *written back* to the capacity disk.

- **KEIGO** as a middleware built specifically for LSM KVS.

We also compare with the following state-of-the-art LSM KVS, built for heterogeneous and **NVMM**-based storage.

- **MatrixKV** [70] is an LSM KVS that places all levels on **SSD**, except for $L_0$ which is stored on **NVMM** in a custom matrix-like data structure. $C_{log}$ is also persisted in **NVMM**. In addition, the system optimizes $L_0$-$L_1$ compactions. We used the default system configurations.
- **NoveLSM** [34] is an LSM KVS that uses **NVMM** to store a large, mutable, and durable $C_m$, avoiding the use of $C_{log}$ and costly serialization operations. SST files are also stored in **NVMM**. We used the default system configurations.

## 5.1 Microbenchmarks: YCSB

**Workload description** Table 4 shows a summary of the YCSB core workloads. We evaluate the uniform, and Zipfian distributions for all the core workloads. We use 16B keys and 1024B values. For the Zipfian distribution, we evaluate two key popularity skews: 1% of keys accessed 99% of the time ($zipf_{99}$), and 20% of keys accessed 80% of the time ($zipf_{80}$).

**Single-threaded YCSB** Figure 12 presents the YCSB workloads, running with 1 client thread, under uniform and $zipf_{99}$ distributions for RocksDB+ext4-DAX, RocksDB+OpenCAS, RocksDB+KEIGO, MatrixKV, and NoveLSM. The dataset size is 50 GB, thus fully fitting in **NVMM**. The dataset was constrained due to scalability limitations of MatrixKV, which crashed while loading datasets larger than 50 GB. As ex-

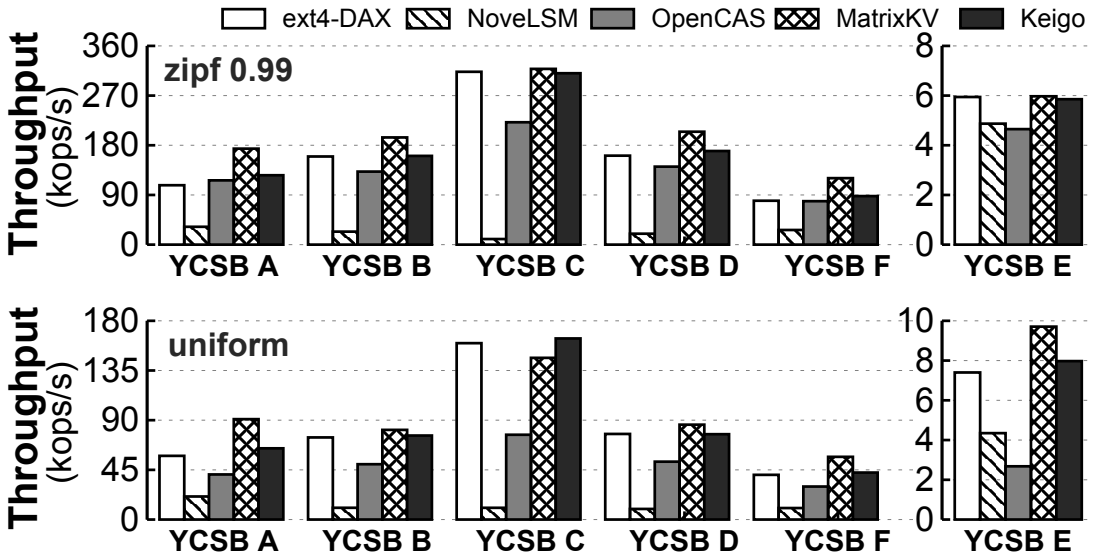



Figure 12: **Single-threaded YCSB workloads.** *We compare RocksDB over ext4-DAX, OpenCAS, and* KEIGO*, with the standalone NoveLSM and MatrixKV.*

pected, MatrixKV outperforms all systems in all workloads except for YCSB C uniform, owing to its re-implementation of compactions. Despite their more general approaches, KEIGO and ext4-DAX achieve almost on-par performance with MatrixKV in most workloads, while the throughput overhead is never higher than 29% and 38% respectively. We report similar conclusions to $\text{zipf}_{80}$ distribution.

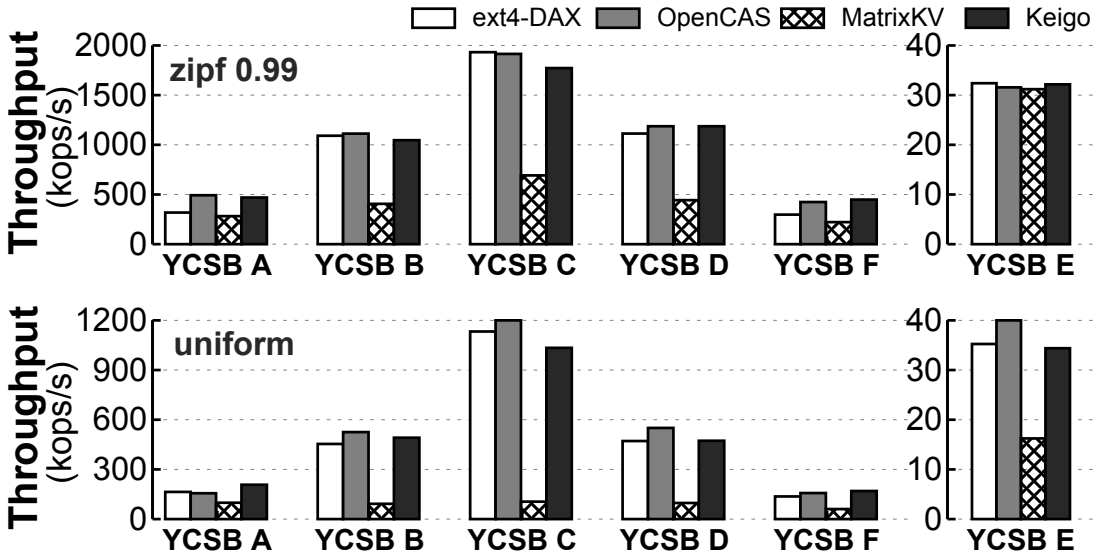



Figure 13: **Multi-threaded (8 client threads) YCSB workloads.** *We compare RocksDB over ext4-DAX, OpenCAS, and* KEIGO*, with the standalone NoveLSM and MatrixKV.*

**Multi-threaded YCSB** Figure 13 presents the core YCSB workloads running with 8 client threads. Due to scalability bottlenecks in multi-threaded workloads, we leave out NoveLSM for the remainder of this sec-

tion. Maybe surprisingly, the general-purpose solutions outperform MatrixKV by up to 12×. MatrixKV's limitation stems from its inflexible approach to data placement. Unlike KEIGO, MatrixKV uses a predetermined data placement, which prioritizes $L_0$ and $L_1$ data movement, and does not account the fact that higher levels can also hold hot data. In contrast, KEIGO's performance is on-par with that of ext4-DAX and OpenCAS. Note that KEIGO's tail latency ($99^{th}$ percentile) is low and stable; we measured under 50 $\mu$s for read-dominated workloads and under 75 $\mu$s for write-dominated workloads.

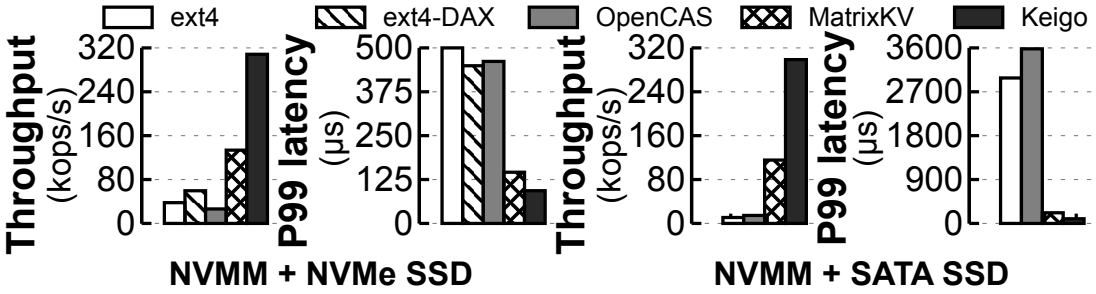



Figure 14: **Durable $C_{log}$ writes.** *We compare RocksDB over ext4, ext4-DAX, OpenCAS, and* KEIGO*, with MatrixKV under YCSB A.*

**Durable $C_{log}$ writes** Figure 14 presents a write-intensive workload, where all writes to $C_{log}$ are immediately persisted via `fsync()`. KEIGO outperforms the next best competitor by 2.2× (MatrixKV), while achieving 27× higher throughput then ext4. Every write goes to the $C_{log}$ placed on the **NVMM** which is PMDK-optimized by KEIGO, unlike in ext4-DAX where requests must transverse the kernel stack. OpenCAS requires flushing dirty data from **NVMM** to the capacity device, justifying the observed slowdown.

**Scalability with dataset size** Figures 15 and 16 throughput of RocksDB enhanced by ext4, OpenCAS, and KEIGO, with 200GB and 400GB datasets, respectively. Given the larger dataset size, this experiment exhibits KEIGO's leverage of deeper storage hierarchies, as data does not fully fit into **NVMM**. This is where KEIGO truly shines, as it dominates in the majority of the workloads, offering better throughput by up to 12× over ext4, and up to 2.8× over OpenCAS on NVMe drives. The only scenario where OpenCAS outperforms KEIGO is for the read-only workload backed by **SATA SSD**, with uniform and $zipf_{80}$ distributions. This is expected since read-dominated workloads benefit from caching (Section 3.1). This performance difference is mainly caused by OpenCAS' promotion and eviction policies, being more aligned with the performance characteristics of **SATA SSD** devices (e.g., poor device-level parallelism and I/O interference) than Keigo's. OpenCAS also does a better job at utilizing **NVMM**'s total capacity, whereas KEIGO reserves a portion of **NVMM** for caching prior to the workload execution.

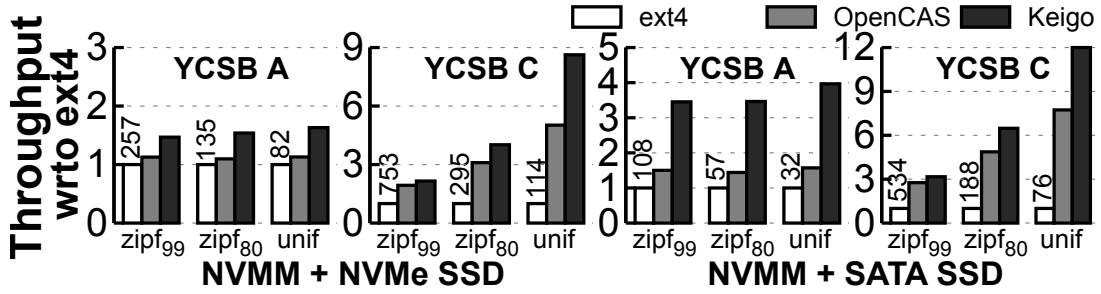



Figure 15: **200 GB dataset (8 client threads) YCSB workloads.** *We compare RocksDB over ext4, OpenCAS, and* KEIGO*, under NVMM+NVMe* **SSD** *and NVMM+SATA* **SSD** *storage setups.*

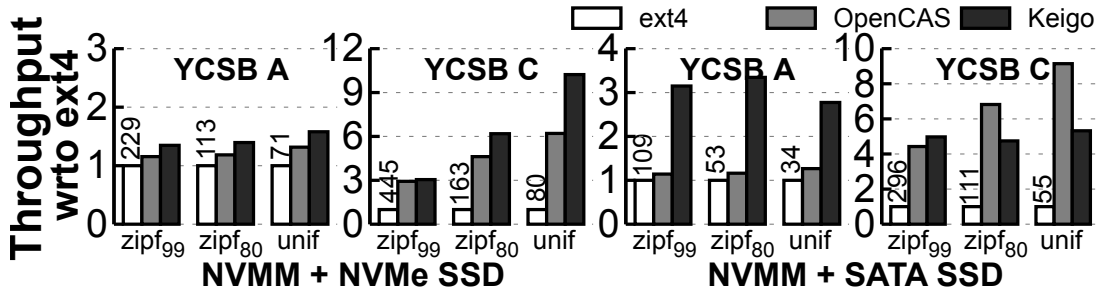



Figure 16: **400 GB dataset (8 client threads) YCSB workloads.** *We compare RocksDB over ext4, OpenCAS, and* KEIGO*, under NVMM+NVMe* **SSD** *and NVMM+SATA* **SSD** *storage setups.*

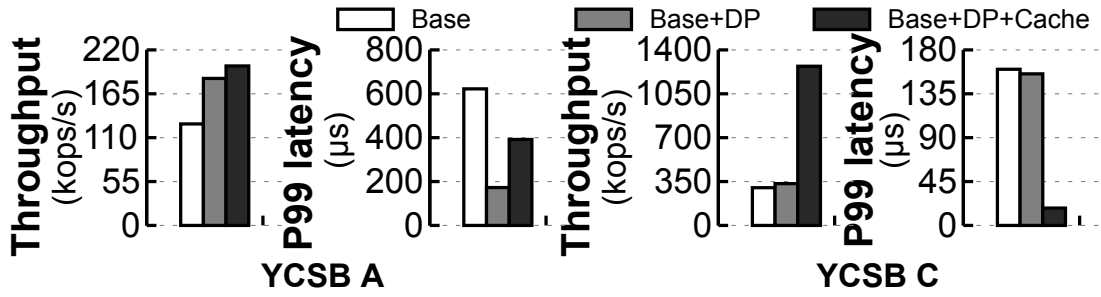



Figure 17: **Performance breakdown.** KEIGO *over RocksDB performance breakdown under YCSB A and C with a zipf*$_{80}$ *distribution.*

**KEIGO performance breakdown** Figure 17 shows the performance breakdown of KEIGO's techniques, in the core YCSB read- and write-intensive scenarios. This breakdown shows the complementarity of KEIGO's techniques, demonstrating how the full system is greater than the sum of its parts. The *Base system* represents vanilla RocksDB, with only the $C_{log}$ moved to **NVMM**, and the rest of the data on NVMe **SSD**. Next, in the *Data placement (DP)* version, we add performance-aware data placement (Section 4.2) to the previous setup. The improved data placement significantly reduces tail latency in the write-dominated workload, by up to 3×, and increases throughput by 45%. Finally, we add the *Caching mechanism (Cache)* (Section 4.3) on top of the previous setup. As shown in Chapter 3, caching dramat-

ically improves performance in the read-dominated workload, obtaining a 3.7× throughput increase and an order of magnitude tail latency decrease.

## 5.2 Production workloads

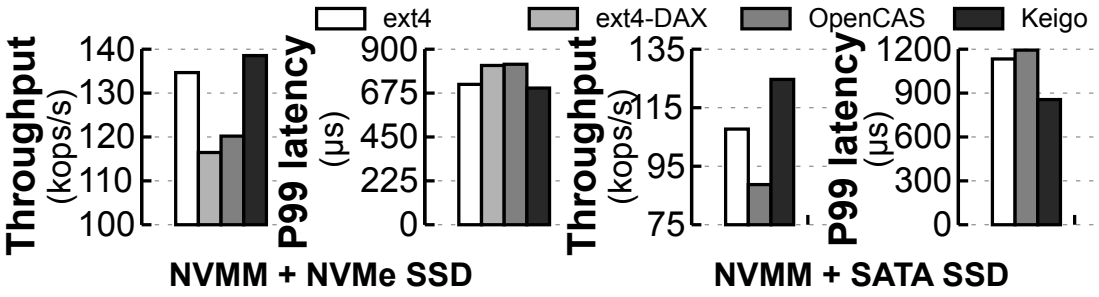



Figure 18: **Meta production workload.** *We compare RocksDB over ext4, ext4-DAX, OpenCAS, and* KEIGO*, under NVMM+NVMe SSD and NVMM+SATA SSD storage setups.*

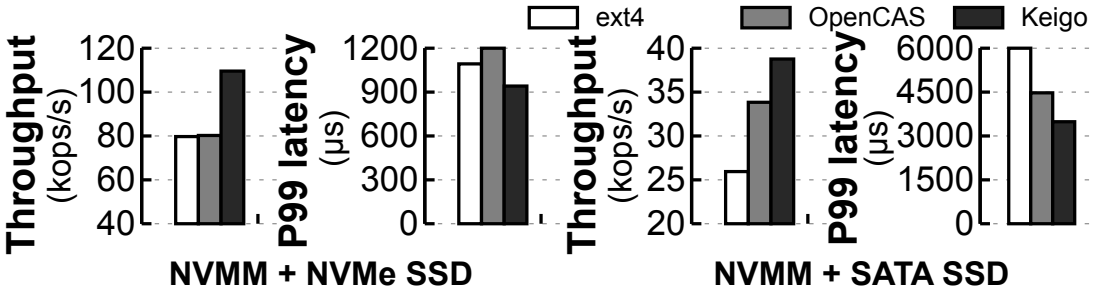



Figure 19: **Nutanix production workload.** *We compare RocksDB over ext4, OpenCAS, and* KEIGO*, under NVMM+NVMe SSD and NVMM+SATA SSD storage setups.*

**Meta workload description** We use the `prefix_dist` Meta production workload [10], a read-dominated workload that encompasses a combination of read, write, and scan operations at a ratio of 83:14:3 and exhibits realistic key hotness patterns. We increased the workload intensiveness using an augmented `sin_d` and reduced `sin_b` (by a factor of $10,000$ and $100,000$, respectively), while keeping other parameters unchanged, as instructed in [10]. Experiments were conducted over a 50M key-value pairs dataset.

**Meta workload results** Figure 18 shows RocksDB using ext4, ext4-DAX, OpenCAS, and KEIGO throughput and tail latency for two storage hierarchies (**NVMM** over NVMe **SSD** and SATA **SSD**). Due to its data placement and caching policy, KEIGO outperforms all baselines, achieving up to a 19% throughput increase over ext4 and up to 40% over OpenCAS.

**Nutanix workload description** The Nutanix workload [8] is a write-intensive workload that encompasses a combination of read, write, and scan operations at a ratio of 40:58:2. The items requested during these operations vary in size, ranging from 100B to 4KB, with a median size of 400B. Experiments were conducted over a 500M key-value pairs dataset. ext4-DAX was not included since the **NVMM** does not have enough space to hold the entire dataset.

**Nutanix workload results** Figure 19 shows the throughput and tail latency results for the Nutanix workload, on the same systems. KEIGO outperforms all baselines. As this workload is write-dominated, KEIGO's data placement policy and compaction concurrency control allow it to achieve up to a 1.5× throughput increase over general-purpose solutions.

## 5.3 KEIGO portability

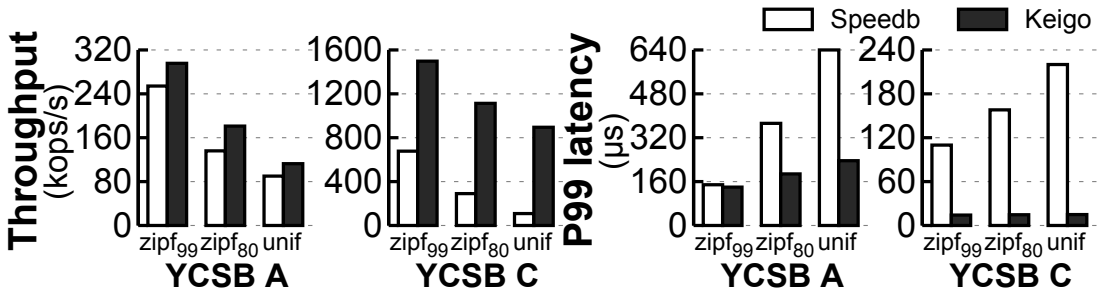



Figure 20: **Speedb integration with KEIGO.** *We compare Speedb over ext4, and* KEIGO *for YCSB A and YCSB C.*

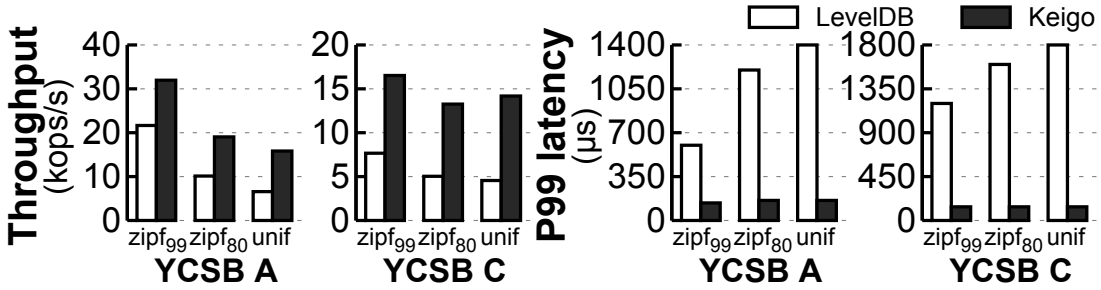



Figure 21: **LevelDB integration with KEIGO.** *We compare LevelDB over ext4, and* KEIGO *for YCSB A and YCSB C.*

We finally show how other LSM KVS benefit from KEIGO. With less than 100 LoC (Table 3) we integrate the KEIGO middleware with two other popular LSM KVS: Speedb [60] and LevelDB [28]. We ran YCSB A and YCSB C workloads.

Figures 20 and 21 compare the vanilla systems (using ext4) and KEIGO-enhanced systems for Speedb and LevelDB, respectively. KEIGO improves throughput and tail latency in all scenarios. For Speedb, KEIGO

accelerates YCSB A up to 1.3× and YCSB C up to 8.2× while decreasing $99^{th}$ percentile latency by up to 2.7× and 14.7× respectively. For LevelDB, KEIGO accelerates YCSB A up to 2.4× and YCSB C up to 3.1×, with similar gains in tail latency as for Speedb.

## 5.4 Evaluation Summary

We next present the main takeaways from our evaluation.

**Scalability and Dataset Size** With small datasets, KEIGO competes with general-purpose solutions and significantly outperforms NoveLSM and MatrixKV. KEIGO truly excels in scenarios where datasets surpass the capacity of a single storage device, showcasing up to a 12× throughput increase over general purpose solution. This is mainly due to its intelligent data placement mechanisms, which enable it to seamlessly distribute and manage data across multiple devices. General-purpose solutions, such as ext4 and OpenCAS, are not aware of the underlying storage hierarchy nor the **LSM KVS**'s data placement and internal operations, and thus cannot leverage the full potential of the storage hierarchy.

**Production Workloads** Our evaluation proves that KEIGO can also perform well in realistic production workloads. In the Meta workload, KEIGO outperforms all baselines in both throughput (up to 40%) and tail latency (up to 28% lower).The Nutanix workload shows that KEIGO excels in a write-intensive scenarios, achieving up to a 1.5× throughput increase over general-purpose solutions.

**Durable $C_{log}$ Writes** KEIGO significantly outperforms competitors in a write-intensive workload where all writes to $C_{log}$ are immediately persisted, achieving up to a 27× higher throughput than ext4. Every write goes to the $C_{log}$ placed on the **NVMM** which is **PMDK**-optimized by KEIGO, unlike in ext4 where requests must go through the kernel stack. OpenCAS requires flushing dirty data from **NVMM** to the capacity device, justifying the observed slowdown.

**Portability** KEIGO successfully integrates with modern, production-ready **LSM KVSs** (including RocksDB, Speedb, and LevelDB) with minimal code changes (up to 81 lines of code) with minimal lines of code, resulting in improved performance for the integrated systems. This distinguishes KEIGO from other solutions which are built from the ground up for heterogeneous storage and require extensive code modifications to integrate with other **KVSs**.

# Chapter 6

# Conclusion

As the trend of increasing size of datasets in storage-based applications continues, there has been a growing interest in methods for storing and processing large amounts of data efficiently. **LSM KVSs** have been highly adopted due to their design that allows for high throughput. Recently with the advent of **NVMM**, new storage technologies have emerged that offer faster access times compared to traditional storage devices, thus accelerating **KVSs**. In this dissertation we show that to fully optimize **KVSs** over **NVMM** it is not enough to simply run the application on top of it. Our study shows that for write-intensive workloads, placing the entire dataset on **NVMM** is detrimental, degrading throughput up to 35% when compared to just placing a small subset of data items. We show that in heterogeneous storage with **NVMM** and block-based devices, caching is the best approach for read-dominated workloads; on the other hand, for write-dominated workloads, the tiering approach presents better performance, while storing the entire dataset on **NVMM** actually degrades it. Current solutions focus on one of this single methods.

To address these challenges, we designed KEIGO, a novel storage middleware that helps **LSM**-based **KVSs** efficiently use storage hierarchies composed of **NVMM** and block-based devices by being aware of the different I/O operations done by the **KVS** (*e.g.,* foreground read and write requests, and background flushes and compactions) and the characteristics of the underlying devices. KEIGO ensures that all operations done on the critical data path are handled by **NVMM** and offloads non-critical work to the block-based device. Simultaneously, it makes sures that the **NVMM** device is not overloaded by concurrent writes, while maximizing **NVMM** hit ratio to improve read performance.

KEIGO offers a balance between general-purpose hybrid storage systems and completely redesigned **KVS** systems with native support for heterogeneous storage devices. It requires minimal code modifications to integrate into production-ready **LSM KVSs**.

We integrated KEIGO with RocksDB, Speedb, and LevelDB, enhancing their throughput by as much as 12.4×. Our experiments also show how it can reduce tail latency by up to 21.3× over both general-purpose storage solutions and **LSM KVSs** built from the ground up for hierarchical storage.

## 6.1 Future Work

KEIGO was built with an extensible design in mind, which can open new prospects for future work:

**Expand the storage hierarchy.** KEIGO can be extended to support more device levels, which allow for more fine-grained control over the data placement and eventually an even higher degree of scalability. This expansion introduces an added layer of complexity to the system architecture, as each additional device level requires intricate integration and management within the existing framework. The challenge lies in maintaining optimal performance and ensuring seamless communication among the various levels with possibly new coordination mechanisms. The payoff is a system that not only accommodates a broader range of hardware configurations but also offers enhanced adaptability to evolving technological landscapes.

**Support other storage devices.** With the discontinuation of Intel's Optane DC Persistent Memory, there is a strong incentive to support other **NVMMs** technology (*e.g.,* IBM PCM [27], Fujitsu ReRAM, Everspin STT-RAM [5]). Diversifying the range of supported **NVMMs** not only mitigates the impact of the Optane discontinuation but also opens up opportunities for leveraging the unique advantages offered by different non-volatile memory technologies. It is important to note that even though the underlying logic for **NVMM** support may be implemented, different **NVMMs** may exhibit distinct idiosyncrasies and behaviors. More recently, with the advent of Compute Express Link (CXL) introducing a promising avenue with its potential to function as a **NVMM**, it is important to consider the implications of supporting such a device in KEIGO.

**Different placement policies.** Since performance might not always be the main concern for some organizations or applications, new placement policies could be added to KEIGO to fit into other scenarios. Introducing placement policies tailored for sustainable computing involves a delicate balance between maintaining acceptable performance levels and reducing the carbon footprint of data storage operations. If the main concern is to ensure that the data is safely stored, we could employ a policy that prioritizes dependability over speed. Our system could then be adapted to support a wide range of placement strategies (*e.g.,* replication across both local and remote devices, storage deduplication, etc.). Implementing such policies adds a layer of versatility to the KEIGO framework, making it adaptable to a spectrum of organizational priorities and environmental considerations.

# Bibliography


[1] Hiroyuki Akinaga and Hisashi Shima. Resistive random access memory (reram) based on metal oxides. *Proceedings of the IEEE*, 98(12):2237–2251, 2010. doi: 10.1109/JPROC.2010.2070830.

[2] Paul Alcorn. Intel optane dimm pricing. https://www.tomshardware.com/news/intel-optane-dimm-pricing-performance,39007.html, 2023.

[3] David Andersen and Steven Swanson. Rethinking flash in the data center. *IEEE Micro*, 30:52–54, 07 2010. doi: 10.1109/MM.2010.71.

[4] apache. kafka, October 2023. URL https://github.com/apache/kafka. [Online; accessed 31. Oct. 2023].

[5] Kazi Asifuzzaman, Rommel Sánchez Verdejo, and Petar Radojković. Performance and power estimation of stt-mram main memory with reliable system-level simulation. *ACM Trans. Embed. Comput. Syst.*, 21(1), jan 2022. doi: 10.1145/3476838.

[6] Oana Balmau, Diego Didona, Rachid Guerraoui, Willy Zwaenepoel, Huapeng Yuan, Aashray Arora, Karan Gupta, and Pavan Konka. TRIAD: creating synergies between memory, disk and log in log structured key-value stores. In *USENIX ATC '17: Proceedings of the 2017 USENIX Conference on Usenix Annual Technical Conference*, pages 363–375. USENIX Association, USA, July 2017. ISBN 978-1-93197138-6. doi: 10.5555/3154690.3154725.

[7] Oana Balmau, Rachid Guerraoui, Vasileios Trigonakis, and Igor Zablotchi. FloDB: Unlocking Memory in Persistent Key-Value Stores. In *EuroSys '17: Proceedings of the Twelfth European Conference on Computer Systems*, pages 80–94. Association for Computing Machinery, New York, NY, USA, April 2017. ISBN 978-1-45034938-3. doi: 10.1145/3064176.3064193.

[8] Oana Balmau, Florin Dinu, Willy Zwaenepoel, Karan Gupta, Ravishankar Chandhiramoorthi, and Diego Didona. Silk: Preventing latency spikes in log-structured merge key-value stores. In *2019 USENIX Annual Technical Conference*, pages 753–766. USENIX Association, 2019. doi: 10.5555/3358807.3358871.

[9] Vinay Banakar, Kan Wu, Yuvraj Patel, Kimberly Keeton, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Wiscsort: External sorting for byte-addressable storage. *Proceedings of the VLDB Endowment*, 16(9):2103–2116, 2023. doi: 10.14778/3598581.3598585.

[10] Zhichao Cao, Siying Dong, Sagar Vemuri, and David HC Du. Characterizing, modeling, and benchmarking rocksdb key-value workloads at facebook. In *18th USENIX Conference on File and Storage Technologies*, pages 209–223. USENIX Association, 2020. doi: 10.5555/3386691.3386712.

[11] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. Bigtable: A distributed storage system for

structured data. *ACM Transactions on Computer Systems*, 26(2), 2008. doi: 10.1145/1365815.1365816.

[12] Hao Chen, Chaoyi Ruan, Cheng Li, Xiaosong Ma, and Yinlong Xu. Spandb: A fast, cost-effective lsm-tree based kv store on hybrid storage. In *19th USENIX Conference on File and Storage Technologies*, pages 17–32. USENIX Association, 2021.

[13] Youmin Chen, Youyou Lu, Fan Yang, Qing Wang, Yang Wang, and Jiwu Shu. Flatstore: An efficient log-structured key-value storage engine for persistent memory. In *25th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020. doi: 10.1145/3373376.3378515.

[14] Alexander Conway, Abhishek Gupta, Vijay Chidambaram, Martin Farach-Colton, Richard Spillane, Amy Tai, and Rob Johnson. Splinterdb: Closing the bandwidth gap for nvme key-value stores. In *2020 USENIX Annual Technical Conference*. USENIX Association, 2020.

[15] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with YCSB. In *1st ACM Symposium on Cloud Computing*, 2010. doi: 10.1145/1807128.1807152.

[16] Tugrul Daim, Pattravadee Ploykitikoon, Elizabeth Kennedy, and Woraruthai Choothian. Forecasting the future of data storage: Case of hard disk drive and flash memory. *Foresight*, 10:34–49, 08 2008. doi: 10.1108/14636680810918496.

[17] Niv Dayan and Stratos Idreos. Dostoevsky: Better space-time trade-offs for lsm-tree based key-value stores via adaptive removal of superfluous merging. In *SIGMOD '18: Proceedings of the 2018 International Conference on Management of Data*, pages 505–520, 05 2018. doi: 10.1145/3183713.3196927.

[18] Niv Dayan, Manos Athanassoulis, and Stratos Idreos. Monkey: Optimal navigable key-value store. In *SIGMOD '17: Proceedings of the 2017 ACM International Conference on Management of Data*, pages 79–94, 05 2017. doi: 10.1145/3035918.3064054.

[19] Siying Dong, Mark Callaghan, Leonidas Galanis, Dhruba Borthakur, Tony Savor, and Michael Strum. Optimizing space amplification in rocksdb. In *8th Biennial Conference on Innovative Data Systems Research*, 2017.

[20] Ulrich Drepper. What every programmer should know about memory, 2007.

[21] Zhuohui Duan, Jiabo Yao, Haikun Liu, Xiaofei Liao, Hai Jin, and Yu Zhang. Revisiting log-structured merging for kv stores in hybrid memory systems. In *28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2023. doi: 10.1145/3575693.3575715.

[22] facebook. facebook/rocksdb: Rocksdb v6.11. https://github.com/facebook/rocksdb/tree/v6.11.6, 2023.

[23] Alexandra Fedorova, Craig Mustard, Ivan Beschastnikh, Julia Rubin, Augustine Wong, Svetozar Miucin, and Louis Ye. Performance comprehension at wiredtiger. In *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2018. doi: 10.1145/3236024.3236081.

[24] Alexandra Fedorova, Keith A. Smith, Keith Bostic, Susan LoVerso, Michael Cahill, and Alex Gorrod. Writes hurt: Lessons in cache design for optane nvram. In *Proceedings of the 13th Symposium on Cloud Computing*, pages 110–125. ACM, 2022. doi: 10.1145/3542929.3563461.

[25] Brad Fitzpatrick. Distributed caching with memcached. *Linux J*, 2004, 01 2004.

[26] Fujitsu. Reram overview white paper. https://www.fujitsu.com/jp/group/fsm/en/products/reram/ReRAM_whitepaper_2023e.pdf, 2023.

[27] Manuel Le Gallo and Abu Sebastian. An overview of phase-change memory device physics. *Journal of Physics D: Applied Physics*, 53(21):213002, mar 2020. doi: 10.1088/1361-6463/ab7794. URL https://dx.doi.org/10.1088/1361-6463/ab7794.

[28] google. google/leveldb: Leveldb v1.23. https://github.com/google/leveldb, 2023.

[29] Shashank Gugnani, Arjun Kashyap, and Xiaoyi Lu. Understanding the idiosyncrasies of real persistent memory. *Proceedings of the VLDB Endowment*, 14(4):626–639, 2020. doi: 10.14778/3436905.3436921.

[30] Frank T. Hady, Annie P. Foong, Bryan Veal, and Dan Williams. Platform storage performance with 3d xpoint technology. *Proceedinds of the IEEE*, 105(9):1822–1833, 2017. doi: 10.1109/JPROC.2017.2731776.

[31] J.L. Harrington. *Relational Database Design and Implementation: Fourth Edition*. Morgan Kaufmann, 05 2016.

[32] Intel. Persistent memory development kit. https://pmem.io/pmdk/, 2023.

[33] Olzhas Kaiyrakhmet, Songyi Lee, Beomseok Nam, Sam H. Noh, and Young ri Choi. SLM-DB: Single-level key-value store with persistent memory. In *17th USENIX Conference on File and Storage Technologies*, pages 191–205. USENIX Association, 2019. doi: 10.5555/3323298.3323317.

[34] Sudarsun Kannan, Nitish Bhat, Ada Gavrilovska, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. Redesigning lsms for nonvolatile memory with novelsm. In *2018 USENIX Annual Technical Conference*, pages 993–1005. USENIX Association, 2018. doi: 10.5555/3277355.3277450.

[35] Riduan Khaddam-Aljameh, Milos Stanisavljevic, Jordi Fornt Mas, Geethan Karunaratne, Matthias Brändli, Feng Liu, Abhairaj Singh, Silvia M. Müller, Urs Egger, Anastasios Petropoulos, Theodore Antonakopoulos, Kevin Brew, Samuel Choi, Injo Ok, Fee Li Lie, Nicole Saulnier, Victor Chan, Ishtiaq Ahsan, Vijay Narayanan, S. R. Nandakumar, Manuel Le Gallo, Pier Andrea Francese, Abu Sebastian, and Evangelos Eleftheriou. HERMES-Core - A 1.59-TOPS/mm$^2$ PCM on 14-nm CMOS In-Memory Compute Core Using 300-ps/LSB Linearized CCO-Based ADCs. *IEEE J. Solid State Circuits*, 57(4):1027–1038, 2022. doi: 10.1109/JSSC.2022.3140414.

[36] Minjae Kim, Bryan S. Kim, Eunji Lee, and Sungjin Lee. A case study of a dram-nvm hybrid memory allocator for key-value stores. *IEEE Computer Architecture Letters*, 21(2):81–84, 2022. doi: 10.1109/LCA.2022.3197654.

[37] Wonbae Kim, Chanyeol Park, Dongui Kim, Hyeongjun Park, Young ri Choi, Alan Sussman, and Beomseok Nam. Listdb: Union of write-ahead logs and persistent skiplists for incremental checkpointing on persistent memory. In *16th USENIX Symposium on Operating Systems Design and Implementation*, 2022.

[38] Youngjin Kwon, Henrique Fingler, Tyler Hunt, Simon Peter, Emmett Witchel, and Thomas Anderson. Strata: A cross media file system. In *26th Symposium on Operating Systems Principles*, pages 460–477. ACM, 2017. doi: 10.1145/3132747.3132770.

[39] Avinash Lakshman and Prashant Malik. Cassandra: a decentralized structured storage system. *ACM SIGOPS operating systems review*, 44(2), 2010. doi: 10.1145/1773912.1773922.

[40] S W Lee, Bongki Moon, Chanik Park, Jae-Myung Kim, and Sang-Woo Kim. A case for flash memory ssd in enterprise database applications. In *SIGMOD '08: Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 1075–1086, 06 2008. doi: 10.1145/1376616.1376723.

[41] Baptiste Lepers and Willy Zwaenepoel. Johnny cache: the end of DRAM cache conflicts (in tiered main memory systems). In *17th USENIX Symposium on Operating Systems Design and Implementation*, pages 519–534. USENIX Association, 2023.

[42] Zhen Lin, Lianjie Cao, Faraz Ahmed, Hui Lu, and Puneet Sharma. When caching systems meet emerging storage devices: A case study. In *15th ACM Workshop on Hot Topics in Storage and File Systems*, pages 37–43. ACM, 2023. doi: 10.1145/3599691.3603413.

[43] Zhen Lin, Lingfeng Xiang, Jia Rao, and Hui Lu. P2cache: Exploring tiered memory for in-kernel file systems caching. In *2023 USENIX Annual Technical Conference*, pages 801–815. USENIX Association, 2023.

[44] Chen Luo and Michael J Carey. Lsm-based storage techniques: a survey. *The VLDB Journal*, 29(1), 2020. doi: 10.1007/s00778-019-00555-y.

[45] Jonathan Mace and Rodrigo Fonseca. Universal context propagation for distributed system instrumentation. In *13th European Conference on Computer Systems*. ACM, 2018. doi: 10.1145/3190508.3190526.

[46] Ricardo Macedo, Yusuke Tanimura, Jason Haga, Vijay Chidambaram, José Pereira, and João Paulo. Paio: General, portable i/o optimizations with minor application modifications. In *20th USENIX Conference on File and Storage Technologies*, pages 413–428. USENIX Association, 2022.

[47] Sara McAllister, Benjamin Berg, Julian Tutuncu-Macias, Juncheng Yang, Sathya Gunasekar, Jimmy Lu, Daniel S. Berger, Nathan Beckmann, and Gregory R. Ganger. Kangaroo: Caching billions of tiny objects on flash. In *28th Symposium on Operating Systems Principles*. ACM, 2021. doi: 10.1145/3477132.3483568.

[48] s. r. o. Netservis. Intel dc p4600 2tb nvme ssd pricing. https://smicro.eu/intel-dc-p4600-2tb-nvme-pcie3-0x4-3d-tlc-hhhl-aic-1dwpd-fw-qdv1013d-ssdpedke020t7-1, 2023.

[49] s. r. o. Netservis. Intel s4610 sata ssd pricing. https://smicro.eu/intel-s4610-1-92tb-sata-6gb-s-3d-tlc-2-5-3dwpd-fw-ssdsc2kg019t801-1, 2023.

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

[51] OpenCAS. Open cache acceleration software. https://open-cas.github.io/, apr 2023.

[52] Dharmit Patel, Iman Sadooghi, and Ioan Raicu. Towards distributed message queues using distributed key/value stores. *ResearchGate*, January 2014.

[53] Ashwini Raina, Jianan Lu, Asaf Cidon, and Michael J. Freedman. Efficient compactions between storage tiers with prismdb. In *28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 179–193. ACM, 2023. doi: 10.1145/3582016.3582052.

[54] Amanda Raybuck, Tim Stamler, Wei Zhang, Mattan Erez, and Simon Peter. Hemem: Scalable tiered memory management for big data applications and real nvm. In *28th Symposium on Operating Systems Principles*, pages 392–407. ACM, 2021. doi: 10.1145/3477132.3483550.

[55] redis. redis, October 2023. URL https://github.com/redis/redis. [Online; accessed 1. Oct. 2023].

[56] Gantz Reinsel and Rydning. Data age 2025: The digitization of the world from edge to core., 2018.

[57] Mendel Rosenblum and John K Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems*, 10(1), 1992. doi: 10.1145/146941.146943.

[58] Hyunsub Song, Shean Kim, J. Hyun Kim, Ethan JH Park, and Sam H. Noh. First responder: Persistent memory simultaneously as high performance buffer cache and storage. In *2021 USENIX Annual Technical Conference*, pages 839–853. USENIX Association, 2021.

[59] Yongju Song, Wook-Hee Kim, Sumit Kumar Monga, Changwoo Min, and Young Ik Eom. Prism: Optimizing key-value store for modern heterogeneous storage devices. In *28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 588–602. ACM, 2023. doi: 10.1145/3575693.3575722.

[60] speedb io. speedb-io/speedb: Speedb v2.6.0. https://github.com/speedb-io/speedb/tree/speedb/v2.6.0, 2023.

[61] Jing Wang, Youyou Lu, Qing Wang, Minhui Xie, Keji Huang, and Jiwu Shu. Pacman: An efficient compaction approach for log-structured key-value store on persistent memory. In *2022 USENIX Annual Technical Conference*, pages 773–788. USENIX Association, 2022.

[62] Teng Wang, Adam Moody, Yue Zhu, Kathryn Mohror, Kento Sato, Tanzima Islam, and Weikuan Yu. Metakv: A key-value store for metadata management of distributed burst buffers. In *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 1174–1183, 2017. doi: 10.1109/IPDPS.2017.39.

[63] Hobin Woo, Daegyu Han, Seungjoon Ha, Sam H. Noh, and Beomseok Nam. On stacking a persistent memory file system on legacy file systems. In *21st USENIX Conference on File and Storage Technologies*, pages 281–296. USENIX Association, 2023. doi: 10.5555/3585938.3585956.

[64] Kan Wu, Zhihan Guo, Guanzhou Hu, Kaiwei Tu, Ramnatthan Alagappan, Rathijit Sen, Kwanghyun Park, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. The storage hierarchy is not a hierarchy: Optimizing caching on modern storage devices with orthus. In *19th USENIX Conference on File and Storage Technologies*, pages 307–323. USENIX Association, 2021.

[65] Kan Wu, Kaiwei Tu, Yuvraj Patel, Rathijit Sen, Kwanghyun Park, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. Nyxcache: Flexible and efficient multi-tenant persistent memory caching. In *20th USENIX Conference on File and Storage Technologies*, pages 1–16. USENIX Association, 2022.

[66] Fei Xia, Dejun Jiang, Jin Xiong, and Ninghui Sun. Hikv: A hybrid index key-value store for dram-nvm memory systems. In *2017 USENIX Annual Technical Conference*, 2017. doi: 10.5555/3154690.3154724.

[67] Lingfeng Xiang, Xingsheng Zhao, Jia Rao, Song Jiang, and Hong Jiang. Characterizing the performance of intel optane persistent memory: A close look at its on-dimm buffering. In *17th European Conference on Computer Systems*, pages 488–505. ACM, 2022. doi: 10.1145/3492321.3519556.

[68] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steve Swanson. An empirical guide to the behavior and use of scalable persistent memory. In *18th USENIX Conference on File and Storage Technologies*, pages 169–182. USENIX Association, 2020. doi: 10.5555/3386691.3386708.

[69] Suli Yang, Tyler Harter, Nishant Agrawal, Salini Selvaraj Kowsalya, Anand Krishnamurthy, Samer Al-Kiswany, Rini T. Kaushik, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Split-level i/o scheduling. In *25th Symposium on Operating Systems Principles*, pages 474–489. ACM, 2015. doi: 10.1145/2815400.2815421.

[70] Ting Yao, Yiwen Zhang, Jiguang Wan, Qiu Cui, Liu Tang, Hong Jiang, Changsheng Xie, and Xubin He. Matrixkv: Reducing write stalls and write amplification in lsm-tree based kv stores with matrix container in nvm. In *2020 USENIX Annual Technical Conference*, pages 17–31. USENIX Association, 2020. doi: 10.5555/3489146.3489148.

[71] Hobin Yoon, Juncheng Yang, Sveinn Fannar Kristjansson, Steinn E. Sigurdarson, Ymir Vigfusson, and Ada Gavrilovska. Mutant: Balancing storage cost and latency in lsm-tree data stores. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 162–173. ACM, 2018. doi: 10.1145/3267809.3267846.

[72] Jinghuan Yu, Sam H. Noh, Young ri Choi, and Chun Jason Xue. Adoc: Automatically harmonizing dataflow between components in log-structured key-value stores for improved performance. In *21st USENIX Conference on File and Storage Technologies*, pages 65–80. USENIX Association, 2023. doi: 10.5555/3585938.3585943.

[73] Baoquan Zhang and David H. C. Du. Nvlsm: A persistent memory key-value store using log-structured merge tree with accumulative compaction. *ACM Transactions on Storage*, 17(3), 2021. doi: 10.1145/3453300.

[74] Wenhui Zhang, Xingsheng Zhao, Song Jiang, and Hong Jiang. Chameleondb: A key-value store for optane persistent memory. In *16th European Conference on Computer Systems*, 2021. doi: 10.1145/3447786.3456237.

[75] Shengan Zheng, Morteza Hoseinzadeh, and Steven Swanson. Ziggurat: A tiered file system for non-volatile main memories and disks. In *17th USENIX Conference on File and Storage Technologies*, pages 207–219. USENIX Association, 2019. doi: 10.5555/3323298.3323318.

[76] Shengan Zheng, Morteza Hoseinzadeh, Steven Swanson, and Linpeng Huang. Tpfs: A high-performance tiered file system for persistent memories and disks. *ACM Transactions on Storage*, 19 (2), mar 2023. doi: 10.1145/3580280.