Technische Universität Ilmenau

# Transactional and Analytical Data Management on Persistent Memory

**Dissertation**

zur Erlangung des akademischen Grades
Doktoringenieur (Dr.-Ing.)

vorgelegt an der

Technischen Universität Ilmenau
Fakultät für Informatik und Automatisierung
Fachgebiet Datenbanken und Informationssysteme

eingereicht von

**Philipp Götze, M. Sc.**
geboren am 2. Februar 1991 in Strausberg

**Gutachter:** **Prof. Dr.-Ing. habil. Kai-Uwe Sattler**
Technische Universität Ilmenau

**Prof. Dr.-Ing. habil. Alfons Kemper**
Technische Universität München

**Prof. Dr.-Ing. habil. Bernhard Seeger**
Philipps-Universität Marburg

Tag der Einreichung: 02. November 2021
Tag der wissenschaftlichen Aussprache: 08. April 2022

DOI: 10.22032/dbt.51870
URN: urn:nbn:de:gbv:ilm1-2022000119

# Abstract

The increasing number of smart devices and sensors, but also social media are causing the volume of data and thus the demanded processing speed to grow steadily. At the same time, many applications need to store data persistently or even comply with strict transactional guarantees. The novel storage technology Persistent Memory (PMem), with its unique properties, seems to be a natural candidate to meet these requirements efficiently. Compared to DRAM, it is more scalable, less expensive, and durable. In contrast to disks, it is significantly faster and directly addressable.

Therefore, this dissertation investigates the deliberate employment of PMem to fit the needs of modern applications. After presenting the fundamental work of and with PMem, we focus primarily on three aspects of data management. First, we disassemble several persistent data and index structures into their underlying design primitives to reveal the trade-offs for various access patterns. It allows us to identify their best use cases and vulnerabilities but also to gain general insights into the design of PMem-based data structures. Second, we propose two storage layouts that target analytical workloads and enable an efficient query execution on arbitrary attributes. While the first approach employs a linked list of multi-dimensional clustered blocks that potentially span several storage layers, the second approach is a multi-dimensional index that caches nodes in DRAM. Third, we show how to improve stream and event processing systems involving transactional state management using the preceding data structures and insights. In this context, we propose a novel Transactional Stream Processing (TSP) model with appropriate consistency and concurrency protocols adapted to PMem. Together, the discussed aspects are intended to provide a foundation for developing even more sophisticated PMem-enabled systems. At the same time, they show how data management tasks can take advantage of PMem by opening up new application domains, improving performance, scalability, and recovery guarantees, simplifying code complexity, plus reducing economic and environmental costs.

# Zusammenfassung

Die zunehmende Anzahl von Smart-Geräten und Sensoren, aber auch die sozialen Medien lassen das Datenvolumen und damit die geforderte Verarbeitungsgeschwindigkeit stetig wachsen. Gleichzeitig müssen viele Anwendungen Daten persistent speichern oder sogar strenge Transaktionsgarantien einhalten. Die neuartige Speichertechnologie Persistent Memory (PMem) mit ihren einzigartigen Eigenschaften scheint ein natürlicher Anwärter zu sein, um diesen Anforderungen effizient nachzukommen. Sie ist im Vergleich zu DRAM skalierbarer, günstiger und dauerhaft. Im Gegensatz zu Disks ist sie deutlich schneller und direkt adressierbar.

Daher wird in dieser Dissertation der gezielte Einsatz von PMem untersucht, um den Anforderungen moderner Anwendung gerecht zu werden. Nach der Darlegung der grundlegenden Arbeitsweise von und mit PMem, konzentrieren wir uns primär auf drei Aspekte der Datenverwaltung. Zunächst zerlegen wir mehrere persistente Daten- und Indexstrukturen in ihre zugrundeliegenden Entwurfsprimitive, um Abwägungen für verschiedene Zugriffsmuster aufzuzeigen. So können wir ihre besten Anwendungsfälle und Schwachstellen, aber auch allgemeine Erkenntnisse über das Entwerfen von PMem-basierten Datenstrukturen ermitteln. Zweitens schlagen wir zwei Speicherlayouts vor, die auf analytische Arbeitslasten abzielen und eine effiziente Abfrageausführung auf beliebigen Attributen ermöglichen. Während der erste Ansatz eine verknüpfte Liste von mehrdimensionalen gruppierten Blöcken verwendet, handelt es sich beim zweiten Ansatz um einen mehrdimensionalen Index, der Knoten im DRAM zwischenspeichert. Drittens zeigen wir unter Verwendung der bisherigen Datenstrukturen und Erkenntnisse, wie Datenstrom- und Ereignisverarbeitungssysteme mit transaktionaler Zustandsverwaltung verbessert werden können. Dabei schlagen wir ein neuartiges Transactional Stream Processing (TSP) Modell mit geeigneten Konsistenz- und Nebenläufigkeitsprotokollen vor, die an PMem angepasst sind. Zusammen sollen die diskutierten Aspekte eine Grundlage für die Entwicklung noch ausgereifterer PMem-fähiger Systeme bilden. Gleichzeitig zeigen sie, wie Datenverwaltungsaufgaben PMem ausnutzen können, indem sie neue Anwendungsgebiete erschließen, die Leistung, Skalierbarkeit und Wiederherstellungsgarantien verbessern, die Codekomplexität vereinfachen sowie die ökonomischen und ökologischen Kosten reduzieren.

# Danksagung

Zunächst möchte ich hervorheben, dass diese Arbeit zwar nur mich als Autor ausweist, dass sie aber ohne die Unterstützung vieler weiterer Personen in dieser Form nicht möglich gewesen wäre. Ich möchte allen Kollegen, Co-Autoren und auch Studierenden, die mich bei meinen Forschungen unterstützt haben und mit denen ich zusammenarbeiten durfte, meine tiefe Dankbarkeit aussprechen.

In erster Linie möchte ich meinem Doktorvater Kai-Uwe Sattler für seine Unterstützung während meiner nahezu gesamten Zeit an der TU Ilmenau danken. Ursprünglich war eine Promotion meinerseits nicht geplant gewesen. Kai hatte mich nach Abschluss meiner Masterthesis dann dazu ermutigt diesen Schritt weiterzugehen. Dadurch hatte ich die Gelegenheit an vielen spannenden Projekten zu arbeiten und eine Vielzahl interessanter Menschen kennenzulernen. Tatsächlich wären einige dieser Projekte ohne seine Unterstützung, auch auf praktischer Seite, in diesem Umfang nicht möglich gewesen. Auch bin ich sehr dankbar für die stetigen Freiräume zur eigenständigen Forschung und Entwicklung.

Weiterhin bedanke ich mich bei meinen Gutachtern Alfons Kemper und Bernhard Seeger für ihre Zeit und ihren Aufwand, die sie in meine Arbeit investiert haben, sowie für die Erstellung der Gutachten. Durch gemeinsame Projekte oder der Teilnahme an Workshops und Konferenzen konnten wir über die letzten Jahre auch bereits einige Anregungen austauschen und aufschlussreiche Diskussionen führen.

Darüber hinaus danke ich nochmal ausdrücklich all meinen (auch ehemaligen) Kollegen für die ergiebigen fachlichen Diskussionen, aber auch für die unterhaltsamen Mittagspausen, Kaffeerunden und Betriebsausflüge. Besonderer Dank gilt Stefan, der all die Jahre fast täglich meine Sprüche im Büro ertragen musste. Gerade unsere musikalischen Untermalungen haben so manche stressigen oder tristen Phasen stark entlastet. Auch danke ich Jens und Matthias für die stets hilfreiche technische Betreuung.

Die vorliegende Arbeit entstand im Rahmen des 2017 gestarteten Schwerpunktprogramms "Scalable Data Management for Future Hardware" (SPP 2037), gefördert durch die Deutsche Forschungsgemeinschaft (DFG). Innerhalb dieser Forschungsgruppe konnten wir viele Erkenntnisse gewinnen und austauschen, besonders in den Breakout-Sessions oder während der Social-Events. Dies führte zu einigen Kooperationen und gemeinsamen Veröffentlichungen, welche mir viel Vergnügen bereitet haben.

Ich möchte zudem meinen Freunden, sowohl nah als auch fern, danken, die mich während dieser Zeit unterstützt und auf andere Gedanken gebracht haben. Explizit gilt mein Dank Alexander und Antonia sowie deren Kindern. Vor allem unser korrespondierender Sinn für Humor, aber auch die Kinderbespaßung hat mir immer eine Menge Heiterkeit beschert. Auch danke ich insbesondere Christoph für die anfängliche gesellschaftliche Eingliederung in Ilmenau sowie die zahlreichen Spieleabende, Wandertouren und anderen diversen Unternehmungen. Schließlich möchte ich mich selbstverständlich zutiefst bei meiner Familie bedanken, die immer an mich geglaubt hat, mich bedingungslos unterstützt und motiviert hat, und auch in stressigen Zeiten nachsichtig mit mir war. Vielen Dank für alles!

Philipp Götze
Ilmenau, April 25, 2022

# Contents

# 1 Introduction

Alongside the regular operation through transactional systems, it is often essential for companies to run real-time analyses on their data to make strategically sound and rapid decisions. This trend is also evident in the ever more frequent emergence of Hybrid Transactional/Analytical Processing (HTAP) systems such as SAP HANA [FCP+11], HyPer [KN11], Hyrise [DKB+19], or SingleStore [Sin20]. Their feasibility, however, also depends heavily on the hardware. As the quantity of sensors, smart devices, and other data sources continues to grow, the abundance and heterogeneity of data are increasing as well. That, in turn, places enormous demands on the necessary processing speed. Existing Database Management System (DBMS) designs can quickly reach their limits here. On the one hand, conventional disk-based systems that use DRAM as buffer are unlikely to achieve the required latencies due to their long I/O path, which the architecture inherently assumes. Alternatively, more recent in-memory systems are conceivable that maintain the entirety of the data in DRAM and ensure persistence, e.g., by logging or snapshotting. The issues with this implementation are first, the cost of the vast amount of DRAM required and second, the density of DRAM, which has already reached its scalability limit. At the same time, heterogeneous data sources combined with stringent processing requirements also call for novel processing models. Thus, application domains with naturally unbounded data sources such as cyber-physical systems, Internet of Things (IoT), Industry 4.0, streaming graphs, streaming data warehouses, cloud services, and others have recently emerged. To meet the size, speed, heterogeneity, and other challenges, these domains expedite the coalescence of high-speed data stream processing with traditional transactional and analytical processing.

The various data sources can thus be present either as a stream or as previously stored data. An elucidatory example is to combine real-time sensor data streams with historical or specification data to detect anomalies. Apart from the direct processing of data streams in operator nodes, the streams can also be interpreted as sequences of inserts, updates, or other modifications to a DBMS. Figure 1.1 illustrates such a scenario in a simplified form, which in turn contains several use cases. It bases on smart metering, where the energy consumption of various devices is measured. We can assume one or multiple tables containing specification data about cautionary or critical device conditions. On these tables, well-known transactional or analytical queries can be executed using SQL ❶. When introducing streams, a lot more applications and use cases are feasible. Stream data is normally processed with continuous queries, whose operators can

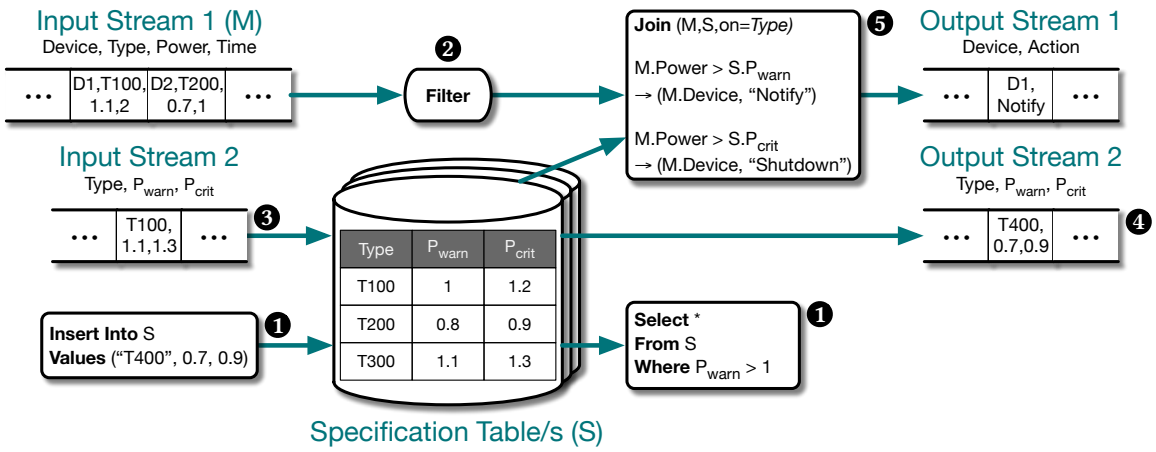



**Figure 1.1:** Simplified scenario mixing streams, tables, and queries. Adapted from [BFKT12].

be stateful or stateless ❷. Tables and states can be treated similarly or even interchangeably when bringing both concepts together. Thus, another use case is that the specifications are not extended or manipulated via ad-hoc queries, but the data arrives via streams ❸. In addition, data streams may also need to be derived from table modifications ❹. A more complex case is the joining of stream and table data since both use different query concepts (continuous vs. ad-hoc queries). In our example, the specification could be provided as a table while the measurement data arrives as a stream. The stream data would have to be combined with the stored data to generate alerts or the like ❺.

These modern scenarios, which we classify as stateful stream processing applications, implicate some requirements. First and foremost is the fast processing and thus a low latency since results are often expected in real time (as in the example above generating alarms). For integrity, statistical, historical, or similar reasons, it is also imperative to persist the data. Especially when there are integrity constraints, it requires compliance with Atomicity, Consistency, Isolation, Durability (ACID) guarantees as known from the database field. It means traditional concurrency and consistency protocols have to be redesigned and reevaluated for such scenarios. Apart from the non-functional requirements, there are additionally rather practical or functional demands. Depending on the workload on the states or tables, the requirements for the underlying data structures also vary. We can broadly distinguish between more read-intensive analytical and more write-focused transactional designs. Moreover, interfaces that determine how tables and streams can interact with each other are necessary. As highlighted in [CFKK20], future data management systems – pre-eminently stream processing systems – should be further enhanced by new features such as shared queryable states, cross-state versioning, and modern hardware support, among others.

Addressing the latter mentioned feature directly, current developments on the hardware level offer new possibilities to cope with these requirements. New storage and memory solutions are of primary interest here to close the access gap between them. In particular, Persistent Memory (PMem) seems to be a natural candidate to tackle both the low latency and fast durability requirements since it provides byte-addressability and direct persistence at near-DRAM speed. Figure 1.2 shows the memory and storage hierarchy of modern computer architectures and where PMem fits in. The large gap between memory and storage technologies stems from the monetary costs, performance, and density. Since PMem should fill this gap, researchers

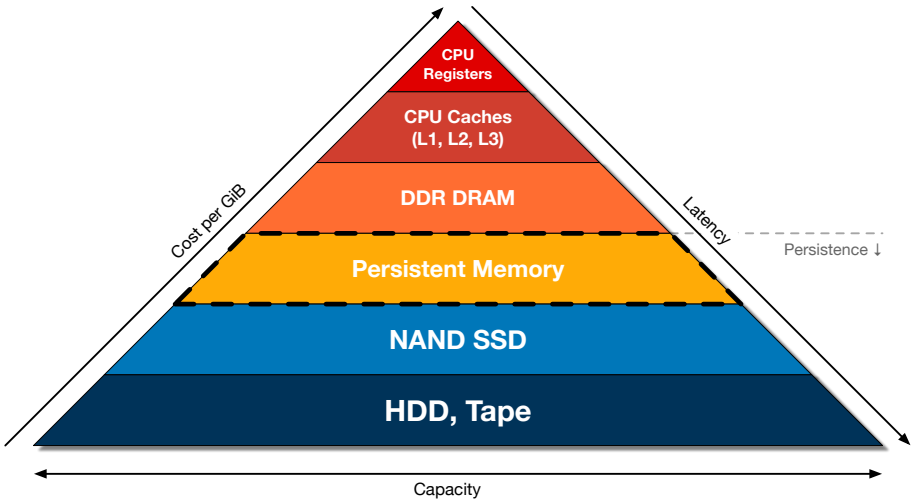



**Figure 1.2:** Memory and Storage Hierarchy.

sometimes refer to it as storage-class memory (SCM). As symbolized by the coloring of the figure, we can now distinguish warm data in addition to the typical hot and cold data. That would be applicable, for example, for a three-layer buffer manager or cache system. However, other integrations and individual solutions such as PMem-centric or hybrid data structures and engines are also conceivable, which we want to explore in this work.

Considering the modern data management and application requirements, it becomes clear that offline processing of large data volumes is not suitable anymore for current and future business needs. Instead, it rather requires a system that can process data and answer queries in real time while fulfilling transaction guarantees. However, using (only) block-oriented devices such as SSDs or HDDs to meet the durability guarantees would lead to extensive logging or snapshotting overhead, compromising the real-time requirement. PMem, on the other hand, with its latency close to DRAM and simultaneous persistence, seems to resolve this conflict. Therefore, we envisage Transactional Stream Processing (TSP) with PMem-aware states as a novel processing model to meet the above requirements. Such a model could cover all the use cases outlined above in one system. In addition to real-time performance and more efficient persistence, PMem can yield further benefits such as instant recovery, no or less data loss, and code simplifications. Of course, PMem is not a panacea and also entails its downsides and challenges. Identifying and addressing these will be among the endeavors of this work.

## 1.1 Problem Statement and Objectives

We have outlined above the requirements of modern data management scenarios and how we envision dealing with them. As we enter uncharted territory with PMem, new challenges arise that are not present in the classic DRAM-disk environment. The main questions we want to tackle are:

- How can PMem be exploited, and what role does it play in conjunction with the other technologies in the memory and storage hierarchy?

- How must existing algorithms, data structures, and systems be adapted to work duly with PMem while meeting the requirements of transactional data management?

These questions are not straightforward to answer. Especially particular properties and the integration into the hardware landscape require a rethinking of certain aspects such as failure atomicity. This dissertation pursues four objectives derived from these questions:

**O1** *Exploring Architectural Patterns for PMem-based Data Management.* Since PMem is a new component in the memory and storage hierarchy, as shown above, its primary and most profitable application has not been fully explored yet. As its properties allow it to perform both the memory and the storage layer tasks, a single-level system would be feasible and possibly very cost-effective. The simultaneous exploitation of several levels and the respective advantages of the technologies are also highly promising. Consequently, multi-level data structures, data management, and caching strategies would be worth exploring. Hence, the goal is to investigate various patterns of PMem and data placements crossing multiple levels in the memory hierarchy.

**O2** *Design and Analysis of Data and Index Structures.* Especially concerning our targeted TSP model – also for data management systems in general – stateful data processing cannot do without corresponding efficient data structures. Primarily, there need to be universal adjustments to account for the unique properties of PMem. In addition, optimization measures depending on the workload and access patterns of the queries must also be considered. Apart from transactional data structures, it is thus also interesting to take a look at adaptations for analytical scenarios.

**O3** *Specification and Implementation of a Unified TSP Model.* To combine data streams and stored data and enrich them with transactional guarantees, their linkage and the semantics of the operators need to be defined beforehand. An important aspect is also the interpretation of transaction boundaries for unbounded data streams. Furthermore, appropriate integration of the developed data structures for, e.g., state representations is necessary. The final goal is to implement and evaluate this model as a prototype in a data stream processing system.

**O4** *Evaluation of PMem-optimized Data Structures and Algorithms on Real Hardware.* Finally, we aim to support all our approaches and statements about them with experimental evaluations. In particular, we plan to provide a set of micro-benchmarks applicable to a multitude of data structures. Since we have real PMem hardware available, we expect to contribute meaningful and reusable results.

## 1.2 Contributions and Outline

The investigated issues regarding data management with PMem form different building blocks from the hardware level to the application layer. In the following, we give an overview of the contributions of this dissertation. We have tagged the objectives they address inline.

### Persistent Memory Access & Data Management

We start with the introduction of the different PMem technologies and the properties that they entail. These are, for instance, the near-DRAM latency, read-write asymmetry, and byte-addressability. Especially the last property changes the access model compared to other persistent media. Furthermore, we show various integration possibilities of this new technology in the hardware landscape (O1). These fundamentals give rise to several challenges for data management, which we resolve in the course of the remaining chapters. The material presented for this contribution was peer-reviewed and published in:

[GvRL+18] Philipp Götze, Alexander van Renen, Lucas Lersch, Viktor Leis, and Ismail Oukid. Data Management on Non-Volatile Memory: A Perspective. *Datenbank-Spektrum*, 18(3):171–182, https://doi.org/10.1007/s13222-018-0301-1, 2018

### Persistent Index Structures

Building on the fundamentals and general data management challenges, it is possible to design data structures optimized for PMem (O2). While examining existing PMem-based index and data structures, we found that their comparison does not always allow accurate conclusions about design decisions. Most proposals only compare complete designs and thus follow a black-box approach. Here, we take a different path and instead disassemble the existing data structures to identify a set of appropriate design primitives. Combined with common low-level access patterns, we can derive performance profiles assigned to a precise primitive or combination thereof (O4). Based on these, we can further infer general design goals for PMem-based storage and data structures. The material presented for this contribution was peer-reviewed (where [GTS20a] is an extended but not reviewed version of [GTS20b]) and published in:

[GTS20b] Philipp Götze, Arun Kumar Tharanatha, and Kai-Uwe Sattler. Data Structure Primitives on Persistent Memory: An Evaluation. In Danica Porobic and Thomas Neumann, editors, *16th International Workshop on Data Management on New Hardware, DaMoN 2020, Portland, Oregon, USA, June 15, 2020*, pages 15:1–15:3. ACM, https://doi.org/10.1145/3399666.3399900, 2020

[GTS20a] Philipp Götze, Arun Kumar Tharanatha, and Kai-Uwe Sattler. Data Structure Primitives on Persistent Memory: An Evaluation. *CoRR*, abs/2001.02172, http://arxiv.org/abs/2001.02172, 2020

### Persistent Analytical Structures

Besides index structures – which are primarily intended for Online Transaction Processing (OLTP) – analytical data structures can also benefit from PMem, for example, by the increased capacity and direct persistence in contrast to DRAM. We present two approaches for utilizing PMem for analytical data structures and accelerating queries on these (O2). The first approach relies on clustering and organizes the data into a sorted linked list of data nodes, which in turn are append-only. Especially the unsorted nodes convert random-access updates into more efficient sequential append operations. With an additional index and pruning steps that exclude ranges of nodes, queries can run efficiently on arbitrary attributes (O4). The second approach

is a multi-dimensional index that is much more compact but less update-friendly. With the help of various developed caching strategies, it is possible to achieve near-DRAM performance (O4). The material presented for this contribution was peer-reviewed (where [JGBS21] is an extended version of [JGBS20]) and published in:

[GBS18] Philipp Götze, Stephan Baumann, and Kai-Uwe Sattler. An NVM-Aware Storage Layout for Analytical Workloads. In *34th IEEE International Conference on Data Engineering Workshops, ICDE Workshops 2018, Paris, France, April 16-20, 2018*, pages 110–115. IEEE Computer Society, https://doi.org/10.1109/ICDEW.2018.00025, 2018

[JGBS20] Muhammad Attahir Jibril, Philipp Götze, David Broneske, and Kai-Uwe Sattler. Selective Caching: A Persistent Memory Approach for Multi-Dimensional Index Structures. In *36th IEEE International Conference on Data Engineering Workshops, ICDE Workshops 2020, Dallas, TX, USA, April 20-24, 2020*, pages 115–120. IEEE, https://doi.org/10.1109/ICDEW49219.2020.00010, 2020

[JGBS21] Muhammad Attahir Jibril, Philipp Götze, David Broneske, and Kai-Uwe Sattler. Selective Caching: A Persistent Memory Approach for Multi-Dimensional Index Structures. *Distrib Parallel Databases*, https://doi.org/10.1007/s10619-021-07327-0, 2021

**Transactional Stream Processing**

Using our PMem-based data structures and design goals, we prototyped the envisioned TSP model (O3). This model combines the traditional relational database approach with more modern data stream management systems. We use PMem to persistently store states as tables and application-specific metadata needed to guarantee the ACID properties. We propose a concurrency control and consistency protocol utilizing the byte-addressability of PMem to eliminate logs and locks. Therefore, we simultaneously achieve near-instantaneous recovery of states and query pipelines (O4). On top of the stream processing model, query optimization is another considerable step that drastically influences the system throughput. In this context, we address the questions about which parameters are essential for a cost model and how query planning changes with PMem and stream processing (in contrast to the relational model). We opt for a hardware-conscious approach since the access latency and bandwidth of the various memory and storage technologies can vary widely. The material presented for this contribution was peer-reviewed (where [GS21] has been submitted but not yet published) and published in:

[PGS17] Constantin Pohl, Philipp Götze, and Kai-Uwe Sattler. A cost model for data stream processing on modern hardware. In Rajesh Bordawekar and Tirthankar Lahiri, editors, *International Workshop on Accelerating Analytics and Data Management Systems Using Modern Processor and Storage Architectures, ADMS@VLDB 2017, Munich, Germany, September 1, 2017*, http://www.adms-conf.org/2017/camera-ready/adms2017_final.pdf, 2017

[GS19] Philipp Götze and Kai-Uwe Sattler. Snapshot Isolation for Transactional Stream Processing. In Herschel et al. [HGR+19], https://doi.org/10.5441/002/edbt.2019.78, pages 650–653

[GPS19] Philipp Götze, Constantin Pohl, and Kai-Uwe Sattler. Query Planning for Transactional Stream Processing on Heterogeneous Hardware. In Holger Meyer, Norbert Ritter, Andreas Thor, Daniela Nicklas, Andreas Heuer, and Meike Klettke, editors, *Datenbanksysteme für Business, Technologie und Web (BTW 2019), 18. Fachtagung des GI-Fachbereichs „Datenbanken und Informationssysteme" (DBIS), 4.-8. März 2019, Rostock, Germany, Workshopband*, volume P-290 of *LNI*, pages 71–80. Gesellschaft für Informatik, Bonn, https://doi.org/10.18420/btw2019-ws-05, 2019

[GS21] Philipp Götze and Kai-Uwe Sattler. Transactional Stream Processing on Persistent Memory. *Inf. Syst. J.*, 2021. submitted in April 2021

### Event Stream Processing

In addition to the TSP model, we also consider a more specialized form of stream processing, namely event processing. This kind of processing usually operates on massive temporal data streams. Especially for maintaining data in so-called event stores (O2), PMem offers new possibilities to improve query performance and recovery guarantees. Based on an existing event store, we show several approaches to build a modern three-layer architecture consisting of DRAM, PMem, and secondary storage (O1). Along with the performance and recovery guarantees (O4), we show that such an architecture can also offer significant economic and ecological advantages. The material presented for this contribution was peer-reviewed and published in:

[GGK+20] Nikolaus Glombiewski, Philipp Götze, Michael Körber, Andreas Morgen, and Bernhard Seeger. Designing an Event Store for a Modern Three-layer Storage Hierarchy. *Datenbank-Spektrum*, 20:211–222, https://doi.org/10.1007/s13222-020-00356-6, 2020

Figure 1.3 visualizes the thematically classified contributions and which chapters discuss them. As indicated by the color shading, the main focus is on the storage and processing layers. The hardware and optimization layers, on the other hand, build on existing libraries

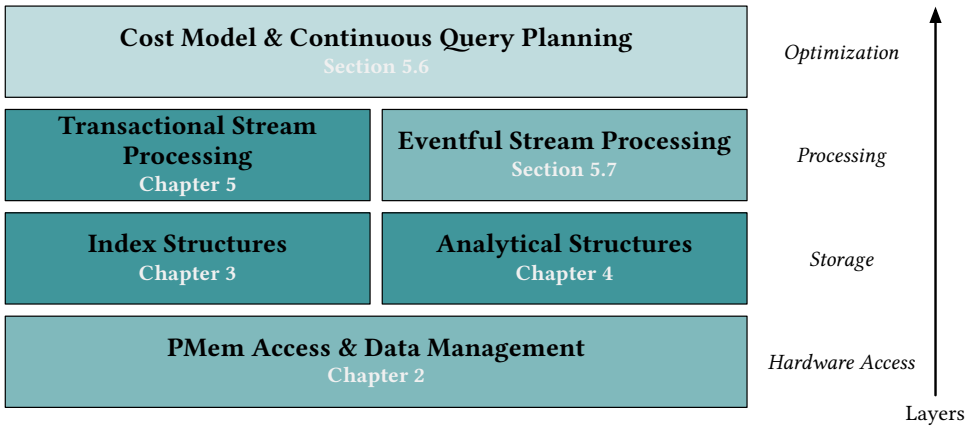



**Figure 1.3:** Overview of contributions allocated to the chapters of this dissertation.

or are considered less extensively, respectively. We give a more detailed motivation for each contribution at the beginning of its respective chapter. Furthermore, we examine related work separately per contribution. Each chapter validates its theses on real PMem hardware in an associated evaluation section to demonstrate the suitability of the proposed approaches. Eventually, each chapter summarizes its main consequences and insights.

The rest of this dissertation is structured as follows. In Chapter 2, we start by elaborating the fundamentals of the PMem technology like its properties, possible integrations into the memory hierarchy, and resulting data management challenges. We describe our employed access methods and show the characteristics of our available system based on initial measurements. Subsequently, in Chapter 3, we survey existing data and index structures proposed for PMem. Based on those, we discuss significant design decisions and extract their underlying design primitives. Combined with ordinary micro-operations, we evaluate their respective impact in detail. Then, Chapter 4 presents two different storage layout approaches targeting analytical workloads. Chapter 5 demonstrates how the before elaborated data structures and design goals can be used to build a stateful stream processing system. Besides the detailed description of the TSP model, we present the practical realization of the necessary concurrency control and consistency protocols. This description starts generic and then details the feasible tweaks for PMem. Furthermore, this chapter considers query optimization for our model and the special form of eventful stream processing. Finally, Chapter 6 concludes this dissertation with a summary of the results and an outlook on future research directions.

# 2 Persistent Memory - A New Paradigm

The general motivation behind PMem is the better scalability and direct persistence in contrast to DRAM that is reaching its physical limits. It combines valuable characteristics of both DRAM and flash technology and thus eliminates the large performance gap between them, as sketched in Figure 2.1. It becomes apparent that this type of memory is much closer to DRAM than to storage technologies in terms of performance. Depending on the use case, PMem can be used either as memory extension, memory or storage replacement, or even as a replacement of the complete memory hierarchy. Although only a few of the PMem technologies have arrived at the broad market yet, they present several attractive features, which are missing in either DRAM or flash storage. With these combined characteristics, new opportunities but also challenges arise, such as the general integration into the memory hierarchy but also more specific novelties regarding data access and management. In this chapter[1], we describe the fundamentals of this new technology type, go into deeper detail about its properties and possible hardware architectures, as well as arising challenges for data management tasks. Furthermore, we elaborate on the usage of PMem from a developer's point of view and report first measurements for the setup of our evaluation server. The contents of this chapter serve as a basis for the reader to understand the unique features and new way of programming introduced by PMem. That is important to comprehend design decisions, necessary adjustments, and performance impacts, among other things, in the subsequent chapters. Furthermore, it can provide a guide for the first own steps with the integration of PMem.

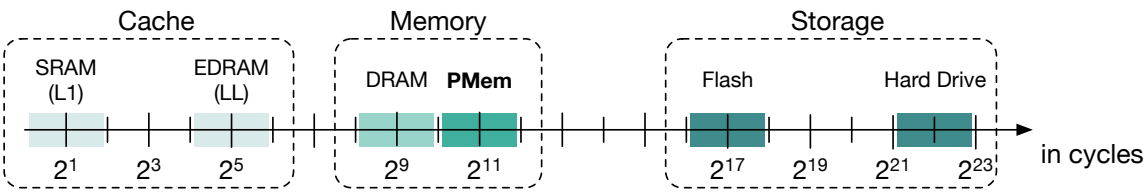



**Figure 2.1:** Typical access latency of memory and storage technologies in terms of processor cycles. Adapted from [QGR11].

[1]The material in this chapter is partly based on [GvRL+18].

This chapter is structured as follows. In Section 2.1, we start by presenting various PMem candidates, their common properties, and how to access these devices. Subsequently, Section 2.2 goes into detail about how to integrate PMem into the existing hardware landscape. These fundamentals already allow us to derive common challenges for efficient data management, which we cover in Section 2.3. Then, Section 2.4 discusses how to program with PMem and how existing libraries already solve some of the mentioned issues. In Section 2.5, we describe the first steps to set up the PMem devices. Moreover, we examine the performance of our system with the help of microbenchmarks to confirm the discussed properties and to reveal potential conspicuities. Finally, Section 2.6 summarizes the most important insights of this chapter and outlines what we can conclude for the following chapters.

## 2.1 Persistent Memory

In this first section, we describe how the term PMem can be practically realized and which technologies are possible candidates. We then discuss the properties resulting from these technologies. In principle, the approaches share many characteristics, but there are also peculiarities that we will address here. Finally, we will show options to access PMem devices – that we expect in the DIMM form factor – from the perspective of the operating system.

### 2.1.1 Technologies

The first practical approaches of PMem tried to use the available technology and enhance it with additional controllers, batteries/ultracapacitors, or flash such as Vikings ArxCisNV [Pro12], Vikings NVDIMM-N [Vik20], or AGIGARAM NVDIMMs [Agi14]. These DIMMs backed by batteries and NAND flash are classed under the term NV-DIMMs. Usually, the flash is not visible to the host system and is only used as a persistent backup. In the case of a power loss, a signal triggers the controller to write the DRAM content to the NAND flash using the onboard battery or capacitors. As soon as power resumes the content is restored. The big advantage of this approach is the low monetary cost since it is entirely based on commodity hardware [NH12, WJ14].

One of the first discussions about possible native PMem technologies started already 1971 where Leon Chua predicted the existence of the memristor [Chu71] as the fourth fundamental circuit element. It belongs to the later termed class of Resistive RAM (RRAM) [SSSW08, GKC+11]. Consisting of two layers of titanium dioxide between two electrodes, the resistance of a memristor changes as an electric current passes through. This is due to the thereby varying thickness of the insulating layer. The difference in resistance is then used to store and retrieve data.

One of the best-known technologies is Phase-Change Memory (PCM) [LZY+10], where data is stored by changing the state of a chalcogenide to a low or high resistance state. For that, the memory cells consist of two electrodes, a resistor, and the phase change material. For reading an electrical current is injected for measuring the electrical resistance. The writing process applies current to the chalcogenide to either crystallize (moderate, long pulse) it or switching it back into an amorphous state (high, short pulse).

Another candidate is Spin Transfer Torque Magnetic RAM (STT-MRAM) [HYY+05], where magnetic properties are utilized. Magnetic RAM uses two ferromagnetic plates separated by a thin insulator, each holding a magnetic field. Since one of the plates is a permanent magnet with a certain polarity and the other can be set at will by applying a current-induced magnetic field, it is possible to store bits. The constructed fields generate a magnetic tunnel, whose electrical resistance depends on the field's orientation. The special feature of spin torque transfer MRAM is the write mechanism, which uses a spin-polarized current to flip the spin of the field.

Carbon nanotubes can also potentially be used to realize PMem and are often referred to as Nanotube RAM (NRAM) [RKJ+00, KRM+10]. One way is to lay out these tubes as I/O wire arrays with bistable cross points. The bistability is achieved when the intersecting nanotubes are either separated or in contact. The state can be read via the resistance of the junction. Temporary charging of the nanotubes generates attractive or repulsive electrostatic forces that change the state.

After tens of years of research, Intel's Optane Data Center Persistent Memory Module (DCPMM), the first broadly commercially available PMem hardware, was released in 2019. It bases on the 3D XPoint [MT20] technology, but its underlying physical mechanism is not completely known. However, the characteristics show a similarity to PCM and NRAM. Due to its availability, we focus on the 3D XPoint technology for the rest of the thesis.

### 2.1.2 Properties

What all technologies have in common is the low latency and byte-addressability like DRAM combined with the density, non-volatility, and economic characteristics of traditional storage like SSD and HDD. Byte-addressability is actually a theoretical aspect. Even if both DRAM and PMem could be accessed byte by byte, modern CPU architectures still access them in cache-line granularity – typically 64 bytes. However, an additional feature is the option to access the persistent memory modules directly via the CPU (or its caches). This means that no costly copying processes into DRAM are necessary. Almost all PMem technologies show a read-write asymmetry concerning performance (latency and bandwidth), cell wear, and power consumption. That means writes are more costly than reads. Exact measurements of the latency and bandwidth for reads and writes for our system are given in Section 2.5. Similar to flash, some PMem technologies support only a limited number of writes. However, we think that cell wearing is addressed at the hardware level like it is the case for SSDs. Due to its persistent property, idle PMem cells do not consume power which is another edge over DRAM.

In addition to the general properties, the 3D XPoint technology exhibits a few more peculiarities. Although the transfer size from the CPU to PMem (and also DRAM) is 64 bytes, the DCPMMs internally operate on 256-byte blocks. A write-combining buffer is used here to reduce write amplification by combining four cache lines to one block write. Interestingly, read operations also benefit when a multiple of the block size is used [vRVL+19, YKH+20]. Not too relevant for our research purposes, but still interesting is the security aspect, i.e., only encrypted data becomes persistent. Therefore, DCPMMs offer encryption as a built-in hardware feature [Int20a]. As mentioned before, performance numbers are reported in Section 2.5.

### 2.1.3 Access Model

Similar to HDD and SSD, the Storage Networking Industry Association (SNIA) recommends managing PMem by a file system [SNI17]. As shown in Figure 2.2, applications then have two options of accessing the device. The first option is to access files through standard system calls like open, read, write, and close. This makes it easy to migrate existing disk-based applications to PMem since the file system interface is the same. On top of that, when developing new applications, file systems already provide a lot of the functionalities required for managing the underlying media such as naming, corruption handling, and allocation. Several PMem-enabled file systems have already been proposed that try to better exploit the media's special properties, such as BPFS [CNF+09], SCMFS [WR11, WQR13], PMFS [RKK+14], HiNFS [OSL16], and NOVA [XS16a, XS16b]. This access path can be realized either traditionally via the operating systems page cache or directly to the PMem device. According to SNIA [SNI17], this corresponds to NVM.FILE and NVM.PM.FILE mode, respectively. A disadvantage of the first is the extra copy to DRAM before the data can be accessed. For the second option, the file system must provide zero-copy memory mapping bypassing the page cache of the operating system. Hence, applications are provided with direct access to the device via load and store instructions through the CPU caches. This feature is called Direct Access (DAX) [Lin21] and is supported, for instance, in ext4 [Lin18] and xfs [Lin19] starting from Linux kernel 4.7. A drawback of this access option is the loss of transparent memory defragmentation and swapping that is only possible with the duality of PMem and DRAM pages. Nevertheless, the benefits prevail and the virtual memory indirection still provides features like process isolation, position-independent code and data, and memory sharing between processes. Without virtual memory, applications would have to be reconsidered in a much more complex manner.

Besides these basic access models, Optane DCPMM offers explicit operating modes when setting up the DIMMs [Int20c]. These are Memory Mode intended to extend the DRAM capacity and App Direct mode allowing byte-addressable access and persistent guarantees. In more detail, in Memory Mode, DRAM acts as a cache above PMem with a larger capacity. The cache management is implemented in the processor's memory controller. For every memory request, the DRAM cache is checked first. If the data is present, it is returned with DRAM latency and if not, it is directly read from PMem. The advantage of this mode is, similar to the native file API above, that existing applications can work transparently, however, with a much higher

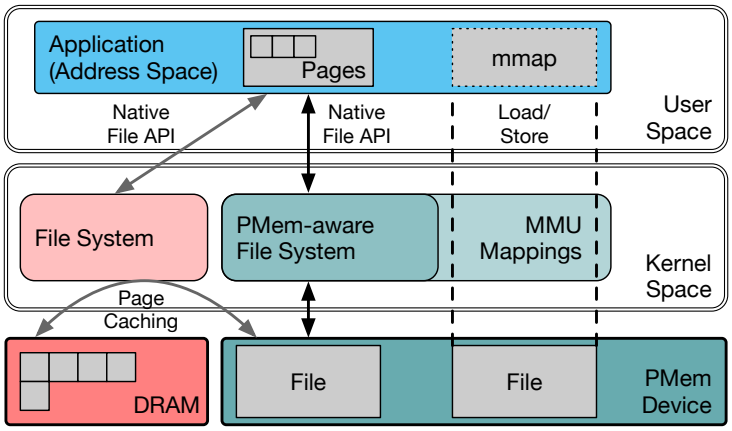



**Figure 2.2:** Application's basic file access options to PMem devices.

memory capacity. This mode is entirely volatile and has no persistence guarantees that are only enabled with App Direct Mode. In this mode, applications are aware of the two different memory types and can directly control which to access. The PMem device is accessed in the form of memory-mapped files as described above. Furthermore, it is possible to separate the device to support both modes (Mixed Mode). Since we need persistence and want to control the placement, we focus on the App Direct Mode in the remainder. Next, we consider the possible incorporation of PMem in the hardware landscape.

## 2.2 Integration into the Hardware Landscape

Since the access model is close to that of hard drives and flash storage, the emergence of PMem could cause a similar paradigm shift to further close the access gap. In the early 90s, flash devices (NAND SSDs) were introduced and are today virtually standard in modern data centers. Technologically, flash is even closer to PMem than to mechanically rotating HDDs what can ease the transition. However, in contrast to PMem, both flash and HDDs expose a block-based interface to the applications. It leads to the fact that data must always be converted from a logical (possibly object-oriented) form in DRAM into a serialized format to store them persistently. Conversely, the data must then be deserialized again to be able to work with them. Flash is certainly faster than HDDs here, but they have higher procurement costs and are not as durable. Due to the common interface and the respective strengths, both flash and HDDs have found their place in the storage hierarchy. With PMem the same file system interfaces can be used, but with direct access. Therefore, the detour via DRAM described above is no longer mandatory. It is this aspect of byte-addressability that allows much greater flexibility for the utilization of PMem and its placement in the hardware landscape. Thus, we are convinced that PMem will sooner or later become a standard component of a modern storage hierarchy, just like flash once did. In addition, DRAM has apparently reached its maximum density level and PMem could thus also serve as a main-memory extension or even replacement. There are many conceivable approaches, how this technology can be integrated into modern and future systems. Below, we classified these into three strategies, which can naturally be combined and extended. They are visualized in Figure 2.3 and were similarly discussed in [OKW17, GvRL+18].

### 2.2.1 PMem below DRAM

The first and probably most intuitive option is to place PMem between DRAM and SSD/disk according to their performance and pricing. Here, the access always passes DRAM. PMem thus serves as the first persistent zone and can be further used as persistent caching in front of SSDs [EGA+18, LPD17]. The opposite is also possible, with PMem acting as a so-called anti-cache [DPT+13] capturing data from DRAM as soon as it reaches a defined limit [DAP+14]. The idea of anti-caching also includes the concept that there is only a single copy of each data element at any given time. It eliminates synchronizations between cached and major copies.

Besides caching, two other data placement strategies are applicable here. Either the data is statically placed in the persistent layers, or the data is dynamically moving between the layers. The logic of when and where parts of the data are placed is defined by the application. Similar to anti-caching, but here related to the persistent domain, it could be implemented in the same

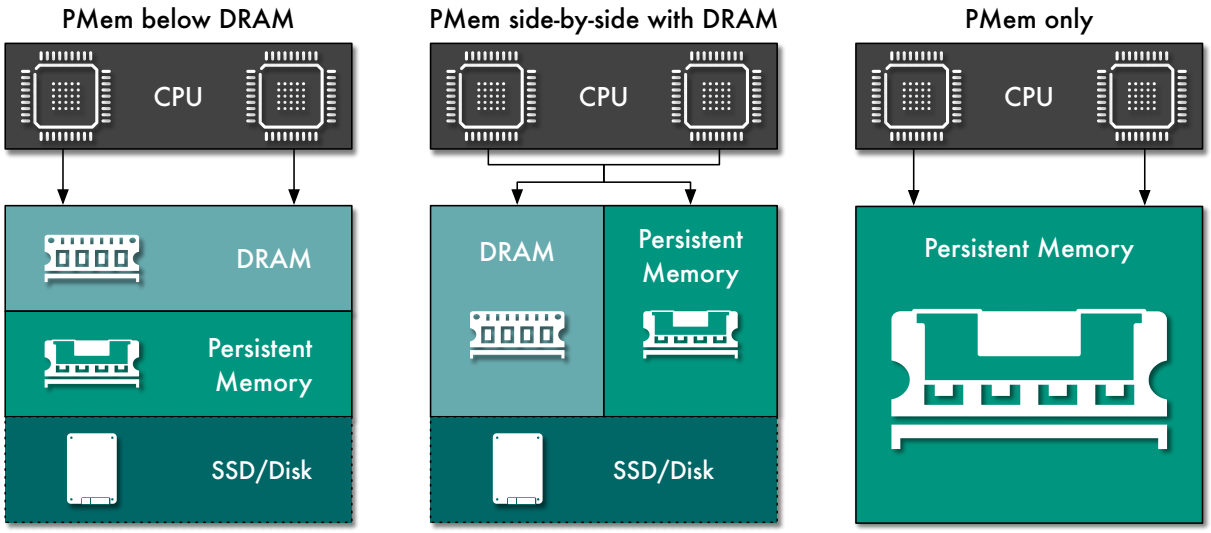



**Figure 2.3:** Placement strategies for PMem in the hardware landscape. Dotted lines denote optional component.

way that there is always only one durable copy of the data. This distinguishes the dynamic strategy from typical caching. However, depending on the data formatting, dynamically moving data between block devices and PMem may incur additional serialization and deserialization costs. Therefore, this movement should probably be done rather occasionally, and if so, then in batches. Alternatively, it is possible to agree on the highest common denominator, the page size of the deepest device in the hierarchy, and work exclusively with byte blocks.

Overall, PMem basically fulfills the role of a storage device in this placement strategy. As it is the case for disks or SSDs, the data would first be copied into DRAM to be further processed in the CPU. The difference here, however, lies in the granularity of the data. Whereas SSDs and HDDs use several KiB for a page, PMem can work, for example, at cache-line scale (or 256 bytes in case of DCPMMs). Another difference to block devices is that developers can control exactly when data is flushed/persisted and do not have to rely on the operating system. This can eliminate many developed mechanisms for consistency maintenance, which significantly reduces code complexity.

### 2.2.2 PMem side-by-side with DRAM

If the access and processing of the data on PMem shall happen directly, it moves in the hierarchy to the same level as DRAM. Thus it can be addressed directly via the CPU through the memory bus, and there are no more copies or movements of the data, which allows more targeted and faster access. This strategy has the highest potential to improve the performance of the system, although the implementation is more complex. If systems have high performance or real-time requirements, such as OLTP or stream processing, the primary data will most likely still reside in DRAM. However, from an economic and ecological perspective, it can be worthwhile to switch to a slightly slower device if this results in significantly lower hardware and energy costs. A sophisticated combination of both technologies and a well-conceived data placement can lead to a perfect compromise.

Similar to the previous strategy, the placement can be organized statically or dynamically. However, the difference is that, for example, the buffering strategy can be more sophisticated by dynamically deciding whether a page or element should be accessed directly in PMem or copied to DRAM first [LLO19]. Another example would be to dynamically move frequently accessed nodes of a tree-like structure to DRAM to exploit its lower latency and higher bandwidth. This could again be implemented as part of a buffer pool or use cost functions to predict or even learn access patterns to help in decision-making regarding the movement. Sequential scans, for example, that typically can trash a buffer, could access PMem directly and hide its higher latency through hardware prefetching. Especially for the DCPMMs, we see that the performance difference for sequential accesses is much lower than for random patterns compared to DRAM (cf. Section 2.5). In the course of this work, this aspect of the access pattern will also be taken up a few times.

If the developer can accurately distinguish between primary data and recoverable secondary data, static placement is a conceivable option. The obvious compromise is a longer recovery time to rebuild the secondary data, which can also happen on-demand or in the background. This can result not only in hybrid high-level components but also in hybrid data structures. The most prominent example is a hybrid $B^+$-Tree as proposed by [OLN+16]. Here, only the leaf nodes are stored in PMem while inner nodes reside in DRAM and are recovered in the case of failure. Different node sizes can then be set to leverage the properties of the underlying technology. This approach saves both DRAM space and recovery time while maintaining a DRAM-like performance. The former is because the number of inner nodes is logarithmically smaller than the total number of nodes. The latter is due to the fact that leaf nodes do not need to be rebuilt. However, the static placement is not always applicable and is limited since it can not react to workload or hardware changes.

In this setup, PMem can serve both the memory and the storage role. Therefore, we believe this is the most versatile and promising strategy.

### 2.2.3 PMem-only

The third and last discussed strategy considers PMem as universal memory completely taking over both the fast memory and persistent storage tasks. This eliminates the need to distinguish between volatile and persistent domains as well as the movement of data between devices. In order to fundamentally fulfill this task, the PMem technology would have to provide better or at least the same performance characteristics as current volatile memory technologies such as DRAM. For the current two generations of Intel's DCPMMs, this is not the case (yet). Thus, this strategy would be more suitable for embedded systems or similar purposes. However, the latest Xeon® Scalable processor generation provides a new feature called extended Asynchronous DRAM Refresh (eADR) [Int21a, Int21b]. This feature basically makes CPU caches also persistent by introducing dedicated ultracapacitors as well as backup and recovery logic. Therefore, the data does not have to be flushed to ensure persistence, which in turn could enable a competitive performance to DRAM. In principle, a discussion about a PMem-only solution thus makes total sense.

The idea of a universal memory is not entirely new and was already discussed in [AJ89]. In the context of databases, the authors describe various algorithms for storage and recovery using non-volatile main-memory. With the announcement of upcoming persistent memory

technologies, this discussion was resumed recently [APD15]. Basically, such an architecture would massively reduce code complexity since recovery and buffering can completely be omitted. Waiting times for copy operations between devices are also eliminated. All the more the focus will shift towards cache-optimized designs while ensuring cache coherence. As above, this strategy can also include additional economic and ecological benefits.

On the other hand, having an always persistent main-memory can also be undesirable. Often systems maintain a clear separation between data to be stored and runtime-related data. For example, persisting usually volatile reference counters, locks, latches or similar states could lead to persistent memory leaks and deadlocks since the associated thread(s) does not exist anymore in case of failure. Thus, in addition to the advantages, new challenges also arise, particularly concerning consistency and fault tolerance.

## 2.3 Data Management Challenges

Since we envisage a transactional data management system built on PMem, we describe below important aspects and our derived issues or challenges in this regard. Recalling the ACID properties as known from transactional database systems, the challenges raised by PMem can be classified into failure atomicity and concurrency. There is also the general question of how to make the best use of the properties of this medium. Finally, after we have already seen the various integration options above, there remains the open question of which and where data and data structures should be placed.

### 2.3.1 Failure Atomicity

Just as with DRAM, memory accesses to PMem pass through the cache hierarchy of the CPU(s). While this drastically improves performance, modifications that initially happen in the cache are not immediately on the memory device. Although it is possible to control when data should be transmitted to PMem, failures can occur at any time. This is not a problem for DRAM because everything is volatile and intermediate results are no longer visible. However, with PMem, there is now a volatile and a persistent area. On modern CPUs, the size of a failure atomic write is only 8 bytes. Therefore, for modifications beyond this, intermediate and possibly inconsistent changes to a data structure could be persisted. So the challenge is either to prevent inconsistencies in the event of a failure or to eliminate them on restart. The former way is usually realized with shadowing techniques and atomic operations such as Compare-and-Swap (CAS) or *fetch-add*. Sometimes a sole use of atomics is even applicable. The elimination of inconsistencies on restart is typically done with logging techniques. The traditional write-ahead logging, for instance, records a before (undo) or an after (redo) image prior to applying the changes. During a restart, this log is used to recover corrupted or incomplete data regions. In Section 2.4, we will go into more detail on how these two ways can be programmatically implemented.

### 2.3.2 Concurrency

Threads simultaneously working on the same set of data have to synchronize with each other. This aspect is already known from DRAM and flash. However, the question is whether the existing approaches are also applicable and follow the same criteria in the case of PMem. Once again, the CPU caches have a major impact here. If two caches contain data from the same memory region, a modification in one of the caches leads to the invalidation of cache lines in the other cache. This is necessary to maintain cache coherence, i.e., data consistency. Due to the higher latency of PMem, the impact of cache misses caused by this procedure is much higher. In addition, a new phenomenon arises: changes often become visible to other threads through the caches before the data is actually persistent. Thus, data visibility and persistence should be kept in mind and not be confused when developing concurrency control mechanisms. Another new problem occurs, for example, when locks or latches are kept persistent. As we already mentioned in Section 2.2, this can lead to persistent deadlocks in the case of failure. Furthermore, setting locks by different threads can again introduce the cache invalidation issue mentioned above. However, it is often useful, especially for latches, to store them close to the data to avoid random access patterns and increase throughput. Overall, this conflict seems difficult to resolve, which is why locking may not be suitable for PMem. Other options would be optimistic validations, timestamp comparisons, or versioning. On top of that, modern CPUs provide hardware transactions. A combination of hardware and software-based concurrency is, for instance, demonstrated in [OLN+16]. Here, a hardware transaction is initiated first and if this fails a defined number of times, it falls back to the programmer-defined approach.

### 2.3.3 Property Utilization

In Section 2.1, we already discussed the properties of PMem. Here, we will outline how these can be exploited or circumvented. Essentially, we see three aspects that should be considered. The first is the reduction of writes (or even reads) to PMem to tackle the read-write asymmetry and the poorer performance compared to DRAM. Furthermore, due to the limited capacities compared to block devices and partly also for performance reasons, the general storage space occupation should be reduced. Secondly, the direct and fine-granular access allows for new and more optimal algorithms for the storage layer, which are partly already known from in-memory structures. Simply reusing these algorithms, however, is probably not ideal due to the read-write asymmetry. Finally, following the same argument of the direct access and thus the almost direct persistence allows a much faster recovery process than with disks. However, depending on the data placement, auxiliary structures in DRAM might still have to be rebuilt.

The open question is how to achieve all three aspects. Anticipating this, we have already given a few examples in Section 2.2. In the literature, the most common optimization steps include leaving data nodes or blocks unsorted to avoid unnecessary writes resulting from entry shifting. However, this can result in the entire block having to be traversed when searching, while for sorted blocks it is possible to terminate the search prematurely. Therefore, new approaches are needed to achieve a good compromise between insertion and search operations. The first two aspects are consequently closely linked. By contrast, in the recovery case, a trade-off must be found between fast restarts and general access times in operational mode. Thus, keeping secondary structures in DRAM provides better performance, but at the same time, leads to higher recovery overhead due to rebuilding.

### 2.3.4 Data Placement

We expect that the PMem side-by-side with DRAM strategy can best exploit the properties of all technologies. However, with the coexistence of two or three manually accessible technologies, several questions arise regarding data placement. For instance, should the whole primary data be on PMem, what can be outsourced to disk, is it possible to efficiently cache hot data in DRAM, or should PMem be the cache in front of disks? Furthermore, should these placement decisions be statically, dynamically, or even adaptive[2]?

To give a first glimpse, we considered different costs depending on how the data is placed in the context of event stores using all three memory/storage layers [GGK+20]. In particular, we consider processing, recovery, and monetary costs. The results are illustrated in Figure 2.4. It bases on a realistic ratio taken from ChronicleDB – which we will further elaborate on in Section 5.7 – with 1 TB primary data and 100 GB reconstructable secondary data. While the monetary costs are exact numbers based on [HHL20], the processing and recovery costs are estimated according to the performance and characteristics of the used technologies. Option 2, 3, and 4 are not always possible since they place high demands on the available capacity of PMem or DRAM. In our system, for example, these are not met (see Table 2.1). Moreover, these options are the most expensive. The traditional placement is given with option 1, which is currently the most affordable solution. We expect that forthcoming generations of PMem will be more cost-effective in monetary terms, especially once there are competitors. This assumption is motivated by the price development of DRAM and flash [HHL20]. Therefore, option 5 comprising only PMem and flash could become the most economical and ecological data placement strategy. This would also lead to the fact that option 6, which spans across all three layers, would provide the best-balanced system considering all aspects.

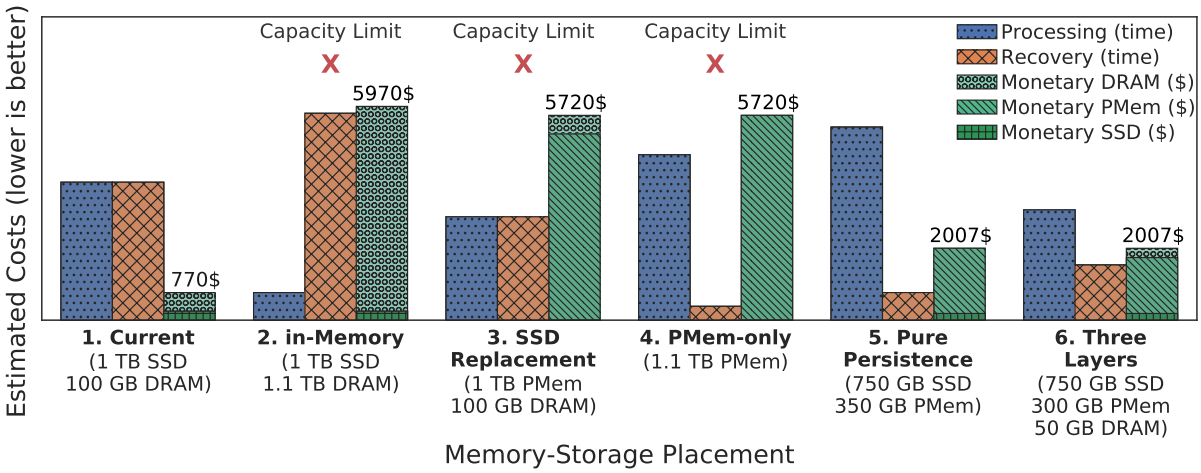



**Figure 2.4:** Various costs depending on the data placement (cf. [GGK+20]).

[2]While a dynamic strategy has a fixed set of data migration thresholds, an adaptive strategy goes even further and might adjust (or learn) these thresholds depending on, e.g., runtime statistics.

## 2.4 Persistent Memory Programming

When programming with PMem, the challenges discussed above have to be kept in mind especially to maintain consistency even in the event of failure. The resulting programming style can be quite different from the typical development with DRAM and disk. The good news is, many of the general programming challenges identified, for instance, by [OBL+17] are already addressed by software libraries like Persistent Memory Development Kit (PMDK) [Int20d]. Still there remain basically two ways to address the challenge of consistency and failure atomicity. The first is to use easier high-level abstractions which follow a transactional-memory-like approach but also introduce the overhead of systematic logging. This provides a generic solution and makes PMem programming more accessible. The logging overhead comprises at least one or two additional writes for every modification. At first, a snapshot of the data to be modified is taken (undo logging) and second the undo log is zeroed out (e.g., for PMDK versions <1.7). The second rather low-level approach to ensuring consistency and durability is the usage of dedicated CPU persistence primitives. These are instructions to flush cache lines (clflush, clflushopt, and cache line write back (clwb)), memory barriers (mfence and sfence), and non-temporal stores (movnt). Together with atomic loads and stores, this approach has the great advantage of enabling low-level optimizations. On the downside, it is a bit more difficult, prone to errors, and requires that the possible application states are considered carefully. Let us consider a typical example of appending an entry to a preallocated list or array, which could have the following structure:

```
struct Array {
  Entry entries[64];
  size_t size;
};
```

The transactional way of appending an entry would look like this:

```
void append(Array &array, const Entry &entry) {
  TX_BEGIN {
    array.entries[array.size] = entry;
    ++array.size;
  } TX_END
}
```

As it can be seen there is not much difference to an in-memory implementation except the transaction wrapper. Most of the work happens in the background like writing to the undo log. In contrast, the low-level optimized variant could look like the following:

```
#define pmem_clwb(addr)\
  asm volatile("clwb %0" : "+m" (*(volatile char *)(addr)));
void append(Array &array, const Entry &entry) {
  array.entries[array.size] = entry;
  pmem_clwb(&array.entries[array.size]);
  _mm_sfence();
  ++array.size;
  pmem_clwb(&array.size);
  _mm_sfence();
}
```

That is much more verbose than the first solution but would need only two writes (depending on the type size of `Entry` and cache line boundaries). The transactional way would need at least double the number of writes. Thus, if high performance is demanded, the low-level approach is highly recommended. However, looking at the code snippet, it becomes clear that this approach is much more prone to errors if the instructions are not correctly placed.

### Persistent Memory Development Kit

With PMDK, different levels of abstractions also including the just described consistency methods are provided. In this thesis, we mainly relied on its C++ bindings within the *libpmemobj++* library. As a high-level abstraction, the *transaction::run* lambda function can be used to encapsulate all modifications that should happen atomically. Starting from version 1.7 the log is no longer zeroed out and instead, the log data is invalidated alongside the log metadata. Therefore the generic method is a bit faster than before. Considering the example again, the code would change into the following (the pool concept is discussed soon):

```
void append(Pool pop, Array &array, const Entry &entry) {
  transaction::run(pop, [&] {
    array.entries[array.size] = entry;
    ++array.size;
  });
}
```

For complete control on the lower level, PMDK offers the methods *pmem_flush*, *pmem_drain*, and *pmem_persist*. The first corresponds to the cache line flush instructions and the second enforces the ordering of stores (i.e., memory barriers). The last method is a wrapper to combine the former two. The advantage of these methods is, on the one hand, the prevention of assembler and thus a cleaner and more understandable code. On the other hand, the library already checks the hardware architecture during initialization and automatically uses the best possible instructions. Furthermore, larger areas than one cache line can be specified and PMDK will then perform multiple optimized flushes. With these wrappers the low-level way becomes much more clearer:

```
void append(Array &array, const Entry &entry) {
  array.entries[array.size] = entry;
  pmem_persist(&array.entries[array.size], sizeof(Entry));
  ++array.size;
  pmem_persist(&array.size, sizeof(size_t));
}
```

Below, we briefly describe other used terms and concepts of PMDK for transaction and object management (cf. [Int16]).

*Persistent memory pools:* As we discussed in the section about the access model, PMem is managed by the operating system using a PMem-aware file system. The direct access method via memory mapping is referred to as pools in this context. PMDK provides the three basic operations create, open, and close to initiate or end the memory mapping. It also offers more advanced features like wrappers to the flush and barrier methods from above. With the C++ implementation, it is also possible to manipulate memory alignments.

*Persistent pointers:* To allow for a recoverable addressing scheme the concept of persistent pointers is introduced. This pointer must be valid across application and system restarts.

Furthermore, it has to be mapped back to the application's accessible virtual address space. For this, PMDK provides the *persistent_ptr* template that contains an 8-byte ID of the persistent memory pool plus another 8 bytes for the offset of the target object within this pool. The template can be used in the same way as C++ smart pointers wrapping an object type. However, the creation usually happens during object construction using the *make_persistent* function, which returns a persistent pointer. This allocation process must be mandatorily encapsulated by a transaction to prevent persistent memory leaks. Calling the corresponding complementary function *delete_persistent* passing a persistent pointer deallocates the underlying object.

*Root object:* The root object is highly coupled with the persistent memory pool initialization. It is the entry point to which all other data structures and variables are attached in the pool. Hence, for each pool exactly one root object always exists. It is initially zeroed and can have any user-defined size. The position of the root object in the pool itself is not fixed. However, a persistent pointer to the current root is kept at a known offset. This allows the application to recover its data.

*Persistent properties:* Besides persistent pointers, another object wrapper is a persistent property (simply *p*). Modifications to such wrapped objects are automatically added to the enclosing transaction, i.e., its undo log. This helps to prevent consistency issues caused by forgetting to register changes. If transactions are not used anyway, these properties are accordingly not necessary. However, they do not add any extra storage overhead and can also just be used as an indicator for an in-place persistent field.

It follows a minimal example of how all the above terms can be used together. Comments describing the process are given inline.

```
/// target object
struct Array {
  persistent_ptr<Entry[]> entries;
  /// alternatively in-place: p<std::array<Entry, 64>>
  p<size_t> size;
};

/// the entry point for the pool - the root object
struct root {
  persistent_ptr<Array> array_ptr;
};

/// pool initialization and object construction
auto path = "/mnt/pmem0/array";
auto pop = pool<root>::create(path, ...); ///< or open if existing
auto &arr = pop.root()->array_ptr;
transaction::run(pop, [&] {
  arr = make_persistent<Array>();
  arr->entries = make_persistent<Entry[]>(64); ///< if not in-place
});

/// do something with array
...

/// if not needed anymore delete it again
transaction::run(pop, [&] {
  delete_persistent<Entry[]>(arr->entries, 64); ///< if not in-place
  delete_persistent<Array>(arr);
});

pop.close();
```

### Non-Uniform Memory Access Control

Another important aspect when programming close to the memory bus is constituted by Non-Uniform Memory Access (NUMA) effects. These arise through the differences in speed of processor access to memory found in multi-socket CPUs. Remote access has a higher latency than local access. Local access means that the requested memory region maps to the same socket as the requesting core and can happen directly. Remote, on the other hand, implies in this context to read or write to memory regions of another socket, i.e., different to the core's socket. The higher latency is caused by the additional communication between the local and remote memory controller as well as the lower throughput of the cross-chip interconnect. Typically, the latency deteriorates by 2× [MG11]. Furthermore, each socket has its own cache hierarchy. Modifying data in the caches of one socket has to invalidate the same cached memory location in other sockets to maintain cache coherence. Therefore, frequent access to the same memory regions of different CPUs leads to regular cache misses and thus poor performance. However, the same principle can be found for all caches at the same level, i.e., also all L1 and L2 caches have to synchronize among themselves. Most of our experiments we run on a single socket to avoid both these effects and complicating the experiments. Especially, we expect to limit fluctuations in performance measures. With the *numactl* [KS04] command, it is possible to control where computing and memory resources are allowed to be allocated. The PMem socket is fixed by the file system path (see Section 2.5).

## 2.5 Initial Measures

Throughout our experiments and evaluations, we use a two-socket system as outlined in Table 2.1. In the following, we will describe our initial measures to set up the PMem devices which are similarly described in [Int20c, vRVL+19]. Each socket consists of twelve DIMM slots and six channels. The PMem and DRAM DIMMs are plugged in so that both cover all channels. To increase the possible bandwidth the DCPMMs are interleaved and grouped to one region per socket. This can be done in the BIOS or via the *ipmctl* command like this:

```
ipmctl create -goal -socket <number> PersistentMemoryType=AppDirect
```

Next, we create a namespace – similar as for SSDs to represent storage units appearing as a separate device – on top of both regions using the *ndctl* command:

```
ndctl create-namespace --mode fsdax --region <number>
```

**Table 2.1:** Server setup used throughout our experiments.

| | |
|---|---|
| Processor | 2 Intel® Xeon® Gold 5215, 10 cores / 20 threads each, max. 3.4 GHz |
| Caches | 32 KB L1d/L1i, 1024 KB L2, 13.75 MB LLC |
| Memory | 2×6×32 GB DDR4 (2666 MT/s),<br>2×6×128 GB Intel® Optane™ DCPMM (2666 MT/s) |
| Storage | 4×1 TB Intel® SSD DC P4501 Series |
| OS & Software | CentOS 7.9, Linux 5.10.6 kernel, cmake 3.15.3, GCC 9.3.1 (-O3), OpenJDK 14.0.1, PMDK 1.9.1 |

Since we want to have DAX support, we have to pass *fsdax* as mode (which is actually the default). The region number(s) can also be determined by *ndctl*. After that, we can create a file system on top of both namespaces, as known from block devices. We opted for ext4. After determining the mapped block device path, the file system can be created simply by:

```
mkfs.ext4 /dev/pmem<number>
```

Finally, the file system has to be mounted to be accessible by applications. It is important to specify the `dax` option here again:

```
mount -o dax /dev/pmem<number> /mnt/pmem<number>
```

With the mount path, it is now possible to create files and memory map them into the user space or to use PMDK pools as described in the previous section.

In order to confirm the PMem properties described above and also to identify peculiarities of our server setup, we measured typical performance indicators like latency and bandwidth. We excluded write latencies as these are hard to measure on PMem due to the write-combing buffers and the write pending queue. We also measured the same indicators for DRAM and the installed SSDs (xfs) to enable a better classification. For this, we used Intel's *Memory Latency Checker* [Int20b] for DRAM and PMem as well as *Flexible I/O Tester* [Axb20] for the flash device. For DRAM and PMem the measurements were done from and to the same socket (local). Our results are given in Table 2.2. We created a 100 GiB file on one of the flash drives. Since the measurements were fluctuating, we report the $99^{th}$ percentile latency. For the bandwidth, we iterated through a range of different block sizes and parallelism degrees (i.e., threads and I/O depth). From these, the best throughput achieved is listed in the table. Similarly, we report the best results achieved for DRAM and PMem as well. As it can be seen, PMem is for both latency and bandwidth about 2-4× slower than DRAM. Considering the latency, the difference between sequential and random access is highly salient for PMem and even more for flash. With DRAM, on the other hand, it is almost negligible. Similarly, we also observe the read-write asymmetry for PMem and flash. One anomaly, we noticed here is the low sequential write bandwidth for DRAM. This should be closer to the read bandwidth since DRAM cells are not read-write asymmetric. For the random bandwidth measurements (number of reads/writes) this behavior blurs. The big difference between the device types stems from the various utilized block sizes

**Table 2.2:** Measured performance and other characteristics of memory/storage technologies within our server.

| | DRAM | DCPMM | NAND TLC Flash |
|---|---|---|---|
| Idle Sequential Read Latency | 81 $ns$ | 174 $ns$ | 14 $\mu s$ |
| Idle Random Read Latency | 88 $ns$ | 325 $ns$ | 206 $\mu s$ |
| Maximum Read Bandwidth | 85 $GB/s$ | 32 $GB/s$ | 3 $GB/s$ |
| Maximum Write Bandwidth | 46 $GB/s$ | 13 $GB/s$ | 0.6 $GB/s$ |
| Random Reads | 931 $M/s$ | 45 $M/s$ | 299 $K/s$ |
| Random Writes | 703 $M/s$ | 30 $M/s$ | 61 $K/s$ |
| Write Endurance | $> 10^{15}$ | $10^5 - 10^7$ | $10^3 - 10^5$ |
| Density | $1X$ | $2X - 8X$ | $8 - 64X$ |

(64 bytes for DRAM, 256 bytes for PMem, 4 KiB for flash). Regarding the endurance data, we relied on the typical known write cycles for DRAM and flash. For the used flash device an endurance value of 1.85 PWS is reported [Int17]. Having 1 TB corresponds to 1,850 write cycles per cell. In the case of PMem, we could not find actual numbers for the first generation used here. Thus, we base the numbers on the report for the second generation [Int21b]. The density factor is based on today's maximum capacity per module of available commodity hardware. Besides the DRAM anomaly, our measurements are largely consistent with specifications and other reports [HHL20, Int19, LHO+19, vRVL+19, YKH+20] and we can thus confirm the properties from above.

## 2.6 Conclusion

The new and unique properties introduced by PMem will cause a paradigm shift closing the large access gap between memory and storage. In this chapter, we have covered the basics of accessing and managing data on this new type of technology. We have described the anticipated characteristics and possible ways of access and integration into the existing hardware landscape. Based on this, we have pinpointed evolving challenges for data management tasks. Furthermore, we have explained the preparation and practical use of PMem devices in more detail, and last but not least, we discussed the idiosyncrasies of the system available to us. All in all, the PMem fundamentals presented in this chapter are necessary to make essential decisions when designing PMem-enabled data structures and systems, which we elaborate on in the following chapters. In particular, the data management challenges will be of further interest for defining general design goals.

# 3 Persistent Index and Data Structures

For the design of PMem-aware index and data structures, it is of immense importance to be familiar with the characteristics of this technology type as described in the previous chapter. Data structures in general play a crucial role in all data management systems. According to [AKM+16], each design is always a compromise among the three performance trade-offs read, write, and memory amplification. Thus, any structure and variations of it must always fit the application purpose. With the emergence of new hardware technologies like PMem, however, these trade-offs can be diminished more and more. During the last few years, a number of PMem-aware data structures have already been presented that have attempted to address just that. However, from our perspective, there are some issues with these in terms of evaluation. First of all, the hardware is reasonably new and is only available since 2019, which means that some proposals have only been tested on an emulation basis. Furthermore, different benchmarks were used and mainly complex designs were compared that contained several new aspects at once. It leads to a kind of black-box testing and makes the approaches less transparent and less comparable. In this chapter, we look at the problem from a different angle by evaluating initially identified design primitives for PMem individually at the micro-level in order to make tangible statements about design decisions[1].

Our idea of identifying core primitives for data structure designs bases on the periodic table of data structures [IZA+18]. With this table and the presented systematic study, the authors claim to be able to argue about nearly the entire design space. Here, we try to support this approach by extending it by primitives of tree-based structures and evaluate different realizations of these primitives on real PMem. In [LHO+19], existing $B^+$-Tree design proposals were already neutrally and intensively tested on real hardware. Yet again, this took place at the macro-level and obscured the impact of the individual underlying ideas. Instead of a black-box (or end-to-end) approach, we focus on read and write primitives including structural changes and analyze their behavior in terms of three PMem-critical design goals: reducing writes, fine-grained access, and consistent and durable operations. Furthermore, we generalize the found approaches for various types of tree-like structures such as $B^+$-Trees, Skip-Lists, Tries, and

[1] The material in this chapter is based on [GTS20a, GTS20b].

Log-Structured Merge-Trees (LSM-Trees). The goal is to get deep insights into PMem-optimized design patterns for data structures typically found in data management systems. In summary, this chapter makes the following contributions:

- **Design Primitives for PMem**: We identify several data structure design primitives on PMem based on the literature and our own hands-on experience.
- **PMem-critical Design Goals**: We categorize the primitives into the three PMem-critical design goals mentioned above (write reduction, fine-grained access, failure atomicity).
- **Identify Mirco-operations**: To enable a white-box evaluation, we identified the typical low-level access patterns that apply to the primitives.
- **Extensive White-Box Evaluation**: We extensively evaluate and report on most of these access patterns using our Optane DCPMM-equipped server.
- **Performance Profiles**: From the results, we conclude a performance profile for each of the prime primitives and also give general recommendations.

We explicitly exclude concurrency control because we assume that it is part of upper levels or an explicit transaction manager, which is common in practice [HSH07, LLS+15]. Instead, we will study concurrency as part of Chapter 5.

The remainder of this chapter is structured as follows. We start by surveying related work in Section 3.1. Based on the literature, Section 3.2 gives an overview of the possible and unexplored design space, while Section 3.3 more specifically extracts the design primitives and corresponding rudimentary design goals. The primitives are here juxtaposed with the recognized micro-operations on which they exert an influence. According to this matching, in Section 3.4, these operations are benchmarked in detail on the primitives. Concluding from our experiments and challenges, we define general guidelines for designing PMem-based data structures in Section 3.5. Section 3.6 summarizes the contributions and insights of this chapter.

## 3.1 Related Work

In this section, we describe related work regarding two aspects: data structures made for PMem and modern approaches studying data structure designs.

### 3.1.1 Index and Data Structures for PMem

Especially index structures are traditionally not directly persisted but backed by logging or shadowing techniques to secondary storage. The properties described in Section 2.1 allow new and more fine-grained methods when designing PMem-based data structures. Several publications addressed particularly the byte-addressability and write reduction.

### $B^+$-Trees

One of the first proposals was consistent and durable data structures (CDDS) from Venkataraman et al. [VTRC11] assuming single-level storage. Their primary focus was on the $B^+$-Tree relying on versioning, atomics, and shadowing to guarantee failure atomicity. The versions become only visible once the global timestamp is atomically updated.

The $B^P$-Tree proposed by Hu et al. [HLN+14] is a $B^+$-Tree variation that reduces writes to PMem by buffering changes in DRAM first. Also, leaf nodes are kept unsorted and new entries are simply appended. With an additional volatile histogram, the future access patterns shall be predicted. It is used to allocate nodes in advance to reduce key movements and thus writes caused by splits and merges.

Chen et al. [CJ15] also proposed new $B^+$-Tree designs, which exploit indirection and keep nodes unsorted to save writes. The indirection is realized by a sorted array at the beginning of each node that maintains the index positions of the entries. They also compare the approaches and effects when adding specific primitives such as bitmaps. Their final product is the $wB^+$-Tree, which is especially targeting a fast insert and delete performance. Due to the atomic update of the bitmap after insertion into a free slot and the indirection array, this process is entirely fault-tolerant. Since the nodes are unsorted, no record shifting is necessary. A deletion simply flips a bit in the bitmap. Overall, this can massively decrease the number of writes. Nevertheless, search performance might suffer from the indirection as the binary search now must navigate between the data and the metadata.

With the NV-Tree, Yang et al. [YWC+15] introduce the term selective consistency. That means that the consistency of leaf nodes is enforced but for inner nodes, it is not. For that, the inner nodes are not flushed directly and instead rebuilt in case of failure. Again, leaf nodes are maintained in an unsorted manner by just appending the new entries. Even updates and deletes are treated as appends. By atomically updating the size field, the new data is made visible. To look up a key and get its latest version, each leaf node is scanned in reverse.

Instead of selective consistency, the FPTree presented by Oukid et al. [OLN+16] extends this idea by proposing selective persistence. So far, most described trees have been based on a pure PMem solution. The FPTree, on the other hand, is a hybrid approach where only the leaf nodes are kept in the persistent layer while inner nodes are placed in DRAM. It has the consequence that any navigation of the tree is significantly faster, but it also requires recovery measures. Compared to the NV-Tree, however, this only happens in DRAM, and compared to a transient $B^+$-Tree, this only affects a logarithmically smaller fraction. A further design principle is the use of fingerprints in the leaf nodes, which are arrays of 1-byte hashes of the keys. Scanning these first when searching for a key reduces the number of actual keys probed and thus leads to less PMem accesses. A derivative of the FPTree that is more specifically optimized for the 3D XPoint technology was presented in [LCW20], namely the $LB^+$-Tree. The authors extend the FPTree by features such as node sizes of 256 bytes or multiples of it, foresighted entry moving between two PMem lines that are written anyway, and distributed headers to avoid random access patterns for larger nodes.

HiKV [XJXS17] is another hybrid memory approach. It consists of a $B^+$-Tree in DRAM and a partitioned hash index in PMem. Hence, costly structure reorganizations only happen in DRAM and no logging is necessary. The $B^+$-Tree is mainly used to accelerate range queries but must be completely rebuilt during recovery.

In [HKWN18], the two PMem-optimized algorithms failure-atomic shift (FAST) and in-place rebalance (FAIR) are proposed and applied to the $B^+$-Tree. FAST is targeting inserts as well as deletes and shifts the sorted entries in nodes in a way that reduces the number of cache line flushes and explicit barriers. In particular, cache lines are only flushed as soon as cache line boundaries are crossed within a node during a shift. Premature cache line flushes could still lead to duplicate entries in a such a case. However, these are easily detectable, can be skipped by reads, and thus are a tolerated inconsistency. FAIR, on the other hand, is responsible for splits as well as merges and uses sibling pointers to avoid costly logging operations. Similar to FAST, duplicates and inconsistencies are still possible but are tolerable.

In [KSKN18] the authors propose the clfB-tree that utilizes cache-line-sized inner nodes to improve cache locality. To enable a higher fan-out with these small nodes they propose differential encoding. That has the additional advantage of reducing the number of writes and cache line flushes.

With the BzTree [ALML18] the authors demonstrate the usage of the persistent multi-word compare-and-swap (PMwCAS) [WLL18] operation. This operation is a software abstraction to allow latch-free atomic updates greater than 8 bytes. It maintains a descriptor table to keep track of the metadata needed to complete an operation. PMwCAS can be seen as a general-purpose method to achieve failure atomicity and concurrency and is fundamentally based on before and after images. Besides the control bits necessary for the operation to work, the nodes of the BzTree built with it consist of several further fields. The complexity is mainly reduced due to the failure atomicity realization being hidden within the operator implementations.

Similar to the $B^P$-Tree, the DPTree [ZSC+19] also buffers recent changes in a volatile $B^+$-Tree. This buffer is backed by a PMem log and merged into the base tree once it reached a defined capacity. The base tree is implemented as a volatile radix tree and a linked list of persistent leaves. Internally, the leaf nodes use the bitmap, indirection, and fingerprinting features as introduced by the $wB^+$-Tree and the FPTree, respectively. To achieve failure atomicity, the leaves are coarsely versioned based on the merging times.

As mentioned at the beginning, some of these $B^+$-Trees variants have already been reevaluated on real hardware in [LHO+19]. However, this was done again based on the whole tree designs and not based on the underlying individual primitives. That causes, for example, the $wB^+$-tree to always underperform due to the persistence of the inner nodes, which leads to costly traversal. Moreover, it is not clear, which of these design ideas can be applied to other data structures besides the $B^+$-Trees. Although, such write-optimized data structures like the LSM-Tree could be a promising alternative. Therefore, we will now briefly survey some other data structures adapted for PMem.

#### LSM-Trees

Several modern key-value stores like RocksDB [Fac20] or Cassandra [The16] are based on the LSM-Tree. There already exist some approaches to converge this concept for PMem. In [LOLS17], for instance, a first approach is given, which adapts the caching policy of SSTables. The authors utilize a dynamic strategy to identify hot blocks and move them to DRAM. On top of that, it is also allowed to access cold blocks directly in PMem.

With NoveLSM [KBG+18], the authors present a PMem-aware redesign of a well-established LSM-Tree implementation, namely LevelDB. They propose a mutable PMem-resident MemTable

implemented as a persistent skip-list in addition to the DRAM MemTable. This larger persistent MemTable is used for concurrent queries as soon as the volatile part is filled and needs to be compacted to an SSTable. Due to the byte-addressable persistence, no logs are needed in this case, and serialization costs are also eliminated.

Based on RocksDB, NVMRocks [LPD17] is another PMem enhanced LSM-Tree. It contains two possible adaptions. The first one moves everything from flash to PMem and omits unnecessary components used for flash optimization. Similar to NoveLSM, the second redesign considers the in-memory components and moves the MemTable to PMem to avoid logging and speed up recovery. In addition, they propose a multi-tiered read cache over the SSTables.

Also built on top of RocksDB, MyNVM [EGA+18] is an LSM-Tree-based key-value store with the ambition to reduce the DRAM footprint while maintaining comparable performance. The aim is to ensure that data center providers have a significantly lower total cost of ownership. Fundamentally, the authors utilize PMem as a block device and use it for a second-level cache while reducing the block size and partitioning the index. The database and logs are still kept on flash.

Concluding the discussion about LSM-Tree, it is arguable whether the log-based design is inherently appropriate for PMem since bandwidth is more important than latency in this case. Though, that is where PMem is very limited. Thus, combining it with, for instance, more fine-grained merge techniques could provide an immense benefit.

### Tries

Additionally, prefix trees, radix trees, or tries such as Adaptive Radix Tree (ART) [LKN13] were considered for PMem and have already been exemplarily implemented [FUJ20]. This type of tree could be even more appropriate for PMem as fewer key comparisons and rebalancing operations are necessary. Therefore, in [LLS+17], the authors propose three write-optimized versions of a radix tree, namely the Write Optimal Radix Tree (WORT), the Write Optimal Adaptive Radix Tree (WOART), and the Copy-on-Write Adaptive Radix Tree (CoW+ART). They commonly enhanced the trees by failure-atomic updates and path compressions. While the first is build on a basic radix tree, the latter two are adaptions of ART, as the name suggests. For the former two, memory barriers and cache line flushes were used to achieve failure atomicity. Similar to [CJ15], they also use a combination of unsorted keys, bitmaps, and indirection arrays in this process. The Cow+ART, on the other hand, makes use of copy-on-write techniques to maintain consistency.

### Hash Tables

The final data structure we will discuss here is the hash table. With PFHT [DHK+15], the authors propose a PCM-friendly cuckoo hash table variant that reduces writes and thus improves insert performance. By preventing continuous movements if a bucket is full and using larger buckets, cascading writes as typical for cuckoo hashing are avoided. However, this is traded for a worse lookup performance.

The NVC-Hashmap [SDUP15] realizes the hash table with so-called split-ordered lists [SS06]. As a result, resizing the hash table requires fewer displacements of entries and easy atomic updates.

In [ZHW18], the level hashing scheme is proposed as another approach to reducing the resizing costs and avoiding cascading writes. It consists of two separate hash levels: the main hash table and another level of overflow buckets, where each is usable by two buckets of the main level. On top of that, two hash locations and respective functions are used to achieve better load balancing.

The cache line-conscious extendible hashing (CCEH) [NCC+19] is another hash table approach adapted for PMem. Compared to the original extendible hashing, the authors add another layer of indirection and use cache line-sized buckets. Here, the most significant bits of the hash key are looked up in a directory to determine the address of a segment of buckets. The offsets of the buckets are calculated by adding the least significant bits times the bucket size on top. Thus, a search always accesses only two cache lines, i.e., the directory entry and the bucket. Essential is also the failure-atomic split operation, if resizing is necessary. Similar to [HKWN18] a failure can lead to inconsistencies, but these are easy to detect and resolve on restart.

Even beyond $B^+$-Trees, in the evaluation parts of the approaches above, we saw that most were evaluated only for operator or end-to-end performance. As this hides the details and trade-offs of the underlying design primitives, little can be inferred about custom designs or use cases. Therefore, we want to take a closer look at precisely that detailed analysis and evaluation in this chapter.

### 3.1.2 Evaluating Data Structure Design Primitives

There already exist several approaches to analyzing access patterns and hardware aspects to choose a suitable data structure or a special variation of it. Like us, some of them also break down the data structures into primitives. A good example is the Data Calculator [IZH+18] and the periodic table of data structures [IZA+18]. The authors interpret data structures as an assembly of first principles and combine analytical models, benchmarks, and even machine learning to gain deeper insights into their performance implications. The goal is to calculate this performance for any design without the need for implementation or having access to the actual hardware. They built a prototypical engine that takes a rough specification of a data structure and its primitives and can predict its performance for an input workload and hardware profile. During this process, a user can interactively exchange features getting direct feedback on the effects. A part of the engine learns a basic set of cost models for given access patterns. From these, it derives the cost for more complex operations. The learning is based on manually added micro-benchmarks per access pattern or hardware aspect. Thus, the whole prediction process heavily depends on appropriate benchmarks. That is where our envisioned evaluation and benchmarks for PMem access patterns can very well tie in. Especially in [IZA+18], the classification of design primitives and their already realized combinations is shown. Here, a large gap in the design space is revealed that has not been investigated so far. Therefore, the examination of PMem-specific designs is another connection point where we want to contribute in this chapter.

## 3.2 Data Structure PMem Adaptions

Before unveiling the individual design primitives, this section will provide a brief outline of the big picture we envisage. Along with this, we address the potentially huge design space as well as our implementations of corresponding data structures in the PMem context[2] to populate it.

### 3.2.1 Glimpse into the Design Space

To get an overview, Figure 3.1 summarizes the typical data and index structures used within a DBMS. They can be categorized into more flat or more branched data structures, with some of them converging. Each of these data structures has its particular application purpose and is more or less suitable for certain scenarios and access patterns than others. According to [AKM+16] and [IZA+18], there is always a trade-off between read, write, and space optimization. Besides the workload running on the data structures, also the underlying hardware has a high impact on performance. As we have seen in Section 3.1, basic structures like the B-Tree/B$^+$-Tree can be extended with features or combined with other structures to fit the circumstances and requirements. This extension and combination can additionally create the opportunity for new access primitives and thus access patterns. However, looking at the figure and related work above, it is evident that the design space is vast and thousands of variations exist that have not yet been explored. In particular, with respect to PMem, there are still numerous gaps. In order to be able to cover these on a large scale, we attempt to center our investigations on aspects that can be found in most structures. Accordingly, we have focused on tree structures such as B$^+$-Trees and LSM-Trees and examined access patterns and design primitives in these on a per-node and per-level basis, respectively.

### 3.2.2 B$^+$-Trees

As they have been most discussed in the literature, B$^+$-Trees also represent the largest part of the object of study in our work. Here, we were particularly inspired by the work of [OLN+16] and [CJ15]. Accordingly, we symbolically outline our implementation of the extracted and adopted features of these two works.

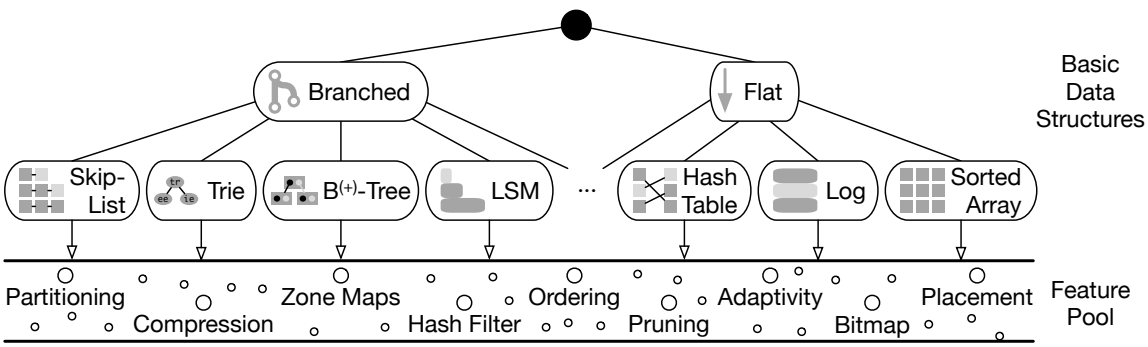



**Figure 3.1:** Overview of typical data and index structures in DBMSs.

[2]Our implementations are open source at `https://github.com/dbis-ilm/PMem_DS`.

### FPTree

From the FPTree, we were mainly interested in the fingerprinting and bitmap feature as well as the hybrid DRAM/PMem placement[3]. While the inner nodes stay sorted and in DRAM using only a size field besides the keys and children pointers, the leaf nodes in PMem were adapted like shown in the following:

```
/// LeafNode structure; M - Number of entries per leaf
struct alignas(64) LeafNode {
  p<Bitmap<M>> bits;                  ///< bitmap for valid entries
  p<array<uint8_t, M>> hashes;        ///< fingerprint array
  persistent_ptr<LeafNode> nextLeaf;  ///< the subsequent sibling
  persistent_ptr<LeafNode> prevLeaf;  ///< the preceding sibling
  char padding[PaddingSize];          ///< padding to align keys
  p<array<KeyType, M>> keys;          ///< the keys
  p<array<ValueType, M>> values;      ///< the values
};
```

We wrapped most of the fields as persistent property and used persistent pointers from PMDK. Additionally, we created and used our own bitmap to be able to extend it by hardware-optimized bit operations. The auxiliary structures are always in a separate cache line (or multiple), and the actual keys and values are aligned to the next.

The differences on an algorithmic basis are mostly related to the accesses to the leaf nodes. Therefore, in Algorithm 1 and Algorithm 2, the lookup and insert operations are shown at the leaf node level and are intended to serve exemplary purposes. The other operations and more details can be looked up in the repository. We excluded concurrency and failure-atomic actions for the time being. As stated before, concurrency will be handled at a higher level. Failure atomicity will be discussed as part of the primitive considerations and the evaluation.

During the lookup of a key, the node is traversed item by item. Since the hashes/fingerprints and the bitmap are checked first, the access optimally remains in the same cache line. Only when both are evaluated as positive, the actual key is checked. If there is no corresponding key, then the payload part is never touched.

**Algorithm 1** FPTree::lookupPositionInLeaf(leaf, key)

1: $pos \leftarrow 0$
2: **while** $pos < M$ **do**
3: **if** $leaf_{\text{hashes}}[pos] = \textsc{fingerprint}(key)$ **and** $leaf_{\text{bits}}[pos] = 1$ **and** $leaf_{\text{keys}}[pos] = key$ **then**
4: **return** $pos$
5: $pos \leftarrow pos + 1$
6: **return** $pos$

The insert part is rather trivial and simply sets the new data accordingly. Due to the bitmap, a simple append is inappropriate and, instead, a free slot must first be determined. For this, we used a 64-bit based multiply and lookup algorithm that counts the consecutive number of one bits[4]. The result corresponds to the first free position in our leaf. That is much faster than iterating through the bits individually as it constantly only costs about five CPU instructions.

[3]Steffen Kläbe did the groundwork of the FPTree reimplementation as part of a student assignment.

[4]It was adapted based on the 32-bit variant presented at https://graphics.stanford.edu/~seander/bithacks.html.

**Algorithm 2** FPTree::insertInLeaf(leaf, key, value)

1: $pos \leftarrow$ GETFREESLOT$(leaf_{\text{bits}})$
2: $leaf_{\text{keys}}[pos] \leftarrow key$
3: $leaf_{\text{values}}[pos] \leftarrow value$
4: $leaf_{\text{hashes}}[pos] \leftarrow$ FINGERPRINT$(key)$
5: $leaf_{\text{bits}}[pos] \leftarrow 1$

### wB$^+$-Tree

Next, we consider the features of indirection and again a bitmap as found in the wB$^+$-Tree. Unlike the FPTree, both leaf and inner nodes are stored on PMem. The remainder is quite similar and shown below:

```
/// LeafNode structure; M - Number of entries per leaf
struct alignas(64) LeafNode {
  p<array<uint8_t, M + 1>> slots;     ///< slot array for indirection
  p<Bitmap<M>> bits;                  ///< bitmap for valid entries
  persistent_ptr<LeafNode> nextLeaf; ///< the subsequent sibling
  persistent_ptr<LeafNode> prevLeaf; ///< the preceding sibling
  char padding[PaddingSize];          ///< padding to align keys
  p<array<KeyType, M>> keys;          ///< the actual keys
  p<array<ValueType, M>> values;      ///< the actual values
};

/// InnerNode structure; N - Number of entries per inner node
struct alignas(64) InnerNode {
  p<array<uint8_t, N + 1>> slots; ///< slot array for indirection
  p<Bitmap<N>> bits;              ///< bitmap for valid entries
  char padding[PaddingSize];      ///< padding to align keys
  p<array<KeyType, N>> keys;      ///< the actual keys
  p<array<Node, N + 1>> children; ///< pointers to child nodes
};
```

Important to note is that the first slot in the indirection array is used as the size field. Again, we made use of our own bitmap and aligned the metadata and payload accordingly. Algorithmically, however, things change a bit, as is apparent in Algorithm 3 and Algorithm 4.

The lookup is now a binary search for a key but with additional sorted indirection slots. Unlike the FPTree, the number of accesses is logarithmically smaller, but there is always a jump between metadata and payload data.

The insert part is slightly more complex than for the FPTree. Here, in addition to the data position determined by the bitmap, the position in the slot array must also be calculated for indirect order preservation. Depending on the latter, the entries must then be shifted accordingly.

### Own Adaptions

In addition to the two remakes just described, we have also created a few adaptations of our own that combine or isolate the features of these. First, we implemented a completely persistent B$^+$-Tree with sorted entries as traditionally known. The only adjustment is the 64-bit alignment of the metadata (size field and sibling pointers) and payload (keys and values). This adaption is also available as a hybrid DRAM/PMem version. Then the next isolated feature is unsorted nodes. It has the same structure as the sorted ones, but the algorithms change (e.g., append-only

**Algorithm 3** wBPlusTree::lookupPositionInLeaf(leaf, key)

```
1:  pos ← 1
2:  left ← 1
3:  right ← leaf_slots[0] /// current size of the leaf
4:  while left ≤ right do
5:      pos ← (left + right)/2
6:      if leaf_keys[leaf_slots[pos]] = key then
7:          return pos
8:      if leaf_keys[leaf_slots[pos]] < key then
9:          pos ← pos + 1
10:         left ← pos
11:     else
12:         right ← pos − 1
13: return pos
```

**Algorithm 4** wBPlusTree::insertInLeaf(leaf, key, value)

```
1:  pos ← getFreeSlot(leaf_bits)
2:  slotPos ← lookupPositionInLeaf(leaf, key)
3:  leaf_keys[pos] ← key
4:  leaf_values[pos] ← value
5:  /// shift entries in indirection array
6:  i ← leaf_slots[0]
7:  while i ≥ slotPos do
8:      leaf_slots[i + 1] ← leaf_slots[i]
9:      i ← i − 1
10: leaf_slots[slotPos] ← pos
11: leaf_bits[pos] ← 1
12: leaf_slots[0] ← leaf_slots[0] + 1
```

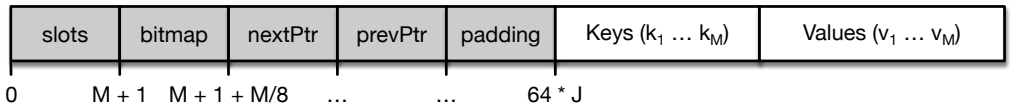
inserts and linear search lookups). We also created a hybrid and non-hybrid tree adaption, which both only substitute the size field with a bitmap without indirection or fingerprint arrays. Finally, we also introduce a hybrid wB$^+$-Tree variant that we claimed to be missing in most comparisons before. Overall, this should cover all variants of node structures that combine or isolate the features found in the literature. Their layouts are summarized again in Figure 3.2, with the metadata highlighted in gray. It also shows the size ratios depending on the number of entries per node. Interesting to note is that only the first layout has a fixed metadata structure of one cache line, while for the rest, it grows with the number of entries, possibly resulting in multiple cache lines. Removing the sibling pointers and replacing the values with child pointers will yield the corresponding inner node layouts. Furthermore, these node structures can hence also be applied to other trees or node-based structures.

### 3.2.3 LSM-Trees

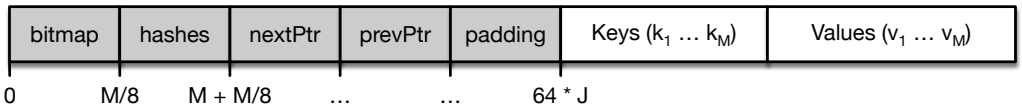
Next, we elaborate on the implementation of PMem-based LSM-Trees as presented in [Tha19]. While B$^+$-Trees are often used as indexes and could thus also be stored completely in DRAM, LSM-Trees always have one or more persistent layers. These are typically write-optimized differential structures and can form a good contrast to our B$^+$-Trees. However, according to [DI18], the design space of an LSM-Tree ranges between write-optimized tiering and read-optimized leveling policies. Tiering means that a merge process within a level is only executed


**(a)** Sorted/Unsorted data nodes.


**(b)** Bitmap-only data nodes.




**(c)** Indirection data nodes.




**(d)** Hashing data nodes.

**Figure 3.2:** Different data node layouts, where: M - number of entries and $J \in \mathbb{N}_{>0}$.

when it and all runs are full. With leveling, on the other hand, a merge is done whenever a new run within a level is created. In the most extreme cases, tiering degenerates into a log while leveling degenerates into a sorted array. Without going into too much detail, we have opted for a tiered variant here because we are aiming for write optimization.

The approach presented here is similar to NVMRocks and keeps the first level ($L_0$) in DRAM while the persistent layers can be configured to remain on PMem, disk, or both. The latter case is visualized in Figure 3.3.

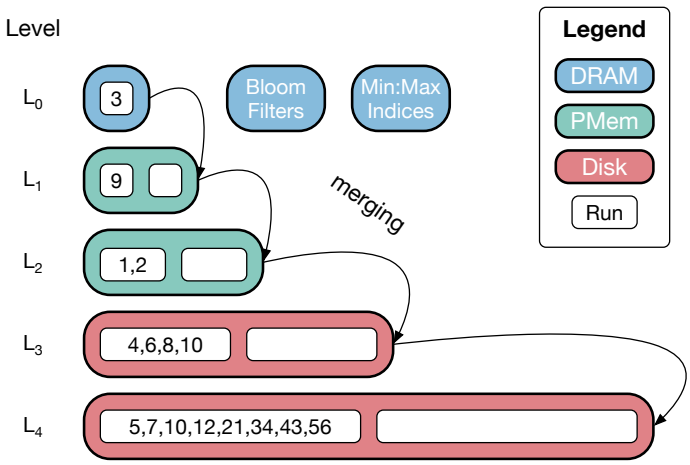



**Figure 3.3:** PMem-based LSM-Tree Layout.

The example has a buffer size of one and a so-called tuning ratio of two, meaning that the size per run doubles in the following level. Furthermore, there are two PMem and two disk levels, which are also configurable. Besides the levels and runs, each level is supported by bloom filters as well as minimum and maximum aggregates in DRAM to accelerate lookups. The object of investigation in our analysis and evaluation of the primitives is mainly the moving of runs to PMem and the merging step. That is because these processes are not covered in this form by $B^+$-Trees yet. We will discuss the details in the next section.

### 3.2.4 Skip-Lists & Tries

Finally, we look at possible PMem adaptions for tries as it was presented in [Lie20] and for skip-lists[5]. Currently, we have implemented all variants as PMem-only solutions.

For the tries, we created a standard trie and a radix tree with radix equal to two (≡ PATRICIA-Trie). As mentioned above, the ART was already adapted for PMem in [FUJ20] and was therefore not investigated further. In all trie variants, the alphabet is arbitrary and can be, for instance, bits, characters, or integers. The path from the root to a leaf composes an entire key, where the leaf contains the corresponding value. The standard trie always saves only one letter of the alphabet per node and has a fixed number of children pointers (depending on the alphabet). The combination of common prefixes in the radix tree results in path compression for memory and traversal optimization. That is utterly reasonable to avoid persistent pointer dereferencing. A possible node structure in PMem could look like the following:

```
/// Patricia node structure; M - Maximum number of key characters
struct alignas(64) TrieNode {
  p<array<char, M> key;                  ///< key prefix
  p<size_t> keylen;                      ///< length of key up to here
  p<bool> isLeaf;                        ///< true if leaf node
  p<ValueType> value;                    ///< value if leaf node
  persistent_ptr<TrieNode> children[2]; ///< pointer to children
};
```

Here, we assume the alphabet to be characters. Since each node can hold prefixes of different lengths, storing them in a fixed structure is not straightforward. One option is to reserve space with a defined maximum size, as shown above. Similar to the ART, it would also be possible to define a few structures with different capacities, chosen according to the prefix size. Another option is the allocation of the prefixes at an external location and using a fixed-size persistent pointer. While the first solution costs additional memory, the second leads to frequent jumping back and forth and thus a random access pattern. So both have advantages and disadvantages. As noticed above, `isLeaf` marks leaf nodes and that a corresponding `value` is stored. Since the radix is equal to two, there are two children pointers. Different from the structures before, a lookup operation needs a key length parameter if different lengths are allowed. Within a leaf, there would only be a simple key prefix comparison. That is why we show the complete lookup operation in iterator form in Algorithm 5.

The loop checks the key with the key part of the current node as long as the check is positive and a result is still possible. The match function depends and should be optimized based on the alphabet. Once all characters have been validated true, the value of the reached leaf is returned.

[5]The skip-list implementations were created as part of a student assignment by Alexander Baumstark.

**Algorithm 5** PatriciaTrie::lookup(key, length, value)

1: $node \leftarrow root$
2: **while** $node \neq null$ **and** $length \geq node_{\text{keylen}}$ **and** $\textsc{doesKeyMatch}(key, node_{\text{key}}, node_{\text{keylen}})$ **do**
3: **if** $node_{\text{keylen}} = length$ **then**
4: $value \leftarrow node_{\text{value}}$
5: **return** $true$
6: $node \leftarrow node_{\text{children}}[\textsc{extractBit}(key, node_{\text{keylen}})]$
7: **return** $false$

During the traversal, the child is chosen based on the most significant bit of the last character of the node.

The insert operation proceeds similarly. The difference is that the value is updated when found, and a new node is inserted when not found. An update also includes the possibility that an inner node is promoted to a leaf node. The value is then set for the first time. However, a new node can lead to a split if the last comparison is negative and, thus, the search stops between two nodes. Therefore, two new nodes are inserted. One leaf node with the new value and the unequal suffix and an inner node above with the common prefix of the last compared node. Furthermore, pointers and prefixes in the existing nodes have to be adjusted. In general, the operations in these trie variants can be seen as a special form of other tree structures with only one element per node. Hence, its primitives are already covered by our considerations based on the $B^+$-Trees and LSM-Trees above and in Section 3.3.

The skip-lists are available having a single key-value pair per node and a write-optimized variant with multiple key-value pairs per node. While the former is more simple, it also leads to a higher rate of pointer chasing during traversals and also more frequently produces small distributed writes. The latter can compensate for this by grouping key-value pairs together. This results in the following structure for a node:

```
/// SkipNode structure; M - Maximum number of levels, N - Bucket size
struct alignas(64) SkipNode {
  p<Bitmap<N>> bits;              ///< bitmap for valid entries
  p<KeyType> minKey;              ///< SMA min
  p<KeyType> maxKey;              ///< SMA max
  p<size_t> nodeLevel;            ///< height of node
  array<persistent_ptr<SkipNode>, M>
    forward;                      ///< pointers to following nodes
  char padding[PaddingSize];      ///< padding to align keys
  p<array<KeyType, N>> keys;      ///< the keys
  p<array<ValueType, N>> values;  ///< the values
};
```

It is noticeable that this structure is almost identical to the node structure of the $B^+$-Tree except for a few extra fields. The operations are also more or less similar. For a lookup, for instance, the corresponding node is first searched following the forward pointers. The search of the key within a node is then the same as with $B^+$-Tree nodes. In the current implementation with unsorted entries, this is a linear search checking first the bitmap and then the key. However, this could be sorted or use fingerprinting or indirection features as well. Therefore, in the following sections, we will consider both structures as interchangeable when extracting and evaluating the structural and access primitives.

## 3.3 Design Primitives

After considering the basic data structure adaptions for PMem, the next step is to extract the design primitives from these and investigate their impact in various access scenarios. The primary goal is to reveal their trade-offs to facilitate design decisions. It is necessary to do this using white-box tests to eliminate side effects in the measurements. Therefore, preferably only one primitive should be exchanged during comparison. As a tangible result, we envision a profile per design primitive, from which performance and memory implications for each type of access pattern are derivable.

### 3.3.1 Design Goals

To classify the primitives, we first identified three central design goals that the primitives address. These are based on the PMem properties and related work.

#### Reduce Writes

We have seen that PMem has a lower write endurance than DRAM. Furthermore, there is a read-write asymmetry, where writes require more energy and time than reads and cause the above-said cell wear. That leads to the first design goal of reducing writes (DG1) and trading off with more reads. For example, consistency constraints can be relaxed, or sorting can be abandoned, both compensated by more frequent reading.

#### Fine-granular Access

Compared to disks, with PMem, storage is now byte-addressable. Even if this is limited to cache lines with typical 64-byte granularity in x86 architectures, much more fine-grained access methods are now possible (DG2). That was previously realized by combining DRAM and disk. For instance, while the structures are held and processed in DRAM, an append log on disk traditionally provides persistence. With PMem, both byte-addressability and persistence are enabled on the same device. That introduces fine-granular operations to the storage layer.

#### Failure Atomicity

The byte-addressability combined with the direct load and store semantics leads additionally to a zero-copy memory mapping. It opens up new possibilities for realizing consistency constraints and durability (DG3). For example, atomic primitives can now be used for efficient persisting, in addition to concurrency. The primitives for this design goal explicitly tackle our first posed data management challenge in Section 2.3.

In the following, we extract the design primitives and access patterns focusing on tree-based structures from the literature and connect them with our defined design goals.

### 3.3.2 Overview and Definitions

In order to be able to classify the terms correctly, our two axes – namely design primitives and micro-operations – should first be explained more precisely.

**Definition** (Design Primitive)**.** In our context, we define a design primitive as an indivisible layout or access concept, as it was similarly specified by [IZA+18].

It is necessary to break down the possible primitives taking into account the properties of PMem, to recognize their advantages and disadvantages. To this end, we examine the approaches as described in Sections 3.1 and 3.2 and map the ideas to the appropriate design goals.

Related to the derived primitives are typical micro-operations, as found for trees or nodes.

**Definition** (Micro-Operation)**.** We define a micro-operation in this context as a low-level access pattern whose (logical) result is independent of the employed primitive(s).

Lower operations – which would yield a different result or are irrelevant depending on the primitive – are of no interest because they are no longer comparable. An example of such an operation would be the shifting of entries in a node after a delete or before an insert. If a bitmap is used, this step is unnecessary. On the other hand, higher-order macro-operations can be formed by combining the micro-operations. Typical examples are get, insert, update, delete, and scan on a data structure. Therefore, we have classified the micro-operations into read-, insert-, and erase-based as well as recovery operations. Table 3.1 summarizes our findings and the relation of design primitives and micro-operations. For design primitives that are not applicable or only indirectly relevant to a few operations, we have left the cells blank. In the following subsections, we will describe these in more detail.

### 3.3.3 Micro-Operations

By studying micro-operations, it is possible to identify bottlenecks and optimization potentials that would remain hidden at the macro-level. For example, inserts are often faster in hybrid DRAM/PMem solutions than in fully persistent approaches, almost regardless of the node layout. As already mentioned in related work, this causes, e.g., the wB$^+$-Tree to perform always worse than the FPTree only due to the more costly traversal to the leaf. Thus, maintaining both the access patterns and the design space concise is crucial for a meaningful comparison.

The first category starts with the micro-operation of searching (or lookup) for a key within a node. That is strongly dependent on the selected node layout and, thus, possible access methods. According to our definition, it is the smallest possible operation still giving the same result in the end, no matter how it is retrieved. To reach a target node, there are typically two ways to navigate in a tree, either vertically (tree traverse from top) or horizontally (tree iterate from left/right). Combining navigations and the searching within nodes builds macro-operations like get and scan.

**Table 3.1:** Design primitives and micro-operations for PMem-aware trees (✓ → applicable).

| PMem-aware Trees | | Read-based | | | Insert-based | | | | Erase-based | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | *design primitives* \ *micro-operations* | *Node Search* | *Tree Traverse* | *Tree Iterate* | *Insert in Node* | *Node Split* | *Move Node* | *Merge Level* | *Erase f. Node* | *Balance Nodes* | *Merge Nodes* | *Recovery* |
| **DesignGoal1** (reduce writes) | sorted [VTRC11, HLN+14, LPD17, ALML18, EGA+18, HKWN18, KBG+18] | | | | ✓ | ✓ | ✓ | | ✓ | ✓ | ✓ | |
| | unsorted [HLN+14, CJ15, YWC+15, OLN+16, LLS+17, ALML18, KSKN18, ZSC+19] | | | | ✓ | ✓ | ✓ | | ✓ | ✓ | ✓ | |
| | bitmaps [CJ15, OLN+16, LLS+17, KSKN18, ZSC+19] | | | | ✓ | ✓ | | | ✓ | ✓ | ✓ | |
| | indirection [CJ15, LLS+17, ZSC+19] | | | | ✓ | ✓ | | | ✓ | ✓ | ✓ | |
| | hashing [OLN+16, XJXS17, ZSC+19, LCW20] | | | | ✓ | ✓ | | | ✓ | ✓ | ✓ | |
| | 2-way merge | | | | | | | ✓ | | | | |
| | K-way merge | | | | | | | ✓ | | | | |
| | split move [VTRC11, HLN+14, CJ15, YWC+15, ALML18, HKWN18, KSKN18, ZSC+19, LCW20] | | | | | ✓ | | | | | | |
| | hybrid placement [HLN+14, YWC+15, OLN+16, LPD17, XJXS17, EGA+18, KBG+18, ZSC+19, LCW20] | ✓ | ✓ | | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| **DesignGoal2** (fine-grained access) | linear search [HLN+14, YWC+15, ALML18, HKWN18, KSKN18] | ✓ | ✓ | ✓ | ✓ | ✓ | | | ✓ | ✓ | ✓ | ✓ |
| | search with bit check [CJ15, OLN+16, KSKN18, ZSC+19, LCW20] | ✓ | ✓ | ✓ | ✓ | ✓ | | | ✓ | ✓ | ✓ | ✓ |
| | search with hash probing [OLN+16, ZSC+19, LCW20] | ✓ | | | ✓ | ✓ | | | ✓ | ✓ | ✓ | ✓ |
| | binary search [VTRC11, HLN+14, YWC+15, ALML18] | ✓ | ✓ | | ✓ | ✓ | | | ✓ | ✓ | ✓ | ✓ |
| | search with indirection [CJ15, ZSC+19] | ✓ | ✓ | ✓ | ✓ | ✓ | | | ✓ | ✓ | ✓ | ✓ |
| | split copy [OLN+16] | | | | | ✓ | | | | | | ✓ |
| | cache sensitive [CJ15, YWC+15, OLN+16, ALML18, KSKN18, ZSC+19, LCW20] | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| **DesignGoal3** (failure atomicity) | PMDK Transactions [GBS18, Int20d] | | | | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| | PMwCAS [ALML18, WLL18] | | | | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| | individually [VTRC11, CJ15, YWC+15, OLN+16, LLS+17, LPD17, XJXS17, HKWN18, KBG+18, KSKN18, ZSC+19, LCW20] | | | | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| | Evaluation reported in Experiment No. | **E1** | **E2** | **E3** | **E4** | **E5** | **E6** | **E7** | **E8** | **E9** | **E10** | |

The next category contains the micro-operations that can be triggered by adding new data to the structure. Foremost, this is the placement of new records such as key-value pairs into a node. Similar to the lookup, this operation is not divisible further than inserting a record without position reference into a node. Deeper operations return a different result depending on the primitive, such as the position or the division of the data. Inserting new data can cause further structure-changing operations. For $B^+$-Trees, tries, and skip-lists, this is usually a type of split that leads to the allocation of new nodes and the division of content. These two steps can be considered as separate micro-operations, although the allocation is independent of the primitive for the same node size and thus can be omitted. Hence, it is mainly the type of data separation that is of interest during a node split. The macro-operations to insert or update entries in a tree would require the micro-operations lookup, traverse, insert, and split. Considering hybrid structures, such as LSM-Trees in general or more specific solutions like the $B^P$-Tree, introduces further structure-changing operations. That is, for instance, the movement or migration of nodes, runs, or levels from DRAM to PMem. Depending on the realization, data can just be copied or must still be adapted beforehand, e.g., by sorting. Another micro-operation is to merge multiple nodes into a new larger one. A representative example is the compaction/merge step of a level in LSM-Trees.

Next, we consider operations that downsize trees and remove data. The most basic micro-operation here is the removal of entries from a node. The cases where these are implemented by inserting tombstone records, such as for LSM-Trees, are already covered by the micro-operations *insert in node* and *merge level*. Thus, the erase operation checks primitives that actually delete data instead. Such an erase can trigger an underflow in a node, which can be handled by balancing or merging with other nodes. The typical delete macro-operation in a tree comprises searching, traversing, erasing, balancing, and merging.

The final category is recovery. During the examination of it, we realized that the steps are basically composed of the operations of the read category plus the recreation of the volatile DRAM parts. Thus, no new or other micro-operations based on PMem are arising anymore. Only the reconstruction can now be done in bulk rather than by individual inserts, which is why this category is still listed separately here. However, since our focus is on PMem and not DRAM, the reconstruction step is left out of the evaluation. What is more interesting for us are the measures, which have to take place before a failure occurs to restore the data in the first place. This is not an explicit micro-operation but is reflected in all modifying procedures and is thus considered more concomitant as part of the primitives of DG3.

### 3.3.4 Primitives

In the following, we describe the set of our found design primitives. For the sake of clarity, we will explain these alongside the design goals they pursue.

#### Reducing Writes

Per the first design goal, i.e., reducing writes, the node layout was reconsidered. The main consensus in the literature was to keep the data nodes unsorted to avoid costly shifting operations. However, this can cause the search for entries to take longer. To compensate for this, features such as indirection, hashing, and bitmaps, as well as combinations of these, were

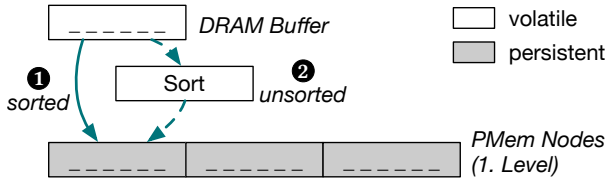


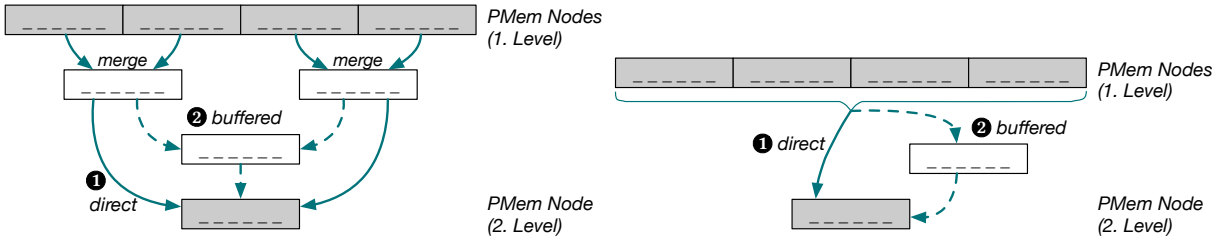

**Figure 3.4:** Move DRAM data to a PMem node.

introduced. In Section 3.2 and Figure 3.2, we already discussed the conceivable layouts in detail. What has become clear is that these options are not limited to $B^+$-Trees, but apply to almost all tree-like structures. To have a fair comparison in the evaluation (see Section 3.4), all of our implementations align the search structure at the beginning and the keys to cache-line boundaries.

The node layout is also decisive when moving nodes from DRAM to PMem, as it occurs with LSM-Trees. For example, initially, the data could be buffered in DRAM and later transferred to a sorted PMem-resident node. The sorting can be done ❶ directly during buffering or ❷ during propagation. Figure 3.4 illustrates these two methods. If presorted, the DRAM data is simply copied to the PMem node, or else it must be sorted before a copy operation. The cost of sorting, in the first case, would be spread over time, while in the latter, it is done only once but to a greater extent. However, the sorting should always already be done in DRAM to prevent unnecessary writing to PMem. Which method runs faster or whether the copy process to PMem costs the majority of performance anyway is hard to assess. However, the size of the node will play a decisive role.

Also more commonly found in LSM-like structures is the merge process of multiple nodes into a larger one. That is typically necessary if all nodes in a level are filled and need to be merged to a new node at the next level. Basically, we see two common approaches, namely the 2-way and the K-way merge. Although the design space for merge algorithms is more exhaustive, we think these two methods cover the main distinctions. Figures 3.5a and 3.5b visualize both methods. The 2-way merge only merges two nodes at once until all nodes are processed. The K-way merge, on the other hand, simultaneously compares all nodes at once and can directly produce a final result. The former way is easier to implement but also leads to more intermediate results. For both methods, the final result can be written either ❶ directly by performing the merge in PMem or ❷ by merging the nodes in DRAM first and copying the complete result to PMem. Unlike the previously considered micro-operation (moving of nodes), no reordering is necessary, so the number of bytes written is for unique keys the same. However, a direct write-out initiates many smaller writes, which can lead to cache lines being flushed multiple times in the case of unpredictable flushes by the operating system. It becomes even more problematic when duplicates exist, i.e., the same key was updated in multiple nodes, and already written data has to be updated. The second variant via DRAM, however, requires an additional copy step.

Splitting full nodes, as found in $B^+$-Trees, skip-lists, and to some extent tries, can be done in two ways. The first primitive, *split move*, involves fewer writes and thus more closely pursues DG1. The idea is to move all keys greater than the split key to a new node. Alternatively, the data could also be moved to two new nodes. That would be slightly easier but would also trigger double the writes and tends thus not to be suitable. The second split primitive is applicable when a bitmap is present in the nodes. It copies the complete full node and only resets the




**(a)** 2-way merge into a PMem node. **(b)** K-way merge into a PMem node.

**Figure 3.5:** Design primitives for merging multiple nodes to the next level.

bits in the bitmap for greater keys in the original node. The inverted bitmap, i.e., enabling the smaller keys, is then applied to the new node. Depending on the allocation process, this split type involves more written bytes but could be faster due to algorithmic simplicity and exploitation of more fine-granular accesses. Therefore, this method is rather fitting DG2.

A final step to reduce the write load is given by the hybrid approaches storing only a necessary part in PMem and managing the rest in DRAM. The best example is the selective persistence of the FPTree, where the inner nodes are in DRAM, and the leaves are on PMem. This type of data placement has an impact on almost all micro-operations. Its influence is thus quantified at the very beginning of the evaluation.

### Fine-granular Access

The different search and access methods assigned to DG2 depend strongly on the underlying node layout. Therefore, the design primitives in this category have, for the most part, already been explained along with it. While a linear search – where all keys are examined – is always a possibility, binary searches are only feasible if some kind of sorting is given. This can be done either by sorting the entries themselves or by an indirection array. If a bitmap is present, the corresponding bit must be checked or flipped, in addition to the entries in each case, so that no deleted or unset data slots are read. With hash probing, a linear search can be accelerated. However, this always requires extra hashes to be calculated. At this point, other algorithms such as interpolation or exponential searches would also be conceivable but have not been mentioned in the literature so far.

The major innovation with PMem is now that the data does not have to be loaded into DRAM first but can be processed or searched directly. Especially the support by features like indirection, bitmaps, and fingerprints/hashing seems to be very well suited for cache sensitive and fine-granular accesses without having to touch the entire node.

### Failure Atomicity

Regarding the last design goal, we extracted three different methods to achieve failure atomicity. The easiest method for developers is by using PMDK transactions. It provides a general-purpose way by encapsulating appropriate sections to happen atomically. As we discussed in Section 2.4, this process usually costs more performance than necessary. Another general-purpose solution is the PMwCAS operation, which provides CAS operations for ranges bigger than eight bytes, as we explained in Section 3.1. There are similar concerns as with PMDK transactions, though.

The final method that was mainly used in the literature is to individually persist the data using flush and fence instructions.

In principle, the methods are relevant for all micro-operations that involve writes to PMem. While we could simply place PMDK transactions around any micro-operation, manually setting instructions must take into account the individual operation and also the primitive(s) used. Thus, deleting an entry in a node with a bitmap can be efficiently implemented by a bit flip, a flush, and surrounding fences. For sorted or unsorted nodes without auxiliary structures, however, the procedure becomes more complicated since the entries on the right must be shifted one to the left, and the counter must be decremented. To make this procedure fault-tolerant, there is probably no way around a before or after image, like in PMDK transactions. Only if duplicates are not allowed, one could detect and fix a failed delete based on such duplicates, as it was similarly done in FAST and FAIR [HKWN18]. In our experiments, we mostly left out correct failure atomicity because of its individual treatment and just persisted all changes. Instead, we consider them exemplarily using the micro-operations *move node* and *merge level*. With these, an individual implementation by flushes and fences independent of the primitive is easily possible, which makes it more comparable with the transactional approach. The idea is to introduce an array storing the pointers or offsets for each level to the next free node, as shown in Figure 3.6.

These pointers can be atomically updated – if aligned correctly – after the move or merge process is finished. It is possible since, on x86 architectures, 8-byte aligned writes are failure atomic. Hence, by limiting the size of the pointers or offsets to 8 bytes, we can completely avoid transactions. Assume a crash occurs while the write operation in node 1 is not complete yet. After the restart, no undo is necessary because the pointer is still at position 0. A new merge or move process would just overwrite the partial changes in node 1. Similarly, this could also be implemented with PMDK transactions or PMwCAS instructions (instead of atomics), only encapsulating the pointer adjustments. It would be necessary if larger pointers than 8 bytes are required. Data consistency remains unchanged irrespective of whether the entire node or just the pointer is added into the log. Consequently, the performance penalty of PMDK transactions could be significantly reduced. All of these methods, we refer to as *individual failure atomicity* in the evaluation. If all the changed data is simply flushed, ignoring the reordering abilities of the CPU, we term this as *no failure atomicity*. The third variant we evaluate encapsulates the complete micro-operation and is marked as *PMDK transaction*. Regarding Table 3.1, all variants fall under DG3. The individual and no failure atomicity variants could also be counted to DG1 as they reduce the number of writes, whereby the latter serves more as a theoretical baseline.

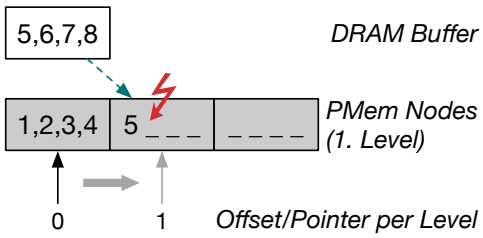



**Figure 3.6:** Individually realizing failure atomicity with an LSM-Tree as an example.

### 3.3.5 Extendability

As it already became clear in Section 3.2, the design space is tremendous and, thus, Table 3.1 only covers an excerpt. However, it can be easily extended by more design primitives and micro-operations. Conceivable extensions can be expected, for example, in the fields of hardware utilization and concurrency. Concerning the former, we already applied cache-line alignment and specific CPU instructions. The latter was excluded as it does not fit into our micro-consideration. Instead, concurrency control is discussed separately and on a higher level in Chapter 5. Also, failure atomicity could be explored more deeply. For the time being, we have focused on PMDK transactions and individual persist operations. Finally, future proposals for the node layout – that we have not thought about yet – can also be incorporated later. In principle, our currently considered micro-operations and primitives should already cover a large part of PMem-aware trees while serving as inspiration for future extensions or applications to other technologies.

### 3.3.6 Metrics

Now that the primitives and operations and their mapping have been inspected, the next step is to evaluate the applicable options from Table 3.1 using appropriate white-box benchmarks. The correspondingly assigned experiments are noted in the last row of the table. Thus, all PMem-based operations are fully covered. Before we delve into the experiments, however, we must first consider relevant metrics. From our point of view, the throughput does not provide usable values at this point because we are at the micro-level. The average time of the micro-operations, i.e., the latency, is more meaningful here in terms of performance. Furthermore, performance counters on the hardware level, such as cache misses or instructions per cycle, can also provide valuable information. We think that especially the number of flushes (persist operations) and written bytes are crucial factors due to the read-write asymmetry. General memory consumption is also a factor that should not be ignored since PMem typically has less capacity than disks.

## 3.4 Evaluation

The benchmarks in this evaluation represent the micro-operations on tree-like data structures as presented in the previous section. From the design primitives, we focused on the following candidates for the node layout: *sorted*, *unsorted*, indirection + bitmap (*indirection*), hash-probing + bitmap (*hashing*), and *bitmap* only, in most of the experiments. Accordingly, we reviewed the access primitives: *binary* search with and without using *indirection* as well as *linear* search with and without using *hashing* and a *bitmap*. In Section 3.2, we already discussed how we reimplemented these approaches to isolate the primitives. A brief recap will still be given for each corresponding experiment. The primary goal is to evaluate the design primitives independently of their original context and identify their benefits and costs. As a comprehensible result, a performance profile for our main primitives shall be created exemplarily at the end.

As already mentioned during the discussion about failure atomicity, we mainly report the results for manually persisting the modified data. We will compare the impact of individual

handling of failure atomicity and PMDK transactions only for the *move node* (experiment E6) and *merge level* (experiment E7). Alternatively, we could also use PMwCAS operations. Since these were not yet integrated with PMDK at the time of the experiments, they were omitted for the time being.

Regarding the data used, we mostly worked with fixed-size keys and values being 8-byte identifiers and 16-byte tuples ($<int, int, double>$), respectively. In cases where this format differs, this will be indicated. The chosen size of the values also corresponds to the size of a persistent pointer (e.g., to the actual payload). As it was already shown in Figure 3.2, we stored the keys, values, and child pointers in separate arrays within the nodes to provide better locality benefits when iterating through the keys. Furthermore, the payload was always cache-line aligned. The fill ratio of the trees was set to 100%, but this ratio is actually irrelevant in most experiments since we access predefined positions. With the different node layouts, also the number of maximum elements differs. That is in particular apparent for smaller node sizes.

In all benchmarks, we created several nodes (and sometimes even trees) with the same content to avoid reporting cache instead of PMem performance. Together, they stretched over a multiple of the LLC size. For each benchmark iteration, a different randomly chosen instance was accessed. Thus, it is rather unlikely for the CPU to prefetch or cache the correct instances. For comparison purposes, we nevertheless ran the measurements additionally always on the same instance, what we refer to as a cached case in the text. In any scenario, the data points in our graphs are supported by several thousand iterations. Like the data structures, also our benchmarks and used scripts can be reviewed in our public repository[6].

### 3.4.1 Read Operations

#### Node Search (E1)

First of all, we consider performance of the most basic micro-operation, namely searching for a key within a node. It is used for nearly all macro-operations such as get, update, and delete and, thus, a crucial performance indicator. The most decisive factor is the node layout and the corresponding possible access primitives, which we have all tested. We varied the node size and the position of the target key, as these can also have a large impact on performance. As a result, we expect binary searches to be faster with and without the help of an indirection array for the middle and last node positions. Figure 3.7 visualizes our measurements. Besides the PMem implementations, we included the results for all variants when running on DRAM summarized in one curve. That should serve as a baseline.

It is visible that the binary search is performing worse than expected. Just when the key is exactly in the middle and, thus, only one access happens, it can keep up with the linear approaches in the PMem case. Also, contrary to our expectations, indirection does not yield any advantages over a direct binary search. It should be noted, however, that it takes much fewer writes to insert and delete, which we will look at later. The problem that causes the slowdown is the alternate access to the indirection and key array, which are all the more apart, the larger the node. The direct binary search only scans through the latter. With one or two exceptions, the hashing approach is the fastest of the linear methods. That is since most of the comparisons are made in the front cache line(s). Only if a hash comparison is positive, the

[6] PMem-based Data Structures - `https://github.com/dbis-ilm/PMem_DS`

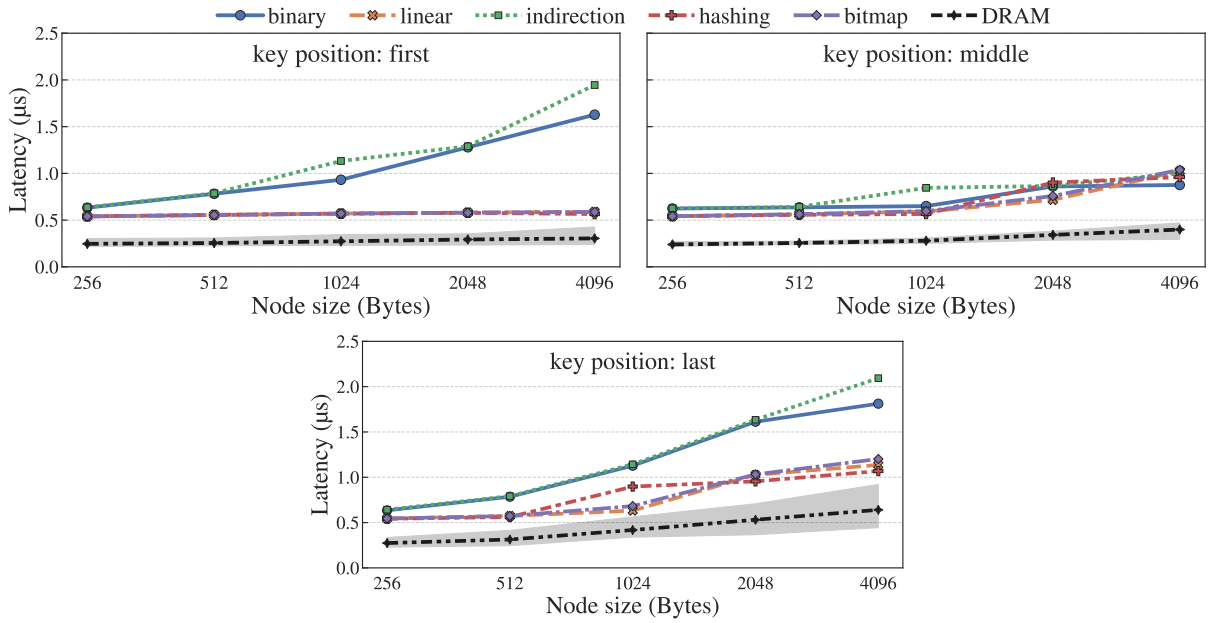



**Figure 3.7:** Searching for a key within a node (E1).

corresponding key is checked. Since in our scenario the hashes are unique, it amounts to a single key comparison. Thus, on average, the fewest cache lines need to be loaded from PMem.

When doing the same experiments with DRAM-resident nodes, binary searches performed much better, especially with the target key in the middle. In a cached case, both for DRAM and PMem, even the back access areas were faster with binary search. However, it is important to mention that the unsorted approaches are not suitable for inner nodes in $B^+$-Trees since the search is based on key ranges and not key equality. That is particularly relevant for hybrid structures as binary search performs better in DRAM anyway.

Furthermore, with the indirection and hashing approach, it can be seen that there are sometimes rapid rises in latency when the node size is increased. It is related to the also increasing search structure at the beginning of the nodes. Thus, for 1 KiB and 2 KiB already two cache lines are needed for the auxiliary structures, and for 4 KiB, three. However, unlike indirection, hashing usually reads fewer of the cache lines. Overall, both approaches (and slightly also the bitmap) have a higher memory footprint.

Staying on the subject of memory footprint, Table 3.2 shows the exact number of possible entries per node depending on their size and layout as well as the total PMem space required for a given number of 50M records. Without a search structure – as with sorted and unsorted nodes – more records will naturally fit in a node. If auxiliary structures like the indirection or fingerprint array are introduced, one additional byte per entry is required. The bitmap requires an additional bit per entry. As we already clarified above, for a fair comparison, we have aligned the base node structure as well, so that the keys also start in a new cache line. In percentage terms, it can be seen that smaller nodes entail a higher memory overhead. However, this is also very dependent on the size of the keys and values. The poorer utilization of space also leads to potentially longer traversing paths.

**Table 3.2:** Calculated number of records per node and memory consumption of a node chain (50M records) for a given node size.

| | Node Size | | | | | |
|---|---|---|---|---|---|---|
| | 256 B | 512 B | 1 KiB | 2 KiB | 4 KiB | Unit |
| Base Node | 9 | 19 | 41 | 83 | 169 | records/node |
| | 1,32 | 1,25 | 1,16 | 1,15 | 1,13 | GB |
| Aligned Node | 8 | 18 | 40 | 82 | 168 | records/node |
| | 1,49 | 1,32 | 1,19 | 1,16 | 1,14 | GB |
| Node with Search Structure | 8 | 18 | 37 | 79 | 160 | records/node |
| | 1,49 | 1,32 | 1,25 | 1,22 | 1,19 | GB |
| Overhead Aligned | +13% | +6% | +3% | +1% | +1% | |
| Overhead Search Structure | +13% | +6% | +8% | +6% | +5% | |

Overall, apart from the higher memory requirements, we see the hashing approach as the best solution for searches in a persistent node. For nodes that are in DRAM (e.g., inner nodes) or that are very likely to be cached, the traditional sorted layout is better suited.

### Tree Traversal (E2)

In the next experiment, we consider the traversal cost from the root to the leaf level, as known from navigations in $B^+$-Trees. Therefore, most of the accesses concern inner nodes. Since the search within nodes is already covered in experiment E1, we only measure timings for dereferencing and chasing pointers. Again, we want to avoid prefetching and cached measurements. Therefore, a random child pointer is always chosen and chased. Without the search, the node layout is hardly decisive. Instead, we thus compare the times for traversing nodes placed in PMem or DRAM. Merely the last access ends in a persistent leaf node, reflecting the idea of hybrid data structures and placement. Regarding the parameters, we varied the depth of the tree or the length of the node chain. Since the node size made virtually no difference here, the results are reported in aggregate form. We expect that the latency per node visit increases by about the corresponding idle random read latency measured in Table 2.2, depending on the technology. Our results are illustrated in Figure 3.8.

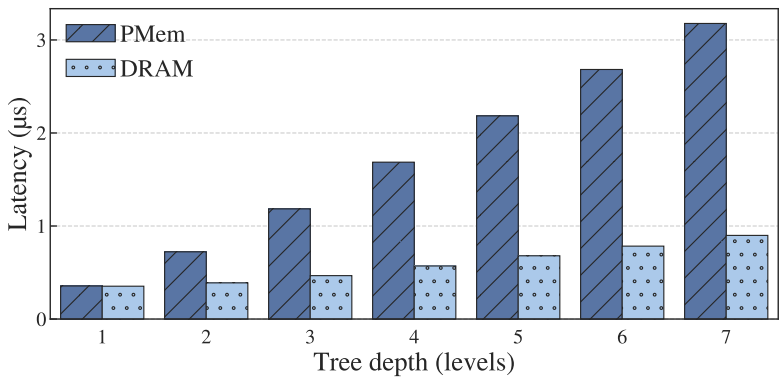



**Figure 3.8:** Traversing a tree w/o search (E2).

As it can be seen, with increasing tree depth or frequent pointer chasing in general, the use of PMem can lead to a drastic loss of performance. For DRAM nodes, the latency increases by about 50-100 ns for each extra level, like anticipated. In the case of exclusively used PMem-resident nodes, on the other hand, each level increases the latency by about 400-500 ns. That is already almost 50% more than the latency measured at the beginning and also reported in [LHO+19, vRVL+19]. It may be due to the additional software overhead (such as PMDK) and possibly not yet fully developed optimizations on the compiler and hardware level.

Apart from that, it becomes clear anyway that all structures would benefit greatly from a hybrid layout. In the previous experiment, we had already pointed out that, in the case of DRAM, sorted nodes can be searched faster and are also necessary. Both direct and indirect sorting is possible. The direct search is more memory efficient, and the indirect one saves write operations, but this is not as vital for DRAM. However, it should always be considered that the DRAM parts have to be restored in case of a crash and, thus, the restart takes longer. A decision must be made here depending on the use case and requirements. For example, if performance is most important and failures are rare, a hybrid approach should definitely be applied.

#### Tree Iterate (E3)

The orthogonal navigation direction is the horizontal traversal of nodes, also called scan or iteration. Since we traverse the nodes not in an ordered manner, we prefer to use the term iterate to avoid confusion with range scans. This micro-operation is the last in the read category that we want to wield. Unlike the experiment before, this one includes pointer chasing and iteration over all keys and values. The number of entries per node has a great influence here. Therefore, we limit the number to the maximum possible based on the nodes having a search structure (see Table 3.2). The benchmark was implemented by letting a tree grow horizontally by making the single inner node, i.e., the root, progressively larger. The size of the leaf nodes also has an influence, but the results are proportionally the same. Hence, we show representatively only the measurements for node sizes of 1 KiB. There are three variants for the access primitives. The sorted and unsorted approaches use the same algorithm since the order is not of interest, and the key and value arrays can be iterated directly. The three approaches with a search structure also share the same algorithm. They all have a bitmap that must be checked for valid entries. Since this causes branching in the loop and alternating accesses, we expect it to perform poorer than the direct variant. The third variant is enabled if an indirection array is present. As this also stores the number of entries in the first byte, the indirection array can be traversed to the fill level using this number, which thus only refers to valid entries. However, since this approach does not necessarily correspond to the order of the key and value arrays – and, hence, results in more random accesses – better performance is rather disputable. Nevertheless, we have tested all three variants and present the results in Figure 3.9.

Interestingly, the iteration via indirection is even worse than expected and slower than the bitmap variant. As a consequence, it can be stated that even if an indirection array is available, the bitmap should be used for iteration. It might look different though if the order should be respected when iterating. In any case, as expected, the direct approach is the fastest. Based on its performance, the overhead of the bitmap is 31%, and that of the indirection is 58%, in the largest tested case. Especially alternating between the cache lines (bitmap/indirection slots, key, and value array) as mentioned above is a drawback. However, the nodes in this scenario are 100% filled, which is already favorable for the bitmap procdedure. Thus, the branch prediction

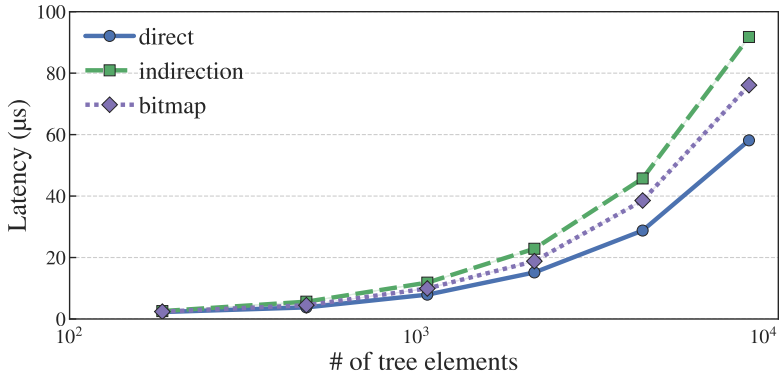



**Figure 3.9:** Iterating through nodes (E3).

should always be positive, and moreover, the number of checked entries is the same as with the other methods (i.e., the loop passes). In the case of indirection and direct iteration, the loop only needs to step over the number of entries actually present. Since our iteration function only reads the keys and values and copies them into a variable, its effort is quite manageable. Also, the bulk of performance goes thus to the algorithmic part of iterating. We assume that with an increasing complexity of the underlying function, the approaches converge in terms of performance.

Overall, we saw that for iterating through persistent data nodes, the direct iteration used for a plain sorted and unsorted layout performs the best. In order to realize range scans based on this micro-operation, sorting must be available. It remains open to verify the overhead of on-demand sorting per node versus the already sorted approaches. Experiment E6 will provide partial insights into this. Finally, this experiment has shown us, in particular for DCPMMs, that frequent alternating between non-sequential cache lines should be avoided.

### 3.4.2 Insert-based Operations

#### Node Insert (E4)

The basic operation for inserting new data into a tree is the placement of a new entry in a target data node. Again, like in experiment E1, the node layout is the main category of the primitives as it determines what else has to be modified. In the benchmarks, we varied the node size and the insert position regarding the order of the already existing keys. Since we now modify data on PMem, we additionally report the number of bytes modified and the actually written bytes to the device (a multiple of cache lines). In preparation, all key-value pairs except the target key are inserted into a node with precisely the space needed. For example, if a pair is to be inserted at the first position in a node with ten slots, the pairs with the keys from 2-10 are inserted in advance. The measured part is confined to the insertion of key 1, i.e., the search for the target position is not included. In order to obtain the macro-operation insert into a tree, we can add approximately the time to traverse, the respective node searches, and the node insert as presented here. Theoretically, the sorted layout has the most overhead since other entries must also be moved. That results in many writes and flushed cache lines. Therefore, we expect it to perform the worst for this micro-operation. In the other approaches, however, the

payload is only appended. The difference becomes apparent when adapting the metadata. Our results are shown in Figure 3.10.

It is quickly evident that maintaining sorted nodes in PMem has an essential performance impact. The direct sorting approach worsens as the node size increases, corresponding to the higher number of bytes written. It can only keep up with 256-byte nodes, which can be justified by the approximately equal number of flushed cache lines. The unsorted variants, on the other hand, perform significantly better. For instance, for the plain unsorted case, only the key and the value need to be appended and the size field updated. In the hashing and bitmap case, the hash and bit are set accordingly instead of the size field. The indirection approach performs much better than the directly sorted layout, especially when the key position is in the front and all slots have to be shifted to the back to keep the indirect ordering. Compared to the unsorted approaches, however, the maintenance of metadata costs significantly more time. As can be seen particularly in the last case with the key position at the back, the overhead is not due to sorting the slots. Instead, it is constituted by the determination of a free bit position plus the given slot position. The indirection approach is thus the only approach that has to find two positions, which is why we have included the overhead of one of them.

In general, the direct sorting of node entries is not suitable for read-write asymmetric devices like PMem. Although the number of changes is similar for indirection, the total number of modified bytes is much smaller, and several slots can be persisted at once. Therefore, apart from the additional determination of the position, indirection can compete with the unsorted variants, which in turn perform all equally well.

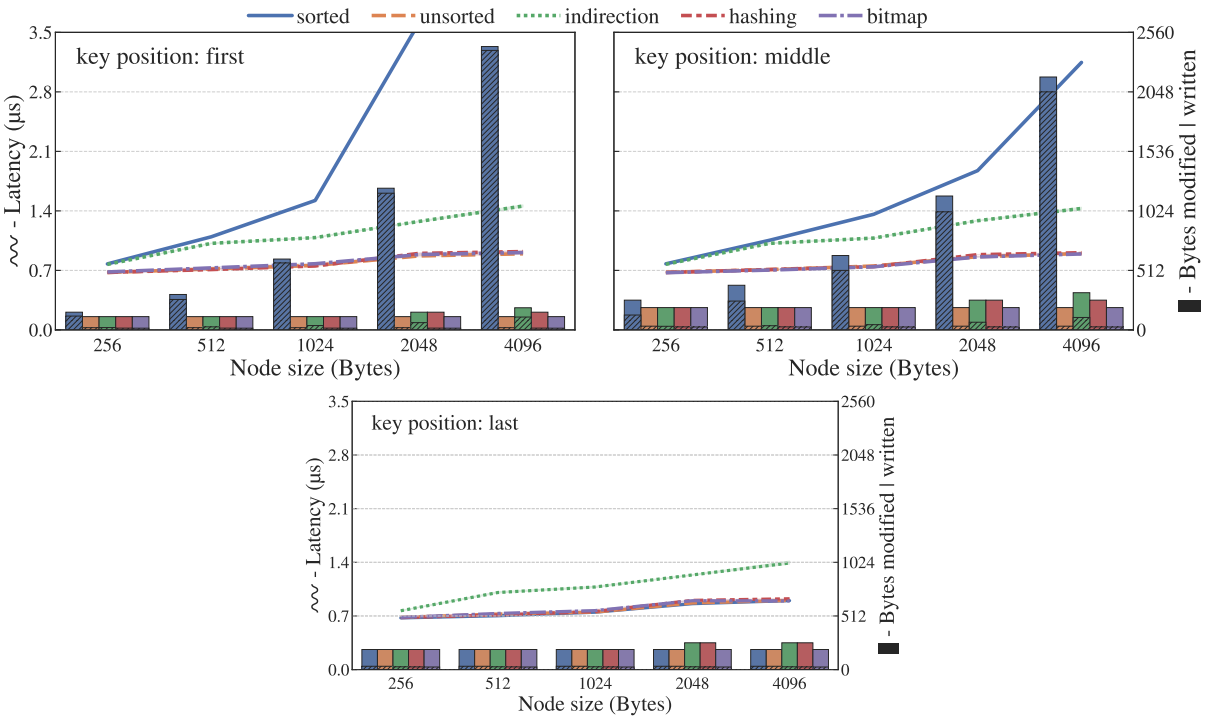



**Figure 3.10:** Inserting a key into a node (E4).

### Node Split (E5)

The next experiment in the insert category is the splitting of nodes. We have focused on data or leaf nodes since these must always be stored persistently to be fail-safe. We chose a similar setup as for experiment E4, except this time, the nodes were filled completely in advance. Therefore, we again report the number of bytes modified and actually written, besides the latency. The two split strategies, as presented in Section 3.3, can be applied to the indirection, hashing, and bitmap approach. The move variant changes fewer bytes, which theoretically leads to a reduction of writes and thus supports DG1. The copy variant, on the other hand, exploits the fine-grained access and thus pursues DG2. If there is no bitmap in the node layout, the copy strategy is not reasonable as it would trigger duplicate writes. The reason for this is that the entire node is copied first, and then all entries are reordered to the left and thus written again. In general, we expect that sorted approaches should be faster since nodes that are already sorted can be split into larger and smaller parts more quickly. Unsorted approaches would have to check each entry against the split key, while with sorted nodes, everything starting from the middle can simply be copied. The measurements are shown in Figure 3.11. Since the bitmap and hashing approach exhibited exactly the same performance, and to keep the figure neat, we summarized them in the diagram as bitmap.

As expected, the approaches with ordered entries are generally faster than the unordered ones. Using an indirection layout and the move strategy results in a similar curve as the sorted variant. It requires, however, a bit more effort for transferring and setting the slots and the bits of the entries. In the case of direct sorting, only the size field has to be adapted at this point. With the bitmap (and hashing) approach, more time is needed by searching for the median in an unsorted array. For this search, we used the *quickselect* algorithm with an average complexity of $\mathcal{O}(n)$, although other selection algorithms could conceivably be used. Accordingly, this procedure is even more dependent on the node size.

Conceptually, the copy strategy modifies more bytes than the move strategy as it copies an entire node. Reflecting on the write endurance and read-write asymmetry of PMem, this could be a shortcoming. However, we performed this copy operation together with the allocation. Since this initiates a sequential write, it seems beneficial for the write-combining buffer on the DCPMMs. As can be seen, the copy strategy performs better than moving. With only the bitmap (or plus hashing slots), the difference is more noticeable since only the bitmap has

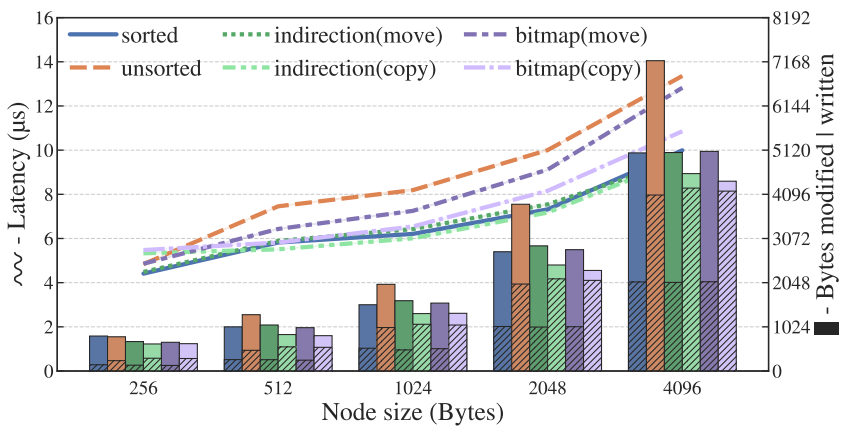



**Figure 3.11:** Splitting a node (E5).

to be updated after the allocation. With indirection, the corresponding slots must be shifted additionally, but it does not require searching for the median. In general, we can deduce that the copy approach is more effective when a bitmap is present. When running the same experiments on DRAM-based nodes (e.g., for inner nodes), we saw a similar result and again recommend either a sorted variant or the copy approach.

By profiling, it has been found that the majority of the time in all approaches is required for allocating the new node (about 80%). As a result, the curves in the figure are all relatively close to each other. Currently, the allocation was implemented with PMDK, which must always be encapsulated in a transaction. In addition, we found that the duration depends on the allocated size. Already in [LHO+19], it was observed that allocations in PMem have a tremendous impact and, therefore, should be used as rarely as possible. In contrast to DRAM, additional mechanisms must be employed to prevent persistent memory leaks. To amortize the overhead, group allocations, as also proposed in [OLN+16], are recommended.

Overall, we would have expected a trade-off of performance against endurance between the copy and move strategy. That is not the case because the copy process is carried out during allocation, and the copy strategy can be recommended without reservation. In the end, however, the sorted variants (direct and indirect) are always better when splitting.

### Move Node (E6)

Another insert-based micro-operation is the movement of DRAM data to a persistent node. It is motivated by LSM-Trees when merging the full DRAM buffer to the first persistent level. In this setup, we varied the node size and also examined the different strategies regarding failure atomicity. These are no failure atomicity, individual failure atomicity, and PMDK transaction as they were explained in Section 3.3. Here, we want to analyze, on the one hand, the effect of the strategies depending on the node size. On the other hand, we want to determine the additional overhead to keep sorted nodes in two variants. The first is to keep the DRAM data unsorted and sort it just before moving it to PMem. Secondly, the DRAM data could be maintained sorted and directly be moved to PMem Thus, the measurements for the first method additionally contains the time for sorting in DRAM. The nodes in this experiment have been simplified a bit and are simple persistent arrays of key-value pairs. In contrast to LSM-Trees, we do not consider duplicates for this operation and use only unique keys. Our results are given in Figure 3.12.

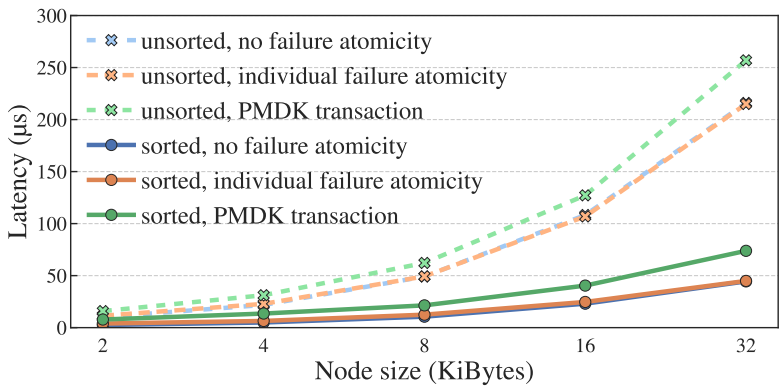



**Figure 3.12:** Move data from DRAM to a PMem node (E6).

Obviously, the additional sorting takes more time.The relative overhead continues to increase as the node size increases. With no failure atomicity measures, the unsorted variant is 4-5× slower than the already sorted variant. Using individual measures and transactions, the overhead is 3-5× and 2-3.5×, respectively. However, the overhead of maintaining a sorted source data structure would be greater. All the more if this is to be done on PMem (see experiment E4). Theoretically, inserting $n$ elements into a sorted array by shifting the entries has an average complexity of $\mathcal{O}(n^2)$. Whereas the one-time sort is possible in $\mathcal{O}(n\ log(n))$. Alternatively, the sorted array can be organized as a tree reducing the complexity also to $\mathcal{O}(n\ log(n))$. In practice, however, larger nodes will then lead to more frequent cache misses and random accesses. For smaller DRAM buffers, we thus see it as sensible to keep them sorted. However, larger DRAM buffers and PMem structures as a whole should be kept unsorted. Since for a typical LSM-Tree application scenario the DRAM buffer is in the order of a few kilobytes, the unsorted variant is the more appropriate approach.

Besides the impact of sorting, the overhead of the PMDK transactions is unmissable in Figure 3.12. These degrade performance in the range of 1.2-3×. Especially with smaller nodes and the presorted approach, this is decisive. On the other hand, we see that the individual realization of failure atomicity – adding a single 64-bit persistent variable into a PMDK transaction or using an 8-byte aligned write (plus manually placed fences) – has almost the same performance as no failure protection. In both cases, the persistent node is still explicitly flushed. Overall, this experiment shows that general-purpose implementations such as PMDK transactions can lead to a significant overhead. Therefore, this should be used exclusively for allocation and deallocations, if possible. Particularly for performance critical applications, failure atomicity should definitely be implemented individually.

#### Merge Level (E7)

Also derived from operations of the LSM-Tree, the next experiment deals with merging multiple sorted nodes into a larger one. We test performance of the 2-way and K-way merge algorithms, as introduced in Figure 3.5a and Figure 3.5b, respectively. On top of this, we considered the two variants of merging, where either the data is merged directly into PMem, or it is first arranged on DRAM and then copied to PMem. In addition, we investigate two extreme cases regarding duplicates. That is, either there are only unique keys across all nodes or 100% duplicate keys, i.e., every key is in all nodes. Again, since the individual implementation of failure atomicity is independent of the primarily tested primitives, its influence can also be well studied here. The results of all these combinations with four source nodes of varying sizes are visualized in Figure 3.13. We summarized no and individual failure atomicity in the same curves because they resulted in about the same performance.

Once again, we can observe that PMDK transactions massively degrade the runtime. Interestingly, the 2-way merging directly to PMem without duplicates is significantly faster than with DRAM buffer. With the K-way strategy, this is not so clear. Using transactions, it is even the other way round. In general, the 2-way strategy with direct merge on PMem performs better than the K-way strategy. The reason for this is probably the emergence of more frequent cache misses and random accesses during the K-way merge since all nodes are considered simultaneously. When duplicates are present (second column of Figure 3.13), the K-way and 2-way merge performance converge. Only when transactions are used the 2-way merge is again faster.

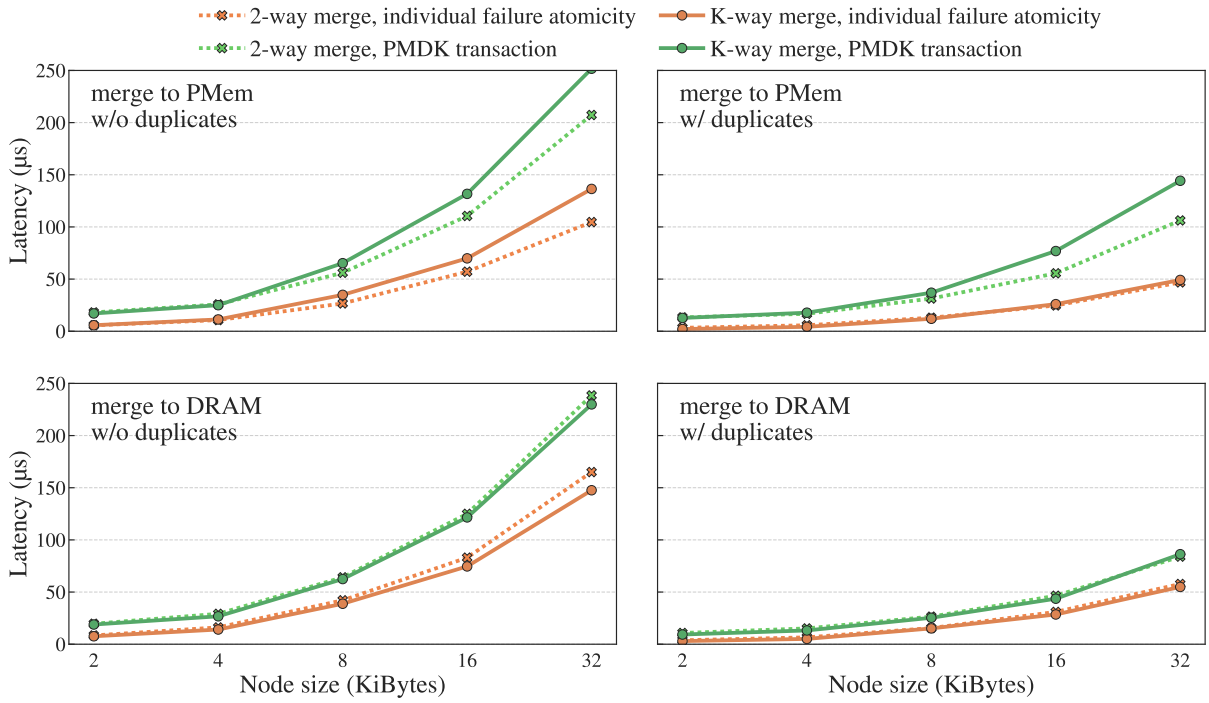



**Figure 3.13:** Merging sorted data in persistent nodes to a new persistent node (E7).

The reason why the merge directly on PMem with transactions behaves this way, we explain as follows. The resulting node after a merge operation may contain a different number of elements varying between two extremes. The least number of elements corresponds to one source node, which is the case with 100% duplicates. If there are no duplicates, the size equals the sum of the elements of all source nodes. Usually, the result size cannot be predicted. For the K-way merge, this means that the maximum size must be assumed and added to the transaction. With the 2-way merge, the size can be reduced by the intermediate results already before, and only the potential maximum of the last binary merge must be added to the transaction. With 100% duplicates, this corresponds only to the number of elements of two nodes. Therefore, the 2-way merge performs better here with PMDK transactions.

With an intermediate buffer on DRAM (the second row in the figure), the K-way merge can take advantage of that since it knows the target size on PMem before allocating and copying the result. Therefore, both merge strategies perform quite similarly. If there are no or only a few duplicates, the K-way merge is even slightly better than the 2-way merge. However, as already stated, the DRAM buffer is not suitable here anyway since it does not reduce the final result size.

So, in summary, we can state the following. In any case, individual failure atomicity should be preferred instead of PMDK transactions. If duplicates are expected to be rare, the 2-way merge with the last merge directly to PMem is the best choice. However, if frequent duplicates are assumed, the approaches show little difference, but the direct merge on PMem uses fewer resources. Lastly, if an individual realization is not possible or PMDK transactions must be used, by all means, these should make use of the DRAM buffer to minimize the size of the logged region.

### 3.4.3 Erase-based Operations

#### Erase from Node (E8)

In the final erase-based category, we start with the basic removal of a single key-value pair from a node. As with the search (E1) and insertion (E4) of a single pair, performance depends on the node size and position of the data, which is why they are varied. In addition to latency, the quantity of bytes written is also of interest. A full node is created in advance so that we can delete entries at any position. Similar to E4, the search for the position is not included in the measurement and can instead be added to it using E1. The complete delete macro-operation excluding underflow handling is composed of traversing the tree, searching each node, and this erase micro-operation. The traditional sorted approach probably performs the worst since the keys and values to the right of the deleted position must be moved to fill the created gap. That, in turn, leads to many writes and flushes. For the unsorted layout without auxiliary structures, we noticed that this is not necessary. In fact, it is sufficient to copy only the rearmost entry to the deleted position and eventually decrement the size field. Since the hashing and indirection approaches are equipped with a bitmap, only one bit has to be reset for them, just like for the pure bitmap approach. Algorithmically, the same process applies, except that the slot array still has to be shifted in case of indirection. Because all of these approaches apply their changes usually in the same cache line, we expect them to run faster than the sorted and unsorted approach. The visualization of our measurements is given in Figure 3.14.

The suitability of the bitmap for fast erase operations is undeniable. There is never a case it is not the best-performing approach. Combined with hashing, not much changes since the algorithm remains the same. Only the slight difference in the node layout seems to have a minimal influence. Even with indirection, the additional effort for shifting the 1-byte slots is

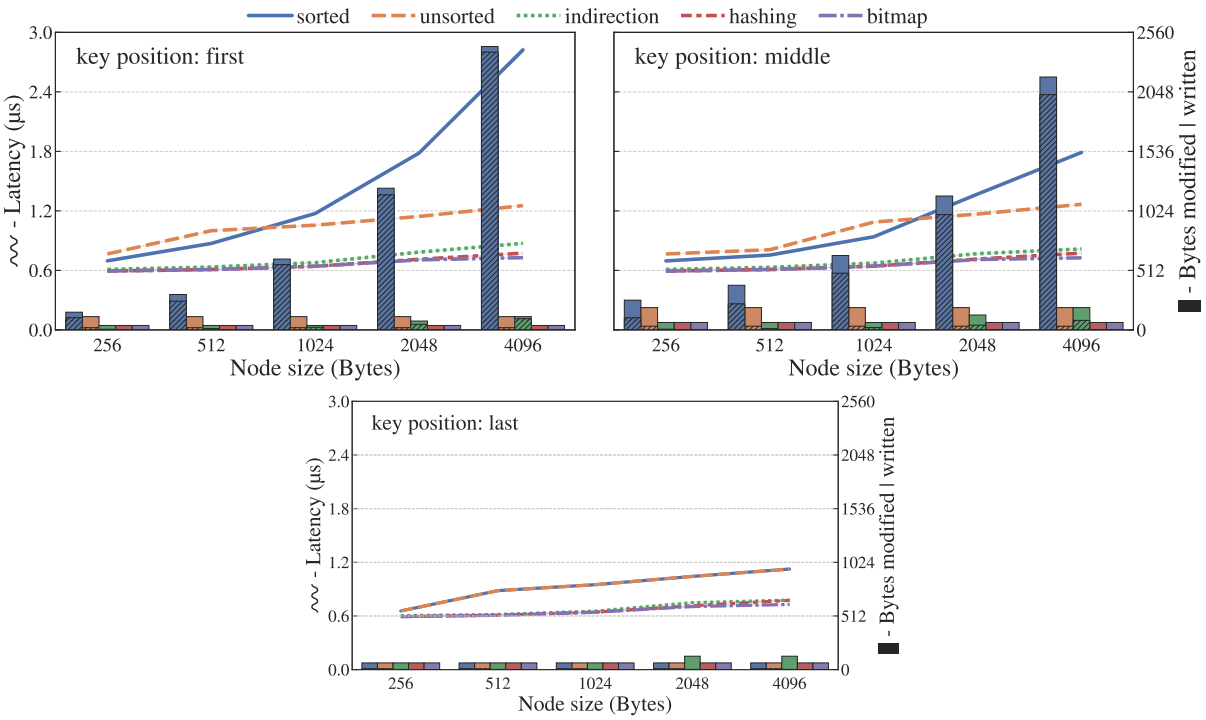



**Figure 3.14:** Erasing an entry from a node (E8).

moderate. Since from 2 KiB, another cache line must be adapted and flushed (more than 64 slots, cf. Table 3.2), the distance to the pure bitmap implementation, which still considers only one cache line, is a little higher there. Only when the target entry is the greatest key of the node, i.e., the last slot, performance stays close to the pure bitmap. That is because no slots have to be shifted and only the bit is reset and the size is updated.

Regarding the remaining two approaches, we would have estimated the alternative approach for the unsorted layout to be more constant as always the same number of bytes are adapted. However, the location of the erased and last position from which the entry is moved is decisive. Furthermore, the size of the node is also crucial since the larger the node, the greater the probability that three cache lines have to be almost always processed. Interestingly, this access pattern is even slower for smaller nodes than the shifting in the sorted approach. In this case, therefore, the original approach should rather be chosen. However, it can be seen that shifting does not scale well with increasing node size and is thus not suitable. The reasons are too many writes and flushes that drastically degrades performance when using PMem. For larger nodes, this approach can only keep up if the last key is deleted and, thus, no shifting is necessary. Overall, we can state that a bitmap is vital for fast erasures.

#### Balance Node (E9)

Erasing entries can lead to the underflow of a node, which can be balanced with a full node. However, there may be other reasons why entries are transferred from one node to another. That is the process we want to address in this experiment. In preparation for this, we create an array with full nodes and an array with half-filled nodes (actually one less than half, representing an underflowed node). The task of the balance micro-operation is to move a quarter of the entries of the full node into the half-filled node. In tree structures, this is usually done using sibling nodes located either on the left or on the right. Practically, this means that the entries are moved either to a node with larger keys or with smaller keys. In the former case, with sorted nodes, the receiving node has to make room in the front for the new smaller entries in advance. On the other hand, when balancing a node having smaller keys than the donating node, the remaining entries of the donor must be shifted to the front. The indirection approach is similar, only that the shifting is applied to the slot array, and the entries can be placed at any free position. With unsorted nodes, however, the whole node must be scanned to find the next minimum or maximum key for every movement, which makes the complexity quadratic. We expect similar results as with the splitting of nodes (E5) since sorted nodes make this procedure more simple. Accordingly, we again vary node sizes and report modified as well as written bytes per operation in addition to the latency. Our results are illustrated in Figure 3.15.

In contrast to our previous results, a contrary pattern emerges so that the number of bytes written is not directly reflected in performance. As expected, the two sorted approaches are consistently the fastest. However, it is unexpected that as the nodes get larger, the performance gap with the unsorted variants becomes so tremendous. In the most extreme case, the directly sorted approach is four times faster than the hashing one. The layouts with bitmap have to additionally search for free space on the receiver side at each move step. Due to the fingerprint array when hashing, extra bytes and possibly even cache lines have to be written. Also, with the indirection, the search for a free slot in the bitmap seems to play a greater role with larger nodes. Since there is less writing, we would have expected it to be better as directly sorted nodes, but the random read and write portion seems more crucial.

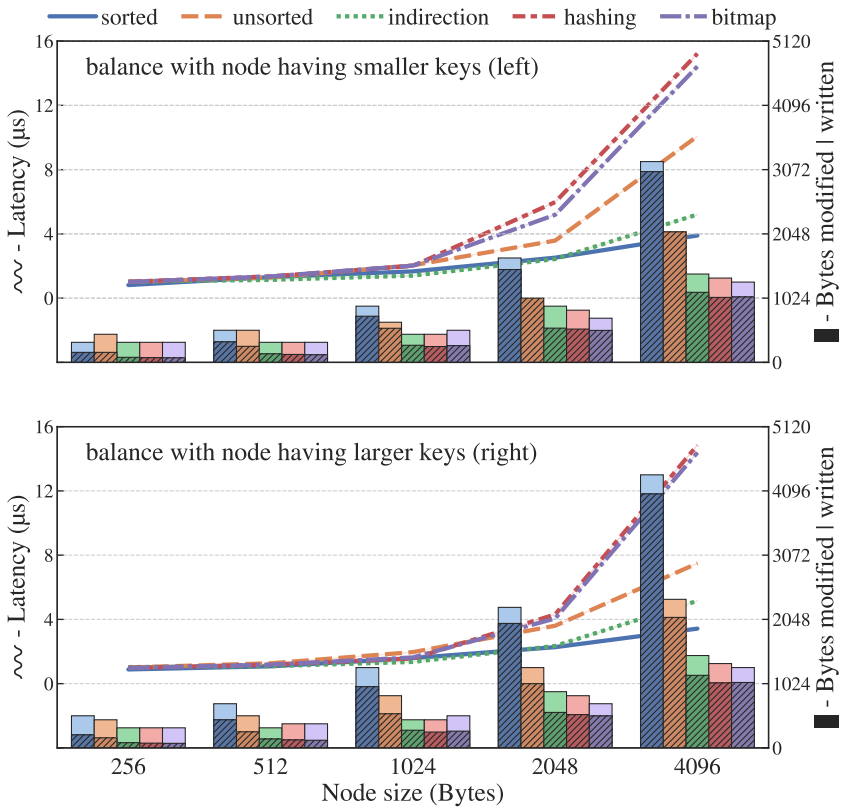



**Figure 3.15:** Balancing two nodes (E9).

If we consider the difference between balancing with the left or right node, we get fewer writes for the former but lower latency for the latter – which is more visible for the plain sorted and unsorted approach. The number of writes can be explained for the sorted case as follows. When balancing with the left node, the entries of the half-filled receiver are first moved to make room, and then a quarter is copied from the donor ($\equiv$ 3/4 of node entries written). However, on the other side, a quarter of the donor is first appended to the receiver node, and then the remaining three-quarters of the donor are shifted ($\equiv$ an entire node written). The second variant involves more writes, but the access pattern is also more sequential and thus slightly faster. Ultimately, with respect to our design goals DG1 and DG2, we would consider indirection the best primitive for this micro-operation as long as the node size does not exceed 2 KiB.

### Merge Nodes (E10)

Besides balancing, underflows in nodes can also be handled by a merge with another node. This merge, unlike experiment E7, affects only two nodes and happens in-place, i.e., no new memory area is allocated. It means that the 2-way and K-way merge strategies are not applicable. Duplicates are also irrelevant for this micro-operation. Again, the node layouts and the corresponding incorporation of the new data are crucial. The merge can be done with a node having smaller or larger keys, as with the balance operation before. However, in this case, merging into a node with larger keys would require an additional shift in the receiving node for the sorted approaches. A merge into a node with smaller keys, on the other hand, does not need to shift anything since the donor node is deallocated or marked free hereafter. Hence, this is always the better option, and we only consider this direction. In general, it can also

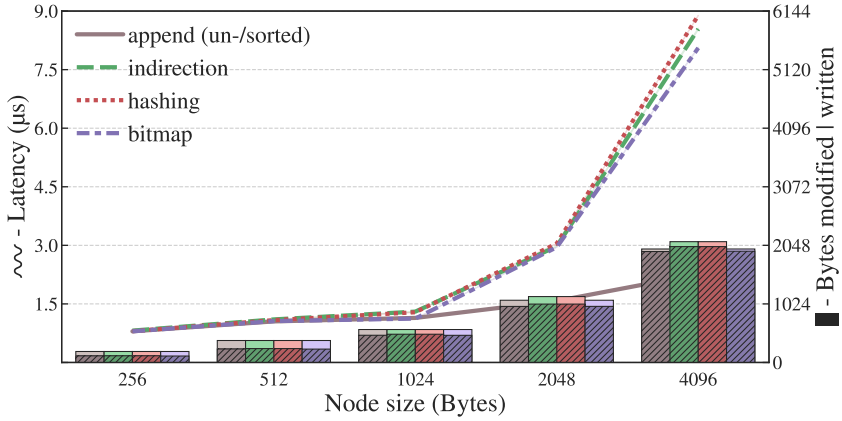



**Figure 3.16:** Merging two nodes (E10).

be seen as an ordered bulk insertion. Algorithmically, the sorted and unsorted approaches proceed exactly the same by appending the entries to the receiver node and finally updating the size field. Accordingly, we have summarized them as *append*. The other primitives require to search for free slots for each new entry, which can be at arbitrary positions and, thus, will lead to a random access pattern. Therefore, we expect them to perform worse than the append procedure. In our measurements, we excluded the time for deallocation of the donor node as it is the same for all approaches and could also be solved by adding the node to a free list or node pool, as indicated before. The measured latencies and modified/written bytes of the approaches are presented as a function of the node size in Figure 3.16.

As with the previous experiment E9, the bitmap has a negative impact. Again, for a node size of 4 KiB, the difference amounts to a factor of four. In the previous section, we already mentioned that it leads to a random access pattern, which becomes even more pronounced as the size of the nodes increases. The iteration of the donor node does not have to check every bit in the case of indirection. However, the indirect access via the slots still causes a random pattern, and the receiver's slot array must also be updated. With the hashing approach, the copying of the fingerprint array is added again and can lead to even more written cache lines. In a separate isolated experiment, we measured the time for plainly deallocating a node. Constantly, this was about $1.6\ \mu s$ independent of the node size. Overall, the result of this experiment is as expected, and the approaches without auxiliary structures are the most suitable for this merge operation.

### 3.4.4 Performance Profiles

Now that we have discussed all the individual experiments in detail, we can assemble our said goal of a PMem-based performance profile per primitive as a comprehensive overview. We have focused on the main layout primitives (sorted, unsorted, indirection, hashing, and bitmap) and their corresponding access primitives as these influence most of the micro-operations. For the access primitives, we have always considered the best-performing of the applicable algorithms based on the experiments. Figure 3.17 presents our result summarizing and contrasting performance and the write reduction of the selected primitives. In the following, we will recapitulate the individual advantages and disadvantages of the approaches.

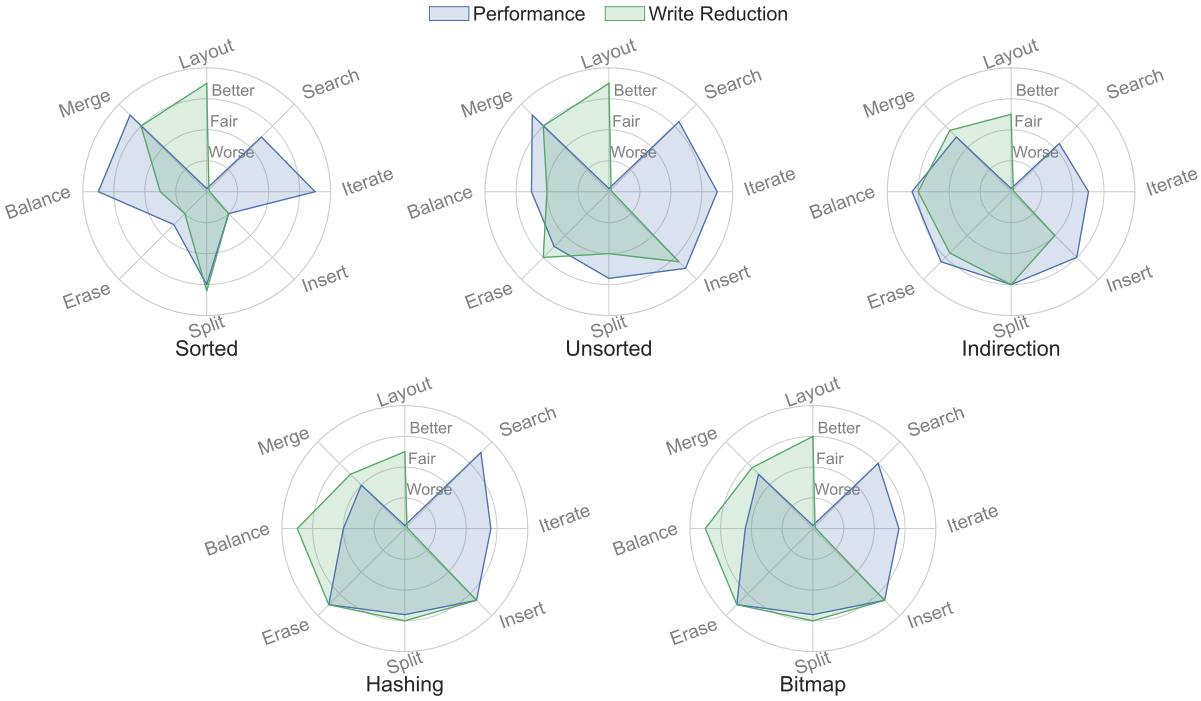



**Figure 3.17:** Performance profile of primary design primitives.

### Sorted

As it becomes evident when looking at the figure, the original sorted approach has significant bottlenecks when placing new or deleting entries in a node. In addition, the search performance is also not optimal in the uncached PMem case. As compensation, iterations and cross-node operations, such as splitting and merging, can benefit from sorting. Furthermore, the layout is the most compact but is paid for with significantly more write and flush operations.

### Unsorted

Due to the omission of node sorting, the basic operations such as search, insert and erase perform significantly better. Particularly for the latter two, it is mainly attributable to the write reduction. If the order is not of interest during the iterations, they also perform equally well as the sorted variant. In return, structural adjustments such as splitting and balancing are subject to significantly higher costs because the order in the entire tree must still be maintained. So if the structure grows and shrinks frequently, these micro-operations will heavily weigh in. Two ways to counteract this would be to opt for larger nodes to minimize the number of these operations or not to handle underflows at all.

### Indirection

Introducing the indirection feature to the unsorted nodes is another method to overcome the problem of costly structural adjustments. Indirect sorting results in a good balance between all micro-operations. In addition, it requires far fewer write operations, which has a positive effect on PMem cell life. Only the node needs a little more memory due to the additional search

**Table 3.3:** Workload assignment for our main design primitives.

| | RUM | Workload | Particularly Positive | Particularly Negative |
|---|---|---|---|---|
| Sorted | ⬍↓↑ | · scan-heavy | + range scans<br>+ lowest memory footprint | - point lookups<br>- insertions & erasures |
| Unsorted | ↑⬍↑ | · read-heavy & read-write | + range scans<br>+ lowest memory footprint | |
| Indirection | ↓↑↓ | · write-heavy | + many structural changes | |
| Hashing | ⬍⬍↓ | · read-write | + point lookups | - costly underflow |
| Bitmap | ⬍⬍⬍ | · read-write | + erase entries<br>+ lower memory footprint | |

structures. However, since searching for a key is slower in this case than for the other layouts, the indirection approach is sooner suitable for write-dominant workloads.

### Hashing

The fingerprint or hashing approach is a little worse in terms of restructuring. In return, the other micro-operations are a bit better than with indirection. Especially for node sizes $\leq 1$ KiB, this design primitive offers the best overall package. As with the unsorted approach, searching, inserting, and erasing a single entry are the key strengths. Only during iteration, due to the unavoidable check of the underlying bitmap, does a deficiency arise. If scans are infrequent and underflow handling is avoided or omitted, hashing is the best choice.

### Bitmap

A similar pattern as with the hashing approach can be seen with the pure bitmap primitive. It is slower only for a few operations, so the combination of fingerprints and bitmap should usually be chosen when considering these two variants. The advantage of the bitmap-only approach is the slightly lower memory footprint and, to some extent, faster underflow operations. However, this minor difference will hardly be noticeable at the macro-level.

To sum up, we assigned the primitives to fitting workloads or application scenarios in Table 3.3. For a quick comparison, we also added read, update, and memory overhead (RUM) indicators as defined by [AKM+16].

## 3.5 General Insights & Design Guidelines

In this section, we want to briefly review the observations of several studies and our experience regarding the properties and challenges introduced by PMem – particularly for DCPMMs. Addressing these, we derive general insights and design guidelines in due consideration of the above experiments for implementing new data structures on PMem, as it was similarly presented in [JBGS21]. These should enable developers to avoid common pitfalls when designing novel efficient data structures or even systems tuned to a modern hardware landscape.

### 3.5.1 Challenges & Characteristics

**C1** In the first place, PMem has a 3× higher latency and also a 3× lower bandwidth than DRAM for a random read pattern. The degradation factor for the write bandwidth is even about seven [vRVL+19, YKH+20].

**C2** Reading and writing operations on PMem have an asymmetrical behavior towards each other. This is manifested in three characteristics, where writes are slower than reads, cost more energy, and lead to cell wear.

**C3** To reduce write amplification, the DCPMMs internally work on 256-byte blocks. An underlying write-combining buffer summarizes four cache lines to one block write.

**C4** The largest failure-atomic store instruction comprises only 8 bytes, which must be aligned on an 8-byte boundary. Larger atomic changes have to be realized via software, either individually or via general-purpose libraries.

**C5** Allocations on PMem are significantly slower than on DRAM, which stems from the fact that cache lines have to be flushed and other additional recovery measures have to be taken. In [LHO+19], it was shown that PMem allocations could take up to 8× more time.

**C6** The dereferencing of persistent pointers may not be optimized by compilers. Since this concept is not part of the standard (yet), its usage is not automatically optimized, as it is with volatile pointers [Sca20].

### 3.5.2 Insights

Besides the statements and inferences about the primitives, our results gave us also some sustainable general insights when designing data structures, choosing suitable access primitives, and combining several ideas. Our findings are partly consistent with those in [LHO+19] and, thus, corroborate them.

**I1** *Extendability and Combinations.* We have already mentioned in Section 3.3.5 and seen when examining the design space that there are still numerous unconsidered primitives and combinations thereof. Thus, for example, in the hashing approach, we have always used a bitmap for foreseeable advantages during erasures. But it would be just as conceivable to replace it with a size field. Furthermore, there are probably other search variations like interpolation or exponential search. A more detailed investigation of known techniques such as compression or zone maps to reduce write and read accesses also seems beneficial. A more explicit recommendation for tree-like structures is to use 1 KiB data nodes supported by fingerprints and a bitmap, while 4 KiB inner nodes should be placed in DRAM using a traditionally sorted layout. If inner nodes should also be persistent (e.g., to keep recovery short or save DRAM), indirection is advisable since it massively reduces the number of writes. In addition, it should be noted that hashing is not exploitable for inner nodes since no exact keys but ranges are searched. The use of indirection for inner nodes and hashing for data nodes basically represents the combination of the ideas from [CJ15] and [OLN+16].

**I2** *Save Writes.* Not intrinsically a new insight, but derived from the original design goal DG1, algorithmically saving writes is an important aspect when designing PMem-based data structures. It reduces not only cell wear and energy consumption but also fundamentally improves the performance of the algorithms (challenge C1 & C2). This correlation between the number of writes and performance was observed particularly in experiments E4 and E8.

**I3** *Hybrid Data Structures.* With the help of our first two experiments E1 and E2, it became clear that in total, there is still a big difference between PMem and DRAM performance (challenge C1). In particular, the traversal experiment E2 has shown that dereferencing and pointer chasing have an even greater impact on PMem if they are not cached (challenge C6). Therefore, we highly recommend adopting a hybrid approach of DRAM and PMem when facing high performance and simultaneous persistence requirements. As indicated in insight I1 and as it became clear from our DRAM and cached measurements, the sorted approaches (direct or indirect) are best suited for inner nodes.

**I4** *Lightweight Failure Atomicity.* Although general-purpose solutions for failure atomicity, such as PMDK transactions, simplify the implementation, they are usually not recommended for performance-critical applications (challenge C4). Particularly in experiments E6 and E7, we observed that the underlying logging and snapshotting of transactions lead to a noticeable overhead compared to individual solutions. Especially the classic copy-on-write conversions should generally be avoided and replaced by in-place approaches. Even if failure atomicity is one of the most intricate aspects of programming with PMem, an individual solution adapted to the problem using lightweight atomic writes is almost always preferable.

**I5** *Strive for Sequential Patterns.* Although PMem allows fine-grained random access, unlike disks, it has been shown that sequential access is still superior (challenge C1). Alternating or jumping between non-consecutive cache lines proved to be particularly expensive. It was evident, for example, for in-/direct binary search in experiment E1, iterating while checking the bitmap or indirection array in experiment E3, and also when erasing and moving entries in unsorted nodes in experiment E8. Especially the latter was kind of unexpected for us but showed the importance of physical proximity.

**I6** *Avoid Frequent De-/Allocations.* Since allocations on PMem are much more expensive than on DRAM and depend on their size as well – at least with PMDK – they should be handled with care (challenge C5). That was particularly well reflected in experiments E5 (allocation) and E10 (deallocation). To mitigate costs, it is possible, for example, to perform group allocations at idle times and reuse nodes instead of frequently deallocating and allocating.

**I7** *Small Node Sizes.* The block or node size on PMem should always be a multiple of 256 bytes, and in our cases, optimally between 256 bytes and 1 KiB. The lower bound of 256 bytes, as well as the multiple of it, is justified by the write-combining buffer of the DCPMMs, which work on this granularity (challenge C3). Also, read operations seem to benefit when using a multiple of this block size [vRVL+19, YKH+20]. The upper bound was established by the experiments, where often for larger nodes than 1 KiB, performance deteriorated drastically. The ordinary 4 KiB for DRAM are thus not optimal here. On the one hand, this could be due to the larger physical distances between the accesses, which means that prefetching does not take effect properly. On the other hand, it could be caused by the search structures at the front, which then occupy more than one cache line. However, smaller nodes also

lead to longer navigational paths. In the vertical case, i.e., traversing, we refer back to I3. For the horizontal iteration, however, small node sizes are a real disadvantage.

## 3.6 Summary

In this chapter, we explored typically used data structures within DBMSs and how to adapt them for PMem. We saw that the design space is massive and, thus, focused on tree-like data structures such as $B^+$-Trees, Skip-Lists, Tries, and LSM-Trees. Furthermore, we identified several data structure design primitives and micro-operations and extensively evaluated them on real PMem hardware. With the help of our white-box conceived evaluation, we could derive performance profiles for the main primitives and compile a set of general insights for designing PMem-based data structures. Essentially, we could already address three of the four data management challenges that we identified in Section 2.3. These are the failure atomicity, property utilization, and data placement.

When looking at the primitives, it has become apparent that there is ultimately no single best solution. Instead, every design decision depends on the target application. Therefore, our analysis and evaluation serve as guidance in finding optimal PMem-specific design parameters and primitives, as well as combinations thereof, for a given use case. For example, in write-heavy workloads with many structural changes as well as for range scans, indirection arrays can provide a tremendous speedup. The fingerprint or hashing approach, on the other hand, is best suited for many point queries and also for inserts and deletes when omitting underflow operations. Especially for frequent deletes, however, the approaches should be supported by a bitmap. Finally, when mainly running iterations or leaner memory usage is required, the sorted and unsorted layouts without additional structures are the best choice. Overall, Table 3.1 and our investigations already include a fair number of primitives and micro-operations. At the same time, it still offers a lot of potential for further extension.

We will use the collated insights and guidelines from our low-level examination for the following chapters and the challenges they entail. More precisely, we can apply and extend the experience to analytical structures in Chapter 4. In addition, the data structures developed in this chapter serve as possible candidates for maintaining persistent states in our envisaged transactional stream processing model in Chapter 5.

# 4 Persistent Analytical Storage Layouts

The data and index structures presented in the previous chapter are intended for fast key lookups and OLTP-like updates. A feature they all lack is to efficiently query tuples or records on other attributes than the key. Employing a full-grown Online Analytical Processing (OLAP) or data warehouse system would cost a lot of installation and administration effort as well as additional storage. Furthermore, the development of such a system optimized for PMem would be beyond the scope of this thesis. However, a wide range of analytical applications would highly benefit from a PMem-enabled storage layout for a table.

Considering the global goal of this work to conceive a TSP system, the question could arise why these analytical layouts are relevant. In fact, the intention is to provide these within a TSP system to represent a persistent state on which both continuous as well as ad-hoc queries can be executed simultaneously. In this case, the key used for indexing is most likely the timestamp of the tuple. That is not very convenient for analytical workloads, which often use other attributes than the key. Therefore, in this chapter, we discuss two layouts to tackle this issue, namely, a clustered table design and a multi-dimensional index structure.

## 4.1 Related Work

We divide related work into two areas. The first subsection focuses on engines using PMem to improve recovery and build times while still providing efficient analytical query performance. The second part considers hybrid DRAM-PMem approaches that follow a similar path as our multi-dimensional index structure.

### 4.1.1 PMem-based Engines targeting Analytical Workloads

A predominant portion of work involving PMem focuses on OLTP workloads and corresponding structures, which we already described in Section 3.1. Analytically targeted engines have so far

mainly been found in the graph domain. For example, Sage [DMK+20] is an approach to parallel graph analysis on PMem. They store the entire graph as a read-only copy in PMem while smaller mutable parts are held volatile. One of these parts is an auxiliary structure that tracks deleted edges for faster graph filtering. In particular, with the proposed parallel semi-asymmetric model, the authors address the asymmetry properties of PMem and provide fundamental algorithms to efficiently solve various graph problems. Similarly, Gill et al. [GDH+20] investigated graph analytics using PMem. However, they used it in Memory mode and, thus, its serves only as a DRAM extension. Nevertheless, they were able to show that their NUMA-enabled algorithms can outperform more expensive DRAM-only cluster organizations on cheaper single-machine setups with DCPMMs. Another representative from the graph domain is Poseidon[1] [JBGS21]. However, the authors aim at an architecture for HTAP and not just OLAP, with the current focus still being transactional processing. Due to the underlying Multi-Version Concurrency Control (MVCC) approach, older snapshots can potentially be archived to PMem or disk for time travel analysis. Equally, a storage engine that is more oriented towards HTAP is SOFORT [OBL+14]. This engine implements column-oriented tables that keep the primary part read-optimized in PMem, similar to Sage. That accelerates analytical workloads. A write-optimized delta storage structure, on the other hand, is used to efficiently handle OLTP queries. It is periodically merged into the main store. As with Poseidon, an engine for mixed workloads is envisaged, but only OLTP and recovery aspects are evaluated. This lack of approaches for analytical purposes motivates this chapter and our two proposals all the more.

### 4.1.2 Selective Persistence

We have already described many of the following approaches in Section 3.1. Here, however, we look at them again from the perspective of a hybrid DRAM-PMem division. The naïve approach towards data structures on PMem is to move everything to the persistent medium. However, so as not to diminish performance excessively compared to DRAM, only essential parts should be kept durable, while recoverable portions should be maintained volatile. The volatile data can then be rebuilt during recovery. The best-known representative is probably the FPTree [OLN+16] – and correspondingly the derivative optimized for DCPMMs called LB+-Tree [LCW20] – which keeps its leaf nodes in PMem as a linked list, while the inner nodes reside in DRAM. Thus, only the last access when traversing the tree is more expensive than a pure in-memory solution while at the same time significantly less DRAM is used. With a persistent hash index and a volatile B+-Tree, HiKV [XJXS17] offers another hybrid memory design. Simple searches involving only a key-value pair (e.g., put, get, update, and delete) can be quickly handled by the hash index. More complex operations such as scanning, which require ordering of the data, make use of the B+-Tree. It is useful because order preservation often leads to sorting, splitting, or merging of nodes resulting in many writes that are too expensive on PMem. A volatile B+-Tree is also used by the DPTree [ZSC+19] that is backed by an append-only log on PMem and serves as a buffer. This buffer is merged into the base tree as soon as it reaches a specified size. The base tree is implemented similar to the FPTree with volatile inner nodes (radix tree) and a persistent linked list of leaf nodes. Another hybrid approach is to use a general-purpose multi-tier buffer management that covers DRAM, PMem, and disk. Examples of this are [vRLK+18] and [APM19]. This kind of approach is very similar our dynamic caching design (see Section 4.3.3).

[1] Poseidon: `https://github.com/dbis-ilm/poseidon_core`

## 4.2 Clustering Approach

Our first approach for an analytical storage layout makes use of clustering[2]. The goal is to find a suitable layout with efficient data placement for analytical workloads in the presence of PMem. For this purpose, we have opted for a table design based on multi-dimensional clustering employing a block-oriented structure. The focus is on significantly speeding up range queries without essentially slowing down point queries.

### 4.2.1 Bitwise Dimensional Co-Clustering

As a clustering approach, we used Bitwise Dimensional Co-Clustering (BDCC) [BBS16] that we briefly describe in the following. BDCC is a processing and storage framework based on multi-dimensional clustering. It covers data structures, data access, and processing techniques that have proven highly beneficial for analytical workloads. Although the original work focused on disk access, BDCC works with fine-grained access to millions of small groups. With PMem's byte-addressability, these groups could be fetched even more efficiently. Simultaneously, the consideration of multiple dimensions empowers rapid query processing beyond the key attribute(s). The approach was designed for column stores and, thus, decoupled indexing from clustering. Hence, it can be moved to byte-addressable memory like DRAM or PMem by calculating data offsets rather than relying on additional indirections.

In summary, BDCC stores similar tuples close to each other using an artificial clustering key referred to as BDCC key value. The similarity is determined by the specified clustering dimensions and their weighting. Such a key value is then composed of a bit interleaving binary representation of the chosen dimensions. The weighting of a dimension results from the number of assigned bits and their positions in the clustering key. The interleaving of the BDCC clustering key can utilize different strategies such as round-robin or major-minor, but also any other interleaving. An optimal choice highly depends on the dataset and the workload. In order to make the specified dimensions efficiently queryable, they must be sorted according to the clustering key. The original study proved that for that purpose, multiple lookups on dimension attributes could be executed as a set of bit operations on the clustering key. It results in very efficient selection pushdowns and fast data reordering based on the various dimension sort orders. An example of the generation of a four-digit wide BDCC clustering key is given in Table 4.1. It is based on two columns, which each is assigned two bits interleaved in a round-robin fashion. After calculating a binary representation of the columns and composing the BDCC key from the two major bits (in this case, all bits), the table can be sorted and partitioned based on this key.

In the original context, the term co-clustered refers to the coherent multi-dimensional storage of tuples across relations based on their foreign keys. Since we consider mostly independent tables without foreign key relationships in the TSP model, this part is of no importance for our purposes. We have chosen BDCC due to its proven flexibility and efficiency. Nevertheless, any other clustering approach could have been chosen for our design.

[2]The material in this section is based on [GBS18].

**Table 4.1:** Example BDCC key calculation from two dimensional columns.

| Original Column Order | | Binary and BDCC key calculation | | | Sorted Columns | | |
|---|---|---|---|---|---|---|---|
| $column_1$ | $column_2$ | $bin_1$ | $bin_2$ | bdcc key | $column_1$ | $column_2$ | bdcc key |
| 400 | B | 11 | 01 | 1011 | 200 | B | 0011 |
| 300 | C | 10 | 10 | 1100 | 100 | C | 0100 |
| 100 | C | 00 | 10 | 0100 | 100 | D | 0101 |
| 300 | D | 10 | 11 | 1101 | 400 | A | 1010 |
| 400 | A | 11 | 00 | 1010 | 400 | B | 1011 |
| 400 | D | 11 | 11 | 1111 | 300 | C | 1100 |
| 200 | B | 01 | 01 | 0011 | 300 | D | 1101 |
| 100 | D | 00 | 11 | 0101 | 400 | D | 1111 |

### 4.2.2 Analytical Table Structure

As the focus is on analytical workloads, reads are more dominant than writes. In particular, we aim for fast range scans and random point queries. Since our table represents not just a key-value store, it should also be possible to query multiple attributes apart from the key.

Looking at the PMem characteristics and our expected access patterns, initially, an appropriate data structure is required. A typical table layout in database systems uses a combination of a persistent table stored on disk, and a volatile part kept in memory. We strive for a better performance than these systems while reducing writes to PMem whenever possible to disguise the read-write asymmetry and avoid premature cell wear. Hence, the data structure should be optimized and utilized for reading operations. The approach covers a three-layer memory hierarchy to provide a high query performance and the possibility to swap cold data in case of a PMem shortage. That is essential both due to the higher latencies compared to DRAM and the lower capacities compared to flash. Before looking at the complete layout, we start with a description of the two main aspects of our approach, namely clustering and block design.

#### Clustering

Since OLAP queries often demand other attributes besides the key for lookups, a technique is required to answer them without executing a full table scan. A variant would be to create secondary indexes per attribute. However, this would lead either to a higher write amplification on PMem or recovery effort when maintaining them on DRAM. Our alternative is to use clustering, which groups similar tuples within blocks. As a clustering approach, we opted for BDCC that allows multi-dimensional accesses and accelerates analytical workloads. As mentioned above, there are also other clustering methods, but BDCC has proven itself quite effective [BBS16] and is the most flexible approach. The underlying bitwise operations are utterly suited for PMem and can massively accelerate queries. By storing similar values close to each other, compression becomes more promising. The clustered blocks are sorted by their BDCC key value ranges and form a linked list (i.e., chunked vector). However, the block's respective contents are heap organized. That results in fewer writes for re-sorting and swapes it with potentially more PMem read accesses during range scans.

### Block Structure

To allow a simultaneous application of PMem and disk for persistence, we have decided on a block-like structure on which both technologies can operate. It also results in better data locality, which, in turn, is beneficial for exploiting caches and prefetching mechanisms. Since the bytes or cache lines of a block in PMem can still be accessed individually, there are no restrictions or negative effects. Similar to *Data Blocks* [LMF+16], we use a PAX-like structure and extend it with Small Materialized Aggregates (SMAs) [Moe98]. The layout of the block structure is shown in Figure 4.1.

The header starts with three fixed-size fields, namely the clustering key (BDCC key value) range, the number of containing tuples, and the free space of this block. The BDCC key values and the tuple count are four bytes each. For the free space, two bytes are sufficient. After the fixed part of the header, the offsets to the aggregates of the attributes and the actual data values are stored. These are calculated based on the number and type of attributes of the tuple. The main part then consists of mini pages for each column. Three basic types are supported: integer, double, and string values. While integer and double values (8 bytes each) are handled equally, the string type must be organized differently due to its variable length property. For integer and double attributes, the SMAs are copies of the minimum and maximum values, whereas, for string attributes, these are offsets to the actual values. The total block size is currently set to 32 KiB as this was proven to be the minimum size before reading from solid-state drives becomes inefficient [BdNSS10]. Apart from that, this corresponds to today's typical L1 cache sizes, which could lead to more cache hits when using the same block several times. We also examine different block sizes experimentally further below. Possible optimizations for future work could be the application of compression and outsourcing of SMAs in DRAM. Both would reduce the PMem storage overhead and possibly accelerate query performance.

### Storage Layout

The complete three-layer storage layout based on Data Blocks [LMF+16] and BDCC [BBS16] is sketched in Figure 4.2. There are three top-level components, namely, the table metadata, an index structure, and the actual table data that is distributed among PMem and disk. The metadata serves, on the one hand, as a root object into the persistent area to find all data in case

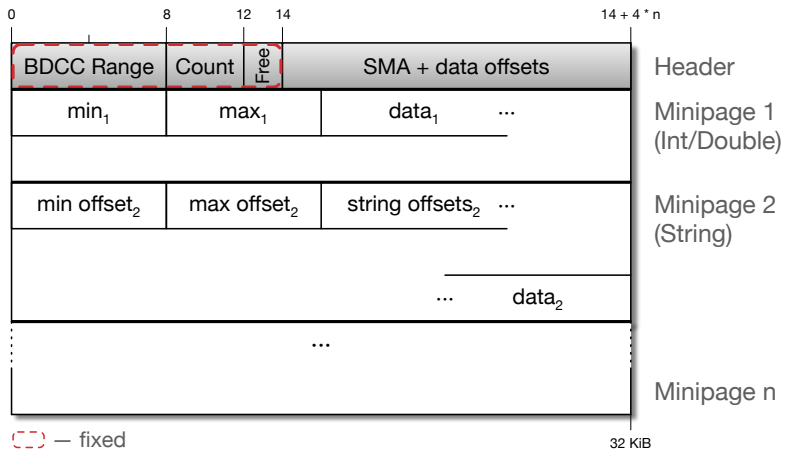



**Figure 4.1:** Single clustered block structure.

of a restart. It holds a persistent pointer to both the index structure as well as the first data node. On the other hand, the schema and clustering information is stored here that is determined by the client on table creation. The default index is currently a basic PMem-persistent $B^+$-Tree version, but it is not limited to a specific structure. Alternatively, a complete in-memory variant could be used, but this must be restored in case of failure. Furthermore, hybrid variants would also be conceivable, such as the FPTree. The key of an index entry complies with the table's primary key. However, the value is not directly the searched tuple but a separate class called *PTuple*. It consists of a persistent pointer to the data node and the offsets of all attributes in the corresponding clustered block. That allows more precise queries (and updates) of selected attributes to touch fewer data areas. The data nodes themselves consist, in addition to the actual clustered data, of a vector of deleted tuple offsets and optionally a histogram. As inserts can trigger splits, the deleted vector marks the positions of tuples which can be skipped during the split procedure or possibly reused for an insertion. Another option would be a bit vector. However, since deletes are rarely expected in analytical workloads, the first variant seems more efficient. Similarly, the histogram is currently only necessary for splits. Depending on the data distribution, the use of the mean on the BDCC range as the pivot key does not necessarily lead to a favorable split. The histogram allows more options such as the median as a split key or the creation of a separate node for frequent BDCC key values.

Persistent pointers are dereferenced once to avoid too much pointer chasing, and then virtual references are used. It yields a higher optimization potential for the compiler and the CPU. For the time being, a persistent pointer is PMem exclusive, and a transcending variant for both disk and PMem will be part of future work. A practical solution would be to use a boolean field associated with a union structure that, in turn, can represent a PMem pointer or disk file offset. Additionally, only the 32 KiB clustered blocks could be moved to disk while the secondary data and links of the data nodes are always kept within PMem. The distinction between hot and cold blocks is another point of discussion. Besides the location (disk or PMem), the question of structural differences would also be interesting, such as the general layout or compression techniques. In the current implementation, however, all structures are stored on PMem. In combination with atomic transactions, the recovery effort is thus almost zero.

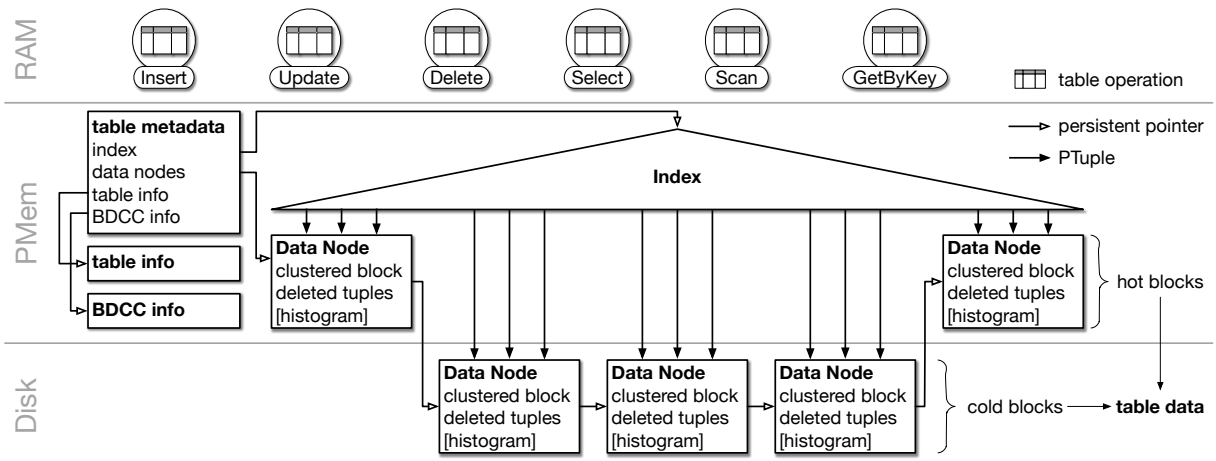



**Figure 4.2:** Three-layer storage layout of a table for analytical workloads.

### 4.2.3 Operations and Optimizations

Besides the basic data manipulation operations – i.e., insert, delete, and update – the table structure supports the primarily targeted analytical operations such as range scans, point queries on the key, and other selections with arbitrary predicates. All operations always store their intermediate results and variables in DRAMs since query recovery was not intended here. The insert operation first calculates the BDCC value of the new tuple and puts it into the appropriate block. Since the entries within a block are unsorted, this process is additionally accelerated. Subsequently, a PTuple is created and added to the index. Due to PMem, the data can be directly persisted and read later without detouring via memory to disk and vice versa. Once a block is full, it must be split. It can be done based on the BDCC range distribution using either a histogram, the middle of the minimum and maximum, or the average. Since this process can be very costly due to reading, comparing, and copying, it is also conceivable to use bulk loading, which avoids splits. In order to delete a tuple again, its position is queried via index, marked as deleted in the corresponding data node, and finally removed from the index. An update is currently implemented as a sequence of delete and insert.

For the typical search and scan requests based on a key, the index is used. If all data are desired, additionally, a conversion from a PTuple to a proper tuple must be conducted. But also individual attributes can be materialized. The goal of our clustering approach is to provide fast and robust range scans even for non-key attributes. To achieve that, the number of required data block accesses must be limited. We introduce a separate block iterator, which first collects all candidates and prunes blocks that do not qualify. Subsequently, only the blocks in the candidate list are scanned. The pruning is accomplished by comparing the BDCC range in the block headers with the selection predicate applying the same BDCC mask to the attributes. If the columns used within the predicate are not part of the clustering, the SMAs can be used instead. An additional block index, such as a binary search tree or a sparse index, could accelerate this process further.

### 4.2.4 Evaluation

With the following microbenchmarks, we want to demonstrate the potential of our clustering layout compared to other PMem-based solutions and the classical DRAM + disk approach. Our procedure is briefly labeled *PTable* in the figures. If an index is used, its corresponding type will be part of the label (e.g., *PTable+PBPTree*). Since this is an analytical approach, we focus exclusively on reading performance in the form of point and range queries. We execute these on the key as well as on non-key attributes, where we expect in particular for the latter a better runtime of our approach than the competitors. These competitors include our own implemented $B^+$-Tree versions based on PMem, namely the PBPTree and FPTree. Any of these trees can be used as an index within our table implementation. Apart from these PMem-based competitors, we have further added an upper and lower baseline. Due to its wide usage in the research community as a persistent key-value store, RocksDB [Fac20] was added as a disk-based reference. It is relevant to point out that RocksDB writes to disk (SSD) for persistence and simultaneously keeps a transient version of the data in memory. As a simple improvement, we also present a variant where RocksDB is writing durably on PMem. Again, the label indicates which version is used. However, it still employs unsuitable disk-based instructions like *msync*, which means that the improvement will probably be limited. Finally,

we also included a proprietary in-memory $B^+$-Tree as a volatile reference implementation. We presume the performance of the PMem-based approaches to be located between these two baselines.

Since we strive for structured data, the values are always tuples of four elements (<*int*, *int*, *string*, *double*>) and an integer key (same as the first attribute). An essential part that influences the performance is the serialization, deserialization, or dereference overhead. It affects both RocksDB and our approach since tuples are not directly stored as such. While RocksDB packs these into so-called *Slides*, the tuples in our case are distributed across mini pages. In both cases, the internal representation of the result must be converted after a query. Because of this impact, we examine the performance both with and without this kind of materialization. Based on our insights of the previous chapter, the node size of the PMem-based index structures is set to 1 KiB. For the transient version, 4 KiB has revealed the best performance. That is most likely due to the OS being optimized for a 4 KiB page size. The FPTree has a leaf size of 1 KiB and a branch size of 4 KiB accordingly.

### Point Queries

We start with the point lookups based on the key. Figure 4.3 shows the performance curve with increasing table size. As can be seen, RocksDB is far behind, although the measures here do not include materialization. Furthermore, it is evident that only for larger tables a difference between SSD and PMem placement becomes apparent. That is most likely since cache hits become increasingly rare and the actual non-volatile devices need to be accessed more often. Since the identical $B^+$-Tree versions are used within our layout, a similar performance is achieved. The difference is that the BDCC approach uses PTuples, whereas the isolated tree uses C++ tuples. The two lines shown for the PTable contain the materialization. Without materialization, the results would be identical to the standalone trees. For one million tuples, point queries on the PTable with volatile $B^+$-Tree and materialization would be just as fast as the pure PBPTree. Overall, the larger the table, the more the curves of the persistent trees and the PTable converge. For the same reason, as already mentioned for RocksDB, PMem is accessed more frequently, which makes the slightly higher complexity of the PTable less prominent. The standalone transient $B^+$-Tree (80-400 ns) outperforms all other solutions. However, it obviously does not provide the desired persistence and, when used for a TSP system, transactional guarantees. As expected, our approach is between both baselines with a clear tendency towards the lower limit. That is already a partial success because we did not want to make queries based on the key much slower than an isolated persistent index structure.

### Range Scans

Next, we consider range scans with typical key-based scans as well as scans on non-key attributes. In particular, the latter is of essential interest. As mentioned before, our later use case shall exploit this structure to represent persistent states in a transactional stream processing system. The key is highly likely the timestamp and, thus, usually not the target of the range predicate. For the following experiments, the data structures were initially filled with one million tuples, and subsequently, the scans were performed varying the selection percentage. For scans based on non-key attributes, we used the first (integer) and fourth (double) fields for the range predicates. The ranges were chosen to overlap by 50 percent and result in the same selection ratio as the key-based scans.

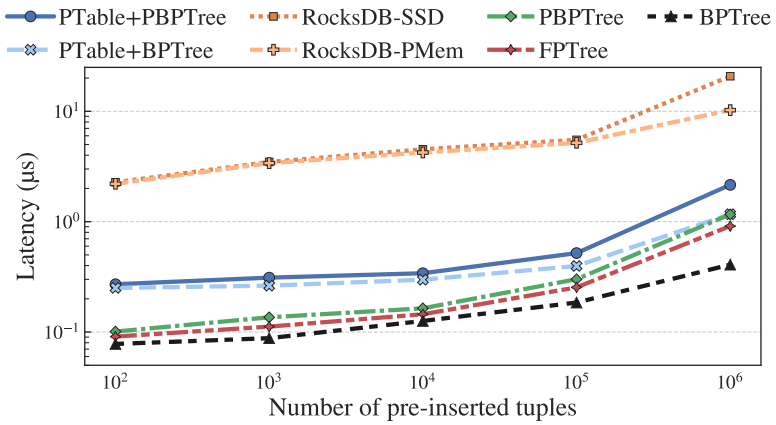
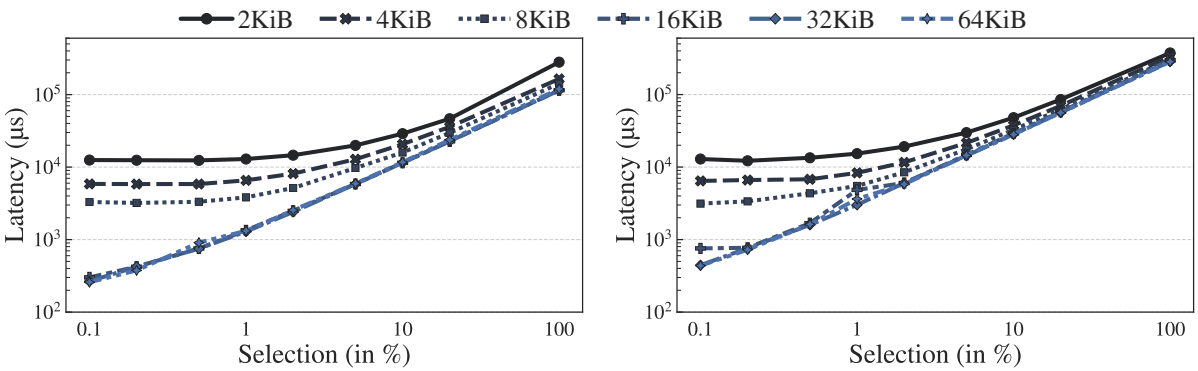



**Figure 4.3:** Point queries on clustering approach.




**(a)** Without materialization.

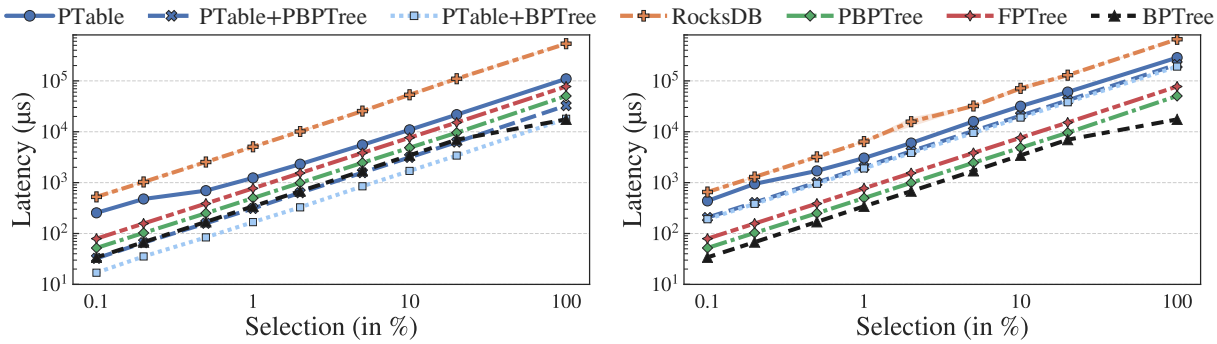
**(b)** Including materialization.

**Figure 4.4:** Range scan varying block sizes using non-key attributes.

The block size, as already stated, was chosen due to efficient SSD I/O operations and the L1 cache size with 32 KiB. In the following, we will investigate whether a similar block size is sensible in connection with PMem. Thus, in a preliminary experiment, the block size was varied while running non-key range scans via the block iterator. Figure 4.4 shows the best-achieved execution times of this query, both with and without materialization. Interestingly, the performance difference is particularly visible for a low selection percentage. That is due to the relatively more frequent chasing of pointers for smaller block sizes. For broader ranges, the performance converges. Including materialization, the curves approach each other even more. However, a 16 KiB, 32 KiB, and 64 KiB block size achieve similar performance. Overall, we thus conclude that a 32 KiB block size is a good choice for this workload on both SSDs and PMem.

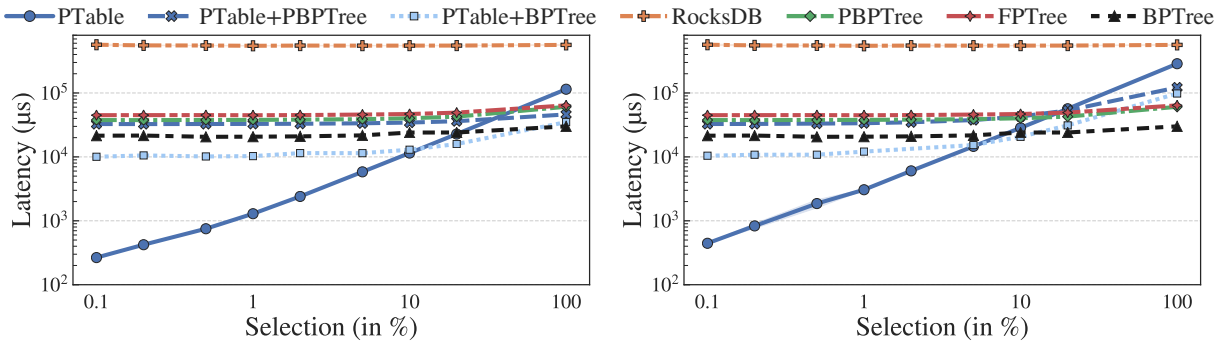
Now, we compare the performance with the other structures and competitors. Our approach can fall back on two types of iterators for range scans. The first is the block iterator that uses an antecedent pruning mechanism as described in the previous section. The second approach is the iterator of the underlying index. We reflect the used variant in the label. Our results of the key-based range scans without and including materialization (for PTable variants and RocksDB) are visualized in Figures 4.5a and 4.5b, respectively.

Although our focus was on queries with non-key attributes, the BDCC based table implementation combined with an index performs similar or even better than the competitor systems when excluding materialization. Despite the fact that the PTuple, in this case, has the same




**(a)** Without materialization. **(b)** Including materialization.

**Figure 4.5:** Key-based range scan on a table of one million tuples.




**(a)** Without materialization. **(b)** Including materialization.

**Figure 4.6:** Range scan using non-key attributes on a table of one million tuples.

memory footprint as the original tuple, it seems to be more efficient to process. However, as soon as these have to be transformed, it clouds the picture a little. Figures 4.6a and 4.6b show the results for non-key attributes.

With non-key attributes for RocksDB and the $B^+$-Trees, all the scan queries degenerate to full-table scans. Using the block iterator in our approach can outperform even the non-volatile $B^+$-Tree and, thus, provides persistence and best performance simultaneously. However, the execution time increases linearly with the selection percentage as fewer blocks can be pruned from the scan. The index iterator gets faster, starting at around 10-20%, still achieving the same performance range as the transient and persistent $B^+$-Tree. For small range windows, the index iterators of the PTable are even faster with materialization than their stand-alone counterparts. Overall, the choice of iterator highly depends on the table size and selectivity. For larger volumes (>1M tuples) and a low selection amount, the block iterator is most likely to be preferred over the index utilization. It is expected that with larger table size, the intersection point will move further to the right. That is because the index structures take longer for a complete scan, regardless of the selection. Consequently, the system would benefit from a cost model which autonomously decides on the appropriate iterator. All in all, it has been shown that our approach can keep up with and sometimes surpass the other compared structures in all aspects considered.

## 4.3 Multi-dimensional Index Approach

The second layout targeting analytical workloads is a multi-dimensional index approach[3]. The particular challenge of exploiting PMem for multi-dimensional index structures is their inability to reconstruct inner tree nodes from the leaf nodes, as it is done, e.g., for $B^+$-Trees. That is because the dimensional data is only stored in the inner nodes to avoid massive amounts of stored data. Therefore, we need another method that universally leverages PMem for multi-dimensional index structures. Accordingly, in this section, we present a selective caching approach that buffers index nodes statically or dynamically in DRAM. We use the Elf data structure [BKSS17] as a representative for multi-dimensional index structures. It employs an explicit main-memory optimized layout, making it ideal for byte-addressable PMem.

### 4.3.1 The Elf Data Structure

Since we have chosen Elf as a multi-dimensional index structure to be the investigation target, we start by introducing its original layout and operation. Elf clusters the column values by their prefix. Because of the DRAM-optimized storage layout, it can efficiently answer analytical queries. Furthermore, all column values are part of the structure and, thus, can be used as a standalone structure or as a supporting index. Below, we explain the fundamental design principles of Elf, followed by the extension to enable parallel search.

#### Design Principles and Optimizations

Clustering column values by prefix conceptually makes Elf a prefix tree similar to ART [LKN13]. However, ART does not work with column values but with characters or digits of the key domain. Each level in the Elf tree representation contains all values of a specific column. The nodes in the tree itself keep their entries sorted, thus achieving a total order and allowing pruning. In Elf, two types of nodes are distinguished, called *DimensionLists*, which are inner nodes with sorted column values, and *MonoLists*, which represent a sort of leaf nodes. While *DimensionLists* contain values of the same column concerning various tuples, *MonoLists* comprise values of a single tuple over multiple columns. Based on the example table in Figure 4.7, which is composed of four columns and seven tuples, Figure 4.8 shows the conceptual Elf in tree representation. The *DimensionLists* in this example are labeled (D1)-(D5), whereas (M1)-(M7) are *MonoLists*. The differentiation of the node types serves the optimization of data access and memory consumption. Therefore, the idea of the *MonoLists* is that if there is no more branching at the end of the tree, the linked single-valued *DimensionLists* are combined into a single *MonoList*. It reduces the number of pointers and random accesses. In this sense, Elf starts as a kind of column store and gradually converges to a row-oriented layout when traversing down the tree. As shown in [BKSS19], this design can efficiently compress the dataset at hand.

To further reduce random access patterns, the conceptual tree is linearized into a flat array, as it is shown in Figure 4.9. It is especially beneficial for read-intensive analytical workloads. The *DimensionLists* and *MonoLists* are stored in the contiguous array in the same order as a depth-first search through the tree. Furthermore, child pointers are converted to offsets within

[3]The material in this section is based on [JGBS20, JGBS21].

| | $C_1$ | $C_2$ | $C_3$ | $C_4$ |
|---|---|---|---|---|
| $T_1$ | 1 | 2.5 | 0 | 1 |
| $T_2$ | 3 | 3 | 2 | 1 |
| $T_3$ | 1 | 2.5 | 1 | 0 |
| $T_4$ | 3 | 3 | 0 | 3 |
| $T_5$ | 2 | 0.5 | 2 | 2 |
| $T_6$ | 3 | 3 | 1 | 2 |
| $T_7$ | 1 | 3 | 1 | 3 |

**Figure 4.7:** Example table.

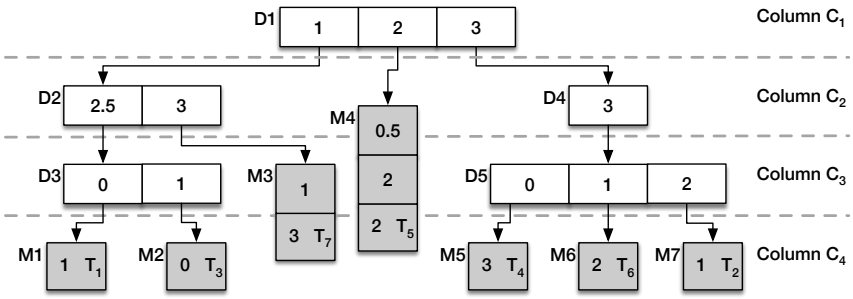



**Figure 4.8:** Conceptual Elf based on example table.

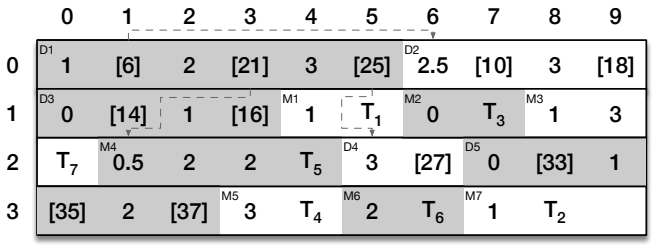



**Figure 4.9:** OLAP and DRAM optimized array layout based on example table.

the array. In [BKSS19], it has already been shown that this approach can accelerate filter queries by a factor of 10. Furthermore, this was also similarly realized already for B-Trees [RR00].

### Parallel Search Algorithms

To make range queries more efficient, among other things, Elf also offers the possibility of searching the structure in parallel with multiple threads. The division of the search space can be done in various manners. Blockhaus [Blo19] has proven that a too fine-granular separation – such as one DimensionLists per thread – results in too much synchronization overhead. Splitting threads by subtrees, on the other hand, has emerged as the best strategy. More precisely, it means that each thread is assigned an entry from the first *DimensionList* and processes the corresponding subtree. If there are more entries than available threads in the first *DimensionList*, those threads that are already finished will be assigned to another remaining entry until all entries are covered. Using our example in Figure 4.8 and assuming two threads, the first thread would be designated for the first entry 1 and process the subtree D2-D3-M1-M2-M3. The second thread would get assigned the second entry 2 and would only evaluate M4. The first thread to finish – most likely the second one – would process the last entry 3 and its subtree D4-D5-M5-M6-M7. In [Blo19], this approach has also been shown to work most efficiently for imbalanced subtrees.

## 4.3.2 Persistent Memory Adaptions

After the basic structure and procedure of the Elf have been described, we will now discuss possible adaptations for PMem. Before we present our idea of selective caching, we start with two baseline implementations. Both consider a naïve translation of the Elf from DRAM to PMem. The first approach is to store the Elf exclusively on PMem providing full persistence

and simple data management, albeit with performance losses compared to the DRAM variant. Quantifying this degradation in performance is the objective of our first experiment in the evaluation section. The second naïve approach – which we refer to as hybrid Elf – maintains both a PMem and DRAM copy. It gives us both DRAM performance and persistence but also costs us double the memory consumption.

### Pure PMem-based Elf

For the translation of Elf to PMem, we used the features and libraries of PMDK, as shown in Figure 4.10. The Elf is stored as a data object located in a file on PMem. For that, we used the pool class template and its basic methods create, open, and close. When the pool is opened, the offset of the Elf object is obtained by following the root and the succeeding persistent object pointer. Subsequently, we use only the virtual pointer of the current application instance to access the Elf object. Besides some member variables, the linearized array occupies the major part, which is stored separately and accessible via the Elf object. The outsourcing of the array took place because of its variable, unpredictable size. Analogous to the entire Elf object, the persistent pointer for the array is dereferenced only once at the beginning, and then the virtual address is always used directly. It eliminates expensive dereferencing on every access and, at the same time, halves the used pointer sizes. In the figure, the virtual pointer is part of the persistent structure, but this is not necessary and only serves visualization purposes. To ensure atomicity during the creation of the Elf, we used PMDK transactions for the persistent memory allocations. Furthermore, we wrapped the member variables, such as the sizes and the number of dimensions, with the persistent property class. Since the focus is on analytical read-only workloads, it is superfluous but paves the way for consistent in-place updates and inserts. The array itself was not maintained as a persistent property since otherwise, updates would cause the entire array to be copied to the undo log. Therefore, when making changes, it is more efficient to manually add the respective ranges to the transaction.

### Hybrid Elf

As mentioned initially, a hybrid structure, where primary data is held in PMem and reconstructable secondary data is kept in DRAM (selective persistence), is desirable but not possible with Elf. Because of its layout, persisting only the *MonoLists* or TIDs would render recovery impossible as the prefixes are irreversibly lost. Thus, the only viable option of a hybrid approach is to create a volatile copy in DRAM after the initial building in PMem or at the restart. All

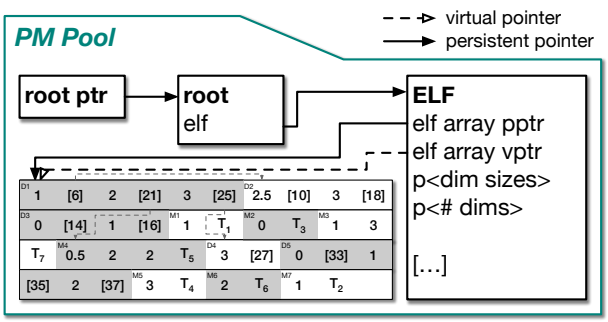



**Figure 4.10:** Organization of persistent Elf in PMem pool.

queries are then answered by the volatile copy leading to a query performance at DRAM speed. At the same time, a persistent copy is preserved, and the structure does not need to be recreated from the original data (if any) in the case of failure. Alternatively, the persistent version is still available to execute query workloads if the available DRAM capacity for the copy is insufficient.

The evident disadvantage of the hybrid Elf is the memory overhead since it maintains two copies now. In addition, it also has a performance overhead since extra time is necessary for copying it to DRAM after building. However, this only happens once, and then only the copying mechanism has to be repeated. We estimate the actual cost of copying to be comparatively low but will evaluate this below. For the native application purpose of Elf, namely data warehousing, periodic inserts would occur, whose modifications must always be propagated to the DRAM copy, too.

### 4.3.3 Selective Caching

Having presented our two more naïve approaches, we now come to our more complex idea based on the pure persistent Elf to build an additional cache in DRAM that is significantly smaller than a complete copy (as in the hybrid Elf). This cache should only hold the crucial parts that are frequently or most likely traversed by the queries. The question now is which parts of the Elf these are and how to find and cache them. In general, a cache entry in our approach corresponds to a *DimensionList*. Since *MonoLists* only store parts of a single tuple, we excluded them from caching. For selective caching, we devised two strategies for building up the DRAM cache:

- **Dynamic Caching:** The first strategy is to cache the *DimensionLists* dynamically while traversing the Elf. In a naïve way, all visited nodes can first be stored in a DRAM-resident hash table. Since this will sooner or later become quite large and inefficient, it should be supplemented with an eviction/replacement strategy. We discuss below what eviction policies are conceivable. However, it must be considered that CPU caches also already use a dynamic strategy. Whether an additional DRAM layer with a similar procedure can thus provide significant improvements at all remains to be investigated.

- **Static Caching:** Alternatively, only predetermined (static) parts of the Elf could be kept in DRAM. For instance, these could be the first x dimension levels that need to be copied at build or recovery time. It would virtually ensemble an FPTree-like hybrid layout [OLN+16] that maintains inner nodes in DRAM and leaf nodes in PMem. However, it is also conceivable to statically cache entire subtrees or paths if it is known in advance that these will be queried very frequently. The static strategy would prevent the situation that the cache has to be probed first and, in the negative case, the persistent part also has to be accessed. That means fewer branch predictions and, thus, mispredictions are necessary. Instead, it is always known which part is in DRAM and which part is in PMem.

Besides these two isolated strategies, it is also possible to combine both of them. Thus, the upper levels could be cached statically and the lower dimensions dynamically, resulting in a sort of split cache.

### Replacement Strategies for Dynamic Caching

Principally, all classical eviction policies, such as Least Recently Used (LRU), Least Frequently Used (LFU), and First In, First Out (FIFO), are conceivable for the dynamic strategy. Briefly recapped, LRU evicts the last element in a list ordered by access time, which is also updated when parts are retrieved from the cache. With LFU, each element is associated with an access counter, and the one with the smallest number gets evicted. FIFO is the most modest approach as elements are evicted in the same order as they came in without positional updates. In the context of PMem, Lersch et al. [LOLS17] already investigated the application of a dynamic cache with an eviction policy for LSM-Trees. They achieved some performance gains for read-only workloads using the LRU and the more sophisticated 2Q policy. Especially for our analytical intent, this appears promising. 2Q divides the cache into two separate queues, which, in turn, can have different eviction policies like LRU. The first queue contains the actual data elements and is called AM. The second, on the other hand, stores only the IDs of elements – offsets, in the case of Elf – referred to as A1. Once an element is requested that is not in any of the queues, only the ID is initially cached in A1, and the data is still loaded from PMem. If it is already present in A1, the data is copied from PMem to AM, and the access is made via the DRAM cache. If the element is in AM, it is accessed directly, which corresponds to a cache hit.

Apart from the existing policies, we have also developed our own candidate based on LFU. In fact, our idea is to populate the cache using access probabilities calculated from the proportion of a *DimensionList* and its subtree to the total tree. More precisely, given the total set of tuples as $T = \{t_1, ..., t_n\}$ and the set of reachable tuples by *DimensionList* $D_i$ as $T_{D_i} = \{t_{i_1}, ..., t_{i_k}\} \subseteq T$, the assigned probability for a *DimensionList* is $P_{D_i} = \frac{k}{n}$. The sum of the probabilities of the same level is one if there is no *MonoList*. Accordingly, the first *DimensionList* (the root) always has a probability of one. In contrast, each *MonoList* has a probability of the inverse of the number of all tuples ($\frac{1}{n}$) since it is only a suffix of a single tuple. For the remaining nodes, the probability results in accordance with the prefix redundancy. Considering the example in Figure 4.8 with a total of seven tuples again, D2 covers three tuples (T1, T3, T7) and, thus, has a probability of $\frac{3}{7}$. Analogously, D1, D3, D4, and D5 yield probabilities of one, $\frac{2}{7}$, $\frac{3}{7}$, and $\frac{3}{7}$, respectively. The *MonoLists* M1-M7 have a probability of $\frac{1}{7}$. An important observation is that *DimensionLists* at a lower level do not necessarily have a lower probability than the nodes of the previous level (cf. D2/D4 and D5). After we calculated all probabilities, the next step is to copy designated *DimensionLists* into the DRAM cache. The selection happens depending on the intended cache size and cut-off probabilities. The copy operation can be either static on build and reboots or dynamically as part of the eviction policy. We refer to the latter as Least Likely to be Accessed (LLA). Essentially, it works in the same way as LFU, except that fixed probabilities are used instead of the frequency values. Furthermore, *DimensionLists* are only evicted if the lowest probability is smaller than the new one to be inserted. It is not the case for the original LFU, where an entry is always evicted, and the access must go through the cache. Another possibility would be to use the probabilities only as a starting value and increase them with each request. However, this would mean additional effort and would eventually converge to LFU.

All in all, both of our caching strategies (i.e., dynamic and static) have their ups and downs, which we will examine in more detail in the following evaluation. However, it should be noted at the outset that selective caching causes the query execution to often switch between DRAM and PMem while both devices use the same CPU caches. Especially for the dynamic variant,

there is thus a higher probability for cache misses. The static strategy, on top of that, costs extra time when building and restoring.

### 4.3.4 Evaluation

The subject of the evaluation will be all three persistent Elf variants (pure, hybrid, and cached) explained above. Since we target analytic workloads, the focus will be on read-only queries, in addition, to build and recovery time. More specifically, these are exact-match, range, and partial-match queries which we will describe further below. For the pure persistent Elf, we will first quantify the performance overhead to the DRAM counterpart. Then we look at our optimizations, starting with the hybrid Elf and followed by the caching techniques to reduce this overhead. Overall, it will be seen that with suitable fine-tuning, a DRAM-like performance is achievable.

#### Experimental Setup

For the experiments, we prepared two datasets. The first test dataset consists of 100M tuples having ten dimensions, whose values are uniformly distributed using various ranges. The dimensions are of integer type[4] with a range of 100 as default. That resulted in about 4 GiB. In addition to our uniform dataset, we considered another dataset with more realistic data and properties, such as the correlation of dimensions. Namely, this is the TPC-H *LineItem* table with 15 dimensions. Using a scale factor of ten, this resulted in about 3.5 GiB. Since the Elf is designed to make efficient use of shared prefixes, the ordering of dimensions is essential. Therefore, in preparation, we sorted the columns of the table in ascending order of cardinality to take advantage of prefix redundancy and improve caching benefits.

The queries use a *Zipfian* distribution [GSE+94] with skewness parameter $\theta = 0.5$ to set the query parameters. It simulates a more realistic access pattern than a uniform random number generator. All experiments were executed for at least ten iterations and, by default, in a single-threaded environment to produce robust results. Exclusively our final investigation will cover multi-threaded executions. The three query types used are described below. Their throughput is reported in queries per second (qps).

*Exact-Match Query:* For exact-match queries, all dimension values are checked for equality with the query parameters, and the TID of the matching tuple is returned in the positive case. In order not to measure the computation times of the *Zipfian* distribution, we extracted the dimension values of the tuples according to this distribution before each run and, thus, only had to execute the queries successively during measurement.

*Range Query:* For this type of query, a lower and upper bound is specified for each dimension x, and it returns a list of TIDs of the tuples whose dimension values are included in all x ranges. As with the exact-match queries, we precomputed the query parameters. For range queries, the extracted dimension values were used as lower boundaries, and the upper ones were set according to the given range size (to be specified in due course).

[4] Larger or even variable data types would also be possible but bring additional parameters into play due to the typical use of dictionaries, making the measurements harder to interpret. Furthermore, they would increase the used cache size in DRAM and put more pressure on CPU caches. Hence, the DRAM cache will most likely be more beneficial as an additional layer between CPU caches and PMem.

*Partial-Match Query:* The last query type is a more extensive variant of range queries. In partial-match queries, upper and lower limits are not given for each dimension but only for preselected ones. Dimensions that are not specified are wildcarded, and the entire cardinality of these dimensions is traversed accordingly. From a technical point of view, the lower limit is set to the minimum and the upper limit to the maximum of the dimension's data type. The creation of the queries is the same as for the range queries except that we additionally pass a boolean array indicating which dimensions are wildcarded.

### PMem versus DRAM

As a first experiment, we juxtapose the pure PMem solution with the DRAM-based Elf. Figure 4.11 shows our measurements based on the uniform dataset. The runtimes presented are means of 1M, 10K, and 1K executed queries for the query types exact-match, range, and partial-match, respectively. For the range and partial-match queries, we plotted a range size of 2% and 100% per set dimension to illustrate the influence of selectivity. In general, we expected that DRAM would outperform PMem. It is interesting to note, however, that the initial building of the Elf entails an overhead of just 18%. We estimated that this is because the build process is writing the array to PMem sequentially. The write-combining buffer of the DCPMMs seems to be quite efficient if there is only a single sequentially writing thread. Similar results can be observed for range and partial-match queries with 100% selectivity, where the overhead is also only 70% and 66%, respectively. Again, the structure is traversed sequentially across long ranges (depth-first search), only this time in a read-only manner. The wide ranges also lead to more commonly passed *DimensionLists*, which will result in more hits in the CPU cache for both the volatile and the persistent Elf. However, this is different for the exact-match queries and the two range-based queries with a low value for selectivity. These queries only have a very tiny query window and, thus, often produce a random access pattern. That results in an overhead of 223%, 210%, and 236%, respectively, which corresponds roughly to the difference between the PMem and DRAM random read latency. Since the resulting random access patterns offer a higher potential for improvement by caching, we will mainly focus on small query windows in the following experiments.

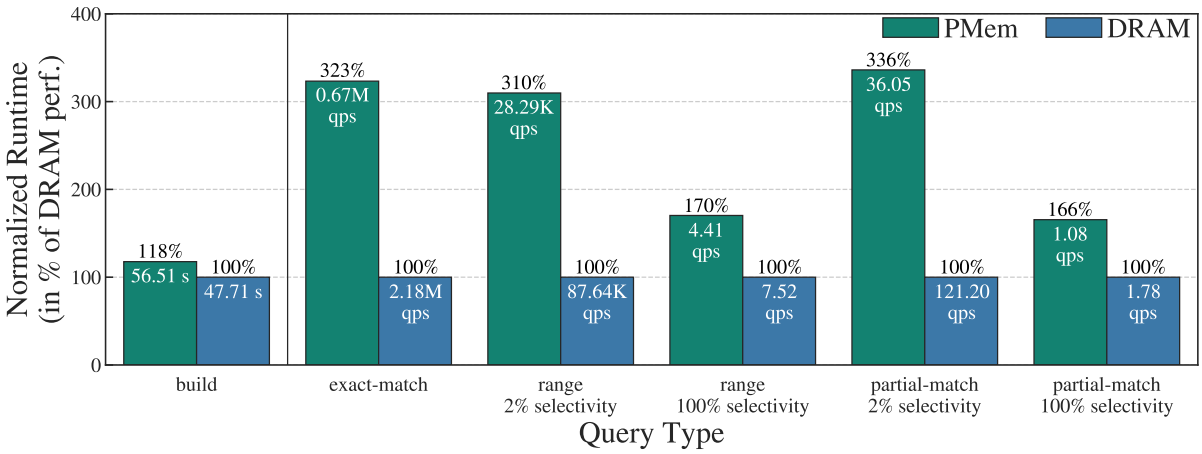



**Figure 4.11:** Build and query performance of Elf.

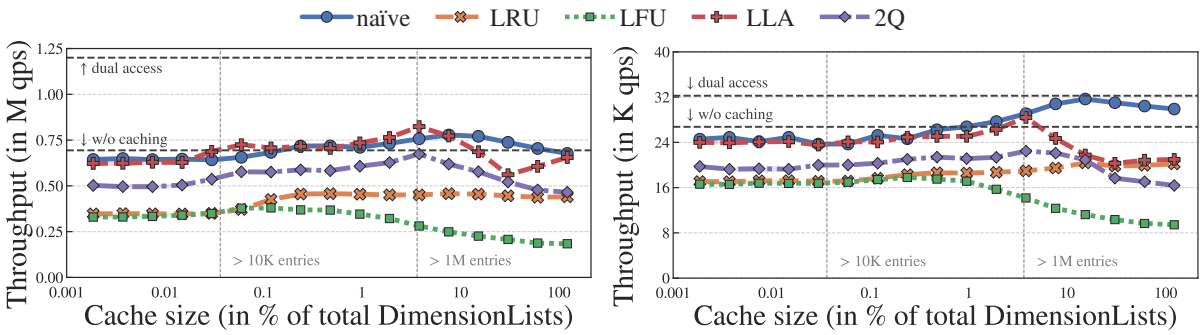
### Hybrid Elf

After considering the two pure variants, we will next look at their combination in the form of the hybrid Elf. Since the query performance will be the same as for DRAM, we evaluate only the build and recovery times here. What is added in comparison to the pure PMem Elf is the allocation and copying of the array into the main memory. For our 100M tuples from the uniform dataset (∼4 GiB), this process took 1770 ms. This amount is in addition to the building time and results in 58.28 s. Thus, this process takes only 3% more time in total, while at the same time causing the recovery in DRAM to shrink from 47.71 s to 1.77 s (i.e., about 27× faster). Hence, if enough DRAM and PMem are available, this is the best performing solution across all tasks (except for the negligible copying overhead).

### Dynamic Caching - Eviction Policies

If the last condition, namely enough DRAM, is not given – as is often the case for analytical tasks – our caching approaches offer an alternative, which we will evaluate in the following. Initially, we examine the efficacy of the eviction policies for dynamic caching dependent on the query type. We implemented the naïve, LRU, LFU, LLA, and 2Q strategies as described above. Furthermore, we have included two baselines. The first is the pure persistent Elf without DRAM caching (labeled as *w/o caching*), which serves as a lower limit that must be reached as a minimum in order to achieve performance benefits. The second baseline is more theoretical to indicate the possible upper limit for our caching approach (labeled as *dual access*). It makes use of the hybrid version and retrieves all *DimensionLists* from the DRAM copy and the *MonoLists* from the PMem copy. Hence, this simulates the case where all *DimensionLists* are cached but exclusive of probing if they are. Our results on the uniform dataset are presented in Figure 4.12a for exact-match queries and Figure 4.12b for range queries. Since the results for partial match queries behave similarly to the range queries, we have omitted them. An influential parameter that we have varied along the x-axis is the cache size.

Prior to the measurements, we warmed up the caches for 100M and 100K queries, each for exact-match and range queries, respectively. The measurements themselves are then reported as the average of 1M and 1K subsequent queries. As can be seen for exact-match queries, the best setting with dynamic caching is LLA, with a capacity of about 1M entries. Concerning the total number of *DimensionLists*, this capacity corresponds to 3.8%. For range queries, the best setting is the naïve approach and a capacity of about 4M entries. It is equivalent to 15.2% of




**(a)** Exact-match queries.

**(b)** Range queries.

**Figure 4.12:** Throughput of dynamic caching variants on the uniform dataset.

the total *DimensionLists*. We also ran the experiments on the TPC-H dataset and saw similar results for range queries. Therefore, for this dataset, we also opted for the naïve approach with a capacity of 4M entries. The exact-match queries also reached their peak using the LLA policy, however, with only 64K entries.

Overall, we can state that our initial concerns – that an additional dynamic DRAM cache may not give salient advantages above dynamic CPU caches – have turned out to be true. With only two of the tested eviction policies, it was possible to outperform the pure PMem variant and even with those only for a few cache sizes. During profiling, we were able to confirm that the number of instructions is about the same for both the caching and pure variants. However, another concern was proven true in this investigation, namely that dynamic caching leads to more LLC misses, which is mainly responsible for the inferior performance. In the warmed-up state, three hotspots were found to cause this performance degradation and cache misses.

**HS1** Lookups in the hash table[5] (∼25% performance impact)
The hash table constitute the central part of the cache structure. The bottleneck becomes particularly noticeable starting at around 100K entries.

**HS2** Accesses to the *DimensionLists* (∼25% performance impact)
These are both in PMem and DRAM.

**HS3** Accesses to the *MonoLists* (∼50% performance impact)
They are always located on PMem.

Hotspot HS2 and HS3 arise because the DRAM and PMem parts evict each other from the CPU caches, especially when *DimensionLists* are continually copied from PMem to DRAM. In the pure persistent variant, PMem practically occupies the CPU caches alone, and *DimensionLists* cannot be cached twice. However, not much more is optimizable for these two hotspots since these are essential accesses that work algorithmically the same as without caching. Therefore, only hotspot HS1 remains as a possible tuning option. The reason for the bottleneck is the following conflict of the used hash table. While increasing the cache size provides better chances for a DRAM cache hit, it decreases the odds for a CPU cache hit and, thus, the general lookup performance. A countermeasure we have applied is the partitioning of the cache into multiple smaller hash tables (Figure 4.12 is already based on this optimization). For larger cache sizes, this already led to a significant performance boost. Nevertheless, even this could not consistently outperform the non-caching variant and probably needs further tweaking. The dual access baseline plotted shows theoretically what performance would be possible if HS1 is eliminated. HS2 and HS3, i.e., the competition of DRAM and PMem for free slots in the CPU cache, are still present. While the exact-match case still offers some room for improvement, we seem unable to achieve much more for range queries.

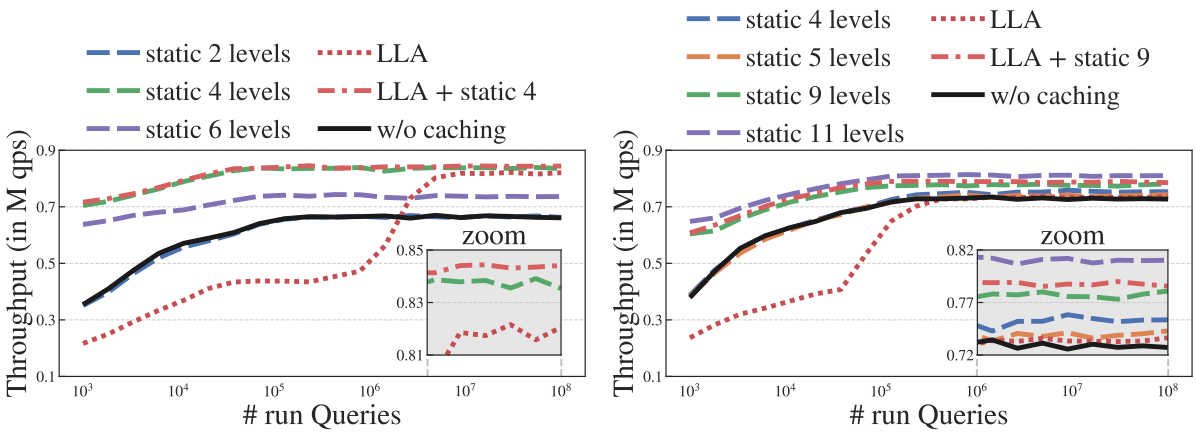
In [JGBS20], dynamic caching has shown to be significantly more beneficial. However, this was due to the fact that the same series of queries was executed twice, which is a sort of selective warm-up. Here instead, we have always queried other tuple or dimension ranges using the *Zipfian* distribution and observe that dynamic caching can only provide minimal gains. As so often, this means that the benefits depend heavily on the workload and its skewness. In the subsequent experiments, we use the best settings for dynamic caching (LLA and naïve), as pointed out above.

[5] robin_hood unordered map: `https://github.com/martinus/robin-hood-hashing`

### Performance of Selective Caching over Time

In the following, we now include static caching in addition to dynamic caching. For both datasets, we show the throughput over time for all three query types. It is to be understood as total queries that ran up to this point in time. The caches are not warmed up and, thus, we show the process of warming up the system and its final steady performance. Besides the best dynamic caching setting, we report the throughput of the pure persistent Elf and the static caching approach. The label `static x levels` means that the first `x` dimension levels in the Elf tree are statically cached in DRAM. On top of that, we also tested the combination of static and dynamic caching. Figure 4.13 shows the throughput over time for exact-match queries.

Since dynamic caching (LLA) has to additionally populate another cache (DRAM), it takes the most time to reach the steady state. However, after this is reached (about 1M-10M queries), it can surpass the non-caching variant of Elf on the uniform data set by 25%. For the TPC-H dataset, dynamic caching reaches its peak performance faster, but it is only 1-2% higher than for the pure PMem Elf. Since too many graphs clutter the figures, we have included only a selection for static caching. If the first one or two levels are statically cached, the performance will hardly change since they will likely end in the CPU cache anyway. The best performance for the uniform dataset is achieved with four cached levels. Three or more than four cached levels settle at `static 6 levels` in Figure 4.13a. It is contrary to the raised hypothesis in [JGBS20] that more cached levels would successively increase the performance. Similar results can be seen with the correlated *LineItem* table, which achieves the best throughput with eleven levels and more. Once again, the behavior is not linear since, for example, four levels are faster than three, but five levels are worse again, and from nine levels onwards, it improves continuously. Other factors besides the number of levels that influence performance are the size of the DRAM cache compared to the CPU caches and, accordingly, the size of the used hash table[6], the rate of successful branch predictions, and frequently traversed *DimensionLists*. For the uniform dataset, for example, levels one and two completely fit in the L1 cache, together with the third level, it becomes a little larger than the L2 cache, and the higher levels are all greater than the LLC. Summing up to level four – the best-performing setup here – results in 136 MiB being




**(a)** On uniform synthetic data.

**(b)** On TPC-H *LineItem* table.

**Figure 4.13:** Continuous throughput of cached Elf variants for exact-match queries.

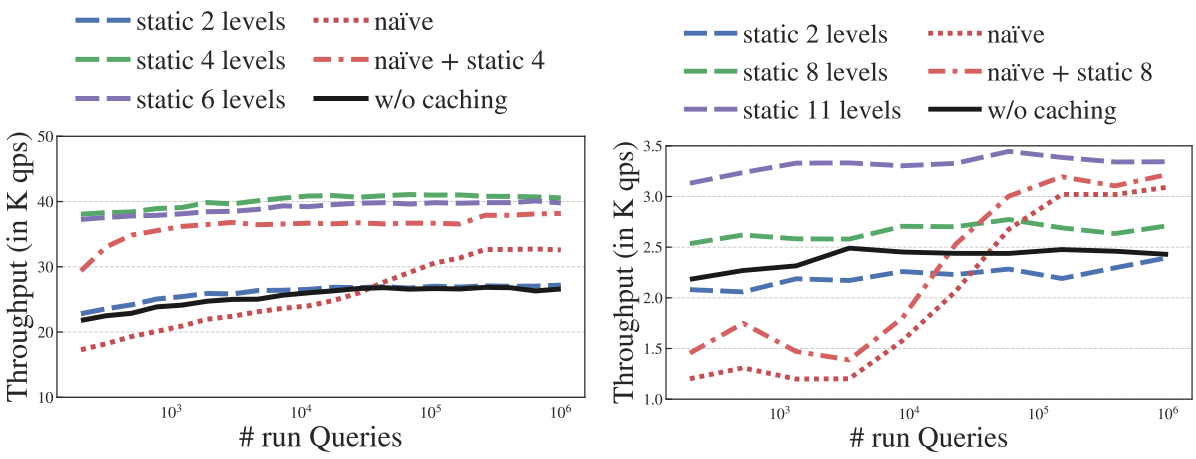
[6]The partitioning of the hash table did not have a positive effect with static caching.

10× larger than the LLC. In comparison with the total size of the tree, this is only 3% more space in exchange for 30% more performance.

Combining the best dynamic approach with the best static one in the uniform case, we get the overall best result. Similarly, for the TPC-H dataset, the combination achieves a better result than the individual approaches. Since the higher dimension levels do not contain many common `DimensionLists` and, thus, there is not much left to be cached by the dynamic part, we used nine static levels instead of eleven. The reason for it is the reordering of the columns of the *LineItem* table. That leads to the fact that dimensions twelve to fifteen are the unique primary and mostly unique foreign keys. Overall, when putting the performance gain in relation to the use of additional DRAM, this combination of dynamic and static caching seems not worthwhile, though. Separately, both caching strategies provide a considerable compromise here.

Next, we will look at the range and partial-match queries using a range size of 2% per dimension. Because many more tree paths are traversed, the throughput is significantly lower than with exact-match queries. The results for the uniform dataset are visualized in Figures 4.14a and 4.15a. The best-performing dynamic approach here is the naïve variant. For range-based queries, it does not achieve quite as sound improvements as for point queries since much more *DimensionLists* are probed per query. This leads to more dynamic cache misses as well as a sequential access pattern. Since PMem can handle this pattern more efficiently (cf. Figure 4.11), an additional cache provides less value. Nevertheless, range queries achieve a gain of about 20% for the uniform dataset. For partial queries, however, it is reduced to 5-10% due to the wildcarded dimensions leading to even more *DimensionLists* being probed. On the TPC-H dataset, as shown in Figures 4.14b and 4.15b, the difference is much more evident. While the dynamic approach speeds up range queries by 30%, it degrades partial-match queries by as much as 10%.

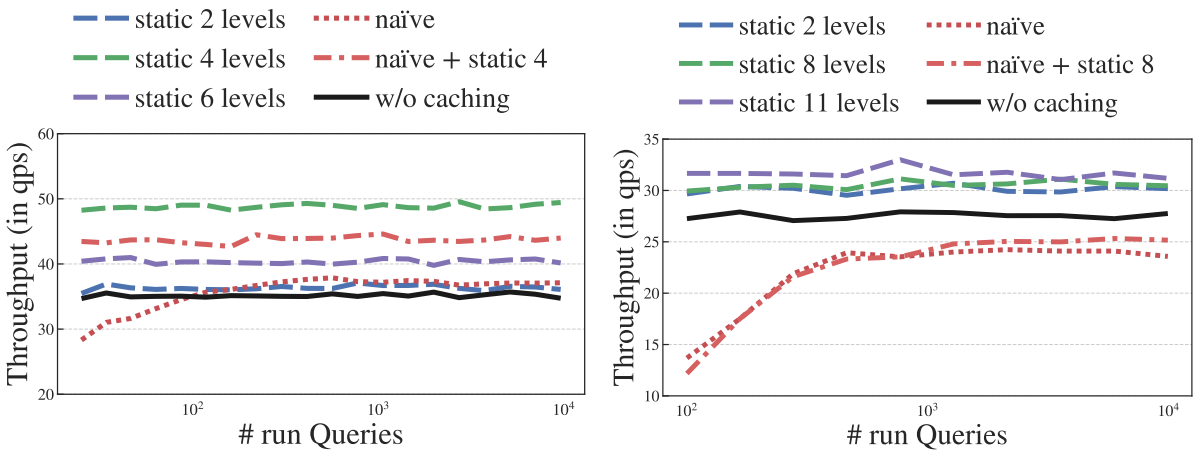
The static approach appears to be more reliable for both data sets, albeit subject to some fluctuations. Similar to the exact-match queries, four and eleven levels perform best for the range-based queries. Range queries are accelerated by 50% and 40% compared to the uncached version for the uniform and TPC-H datasets. Also, the partial-match queries could be improved




**(a)** On uniform synthetic data.

**(b)** On TPC-H Lineitem table.

**Figure 4.14:** Continuous throughput of cached Elf variants for range queries.




**(a)** On uniform synthetic data.

**(b)** On TPC-H Lineitem table.

**Figure 4.15:** Continuous throughput of cached Elf variants for partial-match queries.

by 40% in the uniform and 15% in the correlated case. Two further notable points are that, on the one hand, for range queries in Figure 4.14a, the number of statically cached levels beyond four does not cause much performance degradation and, on the other hand, the static strategy with two levels in Figure 4.14b is slower than the uncached variant by exception. In retrospect, looking at Figure 4.12, the static approach is even better than the customized hybrid dual-access version. It further reinforces the notion of selective caching instead of caching all *DimensionLists*.

Combining both the static and dynamic approach for range-based queries, we get worse performance in this case compared to the standalone static variant. That makes this combination futile. Only with the correlated dataset and range queries (cf. Figure 4.14b), a slight improvement could be achieved. Therefore, we draw the same conclusion as for the exact-match experiments that both strategies are only worth the extra invested DRAM in isolation from each other.

### Parallel Range Queries

As a final experiment of this section and chapter, we consider the impact of running range and partial-match queries in parallel. We only included static caching since dynamic caching would require additional synchronization mechanisms due to the concurrent eviction process. Since the gain from dynamic caching for range-based queries has been small so far in any case, additional synchronization can only be expected to worsen the situation. Furthermore, concurrency control protocols would have to be considered, which would lead to even more parameters in the analysis and complicate it unnecessarily. Therefore, we stick with the best-performing static caching setup (four levels) from before and compare the parallel and sequential performance against the uncached variant. We used the uniform dataset since the cardinalities of the dimensions and range size of the queries can be easily customized. The results shown in Figure 4.16 are based on measurements on a single socket using the same number of threads as available logical cores ($\equiv 20$).

As can be seen, both non-caching and static caching variants can benefit from a parallel execution as long as the cardinality and range size are large enough. Looking at the partial-match results, we see that the parallel variant always performs better. In the case of range queries, the

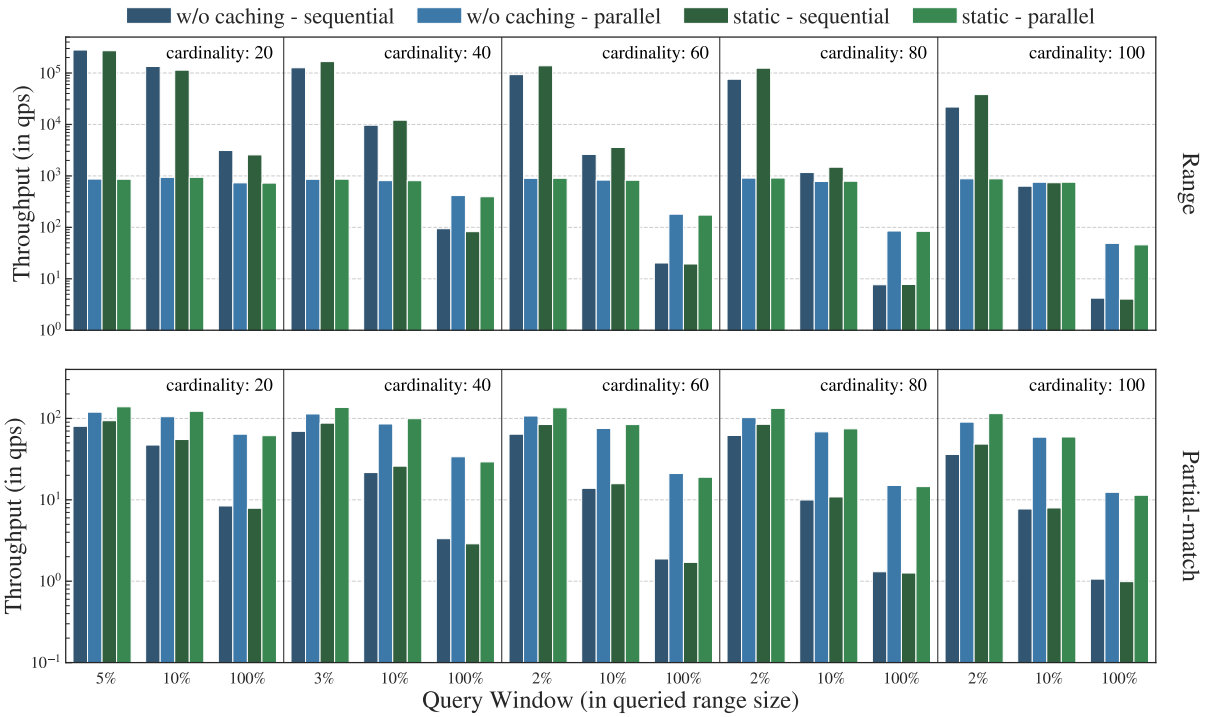



**Figure 4.16:** Sequential vs. parallel range and partial-match queries.

best speedup in this series of experiments was achieved with the maximum tested cardinality of 100 over the entire Elf (see rightmost). The throughput is 12× larger than the sequential version. The cardinality and the number of total tuples can naturally be chosen even higher, which would most likely cause the speedup to increase to some degree as well. Considering the other end of our benchmark, i.e., a cardinality of 20 and a range size of 5%, the parallel execution is utterly counterproductive and worsens performance by two orders of magnitude. Smaller query windows seem to have too much overhead for creating threads and collecting their results. Therefore, the parallel variant is unsuitable as a complete implementation replacement and instead should be chosen based on the passed range parameters and dimension cardinalities. It could be realized, for example, with the help of a cost model, which decides at a certain threshold in favor of either the sequential or the parallel implementation.

If we compare the throughput of the static and the non-caching approach, the differences are hard to discern due to the logarithmic scale. Taking a closer look reveals that the static cache only improves the performance for small range sizes. This insight is consistent with our findings from the previous experiments and is again justified by the sequential access pattern, which can be handled more efficiently by PMem.

## 4.4 Summary

In this chapter, we presented two PMem-based storage layouts targeting analytical workloads. While the first one uses a clustering approach to keep similar data physically close and an interchangeable index, the second is a multi-dimensional index structure that uses DRAM as an additional caching layer.

The introduced clustered PMem-aware storage layout potentially covers all three memory/storage layers (DRAM, PMem, disk) and can thus take advantage of all the properties. The approach allows to efficiently access tuples on non-key attributes, which is especially reasonable for our target TSP system. For example, for selection rates below 1%, range scans can be accelerated by several orders of magnitude. Furthermore, with a volatile index on top, even key-based queries on our structure perform similar to approaches designed for OLTP workloads, and sometimes it even outperforms them.

The multi-dimensional index is a kind of orthogonal approach to the clustering layout. It focuses on the selective caching idea, which statically or dynamically caches tree nodes in DRAM. In our evaluation, we found that random access patterns, in particular, can benefit considerably from investing in additional DRAM to buffer frequently traversed nodes. Thus, for exact-match queries, we reduced the overhead compared to the DRAM implementation from 223% to 150% with just 3% more space. The DRAM implementation would require 100% more memory accordingly. Analogously, we could reduce the overhead of range and partial-match queries with a 2% range size from 210% to 110% and 236% to 140%, respectively. However, it was also evident that a combination of static and dynamic strategies does not always pay off. Only the dynamic naïve and LLA strategies – where the latter is an approach we devised based on access probabilities – could outperform the uncached PMem version. In addition, we have shown that parallel range queries combined with static caching can result in additional performance boosts. Overall, selective caching has proven to be profitable. Also, we think that it is generic enough to apply to other tree-like data structures. Only the granularity of the cached objects will vary. Hence, the optima will need to be determined manually or by an appropriate cost model.

Altogether, the two proposed approaches represent a valuable complement to the previously presented data and index structures to also support analytical applications. In principle, it would be conceivable to combine both approaches, i.e., selectively cache the clustered blocks in DRAM. For our purposes, however, the individual approaches should suffice for the time being. Therefore, we have now scrutinized both OLTP and OLAP preferential data and index structures concerning PMem and can use them as building blocks for more complex systems, such as for the states of our TSP model.

# 5 Stateful Stream Processing

Data stream processing is one of the newer research areas in data management that emerged in the 2000s. Modern applications often require transactional guarantees in addition to fast processing of unbounded data sources, which is driving a convergence of stream and transactional processing. That can be observed, for example, by market trends such as real-time data warehousing or new designs following the lambda architecture that aims to combine batch and online processing using big data platforms. This observation was also described in [CFKK20], which highlights future application requirements and necessary developments in the data streaming domain. These include, for example, transaction guarantees, shared queryable states, cross-state versioning, and modern hardware support. We intend to address all of these aspects within the scope of this chapter. In particular, the exploitation of new hardware, such as PMem, offers sublime opportunities to meet the requirements of modern applications. Since streaming applications often rely on fast response times while requiring fail-safety, PMem with a near-DRAM latency and direct persistence seems to be a natural candidate for transactional and stateful stream processing.

Irrespective of the hardware, the convergence of stream and transaction processing means that the input to such a system is conceived as a continuous stream of data elements. The processing is then realized as a stream processing pipeline (continuous query) with an arbitrary set of persistent tables as sinks. At the same time, updates to these tables can trigger further processing again implemented as a stream processing pipeline. From the traditional point of view, the tables can also be queried in an ad-hoc way, for instance, to create snapshot reports. Thus, such a system consists of a mixture of long-term continuous queries plus usually spontaneous batch (ad-hoc) queries. The data objects, on the other hand, are streams and tables. Probably the most frequently encountered stateful operators are windows. Combined with the table approaches, for example, this offers a wide range of implementation opportunities. Conceptually, a window is nothing more than an ordered table that is filled by a stream and reports the events on the table to another stream. When using existing persistent table implementations, this would support crash recovery basically for free.

However, concurrent access and stream-based queries require extended transaction support. I.e., queries that write to or read from tables must be executed in a transactional context meeting the ACID guarantees. That is not only important for the aforementioned correct failure recovery but also to provide consistent views on persistent subsets of the data stream, as

with the window example above. In this work, we denote this processing style as Transactional Stream Processing (TSP) [BFKT12, MTZ+15]. Summarized the transactional component means that

**A** a stream query writing to tables represents a sequence of transactions, and

**B** stream or batch queries on such tables require transaction isolation.

Following [MTZ+15], a TSP model should address three guarantees: (1) ACID for both OLTP-typical and streaming transactions, (2) ordered execution for streaming, and (3) exactly-once processing of streams. From our point of view, the former is of particular interest since a wide range of implementations is feasible using PMem-based states. Furthermore, this has the most decisive performance impact and, thus, can benefit the most from new memory and storage technologies. The other two guarantees are considered somewhat incidental. The reason is that, for example, for ordered execution (2), nothing needs to be adapted for PMem. For exactly-once processing (3), we could use persistent queues in front of all sources and sinks. During recovery, these queues are checked for already seen tuples or other unique identifiers to determine the last fully processed items in a pipeline. Moreover, external services such as a replayable messaging system [ABD+12, KNR+11] can assist this process.

For illustration purposes, we consider the use case as shown in Figure 5.1: a modern smart metering and monitoring example scenario that could benefit from TSP. It is a more detailed and specialized application of Figure 1.1 introduced in Chapter 1. The data sources are smart meters of private households and measurements from the global infrastructure such as power supplies, generators, accumulators, etc. The idea of this use case is to observe the complete environment and generate alarms if something is not within the specification ranges. This verification step can also be coupled back to realize a self-adapting system. In total, this example contains three continuous and one ad-hoc query. The first continuous query – the beginning is marked as `Stream 1` in the figure – collects the measured values of the private smart meters in 30-minute time windows. That can be both a tumbling as well as a sliding window. Whenever the window operator triggers, it passes the data on to the aggregation operator, which for example, prepares and summarizes the data in a necessary way. Subsequently, the

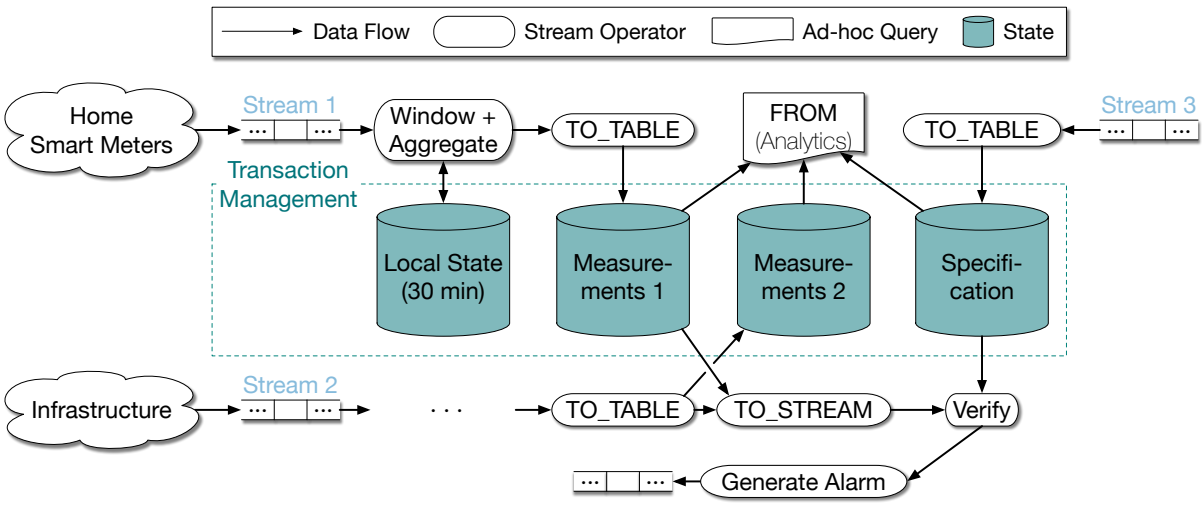



**Figure 5.1:** Smart metering and energy monitoring use case.

results are written to a shared state (table). The second query, based on the measurements of the infrastructure, could be similarly constructed and also writes to a separate state. It then accesses the household measurements and joins them with the current streamed data element (summarized as TO_STREAM). That is compared with the respective specifications. These, in turn, can be updated by another query, e. g., for adapting ranges or adding new machines. If anything falls out of the acceptable range, an alarm is generated. In addition, there could be periodical ad-hoc analyses studying the historical course of events. For this task, a multi-versioning approach which, for instance, also supports time-travel queries, would be sensible. Conceptually, tables are used in this scenario to maintain persistent states to deal with voluminous datasets (e. g., windows ranging over hours or days) and to support crash recovery. However, as the example indicates, such tables are not only appropriate for internal states but can also be queried on an ad-hoc basis to get the current (or even historic) picture of the process or individual process states.

From the specified conditions, desired guarantees, and the example, we can derive explicit requirements. Thus, to support transactional and queryable states, a correctly working system must fulfill the following points:

1. State representations (tables) must be queryable as a matter of principle.
2. Concurrently running stream queries updating the state and ad-hoc queries to these states must ensure the isolation property.
3. Consistency between multiple states associated with the same query is necessary, also in the case of transaction aborts.

For the former, this includes providing a unified interface for operator states and separately created tables and centralized management of these. To realize the isolation property, a scheduling protocol like MVCC is necessary for handling both batch and continuous queries. For consistency preservation across multiple states, there is also the need for a protocol that coordinates all accesses with ACID guarantees. A simplification, which is also evident in the illustrated use case, is that only one continuous query has write access to a state, thus creating a single-writer-multiple-reader scenario.

This chapter aims to explore techniques used to meet the above requirements[1]. The ultimate goal is to prototype a stream processing system with transactional capabilities using PMem-based data structures as state and table representations. Our contributions provided by this chapter are as follows:

- **A Data-centric TSP Model**: We propose and explain our TSP model conception in detail with a distinct focus on transactional state management and queryable states.
- **Concurrency & Consistency Protocol**: We present our implementation of an MVCC and consistency approach for the TSP model. We further discuss suitable optimizations in the presence of PMem. Based on the protocols, we define the necessary steps for failure recovery and application restart.
- **Query Planning**: We outline possible parameters and cost models for query planning in the TSP model.

[1]The material in this chapter is based on [PGS17, GS19, GPS19, GGK+20].

- **Event Stream Processing**: A specialized but well-established form of stateful stream processing is the continuous processing of events. Based on an existing event store, we show how to practically apply our previous findings with PMem to the store's components.

The remainder of this chapter is organized according to the following structure. In Section 5.1, we start by introducing our transactional model for stream processing. Then, Section 5.2 surveys state-of-the-art research regarding stream processing with transactional guarantees and current deployments of MVCC as a concurrency control protocol compared to our approach. Section 5.3 presents our protocols for realizing the snapshot isolation property to meet the transactional requirements in this TSP model. Subsequently, we address the aligned implementation for PMem and how the recovery process has to be adapted in Sections 5.4 and 5.5, respectively. In Section 5.6, we take a brief detour into query planning opportunities and cost models for TSP. Furthermore, we apply our experience to a specialized form of stream processing, namely event stream processing in Section 5.7, before evaluating both processing types in Section 5.8. Finally, Section 5.9 summarizes the contents and insights of this chapter.

## 5.1 Transactional Stream Processing Model

Transactions are a well-established concept known from DBMSs that guarantees consistency for the underlying database even in the event of failures and concurrent user access. Transactional Stream Processing can be seen as a hybrid model of the traditional relational data processing and the data stream processing that provides both with transactional guarantees. Through this combination, we can distinguish between two objects holding data: *tables* for representing states and *streams*. Tables constitute a structured and finite collection of data typically divisible in rows and columns. Streams, on the other hand, provide a potentially infinite sequence of tuples that exhibit either an implicit or explicit ordering. I.e., either the ordering is based on arrival time, or the tuples carry an ordering attribute such as the application timestamp. Still, both objects need to be provided with a concrete schema. A further difference is that streams are volatile and can often be processed completely in-memory, while tables are typically persistent and require a physical storage representation.

In order to formulate queries on data streams the data is passed through a flow of operators such as filter, map, join, group, aggregate, or even user-defined operators. That results in so-called continuous queries. Unlike the relational case, due to the boundlessness of the data stream, certain operators that typically require the entire dataset to return a result cannot be executed in this form. These operators, such as joins and aggregates, are called blocking operators. It is solved with windows that divide streams into bounded subsets. Windows can concern either a fixed period or quantity of tuples or be data-driven and allow various eviction policies such as sliding or tumbling windows. Furthermore, relational queries differ from continuous queries in the form of processing. While the former follows a pull-based model like in the Volcano processing model [GM93], the latter is typically rather push-based. It means that tuples are actively sent from the source(s) to the consumer operator(s). Due to these differences, it must first be clarified how both paradigms can be linked to each other.

### 5.1.1 Linking Operators

Similar to the concepts which were proposed in the query language for STREAM [ABW03], we need two classes of operators to link tables and streams. One for the stream-to-table direction and the other class for the opposite way. In this work, we refer to them as TO_TABLE and TO_STREAM that need to perform the following tasks:

- TO_TABLE ingests tuples from a stream and inserts them into a table or updates or deletes existing ones, e.g., to modify an operator state.
- TO_STREAM monitors arbitrary changes on a table and generates a stream of tuples from them.

The handling of stream tuples with TO_TABLE can be different depending on the context or a potentially connected stateful operator. Usually, though, if a tuple with the same key as the stream tuple already exists in the table, it will be updated and otherwise newly inserted. Deletions can be performed either explicitly by delete tuples or implicitly when a tuple is obsolete, e.g., due to window semantics. For TO_STREAM, we can further subdivide two scenarios. First, stream tuples are processed incrementally, such as in an aggregation, where only a single table tuple is modified and subsequently output to a stream. The second scenario comprises the case where table-wide operations are performed before a new stream tuple can be output, as with a median calculation of all tuples in a table. Thus, TO_STREAM is very similar to the concept of a database trigger that generates one or more tuples once the defined condition on a table is fulfilled. It can also be extended to full trigger semantics, which additionally perform modifications to the table. For that, the output stream only has to be redirected to the table and linked with TO_TABLE. Besides the two classes of operators to combine the streams and tables, it needs another operator class FROM, which allows ad-hoc queries. These also need to cover two flavors. On the one hand, to hook onto a stream and get all tuples starting from the time of attachment (motivated by views) and, on the other hand, to issue typical relational queries to tables. In Figure 5.2, we have visualized the interplay between these three classes of operators. The central semantics are marked with A for atomicity, I for isolation, and T for trigger policy. The handling of transaction boundaries and state access is described in the following subsections.

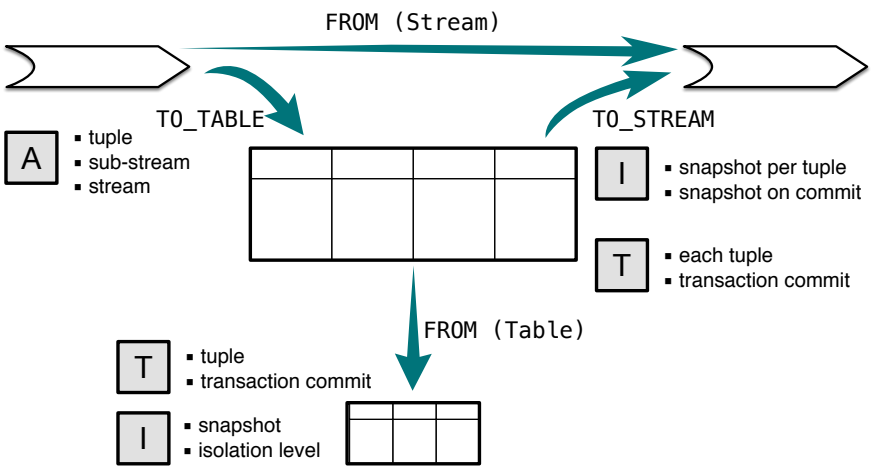


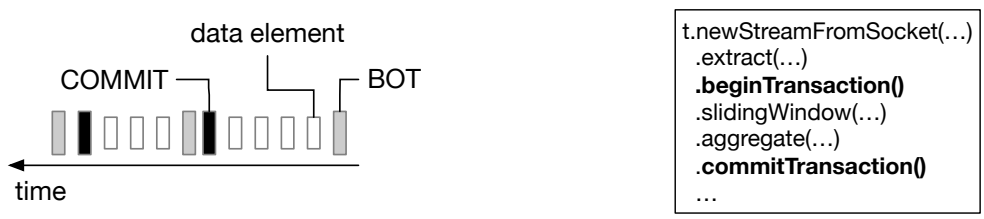

**Figure 5.2:** Overview of linking operators and transactional semantics for TSP.

### 5.1.2 Transaction Boundaries

As already indicated by the figure, there are basically three ways to treat atomicity and transaction boundaries for data streams. The first and most trivial case is when each tuple in a stream represents a separate transaction, which we refer to as *auto-commit*. For a *data-centric* approach, however, we need explicit, dedicated stream elements that mark the transaction boundaries (BOT, COMMIT, ABORT). These pass the stream pipelines alongside the actual data tuples interpreted as inserts, updates, or deletes. For the realization of these concepts, *punctuations* [TMSF03] or *control tuples* are well suited. These can be injected already from the streaming data source or as part of the stream processing pipeline(s) when a predefined condition is met. Therefore, a transaction following this approach is always a bounded sequence of tuples (i.e., a sub-stream). However, Out-of-Order (OOO) events can also occur in data streams, which means that the ordering by arrival is inadequate. Therefore, in this case, tuples of the same transaction must be tagged by concepts such as punctuations, equal timestamps, or similar. Alternatively, transaction boundaries can be specified as part of the query or dataflow program as in the traditional *query-centric* approach. It would define a transaction as a sequence of operations, which seems more reasonable for ad-hoc queries. Figure 5.3 illustrates the data- and query-centric strategies. In summary, a transaction can span the entire data stream to the length of a sub-stream, or it may involve only a single tuple ($\equiv$ auto-commit).

### 5.1.3 Transactional State Management

Data-manipulation operations like inserts, deletes, and updates on tables must be executed in a transactional context so that atomicity for writes and isolation for reads (via TO_STREAM or FROM) can be guaranteed. In our TSP model, a table can only be modified using the TO_TABLE operator. To guarantee atomicity for these writes, the transaction boundaries as described above are required in the first place. Moreover, since a stream query can operate on multiple persistent states simultaneously (cf. Figure 5.1), a consistency protocol is necessary that encompasses all states involved. For the states manipulated in the same transaction, it means that they must be updated synchronously from an external perspective. Thus, depending on the isolation level, other queries that simultaneously read the states should only see updates of the same and possibly the most recent transaction that has already been committed. Since, in our data-centric model, we view a transaction as a sequence of stream tuples containing update information, usual OLTP queries are modeled as a sequence of updates by simply concatenating all changes issued by a transaction. This sequence is then wrapped with transaction punctuations and fed to the TO_TABLE operator to execute the modifications on the table.




**(a)** Data-centric. **(b)** Query-centric.

**Figure 5.3:** Strategies for defining transaction boundaries.

Reads via the FROM operator can be executed using various isolation levels to control the corresponding visibility of simultaneous updates, as known from DBMSs. Furthermore, this concept also needs to be applied when FROM is used to attach to a data stream. Thus, for example, snapshot isolation should only forward committed transactions. As opposed to that, with a more relaxed isolation level, stream tuples of a transaction that is still running can already be emitted.

Reading using the TO_STREAM operator requires a trigger policy added to the consideration of the isolation property for all reads. A trigger policy is a constraint that causes stream tuples to be output or back-to-the-table streams to be produced that contain the transactional modifications. For instance, it is possible to react to arbitrary tuple modifications or only to transaction commits. Furthermore, the trigger policy also affects the isolation property. Hence, a snapshot includes only a single tuple or all tuples that have been modified up to the commit.

### 5.1.4 Shared Queryable States

In order to ensure all relevant facets required for the TSP model with queryable states, we draw on a unified table model as used in DBMSs. Besides user-defined tables, some stream operators such as windows, aggregates, and joins also require underlying and, at best, optimized data structures to maintain their state. For these operators, tables (or table-like interfaces) can be exploited as internal structures to make their state generally queryable or share their contents with other queries. That allows providing shared queryable states while reusing existing persistence and recovery mechanisms.

As an example, we first consider the window operator as known from streaming systems. It can be easily realized with a pair of TO_TABLE-TO_STREAM operators that store and remove tuples in and from a table. That makes the underlying table available to all other queries, e.g., for calculating various statistics or monitoring. In addition, in case of failures, the windows are persistent and recoverable. Generally, the underlying tables can automatically be created based on the query definition. Also, the schema is implicitly provided by the operator's input schema. As noted earlier, a single implementation of the table concept will not be sufficient because the requirements and access profiles vary by operator. Thus, operations on the tables may involve merely append-only, batch appends, and deletes, or need to cover the whole range of update operations. For further processing of the data stream, the TO_STREAM operator allows to incrementally evaluate a window feeding only updates into the data stream. On the other hand, also batch processing with access to the entire window contents is possible. It can be done in the form of ad-hoc queries using the FROM operator, which provides the full range of relational queries.

The same applies to grouping, aggregation, join, or other stateful operators. They can all be implemented similarly to provide external access to their state. For example, in a scenario with multiple data streams from different sources, the same specification table might be required for joining. Rather than creating a hash table for each stream, all join operators can now reuse a single hash table. In this way, redundant work is avoided, and the system's memory consumption is decreased.

## 5.2 Related Work

Several models and systems already exist that use transactions and persistent states for processing data streams. In this section, we summarize the existing approaches and highlight their differences. We divided related work into three categories. The first covers stream processing systems that support transactions and states. Then, we discuss those systems that focus more on scalability on multiple cluster nodes. Finally, we will review current multi-version concurrency control considerations. Related work regarding PMem-based data and index structures as part of the possible state representations was already discussed in Sections 3.1 and 4.1. Besides the differences outlined below, we will see that none of the listed systems consider PMem, and only a few include NUMA effects. It is a contribution that we address in this chapter, among others.

### 5.2.1 Transactional Stream Processing

#### STREAM

One of the first approaches that combines data stream and transactional processing is the STREAM project [MWA+03, ABW03]. They transform relations to streams by introducing the operations *RStream*, *IStream*, and *DStream*. These emit tuples to a stream whenever changes are observed in a relation. While *IStream* and *DStream* generate a stream of tuples based on inserts and deletes into or from a table, respectively, the *RStream* operation emits all tuples of a table enhanced with the same timestamp. For the former two cases, the tuple timestamps correspond to the respective time of insertion or deletion instead. Similar to our model, STREAM also exploits punctuations to model constraints such as transaction borders over data streams. For the opposite transformation direction, windows are used to convert sub-streams into relations. These can then be queried in a typical SQL fashion. However, their system focuses on continuous queries and mostly neglects the interplay between stream- and relational-based transactions.

#### Transactional Stream Processing

With the work Transactional Stream Processing [BFKT12], from which we adopted the name of our model, this notion is further advanced, and also concurrency control and failure-atomicity are incorporated. Here, a uniform transaction model is already being conceived to handle both stream- and relational-based transactions. In order to achieve that, timestamps are assigned to each transaction, and continuous queries are converted into a series of one-time (ad-hoc) queries. States and independent relations are maintained in a separate storage manager and can be queried through a transaction manager interface. The storage manager is based on a storage system for data-intensive stream processing [BAF+09]. As a result, the transaction manager does not distinguish whether continuous or ad-hoc queries are received. Either way, concurrency control utilizes locks (SS2PL), and recovery is based on logical undo logging. However, we see a potential for improvement in this protocol implementation of the transaction manager, particularly when considering modern and upcoming server setups such as multi-socket systems with PMem.

#### S-Store

In a similar way, S-Store[MTZ+15] builds on an existing OLTP system and extends it with data stream processing capabilities. In particular, they reuse the ACID implementations of H-Store [KKN+08] by representing data streaming concepts such as windows and streams as time-varying tables ordered by timestamp. Continuous queries, on the other hand, are realized as a dataflow graph composed of stored procedures, which can also be nested. The transaction scope is defined by a batch ID – derived either from a timestamp or from the number of tuples – and executed atomically per stored procedure. With this way of processing, ordered execution and exactly-once processing can be realized in addition to the ACID properties. Even though S-Store fulfills the functional requirements of a TSP system, it is still based on a relational DBMS and, thus, leaves a lot of room for improvement, especially in the streaming domain with stringent latency requirements.

#### Tidalrace

With Tidalrace [JS15], the authors propose a streaming data warehouse, contrary to the previously introduced streaming OLTP systems. The key idea is a system that acts as a data sink for streams and facilitates ETL tasks as well as final data warehouse analyses. In order to convert the incoming streams into tables, they are partitioned based on timestamps and stored in write-once files. When querying data, Tidalrace focuses on the final warehouse tables, which means that it is not possible to mix stream and relational queries. Furthermore, when ingesting the stream data, the consistency property is relaxed to obtain real-time results. It means that the transactional part, i.e., the preservation of the ACID properties, is mostly omitted.

#### TStream

A more recent approach is TStream [ZWZH20], which is a transactional stream processing system especially targeting modern multicore processors. The authors propose two disjoint scheduling modes to scale the state synchronization. The first is the compute mode in which the system starts. It processes events and collects state access operations in a list per state. Every time a defined period is over, the executors switch to the state access mode. Here, the postponed events now access the state in batches. It can also be done in parallel if these have no data dependencies. Another contribution is the consideration of NUMA-aware processing by allowing different sharing options per state access list.

### 5.2.2 Scalable Stateful Stream Processing

#### Apache Flink Streaming

Considering scalable distributed data stream processing, Apache Flink Streaming is one of the most used and advanced platforms that provides consistent and fault-tolerant states [CFE+15, CEF+17]. The fault tolerance is enabled by a mechanism based on distributed global snapshots, as primordially proposed by Chandy and Lamport already in the 80s [CL85]. For that, punctuations – here referred to as stream barriers – are periodically ingested to all source operators and move alongside the data to the sinks. At each operator in the pipeline, the data flow is blocked until the barriers with the same ID are received from all other inputs. Once that is the case, the

contents of the state are persisted in an arbitrary state backend. If a failure should occur, the last consistent snapshot is used to restore all states of the same streaming pipeline collectively. In addition, it is possible to query states at least in the form of point lookups as long as only one operator/task has write access to this state. However, to the best of our knowledge, they do not support transactions across multiple states and do not comply with the ACID properties in the process.

#### SnappyData

Based on the Spark platform, SnappyData [MRM+17] extends it with an in-memory transactional store and, thus, aims to unify transactions, streaming, and analytical tasks in one framework. The task distribution is such that Spark is used for fast distributed computations and streaming features, while the in-memory store allows for fine-grained concurrent access control. They take a hybrid approach to the storage model, allowing both row- and column-oriented tables. The micro-batches introduced by Spark are used to define the scope of a transaction, which means that transactions must always be of the same size. Another feature is the capability for approximate query processing to meet real-time requirements, for example.

#### TSpoon

TSpoon [AMC20] fits into a similar category, where the authors build a transaction model based on Apache Flink. They introduce the *t-graph* notion, a subgraph containing all stateful operators that must jointly satisfy the ACID properties. It is relatively similar to what we call a topology or state group. Another similarity is that only streams can write to states. However, a difference to our model is that TSpoon does not use punctuations to mark transaction boundaries. Instead, a transaction can only comprise one streaming element at a time. In order to cope with concurrent queries, key-value pairs are versioned and can be managed using either a lock-based or a timestamp-based protocol.

### 5.2.3 Multi-Version Concurrency Control

Particular concerning modern multi-core architectures, MVCC can maximize parallelism while still achieving serializability. Primarily, it is attributable to the principle that read and write operations do not block each other. That makes the system much more scalable and, therefore, many commercial DBMSs have adopted it as their Concurrency Control (CC) scheme. These include, for example, Hekaton [DFI+13], MemSQL/SingleStore [Sin20], and SAP HANA [LMM+13]. MVCC is also favored in almost every current academic and open-source DBMS like Postgres [SR86], HYRISE [GKP+10], HyPer [KN11], Peloton [Car19, WAL+17], and NoisePage [Car21]. However, it is principally not a protocol but rather a group of protocols, and there is no exact standard on how to implement it. Several options and customizations are conceivable, depending heavily on the expected workload. Most of the MVCC implementations only support the snapshot isolation level since full serializability would be disproportionately expensive. Hence, some approaches exist guaranteeing serializability with as little overhead as possible [NMK15, CRF08]. In a single-writer-multiple-readers scenario, snapshot isolation corresponds to serializability since write-skews cannot occur. That is why we do not need such enlargement in our approach for the time being. A comprehensive study examining the

major design decisions for the implementation of MVCC in an in-memory database system can be found in [WAL+17]. More precisely, these design points are the underlying CC protocol, the version storage, the garbage collection, and the index management. The authors discuss tradeoffs and appropriate scenarios for each approach. We adopted some of the results of this study to develop our own MVCC approach (see Section 5.3.2). However, several aspects could not be applied in this way, as our system is not a pure in-memory solution. Even though the design space for PMem remains roughly the same, the performance implications of some design adjustments presented will not be identical to those observed for DRAM due to the different characteristics of the technology, such as the read-write asymmetry.

A few works exist that already use versioning or MVCC for PMem-based data structures. Among them are, e.g., CDDS [VTRC11], SOFORT [OBL+14], Dash [LHWL20], and Zen [LCC21]. Most of these proposals, except Zen, and to some extent Dash, omit an exclusive concurrency comparison with other protocols or have still been evaluated on DRAM-based emulations. However, the Zen approach already reveals that at high skew, MVCC and optimistic protocols achieve higher throughput than lock-based protocols. In this chapter, we will explore in greater detail if and when MVCC on PMem is more appropriate – particularly in the context of stateful stream processing – than protocols without versioning.

## 5.3 Snapshot Isolation Protocols

Considering the transactional semantics as detailed in Section 5.1, we can now further specify and address the requirements described at the beginning of this chapter. Fundamentally, this is an application of the ACID principle to the TSP model. We start with atomicity, which must be taken into account in several places. Primarily, it concerns the afore-said marking of transaction scopes – the unit of work to be executed atomically – via certain transaction boundaries. Furthermore, every operation of the transactions must be executed atomically to ensure fault tolerance on the one hand and consistency even for concurrent access on the other. That brings us directly to the next two requirements, namely persistence (or durability) and isolation. The isolation property includes both continuous and ad-hoc queries. It has to make sure that they do not interfere with each other regarding correctness and consistency. That also holds when they cover multiple states or transaction aborts occur. On the other hand, the persistence property requires that all effects of a successfully committed transaction still exist after a system restart, whether intentional or not. It also includes recoverability, which must ensure that states can either be fully recovered or always remain in a consistent form. In order to comply with these requirements, we have developed a snapshot isolation approach that consists of the following three components.

- Multi-versioned data structures for queryable operator states and custom tables
- A transaction protocol supporting reads, writes, commits, and aborts on these states both by stream operators and ad-hoc queries
- A protocol that maintains consistency across multiple states

We have prototypically implemented and integrated these components into a C++-based data stream processing framework called PipeFabric[2]. The reason we chose an MVCC approach is that it has proven to be the most scalable and widely used CC protocol in the literature for DBMSs [PA16, WAL+17, CM86], as we have already pointed out in the related work section. For a TSP environment, we expect similar scalability and resilience of this approach. However, this assumption still needs to be substantiated. In addition, the suitability of MVCC in combination with PMem has not been extensively studied yet. We will address both aspects later in the chapter.

### 5.3.1 Data Structures

As a data structure supporting snapshot isolation and serving as transactional state representation, we have designed a table wrapper shown in Figure 5.4 on the right. The areas highlighted in gray in the figure represent the parts that need to be stored persistently. The underlying `Base Table` can be any arbitrary data structure as long as it has a key-value mapping, which is the case with frameworks like RocksDB or elementary representatives like a hash table or a $B^+$-Tree. Thus, for each type of state – depending on the access profile – a matching underlying structure can be used, making this design highly versatile. For example, for an aggregation or a grouping, a multimap might prove to be the most efficient. Also, more nested data structures would be conceivable, e.g., for grouped windows, one could map IDs to FIFO queues (cf. Figure 5.1). Inside the base table, each key is mapped to its corresponding value, which in this case is an `MVCC Object`. As it is typical for MVCC [Ree83, WAL+17], a version entry, in turn, has the following form: $< [cts, dts], value >$. The Commit Timestamp (CTS) and Deletion Timestamp (DTS) fields depict the validity range of this version's value. The apparent separation into two arrays (headers and values) is expected to provide a better cache locality when searching for a valid version, particularly when storing many versions or large values. We use a bit vector (omitted in the figure) to atomically handle the available free slots in these two arrays. No additional write lock is necessary since changes are initially stored volatile in the uncommitted write set of the corresponding transaction until commit. Thus, new versions are not prematurely visible to simultaneous readers, and transactions can also be aborted straightforwardly and fast. Furthermore, committed and uncommitted versions are never mixed. The detailed operation of the commit is explained in the following subsection. Finally, the transactional table wrapper maintains a reference to the global state context, as shown on the left of Figure 5.4.

The state context holds all necessary runtime information about the states, state groups, and active transactions registered in the system. In this sense, the state context is also the transaction manager. For the states, we currently only store concise general information such as a unique identifier and their physical location. This location is usually a file system path for both PMem- and disk-based states and a virtual address in the case of volatile states. State groups record which states must be written together atomically. It can typically be derived from the continuous queries, which are called *Topologies* in PipeFabric. Since we currently focus on a single-writer-multiple-readers scenario, a state can only be written by one such topology. It is essential to keep track of the corresponding state groups to correctly apply the concurrency and consistency protocol (see Sections 5.3.2 and 5.3.3). For that, the last committed transaction

[2]PipeFabric - `https://github.com/dbis-ilm/pipefabric`

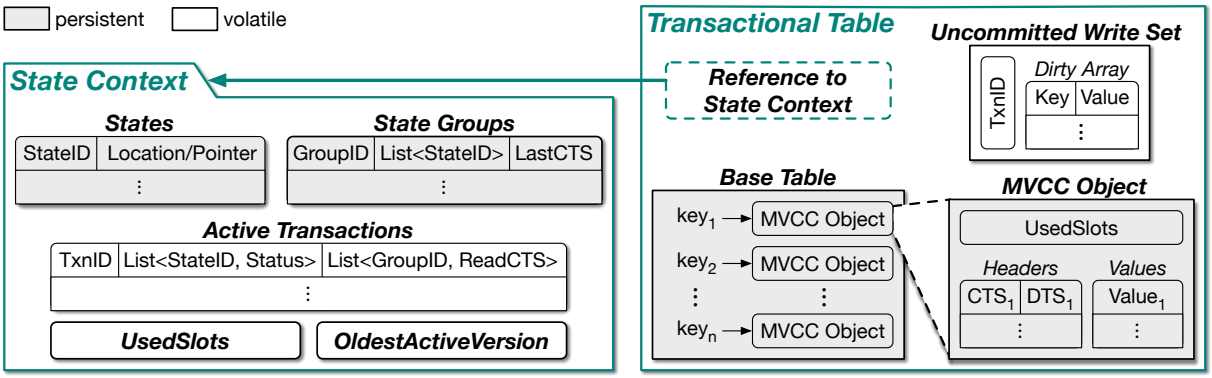



**Figure 5.4:** Transaction components.

timestamp (`LastCTS`) is maintained for each group marking a snapshot. This information must also be stored persistently to prevent individual parts of an incomplete commit from becoming visible after an unexpected failure. Upon starting a new transaction, it acquires a unique timestamp (`TxnID`). These timestamps are generated by a global atomic counter within the state context. All active transactions are assigned with a list of states they accessed or are going to access. This list contains the ID of the states and the access status (`Active`, `Abort`, or `Commit`), which is mainly used in the case of writes. Furthermore, we also keep track of a global commit timestamp at the time of reading (`ReadCTS`) for each state group. For the active transactions, we again use a bit vector[3] to atomically (de)allocate the available slots in this array (`UsedSlots`). All common variants, such as background vacuuming, cooperative cleaning, and so on (cf. [WAL+17]), are possible for garbage collection considering the active transactions. Currently, the invisible or old versions are marked free only if there is no space for the new version in the array. For this purpose, the oldest version considered by active transactions is recorded (`OldestActiveVersion`). All versions with a lower timestamp can be safely marked free or archived if time traveling is demanded. Altogether the state context of the transaction management can be used entirely latch-free exclusively copes with atomic instructions.

### 5.3.2 Multi-Version Concurrency Control Protocol

Initially, we consider the necessary basic transactional operations on a state. These are *read*, *write*, *commit*, and *abort*. More complex procedures such as update, scan, modify-if, etc., can be composed of these. To better separate the protocols, we first consider the simultaneous and consistent access to only one state at a time. The start of a transaction can be signaled either explicitly by punctuation or implicitly by the first read/write operation. In our case, we assume beginning punctuations that trigger the assignment of a timestamp to the transaction and the registration in the context. Depending on the physical placement of the state, different synchronizations of the read and writes of MVCC objects are necessary. For disks, e.g., a lightweight locking strategy with read-write locks (latches) could be used. Specialized handling in the case of PMem will be considered in Section 5.4.

When reading, as shown in Algorithm 6, the first step is to see if the corresponding transaction has already recorded a write for the requested key in its write set and returns this instead (lines 2-5). If this is not the case, the base table is searched for the specified key, and the

[3]In fact, it is a 64-bit integer, which is updated by CAS operations.

associated MVCC object is retrieved (line 7). At this point, a synchronization with commit operations is potentially necessary to maintain atomicity and isolation. If no entry yet exists, the operation will inform the caller accordingly. To achieve snapshot isolation, the first read version timestamp of this transaction must be transiently stored and is used for subsequent reads (lines 11-13). More relaxed isolation levels can merely read the latest (visible) version. For the desired snapshot isolation, however, the corresponding version is next looked up using the previously determined `readCTS` (lines 15-17). No visible version may be available for this transaction, which is handled as not found. As the last step, the read value is copied and returned (line 18-19).

**Algorithm 6** Read(txnID, key, outValue)

```
1:  /// Read own version if written before
2:  status ← ownAvailable(txnID)
3:  if status == SUCCESS then
4:      outValue ← writeSet[key].value
5:      return status

6:  /// Retrieve MVCC object
7:  (status, mvcc) ← baseTable.get(key)
8:  if status != SUCCESS then
9:      return NOT_FOUND

10: /// Handling Consistency
11: readCTS ← StateContext.getReadCTS(txnID)
12: if readCTS == 0 then
13:     readCTS = lastCommitID

14: /// Extracting latest visible version
15: pos ← mvcc.getCurrent(readCTS)
16: if pos == −1 then
17:     return NOT_FOUND
18: outValue ← mvcc[pos].value
19: return SUCCESS
```

The writing procedure, as it can be seen in Algorithm 7, is relatively straightforward since only the new value is added into the write set of the transaction (line 3). No exclusive locks are necessary because we assume and allow only a single writer (lines 1-2). Therefore, write operations are not blocking. If multiple writers are to be supported, the write sets would have to be checked for overlaps. In case of an overlap, the younger transaction could be prematurely aborted or restarted. Alternatively, this check could be done only at commit time, similar to optimistic approaches, to avoid slowing down writes. As blind writes are not always appropriate, an update can be performed as a read-modify-write sequence.

**Algorithm 7** Write(txnID, key, value)

```
1: if writeSet.txnID != txnID then
2:     return NOT_ALLOWED
3: writeSet.append(key, value)
4: return SUCCESS
```

Next, we look at the transaction terminating operations. Aborting a transaction is straightforward as all uncommitted writes are only stored in a volatile structure, which can plainly be discarded or marked free. However, the commit operation requires a little more effort since the changes that were previously only stored in volatile form have to be persisted. Of course, atomicity and isolation must be observed again. The procedure is shown in Algorithm 8. In the first loop, all changes are prepared in memory. Here, each affected MVCC object must be loaded first (lines 4-5). After that, a free version slot and the slot of the latest visible version are searched (lines 6-7). If a free position is not directly available, the garbage collection is triggered at this point. Under certain circumstances, it can lead to waiting situations until older read transactions have been processed. The slot positions are used to insert the new values and limit the validity of the current version (lines 8-10). If no entry exists yet, a new one is created, and the procedure is slightly different, which was omitted here for the sake of clarity. The second loop is then responsible for populating the changes atomically and isolated into the base table (lines 12-13). Here, synchronization with the read operations, e.g., via key-based latches, is required. While our algorithms can handle inter-thread synchronization, the underlying table must guarantee failure-atomicity for a single update. If that is the case, all transactions can be executed with ACID guarantees. The final step is to update the global commit timestamp of the state with the ID of the committing transaction (line 14). Note that this field must also be stored persistently. Updating this timestamp atomically ensures that the changes are either fully visible or not at all. It is possible because the same timestamp is used for reading instead of a transaction's own ID (cf. Algorithm 6 - line 13). Since incomplete changes are not visible, no undo is necessary. To support multiple writers, write locks are required, and also the order of the commits must be respected. For example, it could follow the *first-committer-wins* rule, i.e., if the current version is already newer than the timestamp of the transaction at hand, it must be aborted.

**Algorithm 8** Commit(txnID)

```
1:  /// Buffering new MVCC entries
2:  newEntries ← {}
3:  for each (key, value) ∈ writeSet do
4:      newEntries.append(key, baseTable.get(key))
5:      last ← newEntries.tail.mvcc
6:      iPos ← getFreePos(last.usedSlots)
7:      dPos ← last.getCurrent()
8:      last[dPos].dts ← txnID
9:      last[iPos] ← (txnID, inf, value)
10:     setBitAt(last.usedSlots, iPos)

11: /// Update MVCC objects
12: for each (key, mvcc) ∈ newEntries do
13:     baseTable.update(key, mvcc)
14: lastCommitID ← txnID
```

The algorithms outlined above are kept generic at first and work like this also on block-based storage. However, if PMem is available, all accesses can be fine-grained, and update operations can be atomically performed in-place. That also allows bypassing the OS cache and executing direct load and store instructions. Thus, the logical time of persistence is known. In addition, latches during the commit can be replaced by latch-free methods, e.g., compare-and-swap instructions. Therefore, we present a more optimized variant for PMem in Section 5.4.

Fundamentally, the design of the operations (without PMem optimizations) already allows the elimination of logs for the most part. Furthermore, versioning generally prevents read operations from being blocked by write operations and the other way around. As we have shown, only during the commit, brief synchronizations with reading transactions can occur. There can be no conflict-induced aborts, which should keep the performance stable even for long transactions and high contention situations. Moreover, versioning enables time-traveling queries if demanded. However, compared to, e.g., a simple lock protocol, disadvantages could be increased program complexity and a higher memory and storage consumption.

### 5.3.3 Lightweight Two-Phase Commit Protocol

The previously described procedures were based on only a single state. However, if multiple states are updated in a continuous query, the changes must become visible simultaneously to maintain consistency. In order to demonstrate it more clearly with an example, we will assume a simple scenario, as shown in Figure 5.5. Ignoring the state context, for now, it consists of a continuous query that writes to two states and an ad-hoc query that reads from the same two states. If a commit is received by the first TO_TABLE operator, the visibility of the modifications must be delayed until the commit has reached the second operator. Only if both are able to commit, the changes can be applied in compliance with the ACID guarantees.

To achieve this cross-state consistency, we coordinate the operators using the state context shown in Figure 5.4. Hence, when a commit reaches a state, first, the status field of the transaction for that state is set to `Commit`. As soon as the last stateful operator of the transaction receives and sets the commit, it triggers the actual persisting of changes. The last stateful operator automatically becomes the coordinator and is thus responsible for the global completion of this transaction's commit. If an `Abort` status is set for one or more states, the transaction will be aborted globally. Such an abort can be caused either by punctuations or local conflicts,

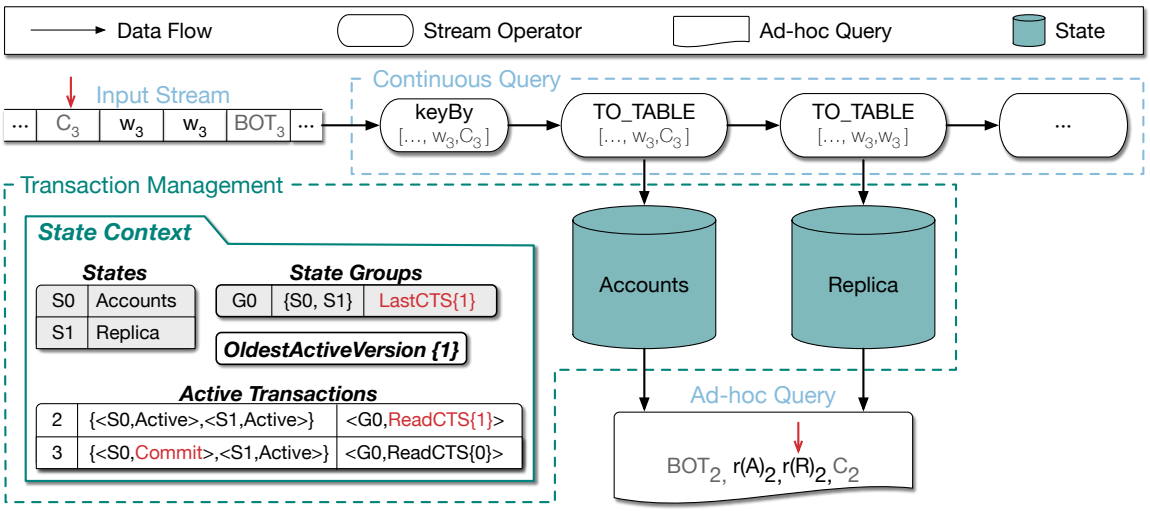



**Figure 5.5:** An example scenario for handling concurrency and consistency: One continuous writing query (`BOT`, `Write`, `Write`, `Commit`) and an ad-hoc reading query (`BOT`, `Read`, `Read`, `Commit`). The first `ToTable` operator has already seen a commit, the second not yet. Therefore, `LastCTS` still holds the previous version timestamp, and also `ReadCTS` of the ad-hoc query keeps its first seen version.

although the latter is precluded in our approach. As such, it is a modified, lightweight version of the Two-Phase Commit (2PC) protocol [LS79], which thus relies on proven concepts without adding overhead in our case. The setting of the status flag corresponds to the collection of votes, and the coordinator's calling of the commit or abort functions is then the distribution of the decision. Eventually, when all local commits are done, the `LastCTS` field for each state group is atomically updated to the transaction's ID. It replaces the state-local setting of the `lastCommitID` as listed in Algorithm 8 at line 14.

The consistency for reading operations is ensured by using the `LastCTS` field to determine the visibility of versions (Algorithm 6 at line 13). Thus, reading transactions initially look at the state groups to see which states are written atomically. For these states, the version must be identical. If it is not, a newer commit has been performed meanwhile. Therefore, the version timestamp used for the first read per state group is noted in the context (`ReadCTS`). Every subsequent read operation of this transaction thus sees the same snapshot, and simultaneous commits are not a problem. If a transaction reads states from multiple state groups resulting in different commit versions (`LastCTS`s), the older version must be read to ensure consistency.

Let us revisit the example in Figure 5.5 to make this process a bit more illustrative. Assuming that a transaction with timestamp 1 was the last successfully committed transaction, the `LastCTS` of the state group must also be 1. Furthermore, the first `TO_TABLE` operator has already seen a commit from transaction 3, but the second one has only seen the last write operation so far. Correspondingly the status fields for this active transaction are `Commit` for `S0` and `Active` for `S1`. Up to this point, all changes are still volatile and not visible. Therefore, the ad-hoc query would still read version 1 at this or a previous time since `LastCTS` has not been updated yet. It would still be the case even if the second `TO_TABLE` operator has seen the commit, but the persistent changes and the final `LastCTS` update have not been incorporated yet. Transaction 2 thus starts with the first read logically before transaction 3 or after it and sets the `ReadCTS` accordingly. In Section 5.8, we discuss more specific concurrency and failure scenarios in detail and how our protocols deal with them.

To sum up, the algorithms outlined above would have to be extended as follows to support cross-state consistency.

**Read:** Compare and set the `ReadCTS` based on the accessed state group instead of the accessed state (line 13).

**Write:** Nothing changes here.

**Commit:**
(1) Delay commits and set the status flag instead to `Commit`.
(2) Once the flag is set for all accessed states of the transaction, execute the local commits and eventually update the global `LastCTS` for the corresponding state group (line 14).

**Abort:** Change the status flag to `Abort` to inform other table operators that this transaction may not be committed.

## 5.4 Persistent Memory Adaptions

The before-considered protocols are rather generic and also work when using block devices. In the following, we discuss specific optimizations that are applicable when using PMem. In summary, we see three possible improvements:

1. The use of in-place updates instead of out-of-place updates
2. The avoidance of latches/locks by employing atomics
3. The manual placement of flush instructions and barriers instead of logs or shadowing

In the base variant, a commit including updates leads to the loading and overwriting of complete `MVCC Objects`. Now, we only retrieve a reference to the PMem location and – due to the byte-addressability – only update the necessary data in-place. The bitmaps and timestamp fields are 8-byte aligned and accessed atomically. The simultaneous observance of the order eliminates the need for latches. Depending on the number of cache lines an `MVCC Object` occupies, we achieve a reduction of the number of written bytes and increase the cache-hit ratio. That is especially beneficial for a higher number of version slots. However, for this optimization to be feasible, the underlying `Base Table` must either return the tuples as a reference or provide a corresponding in-place update method. In addition to performance and concurrency, we must also guarantee failure safety. Therefore, we manually flush all changed data in the `MVCC Objects` using the clwb instruction. As mentioned before, we also have to ensure the order of the atomic stores and flushes. For that, the bitmap is updated at the end for each object using an *sfence* instruction. This approach eliminates the requirement for the underlying table to perform individual updates failure-atomically since we now take care of this ourselves. Algorithm 9 shows the complete procedure for adding a new entry within an object.

**Algorithm 9** newEntry(txnID, iPos, dPos, newValue)

```
1:  /// Limit validity interval of current version (dPos) and set it for new entry (iPos)
2:  headers[dPos].dts.store(txnID);
3:  headers[iPos].cts.store(txnID);
4:  headers[iPos].dts.store(INF);

5:  /// Copy/move new value
6:  values[iPos] ← newValue

7:  /// Flush all changes and set barrier to ensure everything is flushed
8:  clwb(&headers);
9:  clwb(&values[iPos]);
10: sfence();

11: /// Activate slot at new position and flush it
12: usedSlots.store(usedSlots|(1 << iPos))
13: clwb(&usedSlots);
```

As with the universal variant, the global visibility of the entire transaction is controlled by the final write of the `LastCTS` field. In the PMem case, we again use atomic writing followed by a clwb instruction and enclosing barriers. Algorithm 10 outlines this step.

**Algorithm 10** setLastCTS(groupID, txnID)

```
1: /// Barrier to ensure all previous changes are flushed
2: sfence();

3: /// Update and flush LastCTS field and set barrier to ensure persistence of whole transaction
4: stateGroups[groupID].lastCTS.store(txnID);
5: clwb(&stateGroups[groupID].lastCTS);
6: sfence();
```

## 5.5 Query and State Recovery

The introduction of PMem also changes the recovery process in the case of failure. Here, we consider two aspects. These are, on the one hand, the recovery of the states and, on the other hand, the resumption of the query pipelines. In order to be ACID compliant, the effects of the committed transactions must still be present in their entirety, and all uncommitted transactions must not be partially visible. The necessary recovery steps can be summarized as follows.

1. Recovery of the shared state context
2. Recovery of each state
3. Updating the volatile pointers (dereferencing persistent pointers)
4. Recreation of the query pipelines

Since the state context and states are each stored in a separate file, they must first be reopened. We use the PMDK pool feature for handling these files. In addition to the actual opening of the files, other auxiliary structures are initialized here by PMDK, and its logs are traversed. These logs are written when transactions are used, but we only need them for the initial allocation. Therefore, no undo steps are necessary for our protocol during recovery. Furthermore, the volatile parts of the states and context (cf. Figure 5.4) must be recreated, such as the array of active transactions. As already described in earlier chapters, dereferencing persistent pointers is quite expensive. That is why we dereference them only once during start or recovery and subsequently use the current virtual addresses. It is currently done for the linking from the state context to the states and vice versa. As a final step, the continuous query pipelines are rebuilt and linked to the current state addresses. Important to note is that this does not yet guarantee exactly-once processing. For that, the states would further need to be synchronized with the sources to avoid duplicates or loss of transactions. Although it is out of the scope of our work, it could be solved by replaceable messaging systems [ABD+12, KNR+11] or with persistent queues in front of all sources and sinks.

We use an atomic counter to allot unique transaction numbers. Given that this variable is accessed very frequently, we decided to keep it in DRAM. In order to assign strictly ascending numbers, the variable is set to the current timestamp (in nanoseconds) at each restart and then incremented during operation using only *fetch-add*. Since a single action on a state typically takes several hundred nanoseconds, this accuracy is adequate. As a result, we can maintain logic and consistency also across restarts. We achieve atomicity with the `LastCTS` field per state group as was described above. The last ACID property, namely durability, is also given

because all data is flushed before this field is updated. After that, the commit punctuation is forwarded and could be fed back to the client. So it can be said that once a commit arrives in the sink, all changes of the corresponding transaction are stored permanently and consistently.

Finally, we consider the very worst time of a system failure, which is during the commit. Here, only a part of the data may have been persisted. Once a transaction updates `LastCTS` the next time after the restart but excluding one or more of the changed tuples of the failed commit, these would become visible to the following transactions as well. In order to prevent that, all versions younger than the current `LastCTS` field must be discarded or made invisible after a restart. Since discarding requires that almost all data would have to be scanned, we propose a simpler way. Namely, when restarting, the system can declare the period from the `LastCTS` to the current timestamp as an invalid range. However, readers must then additionally check during version determination whether the CTS is within an invalid range. Assuming that failures are rare, the number of these stored invalid ranges should remain manageable at all times.

## 5.6 Query Planning for Transactional Stream Processing

Instead of manually selecting appropriate algorithms or query executions, DBMSs typically use query optimizers or planners to accomplish it automatically. Therefore, in this section, we will look at possible parameters and strategies that can assist in the decision-making process for our TSP model. Since a full-featured query planner would be beyond the scope and is also not part of our objectives, we instead consider the first steps in this direction and highlight the opportunities but also limitations of different approaches. In contrast to typical query planners in a DBMS, our model additionally needs to consider the physical representation of states besides the query execution plan. This automatic selection is obviously only possible if the states are introduced by the continuous query being optimized. Therefore, from our point of view, the decisions must be made along the following three dimensions.

1. State representation: choosing the underlying data structures — what?
2. Placement: using the appropriate device(s) for the data (and algorithms) — where?
3. Algorithms: finding a suitable implementation of the operators — how?

All three dimensions interrelate closely and can hardly be considered separately. First, we want to identify the parameters that are necessary for a suitable selection, on the one hand, and which of them are accessible at all, on the other hand. We then examine possible cost model designs and how they can assess the parameters to make decisions. Eventually, we present an initial practical proposal including cost formulas to realize query planning for (transactional) stateful stream processing.

### 5.6.1 Hardware Considerations

Before we can choose suitable parameters, we will first review the characteristics of the anticipated hardware for our model. The focus is primarily on modern multi-socket/multi-core CPUs with PMem attachments. Particularly PMem implies for an optimizer that storage accesses no longer dominate the cost. That is, data access cardinalities alone may not be sufficient. Due to the asymmetries and lower endurance, also algorithms should be preferred that trade writes for more reads. In addition, the byte addressability influences the data structure and algorithm selection, as we have shown already in detail in the previous chapters. Since there is a physical upper limit regarding the clock rate of a single CPU, the trend is to integrate more and more cores to increase parallelism. The number and clock rate of these are vital for query planning, especially for the parallelization and partitioning of states and their inner processing steps.

However, our model should not be limited to PMem and general-purpose CPUs but can also be extended to other hardware. Especially since there is a shift towards heterogeneous architectures and specialized hardware for dedicated tasks or operators, query planning should recognize and exploit these. For example, GPUs can process vast amounts of independent data in parallel but have comparatively high transfer costs to and from the device. It is thus questionable whether they are suitable for continuous queries at all or whether Single Instruction, Multiple Data (SIMD) registers are sufficient for such purposes. For larger analytical tasks, however, it can be thoroughly profitable. For example, ad-hoc range queries on very large states or linear algebra operators using tensors could be accelerated by GPUs. FPGAs, on the other hand, can already be soldered onto CPU sockets and would thus not incur too high transfer costs. These could be programmed, for example, for special operators or operator pipelines that are very computationally intensive [MTA09]. Furthermore, modern high-speed networks and technologies such as Infiniband, RoCE, and RDMA, respectively, can be beneficial for distributed data management systems [Bin18]. As a central unit, many-core architectures can also be used, which offer even more parallelization options than multi-core CPUs. However, the high number of densely packed cores can lead to heavy heating, which has to be compensated by reduced logic and a lower clock rate. That, in turn, leads to poorer single-thread performance and has to be considered during query planning. In order to match the high core count in the memory area as well, the DRAM in many-core CPUs is often stacked to so-called High Bandwidth Memory (HBM), which brings another memory class into play.

After the short excursion into the modern heterogeneous hardware landscape, it becomes clear that a query optimizer can or even must take a colossal number of factors into account already at the hardware level. Therefore, this will only be considered superficially here for the time being and is limited to non-distributed stateful operators.

### 5.6.2 Cost Factors

As mentioned above, continuous query optimization should consider the physical representation of states and placement of operators in addition to the execution plan. The optimization goal can vary depending on the use case and can, for example, target a low latency, a high throughput, or energy efficiency. To avoid being overwhelmed by a plethora of parameters influencing these optimization goals, our objective is to identify the most important of them. Therefore, we start by looking at what we consider essential parameters for query planning for stateful operators.

We initially see the following three access scenarios to states where the ACID properties must be maintained.

**A** Persisting changes to a state

**B** Querying a shared state

**C** Recovering a state after a failure

The ratio and frequency of these scenarios as well as the access pattern they entail are decisive for the choice of state representation. For example, for a state that is mainly written but rarely queried, a log-like structure is more sensible than a tree structure. For few changes and frequent queries, on the other hand, a tree is probably more appropriate. Based on the size and requirements of the state and the available resources, adequate data placements can be realized. Finally, likewise depending on the available hardware, an appropriately optimized algorithm must be chosen for each operator or operator pipeline. Overall, this process spans a huge decision space, sketched as an example in Figure 5.6. Altogether, it is the task of the optimizer to find a combination of state representations, data placements, and algorithms that is as optimal as possible for the given query. As input for this, it needs the requirement profile and the available resources and matches them with the data structures and algorithms implemented in the system. Therefore, we now discuss available input parameters in more detail.

### Data-driven Parameters

As described above, a crucial factor is the access profile to states. Therefore, it can be mainly distinguished between the types of access (persisting, querying, recovery) being predominant and how often they will occur. Some of this information can be derived, for example, from the operators belonging to the states or their neighbors. From that, we can further find out whether more short ranges, long ranges, or only individual tuples are queried. In a transactional system with shared states, it is also instrumental to have an idea about the number of concurrent accesses, along with the approximate level of contention. A suitable concurrency control protocol can then be used on this basis. This parameter cannot be derived with a sole view on the continuous query being optimized since no data and, depending on the point in time, no

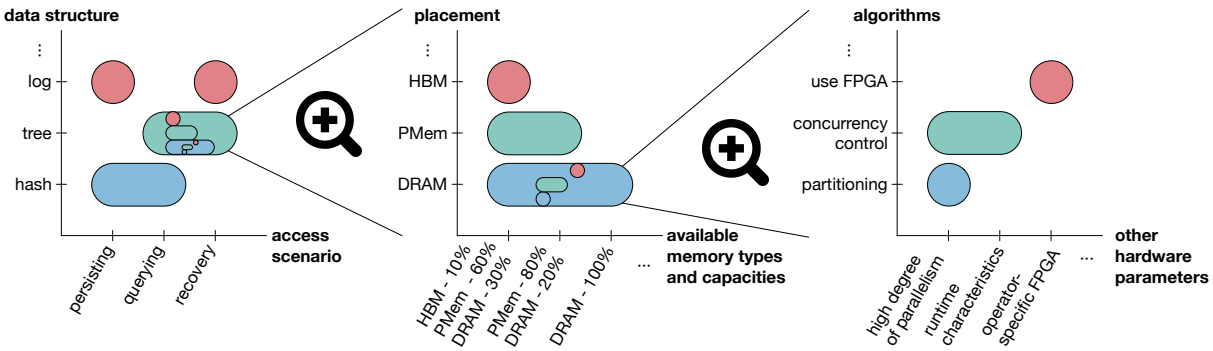



**Figure 5.6:** Sketched space of decision-making for a stateful operator.

competing queries are observable yet. Instead, it could, for example, be based on statistics or heuristics collected from previous queries. Alternatively, a modified form of sampling could be employed. It gets even more complicated when states are introduced as general-purpose tables rather than via continuous queries. Here, user input such as annotations could potentially help, but this only shifts the responsibility. Therefore, a dynamic adaptation relative to the workload progression might be a better choice. However, this also poses new challenges. For example, a change in the suitability of certain data structures, placements, or algorithms requires a potentially expensive state conversion or migration. Doing so could slow down the performance for some time, eminently while there is high demand for the state of concern. That is why robust query plans that can withstand a variety of scenarios can be used as an alternative.

#### Hardware-based Parameters

There are also several factors at the hardware level that are essential to decision making along our identified three dimensions. First of all, an initial calibration can already check which processing units (e.g., FPGA, GPU, many-core CPU), as well as memory and storage technologies (e.g., HBM, PMem, SSD, HDD), are available in the system. At the same time, available performance indicators such as capacity, latency, and bandwidth can also be determined regarding memory and storage. These indicators can then be used, e.g., to efficiently tune multi-layer data structures such as those presented in the previous chapters. The resource allocation can be justified by limited space, energy savings, performance reasons, or others. The configuration of buffers and their eviction policies also depends on these factors. Likewise, the access granularity is different, depending on the device (blocks, pages, cache lines, etc.), and should be adjusted accordingly. On the other hand, for processing units, the degree of parallelism and the clock rate are significant parameters. Accordingly, different partitioning approaches of the states or the parallelization of whole query pipelines can be implemented to exploit the potentials of the heterogeneous units. However, the transfer times to the co-processors and the merging times of the pipelines should always be included in the cost calculation (cf. [PGS17]). In addition to these basic metrics, modern processors, for example, use sophisticated techniques such as caching, prefetching, branch prediction, and reordering, which should also be incorporated into the calculation (cf. [Zeu18]).

### 5.6.3 Prototypical Cost Model

In the following, we will briefly consider how the parameters intruduced above are acquired and used by typical optimizer classes. We then outline our own strategy adapted to the TSP model.

#### Existing Models

The three classes of optimizers we discuss are the hardware-oblivious, the hardware-conscious, and the learning model. In early database systems, cost models without hardware-based parameters (i.e., hardware-oblivious) were quite widespread. Usually, these models were even application-based [LN96]. More precisely, it means that a DBMS is manually tuned according to the available hardware by profiling the most critical performance bottlenecks and

reimplementing them [WK90]. Of course, this shrinks the number of parameters and, thus, the decision space of the model. However, with each change of the hardware or software, the model must also be adjusted manually.

A model that automatically includes hardware parameters such as access latencies or cache sizes (i.e., hardware-conscious) seems more flexible and robust. Typically these parameters can be retrieved by (re)running calibration tools [Man02] whenever the hardware changes. The main difficulty of such an approach is to properly link these parameters because they often have an influence on one another. This interaction already commences at the hardware level, where, for example, a higher clock frequency yields a lower latency for memory accesses as well. Moreover, the hardware characteristics must be weighted appropriately with logical access patterns of the queries and operators.

More recently, cost models are frequently supported by machine learning too. Instead of manually creating cost formulas, the parameters are injected into already trained machine learning models. It eliminates the need for separate tuning or cost model adjustments. A representative of this class is the tool OtterTune [APGZ17], which automatically optimizes the configuration of a DBMS. In [OBGK18], the authors used deep reinforcement learning to incrementally determine the optimal query execution plan based on the properties of identified sub-queries. Although this type of optimizer sounds convenient at first, its correct configuration is crucial for success. Another problem is that machine learning approaches usually create a black box from which decisions cannot be decently explained.

#### Strategy for Transactional Stream Processing

Overall, we think it is reasonable for a transactional stream processing model to mainly decide the state representation by the data-driven parameters and the placement and algorithms by the hardware parameters. For the former, it can be mainly derived from the expected access patterns using the operator characteristics. The factors that are ultimately used can be determined either by a learning or a concrete cost model. Which of these is more suitable remains to be examined. As we have shown above, there is a multitude of parameters, and only the most influential among them should be considered so that the model does not become too complex. Furthermore, it should be taken into account that not all information is available at all times. That could make static decision-making questionable. Hence, an adaptive or progressive optimization seems to be more appropriate from our point of view.

As a practical proposal for the TSP model, we have designed cost formulas that can be used for stateful operators and seen as a supplement to the hardware-conscious model presented in [PGS17]. These rely on prior calibration of the hardware parameters. Among them, PMem has an elevated focus. Due to its asymmetry properties, we have separated accesses and costs into read- and write-based factors. These we further break down into state ($f_{<s>}$) and operator ($f_{<op>}$) dependent cost factors. The hardware parameters should be included in the former factor, which generally represents implementation-specific costs per state access. At the same time, depending on the state representation, there are different additional costs for meeting the ACID properties, such as synchronization. The hardware parameters could also be considered separately, but since they are strongly coupled with the state representations, they can be grouped together right away. In general, the hardware factors can either be included directly or through another cost formula based on, for instance, device latency or special performance counters (cf. [Zeu18]). The operator cost factors, on the other hand, are logically determined

based on typical or expected access patterns of the operator enclosing the state. For example, assume a count-based window with a specified size of 100 and a trigger interval of 10. A new arriving tuple would typically result in one persistent write for inserting the current data from a logical point of view. Every ten tuples, the whole window has to be processed, which means 100 tuples are read. Depending on the state realization, these logical operations mean more or less costs on the physical side. For instance, a ring buffer insertion in a steady-state system overwrites the oldest value and shifts the start and end offsets by one so that a logical write physically requires at least two or three writes. Combined with an expected input rate, we can then derive the most suitable settings having the lowest costs. In summary, the cost of streaming data through a stateful operator thus consists of the read and write costs for the operator-typical access that must be weighted with the state- and hardware-dependent factors. Using these factors, we obtain a cost formula for updating/persisting the state as given in Equation (5.1).

$$c_{<op>} = f_{<op>_r} \cdot f_{<s>_r} + f_{<op>_w} \cdot f_{<s>_w} \tag{5.1}$$

For the state access types querying and recovery, there are no operator-specific costs. Instead, they are more dependent on the state representation ($<s>$). Since the querying of states is similar to typical queries in a DBMS, also the cost formula can be designed analogously by using the cardinality of the state ($<s>_{size}$) and selection ($\sigma$) information. Hence, we provide the formula in Equation (5.2) for calculating the costs for querying a state.

$$c_{<s>\_q} = \sigma \cdot <s>_{size} \cdot f_{<s>_r} \tag{5.2}$$

For the recovery costs, on the other hand, it gets a little more complex. We can distinguish three cases depending on the recovery measures taken during the persistence of the state. If, for example, the state can be updated entirely atomically, no recovery actions may be necessary, and the cost is thus zero. Of course, the corresponding state must still be loaded into the user address space. However, since it is necessary and constant in any case, this part can also be omitted from the calculation. The second case is when the state is made failure-atomic using checkpoints, logs, or similar concepts. Accordingly, for example, only the changes from the last checkpoint need to be undone or redone. The worst-case could be when the entire state has to be read, and inconsistencies have to be overwritten or removed. To cover all three cases, we introduced a delta ($\Delta$) and set it off against the state size ($<s>_{size}$). It would be 0 and 1 for the first and last cases, respectively. For the second case, it can range from 0 to 1 and, thus, represents the percentage of the total size that has to be un-/redone. That yields the cost formula for recovery given in Equation (5.3).

$$c_{<s>\_rec} = \Delta \cdot <s>_{size} \cdot (f_{<s>_r} + f_{<s>_w}) \tag{5.3}$$

The given formulas are for now only theoretical considerations. Their precision still needs to be empirically investigated as part of future work. Nevertheless, we hope that they will already assist in the conceptual design for query planning in a TSP system.

## 5.7 Use Case: Event Stream Processing

Complementary to our TSP model, we now consider a special but widely used form of stateful stream processing known as the continuous processing of events. For this purpose, there are dedicated event stores, which have the challenging task of continuously handling massive temporal data streams. This data must be persisted while meeting severe query and recovery guarantees. To meet those, many of these systems make tradeoffs for a variety of optimization directions. For example, performance-oriented systems keep the majority of data in the main memory, but this entails higher monetary costs and can sometimes result in data loss in the event of a failure. On the other hand, the primary data can be placed on cheaper storage media like SSDs or HDDs, while queries can be accelerated by DRAM caches. PMem can be an opportunity to make fewer compromises and provide an optimal solution in terms of costs (economic as well as ecological), performance, and recovery guarantees. To our knowledge, there is no event store yet that leverages PMem for that. Therefore, we will consider several potential ways to use PMem in an event store in the following. Rather than considering PMem as a universal memory, we will examine it for opportunities to design a three-layer memory hierarchy consisting of DRAM, PMem, and disk. We base the discussion as a case study on the DBMS for event streams called ChronicleDB [SS17, SGKS19]. In particular, we will review three main components of ChronicleDB and examine how to achieve better insertion and query times as well as recovery guarantees. The concepts developed and lessons learned here serve as an important foundation for a comprehensive system that exploits a modern memory hierarchy. In this regard, the focus on ChronicleDB is not limiting but still allows us to apply the insights to both more general (e.g., ingestion and recovery) and more specialized considerations (e.g., temporal indexing and storage designs).[4]

### 5.7.1 ChronicleDB

ChronicleDB is a DBMS with a storage layout that supports high write and query performance for numerous event stream data [SGKS19]. It also offers fault tolerance and recovery guarantees but is not as strict as our TSP model. It can be either used as a standalone database server or integrated into an application using its library. Since ChronicleDB is targeting application scenarios that feature a large amount of continuous data within rapid time frames, it is designed around three main requirements:

**R1** Fast ingestion of high and fluctuating event rates

**R2** The possibility of stream replays and efficient time-travel operations

**R3** Fast access to events via secondary non-temporal attributes, e.g., point, range, and aggregation queries

The first requirement demands that in the case of high data rates, load shedding should be avoided. R2 and R3, on the other hand, address efficient query performance for a plethora of analytical workloads. Examples include post-mortem analysis of event stream queries, continuous processing for dashboards, or traditional OLAP demands. ChronicleDB consists out

[4] The material in this section is based on [GGK+20].

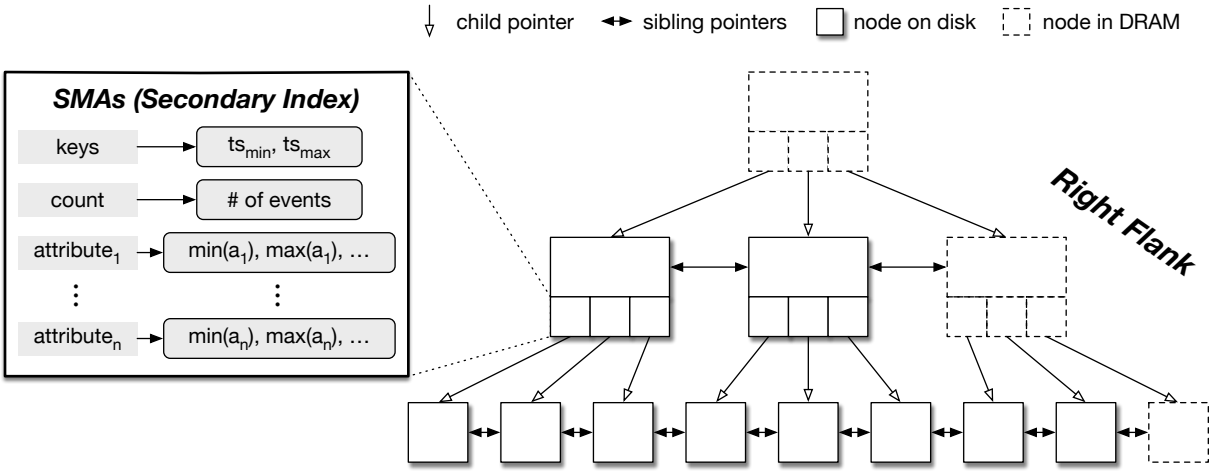



**Figure 5.7:** ChronicleDB's TAB$^+$-Tree (primary index) layout including SMAs (secondary index).

of four core components which we will summarize in the following paragraphs. After that, we will examine these components in combination with PMem and which alternative approaches arise in this context.

### Primary Index

As the primary index, ChronicleDB uses the Temporal Aggregated B$^+$-Tree (TAB$^+$-Tree), an extension to the B$^+$-Tree with event timestamps as its key domain. In more detail, let $T$ be the temporal domain and $A_i$ be the attribute domains for the attribute at the position $i$ for a stream of events $E$. An event $e$ in $E$ is then a tuple of the following form:

$$e = (a_1, ..., a_n, ts) \text{ where } a_i \in A_i \text{ and } ts \in T \tag{5.4}$$

The event stream is directly ingested into the TAB$^+$-Tree. Important to note is that the schema here is fixed per store, and timestamps in $E$ do not have to be unique. An overview of the layout is given on the right in Figure 5.7. Unlike a typical B$^+$-Tree, the sibling nodes here are doubly linked at all levels to improve query and recovery performance (R2). Since the keys are (mostly) monotonically increasing timestamps, the index builds up from left to right. This append-like approach is also found in the storage layout (see below), where the data log represents the database. Together this supports fast ingestions (R1). By default, inserts behave effectively like a continuous bulk load into a traditional B$^+$-Tree index as new events can be simply appended to the most-right leaf node. However, this default behavior takes only effect if the temporal order in the event stream $E$ is maintained. The most recent nodes on each level are referred to as the right flank of the TAB$^+$-Tree. These nodes are kept in DRAM at all times to increase the ingestion performance (R1). However, this also means that in the event of a failure, this data can be lost. In order to entirely avoid data loss, an additional log would be necessary [SS17].

### Secondary Index

As can be seen in Figure 5.7, there is the option of a secondary index in addition to the primary index. It comes in two flavors: a heavyweight and a lightweight index. The variant shown in the figure is the lightweight index which is an adaptation of SMAs [Moe98] and supports

arbitrary aggregate functions on the event's attribute domains. For example, this can be the key range, the total event count, or the attribute's minimum and maximum values. Unlike [Moe98], ChronicleDB stores the aggregates directly in the nodes of the TAB$^+$-Tree. The aggregates are associated with a child pointer to a subtree or single leaf node. That can speed up aggregate queries, range queries (by pruning), or even more complex queries like pattern matching (R3). For example, instead of reading all leaf nodes, temporal aggregation queries can directly use matching aggregates from the inner nodes to obtain answers in logarithmic time. Another benefit is that a range query on secondary attributes can directly exclude entire subtrees using the minimum and maximum aggregates. Furthermore, the interleaving of the secondary index with the primary index removes the need to persist them externally, thus avoiding random I/O while querying a temporal region. Apart from the lightweight index, heavyweight indexes, which are traditional secondary index structures such as LSM-Tree [OCGO96] or COLA [BFF+07], are also possible. Notably, leaf pages of heavyweight indexes refer to the TAB$^+$-Tree pages with a record offset.

### Storage Layout

Since event application scenarios often deal with massive amounts of data, ChronicleDB compresses the nodes of the TAB$^+$-Tree. It can reduce the storage cost significantly as, for instance, sensor data often feature similar values which can be efficiently compressed. The actually used algorithm for compression is configurable. However, this compression leads to variable-sized nodes, which, in turn, prevents a direct mapping to fixed-size block addresses as the physical position cannot be computed. Hence, an address translation layer is necessary that maps logical node IDs to physical addresses. This layer should be stored persistently so that a full scan is avoided during recovery. Separately maintaining this information would be a naïve solution because this would lead to random I/O when switching between the primary and metadata locations. Instead, the address translation is interleaved with the actual data pages to achieve a more sequential access pattern[5] For both the address translation and data nodes, blocks of the same fixed size – a multiple of the uncompressed index node size – are used. In previous work [SGKS19], it corresponded to 32 KiB blocks based on 8 KiB index nodes. A block can contain either compressed TAB$^+$-Tree nodes or the address translation information for such nodes. If a block contains the data nodes, it is referred to as *MacroBlock*. This block stores the number of TAB$^+$-Tree nodes it contains and their respective compressed size besides the actual data. Therefore, accessing a node of the index requires the physical address of the *MacroBlock* plus its offset within this block. This composition fits into a single 8-byte unsigned integer number where 6 bytes are used for the physical address and 2 bytes for the offset. On the other hand, if a block contains the address translation information, it is referred to as *Address Translation Block*. The entries in these blocks are organized as a global tree structure called Address Translation Tree (ATT). Since the IDs of TAB$^+$-Tree nodes are consecutive numbers starting at zero, the ATT only has to store the 8-byte translation information while the ID is used as index position. Therefore, each leaf of the ATT contains the same number of addresses comprising a contiguous range of index node IDs. Also, the inner nodes cover a fixed number of node IDs pointing to the corresponding block of the next level. Similar to the right flank of the TAB$^+$-Tree, the most recent translations are kept in DRAM and are written to disk once a block gets full to retain the ingestion performance high. Consequently, these volatile

[5] This aspect is less critical for SSDs than for HDDs, but there is still a remarkable difference compared to random access patterns [SAJA09]. Also for PMem, as seen in Section 2.5, this performance mismatch is still present.

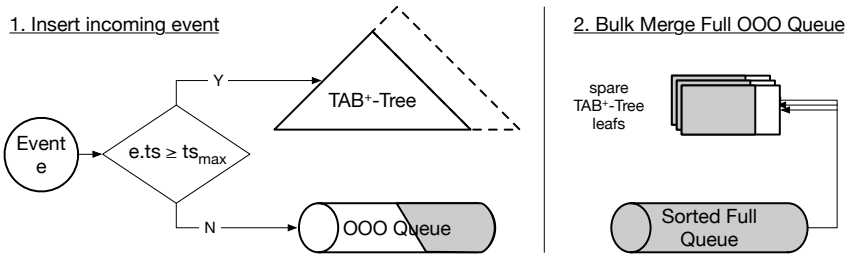



**Figure 5.8:** Handling of OOO events in ChronicleDB.

translations must be recovered in the event of a restart. Within the ATT nodes, back-references to the predecessor on the same level and the predecessor of the parent are introduced to speed up the recovery process. Thus, it can be performed from the end of the database to its start. Expressed that in numbers, recovery only needs the number of translations stored per block (i.e., fan-out) times the height of the ATT ($h$) read operations ($fan\text{-}out * h$).

#### Out-Of-Order Data

Above, it was assumed that new events typically arrive in a temporal order resulting mainly in an append-only process. However, if this order is not maintained, the performance of the TAB$^+$-Tree will degenerate to that typical of a B$^+$-Tree. Events that violate this temporal order are referred to as OOO data. Naturally, this issue can only occur if the temporal domain is based on application time and is irrelevant using system time. To tackle this issue, ChronicleDB adapts a delta-like strategy, as shown in Figure 5.8. If the timestamp of a new event is lower than the maximum seen in the system, then it is classified as OOO data and put into a dedicated OOO queue. That prevents the append-only character from being disrupted. Once the OOO queue is full, it gets bulk merged into the TAB$^+$-Tree. The queue size and, thus, the frequency of merges is tuneable and helps stabilize performance depending on the underlying hardware. Furthermore, to avoid splits or even split cascades during merging, the leaf nodes can leave spare space for possible OOO events. Especially for spinning disks, that is very beneficial since a sequential physical node layout is preserved.

In the following subsections, we discuss how ChronicleDB can be adapted to exploit PMem. This discussion is broken down into the three components TAB$^+$-Tree, the physical storage layout, and OOO handling.

### 5.7.2 TAB$^+$-Tree Adaption for PMem

ChronicleDB strives to efficiently handle massive event streams fulfilling both durability and performance requirements. In order to achieve the balance between these requirements, the architecture of the primary index makes use of a mixture of DRAM and secondary storage as explained above and shown in Figure 5.9 A.a. Now, with PMem, whose properties settle between memory and storage, there are new opportunities to rethink the design of the primary index. We present three approaches for adapting the TAB$^+$-Tree that take advantage of PMem's byte-addressability and direct persistence. The first option is to put the right flank on PMem instead of DRAM (Figure 5.9 A.b). It results in much better recovery guarantees and nearly eliminates the recovery efforts of the right flank in case of failure. The second approach keeps

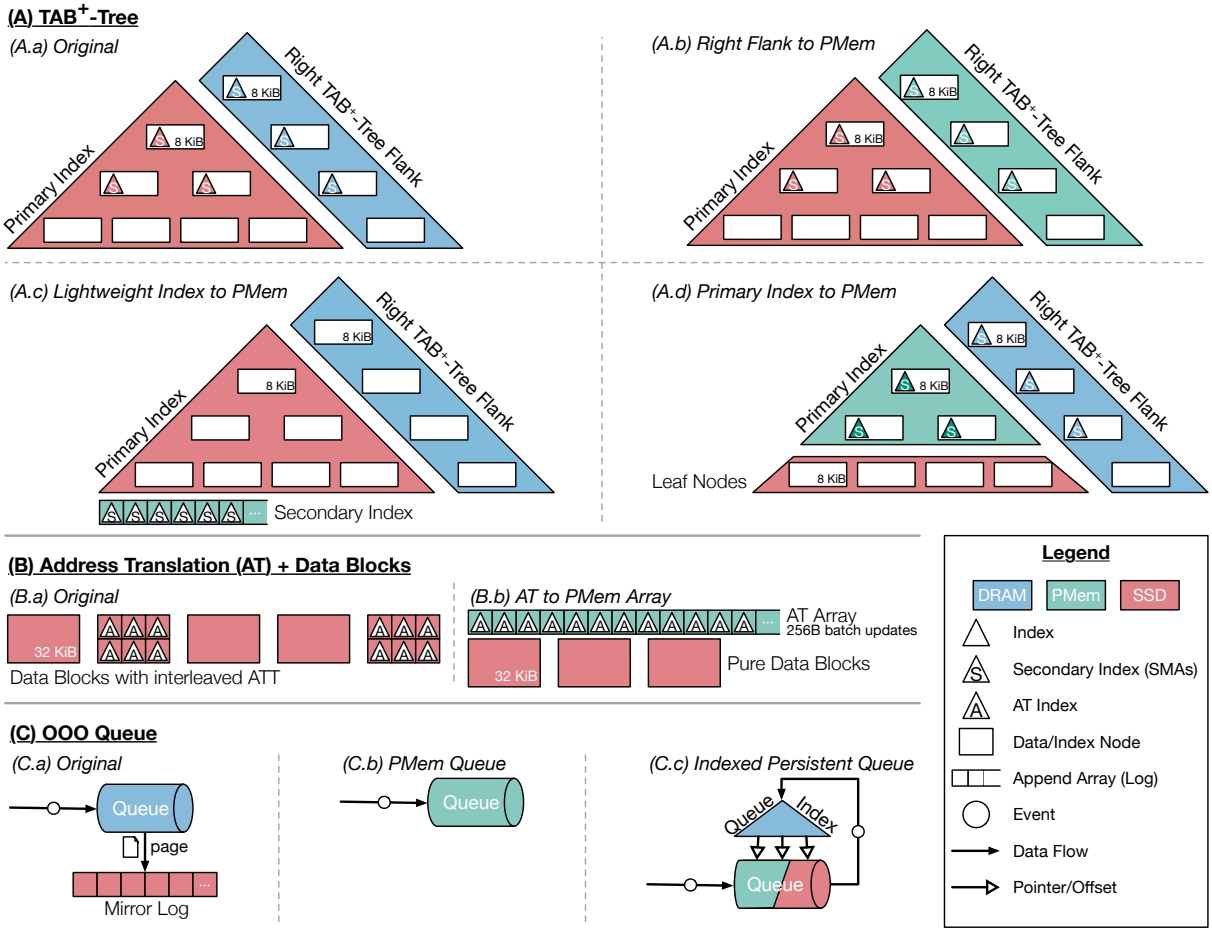



**Figure 5.9:** Overview of PMem-based approaches applied to ChronicleDB.

the flank in DRAM but moves all lightweight index information of all inner nodes to PMem (Figure 5.9 A.c). While keeping the node size the same, it allows a higher fan-out of the nodes, which, in turn, reduces the tree height and potentially access times. The final idea is to move all inner nodes located on disks to PMem (Figure 5.9 A.d). Thus, index navigations and aggregate queries never have to touch secondary storage. Below, we take a closer look at these three approaches and what changes are needed.

### Right Flank

As a measure to balance performance and durability guarantees the right flank of TAB$^+$-Tree is kept in DRAM. Consequently, all entries of the most recent leaf node are not backed in case of a crash. Only the inner nodes can be rebuilt, which, in turn, consumes recovery time. Thus, moving the right flank from DRAM to PMem (Figure 5.9 A.b), we get both better recovery guarantees and speed since the most recent inserts are always stored persistently. The evident drawback will become apparent during insertion performance, which we will quantify in the evaluation section (Section 5.8.2). First, though, we look at how this approach can be implemented in practice.

The pages are already organized as byte arrays and can be moved between memory and storage devices without serialization effort. Therefore, also each incoming event needs to be converted

into its binary representation before appending it to the most recent leaf. Once a leaf or inner node is full, it is written to disk, and a new empty page is allocated. By eventually adapting the parent pointers, this process is managed atomically. However, if the right flank is always persistent in PMem, it must be ensured that the nodes that are not yet full are always in a consistent state – especially after a crash. Therefore, we enforce periodic flushing of altered memory regions as described in the following.

Besides the actual data, each page also has a header region. Within, sibling and parent information as well as the number of events – to determine the valid data region – are maintained. In order to reuse an allocated in-memory or PMem page after it has been written to disk, only this header information needs to be reset and flushed. Hence, to maintain consistent states, first the new event data and then the counter in the header are flushed to PMem. However, single events are generally relatively small, and calling flush instructions for every append operation could lead to drastic performance degradations. To counter this, we propose to flush events in configurable batches, where a sequential access pattern can be applied to the written data, and the counter only has to be flushed once. In Section 5.8.2, we examine this batch size and its tradeoffs.

### Aggregates

As described above, each inner node of the TAB$^+$-Tree maintains a configurable set of aggregates for each child reference. They summarize the data that lies ahead in the corresponding subtree or leaf node and can be used to speed up filter and aggregation queries. However, the aggregates reduce the possible fan-out of the inner nodes, e.g., assuming a page size of 8 KiB as above, a node can reference 459 and 43 children without and including lightweight indexing (storing sum, min, and max for 6$\times$ 64-bit floating-point attributes), respectively. To mitigate this disadvantage, we moved the aggregates to PMem, as shown in Figure 5.9 A.c.

The aggregates are calculated from the bottom to the top of the tree once a node is full and needs to be written to disk. Then the aggregates are computed from the data of this node (either pure events or also aggregates) and propagated upwards, where the parent node attaches this information to the corresponding child reference. However, this propagation step only works if the aggregate functions are decomposable (cf. [THSW15]). By offloading the aggregates to PMem, the original locality advantages of the lightweight index are more or less nullified. Nevertheless, in order to receive benefits besides point and filter queries, the aggregates must be stored and accessible as efficiently as possible. Hence, we implemented them as a flat array on PMem, where each slot contains the aggregated values of a single node in the TAB$^+$-Tree. Since the aggregates have a fixed size and node IDs are consecutive numbers starting at 0, we can use those IDs to directly retrieve the offset of a slot in the array. We set the capacity of a slot to be a multiple of 64 bytes to improve cache efficiency. Therefore, the offset can be calculated as follows. Let $S$ be the size of the aggregated attributes in bytes. Then, the byte offset $o_i$ of the aggregates of node $i$ within the array results in: $o_i = i \cdot \left\lceil \frac{S}{64} \right\rceil \cdot 64$.

Overall, we thus get the benefit of a reduced tree height and, thus, faster queries based on the time domain. However, aggregation queries and inserts (during migration) must now always access two mediums (DRAM or disk and PMem) at a time. The extent to which these two aspects affect performance is discussed in Section 5.8.2.

**Index Nodes**

Our next approach that uses all three storage layers stores only the leaf level on disk and manages the inner nodes on PMem, as illustrated in Figure 5.9 A.d. The lightweight index information is stored here again interleaved with the primary index nodes. With that, both index navigation and aggregation queries can be handled without touching the disk. The PMem part is again managed as a flat array, where one slot matches the configured page size (8 KiB in our case) and, thus, one inner node. The flat array structure allows for a sequential write pattern improving write throughput, and it only requires minimal modifications of the insert mechanism. This time, we cannot reuse the node IDs to calculate the offset as we have to manage two independent storage locations (disk and PMem). Thus, we require two ID sequences, one for the leaf and one for the inner nodes. For this purpose, one bit is reserved to determine which kind of node an ID references.

A page in PMem requires 20 bytes for the header and 16 bytes (key and child reference) plus the aggregate size per entry. Thus, this approach requires additional space on PMem but is still more space-efficient than storing aggregates only. That is due to the 64-byte alignment of the aggregate-only variant. In contrast, this approach avoids the padding of a single aggregate set and only pads the node size (8 KiB). With that, it is losing at most the size of a single index entry (i.e., aggregate size + 16 bytes). Furthermore, traversing inner nodes prevents access to secondary storage entirely now, which should result in better query performance compared to the previous approach.

### 5.7.3 Storage Layout Simplifications through PMem

Next, we consider the storage layout and the corresponding address translation of ChronicleDB. The interleaving of data and tree-based translation pages serves the purpose of achieving a good balance between insert, query, and recovery performance. Performance degradation occurs when the nodes of the TAB$^+$-Tree and also their translation are not part of the right flank (i.e., not in DRAM). Let us assume, for instance, an ATT of height three and a block size of 32 KiB. Even though the root is in DRAM, two random reads with a granularity of 32 KiB are still necessary to determine the node's address.

Hence, it seems reasonable to move the address translation to PMem. That dissolves the interleaving with the data, which could potentially degrade insert performance as a tradeoff for lookup and recovery performance. Once again, we decided to organize the data on PMem in the form of a flat byte array. It replaces the tree-based approach (ATT) with a simple lookup table that is updated as pages are written. The slot at position $i$ in this table comprises the translation information for the corresponding node ID $i$. This approach is visualized in comparison to the original in Figure 5.9 B.

Due to the omission of the right edge of the ATT, i.e., the DRAM component, combined with direct flushing of the translation data after each update, the recovery effort is virtually eliminated. However, a single entry in this array is only 8 bytes, and individually flushing each of those would drastically impact event ingestion. Therefore, we take a similar approach as with the TAB$^+$-Tree and batch multiple updates together. We follow the block size of the DCPMMs and flush in 256-byte granularity. It corresponds to 32 translations that would potentially have to be restored at most, which is much less than the right flank of the original ATT approach.

Even though the average insert performance is likely to deteriorate, the worst-case performance improves significantly. That is the case when the nodes of the entire right flank are full in the ATT and, thus, another insert triggers the persisting of everything to secondary storage. Depending on the height of the tree $h$, $h * 32$ KiB must be written. With the flat structure in PMem, however, the performance is constant (i.e., best case $\equiv$ average case $\equiv$ worst case).

### 5.7.4 Out-of-Order Handling with PMem

The last component of ChronicleDB that we consider is the OOO queue and strategy. As with the TAB$^+$-Tree and ATT, originally, a portion is held in fast DRAM, and a second portion is stored on a persistent disk. That retained the balance between query performance, recovery guarantees, and insertion performance. The queue itself is kept in DRAM, and each full page is backed by a mirror append-only log on disk (see Figure 5.9 C.a). Depending on the frequency of OOO events and the size of the queue, this dual management can quickly drain DRAM reserves. Furthermore, events within the current incomplete page can be lost. Alternatively, mirroring each event directly during insert would cause the original problem that the queue was supposed to solve (write amplification). Therefore, there is an inevitable tradeoff in query, recovery, and insert performance. Below, we will explore how PMem can be used for OOO handling to obtain better compromises. While the first approach is a pure PMem solution, the second approach represents a hybrid layout.

#### Persistent Memory Queues

Among the factors responsible for the necessary tradeoffs are the inherent lack of persistence in DRAM and insufficient access granularity to disk. With PMem, both issues can be satisfied simultaneously. Therefore, our first proposal is to move the queue completely to PMem without the need for a mirror log (see Figure 5.9 C.b). To avoid additional write operations and to utilize sequential writes, events are only appended in an unsorted manner. New events are directly flushed after insertion in the queue to eliminate data loss in case of a system failure. In addition to the strong recovery guarantees, this approach saves half of the original space needed for OOO handling.

#### Indexing

The placement of the queue in DRAM in the original implementation allowed the direct access of individual events avoiding expensive fetching of entire pages from disk. In addition, the DRAM copy is sorted by application time and, thus, enables efficient query processing alongside the time domain and fast merging into the TAB$^+$-Tree. With PMem's byte addressability, the first-mentioned feature can also be fulfilled. However, to also enable fast querying and merging, the queue must be ordered. Since a direct ordering would cause additional random writes, we propose a lightweight in-memory index using application time as the key domain (see Figure 5.9 C.c). The values of the index then constitute the offsets in the PMem queue. It uses significantly less DRAM than the baseline implementation. A disadvantage could be the random access to the events in PMem, although this is much better than with SSDs or HDDs. There is also further potential for more sophisticated merging strategies, such as only merging ranges of a designated size at a time.

## 5.8 Evaluation

Within this section, we want to put the stateful processing of data streams using PMem to a practical test. We start with our TSP model and corresponding protocols, which we have prototypically realized within the stream processing engine PipeFabric. Subsequently, we will further have a detailed look at PMem's performance impact in the specific case of event stream processing.

### 5.8.1 Transactional Stream Processing

In this subsection, we present the results of our experiments as well as a discussion about the suitability of our TSP approaches. With the evaluation of our protocols - which we presented in Section 5.3 - we strive to show that they are the most scalable and resilient choice for TSP. The following hypotheses should be proved in detail:

**H1** The isolation requirements of an application will be correctly fulfilled in a multi-threaded environment. It means that, on the one hand, our concurrency and consistency protocols always keep the states in a correct condition, and, on the other hand, ad-hoc queries always obtain consistent results.

**H2** The overhead of the transaction management is negligible.

**H3** Our MVCC approach is the most scalable and resilient protocol compared to single-version concurrency control strategies in a single-writer scenario.

**H4** The optimization of atomic in-place updates with fine granular flushing to PMem performs better than the disk-based implementation that overwrites complete entries and uses latches as well as undo logging.

**H5** Recovery and application start with PMem happen near-instantaneous.

We compare our MVCC approach against a simple Strict Two-Phase Locking (S2PL) [EGLT76] and a Backward-oriented Optimistic Concurrency Control (BOCC) [Här84] protocol. Both are representatives of typically used classes of concurrency control protocols, namely lock-based and optimistic approaches. In particular, lock-based protocols have been used frequently in related work (see Section 5.2), for which we want to show that they are not suitable for TSP and even less for PMem. We opted for BOCC since the Forward-oriented Optimistic Concurrency Control (FOCC) would completely block writer streams in case of high contention. Thus, it is not applicable for our target system and application purpose. Along these lines, we have chosen a representative for both classes that favors the writing stream. Below, we describe the realization of the competitor protocols in more detail, for which we have made every effort to implement optimized versions, e.g., by profiling.

#### Competitor 1: Strict Two-Phase Locking

With the S2PL protocol, all requested resources (key-value pairs) are locked at the time of request and only released after a transaction is complete. We use a modified approach of wait/die [RSI78]

for deadlock avoidance, with the difference that wait and abort/restart decisions are not based on the age of the transactions but the type. Particularly, read-only transactions are restarted, while transactions containing write operations are waiting if a conflict is detected. This way, we do not need to keep before-images and in general read-only transactions are faster to restart. Furthermore, we have already stated that we intend to prioritize the writer of a state for the transactional processing of data streams. Because of a missing log, the protocol does not yet guarantee failure-atomicity. However, for the sake of simplicity, we have omitted that here. Although in practice it would require additional logging, our initial results have shown that this protocol in its current lightweight form is already inadequate.

#### Competitor 2: Backward-oriented Optimistic Concurrency Control

Just as with the S2PL protocol, if there is a conflict using the BOCC protocol, we only restart reading transactions, as they generally are cleaned up or executed more quickly. As usual for optimistic techniques, we require the read and write set of each transaction. The backward validation proceeds in such a way that a transaction under validation checks its own read set against all write sets that have been committed for conflicts. For performance reasons, only individual writes are isolated during the commit phase. Therefore, even currently active write transactions must be included in the validation check. We have ensured failure-atomicity by wrapping the persisting of the changes during the commit phase with a PMDK transaction. However, a deficiency still exists since the atomicity is only applicable to one state. That is because each state is stored in a separate pool, and a PMDK transaction can only refer to a single one. For conflicts to occur at all, there must be an overlap between the transaction stages. For the purpose of verifying this overlap, we introduce three logical timestamps: the beginning time of a new transaction (TSB), the moment of starting the validation (TSV), and when a transaction is completed (TSC). These timestamps can be used to determine an overlap, which is true if:

**A** the beginning timestamp of the transaction in validation is older than the completion timestamp of the transaction being compared to, and

**B** the start of its own validation followed the validation start of the other transaction.

In formal terms, a reading ($t_r$) and a writing transaction ($t_w$) overlap if and only if:

$$TSB_r < TSC_w \wedge TSV_r > TSV_w \tag{5.5}$$

If the necessary condition of temporal overlap holds, the sufficient condition is checked next, i.e., the intersection of the read and write sets. Provided that one of the conditions is false regarding all committed transactions, the transaction under validation may be committed. The collection of committed write sets is implemented as a deque synchronized with lightweight shared locks. In order not to let this collection grow infinitely, it is periodically purged based on the oldest begin timestamp present among the active transactions.

Analogous to the MVCC protocol, we have also implemented the S2PL and BOCC protocol in a PMem-optimized manner to perform updates in-place. Likewise, essentially the same consistency protocol, as described in Section 5.3.3, is used for handling multiple states. More precisely, it means that transactions are not committed until all stateful operators have received

and registered the corresponding commit punctuation. A difference to the MVCC variant is, however, that conflict-induced aborts can occur. In this case, the transaction is restarted after a penalty time based on the average transaction length. We examined several sleep times in the range of several hundred nanoseconds per operation since it matches the typical access times to PMem. The throughput was, on average, the same up to a specific threshold. Only the rate for repeated aborts was higher if the penalty time was chosen too short. In the end, a delay of 500 nanoseconds per operation times the average transaction length proved to be the best compromise.

### Workloads

As a base table, we used a simple and completely persistent $B^+$-Tree. Naturally, any other state representation, which we described in Chapters 3 and 4, would be possible but would ultimately only move the baseline performance in our comparison. The $B^+$-Tree implementation, however, is the most mature structure available for us without structural changes during updates. Since the values are only modified in-place, we can be sure to measure only the read/write performance plus concurrency control overhead. We set the node size to 1024 bytes since we have seen in Section 3.4 that it is the most efficient size. Only for our MVCC approach with two versions, we doubled the node size to 2048 bytes to have around the same entries per node as the competitors. In a first experiment, we could confirm that this is optimal for all protocols. As a benchmark, we used an extended variant of the scenario in Figure 5.5, having one stream continuously writing to two states and multiple readers querying these states. Both are initialized with a table size of one million key-value pairs, each with an integer key as well as a tuple with three integers and one double attribute (8-byte keys, 32-byte values). In the benchmark, we vary the number of parallel readers (threads), the length of a transaction, and the contention rate. The contention rate is controlled by the parameter $\theta$ according to a *Zipfian* distribution [GSE+94]. A value of zero corresponds to an equal distribution. Increasing the $\theta$ also increases the probability of conflict. Our highest setting is 2.9, which causes the same key to be accessed 82% of the time. The length of a transaction is varied from 2 (short) to 10 (medium) and up to 100 (long) operations. The number of parallel readers was capped by the number of available cores. We measure the total throughput in terms of transactions per second (tps) and the error rate as the ratio of the number of restarts to the total number of completed transactions.

### Correct Handling of Concurrency and Failure Scenarios

The correctness of the protocols has already been proven extensively in the literature, e.g., in [WV02, BHG87], so we will not give formal proof here. Our implementation and modifications do not change anything about the reasoning. For example, for the S2PL, only the deadlock avoidance mechanism has been changed. In the case of the BOCC protocol, our variant is even less sensitive since there is only one writer and no write-write conflicts can occur. Our MVCC approach is most similar to the read-only multi-version protocol in [WV02], whose proof is based on an acyclic serialization graph. The atomicity property of the lightweight 2PC protocol, on the other hand, can be demonstrated using state charts.

Accordingly, we look at some critical concurrency and failure scenarios instead and how our approach deals with them. A first typical scenario would be one or several write operations between two read operations of another transaction (e.g., $r_1(x)$, $w_2(x)$, $w_2(y)$, $r_1(y)$). A correct

execution regarding snapshot isolation requires that the second read operation does not see the write operation(s). Since, in our case, writes are initially stored in a separate area (`Uncommitted Write Set`), that is always given. More dramatically, a commit of changes could occur instead of individual update operations (e.g., $r_1(x)$, $c_2(x,y)$, $r_1(y)$). If that is visible to the enclosing transaction, it violates the consistency condition in the case of snapshot isolation. Precisely for this situation, the reading transaction remembers the timestamp of the version initially read (`ReadCTS`) and uses it to access the following objects. It means that all commits in the meantime are not visible. To push the whole thing to its extremes, it could happen that when accessing multiple states, a commit was already completed in the course of the 2PC protocol for one state, and the other is still at it. Again, this poses no problem because the results are only visible once the commit was successful for all states, which is done by the final atomic writing of `LastCTS`.

Next, we consider how our approach responds to aborts and failures. In normal operation, there can logically be user-triggered aborts. As soon as such punctuation arrives in a table operator, it becomes apparent to all other operators of this query and causes the write sets to be deleted. Even a crash-caused abort – prior to the commit punctuation being visible to all table operators – leaves no trace in the persistent part. The most critical case would be a system failure during a commit. The specific handling here, as already discussed in Section 5.5, is to introduce invalidity periods.

Apart from the theoretical consideration, we also verified the correctness of concurrent transactions by checking the validity of the results. For that, we used the workload as mentioned above. More precisely, we checked that each key looked up always provides the same version and value per transaction. This check was performed both locally per state (≡ repeatable read isolation) and across states (≡ snapshot isolation). In order to ensure that all performance measures are based on correct systems behavior, we varied the number of concurrent ad-hoc queries and the contention rate in the same way as for the subsequent performance evaluation. Finally, we also randomly crashed the application to confirm that no inconsistencies can be provoked. All in all, this provides more than enough indications of the correctness of our protocols and recovery process. Therefore, hypothesis H1 can be confirmed.

### Version Impact

Initially, we will examine the impact of the number of versions in the MVCC approach. Figure 5.10 shows the results for medium-sized transactions (i. e., ten operations per transaction) and 20 ad-hoc reader queries issued from the same socket. As can be seen, with our in-place variant, the number of possible versions hardly influences the performance. However, the slight differences are caused by two opposing factors. On the one hand, there is the increasing cache-hit probability with higher contention. On the other hand, with increasing contention, versions must also be cleaned up more often by the garbage collection, which happens more frequently with fewer version slots. Therefore, the throughput with more versions is marginally higher at the end. A disadvantage for a higher version count is that more version headers may need to be checked during reading. For the traditional out-of-place implementation, the performance also increases initially due to the better cache-hit rate. However, this decreases again due to the used latches at a certain contention level. Furthermore, it can be seen that it is best to keep the number of versions to a minimum since the entire MVCC object is always overwritten. Compared to disks, however, the impact here is not too high because the used instructions

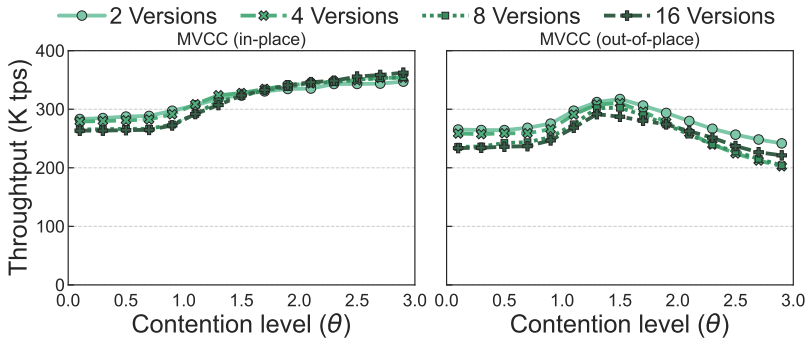



**Figure 5.10:** Effect of the number of versions with 20 readers and medium-sized transactions.

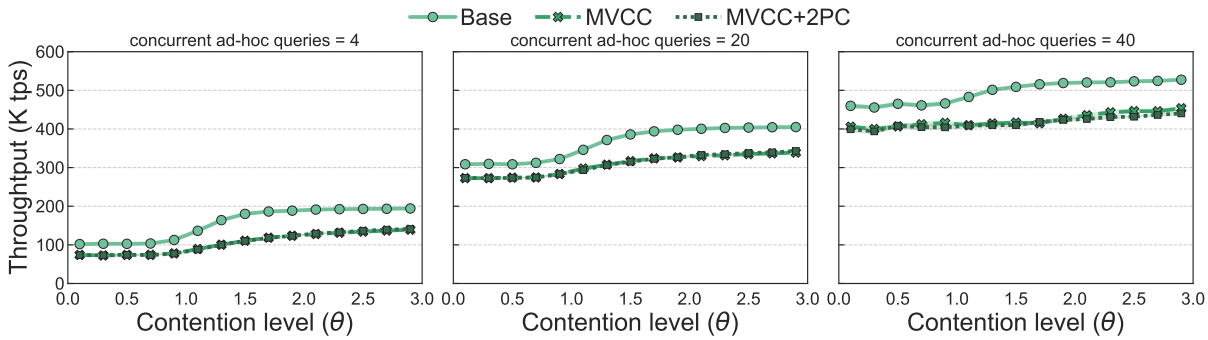



**Figure 5.11:** Transaction management overhead with medium-sized transactions.

only flush cache lines that have been changed at all. Concluding from this measurement, we will continue with two versions in the following experiments, as they give better results on average and have the lowest PMem footprint.

### Protocol Overhead

Next, we examine the overhead of our MVCC approach and the consistency protocol. The results are presented in Figure 5.11 relative to the performance of the underlying base table (persistent $B^+$-Tree) without versioning and consistency assurance. It means for the baseline that there is no transaction management per se, i.e., transaction begin/commit do nothing and read/write are executed directly. As can be seen, the consistency protocol has no noticeable overhead. Principally, as described in Section 5.3.3, only the commit timestamp is written/read globally and commits are executed together once all stateful operators have seen the punctuation, which effectively only changes the order of operations. However, compared with the base table, there is a striking difference, which we had estimated to be smaller. If we take a closer look and compare the read and write throughput separately, we see that the reading performance hardly decreases (about 5%). The write performance, though, worsens by up to 50% because flushing is forced here and more data is written. Since there is only one writer per state, this is less pronounced in the total throughput with a high number of readers. Overall, we observe an overhead of 10-40% for the entire transaction management, depending on the reader count. So we can conclude that especially the explicit cache-line flushing needs a lot of extra time. Nevertheless, it is necessary for any form of transaction management – during commit at the latest – to guarantee the durability property. Thus, we can state that at least our consistency protocol and the CC protocol for readers have a negligible overhead (H2).

### Concurrency Control with Medium-sized Transactions

As a first experiment regarding concurrency control, we compare the performance of all protocols using transactions of medium length. The throughput is shown in Figure 5.12 as a function of the contention level. We also varied the number of parallel read-only transactions. While the first two columns are the results for running and allocating on the same NUMA node, the third column utilizes both sockets. Only the PMem pools are always bound to the same socket since cross-socket variants are currently not possible.

Similar to our previous measurements on SSD [GS19], the MVCC protocol also excels from the others in the PMem case. Most of the performance impact for MVCC results from the manual flushing of versions, which is mandatory to ensure failure-atomicity. Based on profiling, we observed that for BOCC and S2PL, most of the time is spent on locking, followed closely by logging (caused by PMDK transactions). S2PL acquires locks for the entire transaction time and, thus, transactions block each other more frequently and for longer as the contention increases. BOCC, on the other hand, only needs to lock the committed write sets shortly during validation. Therefore, compared to S2PL, BOCC is more resilient to higher skewness. At low contention, however, S2PL seems to scale better than BOCC. It is due to the fact that an exclusive lock with BOCC blocks throughout all reading transactions, while with S2PL, it only holds if equal keys are affected. Still, it is relevant to remind that the S2PL implementation is only failure-atomic per operation and not the entire transaction. Hence, a more extensive undo log per transaction would be required in practice. We can further observe that the out-of-place (i.e., non-optimized) variant of the MVCC protocol is more robust against higher skewness than the already optimized BOCC and S2PL protocols. However, beyond a certain level of contention, it also starts to deteriorate. With the highest setting, it is up to 3× slower than the in-place

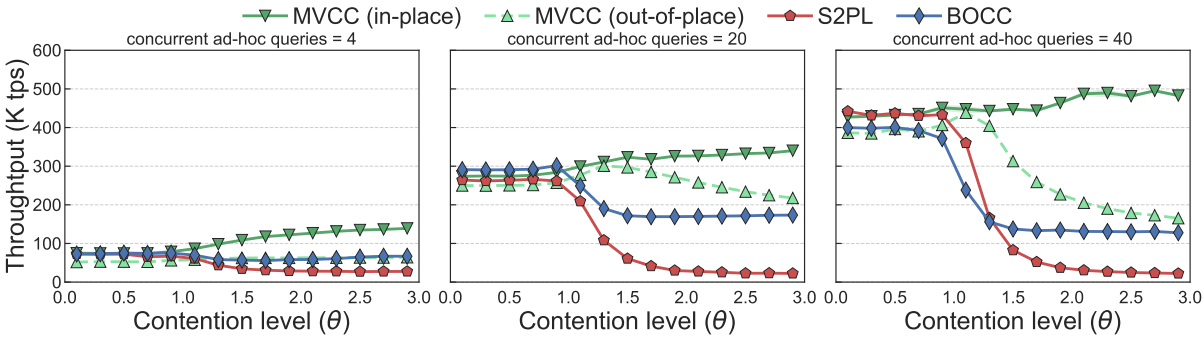



**Figure 5.12:** Resilience and scalability of CC protocols on PMem with medium-sized transactions.

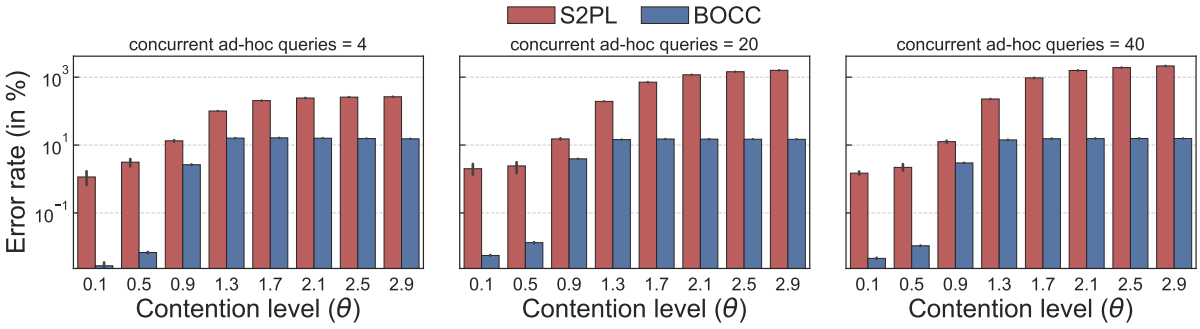



**Figure 5.13:** Abort Rate of CC protocols on PMem with medium-sized transactions.

variant. Our optimizations allow us to avoid locks and logs (for lookup and updates) as used before, which are also the main bottlenecks in the other two protocols. As also recognizable, a higher contention simultaneously leads to a higher cache-hit ratio, which is more exploitable by our optimizations. An alternative for the atomic fields would be to employ a shared mutex per object (i.e., certifier locks). We could only notice a minimal performance loss with this type of synchronization, at least when working on the same socket. As soon as the accesses crossed NUMA boundaries and there was a high contention, the performance dropped by a factor of two – which corresponds to 200-250K transactions per second with 40 concurrent queries. The reason for this behavior is that both readers and writers update the mutex variable. That causes cache invalidations, at a minimum on the other socket. It shows and confirms us that especially for NUMA environments, atomics in combination with the clwb instruction (i.e., no invalidation of cache lines) should be prioritized. For very low contention, on the other hand, we find that the single-version protocols provide comparable performance, and in some cases, are slightly more efficient than MVCC. However, it is also notable that for small states, which can occur frequently, the behavior is comparable to the results for high contention.

To examine the performance drop of the two single-version protocols in more detail, we consider in Figure 5.13 (note the logarithmic scaling) their abort rates in relation to contention. Apparently, as the contention level increases, so does the abort rate. It is also evident that S2PL consistently causes by far the most transaction restarts. At its peak (40 readers and $\theta = 2.9$), the abort rate is 2200%. In other words, on average, each transaction has to be restarted 22 times before it is successful. Accordingly, the throughput also drops sharply. As opposed to this, the error rate for BOCC is comparatively low, with a maximum of 15%. That is partly because the aborts only occur for actual conflicts, while with S2PL, already a potential deadlock causes an abort. BOCC also has a lower overlap probability since only the persistence phase of the writer is crucial. In contrast, our MVCC implementation does not lead to restarts or conflict-induced aborts by its very concept. As stated above, no consistency or isolation constraints are violated by atomically updating the timestamps and bitmaps.

### Concurrency Control with Short Transactions

As a next step, we look at the throughput achieved with very short transactions (one operation per table). The results are illustrated in Figure 5.14. As evident, the S2PL protocol performs better than before because the individual locking periods are correspondingly shorter. For the most part, it is faster than the optimistic approach this time, which is because the latter has more frequent validation phases that are responsible for most of the performance degradation. Furthermore, we observe that the performance drops caused by a higher contention are less severe than with medium-sized transactions. However, the number of parallel readers also changes the ranking among the protocols. For example, for BOCC, many readers with frequent validations mean that transactions almost consistently lock the collection of committed write sets. That, in turn, blocks write transactions in particular and makes them wait for a longer time, but readers also need to validate a higher number of write sets. The MVCC protocol is the clear winner for up to 20 readers. Only with a low contention level and a very high number of parallel readers (40) the pessimistic protocol is slightly better. The reason is that with practically no conflicts, S2PL can read the tuples directly, whereas MVCC requires the appropriate version to be selected based on the headers first. This small additional step becomes naturally more noticeable with an increasing number of readers. If we divide the throughput into read and write

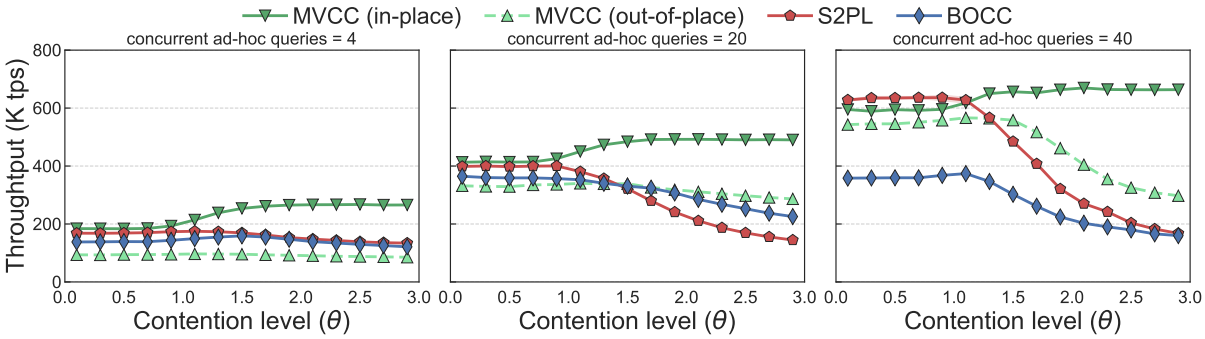



**Figure 5.14:** Resilience and scalability of CC protocols on PMem with short transactions.

portions, the write performance for a low contention is 82K tps for MVCC while S2PL reaches 75K tps. For comparison, BOCC only achieves a write throughput of 9K tps in this case. Since, from our perspective, the write transaction of a state is more crucial in a data stream processing system, we would prefer MVCC across the entire parameter spectrum. In the end, we again compare the values with the out-of-place variant at the highest contention and concurrency setting. The optimized variant achieves a performance gain of 2× and 16× in terms of read and write throughput, respectively. When added together, the throughput improves by a factor of 2.25×. With low competition, S2PL and BOCC – both already optimized – are usually faster than the unoptimized MVCC protocol. It again highlights the need for in-place updates and lock/latch elimination.

### Concurrency Control with Long Transactions

As a final CC experiment, we examine the throughput for the same scenario but with long transactions, as visualized in Figure 5.15. In the absence of contention, our MVCC approach performs worst for once. It is mainly because long reading transactions keep older versions alive longer. Hence, the write transactions have to wait until they can discard those to commit new versions. One way to counteract it would be to increase the number of slots, but this, in turn, increases the storage footprint. BOCC, on the other hand, performs best with low contention, which is since the costly validations occur much less frequently, and hardly any work is lost if there are no overlaps. However, the long transactions provoke overlaps to occur more often when the conflict probability increases, which is why the performance drop appears earlier in the graph. So in the worst-case, BOCC loses 70% compared to its peak performance. The S2PL protocol is even more extreme, and performance drops by 99%. That is because the long transactions usually include all frequently queried keys. Therefore, once a writing transaction starts locking, all readers are blocked for the whole duration of the transaction. A similar but slightly delayed drop is also notable for the MVCC variant featuring out-of-place updates. Here, it is due to synchronizing the simultaneous read and write accesses to the same MVCC objects. With high contention and concurrency, the performance is up to 10× worse compared to its optimized implementation. When updating in-place, readers and writers can continue to work on their versions in parallel, even during long transactions, without interfering with each other to any great extent. From this, we can conclude that our proposed optimization steps are particularly beneficial for long transactions and correspondingly long commit phases.

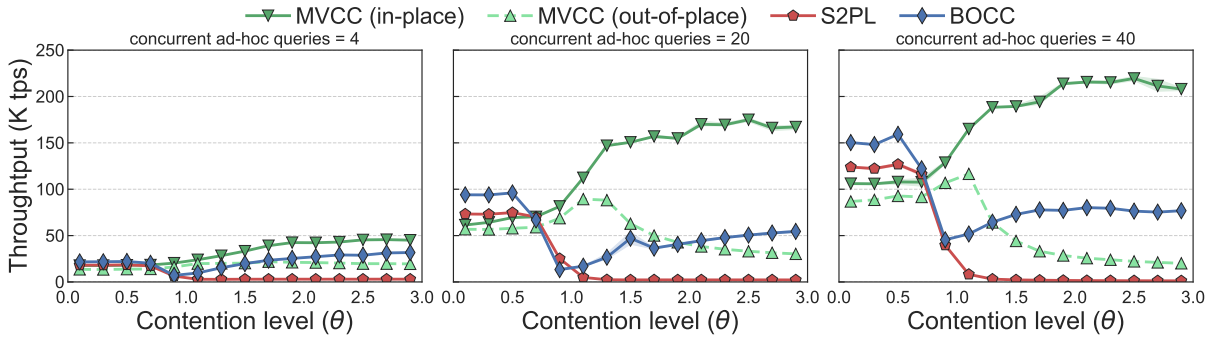



**Figure 5.15:** Resilience and scalability of CC protocols on PMem with long transactions.

### Discussion of Concurrency Control Experiments

All in all, we have seen that versioning combined with our atomic in-place optimizations usually performs better and is much more resilient than the single-version approaches. Therefore, it is the most suitable CC solution for the TSP model when dealing with PMem-based states. In Table 5.1, we provided an overview of the rankings of the protocols depending on the transaction length and contention level. By using in-place instead of out-of-place updates, locks and logs become superfluous. That increased the performance of our MVCC protocol in the scenario above by a factor of up to 2× to 10×. Such optimization made a particularly big difference with long transactions and high contention. It again shows the profitability of optimizing existing persistent data structures and access algorithms for PMem and the entire abolishment of locks and logs. Especially also considering NUMA effects, cross-socket locks turn out to be unsuitable. However, we should also point out the disadvantages of our approach. First of all, there is a higher storage requirement arising from our fixed array structure of the MVCC objects. Instead, a linked list could be used and only link versions where they are needed. Another alternative would be an indirection, in which a pointer to externally stored versions is held instead of the value(s). However, that would preclude the important in-place optimization and make the requirements for the underlying base table more stringent. As often the case, it results in a compromise between memory consumption and performance. Our fixed array variant sonner targets high performance and heavily queried states. Another drawback of our MVCC approach is the slightly increased overhead when virtually no conflicts occur. Furthermore, long transactions can lead to garbage collection loops in the writing thread when only a few version slots are available. Apart from these moderate drawbacks, we can conclude that our results from the CC experiments strongly support the hypotheses H3 and H4.

**Table 5.1:** Decision based on contention and transaction length.

| *contention* \ *transaction length* | Short | Medium | Long |
|---|---|---|---|
| Low | 1. **MVCC/S2PL**<br>2. BOCC | 1. **MVCC**<br>2. S2PL/BOCC | 1. **BOCC**<br>2. S2PL<br>3. MVCC |
| High | 1. **MVCC**<br>2. S2PL/BOCC | 1. **MVCC**<br>2. BOCC<br>3. S2PL | 1. **MVCC**<br>2. BOCC<br>3. S2PL |

### Query Recovery

Finally, we examine the recovery performance for the scenario set up above. Since the BOCC and S2PL protocol use PMDK transactions for atomic write operations, undo operations may be required in the event of failure. Theoretically, the duration depends on the length of the transaction. Simulating a real crash is not that easy since it would require clearing the entire cache immediately, among other things. In addition, we do not want to risk unnecessary damage to our system by suddenly cutting off electricity. Therefore, we only report the recovery time for restarts of the application. Initially, we expected the pool to take longer to open when using MVCC due to versioning and the associated higher PMem footprint. In fact, it made no visible difference, so all CC protocols resulted in about equal recovery times. Hence, we report the average of all protocols and several hundred restarts. Figure 5.16 illustrates the time taken for each of the steps outlined in Section 5.5.

As can be seen, most of the time is requisite for recovering the states and the context. It is mainly attributable to opening the pool files and potentially also includes undo operations. When we took a closer look at the results using profiling, it became apparent that a lot of time is needed to initialize the internal data structures of the PMDK pools. That includes, for example, the log and a recycler for allocations. Moreover, PMDK performs further expensive allocations and (persistent) memory initializations. By comparison, dereferencing and setting the pointers as well as restoring the query pipelines takes significantly less time. Therefore, we have included an additional zoom-in feature in the figure. Four ad-hoc reader queries had to be recovered for the shown results. Though, even with 40 readers, the recovery time of this step just doubled. Overall, we can thus conclude that the number of states constitutes the dominating influence on the recovery time. The number of queries or parameters such as the average transaction length only has a minor impact. In order to reduce the recovery time even further, it would be conceivable to store all states and the context in only one pool file. The problem that arises is the initial estimation of the required pool size. For example, if new stateful query pipelines are added, not enough PMem might have been allocated. A reallocation to a new pool could also be prohibitive and possibly block the entire system. Another variant would be to manage the memory regions manually instead of using PMDK and allow non-contiguous regions/files. That could also eliminate (for us) superfluous checks and initializations since we already take care of failure atomicity, at least in our MVCC and 2PC protocols. Apart from the additional effort due to the state recovery – which is still only in the millisecond range – we can confirm that our initial hypothesis of a near-instantaneous recovery process (H5) holds.

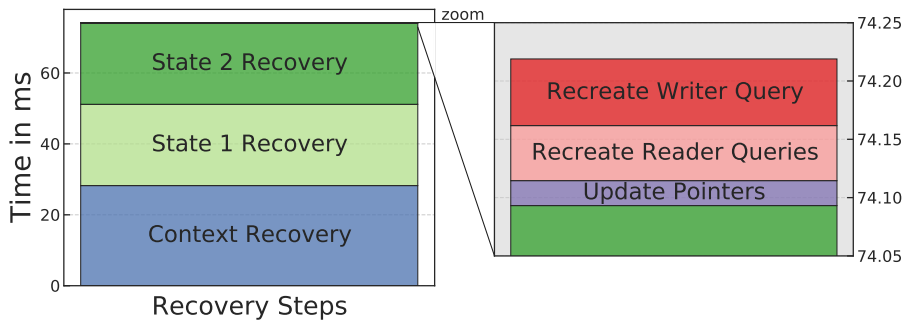



**Figure 5.16:** Temporal proportions of the query recovery steps.

### 5.8.2 Event Processing

In this part of the evaluation, we present micro-benchmarks based on the approaches described in Section 5.7. They are all prototypically implemented in ChronicleDB. The goal of the experiments is not to indicate that it resulted in the best event store but instead to highlight the possibilities and tradeoffs in such a three-layer storage system. Hence, the findings are also usable for similar application purposes. In this process, we show which of our proposed approaches prove judicious in practice and where improvements are still necessary.

#### Experimental Setup

Since ChronicleDB is written in Java, we cannot rely on PMDK being written in C/C++[6]. Instead, we made use of the JDK 14 extensions[7] to access PMem via the `ByteBuffer` interface. For the index, we set a node size of 8 KiB and applied the LZ4 compression algorithm to the nodes. In order to accommodate OOO events, we adjusted the spare space in each node to 10%. On the storage layer, we configured a block size of 32 KiB.

Events were held in DRAM and passed to ChronicleDB using the same Java process to achieve a high input rate. We used two synthetically generated event streams as test data. The first is the *Stock* event stream taken from [ZDI14]. The stream simulates a stock ticker where each item comprises the application timestamp, a sequence number, the symbol, the price, and the volume. Together the events result in a fixed size of 28 bytes each. This stream was used by default in the experiments, if not stated otherwise. The other data stream is called *Sine*. Besides the timestamp, each event has six 64-bit floating-point attributes resulting in a total size of 56 bytes. As the name suggests, the attributes are based on the sine function as follows: $\sin(\frac{i \bmod 1M}{1M} \cdot 2\pi)$ where $i$ is the sequence number of the event during generation. Practically, it means that for every million events, each attribute describes a complete sine wave. With this dataset, especially the selection rate of filter queries can be easily adjusted. It is used when evaluating filter and aggregation queries on the TAB$^+$-Tree nodes on PMem. For both data streams, the application time is consecutive, maintaining the order. The generation of OOO events is discussed in the corresponding paragraph. Generally, we keep the streams running for 100M events.

#### TAB$^+$-Tree

*Right flank:* At first, we consider the approach where the right flank of the TAB$^+$-Tree is moved from DRAM to PMem. We measured the total time taken for the insertion of the complete data stream into the primary index. The same we did for the original DRAM-based implementation and used it as a baseline for the visualization in Figure 5.17. Here, we varied the number of simultaneously flushed events (batch size) in the case of PMem. As expected, direct flushing at each insert (batch size = 1) degrades the performance too much, to about 50%. In return, maximum recovery guarantees would be obtained. Depending on the application, this performance might already be sufficient, as it still corresponds to 2.1M inserts per second. If this is not the case, 95% of the original performance can be achieved with a batch size of 25.

[6] Similar bindings are available such as PCJ (`https://github.com/pmem/pcj`). However, these generated too much overhead in our initial attempts.

[7] `https://openjdk.java.net/jeps/352`

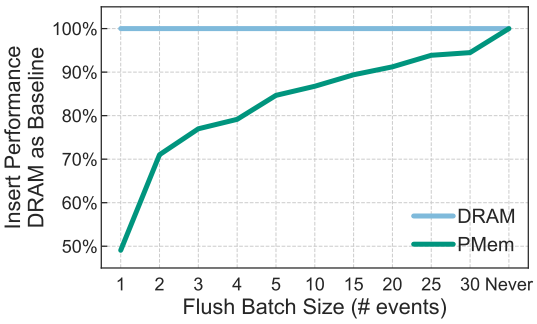



**Figure 5.17:** Insert performance when keeping TAB$^+$-Tree's right flank on PMem vs. DRAM depending on the flush batch size.

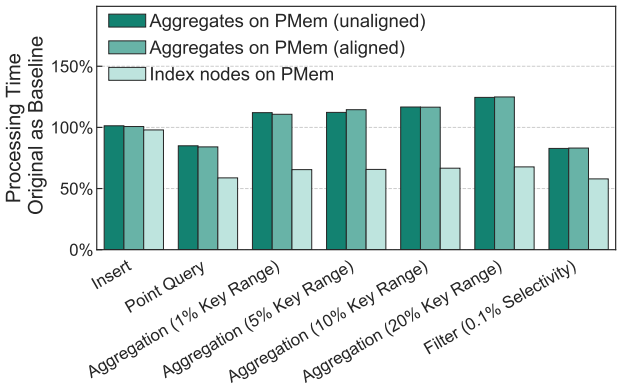



**Figure 5.18:** Insert and query performance for TAB$^+$-Tree aggregates and inner index nodes when maintained on PMem.

In terms of recovery guarantees, the number of possible losses is still far lower. Since a leaf node can hold up to 291 events (having a node size of 8 KiB) that could get lost in the DRAM case, it results in a 91% reduction in maximum data loss. In addition, for a tree height of three, the recovery time after a failure is reduced from 40 ms to less than 1 ms when using PMem.

*Aggregates and inner nodes:* Next, we will look at the tradeoffs of the other two proposed approaches to the TAB$^+$-Tree, namely moving either the lightweight index or the inner nodes of the primary index to PMem (cf. Figure 5.9 A.c & A.d). To investigate the influence of the alignment, we implemented two variants for the aggregate arrays. One of them stores the aggregates densely (i.e., unaligned), and the other aligns them on 64-byte/cache-line granularity. Since we also want to vary filter expressions predictably in this series of experiments, we use the *Sine* data stream here. Besides the initial insertion time, we looked at the processing time of point queries on the time domain, temporal aggregation queries on varying fractions, and finally at a filter query based on secondary attributes and selectivity of 0.1%. The aggregation and filter queries were chosen such that the information from the lightweight index was usable. In Figure 5.18, the results are illustrated using the original implementation as a baseline. When proceeding from left to right, we first see that the PMem adaptions have little effect on insert performance – unlike the first experiment. It stems from the fact that the writing of the leaves is still done on flash and the right flank is always in DRAM. Interestingly, in this and the following queries, the alignment of the aggregates did not yield any noticeable difference. We think it is because our setup is not able to utilize the entire PMem bandwidth. Therefore, a further distinction between aligned and unaligned array slots does not provide additional value and will be omitted accordingly. Continuing with the point queries, we see that storing the SMAs on PMem reduces the average lookup time by about 15%. This speedup results from the higher fan-out of the inner nodes, which, in turn, leads to a lower tree height. On the other hand, temporal aggregation queries do not benefit from the physical separation of the primary and secondary index. The reason for this is precisely this double access to both the index nodes and aggregates during traversal. The performance drop increases with the size of the set time range and reaches about 10% to 25%. However, filter queries can again benefit from the higher fan-out and are about 15% faster than the original implementation despite the additional access to the aggregates in PMem. In the case where we move all inner nodes, including their aggregates, to PMem, we exclusively observe improvements for each type of

query. Overall, a performance boost of 35% to 40% can be achieved, mainly because both index navigation and aggregate lookups no longer need to access flash. Therefore, this approach can be seen as a general improvement over the original implementation. However, in both approaches, the right flank is still in DRAM. That can result in higher data loss and recovery overhead when compared to the variant where it is placed in PMem.

### Address Translation

In the following, we will benchmark the address translation layer of the storage layout. The ATT maintains its right flank in DRAM – similar to the TAB$^+$-Tree – and uses an additional buffer to keep the 100 most recently accessed translation pages in memory. Besides the traditional and the PMem variant, we also built a DRAM-only version to highlight the read-write asymmetry for PMem access. In Figure 5.19, we illustrate the average time per sequential update, random lookup, and the total recovery time of the three implementations. The average timings are based on 10M operations each. The DRAM and flash-based implementations perform similarly regarding updates since the flash variant maintains the right flank and the buffer also in DRAM. Updates to PMem, on the other hand, are 5-6× slower, which stems from PMem's higher write latency in contrast to DRAM. On the contrary, since the read latency is closer to DRAM, lookups do not exhibit such a clear difference (1.8×). Although, the DRAM implementation features the best lookup and update performance, it is not an option for production use cases as it requires a complete rebuild on system restart, i.e., full file scan. Even for the relatively manageable dataset used here, it takes more than two minutes. Comparing the PMem and original flash-based implementation, random lookups are, on average, three orders of magnitude faster on PMem. That is since, in the case of flash, the ATT must be completely traversed if a translation page is not buffered. In the worst case, this corresponds to *(tree-height-1)* × 32 KiB reads from secondary storage for a single lookup. In terms of recovery, we see similar performance differences. While the PMem solution needs less than a millisecond (mapping its region into the address space), the flash-based solution takes 662 ms (rebuilding the right flank of the ATT). As a whole, however, it should be noted that the lookup and update times for address translation only take up a small portion of the system's overall performance. Nevertheless, PMem offers a sound balance between significant recovery times using flash and DRAM access times while also simplifying code complexity, given its flat array structure.

### Out-of-Order Handling

Finally, we examine the various OOO handling approaches. We start by comparing the insert, recovery, and query performance of the queue implementations isolated from the rest of the system. After that, we study the insertion performance in the context of the whole system for different OOO rates. Both parts are configured with a queue size of 100 MiB (≈3.5M events). In total, we compare five implementations. The first is a DRAM-based red-black tree ordered by application time that is utterly suited for fast query and merge latencies. Naturally, the OOO events in this queue are entirely lost in the case of failure. The next two implementations are based on PMem, where one is standalone and the other uses an additional DRAM index (cf. Section 5.7.4). Unlike the DRAM solution, the entries are not sorted during insertion but only appended. Analogously, these two variants are also implemented for flash using 8 KiB pages.

In Figure 5.20, the results of the isolated benchmarks are shown. When writing to the queues, the PMem solutions need more than twice the time than the DRAM-based tree. It is mainly due to

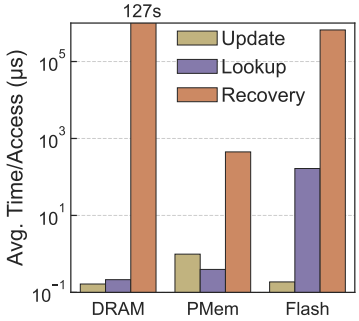



**Figure 5.19:** Update, lookup, and recovery tradeoffs for address translation maintained on DRAM, PMem, and flash.

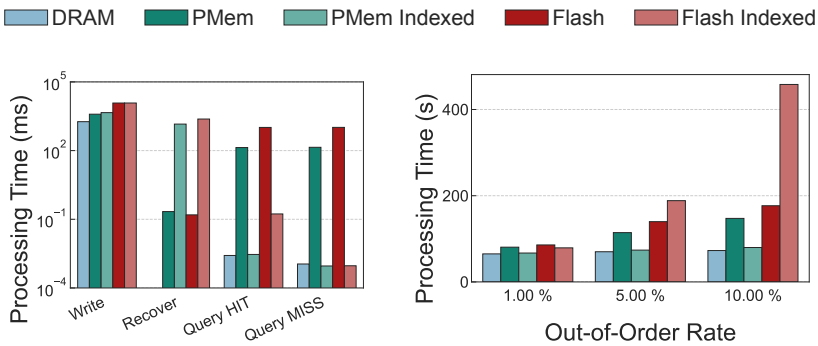



**Figure 5.20:** Insert, recovery, and query performance of proposed OOO queue variants.

**Figure 5.21:** Insertion time for 100M events with varying OOO rates with merges into the index.

the direct flushing of events, which could be compensated by batching, as shown in Figure 5.17. Batching, in turn, is regulated correspondingly with recovery guarantees. Compared with the flash variants, on the other hand, PMem yields a performance boost of 3× (pure) and 2× (indexed), respectively. Regarding recovery, only the indexed variants need to take further steps aside from opening the log. In the case of PMem, the regions must be additionally mapped to the user's address space. Here, the rebuilding of the volatile index dominates the recovery time. Flash only needs about 65% more time than PMem since the data is read sequentially. As a query, we performed lookups on application time, distinguishing HIT (event exists) and MISS (event does not exist). For queries with no corresponding event, the two index variants perform equally since only the DRAM portion is accessed. However, when queries hit, flash with index performs significantly worse (two orders of magnitude) because an entire page must be read from the log for each query. Due to the byte addressability of PMem, this variant can directly access the event instead and achieves similar performance as the pure DRAM solution. Without an index, recovery times are better, but the costs for the queries are clearly too high.

Now we look at the insert and merge performance of the queue implementations in the overall ChronicleDB system. We generated different rates of OOO events for the *Stock* data stream. Specifically, a OOO rate of x% means that (100-x)% of the events are inserted in order of application time. The OOO events are further subdivided into 10% distributed randomly uniformly over the application timespan and 90% distributed uniformly over 10,000 equidistant temporally close batches. That leads to a data stream including occasional OOO events with short bursts. In Figure 5.21, the results for a OOO rate of 1%, 5%, and 10% are shown. Whenever the queue is full (every 3.5M events), it must be merged into the TAB$^+$-Tree, which impacts write performance. Since the indexed variants are indirectly sorted, they can efficiently bulk merge. It is not the case for the non-indexed variants, resulting in a more random access pattern during writing. The path via the index, on the other hand, only produces random access when reading. In the hardware comparison, the indexed PMem approach achieves almost the same performance as the pure DRAM solution but offers persistence at the same time. Interestingly, the indexed flash-based implementation paints a different picture. At 5% and 10%, performance deteriorates drastically as random reads of flash pages during merging are much more costly. Thus, the indexed queue in PMem offers the best compromise overall.

### Lessons Learned

Using ChronicleDB as an example, we have seen that a three-layer approach, including PMem, offers many opportunities to improve the tradeoffs between different requirements of a data management system. To conclude, we summarize our general insights and lessons learned from the experiments, which are also applicable to other systems, as follows.

**LL1** As a replacement for DRAM components, PMem can be used to increase the amount of recoverable data in the event of a failure. However, depending on the necessary recovery guarantees, frequent updates of data/events should be flushed in contiguous batches in order not to degrade the write performance too much. We have shown this on the basis of the right flanks of the TAB$^+$-Tree. For LSM-Tree, for example, it is similarly realizable for the in-memory parts. On the other hand, if updates occur less frequently, as is the case with our OOO queue, batching is not imperative, and it is possible to, e.g., switch to simplified solutions to flatten performance fluctuations instead. Practically, this addresses the main challenge of managing OOO data. In general, these findings can be applied to other buffering or index maintenance techniques as well.

**LL2** From our experiments – where we moved the aggregates from the index to a separate structure in PMem – it is clear that a faster medium does not automatically mean better performance. By separating primary and secondary indexes, we have eliminated spatial locality and, thus, sequential access for queries that need both indexes. Therefore, we have learned that access patterns still have an immense influence. Similar results were also evident from the latter OOO experiment. In conclusion, physical locality as achieved with correlated index structures should be considered even for byte-addressable PMem.

**LL3** For non-performance-critical components in particular, such as the address translation layer demonstrated, PMem offers promising opportunities to reduce code complexity through simplified designs and improve recovery guarantees and performance.

## 5.9 Summary

In this chapter, we applied the data structures and lessons learned from the previous chapters. We integrated them into a stream processing and an event processing engine to extend these with PMem support. In doing so, we initially devised a novel processing model that can process both stream and table data with transactional guarantees. Furthermore, due to our designated MVCC and consistency protocols, our model can provide features such as time-travel queries, shared states, and cross-state consistency. Apart from the functional enhancements over traditional DBMSs or streaming systems, the use of PMem yields quick, durable updates of states and tables and near-instantaneous recovery in such a model. By considering different CC procedures, we could also address the last remaining data management challenge from Section 2.3. Looking ahead, we also considered possibilities for query planning in the TSP model, identifying crucial parameters and potential cost formulas to decide for optimal data structures, data placement, and algorithms.

Our experiments have shown that fundamentally MVCC is the most suitable CC approach for the TSP model using PMem. Particularly with high contention, it was significantly more

scalable and resilient than lock-based or optimistic protocols. That is mainly due to the complete avoidance of locks, which become very expensive, especially when crossing NUMA boundaries. Instead, we created a PMem-optimized adaption to do updates in-place using atomic primitives. In addition, the omission of logs and the manual placement of flush and fence instructions keep the write throughput high. It also enables swift state and query recovery, whose bottleneck is now only the unavoidable opening of the file pool(s). Furthermore, we found that the realization of cross-state consistency does not imply any visible overhead in our tested scenario.

Concerning event processing, we also found a general tradeoff between recovery guarantees and write performance. Here, flushing in batches has proven to be a possible solution, with the batch size serving as a tuning knob. Another interesting finding was that the access pattern combined with the physical proximity of the data sometimes has a higher impact than the used storage medium. Finally, we found that PMem can massively reduce code complexity compared to using disks.

# 6 Conclusion

Persistent memory is a pioneering storage technology that we believe could become a standard feature of data centers in the coming years. However, how it is accessed and how to program with it differs from what was common in the past. Therefore, along with new opportunities, it also holds some pitfalls. In this dissertation, we have studied the impact of PMem across various data management layers to provide sensible techniques and insights for the design of modern transactional and analytical systems. In this chapter, we summarize the findings and contributions of this work and close with a consideration of possible directions for future work.

## 6.1 Contributions

We organize the summary of contributions similarly to Chapter 1, following the building blocks for data management with PMem. However, the focus is now on our achieved results and insights.

### Persistent Memory Access & Data Management

We started by giving an overview of the fundamentals of the PMem technologies. We extracted the common properties and also highlighted special features of the DCPMMs we used. These were particularly important for elaborating on the design goals and guidelines introduced at a later stage. Furthermore, we have shown several integration possibilities of PMem into the hardware landscape. For our purposes, the strategy PMem side-by-side with DRAM has proven to be the best option, as it offers the highest utilization potential of the properties of all technologies. In this case, the most efficient way to access PMem is by mapping files to the application's virtual address space. That allows direct access to the device using load and store instructions. We have also identified evolving data management challenges using PMem, namely failure atomicity, property utilization, data placement, and concurrency. During the development and analysis of transactional and analytical data structures, we addressed the former three challenges in detail. The concurrency aspect, on the other hand, was handled one layer above when processing stream and table data simultaneously.

### Persistent Index Structures

Encouraged by the finding that most existing works proposing PMem data structures followed a black-box approach in their evaluation, we opted to take a white-box approach instead, which can be more precise about the impact of certain design decisions. Therefore, we identified and evaluated several design primitives for data structures at a more fine-grained level. These primitives typically target and thus have been classified into PMem-critical design goals, namely write reduction, fine-grained access, and failure atomicity. In addition, we identified typical low-level access patterns – which we call micro-operations – that can be executed on or by using the primitives. With the results of our extensive experiments, we were able to obtain explicit performance profiles for selected primitives, as well as more general conclusions for designing PMem-based data structures. In particular, the latter point highlights the further advantage of our approach in that insights are more generally applicable and thus valid to different data structures such as $B^+$-Trees, LSM-Trees, Skip-Lists, and Tries. In general, we can state that data structures should perform as few PMem writes or cache line flushes as possible. It can be done, for example, with hybrid data placements such as with the FPTree, where DRAM holds the secondary recoverable data. Also, the omission of sorting records after each modification saves a lot of write operations. For failure atomicity, we have seen that copy-on-write approaches should generally be avoided and replaced by in-place updates using lightweight atomic writes. The access to PMem should follow a sequential pattern if possible so that prefetching and other hardware features can take effect. Furthermore, a node size of 256 bytes to 1 KiB is the most efficient. Finally, allocations on PMem are significantly more expensive than on DRAM, which can be counteracted with group allocations or dedicated free space management.

### Persistent Analytical Structures

We have elaborated on two approaches to exploit PMem for analytical data management tasks. Our first approach uses a clustering mechanism (based on BDCC) to achieve physical proximity for similar data. We distribute the data over a linked list of nodes sorted by the clustering key. The nodes themselves are unsorted to reduce PMem writes and to follow a sequential insertion pattern. Another significant advantage of clustering is the efficient querying of non-key attributes, which is especially useful for stateful stream processing where the key is usually the timestamp. Thus, range queries with low selection rates were several orders of magnitude faster than typical index structures. That was achieved with the support of SMAs in the data nodes and appropriate pruning steps. In addition, we can add any index on top to perform point queries in logarithmic or constant complexity. The design can also include additional storage layers such as disks to distinguish between cold and hot data nodes. Combined with a volatile index, this allows us to leverage the properties of all memory and storage technologies. As an alternative analytical approach, we ported an existing multi-dimensional index called Elf to PMem. In contrast to the clustered structure, it has a much lower memory footprint but potentially needs a complete rebuild in case of updates. This approach is particularly profitable if the index should be persistent or not enough DRAM is available. To still reach a similar performance as its DRAM counterpart, we considered several caching strategies. Since these do not consider all index nodes and are comparable to the FPTree design (selective persistence), we summarized our dynamic and static variants under the umbrella term selective caching. Mostly, a static approach – with predetermined cached levels in DRAM – has turned out to

be faster than a dynamic approach. Only the naïve and LLA dynamic eviction policies have proven competitive, where the latter is our own devised policy based on probabilities. With 97% less DRAM consumption, we could reduce the throughput for point queries by just about half (two-thirds to three-quarters without DRAM cache). Due to the sequential access, very wide range queries to the PMem index were already half as fast as the DRAM variant, even without caching. The caching was then only able to achieve a slight performance improvement. Finally, we have demonstrated that static caching works very well with parallel range scans as well.

#### Transactional Stream Processing

With the above data structures and insights, we implemented a prototype of a TSP system based on PMem. The data structures serve as state and table representations applied according to the expected access pattern. Besides the states, parts of the global state context – comparable to a transaction manager – are also stored in PMem to ensure the ACID guarantees. In addition, we have proposed two protocols for this purpose. On the one hand, we created an MVCC approach adapted to PMem to achieve a scalable and resilient performance for concurrent queries on a state. In particular, compared with traditional pessimistic and optimistic protocols, this approach can cope with very high concurrency and contention. On the other hand, we delay the commit until all involved states of a transaction have received and recorded it and lastly atomically overwrite the commit timestamp. Hence, this procedure is comparable to a 2PC protocol. Both protocols have in common that they perform in-place updates and do not require any logs or locks improving the throughput. It also means that the recovery process does not have to handle logs. Only the pool files in PMem have to be mapped into the virtual address space, and the query pipelines have to be restored. Both take no longer than a few milliseconds. Choosing a suitable data structure, its placement, and the applied algorithms to a state strongly depends on the available hardware and the expected access pattern. Therefore, the establishment of a cost model seems reasonable. To address this need, we have already investigated possible parameters and compiled initial cost formulas for a hardware-conscious model for stateful stream processing. The task of the optimizer is then to decide upon a suitable combination of state representation, data placement, and access algorithms.

#### Event Stream Processing

We have developed various approaches to enhance the existing event store ChronicleDB with PMem support. These show the impact of data placements within a modern three-layer architecture comprising DRAM, PMem, and disk. In general, there is a trade-off between the recoverability of events and their writing speed, which can be controlled by the number of events flushed together. We also saw that the recovery process with PMem is always the most efficient. With DRAM, all secondary structures have to be rebuilt, and with disks, the loading of the logs and the like takes significantly longer. Since we can often avoid such logs with PMem, we can also reduce the code complexity. Furthermore, as with our clustering approach, it has been shown that a physical closeness of correlated data is sometimes more critical than the underlying storage technology. Explicitly, for ChronicleDB, this means that primary and secondary indexes should be stored interleaved in order not to disrupt sequential access patterns.

## 6.2 Future Work

Since the first widely commercially available PMem technology did not hit the market until 2019, we can say that data management research in this area is still in its infancy. Our detailed examination of PMem-based transactional and analytical data structures as well as their usage for stateful stream processing already covers a multitude of application scenarios leveraging this novel hardware. However, we believe there is still a lot of potential for future work that can build on our contributions. Therefore, we will outline promising research directions regarding data management on PMem that expand on our findings below.

#### Extending the Consideration of PMem Primitives and Micro-Operations

Table 3.1 and our extensive evaluation still have room for extensions. That includes more primitives, more micro-operations, other performance counters, and probably even more base data structures. For example, a possible further micro-operation would be to iterate through a list of nodes (see experiment E3) additionally in sorted form (≡ range scan), which would require on-demand sorting for the unsorted approaches. Likewise, other primitives could be included, such as balancing a node (see experiment E9) using *quickselect* for unsorted nodes, which is currently implemented by searching for the next maximum/minimum for each record to be moved. Regarding node layouts or general data structures, there is probably still plenty of designs to be explored yet. For instance, we have considered hashtables relatively sparsely so far, though such extensions could result in entirely new types of data structures that enable new use cases. Also, in addition to the runtime and the number of written bytes, we could include more performance counters. These could be the number of read bytes, cache misses, or even energy usage. That could allow us to understand the behavior of the primitives even better. As a final result, we could correspondingly generate more comprehensive performance profiles.

#### Query Optimization and Cost Models

In this dissertation, we have already covered the first layers of data management with PMem. We have proposed various data structures and alternative variations thereof and several approaches for stateful stream processing. However, future work could examine query planning and automated implementation selections in more detail to relieve or ease the workload of administrators.

Following on from the extension of our data structure primitive consideration discussed above, we envision a system similar to the Data Calculator [IZH+18] that automatically assembles data structures of these primitives according to the hardware profile and workload expectations. Also, our analytical concepts could benefit from an automatic selection of appropriate parameters or implementations. For example, for the clustering approach, we have to choose between a sequential block iterator with pruning or an index-based iterator, which highly depends on the expected selection size. In addition, the block and table size are decisive here. With our multi-dimensional index and the corresponding caching strategies, it would also be helpful to automatically decide, e.g., between sequential or parallel access and about the number of cached dimension levels.

Another beneficial enhancement would be the automatic selection of a state representation per operator for stateful stream processing. Specifically for our TSP model, we have already described the first theoretical steps in this regard in Section 5.6. However, a prototypical or fully functional implementation is beyond the scope of this work and considered as future work. It would initially require calibrating the available stateful operators (state access characteristics) and the hardware latencies. Subsequently, it is necessary to weight these values with the costs per access specific to the state representation. Depending on the dominating access type (updating, querying, or recovery) or other requirements, the most cost-effective structure and corresponding algorithms should be selected.

#### Inclusion of Further Hardware Features

Besides the research on the software side, there is also continual research on the hardware level, and manufacturers introduce new features, from which some also directly address PMem. Observing these developments is always beneficial if data processing should run as efficiently as possible on the target architecture. Therefore, we will briefly look at some of such existing and possibly upcoming features and their implications.

Throughout working with our server, we found out that it does provide the clwb instruction, but it is just a wrapper on clflushopt. It means that any flush to PMem will also invalidate the affected cache line. For us, this implies that our results may differ if the same line has to be read once more after a flush, which will likely be the case with high contention. However, future work using newer generations of servers (i.e., starting from Intel Ice Lake-SP) would have to verify this yet.

Another innovation introduced by the follow-up server generation is eADR. With the help of capacitors and corresponding logic, the CPU caches are also persistent in this sense. The utmost advantage is that explicit flushes are no longer necessary to guarantee failure atomicity. Only the barriers (fences) are still required. Overall, however, this should result in a massive increase in performance, making PMem even more appealing.

In [OLN+16], the authors already considered Hardware Transactional Memory (HTM) in the context with PMem. However, this feature is only supported for DRAM-resident data and is incompatible with persistent primitives like mandatory cache line flushes. Though, coupled with the eADR feature, the forced flushing is eliminated, which would basically allow HTM on PMem. That would simplify it for developers to make multiple regions simultaneously persistent and visible to other threads. Whether the performance is better than individual solutions with fine-granular latches or atomics remains to be seen. A setback is that Intel disabled their HTM implementations by default on most platforms for security reasons and because of potential bugs regarding the memory order [Int21c, Int21d].

Finally, the use of SIMD instructions on PMem would be conceivable, especially for analytical workloads. Thus, the range scans in our clustering and multi-dimensional approach could get an additional performance boost. As already shown, for instance, in [ZDK+21], the employment of SIMD can significantly increase the performance for both DRAM and PMem. However, from a certain degree of parallelism, the clock frequency is reduced, e.g., for thermal reasons, and thus a scalar execution or mixture might be more efficient.

### Scalability and Availability

In this work, we have considered and evaluated our approaches only on a single server machine. Especially in the growing cloud environment, however, several server nodes are often combined into a cluster to perform a common task. Moreover, to handle hardware failures, multiple nodes can be interconnected to replicate selected data or their entire state.

As a first step, our NUMA consideration could be raised from two sockets to a higher number and examine different data distributions. Thus, each socket could either manage only its own allocated area or operate across NUMA boundaries. Generally, it would be reasonable to partition the states (or other data structures), whereby the ACID properties must always remain fulfilled. For this purpose, our 2PC-like protocol, among others, could be refined to maintain consistency not only of all accessed states per transaction but all state partitions.

The next stage would be to distribute states and metadata over the network to scale the system even further. Apart from the partitions, replications might now be added, which could be handled similarly by our consistency protocol. In this case, we could also utilize features like Remote Direct Memory Access (RDMA), which allows comparable latencies to those of remote socket accesses. Particularly in conjunction with PMem, this could be a promising research direction.

# Bibliography


[ABD+12] Aditya Auradkar, Chavdar Botev, Shirshanka Das, Dave De Maagd, Alex Feinberg, Phanindra Ganti, Lei Gao, Bhaskar Ghosh, Kishore Gopalakrishna, Brendan Harris, Joel Koshy, Kevin Krawez, Jay Kreps, Shi Lu, Sunil Nagaraj, Neha Narkhede, Sasha Pachev, Igor Perisic, Lin Qiao, Tom Quiggle, Jun Rao, Bob Schulman, Abraham Sebastian, Oliver Seeliger, Adam Silberstein, Boris Shkolnik, Chinmay Soman, Roshan Sumbaly, Kapil Surlaker, Sajid Topiwala, Cuong Tran, Balaji Varadarajan, Jemiah Westerman, Zach White, David Zhang, and Jason Zhang. Data Infrastructure at LinkedIn. In Anastasios Kementsietsidis and Marcos Antonio Vaz Salles, editors, *IEEE 28th International Conference on Data Engineering (ICDE 2012), Washington, DC, USA (Arlington, Virginia), 1-5 April, 2012*, pages 1370–1381. IEEE Computer Society, https://doi.org/10.1109/ICDE.2012.147, 2012.

[ABW03] Arvind Arasu, Shivnath Babu, and Jennifer Widom. CQL: A Language for Continuous Queries over Streams and Relations. In Georg Lausen and Dan Suciu, editors, *Database Programming Languages, 9th International Workshop, DBPL 2003, Potsdam, Germany, September 6-8, 2003, Revised Papers*, volume 2921 of *Lecture Notes in Computer Science*, pages 1–19. Springer, https://doi.org/10.1007/978-3-540-24607-7_1, 2003.

[Agi14] AgigA Tech, Inc. AGIGARAM DDR4 Non-Volatile DIMM. Retrieved April 12, 2021 from http://agigatech.com/wp-content/uploads/2014/08/AGIGARAM-DDR4-Product-Brief.pdf, 2014.

[AJ89] Rakesh Agrawal and H. V. Jagadish. Recovery Algorithms for Database Machines with Nonvolatile Main Memory. In Haran Boral and Pascal Faudemay, editors, *Database Machines, Sixth International Workshop, IWDM '89, Deauville, France, June 19-21, 1989, Proceedings*, volume 368 of *Lecture Notes in Computer Science*, pages 269–285. Springer, https://doi.org/10.1007/3-540-51324-8_41, 1989.

[AKM+16] Manos Athanassoulis, Michael S. Kester, Lukas M. Maas, Radu Stoica, Stratos Idreos, Anastasia Ailamaki, and Mark Callaghan. Designing Access Methods: The RUM Conjecture. In Evaggelia Pitoura, Sofian Maabout, Georgia Koutrika, Amélie Marian, Letizia Tanca, Ioana Manolescu, and Kostas Stefanidis, editors, *Proceedings of the 19th International Conference on Extending Database Technology, EDBT 2016, Bordeaux, France, March 15-16, 2016, Bordeaux, France, March 15-16, 2016*, pages 461–466. OpenProceedings.org, https://doi.org/10.5441/002/edbt.2016.42, 2016.

[ALML18] Joy Arulraj, Justin J. Levandoski, Umar Farooq Minhas, and Per-Åke Larson. BzTree: A High-Performance Latch-free Range Index for Non-Volatile Memory. *PVLDB*, 11(5):553–565, http://www.vldb.org/pvldb/vol11/p553-arulraj.pdf, 2018.

[AMC20] Lorenzo Affetti, Alessandro Margara, and Gianpaolo Cugola. TSpoon: Transactions on a stream processor. *J. Parallel Distributed Comput.*, 140:65–79, https://doi.org/10.1016/j.jpdc.2020.03.003, 2020.

[APD15] Joy Arulraj, Andrew Pavlo, and Subramanya Dulloor. Let's Talk About Storage & Recovery Methods for Non-Volatile Memory Database Systems. In Timos K. Sellis, Susan B. Davidson, and Zachary G. Ives, editors, *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, pages 707–722. ACM, https://doi.org/10.1145/2723372.2749441, 2015.

[APGZ17] Dana Van Aken, Andrew Pavlo, Geoffrey J. Gordon, and Bohan Zhang. Automatic Database Management System Tuning Through Large-scale Machine Learning. In Semih Salihoglu, Wenchao Zhou, Rada Chirkova, Jun Yang, and Dan Suciu, editors, *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*, pages 1009–1024. ACM, https://doi.org/10.1145/3035918.3064029, 2017.

[APM19] Joy Arulraj, Andy Pavlo, and Krishna Teja Malladi. Multi-Tier Buffer Management and Storage System Design for Non-Volatile Memory. *CoRR*, abs/1901.10938, http://arxiv.org/abs/1901.10938, 2019.

[Axb20] Jens Axboe. Flexible I/O Tester. 2020. Retrieved April 12, 2021 from https://github.com/axboe/fio, version 3.7.

[BAF+09] Irina Botan, Gustavo Alonso, Peter M. Fischer, Donald Kossmann, and Nesime Tatbul. Flexible and scalable storage management for data-intensive stream processing. In Martin L. Kersten, Boris Novikov, Jens Teubner, Vladimir Polutin, and Stefan Manegold, editors, *EDBT 2009, 12th International Conference on Extending Database Technology, Saint Petersburg, Russia, March 24-26, 2009, Proceedings*, volume 360 of *ACM International Conference Proceeding Series*, pages 934–945. ACM, https://doi.org/10.1145/1516360.1516467, 2009.

[BBS16] Stephan Baumann, Peter A. Boncz, and Kai-Uwe Sattler. Bitwise dimensional co-clustering for analytical workloads. *VLDB J.*, 25(3):291–316, https://doi.org/10.1007/s00778-015-0417-y, 2016.

[BdNSS10] Stephan Baumann, Giel de Nijs, Michael Strobel, and Kai-Uwe Sattler. Flashing Databases: Expectations and Limitations. In Anastasia Ailamaki and Peter A. Boncz, editors, *Proceedings of the Sixth International Workshop on Data Management on New Hardware, DaMoN 2010, Indianapolis, IN, USA, June 7, 2010*, pages 9–18. ACM, https://doi.org/10.1145/1869389.1869391, 2010.

[BFF+07] Michael A. Bender, Martin Farach-Colton, Jeremy T. Fineman, Yonatan R. Fogel, Bradley C. Kuszmaul, and Jelani Nelson. Cache-Oblivious Streaming B-trees. In *SPAA 2007: Proceedings of the 19th Annual ACM Symposium on Parallelism in Algorithms and Architectures, San Diego, California, USA, June 9-11, 2007*, pages 81–92, https://doi.org/10.1145/1248377.1248393, 2007.

[BFKT12] Irina Botan, Peter M. Fischer, Donald Kossmann, and Nesime Tatbul. Transactional Stream Processing. In Elke A. Rundensteiner, Volker Markl, Ioana Manolescu, Sihem Amer-Yahia, Felix Naumann, and Ismail Ari, editors, *15th International Conference on Extending Database Technology, EDBT '12, Berlin, Germany, March 27-30, 2012, Proceedings*, pages 204–215. ACM, https://doi.org/10.1145/2247596.2247622, 2012.

[BHG87] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, 1987.

[Bin18] Carsten Binnig. Scalable Data Management on Modern Networks. *Datenbank-Spektrum*, 18(3):203–209, https://doi.org/10.1007/s13222-018-0297-6, 2018.

[BKSS17] David Broneske, Veit Köppen, Gunter Saake, and Martin Schäler. Accelerating Multi-Column Selection Predicates in Main-Memory - The Elf Approach. In *33rd IEEE International Conference on Data Engineering, ICDE 2017, San Diego, CA, USA, April 19-22, 2017*, pages 647–658. IEEE Computer Society, https://doi.org/10.1109/ICDE.2017.118, 2017.

[BKSS19] David Broneske, Veit Köppen, Gunter Saake, and Martin Schäler. Efficient Evaluation of Multi-Column Selection Predicates in Main-Memory. *IEEE Trans. Knowl. Data Eng.*, 31(7):1296–1311, https://doi.org/10.1109/TKDE.2018.2825349, 2019.

[Blo19] Paul Blockhaus. Parallelizing the Elf - A Task Parallel Approach. Bachelor thesis, University of Magdeburg, December 2019.

[Car19] Carnegie Mellon University Database Group. Peloton: The Self-Driving Database Management System. Retrieved April 12, 2021 from https://pelotondb.io/, 2019.

[Car21] Carnegie Mellon University Database Group. NoisePage – Self-Driving Database Management System. Retrieved April 12, 2021 from https://noise.page/, 2021.

[CEF+17] Paris Carbone, Stephan Ewen, Gyula Fóra, Seif Haridi, Stefan Richter, and Kostas Tzoumas. State Management in Apache Flink®: Consistent Stateful Distributed Stream Processing. *Proc. VLDB Endow.*, 10(12):1718–1729, https://doi.org/10.14778/3137765.3137777, 2017.

[CFE+15] Paris Carbone, Gyula Fóra, Stephan Ewen, Seif Haridi, and Kostas Tzoumas. Lightweight Asynchronous Snapshots for Distributed Dataflows. *CoRR*, abs/1506.08603, http://arxiv.org/abs/1506.08603, 2015.

[CFKK20] Paris Carbone, Marios Fragkoulis, Vasiliki Kalavri, and Asterios Katsifodimos. Beyond Analytics: The Evolution of Stream Processing Systems. In David Maier, Rachel Pottinger, AnHai Doan, Wang-Chiew Tan, Abdussalam Alawini, and Hung Q. Ngo, editors, *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*, pages 2651–2658. ACM, https://doi.org/10.1145/3318464.3383131, 2020.

[Chu71] Leon Chua. Memristor – The Missing Circuit Element. *IEEE Transactions on Circuit Theory*, 18(5):507–519, 1971.

[CJ15] Shimin Chen and Qin Jin. Persistent B+-Trees in Non-Volatile Main Memory. *PVLDB*, 8(7):786–797, https://doi.org/10.14778/2752939.2752947, 2015.

[CL85] K. Mani Chandy and Leslie Lamport. Distributed Snapshots: Determining Global States of Distributed Systems. *ACM Trans. Comput. Syst.*, 3(1):63–75, https://doi.org/10.1145/214451.214456, 1985.

[CM86] Michael J. Carey and Waleed A. Muhanna. The Performance of Multiversion Concurrency Control Algorithms. *ACM Trans. Comput. Syst.*, 4(4):338–378, https://doi.org/10.1145/6513.6517, 1986.

[CNF+09] Jeremy Condit, Edmund B. Nightingale, Christopher Frost, Engin Ipek, Benjamin C. Lee, Doug Burger, and Derrick Coetzee. Better I/O Through Byte-Addressable, Persistent Memory. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles 2009, SOSP 2009, Big Sky, Montana, USA, October 11-14, 2009*, pages 133–146, https://doi.org/10.1145/1629575.1629589, 2009.

[CRF08] Michael J. Cahill, Uwe Röhm, and Alan D. Fekete. Serializable Isolation for Snapshot Databases. In Jason Tsong-Li Wang, editor, *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2008, Vancouver, BC, Canada, June 10-12, 2008*, pages 729–738. ACM, https://doi.org/10.1145/1376616.1376690, 2008.

[DAP+14] Justin DeBrabant, Joy Arulraj, Andrew Pavlo, Michael Stonebraker, Stanley B. Zdonik, and Subramanya Dulloor. A Prolegomenon on OLTP Database Systems for Non-Volatile Memory. In Rajesh Bordawekar, Tirthankar Lahiri, Bugra Gedik, and Christian A. Lang, editors, *International Workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architectures - ADMS 2014, Hangzhou, China, September 1, 2014*, pages 57–63, http://www.adms-conf.org/2014/adms14_debrabant.pdf, 2014.

[DBL15] *CIDR 2015, Seventh Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 4-7, 2015, Online Proceedings*. www.cidrdb.org, 2015.

[DFI+13] Cristian Diaconu, Craig Freedman, Erik Ismert, Per-Åke Larson, Pravin Mittal, Ryan Stonecipher, Nitin Verma, and Mike Zwilling. Hekaton: SQL Server's Memory-Optimized OLTP Engine. In Kenneth A. Ross, Divesh Srivastava, and Dimitris Papadias, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, June 22-27, 2013*, pages 1243–1254. ACM, https://doi.org/10.1145/2463676.2463710, 2013.

[DHK+15] Biplob Debnath, Alireza Haghdoost, Asim Kadav, Mohammed G. Khatib, and Cristian Ungureanu. Revisiting Hash Table Design for Phase Change Memory. In Peter Desnoyers and Gokul Kandiraju, editors, *Proceedings of the 3rd Workshop on Interactions of NVM/FLASH with Operating Systems and Workloads, INFLOW 2015, Monterey, California, USA, October 4, 2015*, pages 1:1–1:9. ACM, https://doi.org/10.1145/2819001.2819002, 2015.

[DI18] Niv Dayan and Stratos Idreos. Dostoevsky: Better Space-Time Trade-Offs for LSM-Tree Based Key-Value Stores via Adaptive Removal of Superfluous Merging. In Das et al. [DJB18], https://doi.org/10.1145/3183713.3196927, pages 505–520.

[DJB18] Gautam Das, Christopher M. Jermaine, and Philip A. Bernstein, editors. *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*. ACM, 2018.

[DKB+19] Markus Dreseler, Jan Kossmann, Martin Boissier, Stefan Klauck, Matthias Uflacker, and Hasso Plattner. Hyrise Re-engineered: An Extensible Database System for Research in Relational In-Memory Data Management. In Herschel et al. [HGR+19], https://doi.org/10.5441/002/edbt.2019.28, pages 313–324.

[DMK+20] Laxman Dhulipala, Charles McGuffey, Hongbo Kang, Yan Gu, Guy E. Blelloch, Phillip B. Gibbons, and Julian Shun. Sage: Parallel Semi-Asymmetric Graph Algorithms for NVRAMs. *Proc. VLDB Endow.*, 13(9):1598–1613, https://doi.org/10.14778/3397230.3397251, 2020.

[DPT+13] Justin DeBrabant, Andrew Pavlo, Stephen Tu, Michael Stonebraker, and Stanley B. Zdonik. Anti-Caching: A New Approach to Database Management System Architecture. *Proc. VLDB Endow.*, 6(14):1942–1953, https://doi.org/10.14778/2556549.2556575, 2013.

[EGA+18] Assaf Eisenman, Darryl Gardner, Islam AbdelRahman, Jens Axboe, Siying Dong, Kim M. Hazelwood, Chris Petersen, Asaf Cidon, and Sachin Katti. Reducing DRAM Footprint with NVM in Facebook. In Rui Oliveira, Pascal Felber, and Y. Charlie Hu, editors, *Proceedings of the Thirteenth EuroSys Conference, EuroSys 2018, Porto, Portugal, April 23-26, 2018*, pages 42:1–42:13. ACM, https://doi.org/10.1145/3190508.3190524, 2018.

[EGLT76] Kapali P. Eswaran, Jim Gray, Raymond A. Lorie, and Irving L. Traiger. The Notions of Consistency and Predicate Locks in a Database System. *Commun. ACM*, 19(11):624–633, https://doi.org/10.1145/360363.360369, 1976.

[Fac20] Facebook Open Source. RocksDB: A Persistent Key-Value Store for Flash and RAM Storage (v6.13.3). Retrieved April 12, 2021 from https://github.com/facebook/rocksdb, 2020.

[FCP+11] Franz Färber, Sang Kyun Cha, Jürgen Primsch, Christof Bornhövd, Stefan Sigg, and Wolfgang Lehner. SAP HANA Database - Data Management for Modern Business Applications. *SIGMOD Rec.*, 40(4):45–51, https://doi.org/10.1145/2094114.2094126, 2011.

[FUJ20] FUJITSU Technology Solutions GmbH. libart. Retrieved June 18, 2021 from https://github.com/pmem/pmdk/tree/master/src/examples/libpmemobj/libart, 2020.

[GBS18] Philipp Götze, Stephan Baumann, and Kai-Uwe Sattler. An NVM-Aware Storage Layout for Analytical Workloads. In *34th IEEE International Conference on Data Engineering Workshops, ICDE Workshops 2018, Paris, France, April 16-20, 2018*, pages 110–115. IEEE Computer Society, https://doi.org/10.1109/ICDEW.2018.00025, 2018.

[GDH+20] Gurbinder Gill, Roshan Dathathri, Loc Hoang, Ramesh Peri, and Keshav Pingali. Single Machine Graph Analytics on Massive Datasets Using Intel Optane DC Persistent Memory. *Proc. VLDB Endow.*, 13(8):1304–1318, https://doi.org/10.14778/3389133.3389145, 2020.

[GGK+20] Nikolaus Glombiewski, Philipp Götze, Michael Körber, Andreas Morgen, and Bernhard Seeger. Designing an Event Store for a Modern Three-layer Storage Hierarchy. *Datenbank-Spektrum*, 20:211–222, https://doi.org/10.1007/s13222-020-00356-6, 2020.

[GKC+11] B Govoreanu, GS Kar, YY Chen, V Paraschiv, S Kubicek, A Fantini, IP Radu, L Goux, S Clima, R Degraeve, et al. $10\times10\text{nm}^2$ Hf/$\text{HfO}_x$ Crossbar Resistive RAM with Excellent Performance, Reliability and Low-Energy Operation. In *Electron Devices Meeting (IEDM), 2011 IEEE International*, IEDM, pages 31.36.31—-31.36.34. IEEE, 2011.

[GKP+10] Martin Grund, Jens Krüger, Hasso Plattner, Alexander Zeier, Philippe Cudré-Mauroux, and Samuel Madden. HYRISE - A Main Memory Hybrid Storage Engine. *PVLDB*, 4(2):105–116, https://doi.org/10.14778/1921071.1921077, 2010.

[GM93] Goetz Graefe and William J. McKenna. The Volcano Optimizer Generator: Extensibility and Efficient Search. In *Proceedings of the Ninth International Conference on Data Engineering, April 19-23, 1993, Vienna, Austria*, pages 209–218. IEEE Computer Society, https://doi.org/10.1109/ICDE.1993.344061, 1993.

[GPS19] Philipp Götze, Constantin Pohl, and Kai-Uwe Sattler. Query Planning for Transactional Stream Processing on Heterogeneous Hardware. In Holger Meyer, Norbert Ritter, Andreas Thor, Daniela Nicklas, Andreas Heuer, and Meike Klettke, editors, *Datenbanksysteme für Business, Technologie und Web (BTW 2019), 18. Fachtagung des GI-Fachbereichs „Datenbanken und Informationssysteme" (DBIS), 4.-8. März 2019, Rostock, Germany, Workshopband*, volume

P-290 of *LNI*, pages 71–80. Gesellschaft für Informatik, Bonn, https://doi.org/10.18420/btw2019-ws-05, 2019.

[GS19] Philipp Götze and Kai-Uwe Sattler. Snapshot Isolation for Transactional Stream Processing. In Herschel et al. [HGR+19], https://doi.org/10.5441/002/edbt.2019.78, pages 650–653.

[GS21] Philipp Götze and Kai-Uwe Sattler. Transactional Stream Processing on Persistent Memory. *Inf. Syst. J.*, 2021. submitted in April 2021.

[GSE+94] Jim Gray, Prakash Sundaresan, Susanne Englert, Kenneth Baclawski, and Peter J. Weinberger. Quickly Generating Billion-Record Synthetic Databases. In Richard T. Snodgrass and Marianne Winslett, editors, *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data, Minneapolis, Minnesota, USA, May 24-27, 1994*, pages 243–252. ACM Press, https://doi.org/10.1145/191839.191886, 1994.

[GTS20a] Philipp Götze, Arun Kumar Tharanatha, and Kai-Uwe Sattler. Data Structure Primitives on Persistent Memory: An Evaluation. *CoRR*, abs/2001.02172, http://arxiv.org/abs/2001.02172, 2020.

[GTS20b] Philipp Götze, Arun Kumar Tharanatha, and Kai-Uwe Sattler. Data Structure Primitives on Persistent Memory: An Evaluation. In Danica Porobic and Thomas Neumann, editors, *16th International Workshop on Data Management on New Hardware, DaMoN 2020, Portland, Oregon, USA, June 15, 2020*, pages 15:1–15:3. ACM, https://doi.org/10.1145/3399666.3399900, 2020.

[GvRL+18] Philipp Götze, Alexander van Renen, Lucas Lersch, Viktor Leis, and Ismail Oukid. Data Management on Non-Volatile Memory: A Perspective. *Datenbank-Spektrum*, 18(3):171–182, https://doi.org/10.1007/s13222-018-0301-1, 2018.

[Här84] Theo Härder. Observations on Optimistic Concurrency Control Schemes. *Inf. Syst.*, 9(2):111–120, https://doi.org/10.1016/0306-4379(84)90020-6, 1984.

[HGR+19] Melanie Herschel, Helena Galhardas, Berthold Reinwald, Irini Fundulaki, Carsten Binnig, and Zoi Kaoudi, editors. *Advances in Database Technology - 22nd International Conference on Extending Database Technology, EDBT 2019, Lisbon, Portugal, March 26-29, 2019*. OpenProceedings.org, 2019.

[HHL20] Gabriel Haas, Michael Haubenschild, and Viktor Leis. Exploiting Directly-Attached NVMe Arrays in DBMS. In *CIDR 2020, 10th Conference on Innovative Data Systems Research, Amsterdam, The Netherlands, January 12-15, 2020, Online Proceedings*. www.cidrdb.org, http://cidrdb.org/cidr2020/papers/p16-haas-cidr20.pdf, 2020.

[HKWN18] Deukyeon Hwang, Wook-Hee Kim, Youjip Won, and Beomseok Nam. Endurable Transient Inconsistency in Byte-Addressable Persistent B+-Tree. In Nitin Agrawal and Raju Rangaswami, editors, *16th USENIX Conference on File and Storage Technologies, FAST 2018, Oakland, CA, USA, February 12-15, 2018*, pages 187–200. USENIX Association, https://www.usenix.org/conference/fast18/presentation/hwang, 2018.

[HLN+14] Weiwei Hu, Guoliang Li, Jiacai Ni, Dalie Sun, and Kian-Lee Tan. $\mathsf{B}^p$-Tree : A Predictive $\mathsf{B}^+$-Tree for Reducing Writes on Phase Change Memory. *IEEE Trans. Knowl. Data Eng.*, 26(10):2368–2381, https://doi.org/10.1109/TKDE.2014.5, 2014.

[HSH07] Joseph M. Hellerstein, Michael Stonebraker, and James R. Hamilton. Architecture of a Database System. *Found. Trends Databases*, 1(2):141–259, https://doi.org/10.1561/1900000002, 2007.

[HYY+05] M Hosomi, H Yamagishi, T Yamamoto, K Bessho, Y Higo, K Yamane, H Yamada, M Shoji, H Hachino, C Fukumoto, et al. A Novel Nonvolatile Memory with Spin Torque Transfer Magnetization Switching: Spin-RAM. *IEDM Tech. Dig*, 459, 2005.

[Int16] Intel Corporation. The C++ bindings to libpmemobj. Retrieved June 04, 2021 from https://pmem.io/libpmemobj-cpp/, 2016.

[Int17] Intel Corporation. Intel® SSD DC P4501 Series . Retrieved June 09, 2021 from https://ark.intel.com/content/www/us/en/ark/products/96927/intel-ssd-dc-p4501-series-1-0tb-2-5in-pcie-3-1-x4-3d1-tlc.html, 2017.

[Int19] Intel Corporation. Intel® 64 and IA-32 Architectures Optimization Reference Manual. 2019. Retrieved April 12, 2021 from https://software.intel.com/sites/default/files/managed/9e/bc/64-ia-32-architectures-optimization-manual.pdf, Chapter 11 - Intel® Optane™ DC Persistent Memory.

[Int20a] Intel Corporation. Achieve Greater Insight from Your Data with Intel® Optane™ Persistent Memory. Retrieved April 20, 2021 from https://www.intel.com/content/dam/www/public/us/en/documents/product-briefs/optane-persistent-memory-200-series-brief.pdf, 2020.

[Int20b] Intel Corporation. Intel® Memory Latency Checker v3.9. Retrieved April 12, 2021 from https://software.intel.com/en-us/articles/intelr-memory-latency-checker, 2020.

[Int20c] Intel Corporation. Intel® Optane™ Persistent Memory - Start Up Guide, Revision 2.0. Retrieved April 22, 2021 from https://www.intel.com/content/dam/support/us/en/documents/memory-and-storage/data-center-persistent-mem/Intel_Optane_Persistent_Memory_Start_Up_Guide.pdf, 2020.

[Int20d] Intel Corporation. Persistent Memory Development Kit. Retrieved April 12, 2021 from http://pmem.io/pmdk, 2020.

[Int21a] Intel Corporation. eADR: New Opportunities for Persistent Memory Applications. Retrieved June 03, 2021 from https://software.intel.com/content/www/us/en/develop/articles/eadr-new-opportunities-for-persistent-memory-applications.html, 2021.

[Int21b] Intel Corporation. Intel® Optane™ Persistent Memory 200 Series Brief. Retrieved June 03, 2021 from https://www.intel.com/content/dam/www/public/us/en/documents/product-briefs/optane-persistent-memory-200-series-brief.pdf, 2021.

[Int21c] Intel Corporation. Intel® Transactional Synchronization Extensions (Intel® TSX) Memory and Performance Monitoring Update for Intel® Processors. Retrieved September 7, 2021 from https://www.intel.com/content/www/us/en/support/articles/000059422/processors.html, 2021.

[Int21d] Intel Corporation. Performance Monitoring Impact of Intel® Transactional Synchronization Extension Memory Ordering Issue. Retrieved September 7, 2021 from https:

//www.intel.com/content/dam/support/us/en/documents/processors/Performance-Monitoring-Impact-of-TSX-Memory-Ordering-Issue-604224.pdf, 2021.

[IZA+18] Stratos Idreos, Kostas Zoumpatianos, Manos Athanassoulis, Niv Dayan, Brian Hentschel, Michael S. Kester, Demi Guo, Lukas M. Maas, Wilson Qin, Abdul Wasay, and Yiyou Sun. The Periodic Table of Data Structures. *IEEE Data Eng. Bull.*, 41(3):64–75, http://sites.computer.org/debull/A18sept/p64.pdf, 2018.

[IZH+18] Stratos Idreos, Kostas Zoumpatianos, Brian Hentschel, Michael S. Kester, and Demi Guo. The Data Calculator: Data Structure Design and Cost Synthesis from First Principles and Learned Cost Models. In Das et al. [DJB18], https://doi.org/10.1145/3183713.3199671, pages 535–550.

[JBGS21] Muhammad Attahir Jibril, Alexander Baumstark, Philipp Götze, and Kai-Uwe Sattler. JIT happens: Transactional Graph Processing in Persistent Memory meets Just-In-Time Compilation. In Yannis Velegrakis, Demetris Zeinalipour-Yazti, Panos K. Chrysanthis, and Francesco Guerra, editors, *Proceedings of the 24th International Conference on Extending Database Technology, EDBT 2021, Nicosia, Cyprus, March 23 - 26, 2021*, pages 37–48. OpenProceedings.org, https://doi.org/10.5441/002/edbt.2021.05, 2021.

[JGBS20] Muhammad Attahir Jibril, Philipp Götze, David Broneske, and Kai-Uwe Sattler. Selective Caching: A Persistent Memory Approach for Multi-Dimensional Index Structures. In *36th IEEE International Conference on Data Engineering Workshops, ICDE Workshops 2020, Dallas, TX, USA, April 20-24, 2020*, pages 115–120. IEEE, https://doi.org/10.1109/ICDEW49219.2020.00010, 2020.

[JGBS21] Muhammad Attahir Jibril, Philipp Götze, David Broneske, and Kai-Uwe Sattler. Selective Caching: A Persistent Memory Approach for Multi-Dimensional Index Structures. *Distrib Parallel Databases*, https://doi.org/10.1007/s10619-021-07327-0, 2021.

[JS15] Theodore Johnson and Vladislav Shkapenyuk. Data Stream Warehousing In Tidalrace. http://cidrdb.org/cidr2015/Papers/CIDR15_Paper4.pdf, In *CIDR 2015, Seventh Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 4-7, 2015, Online Proceedings* [DBL15].

[KBG+18] Sudarsun Kannan, Nitish Bhat, Ada Gavrilovska, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Redesigning LSMs for Nonvolatile Memory with NoveLSM. In Haryadi S. Gunawi and Benjamin Reed, editors, *2018 USENIX Annual Technical Conference, USENIX ATC 2018, Boston, MA, USA, July 11-13, 2018*, pages 993–1005. USENIX Association, https://www.usenix.org/conference/atc18/presentation/kannan, 2018.

[KKN+08] Robert Kallman, Hideaki Kimura, Jonathan Natkins, Andrew Pavlo, Alex Rasin, Stanley B. Zdonik, Evan P. C. Jones, Samuel Madden, Michael Stonebraker, Yang Zhang, John Hugg, and Daniel J. Abadi. H-Store: A High-Performance, Distributed Main Memory Transaction Processing System. *Proc. VLDB Endow.*, 1(2):1496–1499, https://doi.org/10.14778/1454159.1454211, 2008.

[KN11] Alfons Kemper and Thomas Neumann. HyPer: A Hybrid OLTP&OLAP Main Memory Database System Based on Virtual Memory Snapshots. In Serge Abiteboul, Klemens Böhm, Christoph Koch, and Kian-Lee Tan, editors, *Proceedings of the 27th International Conference on Data Engineering, ICDE 2011, April 11-16, 2011, Hannover, Germany*, pages 195–206. IEEE Computer Society, https://doi.org/10.1109/ICDE.2011.5767867, 2011.

[KNR+11] Jay Kreps, Neha Narkhede, Jun Rao, et al. Kafka: a Distributed Messaging System for Log Processing. In *Proceedings of the NetDB, Athens, Greece, Jun. 12, 2011*, volume 6, pages 1–7, https://www.microsoft.com/en-us/research/wp-content/uploads/2017/09/Kafka.pdf, 2011.

[KRM+10] Sohrab Kianian, Glen Rosendale, Monte Manning, Darlene Hamilton, XM Henry Huang, Karl Robinson, Young Weon Kim, and Thomas Rueckes. A 3d stackable carbon nanotube-based nonvolatile memory (nram). In *2010 Proceedings of the European Solid State Device Research Conference*, pages 404–407. IEEE, 2010.

[KS04] Andi Kleen and SuSE Labs. numactl(8) - Linux man page. Retrieved June 08, 2021 from https://linux.die.net/man/8/numactl, 2002,2004.

[KSKN18] Wook-Hee Kim, Jihye Seo, Jinwoong Kim, and Beomseok Nam. clfB-tree: Cacheline Friendly Persistent B-tree for NVRAM. *TOS*, 14(1):5:1–5:17, https://doi.org/10.1145/3129263, 2018.

[LCC21] Gang Liu, Leying Chen, and Shimin Chen. Zen: a High-Throughput Log-Free OLTP Engine for Non-Volatile Main Memory. *Proc. VLDB Endow.*, 14(5):835–848, http://www.vldb.org/pvldb/vol14/p835-liu.pdf, 2021.

[LCW20] Jihang Liu, Shimin Chen, and Lujun Wang. LB+-Trees: Optimizing Persistent Index Performance on 3DXPoint Memory. *Proc. VLDB Endow.*, 13(7):1078–1090, https://doi.org/10.14778/3384345.3384355, 2020.

[LHO+19] Lucas Lersch, Xiangpeng Hao, Ismail Oukid, Tianzheng Wang, and Thomas Willhalm. Evaluating persistent memory range indexes. *Proc. VLDB Endow.*, 13(4):574–587, http://www.vldb.org/pvldb/vol13/p574-lersch.pdf, 2019.

[LHWL20] Baotong Lu, Xiangpeng Hao, Tianzheng Wang, and Eric Lo. Dash: Scalable Hashing on Persistent Memory. *Proc. VLDB Endow.*, 13(8):1147–1161, https://doi.org/10.14778/3389133.3389134, 2020.

[Lie20] Leret Liebermann. Eignung von Trie-Datenstrukturen für nicht-flüchtigen RAM. Bachelor thesis, Technische Universität Ilmenau, January 2020.

[Lin18] Linux Kernel Organization, Inc. Ext4 Filesystem. Retrieved April 22, 2021 from https://www.kernel.org/doc/Documentation/filesystems/ext4.txt, 2018.

[Lin19] Linux Kernel Organization, Inc. The SGI XFS Filesystem. Retrieved April 22, 2021 from https://www.kernel.org/doc/Documentation/filesystems/xfs.txt, 2019.

[Lin21] Linux Kernel Organization, Inc. Direct Access for files. Retrieved April 22, 2021 from https://www.kernel.org/doc/Documentation/filesystems/dax.txt, 2021.

[LKN13] Viktor Leis, Alfons Kemper, and Thomas Neumann. The Adaptive Radix Tree: ARTful Indexing for Main-Memory Databases. In *29th IEEE International Conference on Data Engineering, ICDE 2013, Brisbane, Australia, April 8-12, 2013*, pages 38–49, https://doi.org/10.1109/ICDE.2013.6544812, 2013.

[LLO19] Lucas Lersch, Wolfgang Lehner, and Ismail Oukid. Persistent Buffer Management with Optimistic Consistency. In Neumann and Salem [NS19], https://doi.org/10.1145/3329785.3329931, pages 14:1–14:3.

[LLS+15] Justin J. Levandoski, David B. Lomet, Sudipta Sengupta, Ryan Stutsman, and Rui Wang. High Performance Transactions in Deuteronomy. http://cidrdb.org/cidr2015/Papers/CIDR15_Paper15.pdf, In *Seventh Biennial Conference on Innovative Data Systems Research, CIDR 2015, Asilomar, CA, USA, January 4-7, 2015, Online Proceedings* [DBL15].

[LLS+17] Se Kwon Lee, K. Hyun Lim, Hyunsub Song, Beomseok Nam, and Sam H. Noh. WORT: Write Optimal Radix Tree for Persistent Memory Storage Systems. In Geoff Kuenning and Carl A. Waldspurger, editors, *15th USENIX Conference on File and Storage Technologies, FAST 2017, Santa Clara, CA, USA, February 27 - March 2, 2017*, pages 257–270. USENIX Association, https://www.usenix.org/conference/fast17/technical-sessions/presentation/lee-se-kwon, 2017.

[LMF+16] Harald Lang, Tobias Mühlbauer, Florian Funke, Peter A. Boncz, Thomas Neumann, and Alfons Kemper. Data Blocks: Hybrid OLTP and OLAP on Compressed Storage using both Vectorization and Compilation. In Özcan et al. [ÖKM16], https://doi.org/10.1145/2882903.2882925, pages 311–326.

[LMM+13] Juchang Lee, Michael Muehle, Norman May, Franz Faerber, Vishal Sikka, Hasso Plattner, Jens Krüger, and Martin Grund. High-Performance Transaction Processing in SAP HANA. *IEEE Data Eng. Bull.*, 36(2):28–33, http://sites.computer.org/debull/A13june/hana1.pdf, 2013.

[LN96] Sherry Listgarten and Marie-Anne Neimat. Modelling Costs for a MM-DBMS. In *Online-Proceedings of the First International Workshop on Real-Time Databases: Issues and Applications, March 7-8, 1996, Newport Beach, California, USA*, pages 72–78, http://www.eng.uci.edu/faculty/klin/rtdb/LM.ps, 1996.

[LOLS17] Lucas Lersch, Ismail Oukid, Wolfgang Lehner, and Ivan Schreter. An analysis of LSM caching in NVRAM. In *Proceedings of the 13th International Workshop on Data Management on New Hardware, DaMoN 2017, Chicago, IL, USA, May 15, 2017*, pages 9:1–9:5. ACM, https://doi.org/10.1145/3076113.3076123, 2017.

[LPD17] Jianhong Li, Andy Pavlo, and Siying Dong. NVMRocks: RocksDB on Non-Volatile Memory Systems. Retrieved April 12, 2021 from https://web.archive.org/web/20200217045318/istc-bigdata.org/index.php/nvmrocks-rocksdb-on-non-volatile-memory-systems/, 2017.

[LS79] Butler Lampson and Howard E. Sturgis. Crash recovery in a distributed data storage system. This unpublished paper was widely circulated in samizdat., June 1979.

[LZY+10] Benjamin C Lee, Ping Zhou, Jun Yang, Youtao Zhang, Bo Zhao, Engin Ipek, Onur Mutlu, and Doug Burger. Phase-Change Technology and the Future of Main Memory. *IEEE Micro*, 30(1), 2010.

[Man02] Stefan Manegold. *Understanding, Modeling, and Improving Main-Memory Database Performance.* PhD thesis, Universiteit van Amsterdam, 2002.

[MG11] Zoltan Majo and Thomas R. Gross. Memory System Performance in a NUMA Multicore Multiprocessor. In Paula Ta-Shma, José Moreira, and Liuba Shrira, editors, *Proceedings of of SYSTOR 2011: The 4th Annual Haifa Experimental Systems Conference, Haifa, Israel, May 30 - June 1, 2011*, page 12. ACM, https://doi.org/10.1145/1987816.1987832, 2011.

[Moe98] Guido Moerkotte. Small Materialized Aggregates: A Light Weight Index Structure for Data Warehousing. In Ashish Gupta, Oded Shmueli, and Jennifer Widom, editors, *VLDB'98,*

*Proceedings of 24rd International Conference on Very Large Data Bases, August 24-27, 1998, New York City, New York, USA*, pages 476–487. Morgan Kaufmann, http://www.vldb.org/conf/1998/p476.pdf, 1998.

[MRM+17] Barzan Mozafari, Jags Ramnarayan, Sudhir Menon, Yogesh Mahajan, Soubhik Chakraborty, Hemant Bhanawat, and Kishor Bachhav. SnappyData: A Unified Cluster for Streaming, Transactions and Interactice Analytics. In *8th Biennial Conference on Innovative Data Systems Research, CIDR 2017, Chaminade, CA, USA, January 8-11, 2017, Online Proceedings*. www.cidrdb.org, http://cidrdb.org/cidr2017/papers/p28-mozafari-cidr17.pdf, 2017.

[MT20] Inc. Micron Technology. 3D XPoint Technology. Retrieved April 12, 2021 from https://www.micron.com/products/advanced-solutions/3d-xpoint-technology, 2020.

[MTA09] René Müller, Jens Teubner, and Gustavo Alonso. Data Processing on FPGAs. *Proc. VLDB Endow.*, 2(1):910–921, https://doi.org/10.14778/1687627.1687730, 2009.

[MTZ+15] John Meehan, Nesime Tatbul, Stan Zdonik, Cansu Aslantas, Ugur Çetintemel, Jiang Du, Tim Kraska, Samuel Madden, David Maier, Andrew Pavlo, Michael Stonebraker, Kristin Tufte, and Hao Wang. S-Store: Streaming Meets Transaction Processing. *Proc. VLDB Endow.*, 8(13):2134–2145, https://doi.org/10.14778/2831360.2831367, 2015.

[MWA+03] Rajeev Motwani, Jennifer Widom, Arvind Arasu, Brian Babcock, Shivnath Babu, Mayur Datar, Gurmeet Singh Manku, Chris Olston, Justin Rosenstein, and Rohit Varma. Query Processing, Approximation, and Resource Management in a Data Stream Management System. In *CIDR 2003, First Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 5-8, 2003, Online Proceedings*. www.cidrdb.org, http://www-db.cs.wisc.edu/cidr/cidr2003/program/p22.pdf, 2003.

[NCC+19] Moohyeon Nam, Hokeun Cha, Young-ri Choi, Sam H. Noh, and Beomseok Nam. Write-Optimized Dynamic Hashing for Persistent Memory. In Arif Merchant and Hakim Weatherspoon, editors, *17th USENIX Conference on File and Storage Technologies, FAST 2019, Boston, MA, February 25-28, 2019*, pages 31–44. USENIX Association, https://www.usenix.org/conference/fast19/presentation/nam, 2019.

[NH12] Dushyanth Narayanan and Orion Hodson. Whole-System Persistence. In Tim Harris and Michael L. Scott, editors, *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2012, London, UK, March 3-7, 2012*, pages 401–410. ACM, https://doi.org/10.1145/2150976.2151018, 2012.

[NMK15] Thomas Neumann, Tobias Mühlbauer, and Alfons Kemper. Fast Serializable Multi-Version Concurrency Control for Main-Memory Database Systems. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, pages 677–689, https://doi.org/10.1145/2723372.2749436, 2015.

[NS19] Thomas Neumann and Ken Salem, editors. *Proceedings of the 15th International Workshop on Data Management on New Hardware, DaMoN 2019, Amsterdam, The Netherlands, 1 July 2019*. ACM, 2019.

[OBGK18] Jennifer Ortiz, Magdalena Balazinska, Johannes Gehrke, and S. Sathiya Keerthi. Learning State Representations for Query Optimization with Deep Reinforcement Learning. In Sebastian Schelter, Stephan Seufert, and Arun Kumar, editors, *Proceedings of the Second Workshop on Data Management for End-To-End Machine Learning, DEEM@SIGMOD 2018,*

*Houston, TX, USA, June 15, 2018*, pages 4:1–4:4. ACM, https://doi.org/10.1145/3209889.3209890, 2018.

[OBL+14] Ismail Oukid, Daniel Booss, Wolfgang Lehner, Peter Bumbulis, and Thomas Willhalm. SOFORT: A Hybrid SCM-DRAM Storage Engine for Fast Data Recovery. In Alfons Kemper and Ippokratis Pandis, editors, *Tenth International Workshop on Data Management on New Hardware, DaMoN 2014, Snowbird, UT, USA, June 23, 2014*, pages 8:1–8:7. ACM, https://doi.org/10.1145/2619228.2619236, 2014.

[OBL+17] Ismail Oukid, Daniel Booss, Adrien Lespinasse, Wolfgang Lehner, Thomas Willhalm, and Grégoire Gomes. Memory management techniques for large-scale persistent-main-memory systems. *Proc. VLDB Endow.*, 10(11):1166–1177, https://doi.org/10.14778/3137628.3137629, 2017.

[OCGO96] Patrick E. O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth J. O'Neil. The Log-Structured Merge-Tree (LSM-Tree). *Acta Inf.*, 33(4):351–385, https://doi.org/10.1007/s002360050048, 1996.

[ÖKM16] Fatma Özcan, Georgia Koutrika, and Sam Madden, editors. *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*. ACM, 2016.

[OKW17] Ismail Oukid, Robert Kettler, and Thomas Willhalm. Storage class memory and databases: Opportunities and challenges. *it Inf. Technol.*, 59(3):109, https://doi.org/10.1515/itit-2016-0052, 2017.

[OLN+16] Ismail Oukid, Johan Lasperas, Anisoara Nica, Thomas Willhalm, and Wolfgang Lehner. FPTree: A Hybrid SCM-DRAM Persistent and Concurrent B-Tree for Storage Class Memory. In Özcan et al. [ÖKM16], https://doi.org/10.1145/2882903.2915251, pages 371–386.

[OSL16] Jiaxin Ou, Jiwu Shu, and Youyou Lu. A High Performance File System for Non-Volatile Main Memory. In *Proceedings of the Eleventh European Conference on Computer Systems, EuroSys 2016, London, United Kingdom, April 18-21, 2016*, pages 12:1–12:16, https://doi.org/10.1145/2901318.2901324, 2016.

[PA16] Andrew Pavlo and Matthew Aslett. What's Really New with NewSQL? *SIGMOD Rec.*, 45(2):45–55, https://doi.org/10.1145/3003665.3003674, 2016.

[PGS17] Constantin Pohl, Philipp Götze, and Kai-Uwe Sattler. A cost model for data stream processing on modern hardware. In Rajesh Bordawekar and Tirthankar Lahiri, editors, *International Workshop on Accelerating Analytics and Data Management Systems Using Modern Processor and Storage Architectures, ADMS@VLDB 2017, Munich, Germany, September 1, 2017*, http://www.adms-conf.org/2017/camera-ready/adms2017_final.pdf, 2017.

[Pro12] Adrian Proctor. NV-DIMMs - The Fastest Tier in Your Storage Strategy. Retrieved April 12, 2021 from https://docplayer.net/13101700-Nv-dimm-fastest-tier-in-your-storage-strategy.html, 2012.

[QGR11] Moinuddin K Qureshi, Sudhanva Gurumurthi, and Bipin Rajendran. Phase Change Memory: From Devices to Systems. *Synthesis Lectures on Computer Architecture*, 6(4):1–134, https://doi.org/10.2200/s00381ed1v01y201109cac018, 2011.

[Ree83] David P. Reed. Implementing Atomic Actions on Decentralized Data. *ACM Trans. Comput. Syst.*, 1(1):3–23, https://doi.org/10.1145/357353.357355, 1983.

[RKJ+00] Thomas Rueckes, Kyoungha Kim, Ernesto Joselevich, Greg Y. Tseng, Chin-Li Cheung, and Charles M. Lieber. Carbon nanotube-based nonvolatile random access memory for molecular computing. *Science*, 289(5476):94–97, https://doi.org/10.1126/science.289.5476.94, 2000.

[RKK+14] Dulloor Subramanya Rao, Sanjay Kumar, Anil S. Keshavamurthy, Philip Lantz, Dheeraj Reddy, Rajesh Sankaran, and Jeff Jackson. System Software for Persistent Memory. In *Ninth Eurosys Conference 2014, EuroSys 2014, Amsterdam, The Netherlands, April 13-16, 2014*, pages 15:1–15:15, https://doi.org/10.1145/2592798.2592814, 2014.

[RR00] Jun Rao and Kenneth A. Ross. Making $B^+$-Trees Cache Conscious in Main Memory. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, May 16-18, 2000, Dallas, Texas, USA*, pages 475–486, https://doi.org/10.1145/342009.335449, 2000.

[RSI78] Daniel J. Rosenkrantz, Richard Edwin Stearns, and Philip M. Lewis II. System Level Concurrency Control for Distributed Database Systems. *ACM Trans. Database Syst.*, 3(2):178–198, https://doi.org/10.1145/320251.320260, 1978.

[SAJA09] Radu Stoica, Manos Athanassoulis, Ryan Johnson, and Anastasia Ailamaki. Evaluating and Repairing Write Performance on Flash Devices. In Peter A. Boncz and Kenneth A. Ross, editors, *Proceedings of the Fifth International Workshop on Data Management on New Hardware, DaMoN 2009, Providence, Rhode Island, USA, June 28, 2009*, pages 9–14. ACM, https://doi.org/10.1145/1565694.1565697, 2009.

[Sca20] Steve Scargall. *Programming Persistent Memory*. Apress, 2020.

[SDUP15] David Schwalb, Markus Dreseler, Matthias Uflacker, and Hasso Plattner. NVC-Hashmap: A Persistent and Concurrent Hashmap For Non-Volatile Memories. In *Proceedings of the 3rd VLDB Workshop on In-Memory Data Mangement and Analytics, IMDM@VLDB 2015, Kohala Coast, HI, USA, August 31, 2015*, pages 4:1–4:8. ACM, https://doi.org/10.1145/2803140.2803144, 2015.

[SGKS19] Marc Seidemann, Nikolaus Glombiewski, Michael Körber, and Bernhard Seeger. ChronicleDB: A High-Performance Event Store. *ACM Trans. Database Syst.*, 44(4):13:1–13:45, https://doi.org/10.1145/3342357, 2019.

[Sin20] SingleStore Inc. SingleStore is The Database of Now™ Powering Modern Applications and Analytical Systems. 2020. Retrieved April 12, 2021 from https://www.singlestore.com/, previously MemSQL.

[SNI17] SNIA. NVM Programming Model (NPM) - Version 1.2. Retrieved April 21, 2021 from https://www.snia.org/sites/default/files/technical_work/final/NVMProgrammingModel_v1.2.pdf, 2017.

[SR86] Michael Stonebraker and Lawrence A. Rowe. The Design of Postgres. In Carlo Zaniolo, editor, *Proceedings of the 1986 ACM SIGMOD International Conference on Management of Data, Washington, DC, USA, May 28-30, 1986*, pages 340–355. ACM Press, https://doi.org/10.1145/16894.16888, 1986.

[SS06] Ori Shalev and Nir Shavit. Split-Ordered Lists: Lock-Free Extensible Hash Tables. *J. ACM*, 53(3):379–405, https://doi.org/10.1145/1147954.1147958, 2006.

[SS17] Marc Seidemann and Bernhard Seeger. ChronicleDB: A High-Performance Event Store. In *Proceedings of the 20th International Conference on Extending Database Technology, EDBT 2017, Venice, Italy, March 21-24, 2017*, pages 144–155, https://doi.org/10.5441/002/edbt.2017.14, 2017.

[SSSW08] Dmitri B. Strukov, Gregory S. Snider, Duncan R. Stewart, and R. Stanley Williams. The missing memristor found. *Nature*, 453(7191):80–83, http://dx.doi.org/10.1038/nature06932, 2008.

[Tha19] Arun Kumar Tharanatha. Designing an LSM-Tree Optimized for Persistent Memory. Master thesis, Technische Universität Ilmenau, July 2019.

[The16] The Apache Software Foundation. Apache Cassandra. 2016.

[THSW15] Kanat Tangwongsan, Martin Hirzel, Scott Schneider, and Kun-Lung Wu. General Incremental Sliding-Window Aggregation. *Proc. VLDB Endow.*, 8(7):702–713, https://doi.org/10.14778/2752939.2752940, 2015.

[TMSF03] Peter A. Tucker, David Maier, Tim Sheard, and Leonidas Fegaras. Exploiting Punctuation Semantics in Continuous Data Streams. *IEEE Trans. Knowl. Data Eng.*, 15(3):555–568, https://doi.org/10.1109/TKDE.2003.1198390, 2003.

[Vik20] Viking Technology. DDR4 NVDIMM-N. Retrieved April 12, 2021 from https://www.vikingtechnology.com/wp-content/uploads/VikingTechnology_NVDIMM_ProductBrief.pdf, 2020.

[vRLK+18] Alexander van Renen, Viktor Leis, Alfons Kemper, Thomas Neumann, Takushi Hashida, Kazuichi Oe, Yoshiyasu Doi, Lilian Harada, and Mitsuru Sato. Managing Non-Volatile Memory in Database Systems. In Das et al. [DJB18], https://doi.org/10.1145/3183713.3196897, pages 1541–1555.

[vRVL+19] Alexander van Renen, Lukas Vogel, Viktor Leis, Thomas Neumann, and Alfons Kemper. Persistent Memory I/O Primitives. In Neumann and Salem [NS19], https://doi.org/10.1145/3329785.3329930, pages 12:1–12:7.

[VTRC11] Shivaram Venkataraman, Niraj Tolia, Parthasarathy Ranganathan, and Roy H. Campbell. Consistent and Durable Data Structures for Non-Volatile Byte-Addressable Memory. In Gregory R. Ganger and John Wilkes, editors, *9th USENIX Conference on File and Storage Technologies, San Jose, CA, USA, February 15-17, 2011*, pages 61–75. USENIX, http://www.usenix.org/events/fast11/tech/techAbstracts.html#Venkataraman, 2011.

[WAL+17] Yingjun Wu, Joy Arulraj, Jiexi Lin, Ran Xian, and Andrew Pavlo. An Empirical Evaluation of In-Memory Multi-Version Concurrency Control. *Proc. VLDB Endow.*, 10(7):781–792, https://doi.org/10.14778/3067421.3067427, 2017.

[WJ14] Tianzheng Wang and Ryan Johnson. Scalable Logging through Emerging Non-Volatile Memory. *Proc. VLDB Endow.*, 7(10):865–876, https://doi.org/10.14778/2732951.2732960, 2014.

[WK90] Kyu-Young Whang and Ravi Krishnamurthy. Query Optimization in a Memory-Resident Domain Relational Calculus Database System. *ACM Trans. Database Syst.*, 15(1):67–95, https://doi.org/10.1145/77643.77646, 1990.

[WLL18] Tianzheng Wang, Justin J. Levandoski, and Per-Åke Larson. Easy Lock-Free Indexing in Non-Volatile Memory. In *34th IEEE International Conference on Data Engineering, ICDE 2018, Paris, France, April 16-19, 2018*, pages 461–472. IEEE Computer Society, https://doi.org/10.1109/ICDE.2018.00049, 2018.

[WQR13] XiaoJian Wu, Sheng Qiu, and A. L. Narasimha Reddy. SCMFS: A File System for Storage Class Memory and its Extensions. *ACM Trans. Storage*, 9(3):7:1–7:23, https://doi.org/10.1145/2501620.2501621, 2013.

[WR11] XiaoJian Wu and A. L. Narasimha Reddy. SCMFS: A File System for Storage Class Memory. In *Conference on High Performance Computing Networking, Storage and Analysis, SC 2011, Seattle, WA, USA, November 12-18, 2011*, pages 39:1–39:11, https://doi.org/10.1145/2063384.2063436, 2011.

[WV02] Gerhard Weikum and Gottfried Vossen. *Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery.* Morgan Kaufmann, 2002.

[XJXS17] Fei Xia, Dejun Jiang, Jin Xiong, and Ninghui Sun. HiKV: A Hybrid Index Key-Value Store for DRAM-NVM Memory Systems. In Dilma Da Silva and Bryan Ford, editors, *2017 USENIX Annual Technical Conference, USENIX ATC 2017, Santa Clara, CA, USA, July 12-14, 2017*, pages 349–362. USENIX Association, https://www.usenix.org/conference/atc17/technical-sessions/presentation/xia, 2017.

[XS16a] Jian Xu and Steven Swanson. NOVA: A Log-structured File System for Hybrid Volatile/Non-volatile Main Memories. In *14th USENIX Conference on File and Storage Technologies, FAST 2016, Santa Clara, CA, USA, February 22-25, 2016*, pages 323–338, https://www.usenix.org/conference/fast16/technical-sessions/presentation/xu, 2016.

[XS16b] Jian Xu and Steven Swanson. NOVA: A Log-Structured File System for Hybrid Volatile/Non-Volatile Main Memories. *login Usenix Mag.*, 41(3), https://www.usenix.org/publications/login/fall2016/xu, 2016.

[YKH+20] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steven Swanson. An Empirical Guide to the Behavior and Use of Scalable Persistent Memory. In Sam H. Noh and Brent Welch, editors, *18th USENIX Conference on File and Storage Technologies, FAST 2020, Santa Clara, CA, USA, February 24-27, 2020*, pages 169–182. USENIX Association, https://www.usenix.org/conference/fast20/presentation/yang, 2020.

[YWC+15] Jun Yang, Qingsong Wei, Cheng Chen, Chundong Wang, Khai Leong Yong, and Bingsheng He. NV-Tree: Reducing Consistency Cost for NVM-based Single Level Systems. In Jiri Schindler and Erez Zadok, editors, *Proceedings of the 13th USENIX Conference on File and Storage Technologies, FAST 2015, Santa Clara, CA, USA, February 16-19, 2015*, pages 167–181. USENIX Association, https://www.usenix.org/conference/fast15/technical-sessions/presentation/yang, 2015.

[ZDI14] Haopeng Zhang, Yanlei Diao, and Neil Immerman. On complexity and optimization of expensive queries in complex event processing. In Curtis E. Dyreson, Feifei Li, and M. Tamer Özsu, editors, *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*, pages 217–228. ACM, https://doi.org/10.1145/2588555.2593671, 2014.

[ZDK+21] Mikhail Zarubin, Patrick Damme, Alexander Krause, Dirk Habich, and Wolfgang Lehner. SIMD-MIMD Cocktail in a Hybrid Memory Glass: Shaken, not Stirred. In Bruno Wassermann, Michal Malka, Vijay Chidambaram, and Danny Raz, editors, *SYSTOR '21: The 14th*

*ACM International Systems and Storage Conference, Haifa, Israel, June 14-16, 2021*, pages 17:1–17:12. ACM, https://doi.org/10.1145/3456727.3463782, 2021.

[Zeu18] Steffen Zeuch. *Query Execution on Modern CPUs*. PhD thesis, Humboldt University of Berlin, Germany, 2018.

[ZHW18] Pengfei Zuo, Yu Hua, and Jie Wu. Write-Optimized and High-Performance Hashing Index Scheme for Persistent Memory. In Andrea C. Arpaci-Dusseau and Geoff Voelker, editors, *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018*, pages 461–476. USENIX Association, https://www.usenix.org/conference/osdi18/presentation/zuo, 2018.

[ZSC+19] Xinjing Zhou, Lidan Shou, Ke Chen, Wei Hu, and Gang Chen. DPTree: Differential Indexing for Persistent Memory. *Proc. VLDB Endow.*, 13(4):421–434, https://doi.org/10.14778/3372716.3372717, 2019.

[ZWZH20] Shuhao Zhang, Yingjun Wu, Feng Zhang, and Bingsheng He. Towards Concurrent Stateful Stream Processing on Multicore Processors. In *36th IEEE International Conference on Data Engineering, ICDE 2020, Dallas, TX, USA, April 20-24, 2020*, pages 1537–1548. IEEE, https://doi.org/10.1109/ICDE48307.2020.00136, 2020.

# Acronyms

**2PC** Two-Phase Commit. 105, 124, 125, 131, 141, 144

**ACID** Atomicity, Consistency, Isolation, Durability. 2, 6, 16, 89–91, 97–99, 103, 104, 107, 110, 112, 141, 144
**ART** Adaptive Radix Tree. 29, 36
**ATT** Address Translation Tree. 116, 117, 120, 121, 134

**BDCC** Bitwise Dimensional Co-Clustering. 67–73, 140
**BOCC** Backward-oriented Optimistic Concurrency Control. 122–124, 127–129, 131

**CAS** Compare-and-Swap. 16, 43, 101
**CC** Concurrency Control. 98–100, 126, 129–131, 136
**clwb** cache line write back. 19, 106, 128, 143
**CTS** Commit Timestamp. 100, 108

**DAX** Direct Access. 12
**DBMS** Database Management System. 1, 31, 64, 92, 95, 97, 98, 100, 108, 111–114, 136
**DCPMM** Data Center Persistent Memory Module. 11, 12, 14, 15, 22, 26, 50, 52, 61–63, 66, 81, 120, 139
**DTS** Deletion Timestamp. 100

**eADR** extended Asynchronous DRAM Refresh. 15, 143

**FIFO** First In, First Out. 79
**FOCC** Forward-oriented Optimistic Concurrency Control. 122

**HBM** High Bandwidth Memory. 109, 111
**HTAP** Hybrid Transactional/Analytical Processing. 1, 66
**HTM** Hardware Transactional Memory. 143

**IoT** Internet of Things. 1

**LFU** Least Frequently Used. 79, 82
**LLA** Least Likely to be Accessed. 79, 82–84, 88, 141
**LRU** Least Recently Used. 79, 82

**LSM-Tree** Log-Structured Merge-Tree. 26, 28, 29, 31, 34, 37, 41, 42, 53, 54, 64, 79, 116, 136, 140

**MVCC** Multi-Version Concurrency Control. 66, 91, 92, 98–103, 122–131, 136, 141

**NRAM** Nanotube RAM. 11
**NUMA** Non-Uniform Memory Access. 22, 96, 97, 127, 128, 130, 137, 144

**OLAP** Online Analytical Processing. 65, 66, 68, 88, 114
**OLTP** Online Transaction Processing. 5, 14, 65, 66, 88, 90, 94, 97
**OOO** Out-of-Order. 94, 117, 121, 132, 134–136

**PCM** Phase-Change Memory. 10, 11, 29
**PMDK** Persistent Memory Development Kit. 19–21, 23, 32, 40, 43–46, 49, 53–55, 63, 77, 107, 123, 127, 131, 132
**PMem** Persistent Memory. III, V, 2–20, 22–36, 38–55, 57, 59–73, 75–85, 87–92, 96, 99–101, 103, 104, 106, 107, 109, 111, 112, 114–124, 126, 127, 130–137, 139–144
**PMwCAS** persistent multi-word compare-and-swap. 28, 40, 43, 44, 46

**RDMA** Remote Direct Memory Access. 144
**RRAM** Resistive RAM. 10

**S2PL** Strict Two-Phase Locking. 122–124, 127–129, 131
**SIMD** Single Instruction, Multiple Data. 109, 143
**SMAs** Small Materialized Aggregates. 69, 71, 115, 133, 140
**SNIA** Storage Networking Industry Association. 12
**STT-MRAM** Spin Transfer Torque Magnetic RAM. 11

**TAB$^+$-Tree** Temporal Aggregated B$^+$-Tree. 115–121, 132–136
**TSP** Transactional Stream Processing. III, V, 3, 4, 6–8, 65, 67, 72, 88, 90–92, 94, 95, 97, 99, 100, 108, 111–114, 122, 130, 136, 141, 143

# List of Figures

# List of Tables

# List of Algorithms

# Erklärung

Ich versichere, dass ich die vorliegende Arbeit ohne unzulässige Hilfe Dritter und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe. Die aus anderen Quellen direkt oder indirekt übernommenen Daten und Konzepte sind unter Angabe der Quelle gekennzeichnet.

An der inhaltlich-materiellen Erstellung der vorliegenden Arbeit waren keine weiteren als die an den entsprechenden Stellen genannten oder zitierten Personen beteiligt. Insbesondere habe ich hierfür nicht die entgeltliche Hilfe von Vermittlungs- bzw. Beratungsdiensten (Promotionsberater oder anderer Personen) in Anspruch genommen. Niemand hat von mir unmittelbar oder mittelbar geldwerte Leistungen für Arbeiten erhalten, die im Zusammenhang mit dem Inhalt der vorgelegten Dissertation stehen.

Die Arbeit wurde bisher weder im In- noch im Ausland in gleicher oder ähnlicher Form einer Prüfungsbehörde vorgelegt.

Ich bin darauf hingewiesen worden, dass die Unrichtigkeit der vorstehenden Erklärung als Täuschungsversuch bewertet wird und gemäß §7 Abs. 10 der Promotionsordnung den Abbruch des Promotionsverfahrens zur Folge hat.

Ilmenau, April 25, 2022 Philipp Götze