Technische Universität München

TUM School of Computation, Information and Technology

Adaptive Storage Structures

Lukas Vogel

Vollständiger Abdruck der von der TUM School of Computation, Information and Technology der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitz:
Prof. Dr. Viktor Leis

Prüfer*innen der Dissertation:
1. Prof. Alfons Kemper, Ph. D.
2. Prof. Dr. Thomas Neumann
3. Prof. Pınar Tözün, Ph. D.

Die Dissertation wurde am 22.06.2023 bei der Technischen Universität München eingereicht und durch die TUM School of Computation, Information and Technology am 23.09.2023 angenommen.

Abstract

In the last ten years, storage technology has developed at an unprecedented pace. Innovations like inexpensive and fast SSDs or persistent memory have enabled off-the-shelve servers to read and persistently write multiple gigabytes of data per second at nanosecond latency.

This thesis identifies several aspects in which the storage layer of database systems can be better adapted to modern storage technologies and makes the case that large parts of the storage stack must be rethought to optimize the usage of novel hardware. To this end, this thesis proposes two new systems, Mosaic and Plush, which rethink different aspects of a database system's storage layer to use these advancements.

In the modern storage landscape, multiple storage device types exist, each competitive at its price point. Existing database systems have difficulties making use of most of them. Mosaic is a storage engine and storage device purchase recommender for relational database systems. It enables system administrators to utilize all modern storage devices with varying throughput and costs for analytical workloads. With Mosaic, we show that when optimizing for the zoo of modern storage technologies, higher query throughputs than state-of-the-art approaches are possible at the same budget or similar query throughputs at a lower cost.

Persistent memory has DRAM-like performance, is byte-addressable, and excels at small random writes while having the persistency guarantees of conventional block storage. These properties enable developers to design much simpler and faster database systems in theory but are hard to utilize in practice. Plush, a key-value store for persistent memory, demonstrates how applications can be re-designed for groundbreaking new storage hardware to simplify the storage stack.

Advances in storage technology have exposed bottlenecks when moving data within the software stack. Hardware workarounds exist but only provide brittle abstractions and are hard to use. As a third contribution, we thus present Data Pipes, a vision showing how to solve the bottleneck of data movement between and within modern storage hardware and memory.

Zusammenfassung

In den letzten zehn Jahren hat sich die Speichertechnologie in einem noch nie dagewesenen Tempo entwickelt. Innovationen wie billige und schnelle SSDs oder Persistent Memory ermöglichen es Servern, mehrere Gigabytes an Daten pro Sekunde mit einer Latenz von Nanosekunden zu lesen und dauerhaft zu speichern.

In dieser Dissertation werden mehrere Aspekte aufgezeigt, wie die Speicherschicht von Datenbanksystemen besser an moderne Speichertechnologien angepasst werden kann, und es wird belegt, dass große Teile des Speicherstapels neu gedacht werden müssen, um neue Hardware optimal nutzen zu können. Zu diesem Zweck werden in dieser Arbeit zwei neue Systeme, Mosaic und Plush, vorgestellt, die verschiedene Aspekte der Speicherebene eines Datenbanksystems neu überdenken, um diese Fortschritte zu nutzen.

In der modernen Speicherlandschaft gibt es mehrere Arten von Speichergeräten, von denen jedes in seinem Preissegment eine Daseinsberechtigung hat. Bestehende Datenbanksysteme haben Schwierigkeiten, diese zu nutzen. Mosaic ist eine Storage-Engine für relationale Datenbanksysteme, die zudem Kaufempfehlungen für neue Speicherhardware ausspricht. Sie ermöglicht Systemadministratoren, alle modernen Speichergeräte mit unterschiedlichem Durchsatz und Kosten für analytische Workloads zu nutzen. Mit Mosaic zeigen wir, dass bei der Optimierung für den Zoo moderner Speichertechnologien ein höherer Anfragedurchsatz bei gleichem Budget möglich ist als bei vorhergehenden Ansätzen oder ein gleicher Anfragedurchsatz bei geringeren Kosten.

Persistent Memory hat eine mit DRAM vergleichbare Leistung, ist byte-weise adressierbar und ist besonders geeignet für kleine, nicht-sequentielle Schreibvorgänge, während er die Persistenzgarantien eines herkömmlichen Blockspeichers bietet. Diese Eigenschaften ermöglichen es Entwicklern in der Theorie, konzeptuell viel einfachere und schnellere Datenbanksysteme zu entwerfen. In der Praxis sind die Vorteile von Persistent Memory jedoch schwer zu nutzen. Plush, ein Key-Value-Store für Persistent Memory, zeigt, wie Anwendungen für neue Speicherhardware umgestaltet werden können, um den Storage-Stack zu vereinfachen.

Fortschritte in der Speichertechnologie haben Flaschenhälse für den Datentransport innerhalb des Software-Stacks aufgedeckt. Es gibt zwar Hardware-Workarounds, die jedoch nur unzureichende Abstraktionen bieten und schwer zu verwenden sind. Als dritten Beitrag stellen wir daher Data Pipes vor, eine Vision, die zeigt, wie man den Flaschenhals der Datenbewegung zwischen und innerhalb moderner Speicherhardware und dem Arbeitsspeicher lösen kann.

Acknowledgments

To my wife, who not only encouraged me to take this path in life, but also supported me all the way.

Preface

Excerpts of this thesis have been published in advance.

Excerpts of Chapter 2 have previously been published in:

Lukas Vogel et al. “Mosaic: A Budget-Conscious Storage Engine for Relational Database Systems”. In: *PVLDB* 13.11 (2020), pp. 2662–2675

Excerpts of Chapter 3 have previously been published in:

Lukas Vogel et al. “Plush: A Write-Optimized Persistent Log-Structured Hash-Table”. In: *PVLDB* 15.11 (2022), pp. 2895–2907

Excerpts of Chapter 4 have previously been published in:

Lukas Vogel et al. “Data Pipes: Declarative Control over Data Movement”. In: *CIDR*. www.cidrdb.org, 2023

During his doctoral studies, the author also contributed to the following related work.

Alexander van Renen et al. “Persistent Memory I/O Primitives”. In: *DaMoN*. ACM, 2019, 12:1–12:7

Alexander van Renen et al. “Building blocks for persistent memory”. In: *VLDB J.* 29.6 (2020), pp. 1223–1241

Reika Kinoshita et al. “Cost-Performance Evaluation of Heterogeneous Tierless Storage Management in a Public Cloud”. In: *CANDAR*. IEEE, 2021, pp. 121–126

Christoph Anneser et al. “Programming Fully Disaggregated Systems”. In: *HotOS*. USENIX Association, 2023

Contents

Preface **ii**

Listings **ix**

1 Introduction **1**

1.1 Background 2
1.2 Modern Storage Devices 5
1.3 Challenges 5
1.4 Opportunities and Thesis Contributions 8
1.5 Prior Publications and Authorship 10

2 Mosaic: A Budget-Conscious Storage Engine **11**

2.1 Introduction 11
2.2 Background and Related Work 14
2.2.1 Heterogeneous Storage for Big Data Query Engines . . . 14
2.2.2 Heterogeneous Storage in RDBMS 15
2.2.3 Prediction and Storage Recommendation 16
2.3 Mosaic System Design 16
2.3.1 Metadata 17
2.3.2 Storage Format 19
2.3.3 Data Retrieval 20
2.3.4 Data Placement 21
2.4 Data Placement Strategies 21
2.4.1 A Model for Predicting Table Scan Time 22
2.4.2 Responsibilities of a Strategy 25
2.4.3 HOT Strategy at Table Granularity 26
2.4.4 HOT Strategy at Column Granularity 26
2.4.5 Linear Optimization Strategy 27
2.5 Evaluation 29
2.5.1 Benchmarks 30
2.5.2 Mosaic vs. Traditional RDBMS 31

2.5.3 Comparison of Placement Strategies 33
2.5.4 Per-Query Analysis of LOPT 34
2.5.5 Placement Calculation Cost of LOPT 34
2.5.6 Capacity Mode vs. Budget Mode 37
2.5.7 Prediction Accuracy 38
2.5.8 Impact of Workload 39
2.5.9 Mosaic vs. Big Data Query Engines 41
2.6 Mosaic in the Cloud 42
2.7 Summary 43

3 Plush: A Persistent Log-Structured Hash-Table 45
3.1 Introduction 45
3.2 Background 47
3.2.1 Persistent Memory 47
3.2.2 LSM Trees 49
3.2.3 Hashing on PMem 49
3.3 Overarching Design 50
3.4 Architecture 51
3.4.1 Multi-leveled Hash Table 51
3.4.2 Recovery Log 55
3.4.3 Payload Log and Variable-Length Records 57
3.5 Operations 57
3.5.1 Upsert 58
3.5.2 Lookup 61
3.5.3 Delete 61
3.5.4 Recovery 61
3.5.5 Concurrency 63
3.5.6 Crash Consistency 63
3.5.7 Bulk Loading 64
3.6 Evaluation 64
3.6.1 Plush Configuration 65
3.6.2 Comparison to Other Data Structures 67
3.6.3 Fixed-Size Records 68
3.6.4 Space Utilization 72
3.6.5 Plush Tuning 72
3.6.6 Range Queries 74
3.6.7 Recovery 75
3.6.8 Variable-Length Records 76
3.7 Related Work 76
3.8 Summary 78

4 Data Pipes: Declarative Data Movement **79**
4.1 Introduction 79
4.2 Background and Motivation 81
4.3 Case Study: External Sort 83
4.3.1 Data Movements in External Sort 83
4.3.2 The Case for Hardware-Assisted Data Movement 85
4.4 Experiments 86
4.4.1 Fast Load from Storage to Compute 86
4.4.2 Fast Load from Buffer to Memory 87
4.4.3 Lack of Control for Data Spills to Buffer 89
4.4.4 Discussion 90
4.5 Our Vision: Data Pipes 90
4.5.1 Resource Locators 91
4.5.2 Data Pipes 91
4.5.3 Data Pipes Optimization 96
4.6 Data Pipes Principles 97
4.7 Related Work and Research Agenda 99
4.7.1 Operating Systems 99
4.7.2 Hardware 100
4.7.3 Cloud Infrastructure 101
4.8 Summary 102

5 Conclusions and Future Work **103**

Bibliography **105**

List of Figures

1.1 Price of different storage media in relation to capacity. 6

2.1 Estimated performance spectrum of Mosaic compared to big data query engines like Spark and manual data placement. 13
2.2 The components of Mosaic and its interface with the RDBMS. . . . 17
2.3 Metadata recorded, maintained, and stored by Mosaic. 18
2.4 Modus operandi of Mosaic, and two exemplary placement strategies. 23
2.5 LOPT data placement in *budget* mode for TPC-H and TPC-DS (SF 100) at different budgets. 28
2.6 Benchmark runtime for TPC-H (SF 30) with column-granular placement using the LOPT strategy compared to all placement permutations of the four largest tables at table granularity. 32
2.7 Comparison of placement algorithms normalized to HOT table . . . 33
2.8 Runtime per query for two different LOPT variants (TPC-H SF 30). . 35
2.9 Computation time in seconds for a solution within 5% of the lower bound. 37
2.10 Impact of sampling on calculation time and runtime. 37
2.11 Comparison of placement modes for the TPC-H benchmark (SF 30) using LOPT 38
2.12 Predicted vs. actual performance for the TPC-H and TPC-DS benchmarks (SF 30 and 100). 39
2.13 Performance of 4 out of 1000 TPC-DS workloads at different budgets for data placed specifically for the workload and for data placed for the TPC-DS benchmark in general. 40
2.14 Mosaic's TPC-H throughput (SF 30) compared to Umbra and two Big Data query engines. 41

3.1 Plush's component overview with illustrated insert and lookup algorithms. 52
3.2 Migrating the buckets of a directory entry. 53
3.3 Layout of a directory entry, its buckets (for fixed and variable-length keys), and the payload log for fanout 8. 54

3.4 Layout of a recovery log. 56
3.5 Space consumption relative to data set size as stacked area chart. . . 66
3.6 Throughput of core operations under varying thread count for fixed-size records (8-byte keys, 8-byte values). 69
3.7 Throughput of mixed workloads with 48 threads for fixed-size records (8-byte keys, 8-byte values). 70
3.8 Read and write amplification for 16-byte records on DRAM and PMem. 71
3.9 Tail latencies for 16-byte records with 23 threads. 72
3.10 Storage consumption compared to data set size. 73
3.11 Throughput gained by optimizations. 73
3.12 Throughput for range queries. 74
3.13 Recovery time vs. data set size for filters and logs. 75
3.14 Throughput for variable-length entries. 77

4.1 Data movement primitives can shuffle data among storage layers. We claim that the set of primitives is incomplete and incoherent. . . 82
4.2 Hardware-efficient external sort using DDIO and I/OAT across CPU caches, DRAM, PMem, and SSDs. 84
4.3 Performance with enabled and disabled DDIO (one CPU core) with parallel DRAM-intensive STREAM workload. 87
4.4 Impact of the `IIO_LLC_WAYS` register on cache misses with DDIO enabled. 88
4.5 Throughput when moving data between DRAM and PMem with and without I/OAT. 88
4.6 Using data pipes to make algorithms modular. 96

List of Tables

1.1 Bandwidth, latency, and I/O operations per second for different storage and memory devices. 3

2.1 Storage devices of the evaluation system. 29
2.2 TPC-H benchmark speedup (SF 30) of SSD and HDD for different compression algorithms. 30
2.3 LOPT search time for a placement solution for four devices with three different workloads. It shows the time to find a solution that is within 5% or 1% of the theoretical optimum, or is optimal. 36
2.4 EBS types and their performance characteristics compared to a consumer-grade SSD. 42

3.1 Write access characteristics for different media 45
3.2 Investigated Data Structures. 67

Listings

3.1 Insertion algorithm for a fixed-size KV pair. 58
3.2 Migration algorithm for a directory entry. 60
3.3 Lookup algorithm for a fixed-size KV pair. 62
4.1 Straightforward flavor, loading data from SSD and sorting it into runs. 92
4.2 Inversion of Control flavor, store sorted runs on PMem. 94
4.3 OS supported flavor, load and merge sorted runs. 95

CHAPTER 1

Introduction

Excerpts of this chapter have been published previously [148, 149, 150].

We are currently in the Zettabyte Era: According to the International Data Corporation, humanity will have created 175 Zettabytes of data by 2025, a large part of which has to be stored persistently [135]. To this end, an ever-growing amount of storage hardware must be installed yearly. While in 2010, only 0.5 ZB of storage capacity was shipped, this increased rapidly within the last years:

> *"over 22 ZB of storage capacity must ship across all media types from 2018 to 2025 to keep up with storage demands."*
>
> — International Data Corporation [135]

Most data stored on such devices is structureless (e. g., videos, photos, or audio) and stored in binary blobs. However, data critical to a company or private user (e. g., sales, salary information, inventory, ...) usually has a structured format and is stored within a database system. Database systems fulfill two essential purposes: (1) Store new incoming transactions persistently and (2) provide an interface to query stored data. While database systems use volatile memory to store intermediate results and buffer recently accessed data, they must ensure that all primary data is flushed consistently and recoverable to durable storage upon user commit. Otherwise, the system might lose primary data the user had assumed to be durably stored already upon a crash.

Traditionally, durable storage was assumed to be spinning hard disks (HDDs) or – more recently – comparatively slow SATA solid state drives (SSDs), which are orders of magnitude slower than volatile dynamic random access memory (DRAM). Thus, storing data (i. e., in a *transactional* setting where the user inserts or updates records) or retrieving data (i. e., in an *analytical* setting where the user queries the system) was traditionally dominated by the high access latency and low throughput of a database system's storage devices. Great care was taken to ensure that data was stored in such a way that the access patterns were optimal for HDDs (i. e., sequential, large reads and writes), for example, via B-trees [11] or log-structured merge-trees (LSM trees) [115].

In recent years, new storage technologies such as Intel's Optane DC Persistent Memory Modules (PMem) [131] or fast consumer-grade NVMe SSDs have invalidated many long-held assumptions of database systems concerning the storage stack thus requiring us to re-design them to fully leverage the increased performance of such new technologies [52]. This thesis tackles multiple challenges posed by emerging storage hardware and proposes ways to improve the design of database systems so that they can fully profit from the new hardware.

In this chapter, we first introduce the theoretical background of storing data in a database system context (Section 1.1), demonstrate the improvements modern storage devices make over the classic storage stack (Section 1.2), then state the unsolved challenges this new stack poses (Section 1.3), before finally discussing this thesis' contribution in solving those challenges in Section 1.4

1.1 Background

When discussing storing data, one must distinguish between memory and storage. Memory is usually fast but volatile (i. e., it forgets stored data when the system is powered down). At the same time, storage is slower but persistent (i. e., data stored in storage is durable and survives power failures). Newer developments like Persistent Memory (PMem) blur the lines between both.

Volatile memory. CPUs require the data they operate on to be stored in main memory, where they can address it directly. Memory has traditionally been dynamic random access memory (DRAM), which is expensive, volatile, and limited in size, as a CPU only supports a limited number of memory channels. Since data that has been recently accessed is more likely to be processed again soon, CPUs store such data in a hierarchy of caches with lower latency and higher bandwidth than DRAM. However, while caches higher in the hierarchy have better bandwidth and latency, their size decreases accordingly. Additionally, even the largest caches are usually orders of magnitude smaller than the system's DRAM capacity.

Persistent storage. To make sure that data is stored persistently, the system has to move it from volatile DRAM to persistent background storage, which usually has orders of magnitude lower bandwidth and higher latency (cf. Table 1.1). Storage devices expose a logical block interface, where a block is the smallest data unit that can be read or written with a standard block size of 4 KiB. Unfortunately, this block interface has always been a *leaky abstraction*: For

Table 1.1: Bandwidth, latency, and I/O operations per second for different storage and memory devices.

	Tput [MiB/s]		Latency [μs]		IOPS	
Storage Device	read	write	read	write	read	write
DRAM	95000	87000	0.1	0.1	–	–
PMem	39122	6452	0.4	0.1	–	–
NVMe SSD (consumer)	3452	2604	1121	3760	551636	606654
NVMe SSD (enterprise)	2983	1285	3506	2796	376602	32072
SATA SSD	528	489	5800	20075	79096	50133
HDD	246	251	44842	121177	765	800
HDD (RAID 5)	472	181	39921	54874	2018	746

HDDs, which have spinning platters, the high latency almost exclusively stems from physically moving the read head over the magnetic platters. To achieve the maximum throughput of $\approx$ 250 MiB/s, reads thus have to be sequential so that the head is only moved once. When doing random reads, an HDD can only sustain about 765 IOPS/s $\cdot$ 4 KiB/IOPS = $3.1 \text{MiB}/s$ (cf. Table 1.1). Thus, one must ensure that reads and writes to HDDs are *sequential* and multiple megabytes long to achieve peak throughput. Similar issues arise for SSDs: While they are better suited for handling random workloads, they expose the same block interface, even though this does not map cleanly onto their internal structure. Internally, an SSD uses flash memory organized into *erase blocks*, usually a few MiB in size [79]. To write even a single bit to such a block, the SSD must first erase the whole block before it can write to that block, thus limiting performance for small random writes. Furthermore, since each erase block wears out after a limited number of writes, SSDs introduce a *flash translation layer* between logical blocks and erase blocks as well as a DRAM cache [94] to support wear leveling and garbage collection [14] making it challenging to predict latency and throughput without knowing the internal device state [17].

Data movement interfaces. Moving data from and to persistent storage can happen implicitly (triggered by the operating system) or explicitly (triggered by the application). POSIX-compliant systems such as Unix, for example, offer the `mmap` system call as *implicit* interface [108]. Calling `mmap` maps a file stored on a storage device into memory. When the application accesses a file-backed memory address afterward, a page fault is triggered, and the OS loads the corresponding data from the storage device into memory. However, `mmap`'s implicit interface makes it hard to correctly implement database systems without violating consistency guarantees [30]. Alternatively, applications can use the `pread`/`pwrite` system calls to *explicitly* move data between a storage device and a specified memory region on DRAM [124]. While this approach gives the application more control, it also requires the developer to implement their own buffer manager, as the operating system will not be able to implicitly move data back to its storage device when it runs out of memory.

Even the explicit system calls, however, have implicit side effects. Operating systems repurpose some DRAM as *page cache* where they temporarily store recently read or written data. Caching data in such a way is especially beneficial when applications have predictable data access patterns (e. g., accessing data sequentially). The operating system can then pre-fetch data it assumes will be required soon (on the read path) or group data into sequential chunks before writing it back to the storage device (on the write path), which better fits the properties of HDDs as explained above. While this approach is advantageous in the historically common case (i. e., using HDDs and having predictable access patterns), it is a *leaky abstraction* and has two issues:

1. If the access patterns are *mis*-predicted by the OS, it incorrectly prefetches data that will not be used, leading to spurious traffic to and from the storage device and thus to reduced throughput and increased latency.

2. The operating system is not required to instantly store newly written or updated data on the storage device. It is allowed to instead just mark it as *dirty* in the page cache. The operating system then transparently and nondeterministically (to the user) writes dirty data back to the corresponding storage device. To ensure that data is stored persistently at a specific point, the application must manually flush dirty pages to disk, e. g., by using the expensive `fsync` system call on Linux [43].

Both, the explicit and the implicit approach, additionally have the downside of being *blocking*: When accessing memory (`mmap`) or issuing a system call (`pread`/`pwrite`), the operation blocks until the underlying storage device has transferred the data successfully, adding potentially multiple milliseconds of delay in which the application could have done something else. This delay was acceptable in the past when storage devices had such a high latency that they were the bottleneck anyway. However, blocking interfaces and OS overhead have become significant issues with modern storage devices. Newer approaches like `io_uring` [68] solve this issue by running in user space and thus eliminating expensive context switching and by providing an asynchronous interface allowing one to do other computations while waiting for I/O requests to complete.

1.2 Modern Storage Devices

Having introduced the background of storage devices, we present two innovations that have transformed the storage landscape in the last five years.

Cheap and Fast NVMe SSDs. SSDs have long been bottlenecked by the bandwidth of their SATA interface, allowing a maximum throughput of 600 MB per second [41]. Newer SSDs alleviate this issue by using the NVMe protocol [48], which allows for higher parallelism by having multiple queues and higher throughput by connecting to the CPU via PCI Express (PCIe), which offers much higher bandwidth than traditional SATA drives. As shown by Table 1.1, a modern NVMe SSD thus improves by order of magnitude over traditional SATA SSDs by latency and throughput. Applications, however, often have difficulties fully profiting from this speedup as old interfaces (e. g., synchronous blocking accesses and OS overhead through system calls) often become the bottleneck.

Intel Optane Persistent Memory. In 2019, Intel released their Optane DC Persistent Memory Modules (or short: PMem), which blurs the line between memory and storage. It comes in a DIMM form factor pin compatible with standard DDR4 DRAM modules. Unlike DRAM, however, data written to them is not wiped on power loss but is stored persistently. PMem exposes the same `load/store` interface as DRAM but is considerably denser as it offers up to 512 GiB storage per stick. Even though PMem is slower than DRAM by a factor of 3 to 10, it still achieves previously unattained throughput (cf. Table 1.1) and – more importantly – low write latency. While the high throughput can be achieved by accessing multiple slower devices in parallel, latency can only be lowered by improving the storage technology, making PMem peerless if low write latency is required. Its low write latency, together with its byte addressability, thus opens a case for persistent and crash-consistent write-optimized data structures.

1.3 Challenges

The advent of new storage technologies introduces substantial challenges and opportunities to database systems. In this thesis, we tackle three challenges:

(1) The modern storage stack invalidates one-size-fit-all provisioning of database systems. Traditionally, database systems are purpose-built for a specific storage technology (e. g., HDDs, SSDs, or DRAM), leveraging their unique benefits while mitigating their weak points. Such a bespoke architecture worked well when the storage landscape was static. Nowadays, the storage

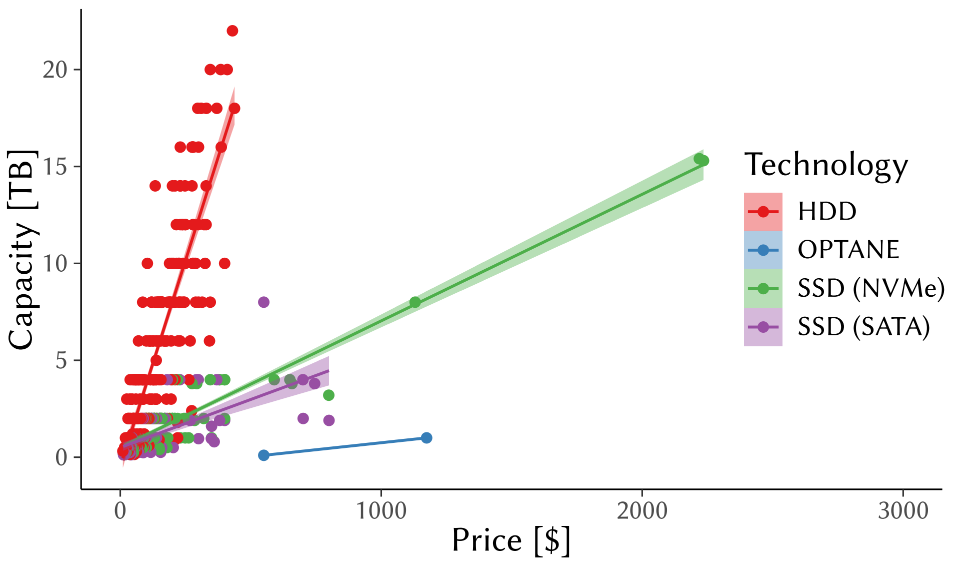


Figure 1.1: Price of different storage media in relation to capacity.[a)]
[a)] Source: Crawled from Amazon via `https://diskprices.com`, accessed: 10.03.2023

landscape is quite heterogeneous and offers different storage technologies at different price points (cf. Table 1.1), each of them being competitive at their different "sweet spots" (cf. Figure 1.1):

- Intel Optane PMem has high throughput and low latency but is very expensive and has limited capacity. It is thus desirable for very hot and to-be-inserted data.
- SSDs have higher throughput and lower latency than HDDs but have a much higher cost per gigabyte of storage. They thus excel at storing warm data, which is usually just a small part of the data set.
- Traditionally, a big part of a database's data set is seldom or never accessed (i. e., cold data). Here, only the persistency aspect of the storage device is important: Throughput and latency do not matter. HDDs can thus store such data at a very low cost without any drawbacks.
- New, unknown storage technologies could be introduced with different drawbacks, upending the existing hierarchy yet again.

A traditional database system does not cope well with this multitude of storage devices that are competitive at their "sweet spots". To make use of different storage technologies, users currently either

1. have to accept that their database system is optimized for a certain storage technology (DRAM for main memory DBMS like Hyper [84], HDDs for traditional DBMS like PostgreSQL, or SSDs for systems like LeanStore [89] and Umbra [112]) and provision their system accordingly,
2. expect a fixed and rigid hierarchy of storage devices (e. g., an HDD as persistent storage, a small transient SSD cache for warm data, and a volatile DRAM cache), which makes the system inflexible to changing workload patterns or newly available hardware,
3. or have to resort to workarounds, such as overriding the system and storing data manually in separate tablespaces on different devices without any guidance, which incurs a high administration overhead and is prone to misjudgments by the user.

Furthermore, existing approaches maximize performance for previously purchased storage devices and do not guide the user in choosing the best combination of storage devices for their workload.

(2) Existing database systems are built with no longer relevant bottlenecks in mind. Expensive DRAM modules make it prohibitively expensive to buffer the whole working set in main memory [112]. Thus, before the advent of fast, modern storage hardware, database system developers could reasonably assume that the system's bottleneck was accessing the disk. This lead to conceptually elegant but slow design decisions, such as e. g., implementing volcano style execution models [46] (incurring a lot of virtual function calls [110, 111, 141]), employing expensive locking (instead of cheaper, but harder to reason about latching or lockless data structures) [16, 87], or relying on synchronous, expensive, and slow (but easy to use) system calls to transfer data between external storage and the CPU. The advent of modern storage technologies partially fixes the storage bottleneck but uncovers previously hidden bottlenecks in those design decisions. Thus, modern database systems must revise the traditional system architecture, introduce abstractions more fitting to the current hardware stack, and develop new interfaces using those abstractions.

(3) Existing database systems are hard to adapt to breakthroughs. As storage device performance improvements have been incremental over the last decades, overall system performance improvements also have been incremental. Consequently, the architecture and application areas of traditional relational database systems (RDBMS) have remained unchanged. However, new hardware often exposes new properties allowing developers to design utterly different database system architectures with new applications. Take, for example, PMem. Its unmatched low write latency, while giving persistence guarantees, enables developers to design conceptually much simpler and faster database systems. Among other benefits, such systems do not need a complex write-ahead log (as persisting data immediately does not come at a performance penalty) or group commits (usually employed to batch writes to background storage) as PMem excels at small random writes.

The increased write performance thus allows us to build simpler (with caveats, as will later be shown in Chapter 3) and more performant applications for previously unfavorable workloads (high write ratio, small inserts). Simpler and more performant database systems, in turn, have the potential to simplify the whole software stack: If a database system is very performant for small reads, no caching layer such as Redis [130] is required reducing the system architecture complexity. Suppose such a system can leverage modern hardware to handle higher query throughputs at lower latencies concurrently. In that case, the system might not need to support scaling out as it can handle more workloads on a single node. Staying on a single node then reduces overall costs (i. e., fewer servers), complexity (e. g., requiring no support for partitioning or distributed operators), and failure modes (e. g., does not have to handle node crashes or network partitioning).

However, as discussed in Challenge 1, many assumptions are baked into current system architectures, making it hard to fully leverage modern storage devices in such a way. While Challenge 1 concerns itself with uncovering and fixing bottlenecks in *existing* database system architectures, this challenge targets storage devices that expose an entirely new set of properties, like PMem, which require a complete system re-design to be fully utilized. We thus should re-think what is possible and open up new use cases that have not been possible before.

1.4 Opportunities and Thesis Contributions

In this thesis, we address the previously stated challenges by proposing Mosaic, a storage engine designed to accelerate *analytical* workloads, and *Plush*, a storage structure optimized for *transactional* workloads. Finally, we present our vision of data pipes which address the complexity of moving data to and from different storage devices.

Mosaic. In Chapter 2, we address Challenge 1 by presenting Mosaic, a storage engine for scan-heavy workloads on RDBMS that manages devices in a tierless pool and provides purchase recommendations for a specified workload and budget. Mosaic enables system administrators to utilize the multitude of available storage device types with varying throughput and costs for analytical workloads. In contrast to existing systems, Mosaic generates a performance/budget curve of possible storage device combinations that is Pareto-optimal. The user can then use this curve to inform their purchase decisions when shopping for storage hardware. Our approach uses device models and linear optimization to find a data placement solution that maximizes I/O throughput for the workload. With Mosaic, we can show that higher query throughputs at the same budget as state-of-the-art approaches are possible, or the user can choose similar query throughputs at a lower cost than existing solutions.

Plush. In Chapter 3, we present Plush which addresses Challenge 3. It is a write-optimized, hybrid hash table for PMem with support for variable-length keys and values, demonstrating how software can be designed to adapt to breakthroughs in storage technology. Plush plays to PMem's strengths of DRAM-like performance, byte addressability, and the persistency guarantees of conventional block storage. We also identify PMem's weaknesses, namely its low write bandwidth and high media write amplification, and present approaches to mitigate them. On a 24-core server with 768 GB of Intel Optane DPCMM, Plush outperforms state-of-the-art PMem-optimized hash tables by up to 2.44× for inserts while only using a tiny amount of DRAM. Established PMem index structures mainly focus on lookups and cannot leverage PMem's low write latency. Plush achieves this speedup by reducing write amplification by 80%. For lookups, its throughput is similar to that of established PMem-optimized tree-like index structures.

Data Pipes. In Chapter 4, we address Challenge 2 by addressing a significant bottleneck in modern database systems: Data movement. Today's storage landscape offers a deep and heterogeneous stack of technologies that promises to meet even the most demanding data-intensive workload needs. The diversity of technologies, however, presents a challenge. Parts of it are not controlled directly by the application, e. g., the cache layers. Parts that *are* controlled often require the programmer to deal with very different transfer mechanisms, such as disk and network APIs. Combining these different abstractions requires great skill, and even programs written by experts can lead to sub-optimal utilization of the storage stack and present performance unpredictability.

To address this problem, Chapter 4 proposes a new programming abstraction called *Data Pipes.* Data pipes offer a new API that can express data transfers uniformly, irrespective of the source and destination data placements. By doing so, they can orchestrate how data moves over the different layers of the storage stack explicitly and fluidly. We suggest a preliminary implementation of Data Pipes that relies mainly on existing hardware primitives to implement data movements. We evaluate this implementation experimentally and comment on how a full version of Data Pipes could be brought to fruition.

1.5 Prior Publications and Authorship

Parts of the work presented in this thesis have previously been published as research papers at multiple database system conferences. Although I am the principal author of all publications, they were published in collaboration with multiple co-authors. Chapter 2 is drawn from the publication "Mosaic, a Budget-Conscious Storage Engine for Relational Database Systems" [149] which was published at VLDB 2020 and has been created in collaboration with my advisor Alfons Kemper as well as Thomas Neumann, Viktor Leis, and Alexander van Renen from TUM, and Satoshi Imamura from Fujitsu. Chapter 3 is drawn from the publication "Plush: A Write-Optimized Persistent Log-Structured Hash-Table" [150] published at VLDB 2022 with the support of my supervisor Alfons Kemper and my collaborators Thomas Neumann, Jana Giceva, and Alexander van Renen from TUM, and Satoshi Imamura from Fujitsu. Chapter 4 builds on the publication "Data Pipes: Declarative Control over Data Movement" [148], which was published as a vision paper at CIDR 2023 and is an international collaboration with Daniel Ritter from SAP, Danica Porobic from Oracle, Pınar Tözün from the IT University of Copenhagen, Tianzheng Wang from SFU, and Alberto Lerner from the University of Fribourg. In this thesis, I use the first person plural to reflect the contributions of my collaborators.

CHAPTER 2

Mosaic: A Budget-Conscious Storage Engine

Excerpts of this chapter have been published previously [149].

2.1 Introduction

For analytical queries on large data sets, I/O is often the bottleneck of query execution. The simplest solution is to store the data set on fast storage devices, such as NVMe SSDs. While it is prohibitively expensive to store all data on such devices, systems can leverage the inherent hot/cold clustering of data. Workloads often have a small working set, and storing the cold data on fast, but expensive devices wastes money. It would be better for a storage engine to store it on a cheap HDD instead, as no performance penalty is incurred.

Traditional RDBMS are unsuitable for this task. Most are optimized for a specific class of storage devices and assume that all data will be stored on a device of the given class. Traditional RDBMS, such as PostgreSQL or MySQL, are optimized for HDD and only maintain a small DRAM cache. Modern systems like HyPer [84], SAP HANA [39], or Microsoft Hekaton [35] are built for DRAM. Our database system, Umbra [112, 155], is optimized for SSD. Some allow system administrators the freedom to choose where to place data, even if they are not designed for multiple types of storage devices, for instance, via tablespaces. Here, the administrator can choose the storage location (and thus the storage device) for each table. However, moving an entire table either wastes fast storage space or negatively impacts on performance, as a table's cold columns are always moved together with its hot columns. Therefore, enabling *column-granular* placement allows for a much more *cost-efficient* storage allocation.

This problem is well-known in the big data world. Big data query engines like Spark [165] are therefore optimized for column-major storage formats like Parquet [70, 134]. These file formats support the splitting of tables and their columns into multiple files, so that they can be distributed between multiple nodes. Heterogeneous, tiered storage frameworks, such as OctopusFS [77], hatS [126], or CAST [26], distribute these chunks over multiple devices. They are very good at eking out every ounce of performance from the storage devices. Their downside is that they cannot judge if the provisioned storage devices are a good fit for the workload, as they are installed after the system has already been purchased and provisioned. Big data query engines using such storage frameworks are often distributed systems optimized for cluster operation. While this enables scaling to very large data sets, it incurs significant CPU and networking overheads. Queries are therefore more frequently CPU- or network-bound than is the case with traditional RDBMS.

It would thus be beneficial to have column-granular table placement on single-node RDBMS. While table-granular placement is not optimal for the reasons mentioned above, a user can at least manually determine a sensible placement on the basis of their experience. For column-granular placement, however, it is considerably harder to find a good solution by manual means, as the number of possible placements grows exponentially in the number of columns and storage devices.

Big data engines and RDBMS with heterogeneous storage frameworks have another shortcoming. A modern server can have storage devices of multiple classes: DRAM, PMem, NVMe SSDs, SATA SSDs, and HDDs in different RAID configurations, and all are competitive at their price point. A system administrator buying a new database server cannot determine the optimum configuration that will achieve the required throughput at the lowest cost.

We therefore propose Mosaic, a storage engine for RDBMS that is optimized for scan-heavy workloads and covers the entire deployment process of a database system: (1) hardware selection, (2) data placement on purchased hardware, and (3) adaption to changing workloads. Mosaic uses purchased storage devices to their full potential with column-granular placement, ensuring an optimum throughput/performance ratio at *all* budgets. So as not to restrict the user in the purchase process, Mosaic does not categorize storage devices into tiers but organizes all devices in a tierless pool. A conventional storage engine, in contrast, has distinct tiers (e. g., HDD, SSD, and DRAM). Mosaic's tierless design allows the user to mix device classes (e. g., adding an NVMe SSD to a system already equipped with a SATA SSD, where a tiered approach would only have an SSD tier).

Figure 2.1 compares Mosaic to existing approaches. The x-axis shows the cost of the installed storage devices, the y-axis the throughput of the system.

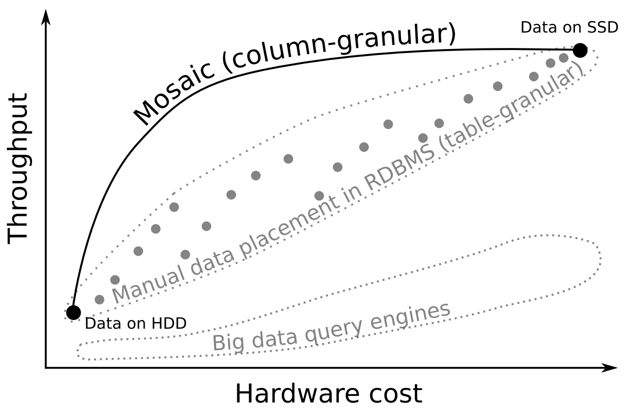


Figure 2.1: Estimated performance spectrum of Mosaic compared to big data query engines like Spark and manual data placement.

Big data engines do not scale well with the price of the storage devices used as they are rarely I/O-bound, even for smaller workloads. RDBMS scale well but offer no automated mechanism for data placement and are restrained to table-granular placement. Manual data placement does not guarantee that the choice is Pareto-optimal or fits the data set. Mosaic's automatic column-granularity placement not only increases throughput for scan-heavy workloads at all price points, but it also empowers the user to find the best configuration within a given budget or subject to specific performance requirements.

Mosaic is an improvement over existing solutions during all stages of the deployment process:

- Before hardware is purchased: given a typical set of queries and a list of devices available for purchase, Mosaic gives purchase recommendations for arbitrary budgets. Each recommendation is guaranteed to be Pareto-optimal, i. e., no other configuration is faster while also being cheaper.

- After purchase: given the trace of a typical set of queries and a set of storage devices, Mosaic places data optimally to maximize throughput. Mosaic can work with any set of storage devices, not only those that have been bought on the basis of its recommendations.

- During operation: Mosaic acts as a pluggable storage engine component for any columnar relational database system.

In summary, our key contributions are:

1. We present Mosaic, a column-based storage engine for RDBMS, optimized for scan-heavy OLAP workloads and using a device pool without fixed

tiers. In contrast to existing approaches, it places data with column granularity.

2. We design a placement algorithm based on linear optimization that finds optimum data placement for a workload.

3. We design a prediction component for Mosaic that gives purchasing recommendations along a Pareto-optimal price/performance curve.

4. We integrate Mosaic into the database sytem Umbra [112] and point out its benefits over state-of-the-art approaches.

2.2 Background and Related Work

While, to the best of our knowledge, Mosaic has no direct competitor, all of its design goals have been achieved in other systems individually, but never together. These systems are therefore not able to leverage the synergies of implementing *all* of Mosaic's design goals.

2.2.1 Heterogeneous Storage for Big Data Query Engines

Data set sizes have over time outgrown the storage capacity of single systems, which is why big data engines were introduced. Most query engines, such as MapReduce [32] or Spark [165], support the Hadoop file system (HDFS) [140]. This splits files into blocks, which it replicates across nodes. Until recently, nodes were unaware of the characteristics of their storage device and therefore could not place data in a workload-aware fashion.

Multiple extensions to HDFS have been developed to rectify this issue. Kakoulli et al. implement OctopusFS [77], a tiered distributed file system based on HDFS. OctopusFS uses a model to infer a data placement for a fixed set of tiers (DRAM, SSD, and HDD) that maximizes throughput. The authors later built on OctopusFS and developed an automated tiered storage manager [55] that uses machine learning to decide on which blocks to up- or downgrade.

CAST [26] recognizes that cost models and tiering mechanisms used for operating systems do not solve problems of OLAP style workloads, as they rely on access characteristics that are atypical for an OS, i. e., large, sequential table scans. Multiple other works have introduced a heterogeneity-aware layer on top of HDFS using fixed tiers [69, 126, 127, 128]. Snowflake does not rely on HDFS but uses its own tiered distributed storage system, optimized for cloud operation [152].

What all solutions building on HDFS have in common is that they focus on opaque HDFS blocks as atomic units of storage. As they do not know what is stored inside those blocks, they cannot make domain-specific optimizations. Mosaic knows its domain (retrieving columns for table scans in an OLAP context), and its placement strategies can take this into account. Mosaic deliberately decides against a tiered architecture common in HDFS approaches, as new hardware does not always cleanly map onto existing tiers.

The approaches referred to in this section do not offer any purchase recommendations. In contrast to Mosaic, they have to manage replication and data locality, as they run on clusters. While replication is orthogonal to Mosaic (i. e., one could extend Mosaic to replicate data), we focus on a single machine for now, to simplify the data model.

2.2.2 Heterogeneous Storage in RDBMS

RDBMS hide the throughput gap between DRAM and background storage with a buffer pool. In the last decade, when SSDs became affordable, a lot of work was done to profit from their improved throughput and random access characteristics. For example, Umbra is optimized for SSDs [112]. It provides main memory-like speed when the working set fits into the DRAM buffer pool and gracefully degrades to SSD-speed with larger working sets, an idea first implemented in LeanStore [89].

Algorithms have been designed for caching data on SSDs into DRAM [73, 74, 145] or using SSDs as cache for HDDs [61]. MaSM [6], for example, uses an SSD cache for out-of-place updates. DeBrabant et al. [33] introduce anti-caching, where DRAM is the primary storage device, and cold data is evicted to HDD. Stoica and Ailamaki [143] reorganize cold data so that the OS can efficiently page it out. Another approach is to build buffer pools with multiple tiers [37, 80]. Here, SSD is a caching layer for slower devices like HDDs. These approaches, however, waste valuable storage space on SSDs, as data is replicated across multiple tiers. While caching is necessary for volatile devices like DRAM, it is not needed for persistent storage. Hybrid storage managers circumvent this issue by placing the data on different device classes without caching [20, 98, 104, 167]. hStorage-DB [104], like Mosaic, uses information from the query engine to place data on HDDs and SSDs.

These approaches focus on HDD and SSD. With new storage technologies, such as persistent memory, the whole cycle of research begins anew as RDBMS now have to integrate a new layer of storage [4, 132]. Mosaic finally breaks this cycle by being device-agnostic. Every device is instead parameterized by the user and added to a tierless pool. The user can add or remove new device classes without having to re-engineer Mosaic. General-purpose data placement systems [64, 116, 154] are not restricted to relational data and therefore, unlike Mosaic, cannot make use of domain knowledge.

2.2.3 Prediction and Storage Recommendation

Mosaic's prediction component also builds on prior work. Wang et al. built a MapReduce simulator [153] to investigate the impacts of different design decisions, such as data placement or device parameters, on performance. In contrast to Mosaic, their tool only plans new setups and does not act as a storage engine. Herodotou et al. designed a 'what if' [54] engine capable of comparing different configurations and giving recommendations for MapReduce jobs. Cheng et al. went a step further and designed CAST [26], a tiered storage framework for MapReduce jobs. It gives data placement recommendations for a cloud context that minimize cost while maintaining performance guarantees. However, they also limit themselves to a predefined set of device classes.

Guerra et al. developed a general-purpose framework for dynamic tiering [49]. Like Mosaic, it has an advisor that gives purchase recommendations and a runtime component that retrieves data. In contrast to Mosaic, however, it operates on opaque data chunks instead of tables. While this approach is more generalized, it has the downside that it cannot make domain-specific optimizations, as explained above.

Wu et al. developed a general-purpose hybrid storage system with an approach similar to that of Mosaic [157]. It forgoes tiering and places data so that the bandwidth of all devices is fully utilized. Unlike Mosaic, however, it only supports a mixture of identical HDDs and SSDs (i. e., not multiple HDDs or SSDs at different speeds).

2.3 Mosaic System Design

Mosaic comprises four components, as shown in Figure 2.2. Mosaic collects metadata from users before they purchase devices and while running as a storage engine. Information about attached devices, their measured performance, and recorded traces is kept in a metadata store (Section 2.3.1). Mosaic stores its managed data in a storage format that is optimized for the access characteristics of its devices (Section 2.3.2). The data retriever is an interface between the storage layer and the DBMS (Section 2.3.3). The data placement component distributes data between attached storage devices so that the data retriever can maximize the average throughput for a workload (Section 2.3.4). It is also responsible for predictions and purchase recommendations.

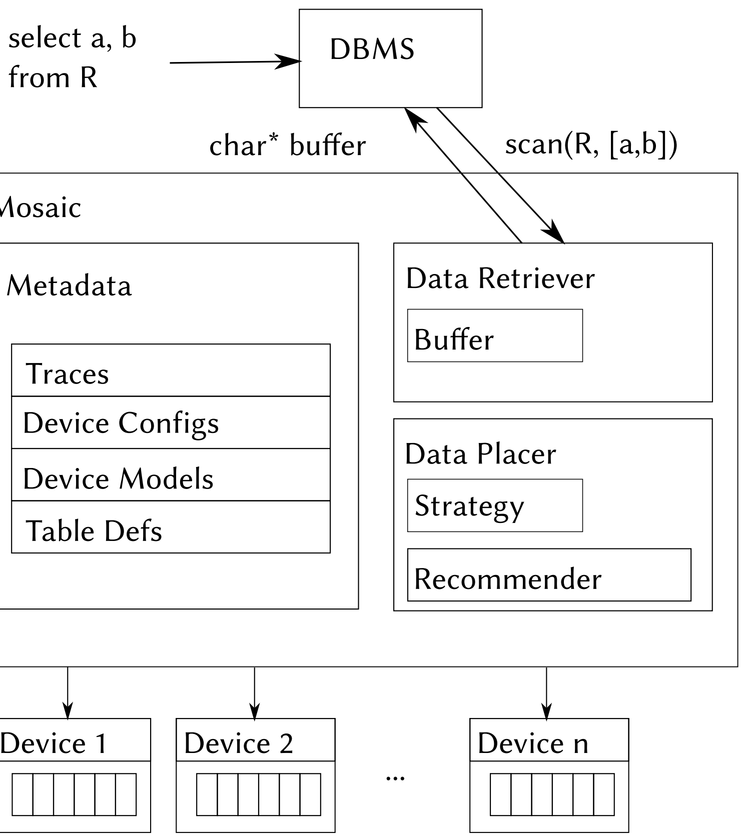


Figure 2.2: The components of Mosaic and its interface with the RDBMS.

The components are interdependent: An inappropriate storage format (e. g., one that uses a slow compression algorithm for a storage device with high throughput) reduces overall throughput, even if the data placer finds an optimal placement. The same goes for the data retriever: If data placement is not optimal or Mosaic chooses the wrong compression type for the data, performance suffers, even if Mosaic uses the whole throughput of a storage device.

2.3.1 Metadata

Figure 2.3 summarizes the metadata gathered by Mosaic. The only information supplied by the user is a list of connected devices (Figure 2.3a). This contains each device's capacity, the optimum number of concurrent reader threads, cost[1],

[1]We use €-cents (ct) as the unit of currency as € is the currency in which we bought our evaluation system. If the absolute cost is unknown or subject to change, it is possible to define the cost relative to the cheapest device. For example, if an SSD costs three times as much as an HDD, enter 3 and 1 respectively.

```
[[device]]
    id = 0
    mnt = "/mnt/nvme"
    name = "NVMe SSD"
    capacity = 60 GB
    threads = 8
    compression = "ZSTD"
    cost_per_gb = 60 ct
```

(a) Device configuration entry

```
[[trace]]
    A:x,y
    B:z
    A:x
    C:u,v
    A:y
    C:v,w
    ...
```

(b) Excerpt of a trace

```
A:  x -> int,
    y -> varchar(200)
B:  z -> int
C:  u -> int64,
    v -> varchar(10),
    w -> int
```

(c) Table definitions

```
[[model]]
    device_id = 0
    none = 2.1 GB/s
    LZ4 = 1.6 GB/s
    zstd = 1.2 GB/s
    seek = 0
```

(d) Device model

Figure 2.3: Metadata recorded, maintained, and stored by Mosaic.

and the preferred compression algorithm. Since Mosaic does not depend on fixed device tiering, the user can add and remove devices to the pool during runtime by editing the device configuration file. Mosaic records table scans of queries being run since the last time the user triggered data placement in a trace file (Figure 2.3b). If the user has not yet purchased any storage devices, but wishes to receive a purchase recommendation, Mosaic can generate a trace file without a data set being present. Mosaic then extracts the table scans from each query and inserts them into the trace file. The trace file allows Mosaic to match the accessed data chunks to columns, and the columns to table scans, and shows which columns the DBMS has frequently accessed together. The data placer uses the trace file to infer the optimum data placement for the recorded workload. This is an advantage over established big data file systems. HDFS, for example, only keeps access statistics per file, with no insight into what a file consists of. Mosaic can derive data interdependencies from the trace file (i. e., it can determine which columns are often queried together).

Mosaic furthermore extracts table definitions from the data set (Figure 2.3c) and periodically measures the throughput of all attached devices for the current workload and stores it in a device model file (Figure 2.3d). The predictor uses this file to predict how the data retriever would perform with hypothetical data placements.

2.3.2 Storage Format

We adapted Apache's established Parquet file format to form the Mosaic data storage format. It is a columnar data storage format, and it has many properties beneficial to Mosaic:

1. Parquet stores data in a column-major format with columns further subdivided into chunks comprising pages. Mosaic extricates column chunks out of existing Parquet files and distributes them between storage devices.

2. Parquet can compress pages individually with a variety of compression algorithms. Mosaic can thus compress column chunks depending on their storage device and recompress them with a different algorithm during migration.

3. Parquet's internal data format has built-in support for partitioning a relation on multiple files. We extend this so that Mosaic can place any column on any storage device.

4. Parquet stores one metadata block per column chunk and separates data and metadata. Mosaic can thus easily move them independently of each other. Instead of storing the column chunk metadata with the data itself, we reserve some storage space on a m<etadata device chosen by the user. This ensures that reading metadata does not affect concurrent reads on devices not suited to random reads (i. e., HDDs).

Mosaic allows the user to choose a compression algorithm for each device. Compressing the stored data has multiple advantages:

1. Mosaic can store more data while staying within budget, as compressed data takes up less space.

2. When the decompression throughput of the CPU is higher than a device's throughput, data compression increases the effective throughput.

3. When data on faster devices is compressed, Mosaic can move a greater percentage of the working set to those devices, thus increasing overall throughput.

Column-major storage and compression make random accesses and updates more difficult. This, however, is no issue for Mosaic, as it focuses on scan-heavy workloads.

2.3.3 Data Retrieval

Mosaic's data retriever component retrieves stored data and converts it into a format that the RDBMS is able to read. The smallest unit of storage it can retrieve on request is a column chunk, which, by default, comprises 5 million values. At a higher level, the RDBMS can also request entire table scans. When the RDBMS triggers a table scan, Mosaic asynchronously fetches chunks of the requested columns in ascending order, until the buffer is full.

Mosaic can keep this buffer small: It assumes that queries are I/O-bound, and the RDBMS is thus limited by the speed at wich Mosaic fills the buffer. The buffer only holds the set of chunks that the RDBMS is actively processing and the set of chunks being concurrently prefetched. The size of the buffer thus depends on the number of columns being scanned and their data type. In our experiments, it never exceeded 1 GiB. Whenever the data retriever has buffered a set of chunks, it notifies the RDBMS of the new data via a callback. As soon as the RDBMS has processed a chunk, Mosaic evicts it from the buffer. Prefetching is straightforward, as Mosaic only needs to support linear table scans.

As the data placer can store columns of a table scan on different devices, Mosaic must read from multiple devices in parallel. Devices such as NVMe SSDs only reach their maximum throughput when multiple threads read concurrently. Mosaic's data retriever maintains a thread pool with reader threads. It assigns each requested column chunk to a reader thread. As most table scans access multiple columns, the data retriever reads the chunks of all requested columns in parallel. This is important, as the slowest reader determines overall throughput. The placement strategy must ensure that the chunks are placed in a way that maximizes the data retriever's throughput. The placer thus has to make sure that a column on a slow device does not stall a table scan whose other columns are on fast devices. Each reader forces the OS to sequentially populate its page cache with the relevant data. Without this step, random access by the RDBMS or the decompressor could reduce the throughput.

While SSDs need concurrent access to maximize throughput, HDDs have a large drop in throughput if accessed concurrently, as sequential access will degenerate into random access when multiple threads contend for the device. Mosaic thus lets the user set a per-device thread limit. A semaphore guards each device to ensure that the number of threads reading from a device in parallel never exceeds the optimum.

Before returning to the *reader* thread pool, reader threads add the chunk's data, which the OS page cache now buffers, to a queue. This queue is ordered by the chunk request time. Mosaic now decompresses the queued chunks. Since the throughput of a storage device could be higher than that of a single thread that is decompressing data, Mosaic maintains a *decompression* thread pool. Whenever a decompression thread is idle, it fetches the first chunk in the queue, decompresses it, and makes the resulting values available to the RDBMS via its callback.

2.3.4 Data Placement

Mosaic places data offline and only reorders data when prompted to do so by the user. To this end, it starts a new trace file after each placement or on manual prompt by the user. It stores information about the columns accessed by the RDBMS in the trace file of that epoch. It enters a record for each table scan of every query executed. Each record stores the table, and the columns requested by the table scan. As explained in Section 2.1, Mosaic's placement module has two modes. Before devices are purchased, Mosaic is in *budget* mode. When they are installed, Mosaic switches to *capacity* mode. In *budget* mode, Mosaic calculates a recommended placement on the basis of a given budget. In *capacity* mode, it distributes the data between the connected devices up to their capacity as specified in the device configuration metadata. When in *budget* mode, Mosaic considers all devices of the device configuration metadata as targets regardless of whether they are present. This mode does not restrict the device's capacities, but it does restrict their cost. Here, Mosaic's recommender provides the user with the recommended hypothetical placement along with a set of devices and their capacity for installation. Mosaic ensures that the total device cost does not exceed the user-defined maximum budget. In both modes, Mosaic uses swappable placement strategies to calculate a data placement. The next section summarizes the strategies employed.

2.4 Data Placement Strategies

For performance predictions, Mosaic not only needs to place data optimally, i. e., to find the best placement solution *qualitatively*, it also has to predict performance *quantitatively*. Mosaic consequently needs a model that can predict how data placement impacts query runtime (Section 2.4.1).

Mosaic supports pluggable data placement strategies (Section 2.4.2). The following three sections present three different placement strategies. The first two of these (Section 2.4.3 and Section 2.4.4) are used by multiple state-of-the-art systems. They were designed for a tiered storage engine, i. e., they assume that 'slow' and 'fast' layers exist, between which they can move the data. However, as Mosaic is a tierless engine, they are not a good fit. We therefore use them as a baseline against which we compare our contribution, which is the third strategy, called LOPT, and is explained in Section 2.4.5.

2.4.1 A Model for Predicting Table Scan Time

Since Mosaic not only offers data placement for installed devices but also predicts performance for hypothetical configurations, it needs a model on which to base its predictions. To keep complexity down, we make three assumptions:

(1) Columns are atomic. We assume a column is stored contiguously on a single device. Mosaic can split columns at the parquet column chunk level and distribute the chunks on multiple devices. The prediction component, however, considers columns to be atomic. This speeds up placement calculation, as only whole columns have to be placed, which reduces the complexity of the model. It also has the added benefit that placement calculation is independent of data set size, as the number of columns and therefore possible placement permutations is constant in the number of tuples. Distributing chunks on multiple devices only benefits runtime performance if some chunks of a column are read disproportionally frequently and therefore profit from being on faster devices. This is only the case if data is either sorted (which is only possible for one column per table) or the query is so selective that chunks can be skipped. This is unrealistic with large chunk sizes. While possible with smaller chunk sizes, Mosaic cannot shrink chunks too far as the placement calculation would become too expensive.

(2) Queries are I/O dominated. To keep the model agnostic of the query execution engine, we ignore computation times, such as aggregation, joins, or predicate evaluation and we only model table scans. Each query comprises one or more table scans, each of which reads one or more columns. Columns on different devices can — and should — be scanned in parallel. While this assumption might reduce absolute prediction accuracy for CPU-bound workloads, predictions will still be correct in relation to each other, as the computation overhead is constant. The overhead only depends on the contents and size of its tables, not on data placement and only adds a constant error to all predictions, assuming the computation overhead is not shadowed by I/O.

(3) The throughput of a device is independent of the number of columns being read in parallel. We assume that Mosaic can saturate a device's I/O bandwidth regardless of how many columns it reads in parallel. This is true for SSDs, which benefit from multi-threaded reads. It is wrong for HDDs, whose throughput decreases when reading columns in parallel, because of their seek time. Since we solve this problem on the architecture side by reading columns a chunk at a time and using per-device semaphores (see Section 2.3.3) that ensure that only one thread at a time can read from a HDD, we need not model it.

The model we built is based on these assumptions. It predicts the total execution time t_{total} of a set of table scans TS given a set of devices D and a set of columns C.

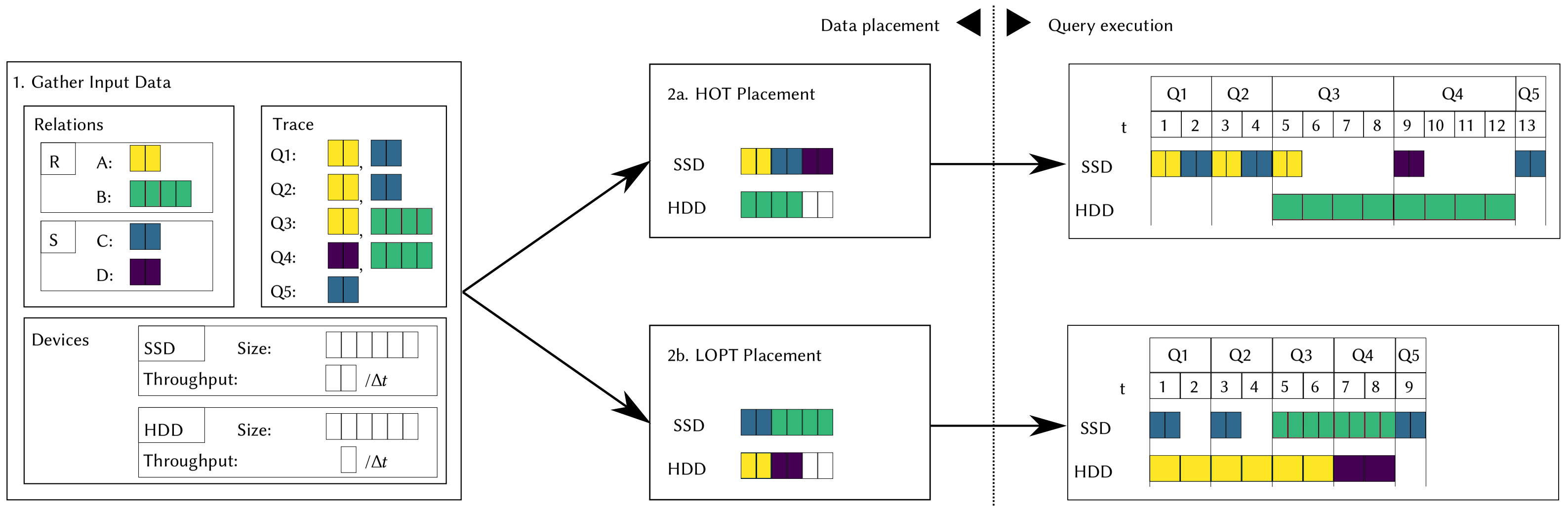


Figure 2.4: Modus operandi of Mosaic, and two exemplary placement strategies. The HOT algorithm indiscriminately moves the most frequently accessed columns to the SSD. The LOPT algorithm finds the data placement with the least storage device idle time and thus speeds up sequential execution of the 4 sample queries by 30%. For demonstration purposes, we assume that the SSD has twice the throughput of the HDD.

For each column $c \in C$, the function *size* returns its size:

$$\text{size} \ : C \to \mathbb{N} : \quad \text{size of column}$$

For uncompressed data, *size* is the product of the number of tuples in the column and the size of the column's data type. For compressed data, Mosaic looks up its size in the metadata of each column chunk.

Each table scan $T \in TS$ is a subset of C, and each device $d \in D$ is modeled as a 5-tuple

$$d = \langle t_{\text{seek}}, cr, t, \text{capacity}, \text{cost} \rangle \tag{2.1}$$

with the following values:

$$\begin{aligned} t_{\text{seek}} &: \quad \text{seek time} \\ \text{cr} &: \quad \text{compression ratio} \\ t &: \quad \text{throughput} \\ \text{capacity} &: \quad \text{capacity} \\ \text{cost} &: \quad \text{cost per unit of storage} \end{aligned}$$

These values are stored in the user-provided device configuration entry (see Section 2.3.1), with the exception of t, the continuously measured throughput stored in the device model metadata.

Equation (2.2) expresses the time $t_{d,c}$ required to scan a column $c \in C$ stored on a device $d \in D$:

$$t_{d,c} = t_{\text{seek}} + \frac{\text{size}(c)}{\text{cr}(d) \cdot t(d)} \tag{2.2}$$

The fraction $\frac{\text{size}(c)}{\text{cr}(d)}$ is an estimation of the compressed size of c on d. If the column has already been stored on d or another device with the same compression algorithm, Mosaic looks up the actual size instead of estimating it.

When the placer stores two or more columns relevant to a table scan on different devices, the retriever can read them in parallel. The runtime of each table scan $T \in TS$, t_T is thus only determined by the device taking the longest, as seen in Equation (2.3).

$$t_T = \max\{\sum_{c \in T} I_{d,c} \cdot t_{d,c} \mid d \in D\} \tag{2.3}$$

I is an indicator function:

$$I_{d,c} = \begin{cases} 1 & \text{if column } c \text{ is stored on device } d \\ 0 & \text{otherwise} \end{cases}$$

The total time required to run the set of table scans TS is the sum of the runtime of each table scan:

$$t_{\text{total}} = \sum_{T \in TS} t_T \tag{2.4}$$

The model allows the approximate cost of a real or hypothetical data placement to be calculated:

$$\text{cost}_{\text{total}} = \sum_{c \in C} \sum_{d \in D} I_{d,c} \cdot \text{cost}(d) \cdot \frac{\text{size}(c)}{cr(d)} \tag{2.5}$$

Mosaic's data placer, given I, moves all columns to the device specified by I. I is an abstraction over specific placement strategies and their implementations. A strategy can either determine I algorithmically (Sections 2.4.3 and 2.4.4) or with a constraint solver (Section 2.4.5). Mosaic's prediction and placement component is therefore independent of the placement algorithm.

2.4.2 Responsibilities of a Strategy

As seen in Figure 2.4, the data placer supplies each strategy with a number of inputs. These are

1. the size of each relation's columns,
2. the throughput, size, price per gigabyte, and optimal number of parallel readers for each attached device, and
3. a trace with the table scans since the start of the current epoch.

A strategy places columns on storage devices in such a way that the average throughput of a workload similar to the trace is maximized. Figure 2.4 shows two such strategies. Strategy (a), called HOT, places the columns read the most often (i. e., the 'hottest' columns) on faster devices. Strategy (b), called LOPT, finds the optimum placement using linear optimization. As can be seen on the right-hand side, the choice of strategy impacts the overall throughput. The HOT strategy cannot profit from the fact that Mosaic reads data from multiple devices in parallel. The LOPT strategy, in contrast, uses both devices, concurrently decreasing the overall table scan time in the example by $\approx 30\%$.

2.4.3 HOT Strategy at Table Granularity

The table granular HOT strategy (HOT table) treats each table as an atomic entity that can only ever live on one single device at a time. The strategy places tables according to their 'hotness'. It assumes that a table that the RDBMS scans often (being 'hot') benefits from being on a fast device. Improving a table scan that runs more often has an overall higher positive impact on average throughput. It places tables descending in order of their number of accesses on the fastest device with enough space for the whole table.

This strategy is an approximation of the toolset available to administrators of many established RDBMS such as PostgreSQL or Oracle. These systems allow database administrators to create different tablespaces on different devices and assign each table to a specific tablespace. While these systems do not allow automatic data placement like Mosaic does, we assume that a system administrator using tablespaces will decide in the same way as the HOT strategy: they will move tables appearing disproportionally often in observed queries to faster devices.

2.4.4 HOT Strategy at Column Granularity

The column-granular HOT strategy (HOT column) is an improvement over the table granular version. As before, data accessed more frequently is considered 'hot' and so is placed on devices with higher throughput. But this time, tables are no longer treated as atomic. Instead, HOT column migrates single columns of tables. This is a huge improvement over HOT table, as even the hottest tables often have multiple columns that are only rarely queried. HOT column will rightfully prioritize warmer columns of cold tables over cold columns of hot tables. While the HOT approach has been proven to be workable by many existing tiered storage engines, it has multiple weaknesses.

1. HOT relies on a tiered architecture in which data is moved up or down one tier at a time. With HOT, Mosaic can emulate such a hierarchy with two or three devices that have large performance gaps (say, an HDD and an SSD). If we, however, add multiple devices whose throughputs are close (i. e., multiple HDDs, or a SATA SSD and a RAID 5 of multiple HDDs) the HOT strategy can no longer cleanly bin those devices into distinct tiers.

2. As can be seen in Figure 2.4, HOT does not place data such that a table scan can be parallelized. If the RDBMS often scans two hot columns together, they would benefit from being on different devices so that Mosaic could read from both devices in parallel. HOT would try to place both on the fastest device available, leaving optimization potential on the table.

3. Mosaic can only apply the HOT placement strategy if it knows the device capacities beforehand. If Mosaic is in *budget* mode, it is not obvious how to choose device sizes to maximize throughput.

2.4.5 Linear Optimization Strategy

Rather than using a heuristic to place data, the linear optimization strategy (LOPT) uses the model defined in Section 2.4.1 to find an optimal solution. LOPT deems a solution optimal if it minimizes the time spent scanning tables for a set of queries. It uses a constraint solver to define the indicator function I in such a way that t_{total} of Equation (2.4) is minimized.

LOPT subjects Equation (2.4) to the following constraints for each column:

$$\forall c \in C : \sum_{d \in D} I_{d,c} = 1 \tag{2.6a}$$

$$\forall c \in C : \forall d \in D : I_{d,c} \in \{0,1\} \tag{2.6b}$$

A column has to be stored exactly once (2.6a) and is completely stored on a device or not at all (2.6b). LOPT enforces one of two additional constraints, depending on the mode:

a) In *capacity* mode, the strategy infers optimal placement for previously purchased hardware. A valid placement must therefore not exceed the storage capacity of any installed device. Mosaic thus subjects Equation (2.4) to the following additional constraint for each device:

$$\forall d \in D : \left(\sum_{c \in C} I_{d,c} \cdot \frac{\text{size}(c)}{\text{cr}(d)} \right) \leq \text{capacity}(d) \tag{2.7}$$

b) In *budget* mode, the strategy predicts the optimum placement for a budget cost_{max}. Since no hardware has been bought yet, Mosaic can ignore all the capacity limitations but has to stay below budget. Mosaic subjects Equation (2.4) to the following additional constraint:

$$\left(\sum_{d \in D} \sum_{c \in C} I_{d,c} \cdot \text{cost}(d) \cdot \frac{\text{size}(c)}{\text{cr}(d)} \right) \leq \text{cost}_{\text{max}} \tag{2.8}$$

Mosaic uses Gurobi [51] to solve this optimization problem. Gurobi is a constraint solver with support for mixed-integer programming (MIP).

LOPT strategies' advantage over HOT variants is that it makes use of all the information encoded into the model which has the following upsides:

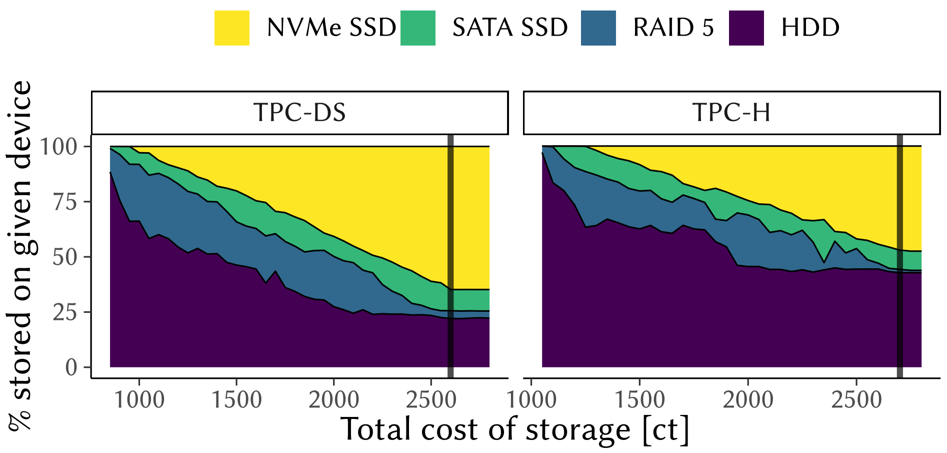


Figure 2.5: LOPT data placement in *budget* mode for TPC-H and TPC-DS (SF 100) at different budgets. The vertical lines indicate from when an increased budget does not increase performance.

- As Figure 2.4 shows, HOT ‘leaves bandwidth on the table’. It underutilizes slower devices, which — while having less throughput than their faster counterparts — could still contribute to overall throughput. This is because HOT tries to concentrate hot data on a few, fast devices. LOPT is free to place hot data on slower devices if a larger column is the bottleneck of the table scan.
- LOPT is aware that it is optimizing table scan performance and makes domain-specific optimizations through its modeling. It does not waste precious space on faster storage devices for columns that are hot but are often queried together with colder columns.
- The user can easily extend LOPT. A user might, for example, want to model a limited amount of expansion slots, a maximum/minimum size of each storage device, or a power budget constraint. With LOPT, they can just add new dimensions to the device model and add additional constraints for those dimensions to the solver. The solver will then find the best solution given the additional constraints. No further changes to Mosaic are needed.

Figure 2.5 shows the advantages of LOPT and its *budget* mode for an exemplary storage configuration. It comprises a fast NVMe SSD, a slower SATA SSD, an even slower HDD, and a RAID 5 of three HDDs. At lower budgets, LOPT in *budget* mode does not spend all the available money on a fast NVMe

Table 2.1: Storage devices of the evaluation system.

Device	Price per GB	Throughput
NVMe PCIe SSD	125 ct	2.10 GB/s
SATA SSD	60 ct	0.41 GB/s
RAID 5 of HDDs	45 ct	0.32 GB/s
HDD	30 ct	0.23 GB/s

drive. It instead distributes data between the four devices, maximizing overall throughput. Only with an increasing budget does LOPT gradually place data on the fast NVMe SSD. Even at high budgets, it still keeps parts of the data on SATA SSD. To save costs, it keeps never-touched data (25% for TPC-DS, 50% for TPC-H) on HDD. LOPT can thus determine when adding additional hardware is just a waste of money. In the figure, this threshold is marked by a vertical line.

While LOPT is more sophisticated than HOT, it is also much harder to compute. Constraint (2.6b) that permits only integers is particularly constricting, as it forces us to employ MIP, which is NP-hard. But it is important to note that run time only depends on the device count and the number of distinct table scans. It is independent of the number of tuples (as we treat columns as atomic units) and queries. If multiple queries 're-use' the same table scans or the user runs a query multiple times, the model does not become more complex. LOPT just multiplies its modeled runtime for that query by the number of reuses, and the optimizer does not need to consider more variables. Section 2.5.5 evaluates placement computation cost in detail.

2.5 Evaluation

Table 2.1 shows the storage configuration of the evaluation system. The system comprises four different storage device classes, each competitive at its respective price point. Besides two SSDs of different speeds, we equip the server with four enterprise grade server HDDs at 10k RPM. We configure three HDDs as a RAID 5 and keep the fourth as a standalone disk. The server is equipped with 192 GB of DRAM and a single socket Intel Xeon Gold 6212U CPU with 24 physical cores @ 2.4 GHz (with SMT: 48 cores).

As explained in Section 2.3, choosing a fitting compression algorithm for each storage device increases throughput. While using no compression incurs no added CPU overhead, it requires the most space. LZ4 has a low CPU overhead with an acceptable compression ratio. Zstandard (ZSTD) has the highest compression ratio with a still acceptable CPU overhead. As expected, the synthetic

Table 2.2: TPC-H benchmark speedup (SF 30) of SSD and HDD for different compression algorithms.

	Speedup over HDD		
Device	None	LZ4	ZSTD
HDD	1	—	2.92
NVMe PCIe SSD	6.2	11.18	12.66

TPC-H SF30 data set compresses quite well, requiring 44.11 GB uncompressed, 16.51 GB if compressed with LZ4, and only 10.03 GB with ZSTD. ZSTD still yields a compression ratio of about 3 on real-world data sets (2 for LZ4) [171]. Table 2.2 shows the relative speedup of the TPC-H benchmark over the baseline for different compression algorithms. ZSTD compressed data takes up less space and increases overall performance compared to LZ4, even on PCIe SSD. For this setup, we therefore configure Mosaic to always compress data with ZSTD.

We run all benchmarks with Umbra as the database engine and Mosaic as its storage engine. We choose Umbra as it provides best-of-class speed and thus rules out CPU bottlenecks, unlike big data query engines. While RDBMS like MySQL also expose an interface for storage engines, they cannot easily be adapted to columnar data storage. A more detailed reasoning as to why we evaluate Mosaic only in conjunction with Umbra can be found in Section 2.5.9.

2.5.1 Benchmarks

For our evaluation, we use two OLAP benchmarks: TPC-H and TPC-DS. TPC-H comprises 22 queries and 8 tables. The largest table, *lineitem*, accounts for 70% of the data set size, while the smallest 5 tables together only make up 3%. Choosing the best placement for the columns of the *lineitem* table thus gives Mosaic a large optimization potential. TPC-DS is a much more complex OLAP benchmark. It comprises 99 queries and 24 tables. Since Umbra does not yet support all features required by TCP-DS, such as window functions, we discard unsupported queries. We thus run a subset of TPC-DS comprising 67 queries. We run both benchmarks at scale factor 30 and 100.

For both benchmarks, we define one run as a measurement of the runtime of each query executed once sequentially, with the execution times then added. To accurately measure a query's runtime, we execute it five times and take the mean. Before each query, we clear the OS cache to force Mosaic to read all data from the underlying storage devices. Running all queries sequentially just once is not a realistic benchmark. In reality, a workload is usually heavily skewed towards just a few queries. It is, however, the worst case for Mosaic and thus a good benchmark. The more distinct queries we run, the harder it is for Mosaic to find an optimal placement. The working set is also larger. Mosaic thus benefits less from expensive storage on faster devices.

2.5.2 Mosaic vs. Traditional RDBMS

In this section, we evaluate how Mosaic compares against the toolkit of a traditional relational database system. We compare Mosaic's column-granular LOPT placement strategy against table-granular placement. Table-granular placement is the status quo and the best option in an RDBMS such as Oracle or PostgreSQL.

We first import a trace of the TPC-H benchmark (executing queries 1 to 22 once in sequence). We then trigger Mosaic's LOPT placement strategy for different budgets. After data placement, we repeat the benchmark and record the runtime. As a baseline, we benchmark all table placement permutations for the four largest TPC-H tables that make up 98% of the total data set size. The remaining four smallest tables are always stored on NVMe SSD to keep the number of possible configurations manageable.

Figure 2.6 shows all unique table-granular placement configurations (▲) for HDD, SATA SSD, and NVMe SSD. Each configuration could have been chosen by a system administrator of a traditional RDBMS with tablespaces. We mark the three configurations in which Mosaic stores all five tables on the same device. The three distinct clusters correspond to the storage location of the *lineitem* table. At 6.8 GB, it contributes 70% of the total data set size, and its placement thus has the greatest effect on the total cost of storage. The Pareto-optimal line (- - -) shows the best case for table-granular placement, i. e., there is no cheaper placement that also reduces benchmark runtime. A system administrator can, therefore, hope at best to hit this line. For most budgets, Mosaic's LOPT placement strategy (-●-) dominates and offers the choice of having the same performance at less cost or more performance at the same cost. As indicated in the figure, at a budget of 536 ct, LOPT offers the same throughput as the Pareto-optimal table placement at 60% of the cost, or 41% of the runtime at the same budget. Table-granular data placement is only competitive if Mosaic places all data on the cheapest or most expensive devices.[2]

This result also shows that when Mosaic just stores a small part of the working set on fast storage, this already drastically increases overall throughput. The cost of this increase is very low if placing data at column granularity. A budget increase of 12% (from 310 ct to 350 ct) speeds up the benchmark by over

[2]To keep the number of variants for table granularity measurements manageable, the four smallest tables always reside on NVMe SSD. The cheapest measurement at column granularity is therefore cheaper than the cheapest measurement at table granularity.

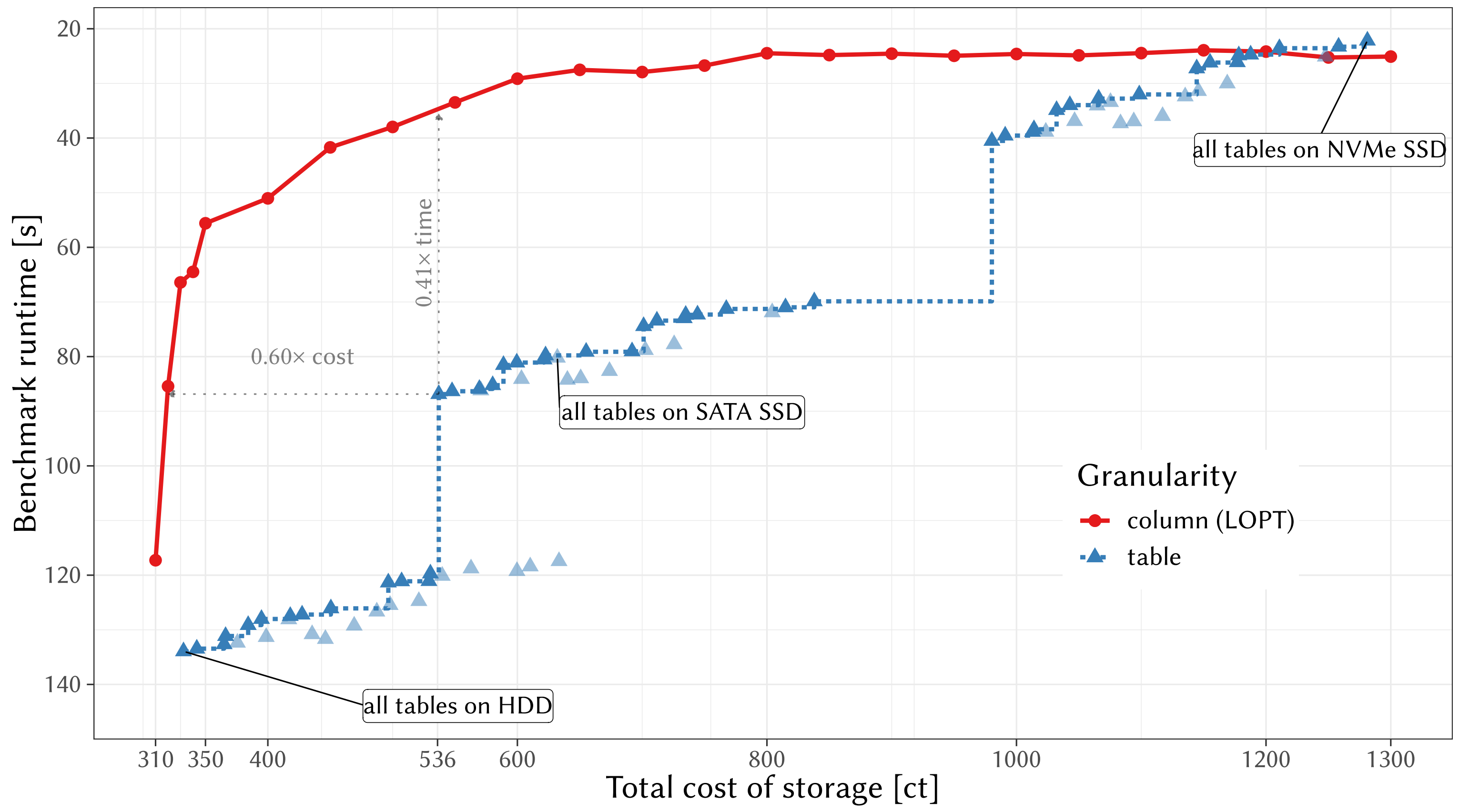


Figure 2.6: Benchmark runtime for TPC-H (SF 30) with column-granular placement using the LOPT strategy compared to all placement permutations of the four largest tables at table granularity. The dashed line indicates the Pareto optimum for table placement. Configurations that are *not* pareto optimal are marked semi transparent. The dotted arrows show that Mosaic using LOPT placement offers the same performance at a lower budget or faster runtime at the same budget.

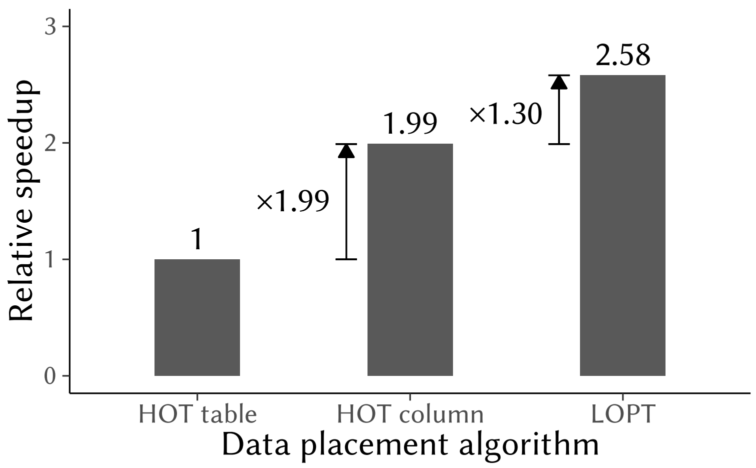


Figure 2.7: Comparison of placement algorithms normalized to HOT table. Each bar is the sum of 56 runs of the TPC-H benchmark (SF 30). Each run uses a distinct device configuration.

100% (from 117 s to 55.6 s). At higher costs, where most data fits on the fastest device, Mosaic cannot gain much advantage from distributing data between devices (as seen in Figure 2.5). It thus has equal or — if the model's throughput estimates are inaccurate — slightly worse performance than if the user placed all data on the fastest device.

Mosaic also visualizes a law of diminishing returns. With a budget of 600 ct, Mosaic is already within 14% of the best performance that requires twice the budget, i. e., 1300 ct. The optimal table granular placement at 600 ct results in a benchmark that takes 3.7 times as long as at maximum budget.

2.5.3 Comparison of Placement Strategies

In this experiment, we compare Mosaic's three placement strategies, LOPT, HOT table, and HOT column, against each other. How much the placement strategies differ in performance depends on the storage configuration. If, for example, only one storage device is available, all strategies place data identically. To obtain a representative comparison of the strategies, we compare performance across a range of device configurations. For each of the four devices in Table 2.1, we fix its proportion of the total storage to a value between 0% and 100% of the data set size, in 20% steps. We then recursively fix the values of the remaining three devices in the same way. We only consider configurations whose storage adds up to 100% of the data set size. Mosaic thus runs the benchmark for 56 configurations for each strategy. We then add the runtimes of those benchmarks.

Figure 2.7 shows the results for the TPC-H benchmark. It shows the speedup of the placement strategies over the baseline, HOT table. HOT table is worse than the other two strategies, as the TPC-H data set has many large but cold columns on otherwise hot tables. At table granularity, these columns waste valuable storage space that hot columns of different tables could have used. Data placement at column granularity provides a 99% speedup, confirming our findings in Section 2.5.2. LOPT is ≈ 25% faster still than HOT column, showing the advantage of a tierless device pool over a tiered architecture even with just four devices. Because throughput gaps between SATA SSD, RAID 5, and HDD are small, LOPT can distribute columns often accessed together between those devices. HOT column places as much data as possible on SATA SSD, preferring it over HDD and RAID 5, leaving optimization potential on the table. We, therefore, chose LOPT as Mosaic's default strategy.

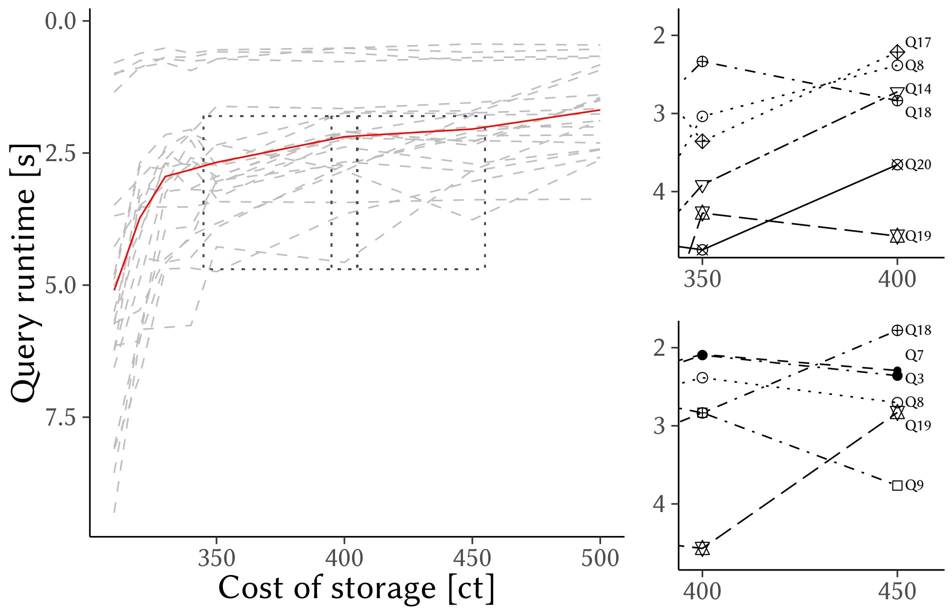
2.5.4 Per-Query Analysis of LOPT

While average query performance increases monotonically with budget, there are 'loser queries' that either do not become faster or even degrade with increasing budget, since LOPT's only goal is to minimize the sum of all query runtimes. Figure 2.8a shows per-query performance at varying budgets. The two zoomed-in sections show the biggest 'winner' and 'loser' queries at 400 and 450 cents. At 400 cents (upper cutout), Q18 and Q19 are slower than at 350 cents, as LOPT moves the columns of `lineitem` read by both queries from RAID back to HDD. This makes space for four columns read by the other queries, reducing overall runtime. When the budget increases to 500 cents (lower cutout), the pattern reverses: LOPT moves `lineitem`'s primary key back to RAID from SATA SSD. This slightly slows down most queries reading it but allows LOPT to move Q18's and Q19's previously demoted columns back to SATA SSD.

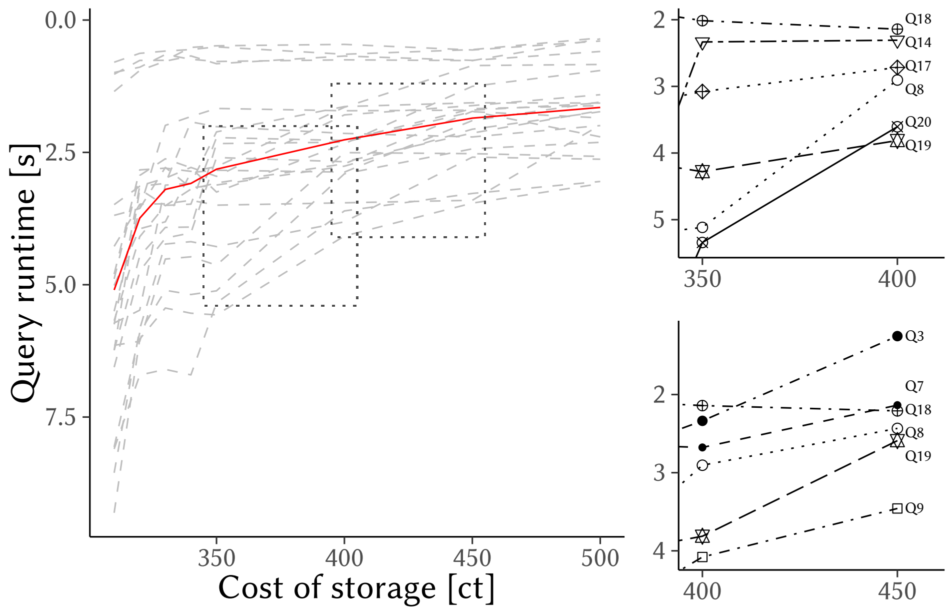
The user may deem such regressions unacceptable, i. e., they require guarantees that some specific subset — or all queries — do not slow down after a system upgrade. In this case, Mosaic supports the addition of a new constraint to LOPT, setting a query's (or all queries') current execution time as an upper bound. Figure 2.8b shows LOPT's performance with this constraint. While the throughput is 10.1% worse on average, there are no more unpredictable performance regressions.

2.5.5 Placement Calculation Cost of LOPT

As stated in Section 2.4.5, LOPT is NP-hard. Heuristics of modern MIP solvers, however, keep computation time at a reasonable level even for larger problems. We first evaluate LOPT's placement calculation time for smaller sized work-



(a) Default LOPT. The zoomed-in sections show the biggest winner and loser queries at budgets of 400 and 450 cents.



(b) Modified LOPT. Placement is constrained so that no query may become slower. The zoomed-in sections show the same queries as (a).

Figure 2.8: Runtime per query for two different LOPT variants (TPC-H SF 30). The solid red line shows average runtime, the dashed lines show runtime of each of TPC-H's 22 queries.

Table 2.3: LOPT search time for a placement solution for four devices with three different workloads. It shows the time to find a solution that is within 5% or 1% of the theoretical optimum, or is optimal.

		table scans		time [s]		
	#queries	total	distinct	$< 5\%$	$< 1\%$	opt
TPC-H	22	86	58	< 1	< 1	< 1
JOB	113	977	62	< 1	< 1	< 1
TPC-DS	67	492	193	< 1	≈ 4	≈ 64

loads. The results are shown in Table 2.3. The JOB workload by Leis et al. [88] benchmarks cardinality estimators with queries that join many tables. Many of its table scans only touch primary and foreign keys. It thus has only a few more *distinct* table scans than TPC-H. In both cases, LOPT finds the optimal solution effectively instantaneous for arbitrary data set sizes. TPC-DS has over 3 times as many distinct table scans as TPC-H. LOPT's performance with TPC-DS is acceptable, but at $\approx$ 1 minute for the optimal solution it is considerably worse.

We now move on to progressively larger workloads, to see how LOPT scales with more devices, tables, and queries. We load multiple independent instances of TPC-H, multiplying the number of tables and queries by up to 80 times (resulting in up to 1760 queries on 640 tables with 4880 columns) and simulate each device up to 20 times (up to 80 in total). Note that this is an adverse workload, since as each column is accessed by 22 queries at most, there is no fast way for Gurobi to prune the solution space, i. e., all TPC-H instances are 'warm'. Figure 2.9 shows how long Mosaic takes to calculate a placement within 5% of the theoretical lower bound for all permutations. The worst case is $\approx$ 52 minutes for 80 devices and 640 tables. Cases that are realistic for a single node (i. e., $\leq$ 10 devices) take less than 8 minutes.

Being NP-hard, LOPT has its limits. With 1200 TPC-H instances (26400 queries, 10800 tables, 73200 columns) on eight devices, computing a solution within 5% of the lower bound takes $\approx$ 21.5 hours. This can be remedied with sampling, i. e., having LOPT only consider a subset of all table scans. Figure 2.10 shows the impact of sampling with 400 TPC-H instances (8800 queries, 3200 tables, 24400 columns) on eight devices. Since all columns are always 'warm' in this adverse workload, each discarded table scan removes valuable information. Even here, sampling is still beneficial. If a predicted slowdown of $\approx$ 9% is acceptable, it is possible to sample 60% of the table scans, thus reducing the placement calculation time by 44%, from 141 to 80 minutes.

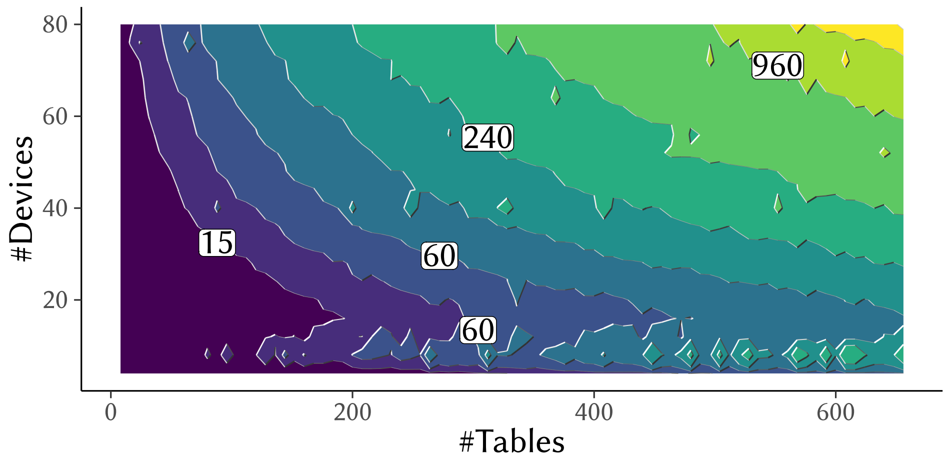


Figure 2.9: Computation time in seconds for a solution within 5% of the lower bound. The z-axis is $\log_2$ scale, i. e., time doubles with each contour step.

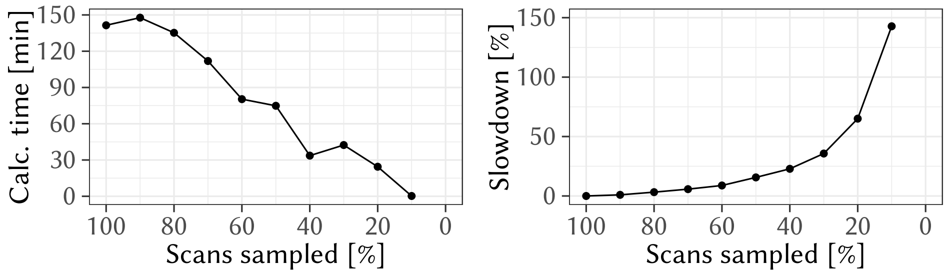


Figure 2.10: Left: Impact of sampling on placement calculation time. Right: Impact of sampling on predicted runtime performance. 400 TPC-H SF 30 instances, 8 devices.

2.5.6 Capacity Mode vs. Budget Mode

Figure 2.11 compares Mosaic's *capacity* mode (-▲-) against its *budget* mode (-●-). For *budget* mode, we repeat the measurement of Section 2.5.2. For *capacity* mode, we use the method of the experiment in Section 2.5.3 to generate 56 device configurations and use the LOPT strategy for placement. Each configuration (▲) could have been chosen by a system administrator using educated guesses. Because Mosaic uses the LOPT strategy for both placement modes, we can now quantify the advantage of having Mosaic assisting in the purchase decision (-●-) over pre-purchasing hardware and only then letting Mosaic place data (▲).

16 out of 56 capacity configurations are Pareto-optimal (---). For TPC-H, there is a probability of ≈ 29% of a system administrator picking a desirable storage device configuration by guessing which could be the best. But even if

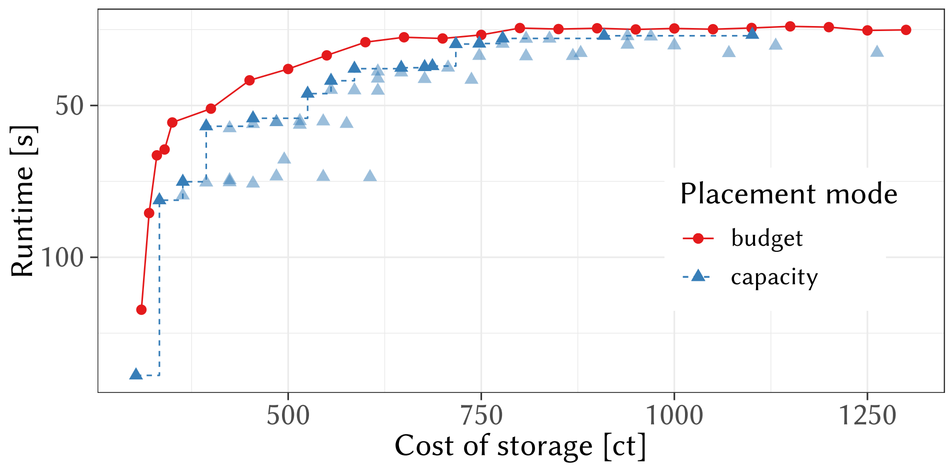


Figure 2.11: Comparison of placement modes for the TPC-H benchmark (SF 30) using LOPT. In budget mode, Mosaic chooses its storage devices for a budget. In capacity mode, Mosaic places data on 56 predefined device configurations.

they pick a Pareto-optimal configuration, its corresponding *budget* counterpart dominates it. On a price-point-per-price-point comparison, the *budget* approach is ≈ 26% faster than the Pareto optimum of the *capacity* mode.

2.5.7 Prediction Accuracy

In this section, we evaluate whether predictions made by Mosaic's table scan model are accurate. For this benchmark, we use the LOPT placement strategy in budget mode. Mosaic predicts the runtime for a range of maximum budgets, both for TPC-H and TPC-DS, at scale factors of 30 and 100. It then places data according to the budget constraint and runs the benchmark. We then compare Mosaic's predicted benchmark runtime with the actual runtime.

Figure 2.12 shows the predicted runtime for a budget (---) and the measured time after Mosaic placed the data (—). For TPC-H, the absolute mean error between predicted and measured time across all scale factors and budgets is only 4.1%. For TPC-DS, it is 19.0%, with higher budgets having a higher error than lower budgets. The reason is that TPC-DS, is CPU-bound on the evaluation system, when Mosaic stores most of the data on the NVMe SSD. At lower budgets, the slower but cheaper devices hide the CPU overhead.

While the prediction is accurate when running an I/O-dominated workload or using slow devices, the prediction becomes inaccurate when the workload becomes CPU-bound. This is because Mosaic cannot predict the throughput of the DBMS's execution engine. While slower devices shadow the execution

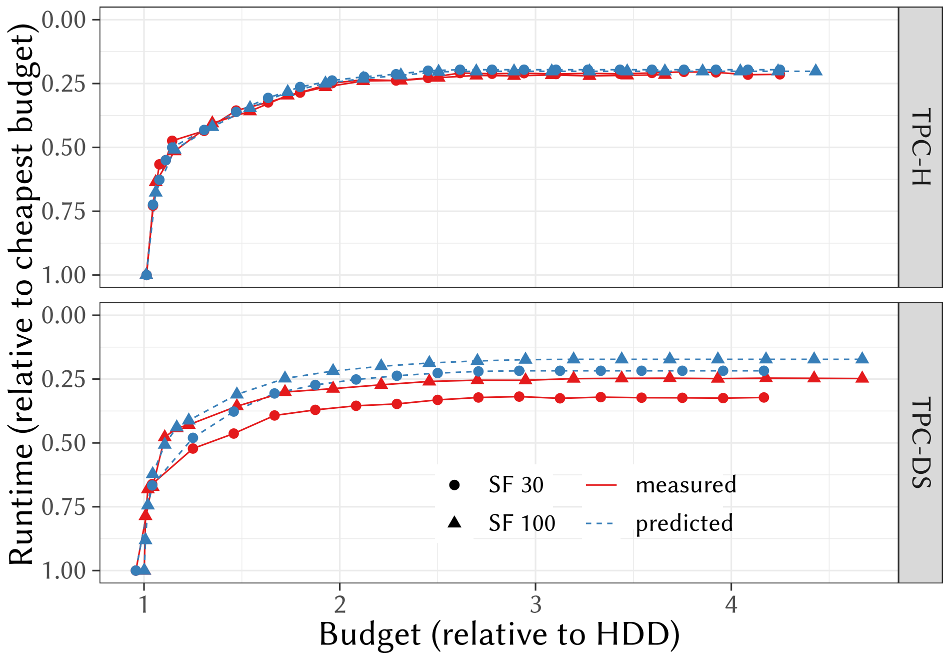


Figure 2.12: Predicted vs. actual performance for the TPC-H and TPC-DS benchmarks (SF 30 and 100).

overhead, faster devices expose it. The experiment, however, shows that Mosaic is useful, even in CPU-heavy workloads for the following reasons: (1) $\approx 20\%$ error is still acceptable when the status quo is having no prediction; (2) Mosaic brings the most benefit when users have limited budget and thus having most of the data on fast devices is not an option. Here, Mosaic is quite accurate, even for TPC-DS; (3) Mosaic correctly predicts the shape of the graph, showing where a small investment makes a huge return and when diminishing returns kick in. Mosaic's purchase recommendations are still valid, and it finds the fastest configuration for the given cost. It just does not take the bottleneck of the execution engine into account. We thus argue that even for CPU-bound benchmarks, Mosaic still offers great benefits over storage engines without predictive capabilities.

2.5.8 Impact of Workload

To evaluate how Mosaic adapts to different workloads, we generate 1000 workloads with 10 random TPC-DS queries each. We pick four of those workloads that deviate the most from the shape in Figure 2.6 and compare their performance at different budgets. We have chosen them for a number of characteristics,

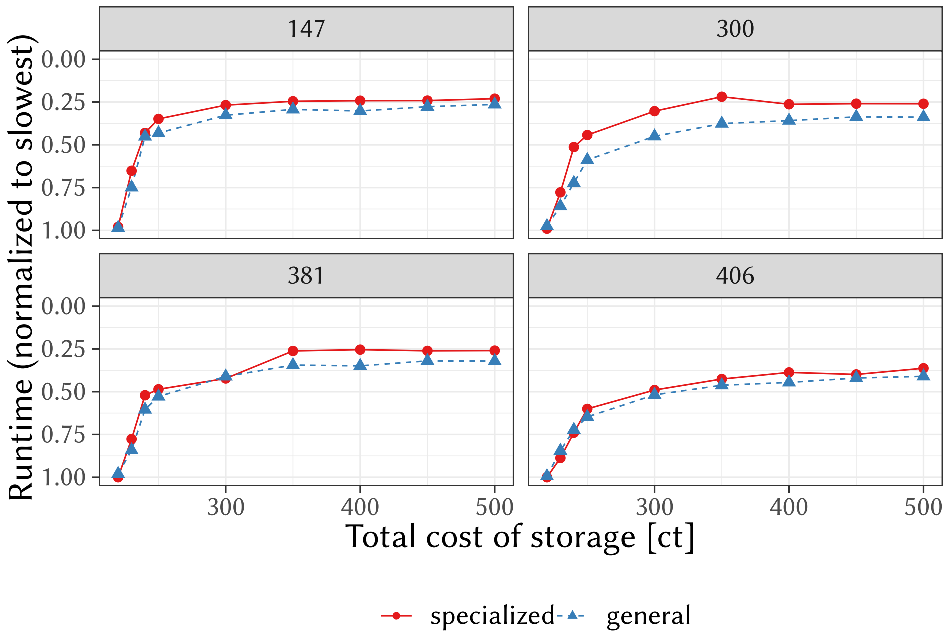


Figure 2.13: Performance of 4 out of 1000 TPC-DS workloads at different budgets for data placed specifically for the workload and for data placed for the TPC-DS benchmark in general.

for instance workload 147 profits above average at low budgets while workloads 300 and 406 profit at higher budgets. Workload 381 has a big performance jump at medium budgets. The performance of all 1000 workloads increases by more than 100% at 500 ct. Figure 2.13 shows Mosaic's performance for the four chosen workloads with data placed specifically for the workload (-●-) and data placed for the original TPC-DS workload (-▲-), which is a superset of the four workloads.

The workloads profit from a placement specifically tailored to them. Since the working set is smaller, Mosaic can move a larger percentage to devices with higher throughput. Each workload, however, also sees improvements with the generic TPC-DS placement. This experiment shows that — while it is beneficial to give Mosaic a trace that represents the actual workload as closely as possible — performance is still acceptable if the trace is a superset. Our earlier evaluations show that Mosaic finds a placement quickly even for large traces. A superset can thus be chosen (e. g., all queries run in the last month) without hurting performance too much.

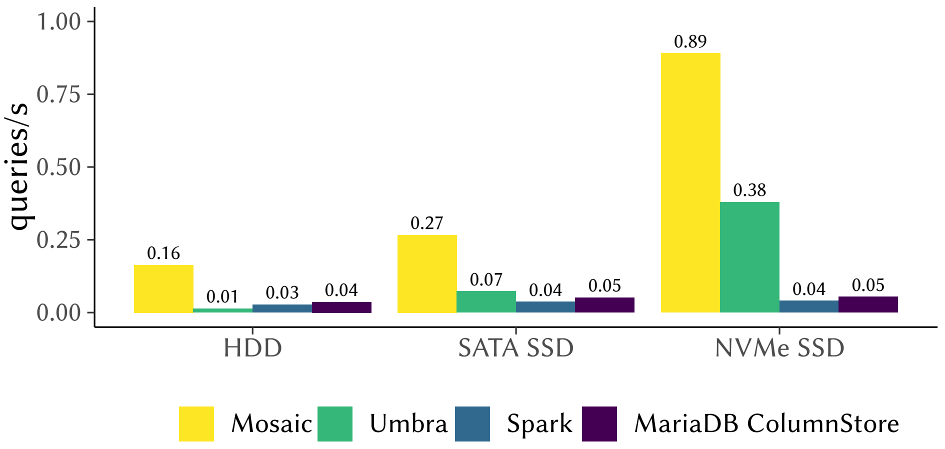


Figure 2.14: Mosaic's TPC-H throughput (SF 30) compared to Umbra and two Big Data query engines, all 22 queries distributed uniformly.

2.5.9 Mosaic vs. Big Data Query Engines

In this section, we compare the performance of Mosaic against Spark and MariaDB ColumnStore as representatives of big data query engines. These OLAP systems are optimized to read data in column-major format. Both claim to be competitive on a single node. We also compare Mosaic against vanilla Umbra as a representative of conventional RDBMS. Umbra buffers data into main memory when first accessed. Consequently, Umbra is an order of magnitude faster when data is already buffered. To benchmark I/O speed, we clear Umbra's buffer between queries.

Figure 2.14 shows the throughput for TPC-H. For all three configurations, we store the data set on just one device. Umbra is optimized for in-memory data sets and SSD. Its performance degrades on devices not suited for random I/O, but it is slower than Mosaic even on NVMe SSD, as Mosaic's compression results in a higher effective throughput. Umbra's table scans furthermore read all columns while Mosaic only reads queried columns. Spark and MariaDB ColumnStore are slower by an order of magnitude. While Umbra and Mosaic speed up when moving the data set to an NVMe SSD, Spark and MariaDB only become marginally faster.

When reading from disk, Spark has a similar throughput to Umbra with Mosaic. It is optimized for distributed workloads and introduces abstraction layers required to make it compatible to its many supported file formats. This, however, results in the computation time shadowing the I/O time when running on a single node. Query 7, for example, takes 42 seconds at SF 30, even with all data on NVMe SSD. Even if we ignore four seconds of startup time, Umbra with

Table 2.4: EBS types and their performance characteristics compared to a consumer-grade SSD.

Type	IOPS	Tpt. [MB/s]	Cost [$/(TB · mo)]	Use case
io2	256000	4000	9824.00	Transactional Wrklds.
gp3	16000	1000	146.92	High-throughput SSD
gp2	16000	250	102.40	Warm SSD storage
st1	500	500	46.08	Warm HDD storage
sc1	250	250	15.36	Cold HDD storage
Samsung 980 Pro	$\approx 10^6$	5000	3.75	General Purpose

Mosaic is ≈ 30 times faster at 1.2 seconds. At SF 100, it is still 10 times faster than Spark at SF 30. Spark spends more time on garbage collection (≈ 2 seconds) than Umbra takes for the whole query. On a single node, there is therefore not much to be gained by integrating Mosaic's smart data placement into big data query engines. Mosaic therefore has an important use case for single node systems with big data sets.

2.6 Mosaic in the Cloud

While we designed Mosaic primarily for on-premise systems where the system administrator has complete control over which storage hardware to purchase and install, we can also apply Mosaic to cloud installations. Cloud providers offer different kinds of attachable storage devices which provide different performance characteristics at different prices. For example, Amazon Web Services (AWS) offers Elastic Block Store (EBS) to store data persistently, exposing a standard block device interface. Table 2.4 shows the different types of EBS storage Amazon offers[3]. This offering is further complicated by the distinction between *burst* and *base* throughput for gp2, `st1`, and `sc1` volumes where utilizing all IOPS (for gp2) or maximizing bandwidth (for `st1` and `sc1`) depletes a bucket of refilling credits making the performance of those types more challenging to model. Furthermore, `io2` volumes offer a (re-)configurable amount of IOPS, making them cheaper if less random access is required.

Users pay a hefty premium for cloud storage: As Table 2.4 shows, a 1 TB SSD with similar throughput and four times the IOPS than the fastest `io2` instance can currently be bought for just $75 while lasting 600 days at an assumed write throughput of 1 TB per day making it cheaper by orders of magnitude[4]. For

[3]Source: `https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/ebs-volume-types.html`, accessed: 09.05.2023

[4]To ensure similar availability to AWS's offerings, it would be necessary to operate multiple

the user, the price premium might still be worth it as cloud storage has the advantage of supporting dynamically scaling attached volumes to the user's need, switching between different device types as required without paying a high up-front price. However, a system administrator faces the same problem as with on-premise systems: It is not apparent what the optimal configuration of EBS volumes for a given workload is. The number of EBS volume types and their tuning knobs opens up a large configuration space. In contrast to on-premise systems, cloud deployments even allow the user to reconfigure attached EBS volumes (i. e., the storage devices) with the click of a button, allowing for more flexible data placement without upfront costs.

Mosaic should thus be a good fit for cloud-native database systems: As cloud storage is more expensive than local storage, Mosaic can save more money for cloud systems than on-premise systems. Furthermore, as the system administrator can easily reconfigure EBS volumes, they can more quickly react to Mosaic's data placement suggestions by dynamically resizing attached EBS devices.

In follow-up work, Kinoshita et al. [86] evaluate Mosaic in such a cloud context, comparing placing data with Mosaic on `sc1`, `st1`, and gp2 volumes against using Linux's application-agnostic `bcache` utiltity [12] to cache hot data on gp2 volumes while keeping cold data on HDD-based EBS types. Here, Mosaic was able to outperform `bcache` by an average of 3.04× for TPC-H and 3.35× for TPC-DS for the same budget as Mosaic recognized cold data and stored it on the cheap `sc1` volume type while using gp2 for hot data. Furthermore, Mosaic could mitigate I/O credit exhaustion of the fast `st1` HDD volume type by partially moving data to slower HDD volumes and the gp2 volume type without increasing the budget. In contrast, `bcache` could not adapt to the dropping performance of `st1`.

2.7 Summary

This chapter introduces Mosaic, a storage engine optimized for scan-heavy workloads on RDBMS. It manages columnar data in a tierless device pool and supports pluggable data placement strategies. We evaluate three such strategies, including our linear programming placement strategy (LOPT), based on a model for predicting the throughput of table scans. In *capacity* mode, LOPT places data on previously purchased devices. In *budget* mode, LOPT predicts performance for a budget and makes purchase recommendations.

We evaluate Mosaic on two data sets to show the advantage of Mosaic's column-granular data placement over existing approaches of RDBMS and big

such SSDs in a RAID configuration. While this increases the operating cost, it is still far lower than the EBS offerings.

data query engines. Mosaic outperforms them by an order of magnitude and beats Umbra without Mosaic in OLAP queries when the working set does not fit into DRAM. We show the accuracy of Mosaic's prediction, which closely follows the Pareto-optimal price/performance curve. It is accurate for I/O-bound benchmarks. We finally show that Mosaic – while not primarily designed for cloud environments – can easily be deployed in the cloud and improves performance by a factor of 3 compared to a less sophisticated caching approach at similar budgets.

CHAPTER 3

Plush: A Persistent Log-Structured Hash-Table

Excerpts of this chapter have been published previously [150].

3.1 Introduction

Persistent memory (PMem) promised low latency, random access performance and throughput comparable to DRAM, as well as data persistency, while being a drop-in replacement of DRAM. PMem-optimized data structures could theoretically achieve the same throughput as their DRAM counterparts while offering granular persistency guarantees without needing additional write-ahead logging. However, in practice, Intel's Optane Persistent Memory Modules (DPCMM) perform considerably worse than DRAM. For example, their read latency is three times higher [131]. While its write latency is comparable to that of DRAM, its write bandwidth is lower by an order of magnitude, as shown in Table 3.1. Furthermore, PMem's internal smallest unit of access is 256-byte blocks, leading to significant read and write amplification for small accesses.

Prior work addresses PMem's high read latency, but often fails to fully leverage its low write latency and mitigate its low write bandwidth. Many

Table 3.1: Write access characteristics for different media

Medium	Latency	Bandwidth	Price	Access unit
DRAM	90 ns	80 GB/s	7.2 $/GB	cache line (64B)
PMem	130 ns	6.5 GB/s	4.0 $/GB	block (256B)
SSD	$\geq$ 1000000 ns	3.4 GB/s	0.3 $/GB	page (4KiB)

current PMem-optimized data structures, for example, follow a hybrid design, storing the recoverable part of their data (e. g., the inner nodes of a B-Tree) on faster DRAM [13, 24, 118]. This approach is excellent for lookups but it does not solve the issues of inserts as every insert has to store its payload on PMem to guarantee data persistency. State-of-the-art PMem-optimized data structures minimize the number of writes per insert to mitigate latency. However, because of PMem's low bandwidth, write amplification is just as much of a problem. PMem's internal 256-byte block structure further exacerbates this problem for small random writes: It internally amplifies each write to update a 256-byte block. This amplification leads to spurious writes saturating PMem's internal buffer, which drives up write latency and thus indirectly lowers throughput even if little actual payload is being stored [50].

Workloads consisting of small writes are a common use case in key-value stores [7]. Yahoo!, for example, states that their typical low latency workloads have more than 50% inserts [138]. We target this use case: We capitalize on PMem's low write latency by optimizing for small writes while mitigating its low write bandwidth by reducing write amplification. Our approach adapts the write amplification-reducing techniques of log-structured merge-trees (LSM trees) to PMem using hash tables.

LSM trees reduce write amplification for SSDs and HDDs [115] but do not leverage PMem's low latency for smaller writes. They buffer records on DRAM before merging them into consecutively larger sorted runs on SSD or HDD. This buffering reduces write amplification as LSM trees can fill each 4 KiB OS page before writing it back. They accept some overhead incurred by keeping their data sorted to ensure access patterns to SSDs/HDDs are favorable, i. e., by enforcing that merges are sequential reads and writes.

PMem-optimized LSM trees like NoveLSM [81] or SLM-DB [76] adapt existing LSM trees like LevelDB [44] and replace some components with PMem-optimized counterparts but keep their overall architecture. However, modern NVMe SSDs are often more attractive for workloads with large writes as they already offer half of PMem's throughput at a tenth of its cost. Therefore, porting LSM trees directly to PMem is hard to justify if one does not need the properties offered by sorting data (i. e., range queries) as their design does not take advantage of PMem's superior random write latency.

In contrast, we make use of the fact, that on PMem, it is unnecessary to generate large sequential runs by sorting records as it already reaches full bandwidth with just 256-byte writes. We instead propose gathering records in a list of unsorted 256-byte buckets, which we address through a hash table. Whenever such a bucket list is full, we propose re-hashing its contents and recursively merging them into a bigger hash table (the "next level"). We thus adapt the LSM tree's merging approach to a PMem-aware hash table. This was previously not workable, as the throughput of conventional block storage devices is too low for writes that small. Larger buckets to accommodate 4 KiB pages were also infeasible as search time would dominate for such large unsorted buckets. We call our approach **Persistent Log-strUctured haSH-table**, or **Plush** for short. In all other aspects, Plush takes proven approaches by LSM trees but adapts them for a PMem-*native* data structure in two ways:

(1) Batch writes to PMem. Like an LSM tree, Plush gathers new data in DRAM before moving it to PMem in batches, minimizing write amplification so that PMem does not have to deal with spurious writes which would increase latency and lower throughput. However, it uses a hash table instead of a skip list. Plush allows a *configurable* amount of DRAM buffer. Established hybrid data structures, in contrast, cannot control their DRAM consumption as it grows with the record count. This puts a limit on record count before running out of DRAM. Plush has no such limit.

(2) Store large records out of place. In contrast to many other PMem data structures, Plush supports variable-length keys and values. It employs a similar approach to LSM trees like WiscKey [101], which stores values in a separate log that periodically collects garbage. This approach reduces write amplification, since Plush does not have to copy values when merging them into the next level.

In summary, our key contributions are:

1. We explore how approaches to reduce write amplification developed for LSM trees can be adapted to PMem with the help of hash tables.
2. We propose Plush, a write-optimized hash table for PMem with *bounded* DRAM usage and low write amplification.
3. We evaluate Plush with fixed and variable-length records and show that it outperforms state-of-the-art PMem data structures for inserts.

To our knowledge, Plush is the first PMem-optimized data structure to take this approach.

3.2 Background

Plush is a *hash table* for *persistent memory* inspired by *LSM trees*. This section introduces each aspect.

3.2.1 Persistent Memory

PMem positions itself as a drop-in replacement of DRAM as it has the same load and store interface: One allocates a chunk of it and treats the memory region just like any allocated memory on DRAM. There are, however, some particularities unique to PMem:

Performance. We measured PMem performance in earlier work [131]. We found that one can expect a 3.2× higher read latency for PMem compared to DRAM but similar write latencies. The write latencies are similar as a store does not have to reach the physical medium but just the CPU's ADR domain which already guarantees persistency [71], while reads have to go all the way to the physical medium. For throughput, PMem falls off more: 2.6× for reads and 7.5× for writes. Thus, a significant advantage of PMem over conventional block storage is its low latency. A persistent data structure should therefore capitalize on this low write latency.

Persistency barriers. Even on PMem, a system crash can lead to data loss: For example, the CPU might cache dirty data and delay flushing it to PMem. On a crash, all dirty cache lines are lost as caches are not persistent. Therefore, one has to use *persistency barriers* to ensure that the persisted data is always in a *consistent* state. A persistency barrier is a `clwb`, `clflush`, or a non-temporal write followed by an `sfence`. Flushing the cache line guarantees that the data reaches PMem. The store fence forbids the CPU to re-order any writes before or after. Such a barrier is expensive as it blocks until the CPU evicts the cache line to its ADR domain and the store fence prevents the CPU from concealing this stall by re-ordering other stores. One should therefore use as few persistency barriers as possible. If a barrier is required, one should ensure that the whole flushed cache line consists of payload to keep write amplification low. Intel's extended ADR (eADR), available since Ice Lake, solves this issue by including the CPU cache in the persistency domain, obsoleting persistency barriers. Since eADR needs a special power supply and not all CPUs supporting PMem support eADR, persistency barriers are still required for backward compatibility.

Torn writes. The system might crash while flushing data to PMem, resulting in a *torn write.* PMem guarantees that writes up to 8 bytes are atomic, larger writes might only be persisted partially after a crash. Torn writes are usually prevented by first writing a record to PMem followed by a persistency barrier. Afterward, this record can be "armed" by atomically writing an at most 8-byte header containing a `valid` bit followed by another persistence barrier. Having to use two persistence barriers makes this approach expensive. We have previously researched and tested approaches requiring only one persistence barrier [131], which we also employ in Plush.

256-byte blocks. Internally, PMem operates on 256-byte blocks, just like CPUs use 64-byte cache lines. The same principles apply: To reduce write amplification, programs should write and read data in 256-byte blocks. When this is not feasible, writes should be sequential so that PMem's write combining buffer can merge multiple writes which also enhances PMem's limited write endurance [65].

3.2.2 LSM Trees

Write amplification is an issue with data structures operating on background storage. Since the smallest unit of access is often larger than the record to be stored (256 B for PMem, 4 KiB for HDD/SSD), it is desirable to batch multiple writes, which reduces write amplification. LSM trees, therefore, buffer new records in DRAM. The LSM tree recursively merges the DRAM buffer into consecutively larger layers of background storage whenever it reaches a size limit. LSM trees keep the records sorted, usually using a skip list on the DRAM layer to make this merge process efficient. A k-way merge then comprises k sequential reads and one sequential write. This merge fits the access patterns of HDDs and SSDs as they benefit from sequential access, but keeping data sorted is expensive: Inserts cannot happen in constant time, and skip lists offer poor cache locality and can suffer from write contention. If, however, records are only stored on DRAM and PMem, there is no performance benefit for keeping them sorted: PMem has exceptional random write latency as long as writes are grouped into 256-byte blocks. Existing approaches to adapting LSM trees for PMem do not leverage this advantage but replace or improve just a few core components of already established LSM trees. NoveLSM, for example, adds persistent skip lists and mutable memtables [81]. SLM-DB only employs a single level and additionally keeps a persistent B-Tree index [76].

3.2.3 Hashing on PMem

Most bleeding-edge PMem-based hash tables use extendible hashing [38] or a variant thereof. Extendible hashing splits the hash table into a set of fixed-size buckets and a hash-addressable directory whose entries point to those buckets. The buckets store the actual records, consisting of a key and its value. To accommodate skew, n directory entries, with $n = 2^k, k \geq 0$ may point to the same bucket. That way, underutilized directory entries do not need their own (nearly empty) bucket taking up unnecessary space. Whenever a bucket is full, it is split into two buckets. All records of the old bucket are re-hashed with one additional bit of the hash function discriminating in which new bucket they belong. Afterward, $n/2$ directory entries of those that pointed to the old bucket point to each of the two new entries. If a bucket was already pointed to by just one directory entry before the split, we cannot discriminate further. In that case, the whole directory is doubled. Afterward, every bucket is pointed to by at least two directory entries and the bucket can be split. This is called a *structural modification operation (SMO)* which is very expensive and hard to do concurrently and with consistency guarantees.

Modern PMem-based hash tables like CCEH [109] or Dash [100] group multiple buckets into a segment to better optimize for PMem block size. They split a segment when *any* of its buckets is full. Therefore, hash tables take great care to improve the segment load factor to reduce the number of splits and SMOs arising from them. Level hashing employs a second level with standby buckets [172], Dash uses stash buckets.

3.3 Overarching Design

Plush combines the highlights of LSM Trees with the highlights of hash tables. As PMem does not depend on sequential accesses as long as the accesses are grouped in 256-byte blocks, we propose doing away with sorting and replacing the LSM tree's layers with hash tables. We still keep a DRAM buffer to group the records into 256-byte blocks, reducing write amplification.

Plush also builds on the foundation of extendible hashing. Like CCEH and Dash, it groups multiple buckets per directory entry. However, when a directory entry would need to be split, Plush does not split in place, but merges its records into a hash table with a bigger directory a level below. We call this a migration. This leveling approach allows Plush to skip expensive SMOs altogether. Unlike CCEH and Dash, Plush can also insert a record into any bucket of a directory entry instead of a specific one determined by the record's hash. While this slows down lookups, it speeds up inserts, as the load factor is higher: A directory entry is only migrated if it is full, resulting in low write amplification. To speed up lookups, Plush uses positive bloom filters in the directory entries which indicate whether a specific bucket contains a record with the requested key.

Plush is, to our knowledge, the first PMem-optimized data structure combining LSM trees and hash tables in this way. Plush capitalizes on PMem's low random write latency for 256-byte blocks. It mitigates PMem's low write bandwidth by reducing its write amplification.

This approach helps Plush to achieve the following design goals:

Hybrid architecture. Plush uses a small but configurable amount of DRAM to increase throughput. Here, DRAM acts as a buffer whose size is configurable. In contrast, other hybrid PMem data structures cannot provide an upper bound for DRAM consumption.

Low write amplification. Plush avoids expensive random writes to PMem and instead groups data into 256-byte chunks in a DRAM buffer to write at once. It keeps a write-ahead log of not yet persisted records to still guarantee per-record persistency. Since Plush writes this log sequentially, PMem can use its write combining buffer for increased throughput. Reducing write amplification also conserves PMem's limited endurance.

Reduce persistency barriers without relaxing persistency guarantees. We tolerate small time windows where data is duplicated to ensure that Plush is always consistent. This concession allows us to reduce the number of persistency barriers. In the event of a crash, Plush amortizes data deduplication over runtime after recovery.

Concurrency without persistent locks. PMem-optimized hash tables often need locks stored on PMem for structural modification operations to reconstruct the current state during recovery. Since Plush is always in a consistent state, we can forgo such persistent locks. Inserts use fine-grained locking on DRAM, while lookups only use optimistic locking [16, 87].

Efficient bulk loading. Plush guarantees persistency with the help of a PMem write-ahead log. For bulk loading, relaxed persistency guarantees (i. e., manual checkpoints) are often sufficient. The user can temporarily turn off logging for increased throughput.

First-class support for variable-length records. In contrast to other PMem data structures, Plush supports both variable-length keys *and* values. Unlike some prior work like Dash, Plush always persists the payload itself, i. e., does *not* treat keys or values as pointers to data managed by a separate data structure like a write-ahead log. No separate write-ahead log is therefore required when using Plush.

3.4 Architecture

Figure 3.1 shows Plush's architecture. It consists of three components: A multi-leveled hash table stores the (fixed-size) records. A table of write-ahead recovery logs ensures that records that have not yet reached PMem are recoverable. An optional table of payload logs stores variable-length records.

3.4.1 Multi-leveled Hash Table

The core of Plush is its hash table. It stores all records. Figure 3.1 shows such a hash table with fanout 2.

Levels. The hash table consists of multiple levels, with the lowest level residing in DRAM. All higher levels are stored on PMem, with each level's directory multiplying in size by a configurable power of 2, the fanout, except for the first PMem level, which may have a smaller fanout. The user can thus adjust the DRAM consumption by varying the size of the DRAM level compared to the first

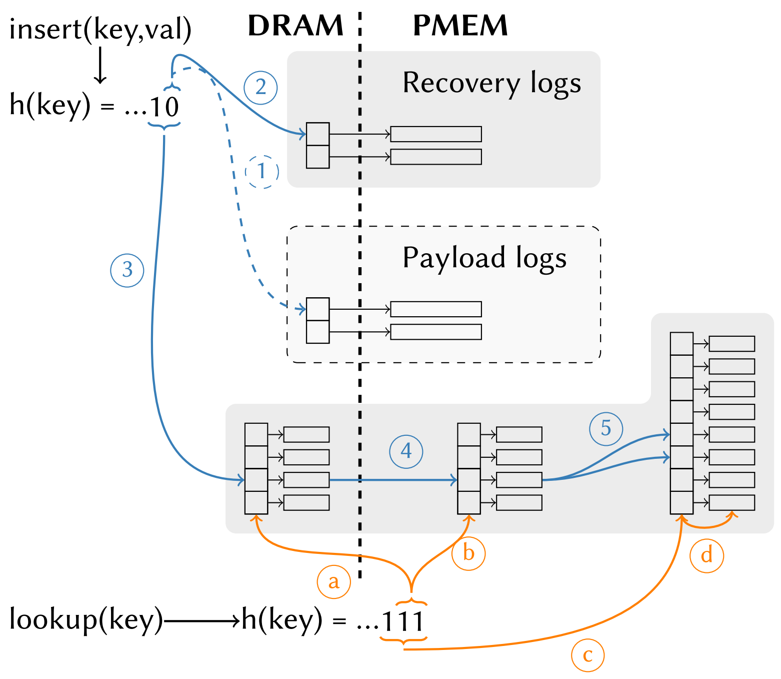


Figure 3.1: Plush's component overview with illustrated insert (1 - 5) and lookup algorithms (a - d). The colored lines represent steps in the algorithms, the black lines pointers of the data structure.

PMem level without modifying the fanout on PMem. Like extendible hashing, each level's hash table consists of a directory whose entries contain a bucket list of up to `fanout` buckets. A bucket holds up to 16 records consisting of 8-byte keys and values. It thus has a total size of 256 bytes which is the same size as a PMem block.

Migration. When all buckets for an entry are full, Plush re-hashes their records using $\log_2$(`fanout`) additional bits of the hash function. It then distributes the records onto `fanout` directory entries on the next level (4), recursively if necessary (5). Figure 3.2 illustrates the migration process. In the example, the directory entry for all keys whose hash ends on 1 is full and has to be migrated to the next level. We assume a fanout of 2, so each directory entry points to two buckets. Let us furthermore assume that the hash of all entry marked with 🟦 hash to 0 on the second-to-last place and all entries marked with 🟧 hash to 1.

When migrating, Plush appends records to the end of the last non-full bucket for the corresponding directory entry on the next level. If that bucket overflows, it allocates a new bucket (in the 🟦 case). Plush then clears the old buckets in the original level, making space for newly inserted records. Records thus slowly move to larger levels like in an LSM tree. Assuming a uniform hash function,

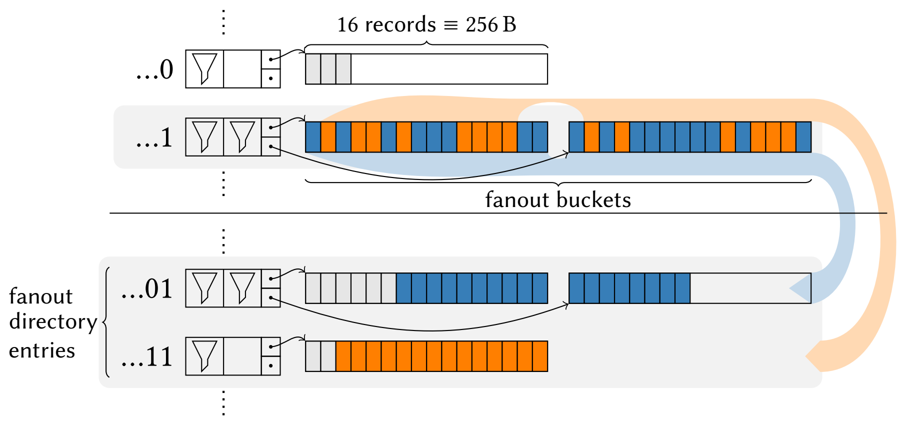


Figure 3.2: Migrating the buckets of a directory entry.

we expect `fanout` full buckets in level n to distribute evenly onto the `fanout` directory entries on level $n+1$ where each directory entry receives ≈ 1 bucket of new records. Migrating a bucket to the next level, therefore, results in `fanout` PMem block reads and one write in the best case (🟧). In most cases, however, there is not exactly one bucket's worth of entries to insert, or the current bucket already contains some elements from a previous migration forcing some records to spill into the next bucket (🟦). We, therefore, have to expect two block writes per migrated bucket on average. Since most records will be on the highest level, the average per-record write amplification grows linearly with the number of levels (and thus logarithmically with the record count). This, however, is also true for conventional PMem data structures. Hash tables, for example, have to deal with SMOs and segment splits, trees with (leaf-) node splits.

DRAM buffer. Plush always inserts new elements into DRAM (③). Buffering records in DRAM guarantees that Plush never stores single records on PMem by itself, which would amplify writes by 16 (updating a 256-byte block for each 16-byte record). The buffering approach is an advantage over pure PMem data structures which cannot buffer and combine writes in DRAM by design.

Lookups. Lookups search for the key by probing a filter in the directory entry at each level consecutively (ⓐ - ⓒ). Only on a filter hit is the actual bucket accessed and searched as well (ⓓ).

Figure 3.3 shows the layout of the hash table's directory and buckets. Each directory entry consists of:

Filter. Plush uses per-directory entry filters to efficiently check whether a bucket contains a key. In contrast to prior work like the FPTree using fingerprints [118], Plush uses a partitioned bloom filter [125, 137] where each partition bf_x with

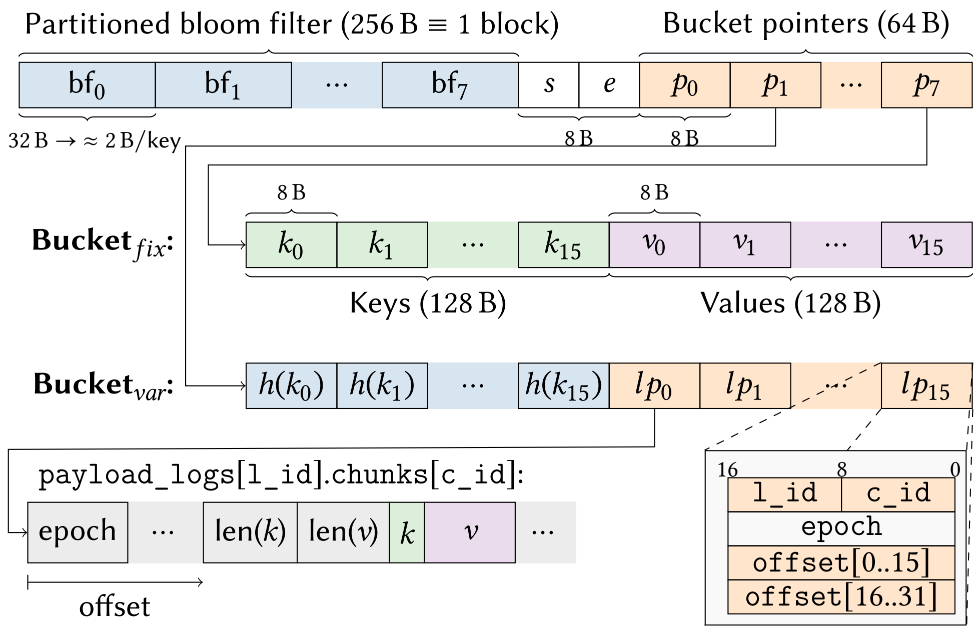


Figure 3.3: Layout of a directory entry, its buckets (for fixed and variable-length keys), and the payload log for fanout 8.

$0 \leq x <$ fanout covers the keys of one bucket. By forgoing fingerprints for bloom filters, we sacrifice the ability to find the offset of potential hits within the bucket upon a lookup (ⓐ) but achieve a far lower false-positive rate. Plush stores all keys $k_0 \dots k_{15}$ within a bucket sequentially, so that it can compare all of them with the search key with two AVX-512 SIMD instructions. Since the PMem block read latency dominates the bucket lookup, this does not lose us significant performance at the benefit of a far lower false-positive rate compared to fingerprinting. A negative lookup on a hash table level thus only has to load the bloom filter, which fits into a single PMem block. When doing inserts, we bulk insert multiple elements into the same directory entry, updating the filter in bulk. An insert of 16 elements thus only requires us to update the filter once (≡ 1 PMem block write).

Size and epoch. Since Plush fills buckets sequentially, a single size field s in the directory entry suffices to calculate into which bucket and position a new element has to be inserted. The recovery log (see Section 3.4.2) uses the epoch field e to determine which entries Plush has already migrated to a persistent hash table level on PMem. Plush increments it after every migration.

Bucket pointers. The directory entry stores an array of pointers p_x with $0 \leq x <$ fanout to the buckets containing the records. Plush updates the array whenever it allocates a new bucket. Plush distinguishes between two kinds of buckets: Buckets for fixed-size records (up to 8B keys and 8B values) called bucket$_{fix}$ and buckets for variable-length records called bucket$_{var}$. We discuss the second kind in Section 3.4.3. Fixed-size buckets are the size of a single PMem block and can thus store 16 records. Plush can optionally drop the bucket pointers and pre-allocate all buckets for all directory entries of the first k PMem levels at pre-defined offsets as an optional optimization. Pre-allocating buckets incurs space overhead, especially if the utilization of the hash table is low. On the flip side, it speeds up lookups and inserts as the pointer dereference (a PMem read) is replaced by an unconditional, pre-calculated jump. Since we expect all but the last level to be full, pre-allocating buckets for the upper levels will not impact space consumption in the long run but considerably speed up lookups and inserts. Section 3.6.5 evaluates the impact of this optimization.

Plush can also store 16-, 32-, and 64-byte keys and values in its buckets. This is disabled by default as it reduces per-level capacity and requires coarser locks for synchronization as C++ atomics cannot be larger than 8 bytes without incurring additional synchronization overhead.

3.4.2 Recovery Log

Plush guarantees that it persists records permanently when the insert method returns. Since Plush stores the newest entries on the DRAM part of its leveled hash table, it has to keep a persistent log of all entries that it has not yet migrated to a PMem hash table. Before it inserts a record into the hash table, it persists the record in such a log (②). Unfortunately, this is a "law of nature" we cannot get around: To guarantee persistency for each 16-byte record, we have to persist each record in the log with an expensive persistence barrier and high write amplification (since $16 \ll 256$). However, PMem's write combining buffer saves us: Since we write to each log sequentially, it can combine multiple writes into a larger write without weakening persistence guarantees. Plush's design thus prevents small random writes to PMem: Instead, it either batches writes (hash table bucket migration) or issues sequential writes (recovery logs) which the write combining buffer batches internally.

Plush employs multiple logs, which it partitions by the key's hash. Partitioning is a trade-off: The more logs Plush uses, the lower the write contention on each log. However, more logs also mean the write combining buffer will not be as effective since more random accesses hit PMem. Plush thus allows a configurable number of logs independent of the DRAM hash table size. It uses 64 logs by default. Since Plush partitions the logs by key, it can still ensure a global ordering of updates on the same key, as it persists updates to the same key in the same (sequential) log.

Figure 3.4 shows the layout of a log. It consists of a configurable number

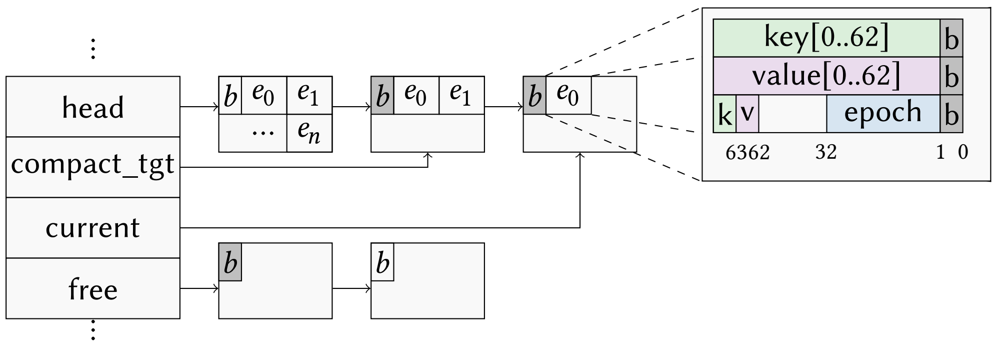


Figure 3.4: Layout of a recovery log. Each log consists of a set of chunks, split into in-use chunks and a free list.

of fixed-size chunks arranged in a linked list and a free list of chunks currently not in use. A `current` pointer points to the chunk that is currently being filled. Each log entry *e* consists of the key, value, and epoch counter. The epoch counter signifies during which epoch of the DRAM hash table's directory entry this record was inserted. Plush increments the DRAM epoch when it persists the corresponding buckets on PMem. It uses this information during log compaction and recovery to determine whether it has already moved the corresponding entry in the DRAM hash table to PMem. If it already moved that entry, it can discard the log entry during compaction respectively skip it during recovery.

The log has to guarantee persistency and prevent torn writes. Plush achieves both through the RAWL approach invented by Mnemosyne [151]: PMem guarantees atomicity for up to 8-byte integers. We thus distribute our data between 3 integers, each having 63 bits of payload and a flag bit *b*. Plush considers an entry valid if its flag bits are equal to the flag in the chunk's header. Since each integer was stored atomically, it is guaranteed that all data is persisted if all flags match. This approach needs just one persistence barrier after writing all three integers. Invalidating a chunk's entries is cheap: Plush flips the flag bit in the chunk header.

When the free list is empty, Plush compacts the oldest log chunk by iterating over its entries. When an entry is obsolete (i. e., its epoch is smaller than that of the DRAM hash table entry it belongs to), it skips the entry. Otherwise, it copies the entry to the chunk pointed at by `compact_tgt` preserving order. Last, Plush invalidates all chunk log entries by flipping the flag bit *b* in the chunk header and inserting it into the free list. Since Plush uses a fixed number of fixed-size chunks per log, the log space consumption is bounded.

Plush also profits from the fact that the capacity of its DRAM hash table is fixed. It sets the log's total capacity so that all logs together can hold more records than the DRAM hash table. This way, Plush will probably already have migrated the corresponding records of the oldest chunk's log entries to PMem by the time it compacts the chunk. In this case, compaction is just a sequential read without writes as all entries can be discarded. Plush additionally stores the highest epoch per chunk in DRAM. If that is lower than the epoch of all DRAM directory entries belonging to the chunk's log partition, it can even skip this read. By choosing the right log size, compaction is thus free. By default, Plush employs 64 logs with six chunks of 5 MiB. This configuration achieved free compaction at Plush's default DRAM directory size of 2^{16} in our tests.

Plush using a DRAM buffer with a separate PMem log has a distinct advantage over other PMem data structures. Unlike Plush, they trigger a PMem block write with high write amplification for every insert as they do not write to PMem sequentially and thus cannot make use of the write combining buffer.

3.4.3 Payload Log and Variable-Length Records

Plush also supports variable-length keys and values. It stores keys and values larger than 8 bytes out of place in a separate payload log. This separation keeps cache locality high, lookups inside buckets fast, and write amplification low. When storing records out of place, Plush does not have to copy them whenever it migrates their bucket to the next level. The payload log is similar to the recovery log except that log entries inside chunks can have arbitrary sizes, and each chunk has a log epoch counter (cf. Figure 3.3). Whenever Plush inserts a variable-length record, it first inserts the record into the payload log (①) followed by a persistence barrier. Only then is the record inserted into the recovery log and hash table. Plush uses special buckets for variable-length values. Instead of a key and value, they store a hash of the key and a pointer to the record's position in the payload logs. Plush pre-faults the recovery- and payload logs at startup to reduce page fault overhead at runtime [27].

3.5 Operations

In this section, we take a closer look at Plush's three operations: *upsert*, *lookup*, and *delete*. We then explore how Plush's design guarantees that it is always in a *consistent state* and ensures it *scales* with multiple threads. We finally explore how Plush handles recovery and how it can further speed up inserts for bulk loading.

```
 1 void upsert(KeyType key, ValType val) {
 2   // Find the correct directory entry
 3   uint64_t hash = hash(key);
 4   DirectoryEntry& entry = dram_directory[hash % dram_size];
 5
 6   lock(entry);
 7
 8   // Migrate, if full
 9   if (entry.size == fanout * 16) {
10     migrate(entry);
11     ++e.epoch;
12   }
13
14   // First persist to log ...
15   Log& log = logs[hash % log_num];
16   LogEntry& log_entry = log.entries[log.size++]; // Entries are
       on PMem, metadata (i.e, size field) is on DRAM.
17   log_entry.store(key, val, entry.epoch); //Using RAWL
18   persist(log_entry);
19
20   // ... then update the bucket
21   size_t bucket_idx = entry.size / 16;
22   size_t pos_in_bucket = entry.size % 16;
23   entry.buckets[bucket_idx].keys[pos_in_bucket] = key;
24   e.buckets[bucket_idx].vals[pos_in_bucket] = val;
25   ++entry.size;
26
27   unlock(entry);
28 }
```

Listing 3.1: Insertion algorithm for a fixed-size KV pair.

3.5.1 Upsert

Like an LSM tree, Plush is an append-only data structure. It, therefore, combines inserts and updates into a single *upsert* operation. Internally, an update is just an insert. The *lookup* operation has to ensure that it returns the latest value inserted for the specified key.

Listing 3.1 shows the upsert algorithm for a fixed-size key-value pair. Plush first determines the target directory entry in DRAM (3-4) and locks it (6). Note that this is a regular lock in DRAM. If there is no space for the record, it migrates the entries' contents to PMem (9-12). Note that the directory entries on the level to which Plush migrates the records only accept records from the current entry and are, therefore, indirectly locked. The migration thus only needs to consider concurrent *lookups*, but no inserts. Plush then inserts the record into the recovery log (15-18), persists it using RAWL (cf. Section 3.4.2), and finally inserts it into DRAM (23-27).

Listing 3.2 shows the migration process. First, Plush allocates a temporary buffer for re-hashing keys for the next level (2). It then iterates over the records of the old level from *back to front*, inserting them into the correct buffer (4-14). It skips records with keys that have already been inserted into the buffer before and thus prunes older versions of updated records. While this check has a runtime complexity of $\mathcal{O}(n^2)$ in theory, n is only 16 on average in practice leading to negligible constant overhead. Afterward, it inserts the contents of the buffers into the correct bucket on the next level (19-39), with a recursive migration (27) if necessary. The order in which it updates and persists the data is critical: Plush first persists new keys, values, and filters (33). Since it has not yet updated the size field, the new values are not reachable yet and Plush will just overwrite them after a crash and recovery. The filters, however, are not protected: If the system crashes after Plush has persisted the filters at least partially, they may yield false positives after a restart. Inconsistent filters are not an issue as they are probabilistic. Plush has to deal with false positives anyway, leading to slightly degraded performance for the partially migrated bucket after recovery at worst. Plush automatically fixes this as it resets the filters during migration. After it has updated and persisted the size (35-37), the migrated records are visible but are shadowed by their still valid old version on the previous level. If a crash occurs now, those values will be duplicated until they are migrated and merged into the next level. While this leads to some potential data overhead after recovery, Plush can migrate data holding no locks on PMem, keeping read- and write amplification low. In the last step, Plush marks the old entry as empty by zeroing and persisting its size and filters (41-43).

For variable-length records, migrations have to consider additional issues: As newly inserted records shadow old records with the same key, the payload log may contain old, no longer reachable entries. When a migration merges bucket entries, it does not purge their records from the payload log leading to two inconsistencies Plush has to consider:

1. Plush updated a record in the hash table, but its old value still lives in the log. Plush solves this through periodic garbage collection. Whenever it runs out of memory, it compacts the oldest chunk of each payload log to a new chunk. Plush checks whether each entry is still reachable, i. e., a lookup with the logged key will yield a pointer to the current log entry. If that is not the case, the entry is stale, and Plush garbage collects it. Otherwise, it moves the record and updates the log pointer. Like with the recovery logs, we assume that the oldest entries are stale most of the time so that Plush does not have to move many records on garbage collection.

```
void migrate(DirectoryEntry& e) {
  List<Record> rehashed[fanout];
  // Rehash all elements
  for (int idx = e.size - 1; idx >= 0; --idx) {
    size_t cur_bucket = idx / 16;
    size_t idx_in_cur_bkt = idx % 16;
    KeyType key = e.buckets[cur_bucket].keys[idx_in_cur_bkt];
    ValType val = e.buckets[cur_bucket].vals[idx_in_cur_bkt];
    List<Record> tgt = rehashed[(hash >> lvl_bits) % fanout];
    // Drop if duplicate - iterating from back to front ensures
     newest version is kept.
    if (!tgt.contains(key)) {
      tgt.append((key, val))
    }
  }

  // Insert rehashed elements into next level

  // For each directory entry in the next level ...
  for (int e_idx = 0; e_idx < fanout; ++e_idx) {
    DirectoryEntry tgt = get_entry(
      lvl + 1,
      index_of(e) * fanout + e_idx);

    // For each bucket in the directory entry ...
    for (int pos=0; pos < rehashed[e_idx].size; ++pos) {
      if (tgt.size == fanout * 16) { // Target full?
        migrate(tgt); // Recursive
      }
      update_filter(tgt, rehashed[e_idx][pos].key)
      insert(tgt, rehashed[e_idx][pos]);
    }
    // Nothing is guaranteed to be persisted yet!
    persist([tgt.keys, tgt.values, tgt.filters]);
    // If crashing here: Filter has false-positives
    tgt.size += elems_inserted;
    tgt.epoch = e.epoch;
    persist([tgt.size, tgt.epoch]);
    // If crashing here: Duplicates! -> will be cleaned up with
     the next migration
  }

  e.size = 0;
  e.filters = 0;
  persist([e.filters, e.size]); // Consistent again!
}
```

Listing 3.2: Migration algorithm for a directory entry.

2. During a migration, Plush might discover that a pointer in the hash table points to a log entry that no longer exists. Pointers dangle if the user updated a record, and Plush then garbage collected its old version but has not yet merged the old pointer on the higher hash table level. For this reason, the log pointer contains the epoch of the chunk it is pointing to. When Plush migrates a chunk, it increments its epoch. It thus can detect a dangling pointer by comparing the epochs and excluding them from migration to the next level.

In contrast to the recovery log, the payload log does not need to detect torn writes: Plush deems log entries valid if pointed at by a pointer. It only persists pointers after it flushed the log entry to PMem. As torn log entries were never valid, no pointer will point to them, and Plush will garbage collect them.

3.5.2 Lookup

Listing 3.3 shows the lookup algorithm. Since updates are just inserts, multiple versions of a record can co-exist on different levels of the hash table. Plush thus needs to ensure *lookup* returns the latest version of a record. To guarantee that, it searches all levels, beginning with DRAM (5-9) and ending with the highest PMem level (12-17), and buckets from back to front (37-41). It only checks a bucket if its bloom filter cannot rule out a hit (21-24). This operation is expensive as we can expect most data to live in the last PMem level, leading to negative lookups at all lower levels. Plush mitigates the issue by optionally storing the directory filters in DRAM instead of PMem. Storing filters in DRAM does not weaken Plush's persistency guarantees as it can recover filters from the records stored on PMem but lengthens recovery time. Sections 3.6.5 and 3.6.7 evaluate the trade-off. DRAM consumption stays configurable as the user can decide which level's filters Plush should store on DRAM.

3.5.3 Delete

Plush uses tombstone records for deletion. Deletes are thus just inserts where the value equals a pre-defined tombstone marker. During migrations, tombstone markers “cancel out” records with the same key, i. e., drops the record.

3.5.4 Recovery

When Plush crashes, it loses the contents of its DRAM hash table and has to rebuild it from the recovery logs. Plush iterates over every log and compares each log entry's epoch with the epoch of its target bucket on the lowest PMem

```
ValType lookup(KeyType key) {
  uint64_t hash = hash(key);

  // First, try DRAM
  DirectoryEntry& e = dram_directory[hash];
  ValType val = lookup_in_bucket(key);
  if (val) {
    return val;
  }

  // Iterate over PMem until hit or reached last level
  for (int lvl = 0; !val && lvl < pmem_levels; ++lvl) {
    DirectoryEntry& e = get_entry(lvl, hash);
    val = lookup_in_lvl(e, key);
  }
  return val;
}

ValType lookup_in_lvl(DirectoryEntry& e, KeyType key) {
RETRY:
  int b_id = e.filter.get_bucket(key);
  ValType result = nullptr;
  // If we have a filter hit, check bucket
  if (b_id != -1) {
    int epoch = e.epoch;
    result = lookup_in_bucket(e.buckets[b_id]);
    if (e.epoch != epoch) {
      // Someone inserted while we checked, retry
      goto RETRY;
    }
  }
  return result;
}

ValType lookup_in_bucket(Bucket& b) {
  // Just check every element - can be sped up using SIMD
   instructions
  for (int i = get_size_of(b) - 1; i >= 0; --i) {
    if (b.keys[i] == key) {
      return b.values[i];
    }
  }
  return nullptr;
}
```

Listing 3.3: Lookup algorithm for a fixed-size KV pair.

level. If the bucket's epoch is higher than that of the recovery log entry, Plush had already persisted that entry before the crash and skips it. Otherwise, Plush reinserts it into the DRAM hash table. As the recovery logs are partitioned, Plush can trivially recover them in parallel.

If Plush crashes during compaction of a log (i. e., while copying a log entry to a new chunk), there is no special case needed: Either it did not wholly persist the new version of the copied log entry and therefore recognizes the new value as torn, or both versions are valid. If both versions are valid, Plush will discard the older entry as it does a duplicate check when inserting recovered entries. Since all recovered entries fit into DRAM by definition (they would not need to be recovered otherwise), no migration is necessary during recovery. Plush's recovery is, therefore, idempotent. It can just restart a crashed recovery. The payload log for variable-length entries does not need a special recovery mechanism. An entry is, by definition, valid if a value in the hashtable is pointing to it. If Plush recovered no such pointer while iterating over the recovery logs, it will drop the entry in the next garbage collection run.

3.5.5 Concurrency

Upserts lock each bucket non-persistently. Assuming a fanout of 16, in 255 out of 256 cases, no migration is required, and the lock is thus just held for a short time. Since Plush uses hashing, inserts are ideally uniformly distributed over the directory. This uniform distribution is an advantage over LSM trees which have to keep data sorted, which leads to higher contention. For Plush, logging is the bottleneck, as multiple directory entries share the same log. Even though the logs use a lock-free atomic counter to assign slots, the CPU's cache coherency protocol adds some overhead. We have found that for our system with 24 cores the best trade-off between synchronization overhead and space consumption is 64 logs.

Lookups do not acquire any locks but use optimistic concurrency control. Therefore, values, keys, epoch, and size have to be atomic variables. Plush reads a bucket's epoch before searching for a record's key. After it finds the key, it re-reads and compares the epoch. If the epochs match, Plush can be sure that the bucket has not been migrated and overwritten during *lookup*. This design allows for multiple lookups and one insert to operate on a directory entry concurrently.

3.5.6 Crash Consistency

As long as the records are still on DRAM, crash consistency is guaranteed: Write-ahead logging ensures that entries are recoverable before Plush inserts them into DRAM. If another thread sees the key in DRAM, it is thus guaranteed to be recoverable. It is more complex during migration. Plush ensures that there is no inconsistent state being read by making migrated values visible after it persisted them and *only then* removing the old version at the previous level. This approach saves a lock and a persistency barrier but allows for a small time window where an entry is valid on both levels simultaneously. If the system crashes here, records are duplicated. This duplication is, however, not an issue. Since Plush's append-only architecture forces it to deal with and merge multiple records having the same key anyway, it will merge the duplicates when it migrates them to the next level. We thus trade off cheap consistency guarantees for the small risk of some temporary data duplication after a crash. Since *inserts* guarantee that Plush is always in a valid state and data is duplicated at worst, Plush does not need any additional consistency checks during *lookups*.

3.5.7 Bulk Loading

Some workloads do not need the strict persistency guarantees given by Plush but would benefit from increased throughput. Take bulk loading as an example: A user has to insert some existing large data set, but a (rare) crash is not world-ending as they can restart the bulk loading process. Here, it would be sufficient that the user can define checkpoints after which all records that a user has inserted before are guaranteed to be persisted (i. e., at the end of bulk loading) instead of having a per-insert persistency guarantee.

Plush can increase insert throughput by disabling write-ahead logging. When disabled, the persistency guarantees weaken as records living in the DRAM hash table are now lost on a crash. Plush mitigates this by allowing the user to create checkpoints manually. Checkpointing moves all records from the DRAM table to the first PMem level. It even supports mixing regular and relaxed persistency inserts: The user can decide upon each insertion if Plush should log the record. Section 3.6.5 evaluates bulk loading.

3.6 Evaluation

In this section, we compare Plush against other persistent trees and hash tables optimized for fixed- and variable-length records. We also evaluate the impact of different optimizations. Finally, we examine Plush's space utilization and recovery performance.

Our evaluation system is equipped with an Intel Xeon Gold 6212U CPU, with 24 physical (48 logical) cores and clocks at a 2.4 GHz base clock. The system has access to 192 GB (6×32 GB) DRAM and 768 GB (6×128 GB) of Intel's first-generation DCPMM DIMMs. It runs Ubuntu 20.04 LTS with kernel version

5.4.0. We configure the DCPMM in *AppDirect* mode and provision it in *fsdax* mode with an *ext4* filesystem. We manage via load and store instructions on an *mmap*'d memory regions and do not use any additional libraries.

We implement Plush in C++20 and make use of AVX-512 instructions. To our knowledge, all CPUs supporting PMem also support the AVX-512 instruction set. We compile Plush and all other data structures with GCC 11.1.0 with the `-O3` flag. If not otherwise mentioned, we use the PiBench benchmarking suite by Lersch et al. [92]. It was designed for persistent *tree* indexes but also supports all workloads applicable to Plush. Using a third-party benchmarking tool ensures our benchmarks do not accidentally favor Plush.

3.6.1 Plush Configuration

Let us first discuss how to best configure Plush depending on workload characteristics. Assume we have N records and a fanout f. Each directory entry thus holds up to $16f$ records. Further, assume that the hashing is not perfectly uniform, so migrating a batch of $16f$ records to the next level writes to two buckets on average (cf. Section 3.4.1). Thus a migration incurs $2 \cdot 16f$ bucket writes and $16f$ filter updates, so a media write amplification of 3 per record. Due to the write combining buffer, we get away with a write amplification of 1 for the log. Since log compaction is free (cf. Section 3.4.2), there is no additional amplification. We thus expect a per-record write amplification of $3 \cdot \log_f N + 1$. We thus want to choose the fanout as large as possible to minimize level count and thus write amplification while still having acceptable directory size spikes when Plush adds a new level. A high fanout also reduces read amplification as fewer levels have to be searched per lookup.

We choose $f = 16$ resulting in 2^{16} DRAM directory entries and the same amount of PMem directory entries at level 1. With this configuration, Plush has a directory sized 32 MiB, 544 MiB, 8.7 GiB, and 147 GiB at 1, 2, 3, and 4 levels holding 3.3×10^7, 3.0×10^8, 4.5×10^9, and 7.3×10^{10} records, respectively. Plush uses 64 recovery logs, each with six chunks of 5 MiB, the empirically determined sweet spot between minimizing contention (more logs are better) and maximizing write combining buffer usage (fewer logs are better).

Plush's behavior in this default configuration is demonstrated by Figure 3.5, which shows overall space consumption in relation to data set size broken down by bucket-, directory-, and log size for an insert workload of uniformly distributed 16-byte records (8-byte keys and values). Note the constant size of the log and growth of the directory whenever Plush creates a new PMem level. As stated earlier, the high fanout of 16 leads to big space consumption spikes for each new level, as Plush has to create a complete directory for the new, still nearly empty level. This spike is a downside of Plush's hash table approach

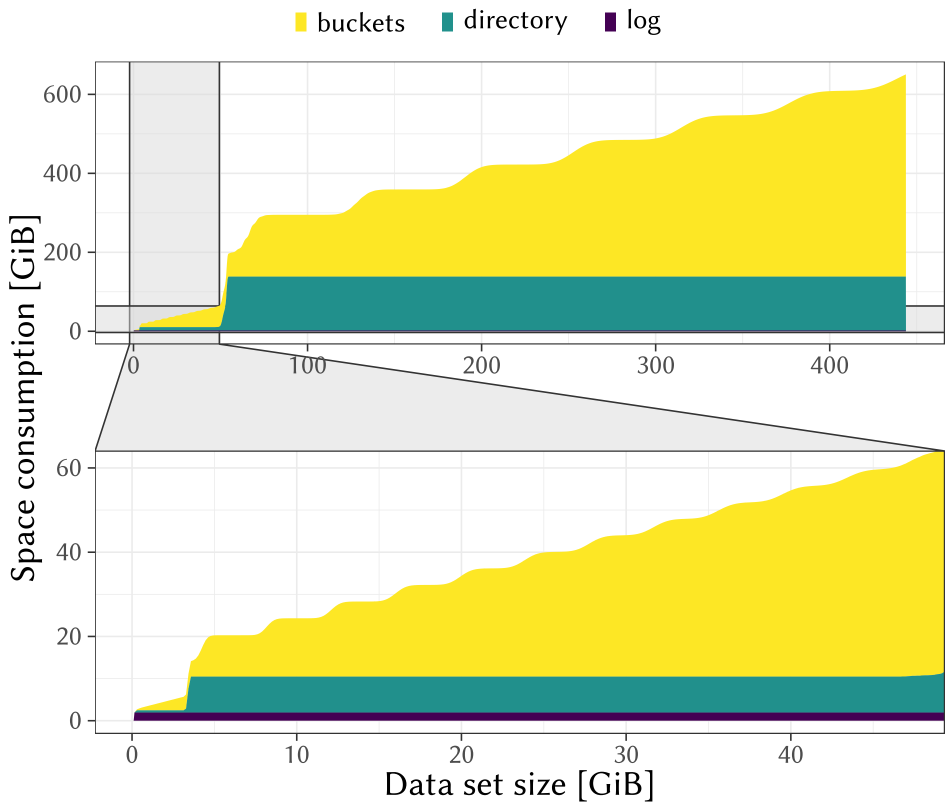


Figure 3.5: Space consumption relative to data set size as stacked area chart. Broken down by bucket-, directory-, and log size.

compared to traditional LSM trees, which do not need a directory. If storage space is a concern, the user should thus choose a lower fanout, leading to less severe spikes. However, note that the buckets' space consumption rises more well-behaved with the data set size. Although a directory entry can point to up to 16 buckets, Plush only creates buckets on demand (i. e., if the preceding bucket is already full). Therefore, no storage space is wasted. The stair-like pattern is an artifact of the workload: Since it is uniformly distributed and keys are hashed, all parts of the hash table fill at approximately the same rate. Thus, when all n-th buckets are full simultaneously, Plush creates the $n + 1$-th buckets in parallel, increasing the space consumption. This rise is followed by a plateau during which Plush (recursively, through all previous layers) fills the newly created buckets. Since the default fanout is 16, this pattern repeats 16 times until the level is full, and Plush has to create the next level.

Table 3.2: Investigated Data Structures.

Name	Type	Var-length records	DRAM	Range queries
Plush	LSM+ht	✓	$\mathcal{O}(1)$	✓[1]
μTree	tree	just values	$\mathcal{O}(n)$	✗
FPTree	tree	✗	$\mathcal{O}(n)$	✓
FAST+FAIR	tree	✗	$\mathcal{O}(n)$	✓
DPTree	tree	✗	$\mathcal{O}(n)$	✓
Dash	ht	just keys	-	✗
PmemKV	ht	✓	-	✗
Viper	ht	✓	$\mathcal{O}(n)$	✗
RocksDB	LSM	✓	$\mathcal{O}(1)$	✓
FASTER	ht+log	✓	$\mathcal{O}(1)$	✗

3.6.2 Comparison to Other Data Structures

We compare Plush against nine indexes listed in Table 3.2 which can be grouped into *hash tables* and *tree-like data structures*. Plush combines aspects of all those approaches allowing us to compare different trade-offs made by each approach. We also compare two versions of Plush. One, where the filters for the top two levels are stored in DRAM (cf. Section 3.5.2), and one, where all filters are stored on PMem. As the DRAM overhead of the filters is just 256 MiB, we treat this case as the default case if not otherwise mentioned. Dash [100] and PMemKV [123] with the cmap backend are PMem-only hash tables. Both support the same operations as Plush but do not use any DRAM. PMemKV also supports variable-length records, while Dash only supports (pointers to) variable-length keys. Viper [13] stores its records in PMem but keeps a hash table with fingerprints of all keys in DRAM, thus requiring large amounts of DRAM linearly growing with the number of records stored.

We also compare against persistent B-Trees: FPTree[2] [118], FAST+FAIR [62], DPTree [169], and μTree [24]. They store inner nodes (e. g., FPTree) or keys (e. g., μTree) on DRAM. As they sort their records at the cost of some insert throughput, most support range queries in contrast to the hash tables. The DRAM consumption of all trees and Viper grows with the record count. The DRAM consumption of all other hash tables and Plush is either constant in the record count or zero for the PMem-only indexes.

FASTER [22] is a log-based kv-store designed for SSD and gives weaker persistency guarantees. We place the persistent part of its log on PMem. We

[1]In range partitioning mode, see Section 3.6.6.

[2]We use SFU's open-source re-implementation: `https://github.com/sfu-dis/fptree`

configure FASTER to have the same amount of DRAM available as Plush, with half reserved for its hash index and the other half for the mutable part of its log. PMem-RocksDB [122] is a fork of *RocksDB* optimized for PMem. It serves as a representation of all established LSM trees that were adapted to PMem.

3.6.3 Fixed-Size Records

First, we evaluate how Plush deals with fixed-size records. We preload 100 million 16-byte records consisting of 8-byte keys and values. Keys are uniformly distributed. We then execute 100 million operations for varying thread counts. Figures 3.6 and 3.7 show the results.

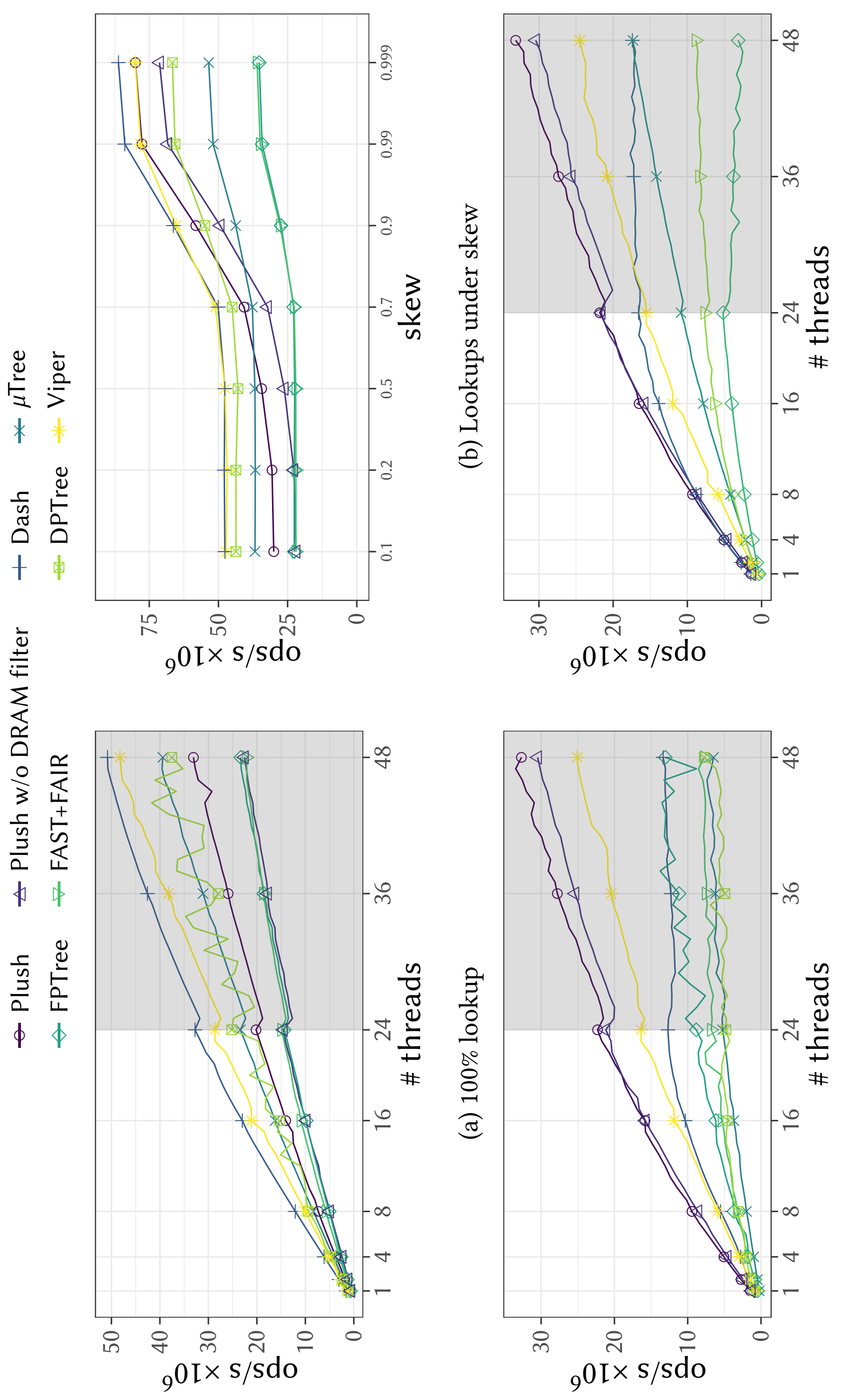
Throughput

For lookups (Figure 3.6a), Viper and Dash win over Plush and the tree data structures as both have just a single level. Plush has to look up the key at every level on average as it stores most records on the last level and therefore cannot end the search early. For lookups, it thus behaves like a tree. It still has an advantage (×1.41) over FPTree but is beaten by μTree and DPTree which store a copy of all keys in DRAM. Plush beats the other trees because of its hybrid design where it optionally stores bloom filters in DRAM (cf. Section 3.6.5), speeding up negative level lookups. Without this optimization, Plush has a similar lookup performance to those trees.

Plush's tiered design benefits from *temporal skew*: If Plush looks up a key that was recently inserted, it will probably still be in DRAM or a low PMem level: It can end the search early. Figure 3.6b illustrates this advantage. Here, lookups are Zipf-distributed [47]. The higher the skew factor, the more likely a lookup is for a record that was inserted just recently. While all data structures profit from skew due to caching effects, Plush profits disproportionally more, catching up with Viper and nearly reaching Dash.

Plush outperforms all data structures for inserts (Figure 3.6c), scaling nearly linearly up to 48 threads. It improves over Viper by 1.44×, Dash by 2.44×, and FPTree, the fastest tree, by 3.31×. The picture looks similar for deletes (Figure 3.6d), where Plush's performance is identical to inserts as a delete is just an insert of a tombstone.

Figure 3.7 shows how the data structures behave for mixed workloads on 48 threads ranging from only lookups (left) to only inserts (right). For insert ratios below 30%, Viper has higher throughput than Plush as it can leverage its superior lookup throughput below those insert ratios. Above that, Plush's insert throughput dominates.



(a) 100% lookup

(b) Lookups under skew

(c) 100% insert

(d) 100% delete

Figure 3.6: Throughput of core operations under varying thread count for fixed-size records (8-byte keys, 8-byte values).

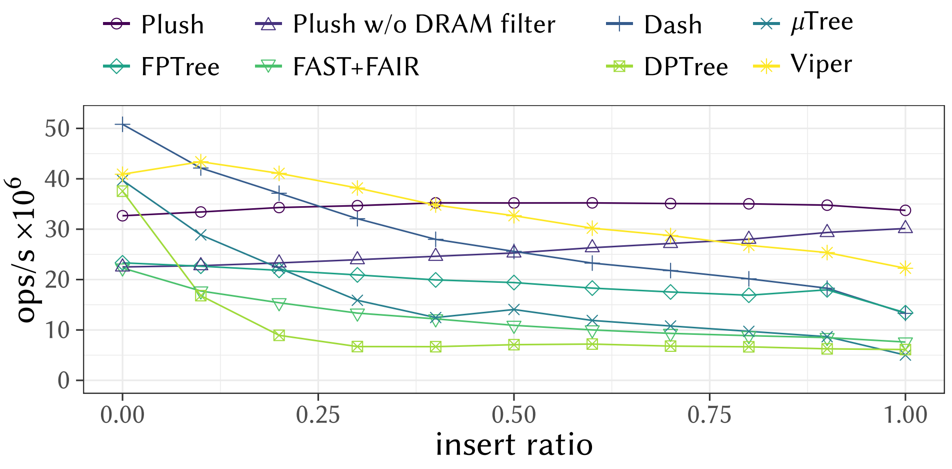


Figure 3.7: Throughput of mixed workloads with 48 threads for fixed-size records (8-byte keys, 8-byte values).

Overall, Plush is very *predictable*: It shows stable throughput at all insert ratios, scales well for all operations due to its partitioning design, low write amplification, lock-free lookups, and profits more than other data structures from temporal skew.

Read/write amplification

Figure 3.8 explains the throughput differences. It shows how much data is read/written per operation by the CPU at *cache line granularity* (64 bytes) on average. The overlayed hatched columns show data reaching the physical storage medium at *PMem block granularity* (256 bytes). The difference between the yellow and the hatched column is thus the overhead incurred by not packing writes and reads into 256B-blocks perfectly.

Plush's batching of reads and writes with a DRAM buffer decreases amplification on inserts: Its read and write amplification is just 63% resp. 70% of Viper, the runner-up. Viper, however, has *lower* end-to-end write amplification, as it *always* writes to PMem sequentially. This is the optimal write pattern for PMem, but it comes at a cost: Viper needs to keep a large hash table in DRAM that indexes all records, significantly increasing DRAM read- and write amplification. This experiment confirms that DRAM latency is not negligible for inserts, supporting our earlier observation that PMem's write latency is similar to DRAM's. Plush, Viper, DPTree, and Dash have similar PMem read amplifications for lookups, but Plush is at a disadvantage as it also has to read from DRAM. Even though μTree reads 56% less from PMem than Plush, its throughput is only 18% higher as it reads over twice as much data from DRAM.

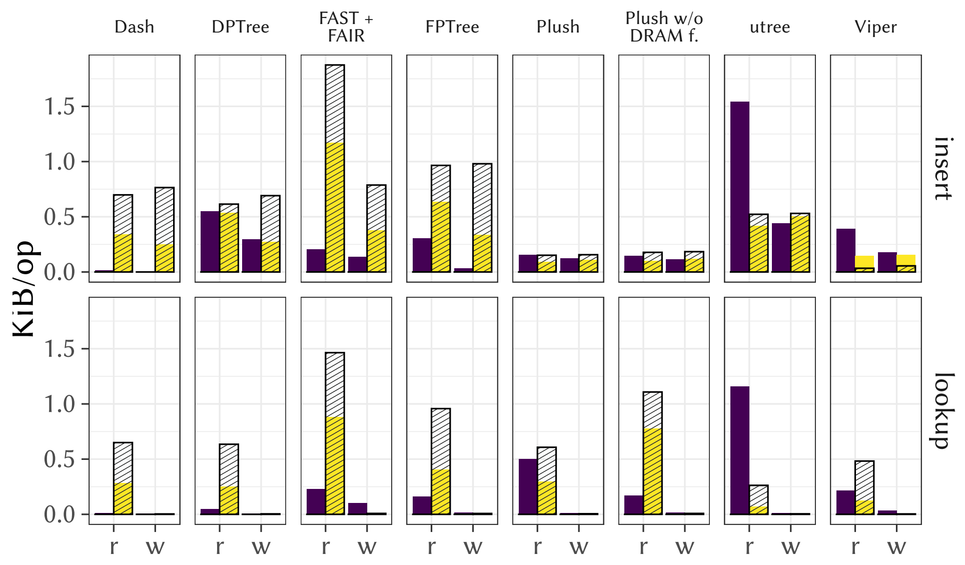


Figure 3.8: Read and write amplification for 16-byte records on DRAM (▮) and PMem (▮). Overlayed hatched columns (▨) show PMem media amplification.

This experiment confirms our initial assumption that the throughput of PMem data structures is heavily influenced by their write amplification, as write bandwidth is a bottleneck of PMem.

Latency

Plush's weakness is its migration step. To increase throughput by reducing communication overhead, Plush does *not* employ separate background migrator threads. Thus, a single "unlucky" insert might have to move a lot of data since a migration can start a chain of recursive migrations leading to higher tail latencies. We investigate the impact of this design decision on latency. For this, we run our workload on 23 threads (and one monitor thread) to rule out any SMT effects. Figure 3.9 shows the latency at different percentiles. Plush's insert latency is close to Dash's at lower percentiles. At the 99.9-percentile, Plush's latency increases sharply. This increase is expected, as on average, every 256th insert triggers a migration. However, Plush's latency does not worsen significantly at higher percentiles, and it even outperforms Dash again. Plush's and the other tree-like data structures' advantage is that they do not have to do any rare but costly structural modification operations like directory splits. For lookups, all data structures behave similarly.

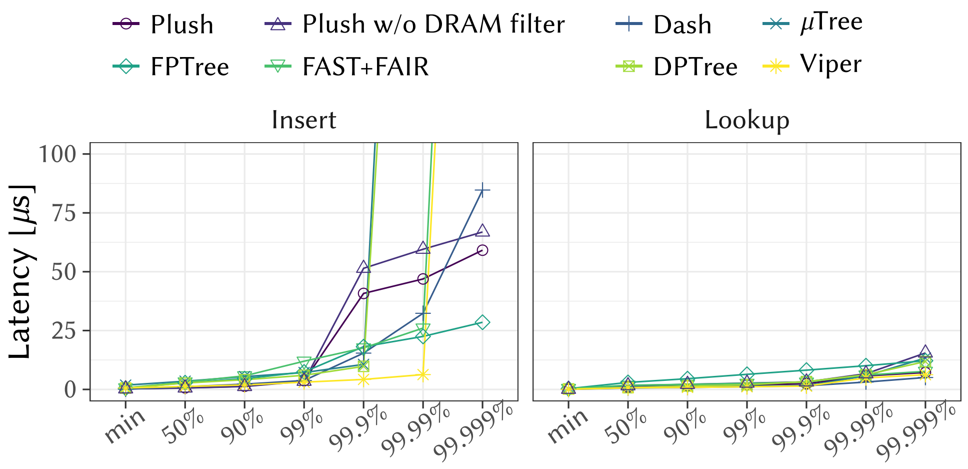


Figure 3.9: Tail latencies for 16-byte records with 23 threads.

3.6.4 Space Utilization

We fill all data structures with 16-byte records until we run out of storage space on our PMem partition, out of DRAM, or just crash. We then plot the total data set size (i. e., record count × 16 bytes) against the actual space consumption. Figure 3.10 shows the results for DRAM consumption (left) and overall space consumption (right).

DRAM consumption for both Plush variants and Dash is constant, with Plush using $<$ 1 GB and Dash not using DRAM at all. Viper, μTree, and DPTree store the bulk of their data in DRAM and are thus limited by DRAM capacity and cannot scale with the amount of installed PMem. They instead run out of DRAM or crash at $\approx$ 70 GiB inserted records, which can be seen on the right.

Regarding overall space consumption, both Plush variants show a huge increase at $\approx$ 50 GiB when they create the directory for PMem level 4 (directories for levels 1-3 are too small to be visible here) and the characteristic stair pattern already described in Section 3.6.1. While the trees show a more linear growth, the hash table Dash has even steeper jumps in size as it doubles the directory whenever full.

3.6.5 Plush Tuning

As explained in Sections 3.4.1, 3.5.2 and 3.5.7, Plush supports multiple optimizations trading off DRAM consumption, PMem consumption, or persistency guarantees for throughput. Figure 3.11 compares the impact of the different optimizations compared to the baseline.

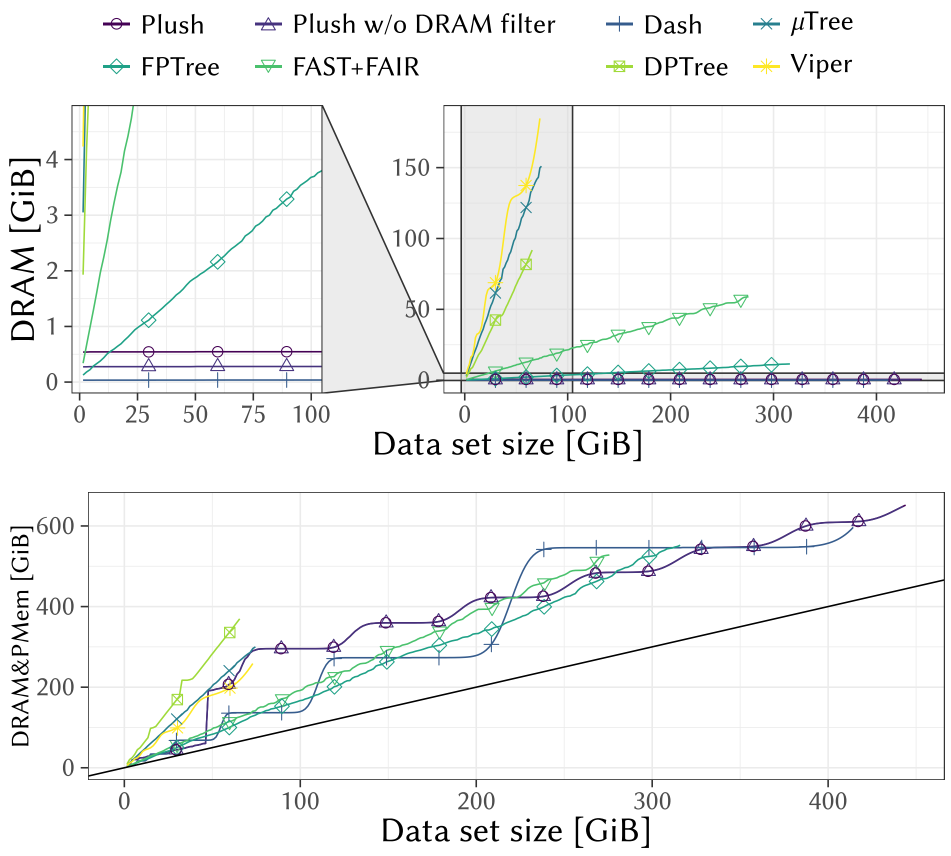


Figure 3.10: Storage consumption compared to data set size. Top: Just DRAM, Bottom: DRAM and PMem combined.

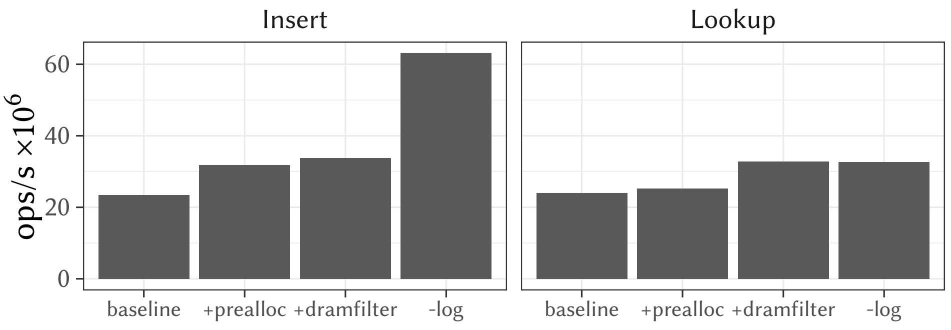


Figure 3.11: Throughput gained by optimizations.

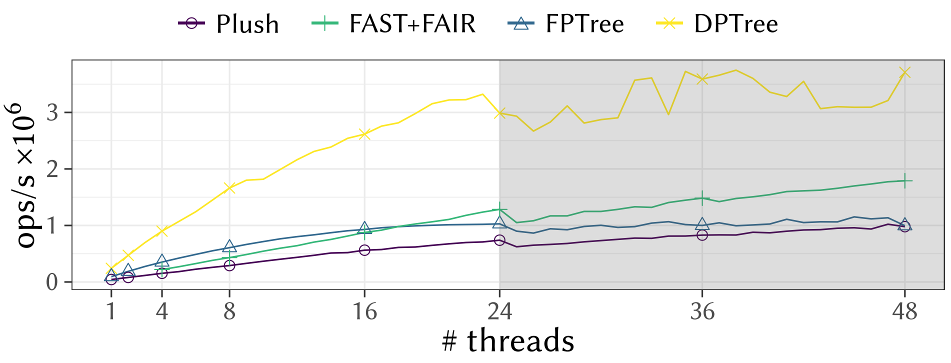


Figure 3.12: Throughput for range queries.

Pre-allocation. When enabled (`+prealloc`), all buckets up to the second PMem level are pre-allocated upon initialization. This setting does not increase memory consumption when enough records are inserted (as all levels except the last level are full anyway). However, it increases insert throughput (×1.36) and marginally increases lookup throughput (×1.05). Inserts are disproportionally faster as every migration saves `fanout` pointer dereferences while reading the records and ≈ fanout · 2 pointer dereferences for storing records on the next level. For our experiments, we pre-allocate up to the second PMem level which reserves 4.26 GiB PMem for the buckets.

Filters on DRAM. When additionally storing first two level's filters in DRAM (`+dramfilter`), insert throughput improves only slightly (×1.05) while lookup throughput improves considerably (×1.30). Here, lookups improve disproportionally as checking if a level contains a key no longer involves a costly PMem block read. As write latency is lower and inserts batch writes to PMem filters anyway, inserts do not benefit.

Skipping logging. When additionally not logging inserts (`-logs`), insert throughput increases dramatically (×1.89) while lookup throughput stagnates as lookups log nothing. As explained in Section 3.5.7, disabling logging is helpful for bulk loading. Creating a checkpoint with a single thread takes ≈ 840 ms. Plush also supports creating checkpoints concurrently. This is useful when Plush currently does not run other operations, e. g., after a bulk load before accepting requests. With 32 threads, checkpoint creation takes ≈ 69 ms.

3.6.6 Range Queries

By default, Plush partitions the key space into disjunct directory entries *by hash* to avoid skew. However, it supports arbitrary partitioning functions. When choosing a range partitioning function, Plush supports range queries: While

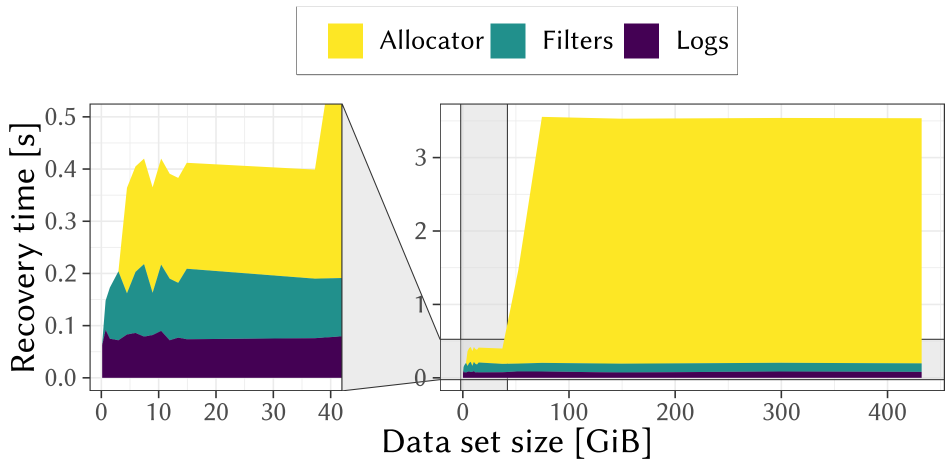


Figure 3.13: Recovery time vs. data set size for filters and logs.

records within a directory entry's buckets are still unsorted, it can use *divide and conquer* to only iterate over a few such entries. This is prone to skew but is an advantage over pure hash tables.

Figure 3.12 shows Plush's range query performance in range partition mode. The workload consists of a data set with 100 million records and 100 million lookups of random keys for which the next 100 larger records are returned in order. Plush keeps up with FPTree but is outperformed by the other trees as they can scan their sorted leaf nodes while Plush still has to check all unsorted buckets with potential candidates.

3.6.7 Recovery

While crashes in a production system should be rare, Plush should still recover quickly to keep downtime low. We load an increasing number of 16-byte records, forcibly terminate Plush during loading, and then measure the recovery time. Figure 3.13 shows the results for the three types of data Plush has to recover:

1) Records in the PMem log that had not been persisted in the hash table,
2) bloom filters that had been stored on DRAM (cf. Section 3.6.5),
3) and the status of the allocator (i. e., until which address it already had allocated buckets).

1) is constant as the log has a fixed size, 2) is constant as Plush only allows DRAM filters up to a given level (2, in this case) and 3) is linear in the number of levels (so logarithmic in the number of records). Note that this is the worst

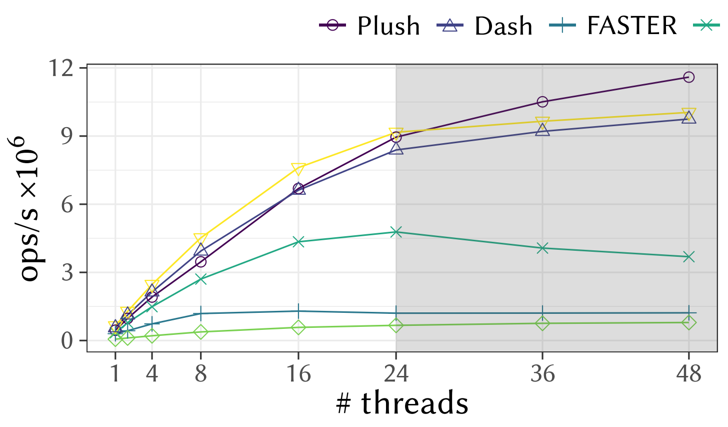
case as recovery time only grows with the number of records, not the data set size: If storing records larger than 16 bytes, Plush does not read the keys and values themselves during recovery, but just the pointers referencing them.

3.6.8 Variable-Length Records

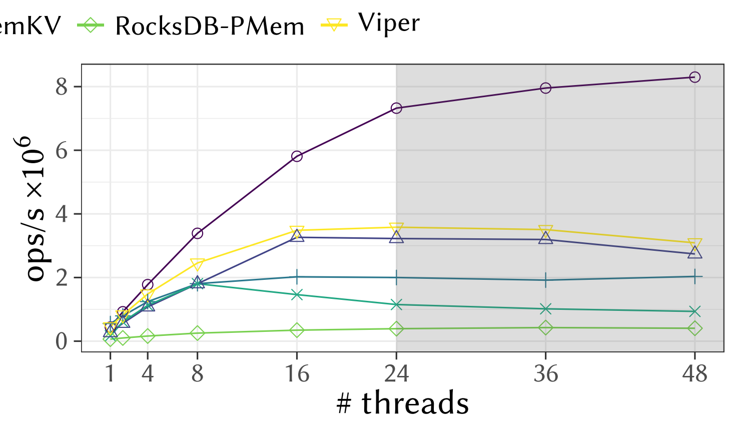
Figure 3.14 shows throughput for records having 16-byte keys and 1000-byte values. We evaluate a read-heavy (a), a mixed (b), and a write-heavy workload (c) with varying thread count. Plush scales well for read-heavy workloads but does not scale beyond 24 threads for write-heavy workloads. This observation aligns with our earlier findings that PMem's write bandwidth can be saturated by just a few threads. The more inserts we issue, the more bandwidth-starved all data structures become. For read-heavy workloads, Viper and Dash keep up with Plush (a). For low thread counts on an insert-heavy workload, FASTER overtakes Plush. Since FASTER gives fewer persistency guarantees, it may store records only in DRAM and flush them in bulk. This is an advantage over Plush, which keeps a write-ahead log. All data structures beat the RocksDB fork showing the advantage of designing a data structure optimized for PMem from the ground up instead of adding it as an afterthought. Figure 3.14d shows how throughput for the mixed workload changes with varying record sizes. Plush is latency-bound for smaller sizes but becomes bandwidth-limited as the record size increases. Plush profits from its low write amplification as it stores records out of place and from pre-faulting the payload log. Out-of-place storage is also the reason for the drop in throughput from 16-byte records to 32-byte records as storing the latter records out of place, leading to a level of indirection.

3.7 Related Work

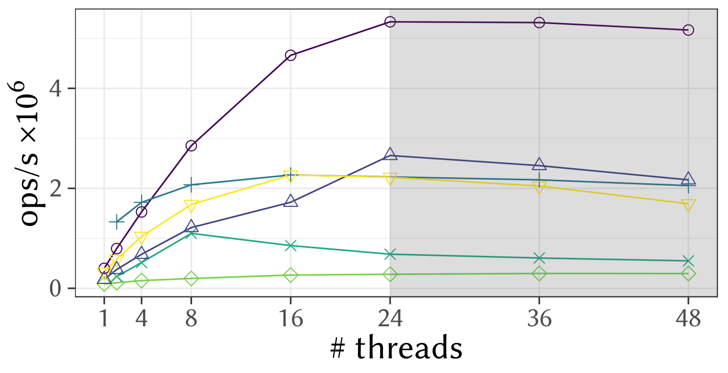
PMem's low bandwidth (compared to DRAM) has been researched extensively in the past [15, 31, 60, 71, 95, 156]. Plush specifically addresses issues raised by Gugnani et al. [50] and Woo et al. [156]. Most data structures mitigate this by adapting a hybrid design where reconstructible data resides in DRAM [13, 24, 58, 118, 163, 169]. Such data structures are usually B-Trees that store inner nodes inside DRAM (e. g., NV-Tree [163] or FPTree [118]), or even store (fingerprints of) keys in DRAM (e. g., μTree [24]). Viper [13] and HiKV [159] are hybrid hash tables. Viper logs records to PMem and stores their fingerprints in a DRAM hash table for efficient lookups and synchronization between threads. HiKV combines a PMem hash index with a DRAM B-Tree. These approaches do not have control over how much DRAM they consume as it depends on the data set size. LibreKV [96] uses a fixed-size hash table on DRAM like Plush but



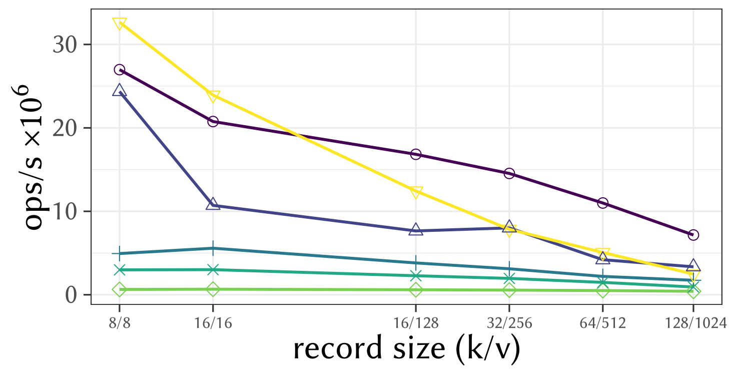
(a) Read-heavy (10% inserts)



(b) Mixed (50% inserts)



(c) Write-heavy (90% inserts)



(d) Varying record size

Figure 3.14: Throughput for variable-length entries. (a) - (c) show throughput for 3 different workloads with 16-byte keys and 1000-byte values, (d) shows throughput for varying record sizes under a mixed workload (50% inserts).

does not optimize for write amplification. Multi-tiered buffer managers [91, 132, 170] also make use of DRAM. PMem-only data structures do not consume any DRAM [5, 23, 57, 100, 105, 109, 172]. They are great for lookups but suffer from high write amplification during inserts. LB+-Tree [97] and write-optimized skip lists [160] minimize write amplification, but are PMem-only and thus have limited possibilities to reach that goal. Plush straddles the line by using a *fixed* amount of DRAM.

Most approaches to making LSM trees PMem-aware use an existing LSM tree like LevelDB as a foundation [21, 76, 81, 161]. They are, therefore, limited by LevelDB's architecture which was not designed with PMem in mind. NVLSM [166] is built for PMem from the ground up. Plush is also inspired by approaches employed by LSM trees not originally meant for PMem: Its value separation approach was initially proposed by WiscKey [101] and has since been adapted by other LSM trees like Parallax [158] or HashKV [21]. HashKV also inspired Plush's approach to partition its logs by hash. Haubenschild et al. propose global sequence numbers [53].

Plush's logging approach was invented by Mnemosyne [151] and further refined by our previous work [131]. PiBench [92] is a benchmarking framework for persistent indexes that we also used for our evaluation. Hu et al. adapted it for hash indexes [56].

3.8 Summary

This chapter introduces Plush, a write-optimized data structure for persistent memory. It employs techniques popularized by LSM trees to minimize write amplification and adapts them to PMem by replacing sorted runs with leveled hash tables and optimizing for 256-byte blocks. Because of Plush's low write amplification, it has higher insert throughput than existing PMem-optimized hash tables (2.44× of Dash) while having a lookup throughput comparable to fast tree-like PMem data structures. This confirmed our initial hypothesis that PMem data structures are often bandwidth-limited. Plush profits from temporal skew and excels at write-heavy workloads as its throughput stays constant even at high insert ratios. As Plush's DRAM consumption is independent of the data set size, Plush is arbitrarily scalable and only limited by PMem capacity.

CHAPTER 4

Data Pipes: Declarative Data Movement

Excerpts of this chapter have been published previously [148]. With contributions from Daniel Ritter, Danica Porobic, Pınar Tözün, Tianzheng Wang, and Alberto Lerner.

4.1 Introduction

The storage hierarchy in state-of-the-art computing systems has become deeper and more heterogeneous. Besides the traditional cache layers and DRAM, Persistent Memory (PMem) is now available on Intel platforms [162], and soon High-Bandwidth Memory (HBM) will also be on the market [139] Now widespread, technologies such as RDMA-enabled networks can connect stacks from different machines with low latency [42, 144]. In addition, platforms that enable computational storage and near-data processing are becoming more widely available [10, 90]. Programming with such a diverse set of technologies requires quite a skill set. Such a skill set is especially crucial in data-intensive systems, where efficient data movement is paramount.

The challenges to obtaining efficient and predictable data movement are numerous. The following is a non-exhaustive list:

1. HDDs, SSDs, PMem, and DRAM all have different device characteristics requiring the programmers to adopt different system optimizations.

2. There are various interfaces of different granularity when accessing these devices (e. g., block, zone, key-value, byte).

3. Different workloads (e. g., OLTP, OLAP, Machine Learning) exhibit different data access patterns and require different hardware-conscious optimizations.

4. The cloud and high-performance computing infrastructure have disaggregated storage, adding network-induced unpredictability.

5. Different CPU vendors offer support for different storage, such as Intel historically supporting persistent memory while AMD supports more PCIe lanes.

The conventional wisdom in writing data-intensive systems is to deal with each storage type individually, using whichever OS, file system, or library API that is available. This approach is valid when storing large data structures but only moving little data, e. g., indexes. The programmer decides which storage layer data should reside on and creates efficient access methods. This permits targeting optimizations for particular storage device [2, 13, 99, 119, 136, 150], stack [36, 59, 85, 103], or primitive [78, 129].

This approach breaks down when one wants to move large amounts of data. One such example is an external sort. Sorting is one of the most fundamental operations in data-intensive systems and, as we will show shortly, one that can significantly benefit from different storage technologies. The reason is that implementing all the data movements that an external sort requires is difficult precisely because of the diversity of storage options. The programmer is exposed directly to all the individual idiosyncrasies of each layer. We will use sort as an example throughout this chapter. However, many other operators and patterns exist in data-intensive systems that can benefit from significant data movement across storage layers. Rather than exposing the programmer to a jamboree of APIs and specific behaviors, we propose to give her an abstraction that can move data across any layers efficiently. We call this abstraction *Data Pipes*.

Data pipes is a *holistic*, *top-down* approach to creating data-intensive systems that utilize modern storage stacks. More specifically, data pipes offer programmers a *declarative* interface for *explicitly* controlling how data moves from one layer of the storage stack to another. Underneath, a framework determines which primitives to use based on the available hardware and requested data path. In other words, a data pipe will resort to a hardware-assisted data movement instead of wasting precious CPU cycles with `load` and `store` instructions whenever such a hardware unit exists. For instance, modern CPUs offer a little-known *uncore* DMA unit called I/OAT [67]. The I/OAT unit can move data from PMem to DRAM and back. For another instance, data pipes will resort to optimizations, such as DDIO [66], to move data directly to caches, skipping DRAM whenever possible. The main goal of the framework underneath the Data Pipes abstraction is to minimize data movement traffic and latency.

In summary, the contributions of the Data Pipes abstraction in this chapter are the following:

1. We survey the primitives that give us more control for orchestrating data movement over the storage hierarchy (Section 4.2).

2. We demonstrate the potential of these primitives over the use case of an external sort in terms of minimizing data movement latency and traffic (Section 4.3) and quantify the performance of two primitives, I/OAT and DDIO, in this context (Section 4.4).

3. We present our vision of *Data Pipes*, a *top-down* and *holistic* approach to creating a *declarative* control plane over data movement in data-intensive systems (Section 4.5) and discuss the guiding principles behind their design (Section 4.6).

4. We identify a research agenda targeting different system layers (from hardware to applications) to better support the data pipes vision (Section 4.7).

4.2 Background and Motivation

We have mentioned several storage layers but have yet to describe them. Figure 4.1 depicts the typical stack in a modern computing system. The right side shows the layers as a classic *size vs. latency* pyramid. The lower storage layers are larger and slower, while the upper layers are smaller but present better latency and bandwidth. On the left side of the figure, we present a partial list of mechanisms that allow a programmer to directly or indirectly move data between two layers. Note that the mechanisms range from complete protocols such as NVMe [114] to technologies such as DDIO to CPU instructions.

We classify the mechanisms by color-coding them according to how easy to use and accurate they are from a programming point of view. The green mechanisms are effortless for a programmer to access —e. g., via I/O system calls—and can issue precise data transfers. One such example is the NVMe protocol. On the other end of the spectrum, we portray highly specialized mechanisms in red. They require low-level programming skills and are sometimes advisory. Examples of such mechanisms are DDIO and I/OAT. In between, a savvy programmer can use a growing number of instructions to interact with the caching storage layers. When combined, these mechanisms can move data between almost any two distinct layers of the stack.

We note how non-uniform these mechanisms are. Some are transparent, such as promoting an entire cache line as a side-effect of reading data; others are explicit, such as issuing the recent `CLDEMOTE` instruction to perform the opposite movement. Some are implemented as hardware instructions, as above, while

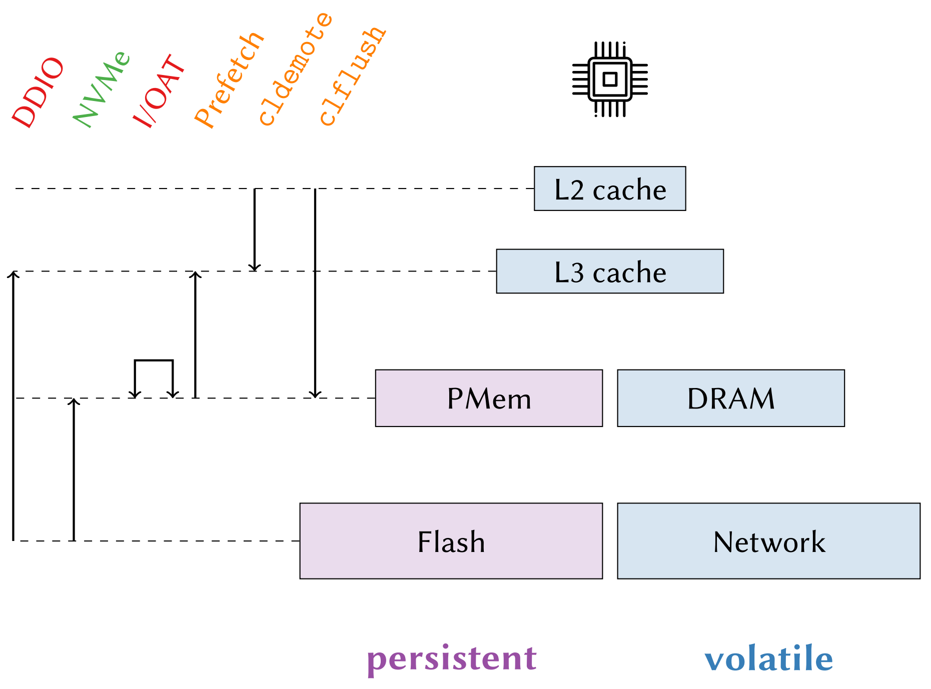


Figure 4.1: Data movement primitives (left) can shuffle data among storage layers (right). We claim that the set of primitives is, at best, incomplete and, arguably, incoherent from the programmer's point of view.

others are libraries. In particular, I/OAT defies classifications. As mentioned before, the I/OAT is a DMA unit that can move data without CPU intervention.

Some frameworks try to unify these mechanisms and make them more accessible to programmers, the most notorious one being Intel's Storage Performance Development Kit (SPDK) [142]. Indeed, we extensively used SPDK for the experiments we present in Section 4.4. SPDK is, however, very "opinionated" on how programs must be structured. It gives the programmer access to even the I/OAT unit but forbids her from using, for instance, established threading and many other useful libraries.

In contrast, we propose an API for data pipes that encapsulates the exact mechanisms SPDK does without imposing any other programming style or limitation to the application. As we commented above, we introduce these notions with the help of a motivation sorting example. Sorting is an operation we call *well-behaved* concerning data movements. Even though we cannot predict the amount of data to move, these operations have predictable data access and movement patterns. Some other well-behaved operations and workloads in data-intensive applications are the following: *logging*, which comprises data movements of small sequential records to low-latency persistent storage; and *checkpointing*, which moves several large chunks of DRAM-resident data structures in parallel, also into stable storage, to cite a few.

The advantage of recognizing a workload as predictable—well-behaved—is that **it allows the programmer to declare the data transfers in advance**. One of the goals of Data Pipes is to give the programmer the syntax to encode these declarations. Let us see one such example in practice.

4.3 Case Study: External Sort

In this section, we look at a typical external sort algorithm but concentrate on the types of data movement it generates. We pick external sort because it is a central building block for many data-intensive operations (e. g., compaction of log-structured merge trees [115], deduplication [120], and sorting query results).

An external has two phases: first, data is partitioned into small batches, and each batch is sorted, then the sorted batches are merged. Unsorted batches move "up" the storage stack to be sorted by the CPU and then need to be moved "down" to make space for new batches. The recently sorted batches should be kept as close to the CPU as possible, as the latter will operate on them again when merging. Figure 4.2 depicts the data movements for sorting (way up from storage/memory) and merging (way down). We discuss each of these movements in turn.

4.3.1 Data Movements in External Sort

We assume in the following that our system contains PMem. PMem can be used in different ways, but we focus on deploying it as a staging layer between sorting and merging.

Way up/Sorts. As shown in step ① of Figure 4.2, with `DDIO`, the data to be sorted is read directly from persistent storage (i. e., data source) to CPU caches via the DMA engine, bypassing main memory (hence avoiding extra memory allocations). The size unit of these fetches (*runs*) can be half the LLC size per core. We need twice the space to allow double buffering in LLC rather than going to DRAM while a core sorts the fetched data.

Larger-than-Memory Data Spills. We first sort all the runs before merging them in a later step. Each core performs soring until the DRAM size is exhausted. As seen in step ②, the sorted runs can be efficiently spilled to a staging area backed by a persistent storage device larger than DRAM, using `I/OAT` as soon as its initial block/page is produced. The staging area could use a different device

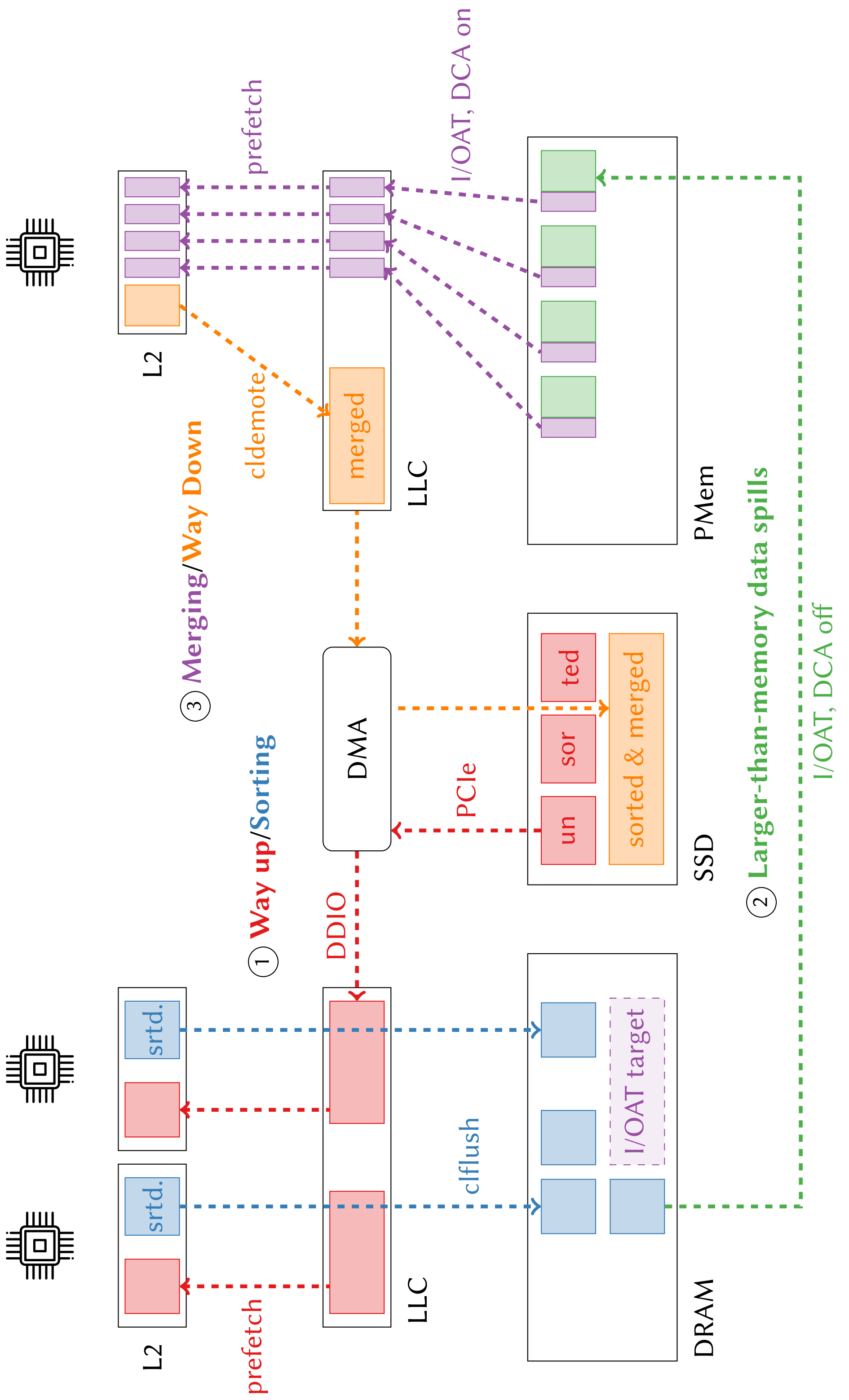


Figure 4.2: Hardware-efficient external sort using DDIO and I/OAT across CPU caches, DRAM, PMem, and SSDs.

than the data source (if available). Ideally, the device should exhibit low access latency, preferring PMem or the latest low-latency SSDs.

Way down/Merges. In step ③, the sorted runs are read from the staging area, using `I/OAT` via DMA as a fetch unit similar to the way up/sorting phase. However, since the runs are already sorted, the CPU only needs to perform a merge. This process is repeated until the run is sorted.

4.3.2 The Case for Hardware-Assisted Data Movement

Data movements should be computationally inexpensive in theory, and in many cases, they are. Take a transfer from a modern SSD, for instance. The NVMe protocol, which fast SSDs overwhelmingly use, can be seen as a layer atop the SSD's DMA engine. NVMe allows an application to point to data it wants to write instead of moving the data itself. However, not all layers are built as NVMe, and, in practice, transfers could be CPU intensive for multiple reasons: 1. They waste CPU cycles if synchronous storage APIs are used (e. g., `read()`, `write()` system calls); 2. Moving data between different layers can incur non-trivial overheads on the side (e. g., entering kernel mode or copying buffers); 3. They are inherently expensive because of implicit aspects (e. g., networking or PMem's load/store interface requiring CPU instructions).

Besides efficiency, another observation is that some data movements may occur *explicitly* (e. g., reading a block from an NVMe SSD). In contrast, other transfers are *implicit* (e. g., the CPU flushes a cache line). Anecdotally, practitioners invest significant time trying to coerce *implicit* data movements into efficient patterns for their applications.

With Data Pipes, we aim to make all data movements explicit. One way to achieve so is to assume that DMA units, rather than the CPU, will perform all movements. As mentioned above, this is already the case when transferring data in or out of NVMe devices. It is also the case when using RDMA-capable network interface cards (NICs). We resort to the I/OAT unit as a DMA agent for the remaining movement possibilities. In other words, we avoid using the CPU `load` and `store` instructions to transfer data whenever possible. We call this strategy *hardware-assisted data movement.* Putting it differently, one may see Data Pipes as wrappers to different DMA units available in a system. We discuss what Data Pipes look like shortly, but before doing so, we perform preliminary experiments to quantify the effects of hardware-assisted data movements.

4.4 Experiments

When using Data Pipes instead of coding data transfers by hand, a programmer can move the responsibility of optimizing those transfers to the Data Pipes implementation. This section evaluates such potential optimizations using two hardware-based mechanisms: `DDIO` and `I/OAT`. We start with one experiment of data loading from storage into LLC/L3 with DDIO. Then, we evaluate I/OAT for data spills from memory to storage and vice versa in two separate experiments.

In our experiments, we use the same machine es in the earlier chapters: A server equipped with a Xeon Gold 6212U CPU with 24 physical (48 logical) cores, 192 GiB RAM, 768 (6 · 128 GiB) first generation PMem, and a Samsung 970 Pro (PCIe 3) SSD.

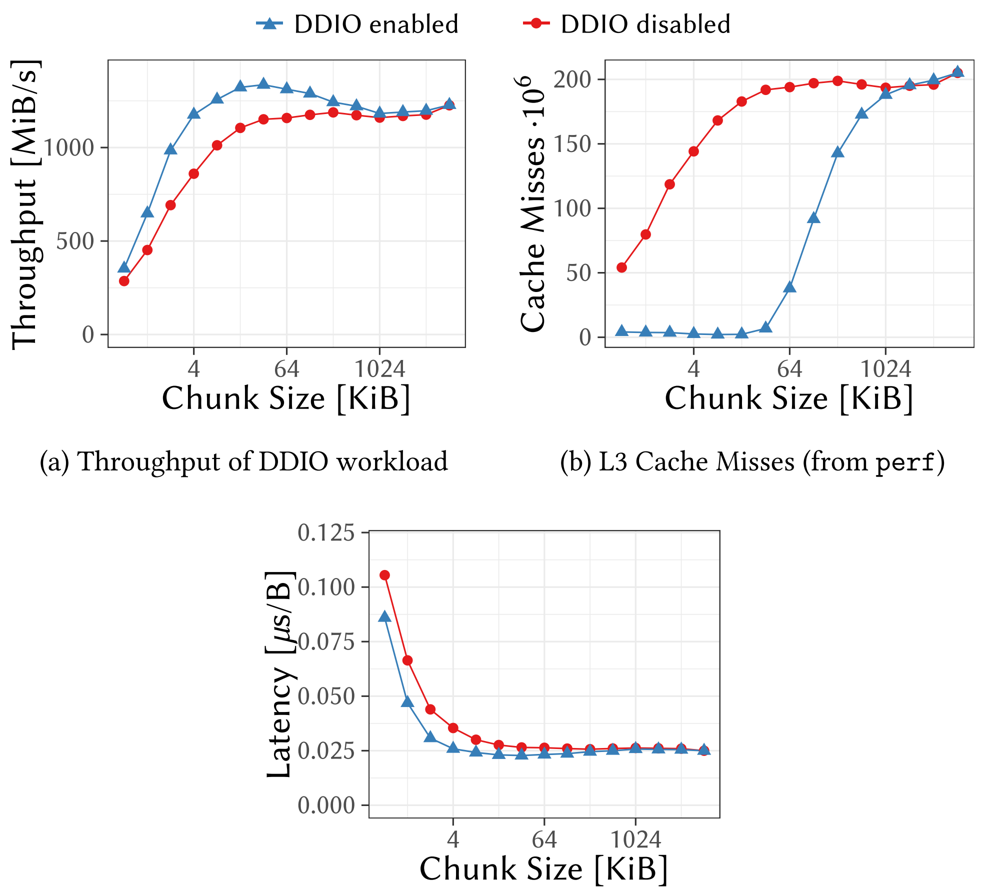
4.4.1 Fast Load from Storage to Compute

In our first experiment, we use the data movement accelerator DDIO, enabled by default on current Intel platforms, to move data from SSD toward the CPU for sorting. DDIO directly places data with DMA via PCIe into the L3 cache, assuming that requested data will be needed soon. Since we can also use DMA to read data from NVMe SSDs, we can re-purpose DDIO to load data from SSD and put it directly into L3 (cf. step ① in Figure 4.2).

The experiment consists of issuing reads at queue depth 32 and then iterating over the read data once whenever a request is finished (forcing all data into the caches). We use SPDK on one CPU core to copy integers in increasing chunk sizes to DRAM (with DDIO disabled) or L3 cache (with DDIO enabled) and sum them up. To show that we can perform the storage-to-cache movement without using DRAM bandwidth, we fully saturate the DRAM bandwidth by creating heavy artificial traffic using STREAM benchmark [106] in parallel in the background.

Figure 4.3 shows the resulting performance of DDIO on this DRAM-intensive workload. In Figure 4.3b, we observe that leveraging DDIO reduces cache misses (i. e., minimizing the side-effects of heavy DRAM traffic) as long as the chunk fits into the L3 cache. In addition, when the memory subsystem is taxed, DDIO can increase throughput (cf. Figure 4.3a) and reduce latency (cf. Figure 4.3c) even for "slow" SSDs (compared to NICs, i. e., DDIOs original use case), while freeing the CPU to do other computations (e. g., sorting).

We note that DDIO can be hard to use optimally as it depends on some hidden tuning knobs: Tuning the value of the undocumented `msr` register `IIO_LLC_WAYS` as explained by Farshin et al. [40] has a significant impact for this workload, as seen in Figure 4.4. Increasing its value from 2 to 11 reduces cache misses by up to 41% at chunk sizes below 1 MiB.



(a) Throughput of DDIO workload

(b) L3 Cache Misses (from `perf`)

(c) Latency of DDIO workload

Figure 4.3: Performance with enabled and disabled DDIO (one CPU core) with parallel DRAM-intensive STREAM workload.

4.4.2 Fast Load from Buffer to Memory

In our second experiment, we leverage I/OAT as a DMA engine to offload data movement between PMem (i. e., storage) and DRAM. This movement optimization comes in handy when loading spilled data during sorting (cf. step ③ in Figure 4.2). Here, offloading data movement is especially valuable since PMem uses a `load()`/`store()` interface like DRAM, where each access is a CPU instruction. This experiment uses SPDK's `accel_fw` [142] on a single core to issue copy requests of increasing size from PMem at queue depth 8, comparing `memcpy` (internally using non-temporal load, store) to the I/OAT backend.

The resulting throughput is shown in Figure 4.5a. When deploying I/OAT, moving data from PMem to DRAM is up to ≈ 2.57× faster compared to `memcpy`

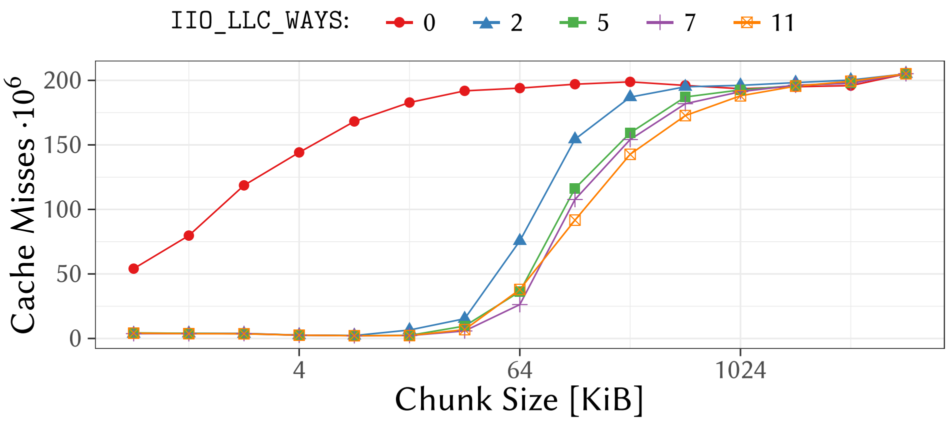
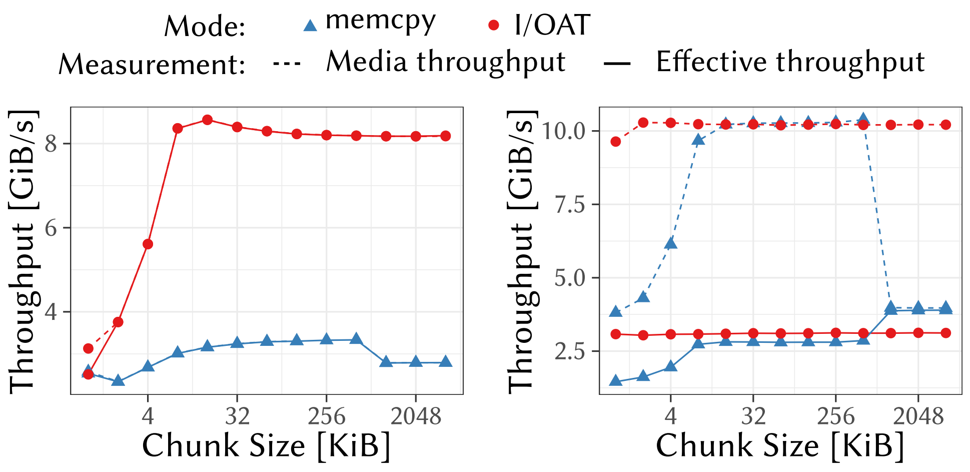


Figure 4.4: Impact of the `IIO_LLC_WAYS` register on cache misses with DDIO enabled.



(a) Moving data from PMem to DRAM (b) Moving data from DRAM to PMem

Figure 4.5: Throughput when moving data between DRAM and PMem with and without I/OAT.

on a single core. Hence, for a throughput comparable to I/OAT, three CPU cores need to be dedicated to data movement. This emphasizes the benefit of offloading data movement from the CPU.

4.4.3 Lack of Control for Data Spills to Buffer

In our third experiment, we look into the reverse data movement from the experiment in Section 4.4.2. We use the I/OAT unit once again, this time to move data from DRAM to PMem (cf. step ② in Figure 4.2).

Figure 4.5b shows the resulting throughput. These experiments reveal some issues with I/OAT. While writing to PMem, I/OAT is still marginally faster at chunk sizes $\leq$ 512 KiB, but it only reaches a third of the read throughput. That is far below PMem's potential write throughput. To investigate further, we measured the PMem media throughput (i. e., the write throughput the physical DIMM experiences), which is $\approx$ 10 GiB/s, close to PMem's maximum write throughput [131]. The reason for the high write amplification was first discovered by Kalia et al. [78]: I/OAT implicitly puts data into the L3 cache when it moves it to PMem using DMA, assuming it will be processed soon. This feature is called Direct Cache Access (DCA) and is not readily usable on modern CPUs. While this is great when moving data *in* from PMem, it is a huge bottleneck when moving data *out* to PMem for two reasons:

1. It evicts other data from the L3 cache and replaces it with data not intended to be accessed (otherwise, we would not have moved it out of DRAM).

2. When the CPU finally evicts the data, it does not evict it sequentially but semi-randomly. As each cache line is 64 bytes and PMem's internal block size is 256 bytes, each cache line eviction triggers a block write, resulting in up to 4× write amplification ($\approx$ 3.27× in our measurements).

These experiments show that using I/OAT to offload memory movement to/from PMem is hard to implement in practice. What compounds the issue is that Intel does not document details about DCA or how to toggle it. Even on CPUs that nominally support disabling it, it often is not exposed in the BIOS configuration and requires error-prone fiddling with Intel's `msr` registers. In our setup, Intel does not expose the `msr` register, leaving us with no way to disable DCA. In other words, hiding these complexities behind a Data Pipe interface allows us to adapt to systems that permit the configuration and fall back to a less optimized transfer in systems that do not.

4.4.4 Discussion

Our experiments showed that existing hardware-assisted data movement mechanisms such as DDIO and I/OAT are beneficial for database workloads, but they also uncovered several challenges.

The main issue is how obscure some of these mechanisms are. I/OAT's DCA issue when copying to PMem and DDIO's hidden tuning knobs highlights a central problem with our current data movement primitives. These primitives provide significant speedups if use cases match the scenarios for which they were originally designed. It is possible to deviate from those scenarios but not without mixed results (i. e., DDIO is great with reducing traffic in caches but shows limited latency improvements and is hard to tune; I/OAT is hard to handle). We require further experiments to determine all the constraints, tuning knobs, and implicit assumptions required to make the hardware mechanisms work for Data Pipes.

4.5 Our Vision: Data Pipes

We discussed Data Pipes as an abstraction and showed that some powerful hardware-assisted data movement mechanisms to support them are available, even if these mechanisms are tricky to configure. Therefore, we hide the implementation complexity under a friendlier API and expose only concepts familiar to data-intensive programmers. We chose to do so by making Data Pipes resemble a new type of descriptor/object in a C/C++ sense. Once a pipe is instantiated, it can transfer data from source to destination via a special call. We make the pipe's source and destination explicit by introducing the concept of *resource locators.* These decisions still allow us to experiment with different programming styles when using Data Pipes.

Data Pipe Flavors. We propose three different flavors of data pipes. They mainly differ on whether the programmer wants to: (1) wait on the transfer, (2) be asynchronously notified when it is done, (3) or whether there is OS support for the wait. Listing 4.1 implements the data movement step ① of Figure 4.2 using flavor (1), i. e., it moves data in L2 cache-sized chunks to the cache of each core, where it is sorted and then demoted to DRAM. Listing 4.2 implements step ② using flavor (2), spooling sorted runs from DRAM to the backing PMem. Listing 4.3 illustrates flavor (3) by retrieving sorted runs from PMem and merging them (step ③).

We discuss these flavors shortly, but first, we present the abstraction of the *resource locator* in more detail.

4.5.1 Resource Locators

Resource Locators can declare a source or destination of a data movement. We highlight resource locators code in green. Different types of resource locators are used for different storage devices, e. g., a `DRAMResourceLocator`, an `SSDResourceLocator`, or a `CoreCacheResourceLocator` for the L2 cache of a given core. Employing this abstraction has at least two advantages. First, it presents a uniform start and endpoint to which a data pipe can connect. Second, it enforces a type system of sorts on data movement. Traditionally, most data access is "loosely typed" as a pointer and an offset (memory-mapped devices) or file descriptor (block devices). With the thin abstraction of resource locators and pipes, data movements have become intentional and "strongly typed." While a locator internally might be a pointer, the user is now forced to think about what that pointer represents and where this data is supposed to be moved to, which aligns with our earlier goal of making data movement *explicit* and *declarative*.

Underneath each locator type, we include code that makes that storage area available for use. The locator is responsible for acquiring/releasing that resource (e. g., `malloc()`/`free()` for DRAM, issuing NVMe commands, through `io_uring` for instance, for SSDs). The design of each locator thus depends on the resource it manages:

- Locators backed by a file take the path to a file as an argument (e. g., the `SSDResourceLocator` in Listing 4.1, Section 4.5.2)
- A DRAM locator only needs a size to be constructed (Listing 4.1, Section 4.5.2) as it allocates its memory by itself.
- Locators not referencing addressable memory, such as the core cache locator that references L2 cache (Listing 4.1, Section 4.5.2), need to be backed by a memory area.

4.5.2 Data Pipes

A Data Pipe connects two resource locators, *A* and *B*. For simplicity, it is unidirectional, so it can only `transmit` data from *A* to *B*. As Listings 4.1 to 4.3 show, all flavors are declarative.

To use a Data Pipe, a programmer first declares the locators she intends to use. She then prepares data movement by connecting the locators with pipes before moving data and performing additional computations. This concept can be rendered in different ways. We describe three variations next.

```
size_t buffer_sz = 1 * GB;
size_t run_sz = CoreCacheResourceLocator::CacheSize;

SSDResourceLocator ssd_locator("/path/to/ssd/file");
DRAMResourceLocator dram_locator(buffer_sz);

do_parallel_foreach_core {
    size_t offset = core_idx * run_sz;

    // Allocate cache at the local core
    // backed by a memory area
    CoreCacheResourceLocator cache_locator(
        run_sz,
        dram_locator + offset);

    Pipe ssd_uppipe(ssd_locator, cache_locator);
    Pipe cache_downpipe(cache_locator, dram_locator);

    // Will try to use DDIO since this is
    // a disk-to-cache transfer
    ssd_uppipe.transfer(
        offset /*ssd offset*/,
        0 /*cache offset*/,
        run_sz);

    sort(cache_locator.data(), run_sz);

    // Will try to use CLDEMOTE since this is
    // a cache-to-DRAM transfer
    cache_downpipe.transfer(
        0 /*cache offset*/,
        offset /*dram offset*/,
        run_sz);
}
```

Listing 4.1: Straightforward flavor, loading data from SSD and sorting it into runs. Step ① in Figure 4.2.

Straightforward flavor

Here (Listing 4.1), pipes are objects initialized with source and target resource locators (Lines 16 – 17) and provide a `transmit()` method. This method takes two offsets, one for the source and one for the target resource locator. The `transmit()` call blocks until data is successfully moved to the target locator. This flavor is easy to implement, e. g., as a library, and integrate into an existing application as the caller of the pipes never relinquishes control (similar to traditional blocking I/O).

Inversion of control flavor

This flavor (Listing 4.2) schedules data movement asynchronously. A runtime, initialized once (Lines 10 – 13), runs concurrently with the application and is responsible for scheduling. The pipe's `transmit()` method is asynchronous and signals their completion via a `future` argument (lines 20–30).[1] The application thus relinquishes control to the pipe runtime.

The advantage of this approach is that multiple data pipes can run in parallel with a central coordinator keeping track of progress and scheduling data movements optimally. In this example, a single thread can trigger multiple data movements in parallel, leaving scheduling and CPU allocation to the pipe runtime. The downside is that inversion of control is often hard to incorporate into an existing application as it might complicate the programming model and add synchronization overhead. Communicating over a future involves a mutex which might add negligible overhead when moving megabytes of data but is very expensive if moving just a few bytes between caches.

OS-supported flavor

The previous approaches depend on a library written in the application's language. In this third flavor (Listing 4.3), a pipe is an abstraction at the OS level:[2] Here, pipes are OS concepts represented by file descriptors (Lines 4 – 6), and so one can `transfer` data (Lines 24 – 30) analogous to `pread`/`pwrite`. Leveraging OS support, such as `epoll` in Linux, the application can monitor the pipe's state. As shown by Lines 8–12, the application uses `epoll` to obtain a file descriptor that allows it to get notified when the pipe can start a new transfer. Afterward, we spawn multiple threads that wait until the pipe is ready to accept new requests (Lines 15 - 20) via the `epoll` API and then issue a transfer request.

[1]We use C++'s promises, but C-style callbacks would work just as well.

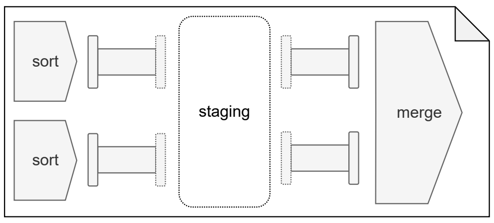
[2]Note that this is just a proposal on how such an interface could look. The implementation would require specialized kernel support.

```
size_t base = 0;
size_t pmem_offset = 0;

PMemResourceLocator pmem_locator(
    "/dev/dax0.1",
    pmem_offset,
    sz);

// Let the runtime run concurrently in the background
PipeRuntime runtime;
runtime.fork_and_start();

Pipe dram_downpipe(dram_locator, pmem_locator, &runtime);

while (is_sorting_runs) {
    //Collect pending tasks
    vector<future> futures;
    for (size_t offs = 0; offs < wtrmark; offs += run_sz) {

        promise<void> write_promise;
        futures.push_back(write_promise.get_future());

        // Uses I/OAT without DCA to not pollute the cache.
        // The runtime schedules moves asynchronously.
        // The promise is transferred to the runtime.
        dram_downpipe.transfer_with_cb(
            offs /*dram offset*/,
            base + offs /*pmem offset*/,
            run_sz,
            move(write_promise));
    }
    base += wtrmark;
    // Block until all moves are done
    wait_all(futures).wait();
    futures.clear();
}
```

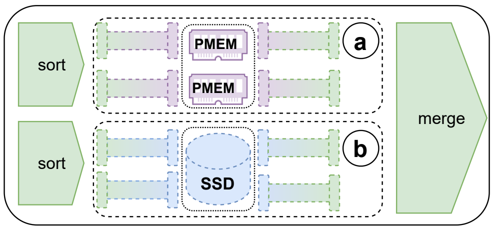
Listing 4.2: Inversion of Control flavor, store sorted runs on PMem. Step (2) in Figure 4.2.

```
int k = 4;
size_t merge_sz = k * run_sz;

int pmem_uppipe_fd = create_pipe(
    pmem_locator,
    cache_locator);

int epoll_fd = epoll_create1(0);
epoll_event pmem_pipe_op;
pmem_pipe_op.events = EPOLLTRANSFER;
pmem_pipe_op.fd = pmem_uppipe_fd;
epoll_ctl(epoll_fd, EPOLL_CTL_ADD, 0, &pmem_pipe_op);

do_parallel_for_each_core {
    // Wait until the pipe can transfer,
    // as the I/OAT unit might be occupied elsewhere.
    epoll_event event;
    epoll_wait(epoll_fd, &event, 1);
    if (/* error */)
        break;

    size_t offset = core_idx * merge_sz;

    // Issue a DMA request from PMem using I/OAT.
    // Recall that a memory area backs the cache locator
    pipe_transfer(
        pmem_uppipe_fd,
        offset /*pmem offset*/,
        0 /*cache offset*/,
        merge_sz);
    merge(k, event.locator);
}
```

Listing 4.3: OS supported flavor, load and merge sorted runs. Step ③ in Figure 4.2.



(a) Blueprint for a declarative external sort using pipes.



(b) Instantiated template with data pipes: Operator can be instantiated using PMem(a) or SSD as a staging area (b).

Figure 4.6: Using data pipes to make algorithms modular.

From a programmer's perspective, this approach is an abstraction level below the other two approaches: The user has to check whether the data pipe is in the correct state before issuing any requests. While this approach is more involved than the other two, it comes with two advantages: (1) It takes a big step towards being programming language agnostic, as it relies on and extends a universally known and supported interface (`epoll`, `pread/pwrite`) of the operating system. (2) It serves as a foundation upon which a library for the other two approaches can be built: Encapsulating Lines 15 – 30 into a `transfer` method yields the behavior of the "straightforward" approach, adding a request to a queue before `epolling` on a separate thread yields the "inversion of control" approach. This approach, however, also has the downside of requiring still-to-be-developed kernel support.

4.5.3 Data Pipes Optimization

We note that in well-behaved workloads, there are often options for where to move data. In our external sort illustrated in Figure 4.2, for example, we arbitrarily decided to spill runs to PMem. However, PMem is not universally available, e. g., in AMD CPUs. Depending on the system configuration, we could store sorted batches on the source SSD or a second SSD, if available. Since data pipes already follow a declarative approach, we can abstract over which intermediate storage device the runs are placed. We illustrate here a fourth possibility: that in which an optimizer within the runtime picks the "proper" storage device during execution instead, considering the storage devices available in the system.

Figure 4.6a illustrated this possibility. It depicts how multiple threads sort small runs, which are transferred via (for now) abstract data pipes to a staging area and, from there, are merged, producing the sorted output. Since data pipes are declarative, an optimizing runtime can decide during execution how to *instantiate* the abstract pipes. Figure 4.6b shows two such options. A sort might start with path (a), storing sorted runs in PMem, and switching to option (b) using an SSD when PMem is exhausted. Since pipes are declarative, this would come with minimal overhead for the programmer and would allow the algorithm to be split into smaller parts that can be connected via pipes in a dynamic manner.

4.6 Data Pipes Principles

We have shown a few examples of how data pipes could be used as a programming artifact, but we have yet to discuss the guiding principles behind their design. This section does so. We start with two principles that were already demonstrated:

(1) Declarative The programmer declares beforehand which data pipes they wish to use, e. g., from a PCIe NVMe SSD to a cache layer or from there to PMem. The upfront declaration makes the intention of the programmer *explicit*, which allows the system to 1. make specific optimizations (e. g., enabling or disabling DCA), and 2. *reject* ways to move data for which no suitable optimization is implemented. The programmer can always check if the intended data path is used and adjust the application logic otherwise.

(2) Composable As seen in our external sort example in Section 4.3 and Figure 4.6, data movement primitives depend on each other's results. Data is loaded, operated on, and then moved again (i. e., spilled back to background storage or moved to the cache of a different core). Making data pipes declarative allows the programmer to *compose* them and thus indicate which dependencies between computation and data movement exist. This approach is very similar to traditional query engines where optimizing data movement also plays a big part (i. e., vectorized vs. code generation).

There are additional principles we now introduce that should be just as an integral part of data pipes:

(3) Configurable Current data movement primitives are hard to configure. They either (1) cannot be configured at all (caching behavior), (2) only be configured coarsely (e. g., globally turning the prefetcher on/off), or (3) rely on undocumented `msr` registers (e. g., DCA, amount of L3 cache available to DDIO). Data pipes can instead expose those tuning knobs to the programmer. This allows for tighter integration between software and hardware, as each knob can be tuned to the application's specific needs. In our DDIO experiment, tuning the value of the undocumented register `IIO_LLC_WAYS` significantly reduces cache misses, as shown in Figure 4.4. Exposing and documenting features like this would thus make it easier to benefit from DDIO.

(4) Visible State Being configurable alone is no silver bullet: Some aspects cannot be tuned as they are fixed hardware properties (e. g., cache associativity or cache line size), and it is unrealistic to expect hardware vendors to make them tuneable. Data pipes, however, can make such aspects and their state visible to the programmer. As such, the programmer is not forced to guess or infer such constants via heuristics (e. g., CPU generation, manufacturer, benchmarking), thus making them more confident in the applicability and benefits of a particular pipe upfront.

Lastly, we have indirect goals we wish the data pipes API also to achieve:

(5) Orthogonal to existing primitives Data pipes do not rely on hardware vendors implementing *new* data movement primitives. While we believe that programmers would profit from additional data movement primitives that are not currently available, there is already a huge benefit in making existing primitives more easily accessible and configurable. We thus do *not* urge hardware vendors to invent *new* ways to resolve all challenges in designing hardware-conscious data management software. We instead want them to enhance the interfaces to current data movement primitives to make it easier for applications to benefit from them.

(6) Inspiring new primitives On the other hand, data pipes can pave the way to creating new primitives that benefit data-intensive systems, since they offer a straightforward and user-friendly way for programmers to indicate their desired data orchestration motives. This mode of indicating desired movements and paths could ease the communication between hardware vendors and software programmers to better address the needs of data-intensive systems.

Our work is aligned with other efforts to provide higher-level abstractions on which to build data-intensive systems. Some recent examples of this line of work are `DFI` [144], which help programmers to make use of RDMA networks, and `xNVME` [102], which does the same in spirit but for fast NVMe devices. We share several traits with these abstractions but try to take a unified view of data movement independent of data location.

While data pipes can easily be used to implement well-behaved workloads efficiently, other workloads could be more challenging. For example, OLTP is characterized by the unpredictability of the read and write patterns. In literature, a common way to handle such not-well behaved workloads is to create hardware-conscious data structures such as log-structured merge trees [115, 136], B-epsilon tree [18], Plush [150], Apex [99], and others. When creating such data structures, the main design goal is to morph the workload's unpredictable data access patterns or movement into a more well-behaved pattern for the target storage device. There are also recent works, such as Umzi [103], Mosaic [149], or NovaLSM [59] that target multiple layers of storage hierarchy or disaggregated storage. Enhancing these proposals with data pipes should not be an issue as long as the predictable access patterns are identifiable in the system.

4.7 Related Work and Research Agenda

It is possible to turn data pipes vision into a real-world framework even by using the available primitives and software libraries today. However, to be able to create a flexible and efficient framework across different programming languages and computer infrastructures (bare metal to the cloud), additional support from different computer systems layers is essential. We are not the only ones researching such solutions. Therefore, in this section, we identify related and future research directions that can enable better support for data movement in general and Data Pipes in particular. We divide these efforts according to the context in which they are being studied: from the OS, the hardware, or the cloud infrastructure perspective.

4.7.1 Operating Systems

While performing the preliminary experiments for our vision of *data pipes* in Section 4.4, we relied heavily on SPDK to get access to the low-level primitives to control data movement. However, this comes at the cost of using a niche and unintuitive programming model. To keep the data pipes programming model as simple and adaptable as possible across programming languages and infrastructure deployments, it would be ideal to have better OS support for accessing low-level primitives rather than always relying on OS-bypass techniques (such as Arrakis [121], Demikernel [168], Persephoné [34]).

Furthermore, enabling an application that uses data pipes to collocate with other applications that may not use data pipes requires OS support as well. The OS needs to be aware of the data pipes and avoid swapping memory or evicting last-level cache (LLC) blocks that are explicitly needed by a pipe. One solution is to reserve part of the main memory/CPU caches to be exclusive to data pipes or to give priority to data pipes to prevent other collocated applications from thrashing memory or LLC.

These wishes are plausible and would benefit more approaches than data pipes, given recent efforts that crave more explicit control over memory regions. For example, to tackle DDIO's problems, IAT [164] and IDIO [1] devise efficient frameworks that can monitor I/O and cache traffic to customize data placement in the LLC for better performance isolation. Performance isolation on LLC can also be achieved by using Intel's Cache Allocation Technology (CAT) [113] or by configuring DDIO usage via some recently discovered mechanisms [40]. Furthermore, DimmStore [83] explores different data layouts in main memory chips for energy efficiency. Differentiated storage service [107] allows classifying I/O operations to process different requests with proper policies. Data pipes can leverage such differentiated storage services and LLC management techniques to use dedicated policies with priority, avoiding thrashing across layers.

4.7.2 Hardware

The current I/OAT unit in the Intel Xeon line of chips is an example of a DMA unit that can support transfer between "upper" layers of the storage hierarchy, such as caches, DRAM, and PMem. It has been shown that it can free the CPU while it performs asynchronous memory copies [146, 147]. This DMA unit, however, can be improved in at least three ways. First, while it delivers latency benefits over, for instance, the highly optimized glibc's `memcpy()` [25], it may present lower bandwidth when it comes to small data transfers. Second, the unit presents a limited number of channels. The exact number is a piece of information protected under NDA, but the maximum number of channels reported has been 16 [25]. For comparison, we note that the number of *tags* in a PCIe Gen 3 system, arguably the equivalent mechanism to support parallel transfers, is 256. This number keeps growing; it is 1024 for PCI4–and both PCIe generations allow *extended tags*, further increasing this number. Third, the I/OAT unit does not support advanced transfer mechanisms such as scatter/gather [29], in which several non-contiguous memory ranges are transferred in one operation. Despite all the limitations, there have been reports of successfully incorporating I/OAT into sophisticated data movement schemes [19].

Studies, such as *memif* [93], have also experimented with more powerful DMA units. That work, however, confined the use of the DMA unit to the operating system's use, for instance, for data movement caused by page migration. They justify the decision by noting the lack of mechanisms to notify an application once a requested transfer is done. We demonstrated three possible ways of dealing with the issue. Putting it differently, we believe that a DMA unit that overcomes the challenges we listed above can be quite valuable for data pipes and can be successfully made accessible to applications.

We seek a future DMA unit with extended capabilities in another specific direction: increased reach. By increased reach, we mean accessing a portion of the storage hierarchy that remains closed. For instance, nothing can reach the CPU registers that do not come from the L1 cache. Recent examples, such as the `nanoPU` NIC [63], show that transferring data straight into the CPU registers can significantly reduce communication latency. This, in turn, can support new algorithms such as record-breaking sorting techniques [72]. Moreover, we also mean by the increased reach that a more modern unit should keep pace with any new type of memory that newer systems will bring. One such imminent example, as we mentioned above, is High-Bandwidth Memory [75]. The next generation of Intel Xeon chips, codenamed Sapphire Rapids, will support this type of memory [139], and there have been reports of the HBM benefits for the kind of data-intensive applications that we address here [82].

4.7.3 Cloud Infrastructure

Cloud infrastructure is becoming the de-facto environment for the development and deployment of modern applications, and data pipes have the potential to be very valuable both to cloud infrastructure providers and cloud application developers. Current cloud infrastructure providers offer much flexibility in compute and storage deployments ranging from a wide variety of virtual machine or bare metal instances [9, 45] to fully flexible resource sizing [117] to stateless compute [8]. However, data-intensive applications often make resource sharing challenging and can quickly become a noisy neighbor to others. Data pipes that make data movement predictable can alleviate this problem and help schedule and balance resource usage in shared infrastructure environments.

From the cloud application perspective, the goal of predictable data movement performance often requires over-provisioning shared or provisioning dedicated infrastructure to avoid noisy neighbors. Exposing data pipes as a first-class resource with predictable throughput and latency would make it much easier to ensure performance predictability at the application level. Furthermore, using a common API for data movement across different layers of memory and storage hierarchy would make it much easier for applications to use each new and improved generation of devices without significant application changes.

4.8 Summary

In this chapter, we motivated and illustrated a vision, *data pipes*, where the programmers can dictate and fine-tune how data moves from one storage layer to another in a declarative manner. Data pipes can make data movement more *visible* and *configurable* at the application layer. Moreover, they would not clash with existing low-level primitives to control data movement while having the potential to inspire new ones. It allows a user-friendly abstraction while making use of the modern storage stack to achieve low latency and reduce data movement traffic in data-intensive systems.

CHAPTER 5

Conclusions and Future Work

Modern storage devices have developed unprecedentedly in the last ten years, forcing database systems to adapt. In this thesis, we identified several areas where recent improvements in storage hardware have made it essential to rethink the storage layer of database systems: Some improvements have exposed previously hidden bottlenecks, such as the software stack bottlenecking fast NVMe SSDs. Other innovations, such as PMem, have changed how one should build applications, making it essential to custom-tailor applications to make the most of their appealing new properties. In all cases, we must rethink how to move data between storage and memory devices. In this thesis, we have adapted the storage layer to use these advancements by introducing two new systems, *Mosaic* and *Plush*, and a vision, *Data Pipes*.

Mosaic is a storage engine and storage device purchase recommender for relational database systems and addresses the modern heterogeneous storage stack. *Plush*, a key-value store for PMem, demonstrates how applications can be re-designed for groundbreaking new storage hardware. Finally, Data Pipes are a vision of how to move data inside/between the storage stack and memory. Nevertheless, all contributions also come with limitations and, thus, opportunities for future work. While Mosaic already works in cloud settings (cf. Section 2.6), we did not initially design it for the cloud. The cloud offers unmatched elasticity: Resources can be allocated or dropped dynamically within seconds without high upfront costs. This elasticity prompts more and more customers to move their database systems into the cloud. This flexibility, however, comes at a cost, as cloud providers charge high hourly prices for this privilege. A cloud-native version of Mosaic could thus offer considerable benefits to customers running their database systems in the cloud. To this end, Mosaic should know that resources (i. e., storage devices) can be scaled instantly. Thus, placement decisions and storage device "renting" recommendations should be updated far more often. Furthermore, Mosaic's storage device model should be aware of the idiosyncrasies of cloud storage volumes, such as higher latency, lower IOPS, or credit models.

Plush is optimized for PMem, which has produced a considerable buzz in the research community but has not seen widespread adoption in the industry due to its high price compared to SSDs and the difficulty of programming for it. Furthermore, scaling is limited in practice since a CPU socket only supports 3 TiB of PMem at most. Intel has discontinued its PMem DIMM offerings but promised to release PMem modules for CXL [28], allowing cache-coherent access over PCIe and solving the scaling issue. While we do not believe that PMem will die as technology – since its low write latency is not replicable by any other technology – we probably will not see servers equipped with dozens of terabytes of PMem soon. In future work, we should thus extend Plush to scale to storing data to SSD once PMem is exhausted by putting its larger levels on SSD. Each directory entry currently points to up to 4 KiB of unsorted records. Plush could sort those during migration and write them to an SSD page. During future migrations, Plush could merge values into this sorted run which is the root of a conventional LSM tree. With growing data set size, more and more data would live in this LSM forest with performance gracefully declining to SSD speed.

Finally, *Data Pipes* are currently just a vision. While we have shown promising experiments in Chapter 4, much work must be done before Data Pipes are ready for production. We have shown and discussed approaches to implement Data Pipes and possible interfaces to interact with them, but we leave a full implementation for future work.

Bibliography

[1] Mohammad Alian et al. “IDIO: Network-Driven, Inbound Network Data Orchestration on Server Processors”. In: *MICRO*. 2022, pp. 480–493.

[2] Thomas E. Anderson et al. “Assise: Performance and Availability via Client-Local NVM in a Distributed File System”. In: *OSDI*. 2020.

[3] Christoph Anneser et al. “Programming Fully Disaggregated Systems”. In: *HotOS*. USENIX Association, 2023.

[4] Raja Appuswamy, David C. van Moolenbroek, and Andrew S. Tanenbaum. “Cache, cache everywhere, flushing all hits down the sink: On exclusivity in multilevel, hybrid caches”. In: *MSST*. IEEE Computer Society, 2013, pp. 1–14.

[5] Joy Arulraj et al. “BzTree: A High-Performance Latch-free Range Index for Non-Volatile Memory”. In: *PVLDB* 11.5 (2018), pp. 553–565.

[6] Manos Athanassoulis et al. “MaSM: efficient online updates in data warehouses”. In: *SIGMOD*. ACM, 2011, pp. 865–876.

[7] Berk Atikoglu et al. “Workload analysis of a large-scale key-value store”. In: *SIGMETRICS*. ACM, 2012, pp. 53–64.

[8] *AWS Lambda*. `https://aws.amazon.com/lambda/`. [accessed June 09, 2023].

[9] *Azure Virtual Machine series*. `https://azure.microsoft.com/en-us/pricing/details/virtual-machines/series/`. [accessed June 09, 2023].

[10] Antonio Barbalace and Jaeyoung Do. “Computational Storage: Where Are We Today?” In: *CIDR*. 2021, pp. 1–6.

[11] Rudolf Bayer and Edward M. McCreight. “Organization and Maintenance of Large Ordered Indices”. In: *Acta Informatica* 1 (1972), pp. 173–189.

[12] *bcache*. `https://bcache.evilpiepirate.org/`. [accessed June 09, 2023].

[13] Lawrence Benson, Hendrik Makait, and Tilmann Rabl. "Viper: An Efficient Hybrid PMem-DRAM Key-Value Store". In: *PVLDB* 14.9 (2021), pp. 1544–1556.

[14] Matias Bjørling et al. "ZNS: Avoiding the Block Interface Tax for Flash-based SSDs". In: *USENIX Annual Technical Conference*. USENIX Association, 2021, pp. 689–703.

[15] Maximilian Böther et al. "Drop It In Like It's Hot: An Analysis of Persistent Memory as a Drop-in Replacement for NVMe SSDs". In: *DaMoN*. ACM, 2021, 7:1–7:8.

[16] Jan Böttcher et al. "Scalable and robust latches for database systems". In: *DaMoN*. ACM, 2020, 2:1–2:8.

[17] Luc Bouganim, Björn Þór Jónsson, and Philippe Bonnet. "uFLIP: Understanding Flash IO Patterns". In: *CIDR*. www.cidrdb.org, 2009.

[18] Gerth Stølting Brodal and Rolf Fagerberg. "Lower bounds for external memory dictionaries". In: *SODA*. 2003, pp. 546–554.

[19] Darius Buntinas et al. "Cache-Efficient, Intranode, Large-Message MPI Communication with MPICH2-Nemesis". In: *ICPP*. IEEE Computer Society, 2009, pp. 462–469.

[20] Mustafa Canim et al. "SSD Bufferpool Extensions for Database Systems". In: *PVLDB* 3.2 (2010), pp. 1435–1446.

[21] Helen H. W. Chan et al. "HashKV: Enabling Efficient Updates in KV Storage via Hashing". In: *USENIX Annual Technical Conference*. USENIX Association, 2018, pp. 1007–1019.

[22] Badrish Chandramouli et al. "FASTER: A Concurrent Key-Value Store with In-Place Updates". In: *SIGMOD Conference*. ACM, 2018, pp. 275–290.

[23] Shimin Chen and Qin Jin. "Persistent B+-Trees in Non-Volatile Main Memory". In: *PVLDB* 8.7 (2015), pp. 786–797.

[24] Youmin Chen et al. "uTree: a Persistent B+-Tree with Low Tail Latency". In: *PVLDB* 13.11 (2020), pp. 2634–2648.

[25] Zhenke Chen et al. "RAMCI: a novel asynchronous memory copying mechanism based on I/OAT". In: *CCF Trans. High Perform. Comput.* 3.2 (2021), pp. 129–143.

[26] Yue Cheng et al. "CAST: Tiering Storage for Data Analytics in the Cloud". In: *HPDC*. ACM, 2015, pp. 45–56.

[27] Jungsik Choi, Jiwon Kim, and Hwansoo Han. "Efficient Memory Mapped File I/O for In-Memory File Systems". In: *HotStorage*. USENIX Association, 2017.

[28] *Compute Express Link.* `https://www.computeexpresslink.org/download-the-specification`. [accessed June 09, 2023]. 2023.

[29] J. Corbet. *The chained scatterlist API.* `https://lwn.net/Articles/256368/`. [accessed June 09, 2023]. 2007.

[30] Andrew Crotty, Viktor Leis, and Andrew Pavlo. "Are You Sure You Want to Use MMAP in Your Database Management System?" In: *CIDR 2022, Conference on Innovative Data Systems Research.* 2022.

[31] Björn Daase et al. "Maximizing Persistent Memory Bandwidth Utilization for OLAP Workloads". In: *SIGMOD Conference.* ACM, 2021, pp. 339–351.

[32] Jeffrey Dean and Sanjay Ghemawat. "MapReduce: simplified data processing on large clusters". In: *Commun. ACM* 51.1 (2008), pp. 107–113.

[33] Justin DeBrabant et al. "Anti-Caching: A New Approach to Database Management System Architecture". In: *PVLDB* 6.14 (2013), pp. 1942–1953.

[34] Henri Maxime Demoulin et al. "When Idling is Ideal: Optimizing Tail-Latency for Heavy-Tailed Datacenter Workloads with Perséphone". In: *SOSP.* ACM, 2021, pp. 621–637.

[35] Cristian Diaconu et al. "Hekaton: SQL server's memory-optimized OLTP engine". In: *SIGMOD.* ACM, 2013, pp. 1243–1254.

[36] Jialin Ding et al. "Instance-Optimized Data Layouts for Cloud Analytics Workloads". In: *SIGMOD.* 2021, pp. 418–431.

[37] Jaeyoung Do et al. "Turbocharging DBMS buffer pool using SSDs". In: *SIGMOD.* ACM, 2011, pp. 1113–1124.

[38] Ronald Fagin et al. "Extendible Hashing - A Fast Access Method for Dynamic Files". In: *ACM Trans. Database Syst.* 4.3 (1979), pp. 315–344.

[39] Franz Färber et al. "SAP HANA database: data management for modern business applications". In: *SIGMOD Record* 40.4 (2011), pp. 45–51.

[40] Alireza Farshin et al. "Reexamining Direct Cache Access to Optimize I/O Intensive Applications for Multi-hundred-gigabit Networks". In: *USENIX.* 2020, pp. 673–689.

[41] *Fast Just Got Faster: SATA 6Gb/s.* `https://sata-io.org/system/files/member-downloads/SATA-6Gbs-Fast-Just-Got-Faster_2.pdf`. [accessed June 09, 2023].

[42] Philip Werner Frey and Gustavo Alonso. "Minimizing the Hidden Cost of RDMA". In: *IEEE ICDCS.* 2009, pp. 553–560.

[43] *fsync manual page*. https://man7.org/linux/man-pages/man2/fsync.2.html. [accessed June 09, 2023].

[44] Sanjay Ghemawat and Jeff Dean. *LevelDB*. 2011. URL: https://github.com/google/leveldb.

[45] *Google Cloud Machine families resource and comparison guide* . https://cloud.google.com/compute/docs/machine-resource. [accessed June 09, 2023].

[46] Goetz Graefe. "Volcano - An Extensible and Parallel Query Evaluation System". In: *IEEE Trans. Knowl. Data Eng.* 6.1 (1994), pp. 120–135.

[47] Jim Gray et al. "Quickly Generating Billion-Record Synthetic Databases". In: *SIGMOD Conference*. ACM Press, 1994, pp. 243–252.

[48] NVM Express Work Group. *NVM Express Base Specification*. 2023. URL: https://nvmexpress.org/specification/nvm-express-base-specification/.

[49] Jorge Guerra et al. "Cost Effective Storage using Extent Based Dynamic Tiering". In: *FAST*. USENIX, 2011, pp. 273–286.

[50] Shashank Gugnani, Arjun Kashyap, and Xiaoyi Lu. "Understanding the Idiosyncrasies of Real Persistent Memory". In: *PVLDB* 14.4 (2020), pp. 626–639.

[51] Gurobi Optimization LLC. *Gurobi Optimizer Reference Manual*. 2019. URL: http://www.gurobi.com.

[52] Gabriel Haas, Michael Haubenschild, and Viktor Leis. "Exploiting Directly-Attached NVMe Arrays in DBMS". In: *CIDR*. www.cidrdb.org, 2020.

[53] Michael Haubenschild et al. "Rethinking Logging, Checkpoints, and Recovery for High-Performance Storage Engines". In: *SIGMOD Conference*. ACM, 2020, pp. 877–892.

[54] Herodotos Herodotou and Shivnath Babu. "Profiling, What-if Analysis, and Cost-based Optimization of MapReduce Programs". In: *PVLDB* 4.11 (2011), pp. 1111–1122.

[55] Herodotos Herodotou and Elena Kakoulli. "Automating Distributed Tiered Storage Management in Cluster Computing". In: *PVLDB* 13.1 (2019), pp. 43–56.

[56] Daokun Hu et al. "Persistent Memory Hash Indexes: An Experimental Evaluation". In: *PVLDB* 14.5 (2021), pp. 785–798.

[57] Jing Hu et al. "Parallel Multi-split Extendible Hashing for Persistent Memory". In: *ICPP*. ACM, 2021, 48:1–48:10.

[58] Chenchen Huang, Huiqi Hu, and Aoying Zhou. "BPTree: An Optimized Index with Batch Persistence on Optane DC PM". In: *DASFAA (3)*. Vol. 12683. Lecture Notes in Computer Science. Springer, 2021, pp. 478–486.

[59] Haoyu Huang and Shahram Ghandeharizadeh. "Nova-LSM: A Distributed, Component-based LSM-tree Key-value Store". In: *SIGMOD*. 2021, pp. 749–763.

[60] Kaisong Huang et al. "SSDs Striking Back: The Storage Jungle and Its Implications to Persistent Indexes". In: *CIDR*. www.cidrdb.org, 2022.

[61] Sai Huang et al. "Improving Flash-Based Disk Cache with Lazy Adaptive Replacement". In: *TOS* 12.2 (2016), 8:1–8:24.

[62] Deukyeon Hwang et al. "Endurable Transient Inconsistency in Byte-Addressable Persistent B+-Tree". In: *FAST*. USENIX Association, 2018, pp. 187–200.

[63] Stephen Ibanez et al. "The nanoPU: A Nanosecond Network Stack for Datacenters". In: *OSDI*. 2021, pp. 239–256.

[64] Ilias Iliadis et al. "ExaPlan: Efficient Queueing-Based Data Placement, Provisioning, and Load Balancing for Large Tiered Storage Systems". In: *TOS* 13.2 (2017), 17:1–17:41.

[65] Intel. *Intel Optane DC persIstent Memory Data sheet*. 2022. URL: `https://www.intel.de/content/dam/www/public/us/en/documents/product-briefs/optane-dc-persistent-memory-brief.pdf`.

[66] *Intel Data Direct I/O Technology*. `https://www.intel.ca/content/www/ca/en/io/data-direct-i-o-technology.html`. [accessed June 09, 2023].

[67] *Intel I/O Acceleration Technology*. `https://www.intel.ca/content/www/ca/en/wireless-network/accel-technology.html`. [accessed June 09, 2023].

[68] *io_uring*. `https://man.archlinux.org/man/io_uring.7.en`. [accessed June 09, 2023].

[69] Nusrat Sharmin Islam et al. "Triple-H: A Hybrid Approach to Accelerate HDFS on HPC Clusters with Heterogeneous Storage Architecture". In: *CCGRID*. IEEE Computer Society, 2015, pp. 101–110.

[70] Todor Ivanov and Matteo Pergolesi. "The impact of columnar file formats on SQL-on-hadoop engine performance: A study on ORC and Parquet". In: *Concurrency and Computation: Practice and Experience* 32.5 (2020).

[71] Joseph Izraelevitz et al. “Basic Performance Measurements of the Intel Optane DC Persistent Memory Module”. In: *CoRR* abs/1903.05714 (2019).

[72] Theo Jepsen et al. “From Sand to Flour: The Next Leap in Granular Computing with NanoSort”. In: *CoRR* abs/2204.12615 (2022).

[73] Zhiwen Jiang et al. “A Cost-aware Buffer Management Policy for Flash-based Storage Devices”. In: *DASFAA*. Vol. 9049. Lecture Notes in Computer Science. Springer, 2015, pp. 175–190.

[74] Peiquan Jin et al. “AD-LRU: An efficient buffer replacement algorithm for flash-based databases”. In: *Data Knowl. Eng.* 72 (2012), pp. 83–102.

[75] Hongshin Jun et al. “HBM (High Bandwidth Memory) DRAM Technology and Architecture”. In: *2017 IEEE International Memory Workshop (IMW)*. 2017, pp. 1–4.

[76] Olzhas Kaiyrakhmet et al. “SLM-DB: Single-Level Key-Value Store with Persistent Memory”. In: *FAST*. USENIX Association, 2019, pp. 191–205.

[77] Elena Kakoulli and Herodotos Herodotou. “OctopusFS: A Distributed File System with Tiered Storage Management”. In: *SIGMOD*. ACM, 2017, pp. 65–78.

[78] Anuj Kalia, David G. Andersen, and Michael Kaminsky. “Challenges and solutions for fast remote persistent memory access”. In: *SoCC*. 2020, pp. 105–119.

[79] Jeong-Uk Kang et al. “The Multi-streamed Solid-State Drive”. In: *HotStorage*. USENIX Association, 2014.

[80] Woon-Hak Kang, Sang-Won Lee, and Bongki Moon. “Flash-based Extended Cache for Higher Throughput and Faster Recovery”. In: *PVLDB* 5.11 (2012), pp. 1615–1626.

[81] Sudarsun Kannan et al. “Redesigning LSMs for Nonvolatile Memory with NoveLSM”. In: *USENIX Annual Technical Conference*. USENIX Association, 2018, pp. 993–1005.

[82] Kaan Kara et al. “High Bandwidth Memory on FPGAs: A Data Analytics Perspective”. In: *FPL*. IEEE, 2020, pp. 1–8.

[83] Alexey Karyakin and Kenneth Salem. “DimmStore: Memory Power Optimization for Database Systems”. In: *PVLDB* 12.11 (July 2019), pp. 1499–1512.

[84] Alfons Kemper and Thomas Neumann. “HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots”. In: *ICDE*. IEEE, 2011, pp. 195–206.

[85] Jongyul Kim et al. “LineFS: Efficient SmartNIC Offload of a Distributed File System with Pipeline Parallelism”. In: *SOSP*. 2021, pp. 756–771.

[86] Reika Kinoshita et al. “Cost-Performance Evaluation of Heterogeneous Tierless Storage Management in a Public Cloud”. In: *CANDAR*. IEEE, 2021, pp. 121–126.

[87] Viktor Leis, Michael Haubenschild, and Thomas Neumann. “Optimistic Lock Coupling: A Scalable and Efficient General-Purpose Synchronization Method”. In: *IEEE Data Eng. Bull.* 42.1 (2019), pp. 73–84.

[88] Viktor Leis et al. “How Good Are Query Optimizers, Really?” In: *PVLDB* 9.3 (2015), pp. 204–215.

[89] Viktor Leis et al. “LeanStore: In-Memory Data Management beyond Main Memory”. In: *ICDE*. IEEE, 2018, pp. 185–196.

[90] Alberto Lerner and Philippe Bonnet. “Not your Grandpa’s SSD: The Era of Co-Designed Storage Devices”. In: *SIGMOD*. 2021, pp. 2852–2858.

[91] Lucas Lersch, Wolfgang Lehner, and Ismail Oukid. “Persistent Buffer Management with Optimistic Consistency”. In: *DaMoN*. ACM, 2019, 14:1–14:3.

[92] Lucas Lersch et al. “Evaluating Persistent Memory Range Indexes”. In: *PVLDB* 13.4 (2019), pp. 574–587.

[93] Felix Xiaozhu Lin and Xu Liu. “*memif*: Towards Programming Heterogeneous Memory Asynchronously”. In: *ASPLOS*. ACM, 2016, pp. 369–383.

[94] Haodong Lin et al. “DRAM Cache Management with Request Granularity for NAND-based SSDs”. In: *ICPP*. ACM, 2022, 29:1–29:10.

[95] Haikun Liu et al. “A Survey of Non-Volatile Main Memory Technologies: State-of-the-Arts, Practices, and Future Directions”. In: *J. Comput. Sci. Technol.* 36.1 (2021), pp. 4–32.

[96] Hao Liu et al. “LibreKV: A Persistent in-Memory Key-Value Store”. In: *IEEE Trans. Emerg. Top. Comput.* 8.4 (2020), pp. 916–927.

[97] Jihang Liu, Shimin Chen, and Lujun Wang. “LB+-Trees: Optimizing Persistent Index Performance on 3DXPoint Memory”. In: *PVLDB* 13.7 (2020), pp. 1078–1090.

[98] Xin Liu and Kenneth Salem. “Hybrid Storage Management for Database Systems”. In: *PVLDB* 6.8 (2013), pp. 541–552.

[99] Baotong Lu et al. “APEX: A High-Performance Learned Index on Persistent Memory”. In: *PVLDB* 15.3 (2021), pp. 597–610.

[100] Baotong Lu et al. “Dash: Scalable Hashing on Persistent Memory”. In: *PVLDB* 13.8 (2020), pp. 1147–1161.

[101] Lanyue Lu et al. “WiscKey: Separating Keys from Values in SSD-Conscious Storage”. In: *ACM Trans. Storage* 13.1 (2017), 5:1–5:28.

[102] Simon AF Lund et al. “I/O interface independence with xNVMe”. In: *Proceedings of the 15th ACM International Conference on Systems and Storage*. 2022, pp. 108–119.

[103] Chen Luo et al. “Umzi: Unified Multi-Zone Indexing for Large-Scale HTAP”. In: *EDBT*. 2019, pp. 1–12.

[104] Tian Luo et al. “hStorage-DB: Heterogeneity-aware Data Management to Exploit the Full Capability of Hybrid Storage Systems”. In: *PVLDB* 5.10 (2012), pp. 1076–1087.

[105] Shaonan Ma et al. “ROART: Range-query Optimized Persistent ART”. In: *FAST*. USENIX Association, 2021, pp. 1–16.

[106] John D. McCalpin. *STREAM: Sustainable Memory Bandwidth in High Performance Computers*. `https://www.cs.virginia.edu/stream/`. [accessed June 09, 2023].

[107] Michael Mesnier et al. “Differentiated Storage Services”. In: *SOSP*. 2011, pp. 57–70.

[108] *mmap manual page*. `https://man7.org/linux/man-pages/man2/mmap.2.html`. [accessed June 09, 2023].

[109] Moohyeon Nam et al. “Write-Optimized Dynamic Hashing for Persistent Memory”. In: *FAST*. USENIX Association, 2019, pp. 31–44.

[110] Thomas Neumann. “Efficiently Compiling Efficient Query Plans for Modern Hardware”. In: *PVLDB* 4.9 (2011), pp. 539–550.

[111] Thomas Neumann. “Evolution of a Compiling Query Engine”. In: *PVLDB* 14.12 (2021), pp. 3207–3210.

[112] Thomas Neumann and Michael J. Freitag. “Umbra: A Disk-Based System with In-Memory Performance”. In: *CIDR*. www.cidrdb.org, 2020.

[113] Khang T Nguyen. *Introduction to Cache Allocation Technology in the Intel® Xeon® Processor E5 v4 Family*. URL: `https://www.intel.com/content/www/us/en/developer/articles/technical/introduction-to-cache-allocation-technology.html`.

[114] NVM Express. *Everything You Need to Know About the NVMe® 2.0 Specifications and New Technical Proposals*. 2022.

[115] Patrick E. O'Neil et al. "The Log-Structured Merge-Tree (LSM-Tree)". In: *Acta Informatica* 33.4 (1996), pp. 351–385.

[116] Kazuichi Oe and Koji Okamura. "A Hybrid Storage System Composed of On-the-Fly Automated Storage Tiering (OTF-AST) and Caching". In: *CANDAR*. IEEE, 2014, pp. 406–411.

[117] *Oracle Cloud Infrastructure Compute Shapes*. `https://docs.oracle.com/en-us/iaas/Content/Compute/References/computeshapes.htm`. [accessed June 09, 2023].

[118] Ismail Oukid et al. "FPTree: A Hybrid SCM-DRAM Persistent and Concurrent B-Tree for Storage Class Memory". In: *SIGMOD Conference*. ACM, 2016, pp. 371–386.

[119] Ismail Oukid et al. "FPTree: A Hybrid SCM-DRAM Persistent and Concurrent B-Tree for Storage Class Memory". In: *SIGMOD*. 2016, pp. 371–386.

[120] G. N. Paulley and Per-Åke Larson. "Exploiting Uniqueness in Query Optimization". In: *CASCON*. IBM Press, 1993, pp. 804–822.

[121] Simon Peter et al. "Arrakis: The Operating System Is the Control Plane". In: *ACM Trans. Comput. Syst.* 33.4 (2016), 11:1–11:30.

[122] *PMem-RocksDB*. `https://github.com/pmem/pmem-rocksdb`. [accessed June 09, 2023].

[123] *PMemKV*. `https://pmem.io/pmemkv/`. [accessed June 09, 2023].

[124] *pread/pwrite manual page*. `https://man7.org/linux/man-pages/man2/pread.2.html`. [accessed June 09, 2023].

[125] Felix Putze, Peter Sanders, and Johannes Singler. "Cache-, hash-, and space-efficient bloom filters". In: *ACM J. Exp. Algorithmics* 14 (2009).

[126] Krish K. R., Ali Anwar, and Ali Raza Butt. "hatS: A Heterogeneity-Aware Tiered Storage for Hadoop". In: *CCGRID*. IEEE, 2014, pp. 502–511.

[127] Krish K. R., M. Safdar Iqbal, and Ali Raza Butt. "VENU: Orchestrating SSDs in hadoop storage". In: *BigData*. IEEE, 2014, pp. 207–212.

[128] Krish K. R. et al. "On Efficient Hierarchical Storage for Big Data Processing". In: *CCGrid*. IEEE, 2016, pp. 403–408.

[129] Amanda Raybuck et al. "HeMem: Scalable Tiered Memory Management for Big Data Applications and Real NVM". In: *SOSP*. 2021, pp. 392–407.

[130] *Redis*. `https://redis.io/`. [accessed June 09, 2023].

[131] Alexander van Renen et al. "Building blocks for persistent memory". In: *VLDB J.* 29.6 (2020), pp. 1223–1241.

[132] Alexander van Renen et al. “Managing Non-Volatile Memory in Database Systems”. In: *SIGMOD*. ACM, 2018, pp. 1541–1555.

[133] Alexander van Renen et al. “Persistent Memory I/O Primitives”. In: *DaMoN*. ACM, 2019, 12:1–12:7.

[134] Alice Rey, Michael Freitag, and Thomas Neumann. “Seamless Integration of Parquet Files into Data Processing”. In: *BTW*. Vol. P-331. LNI. Gesellschaft für Informatik e.V., 2023, pp. 235–258.

[135] David Reinsel-John Gantz-John Rydning, J Reinsel, and J Gantz. “The digitization of the world from edge to core”. In: *Framingham: International Data Corporation* 16 (2018).

[136] Subhadeep Sarkar and Manos Athanassoulis. “Dissecting, Designing, and Optimizing LSM-based Data Stores”. In: *SIGMOD*. 2022, pp. 2489–2497.

[137] Tobias Schmidt, Maximilian Bandle, and Jana Giceva. “A four-dimensional Analysis of Partitioned Approximate Filters”. In: *PVLDB* 14.11 (2021), pp. 2355–2368.

[138] Russell Sears and Raghu Ramakrishnan. “bLSM: a general purpose log structured merge tree”. In: *SIGMOD Conference*. ACM, 2012, pp. 217–228.

[139] Galen M. Shipman et al. “Early Performance Results on 4th Gen Intel(R) Xeon (R) Scalable Processors with DDR and Intel(R) Xeon(R) processors, codenamed Sapphire Rapids with HBM”. In: *CoRR* abs/2211.05712 (2022).

[140] Konstantin Shvachko et al. “The Hadoop Distributed File System”. In: *MSST*. IEEE, 2010, pp. 1–10.

[141] Moritz Sichert and Thomas Neumann. “User-Defined Operators: Efficiently Integrating Custom Algorithms into Modern Databases”. In: *PVLDB* 15.5 (2022), pp. 1119–1131.

[142] *SPDK: Acceleration Framework*. `https://spdk.io/doc/accel_fw.html`. [accessed June 09, 2023].

[143] Radu Stoica and Anastasia Ailamaki. “Enabling efficient OS paging for main-memory OLTP databases”. In: *DaMoN*. ACM, 2013, p. 7.

[144] Lasse Thostrup et al. “DFI: The Data Flow Interface for High-Speed Networks”. In: *SIGMOD Conference*. ACM, 2021, pp. 1825–1837.

[145] Cristian Ungureanu et al. “TBF: A memory-efficient replacement policy for flash-based caches”. In: *ICDE*. IEEE, 2013, pp. 1117–1128.

[146] Karthikeyan Vaidyanathan and Dhabaleswar K. Panda. “Benefits of I/O Acceleration Technology (I/OAT) in Clusters”. In: *ISPASS*. IEEE Computer Society, 2007, pp. 220–229.

[147] Karthikeyan Vaidyanathan et al. "Designing Efficient Asynchronous Memory Operations Using Hardware Copy Engine: A Case Study with I/OAT". In: *IPDPS*. IEEE, 2007, pp. 1–8.

[148] Lukas Vogel et al. "Data Pipes: Declarative Control over Data Movement". In: *CIDR*. www.cidrdb.org, 2023.

[149] Lukas Vogel et al. "Mosaic: A Budget-Conscious Storage Engine for Relational Database Systems". In: *PVLDB* 13.11 (2020), pp. 2662–2675.

[150] Lukas Vogel et al. "Plush: A Write-Optimized Persistent Log-Structured Hash-Table". In: *PVLDB* 15.11 (2022), pp. 2895–2907.

[151] Haris Volos, Andres Jaan Tack, and Michael M. Swift. "Mnemosyne: lightweight persistent memory". In: *ASPLOS*. ACM, 2011, pp. 91–104.

[152] Midhul Vuppalapati et al. "Building An Elastic Query Engine on Disaggregated Storage". In: *NSDI*. USENIX Association, 2020, pp. 449–462.

[153] Guanying Wang et al. "A simulation approach to evaluating design decisions in MapReduce setups". In: *MASCOTS*. IEEE, 2009, pp. 1–11.

[154] Hui Wang and Peter J. Varman. "Balancing fairness and efficiency in tiered storage systems with bottleneck-aware allocation". In: *FAST*. USENIX, 2014, pp. 229–242.

[155] Christian Winter et al. "On-Demand State Separation for Cloud Data Warehousing". In: *PVLDB* 15.11 (2022), pp. 2966–2979.

[156] Youngjoo Woo et al. "Analysis and Optimization of Persistent Memory Index Structures' Write Amplification". In: *IEEE Access* 9 (2021), pp. 167687–167698.

[157] XiaoJian Wu and A. L. Narasimha Reddy. "Exploiting Concurrency to Improve Latency and throughput in a Hybrid Storage System". In: *MASCOTS*. IEEE, 2010, pp. 14–23.

[158] Giorgos Xanthakis et al. "Parallax: Hybrid Key-Value Placement in LSM-based Key-Value Stores". In: *SoCC*. ACM, 2021, pp. 305–318.

[159] Fei Xia et al. "HiKV: A Hybrid Index Key-Value Store for DRAM-NVM Memory Systems". In: *USENIX Annual Technical Conference*. USENIX Association, 2017, pp. 349–362.

[160] Renzhi Xiao et al. "Write-Optimized and Consistent Skiplists for Non-Volatile Memory". In: *IEEE Access* 9 (2021), pp. 69850–69859.

[161] Baoyue Yan et al. "Revisiting the Design of LSM-tree Based OLTP Storage Engine with Persistent Memory". In: *PVLDB* 14.10 (2021), pp. 1872–1885.

[162] Jian Yang et al. “An Empirical Guide to the Behavior and Use of Scalable Persistent Memory”. In: *USENIX FAST*. 2020, pp. 169–182.

[163] Jun Yang et al. “NV-Tree: A Consistent and Workload-Adaptive Tree Structure for Non-Volatile Memory”. In: *IEEE Trans. Computers* 65.7 (2016), pp. 2169–2183.

[164] Yifan Yuan et al. “Don’t Forget the I/O When Allocating Your LLC”. In: *ISCA*. 2021, pp. 112–125.

[165] Matei Zaharia et al. “Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing”. In: *NSDI*. USENIX, 2012, pp. 15–28.

[166] Baoquan Zhang and David H. C. Du. “NVLSM: A Persistent Memory Key-Value Store Using Log-Structured Merge Tree with Accumulative Compaction”. In: *ACM Trans. Storage* 17.3 (2021), 23:1–23:26.

[167] Gong Zhang et al. “Automated lookahead data migration in SSD-enabled multi-tiered storage systems”. In: *MSST*. IEEE, 2010, pp. 1–6.

[168] Irene Zhang et al. “The Demikernel Datapath OS Architecture for Microsecond-scale Datacenter Systems”. In: *SOSP*. ACM, 2021, pp. 195–211.

[169] Xinjing Zhou et al. “DPTree: Differential Indexing for Persistent Memory”. In: *PVLDB* 13.4 (2019), pp. 421–434.

[170] Xinjing Zhou et al. “Spitfire: A Three-Tier Buffer Manager for Volatile and Non-Volatile Memory”. In: *SIGMOD Conference*. ACM, 2021, pp. 2195–2207.

[171] *zstd*. `https://facebook.github.io/zstd/`. [accessed June 09, 2023].

[172] Pengfei Zuo, Yu Hua, and Jie Wu. “Write-Optimized and High-Performance Hashing Index Scheme for Persistent Memory”. In: *OSDI*. USENIX Association, 2018, pp. 461–476.