Dissertation zur Erlangung des
akademischen Grades eines
Doktors der Ingenieurwissenschaften (Dr. Ing.)
in der Wissenschaftsdisziplin
Praktische Informatik

Automatic Tiering for In-Memory Database Systems

Markus Dreseler

October 29, 2021

Hasso-Plattner-Institut für Digital Engineering
Enterprise Platform and Integration Concepts

August-Bebel-Str. 88
14482 Potsdam, Germany

Supervisors:
Prof. Dr. h.c. mult. Hasso Plattner
Prof. Dr. Volker Markl
Prof. Dr. Carsten Binnig

Date of final exam: 09.06.2022

Published online on the
Publication Server of the University of Potsdam:
https://doi.org/10.25932/publishup-55825
https://nbn-resolving.org/urn:nbn:de:kobv:517-opus4-558253

Abstract

A decade ago, it became feasible to store multi-terabyte databases in main memory. These in-memory databases (IMDBs) profit from DRAM's low latency and high throughput as well as from the removal of costly abstractions used in disk-based systems, such as the buffer cache. However, as the DRAM technology approaches physical limits, scaling these databases becomes difficult. Non-volatile memory (NVM) addresses this challenge. This new type of memory is persistent, has more capacity than DRAM (4×), and does not suffer from its density-inhibiting limitations. Yet, as NVM has a higher latency (5-15×) and a lower throughput (0.35×), it cannot fully replace DRAM.

IMDBs thus need to navigate the trade-off between the two memory tiers. We present a solution to this optimization problem. Leveraging information about access frequencies and patterns, our solution utilizes NVM's additional capacity while minimizing the associated access costs. Unlike buffer cache-based implementations, our tiering abstraction does not add any costs when reading data from DRAM. As such, it can act as a drop-in replacement for existing IMDBs. Our contributions are as follows:

(1) As the foundation for our research, we present Hyrise, an open-source, columnar IMDB that we re-engineered and re-wrote from scratch. Hyrise enables realistic end-to-end benchmarks of SQL workloads and offers query performance which is competitive with other research and commercial systems. At the same time, Hyrise is easy to understand and modify as repeatedly demonstrated by its uses in research and teaching.

(2) We present a novel memory management framework for different memory and storage tiers. By encapsulating the allocation and access methods of these tiers, we enable existing data structures to be stored on different tiers with no modifications to their implementation. Besides DRAM and NVM, we also support and evaluate SSDs and have made provisions for upcoming technologies such as disaggregated memory.

(3) To identify the parts of the data that can be moved to (s)lower tiers with little performance impact, we present a tracking method that identifies access skew both in the row and column dimensions and that detects patterns within consecutive accesses. Unlike existing methods that have substantial associated costs, our access counters exhibit no identifiable overhead in standard benchmarks despite their increased accuracy.

(4) Finally, we introduce a tiering algorithm that optimizes the data placement for a given memory budget. In the TPC-H benchmark, this allows us to move 90% of the data to NVM while the throughput is reduced by only 10.8% and the query latency is increased by 11.6%. With this, we outperform approaches that ignore the workload's access skew and access patterns and increase the query latency by 20% or more.

Individually, our contributions provide novel approaches to current challenges in systems engineering and database research. Combining them allows IMDBs to scale past the limits of DRAM while continuing to profit from the benefits of in-memory computing.

Acknowledgments

The outstanding teamwork that I was allowed to experience during the work on Hyrise is something that I will always be thankful for. This includes my collaborators and fellow PhD students Martin Boissier, Stefan Halfpap, and Jan Kossmann as well as all students who have contributed to Hyrise as part of the *Develop your own Database* seminar, their Master's theses, or as student assistants.

I am grateful to Prof. Hasso Plattner, who has not only created HPI as the institution at which I enjoyed a one-in-a-kind computer science education, but who has also formed a research group in which I could follow my interests, gained valuable experience, and grew as a person. Both he and his chair representatives, Dr. Matthias Uflacker and Dr. Michael Perscheid, provided helpful guidance and support for this thesis as well as for my overall PhD journey.

My sincere thanks also go to SAP for funding my research as well as for our regular exchange of ideas and for introducing me to the industry side of database development.

My time in Potsdam would not have been as enjoyable if it had not been for the collaboration and friendship in our research group. Thank you for the enjoyable years!

I thank Lawrence Benson, Michael Brendle, Martin Boissier, Jan Kossmann, Dr. Michael Perscheid, Dr. Daniel Ritter, Dr. Rainer Schlosser, and Dr. Ralf Teusner for their helpful feedback on different parts of this work.

Also, I thank Dr. Martin Grund and David Schwalb, with whom I had the pleasure of working on the first version of Hyrise and who inspired me to embark on this journey.

Finally, I wish to thank my parents Lothar and Beate and my sister Christiane for their encouragement and unwavering support.

Zusammenfassung

Seit etwa einem Jahrzehnt können Datenbanken mit einer Größe von mehreren Terabytes im Hauptspeicher abgelegt werden. Diese Hauptspeicherdatenbanken (*In-Memory Databases*) profitieren einerseits von der niedrigen Latenz und dem hohen Durchsatz von DRAM und andererseits vom Fehlen teurer Abstraktionsschichten, wie dem *Buffer Cache*, welcher in Festplatten-basierten Datenbanksystemen von Nöten war. Dadurch, dass die Entwicklung der DRAM-Technologie mehr und mehr auf physikalische Grenzen stößt, wird es jedoch zunehmend schwierig, Hauptspeicherdatenbanken zu skalieren. *Non-volatile Memory* (NVM) adressiert diese Herausforderung. Dieser neue Speichertyp ist persistent, hat eine um einen Faktor 4 höhere Kapazität als DRAM und leidet nicht unter den Einschränkungen, welche die Erhöhung der Speicherdichte von DRAM limitieren. Da NVM jedoch eine höhere Latenz (5-15×) und einen niedrigeren Durchsatz (0.35×) aufweist als DRAM, kann es DRAM noch nicht vollständig ersetzen.

Bei der Entwicklung von Hauptspeicherdatenbanken muss daher der Zielkonflikt zwischen den beiden Speichertypen ausbalanciert werden. Die vorliegende Arbeit präsentiert eine Lösung für dieses Optimierungsproblem. Indem wir Informationen zu Zugriffshäufigkeiten und -mustern auswerten, können wir die zusätzliche Kapazität von NVM ausnutzen und gleichzeitig die mit NVM verbundene Erhöhung von Zugriffskosten minimieren. Anders als bei bestehenden Ansätzen, welche auf einen Buffer Cache aufsetzen, bleiben bei unserer Ansatz die Kosten von Zugriffen auf DRAM unverändert. Dadurch kann unsere Lösung als unmittelbarer Ersatz für existierende Hauptspeicherdatenbanken genutzt werden. Unsere Arbeit leistet hierfür die folgenden Beiträge:

(1) Als Grundlage für unsere Forschung präsentieren wir Hyrise, eine quelloffene, spaltenorientierte Hauptspeicherdatenbank, welche wir von Grund auf neu entwickelt haben. Hyrise ermöglicht realistische *End-to-End Benchmarks* von SQL Workloads und weist dabei eine Performance auf, welche mit anderen Datenbanksystemen aus Industrie und Forschung vergleichbar ist. Hierbei ist Hyrise leicht zu verstehen und modifizieren. Dies wurde durch den wiederholten Einsatz in Forschung und Lehre demonstriert.

(2) Wir präsentieren ein neuartiges Speicherverwaltungs-Framework, welches verschiedene Speicherebenen (*Tiers*) unterstützt. Indem wir die Allokations- und Zugriffsmethoden dieser Speicherebenen kapseln, ermöglichen wir es, bestehende Datenstrukturen auf diese Ebenen aufzuteilen ohne ihre Implementierung anpassen zu müssen. Neben DRAM und NVM unterstützt unser Ansatz SSDs und ist auf zukünftige Technologien wie Disaggregated Memory vorbereitet.

(3) Um jene Teile der Daten zu identifizieren, welche auf langsamere Ebenen verschoben werden können, ohne dass die Performance des Systems als Ganzes negativ beeinträchtigt wird, stellen wir mit unseren *Access Countern* eine Tracking-Methode vor, welche ungleich verteilte Zugriffshäufigkeiten sowohl in der Zeilen- als auch in der

Spaltendimension erkennt. Ebenfalls erkennt die Tracking-Methode Zugriffsmuster in aufeinanderfolgenden Zugriffsoperationen. Trotz ihrer hohen Genauigkeit weisen unsere Access Counter keine messbaren Mehrkosten auf. Dies unterscheidet sie von bestehenden Ansätzen, welche ungleichverteilte Zugriffsmuster weniger gut erkennen, gleichzeitig aber Mehrkosten von 20% verursachen.

(4) Abschließend stellen wir einen *Tiering*-Algorithmus vor, welcher die Verteilung von Daten auf die verschiedenen Speicherebenen optimiert. Am Beispiel des TPC-H-Benchmarks zeigen wir, wie 90% der Daten auf NVM verschoben werden können, wobei der Durchsatz nur um 10.8% reduziert und die durchschnittliche Antwortzeit um 11.6% erhöht wird. Damit übertreffen wir Ansätze, welche Ungleichverteilungen in den Zugriffshäufigkeiten und -mustern ignorieren.

Einzeln betrachtet stellen unsere Beiträge neue Herangehensweisen für aktuelle Herausforderungen in der systemnahen Entwicklung und der Datenbankforschung dar. In ihrem Zusammenspiel ermöglichen sie es, Hauptspeicherdatenbanken über die Grenzen von DRAM hinaus zu skalieren und dabei weiterhin von den Vorteilen des *In-Memory Computings* zu profitieren.

Contents

1 Introduction

In this chapter, we present the motivation for our research on automatic tiering for in-memory databases and state the research questions that are answered by this thesis. Furthermore, we explicitly list our contributions, describe how these advance the current state of the art and discuss how they can be applied beyond the scope of automatic tiering. Finally, we define the scope of this thesis and explain its structure.

1.1 Motivation

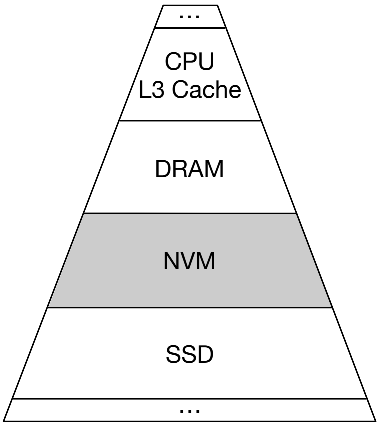
By storing data on fast DRAM instead of slow SSDs or HDDs, in-memory databases (IMDBs) outperform traditional disk-based databases. At the same time, DRAM suffers from one fundamental restriction: the future scalability of single-node in-memory databases is constricted by the stagnation of DRAM capacities [159]. Even though the physical boundaries have been repeatedly pushed, the underlying problems have not been solved [227]: physical limitations make it difficult to shrink cells (the one-bit building blocks of DRAM) in order to increase the DRAM density and thus its capacity. Shrinking the individual components of DRAM even further increases the error rates and has already led to security issues [123]. As such, researchers expect that the density, and with it, the capacity of DRAM will soon hit an upper limit [227]. Ultimately, this means that without new hardware innovations, it will become harder for single-node in-memory databases to keep up with continuously growing amounts of data.

An innovation that can help in moving past this limitation is non-volatile memory (NVM[1]). As of mid-2021, only one NVM product is publicly available, namely *Intel Optane Persistent Memory*. It is also marketed as DCPMM[2], which is the term that we use when specifically referring to this product. While DRAM uses a traditional transistor/capacitor architecture, DCPMM uses a new physical approach based on *phase-change memory*. This approach has several benefits over DRAM. First, because it does not rely on a capacitor to hold the state of a cell, it maintains its state even without power. This makes it *non-volatile*. Second, DCPMM can be produced with a higher density than DRAM, which allows the modules to achieve a higher capacity. Besides DCPMM, the advantage of higher capacities is projected for several upcoming types of NVM. It is the aspect of better scalability that is of interest in this thesis.

The advantages of NVM currently still come with a cost, namely higher access latencies and lower bandwidth compared to DRAM. While the read and write latencies of NVM come close to those of DRAM, they are not yet on par. This is visualized in the memory

[1] Also referred to as NVRAM, Persistent Memory (PMem), and Storage-Class Memory (SCM).

[2] DC stands for “data center”, PMM for “persistent memory module”.



Latency	Throughput	Capacity
15 ns	240 GB/s per CPU	< 50 MB per CPU
70 ns	r: 101 GB/s w: 80 GB/s	128 GB per DIMM
read (r): 350 ns write (w): 1050 ns	r: 37 GB/s w: 6.8 GB/s	512 GB per DIMM
20,000 ns	r: 3.5 GB/s w: 2.7 GB/s	2 TB per SSD

Figure 1.1: The memory and storage hierarchy, including key performance figures. Numbers are only given for reference and vary from product to product. Partially based on data taken from [104, 111, 242, 256].

pyramid shown in Figure 1.1. Programs that want to use NVM's higher memory capacity need to address this trade-off between capacity and performance. In the case of in-memory databases, a solution is to move pieces of data that are less-frequently accessed to NVM, thus keeping the frequently used pieces in DRAM. This process is known as *tiering* and the different devices that data can be stored on are referred to as *tiers*.

Tiering must not affect the correctness of query results. All pieces of the data that can contain values relevant to the result must be accessed regardless of their location. It is the goal of a tiering mechanism to minimize these accesses by making efficient tiering decisions in which only less-frequently accessed data is stored on (s)lower tiers.

Both prior work and our own research demonstrate that such less-frequently accessed data exists in the cosmos of enterprise IMDBs. Höppner and Rauhe show that in an analyzed ERP system, 65% of the data is used by less than 6% of the executed queries [106]. In one of our own case studies, we analyzed the meta data of a real-world ERP system. Here, we found an unusual amount of unused data stored in the audit log tables. This data was inserted by automatic processes that were inefficiently updating data in the customer address tables. Instead of changing a single field, e.g., *last contacted on*, the processes deleted and re-inserted the entire row. This led to an unnecessary amount of audit logs being generated. In total, these logs amounted to almost 250 GB of DRAM. While the DBAs were aware that the audit log tables were large, they were previously unaware of where this data was generated and whether it served a purpose. From this case study, we deduce (1) that less-frequently or even unused data exists in in-memory databases, and (2) that the responsibility of identifying rarely used data should not be imposed on the DBA.

Thus, we argue that future database systems need to make more decisions automatically. With this thesis and the presented implementation of automatic tiering, we work towards this goal.

So far, we have discussed two tiers, namely DRAM and NVM. However, in computer science, "*[t]he only reasonable numbers are zero, one and infinity*" [157]. It is thus unreasonable to settle for just DRAM and NVM. While SSDs are too slow to compete with the two memory types, they can still be used to further extend the capacity of in-memory databases [225]. This allows the system to store data that would otherwise have to be removed or *archived*, i.e., moved to storage outside of the database's control. By including SSDs in our tiering approach, we enable continued access to such data without the need of de-archiving the data. At the same time, designing an architecture that can handle an arbitrary number of tiers makes our approach more future-proof. This design decision has already proved fruitful as we established a new collaboration with a hardware vendor and will explore such an additional tier as part of future work.

While we are not the first to discuss tiering data in in-memory databases, existing solutions are lacking in one or multiple dimensions. Previous approaches either (1) require manual user intervention, (2) do not identify access skew in both the row and column dimensions as well as varying access patterns, or (3) only support two tiers, usually DRAM and either NVM or SSD. We discuss these existing solutions in Chapter 3.

1.2 Research Questions

With the goal of using NVM's capacity efficiently while keeping the performance close to that of a DRAM-only in-memory database, we pose the following two research questions:

1. *How can data be stored on different memory and storage tiers in a transparent manner that is consistent with the DRAM-first approach of in-memory databases and does not negatively affect the performance when accessing data stored on DRAM?*
   Storing data in main memory has proven beneficial from both a performance and an ease-of-development perspective. Our goal is to benefit from the existing work on IMDBs and to enrich them with additional capabilities to ensure their growth beyond DRAM scalability boundaries.
   From an engineering perspective, the question is how multiple tiers can be included in a manner that does not permeate the entire system's architecture but instead allows components below and above the tiering layer to remain unmodified. Neither should it be necessary to reimplement data structures to enable their storage on different tiers, nor should operators need to be adapted to access data on different tiers. From a user's perspective, tiering should neither complicate the administration of the system, nor deteriorate its performance.
   The performance aspect of this research question has previously been stated by Ma et al.:

   > *"The main challenge in supporting larger-than-memory databases on an in-memory DBMS lies in accessing the data stored on a secondary storage device without slowing down the regular in-memory operations."*

   While several approaches for this have been proposed in the past, none meet all of our criteria for flexibility, performance, and non-intrusiveness.

2. *How can a DBMS automatically (a) identify those parts of data that should remain on DRAM for performance reasons, and (b) migrate the remainder of the data without disrupting the continuous operation of the system?*
   To automatically migrate data, the system needs to gather information on the access characteristics of different parts of data and use this information to solve the underlying optimization problem. For part *a* of this question, we want to understand which information is valuable, how it can be gathered efficiently, and how it can be used in a decision making process.
   Second, when applying a new configuration, this needs to be done in a way that keeps the impact on concurrently executed queries to a minimum (part *b*). The migration needs to handle concurrent modifications of the migrated data in a way in which no updates are lost and the concurrent queries do not have to wait for the migration to finish.

To be able to answer these questions, several prerequisites have to be met. The most substantial of these is the availability of an open-source research platform that allows us to test and evaluate different approaches for automatic tiering. Existing systems are either closed-source or do not provide the flexibility needed to implement the proposed storage concepts. In the following section, we describe the contributions made both to meet the prerequisites as well as those made to answer the listed research questions.

1.3 Contributions

This thesis makes the following contributions: first, we present the in-memory DBMS *Hyrise* (Chapter 4). Hyrise is being developed by our group as an open-source research IMDB that facilitates our research on autonomous database management. Since starting in 2017, we have re-engineered the previous implementation of Hyrise and re-written the system from scratch. Additions include SQL support, a comprehensive query optimizer, and a flexible storage layer that provides a high degree of flexibility with regards to physical data management. This new version of Hyrise now also enables realistic end-to-end evaluations. In the TPC-H benchmark, the performance of Hyrise is competitive with that of comparable systems such as MonetDB, DuckDB, HyPer, and Umbra.

Hyrise is being worked on by a team of PhD and master's students without whom a project of this size would not be possible. Since the start of the Hyrise rewrite, I have significantly guided its architecture and the project's development process. As such, I consider Hyrise to be a significant contribution made as part of the work on this thesis. Still, I do not claim full credit for Hyrise and cannot overemphasize the value of my collaborators' input.

Second, we contribute a new abstraction layer for data stored on different memory and storage tiers (Section 5). This abstraction layer makes it possible to store arbitrary data structures (including, but not limited to, arrays, trees, and hashmaps) on DRAM, NVM, and file-based block devices. Our approach is less intrusive than previous memory management approaches. It can be applied to existing code by adding a single template argument to a variable's definition, which then enables them to be stored on different tiers. The implementation of the data structure itself does not need to be modified. This allows developers to write code that works regardless of where the data is stored. Despite this high degree of flexibility, the abstraction layer adds an overhead of only 0.6% compared to a DRAM-only system. With this, we solve the cited challenge of accessing data on secondary storage without slowing down the regular in-memory operations.

Third, we introduce a new type of multi-dimensional data access tracking method (Section 6). Unlike existing approaches, which mostly track accesses on either a row or a column level, we track accesses in both dimensions. With this, we can identify types of data skew that could not be identified if only one dimension was covered. Additionally, our tracking method identifies access patterns, such as sequential, monotonically increasing, or random accesses. This allows us to better take the performance characteristics of the underlying memory and storage tiers into account. We show how the access tracking can be implemented as an extension to the existing access abstraction layer [25] in a way that adds no measurable performance overhead.

Fourth and finally, we propose automatic decision making for horizontally partitioned, columnar in-memory databases (Section 7). This final contribution builds upon the previous three contributions: it uses data obtained from the access tracker to identify those parts of the data for which moving them to a lower tier comes with the lowest performance impact. The tiering decisions are then transparently applied via our multi-tier memory management framework. The decision making algorithm is fully implemented in Hyrise, which allows us to perform end-to-end benchmarks with industry datasets and standard research benchmarks. In the case of the public TPC-H benchmark, when 90% of the data is moved to NVM, the throughput is decreased by only 10.8% and the runtime is increased by 11.6%. Because the tiering decisions are periodically re-evaluated, they help the system to dynamically adapt to changing workloads without requiring any user intervention.

The first three contributions (Hyrise itself, the abstraction layer, and the access tracking) are self-contained in that they can also be used outside the scope of this thesis: (1) Besides being used as part of several other PhD theses, Hyrise is being used in Master's courses to introduce students to database and system development. Internally, we use Hyrise for collaborations with multiple hardware vendors as a means to evaluate the performance of in-memory databases on hardware prototypes. Furthermore, it is the basis for proof-of-concepts developed in our group as part of externally funded database research. Hyrise or parts of it have also been used by researchers outside of our group [50, 217]. (2) The memory abstraction layer, which allows data structures to be transparently stored on and migrated across different tiers, is not limited to Hyrise or even database systems in general. Our approach can be applied to various scenarios in which C++ data structures have to be allocated, accessed, and migrated transparently across different memory and storage tiers. (3) Our access tracking is internally used by other self-driving components that require fine-grained information about access frequencies and patterns. Its approach can be transferred to other database systems that use similar indirections and abstractions to access their table data.

1.4 Scope of this Thesis

To answer our research questions, it is necessary to make some assumptions and define the scope of this work. To the best of our understanding, none of these constitute a fundamental limitation to the approaches used in this thesis. Instead, we expect that future work will ease some of these constraints and thus extend the applicability of our work to additional scenarios.

Database Paradigms

We limit our discussions to the world of relational databases. While we have reason to believe that many results are transferable to, e.g., graph databases, these database paradigms are outside our area of expertise. Furthermore, we only consider column-oriented databases. In Section 3.1, we discuss how previous publications have addressed similar questions in the world of row-oriented databases. Finally, our starting point is the textbook in-memory database system [194], which stores all data on DRAM. We extend the idea of in-memory databases to cover additional memory and storage layers. Still, we orbit around the core paradigm, which assumes that all relevant data is stored on fast DRAM. This is in contrast to other tiering approaches that aim at building "*disk-based system[s] with in-memory performance*" [177].

Interaction with other Self-Driving Components

Automatic Tiering is just one part of our group's vision for an autonomous, or *self-driving*, database system. Other PhD candidates are working on topics like automatic index selection, automatic data compression, and automatic partitioning. In the future, multi-dimensional decisions will profit from the interactions between these optimization aspects. However, as these components are still being developed, we have to evaluate automatic tiering in isolation.

Our plan is for Hyrise to have a central *driver* that provides holistic tuning across different dimensions [127]. This driver will analyze and forecast workloads, take runtime information into account, and use an internal tuner to balance different optimization goals. This component, too, is still being researched. In its absence, we evaluate our automatic tiering implementation in a stand-alone fashion. This means that we exclude those features that will be part of the driver and instead focus on those that are unique to automatic tiering. At the same time, our architecture is designed in such a way that it is possible to adapt it into the planned comprehensive self-driving framework.

The automatic tiering approach further benefits from data being partitioned in a way that groups frequently used data together. A Hyrise plugin that performs this task has been developed as part of a Master's thesis [154] but has not yet reached the code stability needed to be fully included in Hyrise. As such, this thesis uses static partitioning. In case of the TPC-H benchmark, this means that the data is clustered by certain date columns, simulating the natural insert order of business data. This is done in accordance with the TPC-H specification [239].

Persistency

When data is moved to non-volatile tiers, including NVM or SSDs, these tiers can also be used as part of the DBMS's persistency concept. Data that is no longer stored on volatile DRAM does not need to be recovered in the case of a sudden power outage. In our previous work, we have shown how our proposed multi-tier memory management framework can be extended to persistently store data on NVM [62]. For this thesis, however, we exclude the persistency aspect. The reason for this is that the programming model for persistency on NVM will greatly change with Intel's *3rd Generation Intel Xeon Scalable Processors* and the next-generation of *Optane DC Persistent Memory*, codenamed *Barlow Pass*. Together, these introduce a new feature called *Extended Asynchronous DRAM Refresh* [92, 110]. Simply said, this solves an issue with the current generation of NVM in which data needs to be explicitly flushed from CPU caches in order to be persistently available after a power failure. We explain eADR in more depth later. Dropping the requirement for explicit flushes significantly changes the performance implication of using NVM for persistency. As such, a discussion of the persistency aspects would have to be revisited.

1.5 Outline

The remainder of this thesis is organized as follows: in Chapter 2, we give background information on the problem space in which we operate. This includes a high-level description of the state of the art in IMDBs and an introduction to non-volatile memory. We present a discussion of work related to tiering in in-memory databases in Chapter 3. Next, we present our in-memory database Hyrise in Chapter 4. This is followed by the three building blocks of our automatic tiering approach, namely the memory management (Chapter 5), the access tracking (Chapter 6), and the decision making (Chapter 7). Because these are self-contained contributions, they are individually evaluated in their corresponding chapters. We conclude the work with a summary of the presented results and a discussion of further opportunities in Chapter 8.

2 Background

In this chapter, we provide background information on (1) the type of database systems that we discuss in this thesis as well as on (2) non-volatile memory. Both topics are complex enough that entire books have been written about them [74, 194, 214]. We thus cannot discuss them exhaustively. Instead, the selection of which aspects to cover is based on the knowledge that later chapters build on.

2.1 DBMS Design Space

Our automatic tiering approach is designed for relational, column-oriented in-memory database systems. This means that the data stored by the user in our database is organized in two-dimensional tables with rows and columns. The term *relational database system* originally strictly referred to systems that used the relational model developed by Codd in 1970 [42]. These days, however, *relational* and *SQL-based* are often used synonymously [17]. Strictly speaking, there are several differences, such as duplicate or NULL values that are supported in SQL, but not in the relational model. Codd called these deviations from his original model "*serious flaws*" of SQL [44, p. 371]. Additionally, some terminologies differ between the two models: while Codd discusses relations, tuples and attributes, SQL uses tables, rows and columns. While we build and use a SQL-based DBMS, the finer disagreements between the two models are of no relevance to our tiering approach and we use the two terms interchangeably.

2.1.1 In-Memory Databases

In-memory databases can be best explained by contrasting them with traditional buffer-cache-driven disk-based systems. For decades, the limited amount of main memory forced database systems to store their data on hard drives. When accessed, data was moved into the main-memory *buffer cache* (aka. buffer pool) where it resided until it was evicted as other data was loaded into DRAM. This architecture was used to alleviate the orders of magnitude that lie between DRAM and HDD access latencies. With disk accesses being substantially more expensive than DRAM accesses, a fine-tuned buffer cache management was needed to benefit from DRAM's performance as often as possible. These caches were implemented in the DBMS, as the operating system caches were found to be insufficiently optimized for database workloads [231]. As such, the buffer caches complicated the DBMS architecture. At the same time, they added costs even for data that was already present in DRAM as the cache indirection had to be resolved before data could be accessed. Harizopoulos et al. quantified the overhead caused by the buffer cache for memory-resident data as 35% of the overall runtime [96].

With DRAM capacities increasing and prices decreasing, it became viable to keep *all* data in main memory. DeWitt et al. predicted this development as early as 1984 [55]. Still, it took another 25 years before general-purpose in-memory databases became commercially successful. From many perspectives, SAP HANA, introduced in 2010, can be considered the first widely adopted in-memory database [75]. Other large vendors followed with their own in-memory database systems two years later [213], including IBM solidDB [153], Oracle TimesTen [139], and Microsoft Hekaton [57]. A more comprehensive overview of In-memory databases is given by Zhang et al. [264].

At first, some "in-memory" systems took only a half-hearted step away from their disk-based legacy. Stonebraker et al. noted that "*some main memory database products on the market, such as TimesTen and SolidDB, [...] inherit the baggage of System R*" and continue to use components that impose "*substantial performance overheads*", including the buffer cache [232]. In contrast, most modern in-memory databases do not use a buffer cache anymore but perform direct loads and stores to DRAM. Leis et al. go as far as to call the lack of a buffer manager "*one of the defining characteristics of main-memory databases*" [148].

When the buffer cache and other "*baggage*" were removed, memory suddenly became "*the new bottleneck*" [29]. Previously, the cost of disk accesses dominated that of memory accesses. With disks gone, the details of memory accesses, including caches, prefetching, and NUMA effects became relevant contributors to a query's runtime. While main memory was previously considered to be homogenous, developers and researchers now consider it to be "*less and less appropriate to think of the main memory [...] as 'random access' memory.*" [29].

The fact that *Random Access Memory* can no longer be considered to have equal access times for random addresses shows in a number of places. We give four examples: first, data stored in one of the CPU caches is accessed significantly faster than data that has to be retrieved from actual DRAM. Depending on the system, this can make a difference of more than an order of magnitude. As a result, the designs of data structures [202] and algorithms [99] are designed to be more cache-sensitive. In Hyrise, this can be found, e.g., in the hash-join operator, which partitions the input data to better match the CPU's cache structure. Second, when data is read from consecutive addresses, CPUs can anticipate the next address that will be needed and *prefetch* the data before it is even requested by the application. This means that sequential accesses are faster than random accesses. In our work on automatic tiering, we see this pattern for both DRAM and NVM. In Section 7.1, we discuss how we include this knowledge in our decision making algorithm. Third, IMDBs usually use large servers with multiple NUMA nodes. Here, accesses to node-local memory are faster than accesses that have to traverse one or multiple NUMA hops. Psaroudakis et al. have shown how placing tasks and data in a NUMA-aware manner can improve the throughput by 5× [196]. In Hyrise, we use NUMA-aware task placement and implicit data placement. In Section 5.6, we outline how some of our contributions can be used to enable NUMA-aware data placement. Fourth, the physical organization of the data on DRAM introduces another source of non-randomness. It is described in the next subsection.

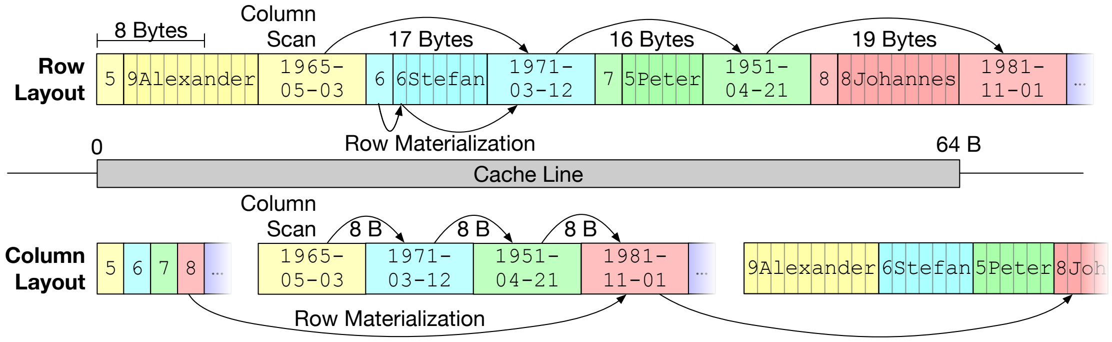


Figure 2.1: Visualization of row- and column-oriented storage in the case of a table with three columns: ID, Name, and Date of Birth (DOB).

2.1.2 Column-Oriented Storage

CPU caches are faster than DRAM by more than one order of magnitude [61]. At the same time, their size is limited. On a current Intel server processor, the Level 1 Data cache (L1D) has a size of 48 KB - around eight orders of magnitude less than the possible DRAM size [111]. This means that CPUs must have a sophisticated cache eviction strategy to manage the available cache slots efficiently. Part of this strategy is to use *tags* that establish a mapping between a cache slot and the physical address of the data that is stored in the slot. The use of these tags allows for a more flexible mapping between DRAM addresses and cache entries and thus increases the cache hit rate. At the same time, storing these tags requires additional space besides the 48 KB available for the cached data. Thus, the size of the tags needs to be limited [61]. To reduce the space consumption of the tags, the least significant five bits of the cached address are not stored in the tag. As a result, cache management happens on the granularity of $2^6 = 64$ Bytes[1]. This is known as one *cache line*. Even when accessing a single 8-Byte word, an entire cache line has to be transferred. This involves eight individual 64 Bit reads (aka. *bursts*) from DRAM [15]. Having just transferred 56 additional bytes, it is only efficient to use this data instead of treating it as cutting scrap.

Organizing data in a way in which all transferred data can be used is the core of the row versus column store debate. When storing a two-dimensional table on conceptually one-dimensional storage or memory, data can be stored in a row-oriented or a column-oriented format. An example of a table with three columns is shown in Figure 2.1. On the top, the table is organized in a row-oriented format. The three values of the first row are stored contiguously, followed by those of the second row. For each column, this means that the individual entries are stored one tuple size apart. Because of the variable length of the strings, the tuple size corresponds to 20 Bytes (2 Bytes ID, 1 Byte string length, 9 Bytes "Alexander", 8 Bytes Date of Birth, DOB), 17, 16, or 19 Bytes, respectively. When scanning the 8-Byte DOB column, a cache line of 64 Bytes[2] thus

[1] For the purpose of this discussion, we stick to current Intel Xeon architectures.

[2] The cache line is visualized as a grey bar in the middle of the figure

only holds 24 Bytes of used information, i.e., three dates of birth. In other words, 62.5% of the memory bandwidth is wasted. If the table contained additional columns, this number would increase further. For wide tables, it is not unrealistic to use only a single byte out of a 64 Byte cache line.

In the columnar layout on the bottom, all values of a column are stored contiguously. This means that a scan on the DOB column can utilize the entire cache line. Up to eight dates of birth fit into a single cache line. For scans on large tables, this means that no transfer bandwidth is wasted. While the columnar layout obviously excels for column scans, reconstructing a row becomes more expensive. Here, the *row*-oriented layout is at an advantage, as it stores the row in contiguous memory.

Hybrid approaches exist that try to find the optimal balance between row- and column-oriented storage [6, 12, 91]. Still, the two approaches remain conceptually irreconcilable. For a long time, relational databases have stored their data in the row-oriented layout, also known as the *n-ary storage model* (NSM) [46]. Column stores became popular in combination with in-memory databases, but the discussion of the columnar model's benefits predates in-memory databases by decades. A first comprehensive discussion of the two models was written by Copeland and Koshafian in 1985 [46]. In their work, the columnar layout is referred to as the *decomposition storage model* (DSM).

Traditionally, row-based layouts are associated with transactional workloads in which single rows are accessed via indexes and modifications across the entire width of the table are common. Column-based layouts, on the other hand, are considered beneficial for analytical applications, "*which typically examine a small number of columns from a large number of rows*" [138]. This distinction has been questioned, arguing that column stores can also be suitable for transactional applications [228]. Especially for enterprise (e.g., ERP) data, the columnar layout has a number of advantages:

- Tables in ERP systems are notoriously wide. In the case of an SAP ERP system, several tables have hundreds of attributes [26]. Virtually no transaction needs to access all columns. One of the reasons for these wide tables is that ERP systems are generalized for different use cases. In turn, this means that, for a given instance of an ERP system, several of these columns remain unused. In a columnar layout, the system can easily ignore these unused columns.
- With a growing number of columns, a *full* tuple reconstruction (i.e., the materialization of all values in a row) is rarely needed, even in transactional workloads. For fast data ingestion in these transactional workloads, writes can be buffered in a write-optimized, row-oriented area and periodically merged into the main part of the table [228].
- Having all values of a column stored in contiguous memory enables fast operations on all data in that column [67, 252, 253]. This in turns removes the need for pre-materialized aggregates and analytical indexes [138, 195].

By physically arranging data according to its predominant access patterns, column-oriented database systems thus improve the accesses to that data. Automatic tiering has similar goals: as we cannot make individual tiering decisions for every single value stored in the database, we need to group values with similar access characteristics. We discuss the two dimensions (row vs. column) of tiering in Section 6.1.1.

2.2 Non-Volatile Memory

In our automatic tiering concept, non-volatile memory (NVM) is the second-most important tier after DRAM. In fact, one can easily argue that automatic tiering for in-memory databases only became feasible with the introduction of NVM. This is because only NVM shows performance characteristics that are similar enough to DRAM to be treated as larger, but slightly slower DRAM. SSDs on the other hand, show substantially different characteristics and are too slow to be used in the place of DRAM. While we also support SSDs, we consider them more of a fallback tier and focus on DRAM and NVM.

Given that NVM is relatively new, having only been commercially available since the end of 2018, we give some background information on this new technology. We first define what separates NVM from other memory and storage types. Next, we explain the physical foundations of these hardware components. This helps to better understand what causes the volatility of DRAM and why the different technologies exhibit different performance characteristics. After this, we describe how NVM is integrated into the system and how it is made available to applications. Finally, we discuss the challenges associated with using NVM for persistent data and explain why we exclude persistency from the scope of this thesis.

2.2.1 Definition

The term *non-volatile memory* has been used as early as 1982 by Klein and Tchon [125]:

> *"The nonvolatile RAM most closely realizes an ideal solid state nonvolatile memory product. It combines fast read/write memory and nonvolatility. A near perfect memory."*

For this, they present a type of memory that is a combination of static RAM (SRAM) and an EEPROM cell. The former is used for reads and writes; the latter serves as a "*crash-protection device*". In case of a power failure, the energy remaining in the system, which is available for around 30 ms after the power failure, is used to write all SRAM contents into the corresponding EEPROM. A similar concept is called battery-backed DRAM, in which a traditional DRAM DIMM is combined with some type of non-volatile storage, such as NAND flash. Here, a battery or a supercapacitor is used to guarantee that the contents of the DRAM can be persisted to NAND flash. Narayanan and Hodson extend this concept to include the CPU caches [174]. These approaches guarantee the persistency of the data and are valuable when it comes to adding persistency to a system in a non-disruptive manner. Because they rely on SRAM and DRAM as memory components, they do, however, not provide any memory capacity beyond the existing limitations of SRAM and DRAM. As such, they are not relevant to our goal of expanding the capacity of in-memory databases.

A second type of non-volatile memory uses entirely new physical foundations. Instead of relying on existing memories such as SRAM and DRAM and combining them with durable storage, they combine both aspects in a single device. These new types of hardware are sometimes also referred to as *emerging memories* [163]. This group includes phase-change memory (PCM), which is the basis for Intel's DC Persistent Memory Modules (DCPMM) [40, 245] and which we use as the second memory tier. Other emerging memories are Ferroelectric RAM (FeRAM) and Magnetoresistive RAM (MRAM). The advantage of these emerging memories is that they are not bound by the limitations of SRAM and DRAM. They have the potential to be faster than DRAM (e.g., STT-RAM) or to allow for higher capacities (e.g., PCM) [32]. While some of these memories are already used, e.g., in chip cards, only DCPMM is advanced enough to challenge DRAM. The biggest purchasable DCPMM DIMM is four times larger than a traditional DRAM DIMM. It is this aspect of non-volatile memory that is most relevant for this thesis.

We use *NVM* synonymously for both non-volatile memory as a concept and Intel's NVM product. We do this because the architecture of our tiering component is largely agnostic of the actual NVM product. Whenever the difference between the two *does* make a difference, we use *NVM* for the concept and *DCPMM* for Intel's product.

2.2.2 Physical Foundations

We now describe the physical principles that underlie DRAM, SSDs, and different types of NVM. Discussing this helps to understand why these memory and storage devices differ in dimensions like their access latency, capacity, durability, and endurance. For the purpose of this thesis, we have to condense the explanation of these physical properties. Details such as multi-level cells, the layout beyond a single cell, and the implications on accessing multiple bytes have been omitted. For more in-depth information, including performance numbers of NVM technologies other than DCPMM as well as a literature review of their uses, we refer to the works of Boukhobza et al. [32] and Meena et al. [163].

All memory and storage devices exploit some physical property to store individual bits. For CDs, this is a gap in the data layer that allows light to reflect from the reflective layer below. Hard drives (HDDs) use platters coated with magnetic materials. The strength of the remanent magnetism is used to distinguish between ones and zeroes. Both CDs and HDDs are *non-volatile* storage mechanisms: once an information is written, it is held for a long period of time, which is measured in years. Their main limitation is their access performance. Because they require the storage location to be physically moved to the *reader*, i.e., the laser diode or the magnetic head, they are limited by how fast this movement can occur. This affects both their latency (how fast can the reader be positioned) and their throughput (how fast can the storage device spin). More modern devices, including DRAM, NVM, and SSD, do not rely on any physical movement and are based on the movement of electrons instead.

DRAM

DRAM does not require any physical movement. Instead, it uses semiconductors, more specifically, one capacitor and one transistor, to store one bit in a *cell.* To write a value, the transistor's *gate* is supplied with voltage, thus selecting the cell[3]. Depending on whether the *source* has a higher or lower voltage, the capacitor is then either charged

[3] We omit the source gate precharge and the details of true/anti cells for brevity.

or discharged. Similarly, data can be read by selecting the cell and measuring whether electricity flows from the capacitor back through the transistor into the source gate. This would represent a boolean one [19]. The process of writing to and reading from DRAM is also visualized in Figure 2.2.

Because the charge level of the capacitor is changed when it is accessed, the read operation is *destructive* [134]. As such, after a value has been read, the capacitor has to be recharged to its original value. The time needed for the capacitor to discharge together with the time to recharge it limits the read performance of DRAM [61]. As a second limitation, the capacitors leak charge even when they are not accessed. To prevent data from being lost, the charge has to be periodically *refreshed*. For a single cell, this refresh process happens every 64 milliseconds and consists of a read and a write of the stored value. While a cell is being recharged, it cannot be accessed [39]. Additionally, these recharges add up to significant power consumption [24].

SSDs

Unlike DRAM, the cells used in flash memory, e.g., in SSDs, use only a single transistor and do not require a capacitor[4]. This makes it possible to build flash memory with a higher density (and thus capacity) than DRAM. Instead of storing a charge in the capacitor, SSDs store their state within the transistor itself. Various methods for this exist. Modern 3D NAND flash memory *traps* electrons in a *charge trapping layer* in a process called *hot-carrier injection*. Because of this, the technology is called *charge-trap flash*, or short CTF. The trapped electrons cause a *field effect* that affects the conductivity of the transistor, making it possible to distinguish between zeroes and ones. Again, this is visualized in Figure 2.2.

Whereas in DRAM a read operation depletes the capacitor, the electrons in CTF remain trapped when the cell is read. Electrons can only move in and out of the electrically isolated charge trap if a sufficiently high voltage difference is applied to the control gate. For this reason, the state of a CTF cell remains stable over a long period of time and does not require periodic refreshes. One disadvantage, however, is that reliably trapping electrons takes longer than charging a DRAM's capacitor, resulting in higher write latencies for CTF cells compared to DRAM. A second disadvantage is that applying this high voltage deteriorates the isolation layer. Over time, this leads to a cell not being able to reliably hold its data anymore, causing potential data loss. This limits the lifetime (or *endurance*) of CTF. One vendor states that 40 nm CTF can endure only 100k write cycles [132]. To compensate for this, SSDs overprovision the number of cells and use wear-leveling to disable cells that are expected to fail soon. The deterioration of the isolation layer also limits the scaling of SSDs: when shrinking from 40 nm CTF to 28 nm, it sustains only a tenth of the write cycles [132].

[4] The description of flash memory is largely based on Micheloni's book *3D Flash Memories* [165].

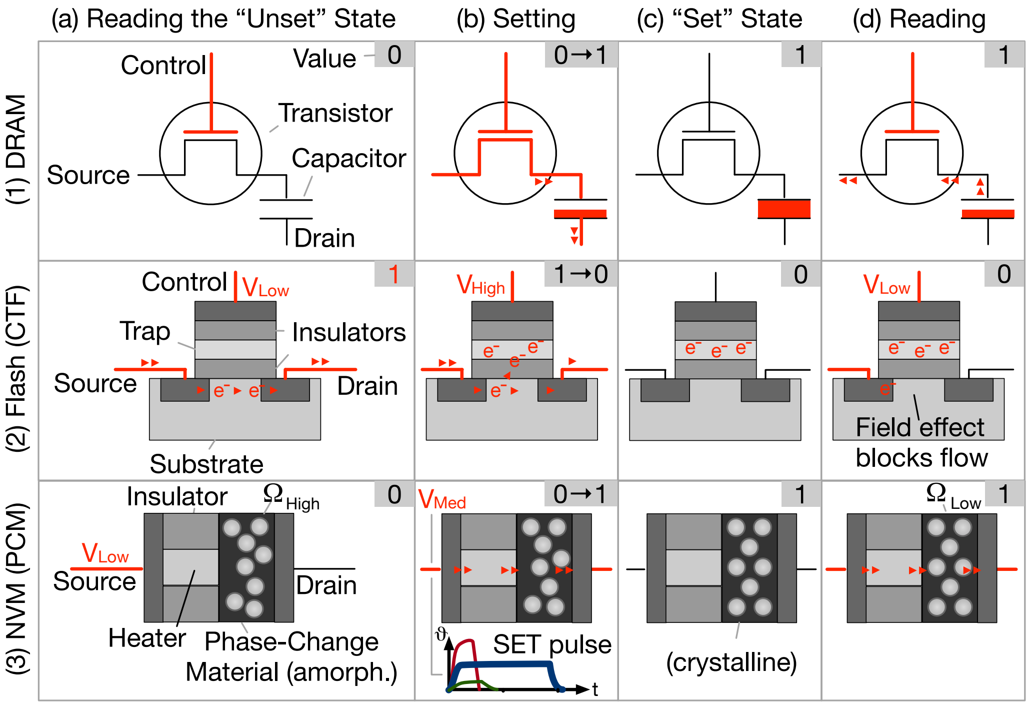


Figure 2.2: Visualization of (1) DRAM, (2) Charge-Trap Flash (CTF), and (3) Phase-Change Memory (PCM) in four situations: (a) reading in the unset state, (b) setting, (c) set, and (d) reading in the set state. The figure combines own work and visualizations from several sources [32, 68, 93, 212].
DRAM is read by applying voltage to the control gate. In the unset state, the capacitor is empty and no electrons flow between source and drain (1a). It is set by applying voltage to the source gate and the control gate (1b). This charges the capacitor (1c). If the transistor is selected and the capacitor holds charge (1d), electrons flow from the capacitor to the source, where a *one* can be sensed.
For CTF in the unset state (2a), the trap does not hold any electrons. This allows electrons to flow from the source to the drain if a *low* voltage is applied to the control gate, representing a *one*. To set the cell (2b), a *higher* voltage is applied to the control gate while also keeping the source gate powered. This causes some electrons to move into the trap where they remain when the cell is depowered (2c). When the cell is then read (again using a *low* control voltage), the field effect caused by the trapped electrons blocks the flow from the source to the drain, thus representing a *zero* (2d).
Finally, for PCM in the unset state (3a), the PC material is amorphous. This state has a high resistance (Ω_{High}) and does not allow electrons to flow from the source to the drain. This represents a *zero*. To set the cell, a higher voltage is applied for a longer period of time and slowly reduced (cf. Figure 2.3) (3b). This causes the PC material to change into the crystalline state (3c). This crystalline state has a lower resistance (Ω_{Low}) and allows electrons to flow, thus representing a *one* (3d).

NVM

DRAM and SSDs are the two rims forming the gap in the memory hierarchy that NVM is trying to fill. Having discussed the physical structure and the resulting performance implications of DRAM and SSDs, we can now talk about how NVM is supposed to combine the best of both worlds, namely

- a non-volatile data storage that does not rely on a continuous power supply and periodic refreshes;
- a higher capacity than DRAM, enabled by a lower number of elements (e.g., transistor or capacitors) and a lower size of these elements;
- faster accesses than SSDs by using a write mechanism that is faster than trapping electrons; and
- better write endurance than SSDs by limiting the physical degradation of the components.

Several technologies have been identified that could meet these goals. Together, they are known as *emerging memories* [163]. While many show promising in-vitro results, most of these still are lacking in at least one of the four named dimensions and only few are already commercially available. Three advanced technologies are FeRAM, MRAM, and PCM:

Ferroelectric RAM (FeRAM) has been used productively even before DCPMM was announced. It can be found in railway passes, automobile equipment, or domestic appliances [79]. Its design is similar to that of DRAM in that it uses one capacitor and one transistor. Unlike DRAM, however, the state is not stored as a charge but as a polarization state in the capacitor. While reads continue to be destructive, the polarization state is not lost over time [116]. FeRAM has a significantly higher endurance than flash memory and performance characteristics that come reasonably close to DRAM. However, its density and thus, its capacity, is still only in the range of megabytes and still insufficient to fill the gap between DRAM and SSDs [32, 51].

Magnetoresistive RAM (MRAM) uses magnetic effects similar to those used by modern HDDs. Instead of relying on a read/write head to move to the individual bits, MRAM includes the read/write component in the individual cells. The magnetic state of the cell influences its resistance, thus making it possible to sense its value [122]. Within the category of MRAM, several mechanisms are used, including Spin-Torque Transfer (STT-MRAM), which is the first type of MRAM to be commercially available [113]. Current commercial STT-MRAM chips are too small to solve the capacity issues of DRAM (1 Gb STT vs. 16 Gb DRAM), but forecasts by the vendor expect the gap to be closed soon [4].

Finally, *Phase-Change Memory* (PCM) is the only technology that, as of now, fulfills the four criteria listed above and is commercially available. It stores information in the *phase* of a special type of glass called *chalcogenide glass* that can be toggled between an amorphous ("glassy") and a crystalline state. Interestingly, this material is similar to that previously used for rewritable CDs (CD-RWs) [137]. Unlike CD-RWs, however, PCM does not use the optical but the electrical properties of chalcogenide glass. The amorphous and the crystalline state exhibit a different electrical resistance, which can be used to measure the state of the cell in a non-destructive way [32]. To write to a cell, the

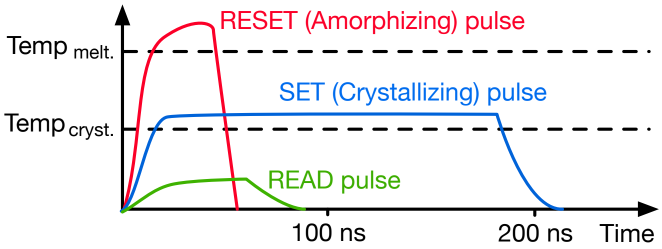


Figure 2.3: Phase change memory can be read and written by applying an electrical pulse. Depending on the voltage and length of the pulse, the material is either unchanged (read pulse) or changes its phase [93].

material is either quickly heated to a high temperature (placing it in an amorphous state when cooling) or slowly heated to a lower temperature (placing it in a crystalline state) - see Figure 2.3. Comparing the length of the SET pulse, to that of the READ pulse also explains why writes take significantly longer than reads, which is a main characteristic for main types of NVM.

While PCM can endure more writes than flash memory, it does not reach the endurance of DRAM. Also, the phase modifications currently take longer than charging a DRAM capacitor does. A big advantage of PCM is that its features are expected to scale better than those of DRAM and CTF. Together with the possibility of storing multiple values within a single cell [136], this could allow PCM to scale beyond the limitations seen for DRAM [143]. DCPMM is the first commercial product that uses PCM to fill the DRAM/SSD gap [245]. We discuss its performance characteristics in Section 5.5.

2.2.3 Hardware/Software Interaction

The difference between non-volatile *memory* and (non-volatile) *storage* is how it is accessed. While memory is directly attached to the CPUs' memory controllers, storage devices are attached via interfaces such as a PCIe/NVMe. According to Sun et al., these interfaces "*[burn] well in excess of 10000 CPU cycles for every I/O, adding several microseconds of latency and making small accesses costly in time and power*" [235].

A second performance factor of storage devices is the software layer. Unlike DRAM, which can be immediately accessed by the CPU, storage devices use abstraction layers including blocks, file systems and page caches. With modern SSDs, these software layers contribute as much to the read latency as the actual hardware access does [144, 209]. Attaching NVM directly to the memory controller addresses these issues. It means that, from a conceptual perspective, individual bytes on NVM can be accessed by the CPU in the same way that DRAM is used, namely by using load and store instructions. The actual size of these transfers is determined by the data transfer size (64 bits), the cache line size (64 byte), and, for DCPMM, the internal block size of 256 byte [242].

On a higher abstraction layer, however, there is an important difference between DRAM and NVM: in the case of DRAM, the lifetimes of the operating system (i.e., from startup to shutdown) and of the DRAM contents are identical. The OS kernel can assume that the DRAM address range is conceptually empty when the system starts. It then hands out DRAM pages to processes, which subdivide the pages to store individual data structures. When a process exits, these pages are returned to the kernel and can be reused. For *non-volatile* memory, contents from the previous run of the OS may still be present when the system boots. Not only must these not be overwritten, but they must be made accessible to the newly started processes. Because there is no continuity of individual processes across system restarts, the mapping of pages to processes is either lost or invalid. To re-establish this mapping, an abstraction layer that organizes pieces of data and makes it possible to access them across restarts is needed.

Conceptually, such an abstraction layer already exists in the form of filesystems. Processes can access the data stored in files across restarts. The filesystem does not only provide the functionality of a catalogue but also enforces access permissions. Additionally, filesystems make it easy for the user to manage stored data, which includes moving or deleting it as well as creating backups of the data. As such, filesystems are well suited for organizing data on NVM across restarts.

However, traditional filesystems come with two limitations that would significantly reduce the performance of NVM: first, because the filesystem and the device drivers are part of the kernel, reads and writes involve context switches. For traditional storage devices, for which the physical access time is in the range of tens of microseconds, these context switches are acceptable. For NVM with a sub-microsecond latency, they become "*killer microseconds*" [21]. Second, when data on filesystems is accessed, it is first copied into the DRAM page cache before it can be used by the CPU. This wastes both DRAM capacity and bandwidth. Dulloor et al. showed that avoiding these copies can improve the bandwidth of `cp` by 2.8x [201].

These limitations are addressed with a kernel feature called Direct Access (DAX), which is available for both Linux and Windows[5]. DAX allows applications to bypass the filesystem for individual reads and writes and to instead perform direct accesses to NVM. The corresponding NVM configuration is called *App Direct Mode*. The left side of Figure 2.4 visualizes this concept. While traditional file API accesses (e.g., `read` and `write`) are still possible, the preferred access mode is to use memory mappings. DAX allows applications to use mmap to create 1:1 mappings between virtual addresses and physical NVM addresses. Once such a mapping is established, the CPU can directly access data on NVM. These accesses bypass both the filesystem and the page cache. Besides reducing the latency, this also removes the need for a DRAM page cache.

Because the mapping is created by the filesystem, this method of interacting with NVM is called *fsdax*. A second way to use NVM's App Direct Mode is called *devdax* and uses NVM without a filesystem. It is not shown in the figure. Instead of creating a mapping to a file in the NVM filesystem, a character device file `/dev/daxX.Y` is mapped. It now becomes the responsibility of the application to manage the memory region. `devdax` is only viable in cases in which the entire NVM device is under control of the application. Because of this, `fsdax` is suggested as the preferable way of interacting with NVM [161].

[5] We exclude other approaches to these limitations, including SPDK, user-space NVMe, and MMIO for the sake of brevity.

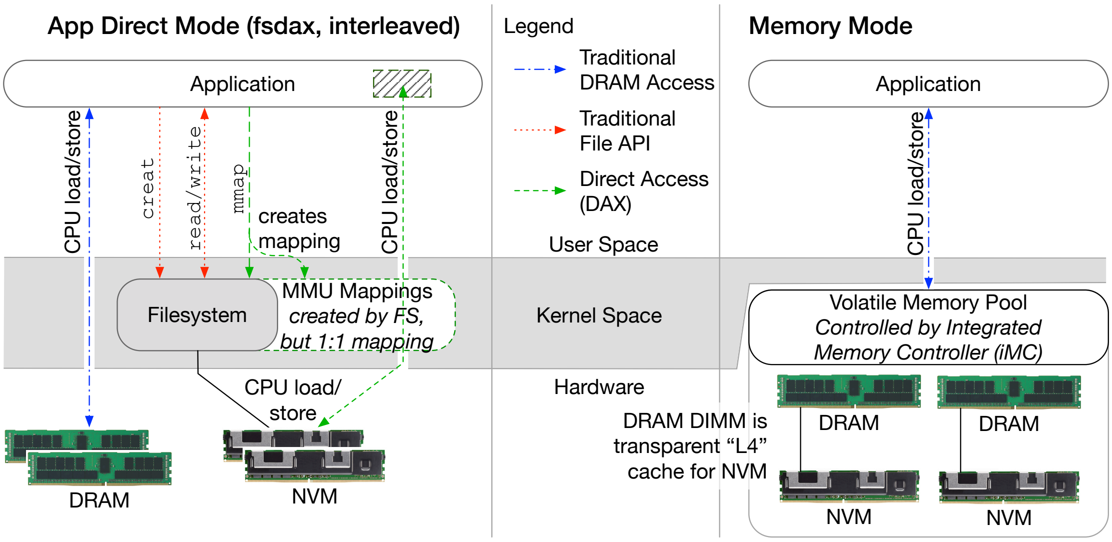


Figure 2.4: Two different methods to access NVM: App Direct Mode on the left, Memory Mode on the right, partially based on [59].

Recently, Daase et al. found that because devdax can forgo page fault handling, it exhibits a 5%-10% bandwidth improvement in certain benchmarks [50]. This finding is not yet reflected in our results. However, our benchmarks in Section 5.5 suggest that these 5%-10% would not have a significant impact on the bandwidth difference between the DRAM and NVM tiers.

Besides the App Direct Mode (fsdax and devdax), NVM supports a second configuration mode, called *Memory Mode*. In Figure 2.4, it is shown on the right. Instead of exposing NVM to the OS, DRAM and NVM DIMMs on the same channel are combined into a volatile memory pool. In this pool, DRAM serves as a transparent cache to NVM. The pool is presented to the OS as if it was DRAM. The OS (and thus the applications) benefits from a larger capacity and, if the cache hit ratio is sufficiently high, from DRAM-like performance. On first sight, this appears as if it might be a solution for our initial problem, namely the limited capacity of in-memory databases. However, there are two reasons why the Memory Mode cannot replace automatic tiering:

- Because the Memory Mode is entirely transparent to the OS, it does not provide any control over where data is placed and which criteria are used for eviction. This makes it hard to leverage the database system's knowledge about access patterns.
- The DRAM cache is inclusive [248], meaning that the DRAM cache duplicates data stored on NVM. While this makes the eviction of unmodified cache lines cheaper, it limits the capacity of the volatile memory pool to the size of the NVM DIMMs. In other words, the capacity of the DRAM DIMMs used for the pool is lost.

As a result, one design decision for our work was to use NVM in *App Direct Mode* and with *fsdax*. A second design decision was to exclude the persistency aspect for now. We explain this decision in the following subsection.

2.2.4 Persistency

In this subsection, we explain how data structures can be stored persistently on NVM, what the associated challenges are, and why we decided to exclude persistency from the scope of this thesis (cf. Section 1.4).

Having *non-volatile* memory with DRAM-like performance could allow us to resolve the "*dichotomy between memory and durable storage in database management systems*" [11]. Instead of storing data once in DRAM (for processing) and once on persistent media (for durability), a DBMS could store a single copy both for processing and durability on NVM. Doing so would simplify the architecture of database systems, as entire subsystems built around logging, checkpointing, and recovery could be removed. If data no longer needs to be recovered, but is inherently available for consumption by the CPU, this also reduces the restart times of large IMDBs from several hours to minutes [223]. To date, no such system has reached commercial relevance. While SAP HANA uses NVM for persistency, some parts of the data still require traditional persistency methods (cf. Section 3.1). In the following, we discuss the technical background that makes this vision hard to realize.

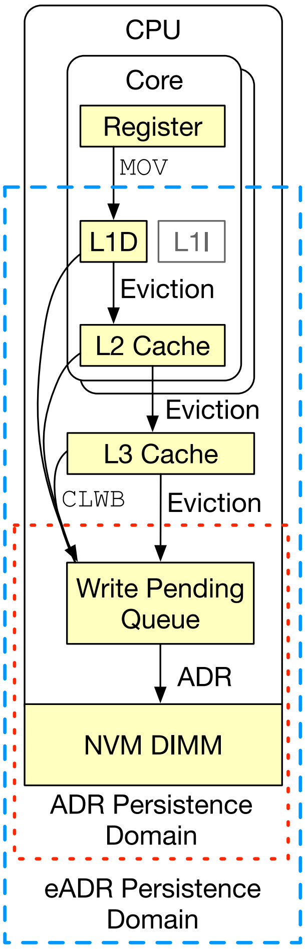


Figure 2.5: The DCPMM persistency domain, showing both ADR and eADR, based on [110].

The challenges when trying to store data persistently on NVM are not primarily caused by the current latency and throughput of NVM. Even if NVM was as fast as DRAM, we would see many of the same problems. This is because the challenges lie not on the NVM side, but on the CPU side: while NVM is non-volatile, the CPU caches continue to be volatile. As such, cache lines in the CPU caches are lost in case of a power failure. They become persistent only when the CPU evicts them from the cache or when the programmer explicitly requests that their contents be written back to NVM. For this, the `CLFLUSH[OPT]` (cache-line flush [optimized]) and `CLWB` (cache-line write back) instructions are used [11].

More accurately, `CLWB` does not write data from the CPU caches directly to NVM, but into the *Write Pending Queue* (WPQ). The WPQ is part of the CPU, more specifically it is part of its integrated memory controller (iMC). In the case of a power failure or a system crash, an interrupt is triggered, which causes the contents of the WPQ to be persisted on NVM. This feature is called *Asynchronous DRAM Refresh*, or short ADR. It uses residual energy that remains in the system even after the power supply unit signals an error state. As such, while both the caches and the WPQ are part of the CPU, only the latter holds persistently stored data. Together with the NVM DIMMs, the WPQ forms the *ADR Persistency Domain* as visualized in Figure 2.5.

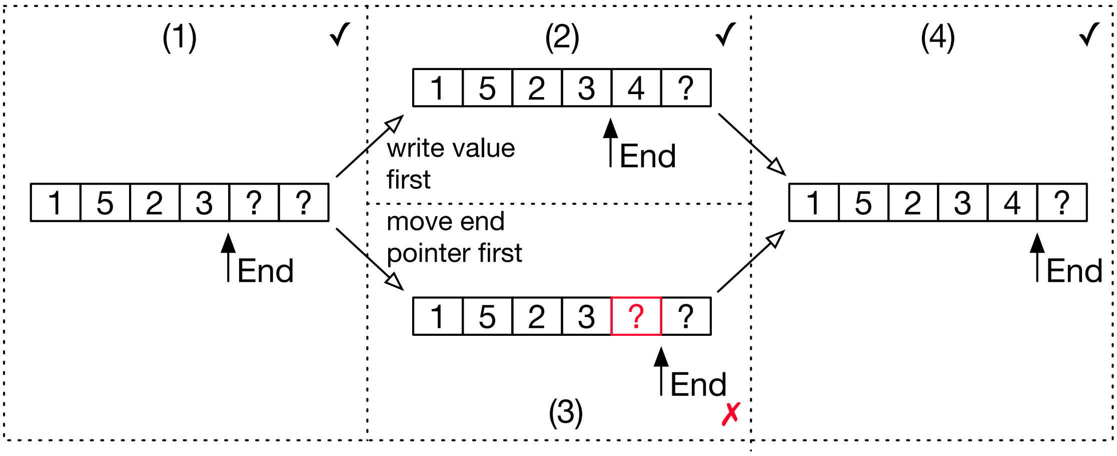


Figure 2.6: Modification of a persistent vector with different modification orders [219].

Calling CLWB to move data into the persistency domain guarantees that this data is persisted. However, the opposite is not true: even when CLWB is not called, data *might* reach the WPQ as part of the CPU's cache eviction mechanisms. If a modified (aka. dirty) cache line in the last-level cache (here: L3) is evicted, it is written back to its physical address, i.e., to NVM. As such, there is no guarantee of ordering between explicit (CLWB) and implicit (eviction) write backs.

Figure 2.6 shows an example in which this may corrupt the data. It shows a vector (a resizable array) with a capacity of 6. In State 1, $n = 4$ values have been inserted and the *end* pointer points to the element at position $n + 1$. The remaining two elements have an undefined value. We want to insert a fifth value into this vector. The result is shown in State 4. Between the initial and the target state, two modifications have to be made: writing the new value and advancing the end pointer. Usually, the order in which these two steps happen does not make a difference as long as the vector is not read before the modifications have completed.

With NVM, only one transition is safe, namely writing the value first, calling CLWB on the written value, then advancing the end pointer and calling CLWB again, this time on the location of the end pointer itself. If the system crashes, it is either in State 1, 2, or 4. While the in-flight modification might have been lost, the data structure continues to be in a consistent state. Thus, the modifications are atomic. On the other hand, if the end pointer was incremented first and the value written afterwards, the system would pass through State 3 in which a crash would leave it with a corrupted data structure: because the end pointer has already been incremented but no valid value has been written, the vector would be inconsistent after recovery.

Given the complexity of adding a value to a simple vector, defining an order in which modifications need to occur becomes more difficult for bigger data structures. Unfortunately, it does not end here. Besides implicit and explicit cache-line write backs, the order in which data appears in the persistency domain is also influenced by out-of-order execution (OOO). Two types of OOO are important here. First, modern CPUs do not execute instructions in the order defined by the program, but reorder them in a way that makes better use of the available CPU resources [45]. The CPU architecture guarantees that despite instructions being executed out-of-order, the externally visible state is consistent with the original order. This guarantee, however, is only valid for single-threaded execution on a continuously running system.

A system may lose power at any moment. As such, a different instruction order in which not all writes are persisted on NVM may result in inconsistent data structures. This means that even if the correct order of modifications and `CLWB`s is used by the programmer, the CPU may reorder the commands and break the consistency guarantees. Furthermore, Intel CPUs, which are the only ones that support DCPMM, only guarantee that eight bytes are written atomically [150]. This means that when writing a bigger data structure (for example a longer string), cache eviction may lead to incomplete writes even within the same cache line.

The second type of OOO is introduced by the compiler of the programming language. For example, the C++ as-if-rule "*[a]llows any and all code transformations that do not change the observable behavior of the program*" [47]. The definition of "*observable behavior*" does not cover what is visible in the persistency domain. As long as the continously running program is deterministic, the order in which its memory contents are written can vary. To address the two types of out-of-order execution, programmers thus need to explicitly state that certain instructions (most importantly, the `CLWB`s) may not be reordered. This is usually done by the use of memory fences (e.g., `SFENCE`).

Requiring `CLWB` and `SFENCE` has two drawbacks: first, it increases the burden on the developer. Having to always maintain the correct order of writes to the persistency domain negatively impacts their productivity and is likely to introduce additional bugs. Libraries, such as PMDK [214], are designed to relieve the developer from some of this burden. Still, from a programmer's perspective, it appears as if the "*dichotomy between memory and durable storage*" [11] is still easier to maintain than to remove.

The second drawback is that with explicit write-backs, we are now introducing additional writes from the CPU caches into the memory subsystem. Taking the vector as an example, the end pointer would likely be kept in the CPU cache. To make the vector persistent, we now need to write it to NVM every time we change the vector's size. As such, we not only deal with NVM's higher cost of individual writes, but also with an increased number of such writes. This *write amplification* means that even if NVM was as fast as DRAM, we would still suffer from increased write costs.

Fortunately, these additional costs will be addressed with the next generation of DCPMM (codenamed Barlow Pass) and the 3rd generation of their Xeon SP processors (codenamed Ice Lake-SP). While the former is already available, the latter is on back order and only expected for late 2021 [207, 208]. Together, these introduce a feature called *Extended Asynchronous DRAM Refresh*, or short eADR [110]. With eADR, the persistency domain is extended to include the CPU caches. This is also visualized in Figure 2.5. In case of a power failure, a backup battery on the mainboard is used to provide sufficient energy to perform the writebacks of modified cache lines [169], not only from the WPQ, but also from the CPU caches. As such, `CLWB` and fences are no longer needed.

It is easy to see that this will cause changes to the previously discussed programming model. For some data structures, eADR improves the bandwidth by more than 7× [207]. With these upcoming changes, many existing pieces of code and published performance evaluations will have to be revisited. It is for this reason that we decided to exclude the aspect of persistency from the scope of this thesis.

In our previous work, we have shown how our memory management framework (most notably, PMR-based memory management as described in Chapter 5) can be modified to include the steps required for persisting data on current NVM [62]. We have decided to exclude this discussion from this thesis in the light of the upcoming changes and because we consider them to be only a possible, but not required, addition to our self-contained automatic tiering approach.

3 Related Work

In this chapter, we discuss existing approaches to modern, NVM-aware multi-tier memory management for in-memory databases (IMDBs) as well as the surrounding research areas of (1) library support for NVM-aware tiering and (2) NVM-optimized data structures. We discuss how existing systems relate to our research questions and contextualize our contributions.

Two aspects are excluded from this chapter: we have integrated the discussion of related work in the area of access tracking (Research Question 2a) into Section 6.1.1. This was done because it is easier to follow the decision for a certain access tracking granularity with all alternatives being presented side-by-side. Secondly, concerning the work that is related to Hyrise as a DBMS but not to tiering in particular, we refer to the respective sections in previous publications [30, 66].

3.1 Modern Multi-Tier DBMS Architectures

All traditional database systems could be considered to be multi-tier systems. Because the capacity of DRAM was insufficient to store all data, data was stored on secondary storage (HDDs and later SSDs) and loaded into DRAM by the means of a buffer cache. As DRAM grew bigger and IMDBs became viable, one of the first optimizations was to reconsider the need for a buffer cache. Because existing buffer cache implementations were found to cause a 35% overhead even for data stored on DRAM [96], most IMDBs removed this abstraction layer. Together with the buffer cache, however, they lost their ability to store data on tiers other than DRAM. A growing interest in benefiting from the speed of IMDBs without sacrificing database capacity as well as the emergence of NVM prompted researchers and developers to reconsider multi-tier database systems.

3.1.1 Anti-Caching as a Concept

A seminal paper on how to handle larger-than-memory data sets in in-memory databases was published in 2013 by DeBrabant et al. [54]. Their goal was to enable in-memory databases to store data beyond the capacity limits of DRAM without incurring the overhead imposed by the traditional buffer cache approach. For their research DBMS *H-Store* [117], they developed the concept of *Anti-Caching* and described it as follows:

> *"In this new architecture, main memory, rather than disk, becomes the primary storage location. Rather than starting with data on disk and reading hot data into the cache, data starts in memory and cold data is evicted to the anti-cache on disk."*

Besides the implementation in H-Store by DeBrabant et al., this concept can also be found in other multi-tier architectures, including Siberia [71], which is discussed in Section 3.1.3. It is also the concept that we use for our work on automatic tiering. While these systems share the *idea* of anti-caching, their *implementations* differ significantly. In their survey on anti-caching, Zhang et al. [264] decoupled anti-caching as a concept from its implementation in H-Store. We follow this distinction and describe the two aspects in separate subsections.

Anti-caching competes against caching and paging. They share the same goal, namely, to balance the latency and bandwidth advantages of DRAM with the increased capacity of lower tiers (usually SSD or HDD). In the case of caching, the key difference lies in what DeBrabant calls the "*primary storage location*". Traditional databases store their data on slow SSDs and HDDs and use the DRAM buffer *cache* to profit from faster accesses to cached data. Data read from lower tiers into the cache may occupy any cache slot. As such, even when accessing data that is already present on DRAM, operators need to obtain the virtual address of the cache slot and calculate the offset of the requested data. Additionally, the cache slot needs to be latched in order to make sure that it is not evicted while it is being read [87]. This indirection adds complexity and introduces the buffer manager as a contention point. In contrast to caching, anti-caching stores data natively on DRAM and uses lower tiers to profit from their higher capacity. This allows for memory-resident data to be accessed without indirections.

Drawing the line between anti-caching and paging is more difficult. The goal of paging is to allow processes to allocate memory beyond what is available in the form of DRAM. If all DRAM is used, data is paged out to storage. As such, one could consider paging to fulfil the definition of anti-caching cited above. Indeed, the survey by Zhang et al. [264] lists OS paging as one possible implementation of anti-caching. We, however, see three important differences: first, OS paging is driven by the kernel and the DBMS process does not influence which pages get evicted. Depending on the page replacement algorithm, pages that have not been recently accessed are chosen for eviction. Anti-Caching, on the other hand, uses domain knowledge to exert control over which data is moved to a lower tier. We discuss the advantages of the different tiering granularities in Section 6.1.1. Second, the lack of fine-grained control over the OS paging mechanisms means that multiple database instances (e.g., in a multi-tenant scenario) compete over the available DRAM. Regardless of whether these instances run in individual processes or in a single DBMS process, the tenant that produces the highest load will cause other tenants to be paged out. With anti-caching, in which the DBMS actively limits the amount of used DRAM, multiple instances can co-exist and better stick to defined resource limits. Third, OS paging is limited to two tiers (DRAM and the device of the page file), whereas anti-caching can be programmatically extended to three or more tiers as proposed in this thesis.

3.1.2 Anti-Caching as Implemented in H-Store

DeBrabant et al.'s original implementation of anti-caching [54] is tailored to the architecture of H-Store and its unique transaction and partitioning model [117]. H-Store is a distributed, row-oriented research DBMS that is optimized for transactional (OLTP) workloads. Its commercial spin-off is VoltDB [233]. In H-Store, data is partitioned into *sites*. Each site has exactly one worker thread that operates on the site's data. This removes the need for any type of concurrency management within that site and greatly increases the throughput of transactional workloads. Across sites, a distributed transaction framework is used to guarantee the serializability of multi-site queries. Because of the transaction-focused nature of H-Store, most read accesses are expected to utilize an index. Originally, H-Store was designed under the assumption that all data fits into DRAM. The work of DeBrabant et al. relaxes this assumption.

To identify data that is less frequently accessed, H-Store maintains an LRU (least recently used) chain[1] across the rows of a table. Each tuple holds a pointer to the previous and the next entry. If a row is accessed, it is removed from its current position and placed at the end of the list. The overhead of maintaining this list is measured to be within 5% of the baseline. An independent publication by Levandoski et al., however, measured an increased overhead of up to 25% for this approach [151].

When the memory consumption of the database exceeds a user-defined limit, a synchronous eviction process takes the least recently used rows from the front of the list. These rows are copied into a 1 MB *block* that is then written to disk after which the DRAM space occupied by those rows is released. To be able to retrieve the row without scanning the entire disk storage, a *block table* keeps a mapping from tuple ID to block ID. The migration process is done by the site's worker thread. While this avoids concurrency issues, it also blocks other queries from executing while a migration is ongoing.

As part of the query execution, a *pre-pass phase* identifies all rows that will be accessed as part of the query. This is possible because in H-Store, most lookups utilize an existing index, which returns the tuple IDs matched by the query's predicates [117]. If all of these rows are memory-resident, the query continues regularly. If any rows have been evicted earlier, they are now loaded back into memory. However, the pre-pass phase cannot always identify all accessed rows. When multiple tables are joined, the rows accessed by a later join are only known after earlier joins complete. In this case, the pre-pass phase is repeated. The worst case is when data is accessed by non-indexed attributes. In our understanding, this could require entire tables to be unevicted.

Because each site in H-Store is handled by exactly one thread, waiting for data to be loaded from disk would prevent other queries on the same site from being executed. Even data that is fully memory-resident would not be accessible anymore until the earlier query finishes. To fix this blocking behavior, a query that causes a load from disk is aborted together with its corresponding transaction. This allows following queries to be executed. The rows from the earlier query are then asynchronously loaded back into memory by a different thread. After this, the transaction is restarted. Restarting the transaction is only possible because H-Store transactions are self-contained stored procedures and do not allow for any user interaction. While this approach works well for H-Store, it is hard to generalize. In regular SQL, transactions cannot be restarted by the DBMS alone. Instead, an aborted transaction has to be restarted by the user.

[1] The paper uses the term "(linked) chain" instead of the more common "linked list".

This increases the cost of restarts and thus, the cost of accessing disk-resident data. It also makes it difficult to use H-Store's anti-caching implementation in scenarios in which queries involve some type of user interaction, such as order processes that include Available-to-Promise checks [167] and reserve order items for the duration of the process. Here, an aborted transaction would significantly impact the user experience.

A second drawback of the implementation is that its pre-pass phase depends on in-memory indexes to identify the rows that need to be loaded from disk. This inhibits its applicability to unindexed ad-hoc queries, for which the entire table needs to be loaded from disk, as well as to queries with complex joins for which it is not possible to identify all accessed rows in a single pre-pass phase. Relying on an index-based pre-pass phase also means that those indexes must remain in DRAM, limiting the space savings of the eviction method [71].

A later paper by DeBrabant et al. describes how anti-caching could be used to utilize the additional capacity of NVM [53]. Here, evicted data is moved to NVM instead of disk. Even though NVM is byte-addressable, the data is still un-evicted into DRAM before it is processed. As such, NVM is only used as a faster disk and most of its capabilities remain untapped.

H-Store's anti-caching implementation and our tiering implementation share the same goals, but the two approaches are different in many regards:

- Hyrise can directly access data on both DRAM and NVM. It is not necessary to copy data from NVM to DRAM to access it (cf. Section 5.1).
- Data migrations in Hyrise are non-blocking. We exploit the existing concurrency scheme to ensure that transactions can continue while data is being migrated (cf. Section 5.4).
- Because of its column-oriented nature, Hyrise cannot migrate data on the row level. Instead, we track and migrate data on the segment granularity (cf. Section 6.1.1).
- H-Store supports only two tiers, either disk [54] or NVM [53]. While multi-tier support was discussed, it was never implemented [189]. Hyrise supports multiple tiers (cf. Chapter 7).

3.1.3 Microsoft SQL Server, Hekaton, and Siberia

Eldawy et al. describe another implementation of the anti-caching concept, which they call *Siberia* [71]. Their work adds secondary storage to *Hekaton* [57]. Hekaton is an in-memory, row-based OLTP engine within Microsoft's SQL Server. DBAs can move tables between the disk-based SQL server and the memory-optimized Hekaton engine. Queries can access tables in both locations and can even combine disk-based (also called "regular") and Hekaton tables. However, certain optimizations, such as just-in-time compilation, are only available if all accessed tables are stored in Hekaton. As such, research publications usually discuss Hekaton as an isolated engine [7, 71, 151]. Similar to anti-caching in H-Store, the goal of Siberia is to move less frequently accessed rows from DRAM to secondary storage. However, Siberia addresses several shortcomings of the H-Store approach, specifically the need for in-memory indexes and the need for a pre-pass phase. The authors divide the challenge of anti-caching into four subproblems:

1. Cold data classification: To identify less frequently accessed rows, the authors evaluated H-Store's LRU chain approach but found that maintaining the LRU chain adds an overhead of 25% to the cost of lookups and 16 bytes to every record[2]. Instead, they decided for an approach in which accesses are logged and analyzed offline [151]. Not only does this reduce the runtime and space overhead, but it also allows for the log to be processed externally.
2. Cold data storage: Evicted data needs to be stored in some disk-based data structure. The paper does not go into detail here but states that "*records in the cold store contain the same key and payload fields as their hot counterparts plus a field TxnId that stores the Id of the (migration) transaction that inserted the record into the cold store*". We thus assume that no additional transformation occurs besides the addition of the transaction id.
3. Cold storage access reduction: Siberia explicitly does not store information about individual cold records in memory. However, they can use compact Bloom and range filters to reduce the number of accesses to cold records [7]. Within the cold store, indexes may be used for faster accesses to the individual record. This is unlike H-Store, which requires in-memory indexes and an in-memory block table for efficient accesses to cold data.
4. Cold data access and migration mechanisms: Siberia migrates data between the two tiers by deleting it in one tier and inserting it in the other. For this, it uses Hekaton's Multi-Version Concurrency Control scheme [142]. This allows them to ensure that exactly one version of the record is visible at a given moment. Executing the migration within a transaction also ensures that concurrent modifications can be identified. To read data, Siberia uses transparent cursors that can access hot and cold data alike. If cold data is accessed, the row is copied into an in-memory cache that is private to the current transaction. While this copy blocks the current transaction, it does not block other transactions from making progress. Siberia also differs from H-Store in that it is not limited to only one transaction on a given site and allows for accesses to cold records without causing the transaction to be aborted. Furthermore, Siberia does not automatically un-evict rows that are accessed once but waits for their access frequency to meet the overall threshold for being considered frequently accessed. Finally, Siberia's use of its MVCC protocol solves H-Store's issue of migrations blocking the execution of regular queries.

One goal of Siberia is to reduce the impact of their implementation on the overall architecture and on the execution performance. This is achieved by (1) tracking the accesses outside of the individual rows, (2) hiding the details of accesses to hot and cold stores behind a common cursor interface, and (3) using the existing MVCC mechanisms to guarantee the transactional safety. When devising the design of our tiering approach, the abstraction introduced by Siberia served as an inspiration. Similar to Siberia, Hyrise uses a cursor (in Hyrise: *iterator*) abstraction that hides the access method from the accessing operators (cf. Section 4.3.2) and that can read data from lower tiers without

[2] DeBrabant et al. state that they use two 4 byte offsets instead of 8 byte pointers [54] for their LRU chain. This is confirmed by Leis et al. [148].

the need to un-evict it (cf. Section 5.1). We also employ filters to reduce the number of accesses to evicted segments (cf. Section 4.3.1). Finally, we exploit the immutability guaranteed by the combination of our chunk concept and our MVCC implementation to migrate data between tiers without jeopardizing the transactional safety (cf. Section 5.4).

A main difference between Siberia and Hyrise is the granularity of our tracking and eviction methods. Whereas Siberia is row-oriented, Hyrise works on the level of segments. A second difference is that Hyrise uses the iterator abstraction not only for accessing data, but also for tracking the access frequencies. In Chapter 6 we describe how this is implemented with virtually no runtime or space overhead. Third, Hyrise supports multiple tiers and allows for immediate accesses to DRAM and NVM. While Microsoft discussed experiments that used NVM for persistency [56], we are not aware of Siberia being extended to support multiple tiers.

3.1.4 SAP HANA

SAP HANA [75, 146, 162, 195, 228] was one of the first commercially successful in-memory databases. To enable high throughput rates for both analytical and transactional queries, it organizes its data using a main/delta architecture in which tables consist of two horizontally divided fragments [228]: most of each table is stored in the *main* fragment of the table, a heavily read-optimized column store. This fragment employs compression techniques that are efficient in terms of their memory savings and throughput, but do not allow for modifications, such as ordered dictionaries and bit-compressed vectors [253]. To amortize the cost of updating the main fragment and to enable fast inserts into the table, a write-optimized *delta* fragment is used for new rows. Because of its insert-only approach, these new rows include updated rows for which the original version was invalidated and a new version was inserted. Periodically, the delta fragment is merged into the main fragments [133].

In 2018, HANA was the first commercial DBMS to support non-volatile memory [8]. For users, NVM brings two advantages: first, because it is non-volatile, data stored on NVM does not have to be restored after a planned or unplanned restart. In HANA without NVM, the recovery time grows by one hour per loaded terabyte [211]. For data stored on NVM, restoring access is orders of magnitude faster and can be done in almost constant time. However, no end-to-end recovery durations are given in the paper.

Much of the engineering effort went towards enabling this type of persistency and recovery in an ACID-compliant manner. As discussed in Section 1.4, this thesis excludes the aspect of persistency. The second advantage of NVM support in HANA is that it increases the amount of available memory. This aspect is of greater interest in the context of this thesis.

The NVM support of HANA is unlike that of H-Store, Hekaton, and Hyrise in that it uses NVM as the primary storage location. By default, all main fragments of the table are stored on NVM. Even when they are heavily accessed, HANA does not migrate them to DRAM in the way that the other three systems do. At the same time, the delta fragments, as well as all temporary data, always remain on DRAM. This is done for two reasons: first, persistency can be implemented more efficiently for data that is not undergoing constant modification. Second, accesses to the main fragment are dominated by sequential reads, which are less affected by NVM's increased latency than the random reads and writes seen for the delta fragment and temporary data [8]. The decision to statically assign the fragments to different tiers means that accesses to the main fragment cannot profit from the lower latency and higher throughput of DRAM. While the use of NVM can be disabled for individual tables, partitions, or columns within partitions, this decision has to be manually made by the DBA [211].

By default, all data stored in HANA is memory-resident, i.e., stored on either DRAM or NVM. To prepare for situations with high memory pressure, columns are kept in a least recently used list [225]. When necessary, the least recently used columns are evicted (or "*unloaded*" in HANA terminology) from DRAM (or NVM). This mechanism is called *column-loadable data*. Because this setting applies to entire columns, it does not allow for fine-grained tiering decisions. With the *Native Storage Extension*, or short *NSE*, HANA adds a second tiering mode, called *page-loadable data* [225]. DBAs can mark tables or parts thereof as page-loadable, which means that, instead of always being present in memory, they are primarily stored on a lower tier and are brought back into memory when accessed. For this, the existing data structures have been modified in a way that makes them partially accessible on a per-page basis [226]. These pages can then be individually loaded into a DRAM buffer cache. This is very similar to the buffer caches used in traditional disk-based databases. The main difference to traditional systems is that, in NSE, the buffer cache is only used selectively: data that is *not* marked as page-loadable (i.e., column-loadable data) remains in main memory and is accessed without any additional abstraction layer.

The DBA is aided in the decision of which data to mark as page-loadable by the *NSE Load Unit Advisor* [225]. The advisor bases its recommendations on access statistics, which are gathered for each column fragment. The number of accesses to a fragment is put into relation to the fragment's size. This ratio of accesses is called the *scan density*. Depending on system-wide thresholds, the objects with the lowest scan density are suggested to be marked as page-loadable. The DBA then has to decide whether the proposed layout is applied.

For both HANA with NVM and NSE, the tiering decisions have to be made manually. For HANA with NVM, the default is for all main fragments to be stored on NVM. If one of these fragments is heavily accessed and would profit from the performance of DRAM, a DBA has to identify this pattern and act accordingly. For NSE, the DBA is supported by the Load Unit Advisor but still has to manually verify and apply the suggested configuration. In our approach presented, these tiering decisions are made automatically. From our understanding, there is no conceptual reason why HANA could not also apply the advisor's suggestions automatically.

Our work shares several design decisions with NSE: both implementations keep track of the column access frequency. NSE uses column fragments, Hyrise uses segments. In addition to the number of accesses, Hyrise also identifies access patterns (cf. Chapter 6). As a second similarity, the Load Unit Advisor and Hyrise share a comparable concept for deciding which data should be placed on which tier. NSE orders the fragments by their scan density and Hyrise uses a branch-and-bound knapsack implementation.

The scan density used by NSE is equal to the profit density used in a greedy knapsack algorithm [60]. In addition to the number of accesses and the size of the segments, Hyrise uses previously tracked access patterns (cf. Section 7.1) to identify which segments are read mostly sequentially and which are accessed in a random fashion. This is because sequentially accessed segments are less likely to be impacted by NVM's higher latency. The same reason was given by Andrei et al. for placing the main fragment on NVM [8].

A main difference between NSE and Hyrise is the primary location of the data. Hyrise follows the anti-caching approach in which DRAM is the primary location and less-frequently accessed data is moved to lower tiers. NSE, on the other hand, uses multiple primary storage locations. In a system that uses both NVM and NSE, frequently accessed data is either stored on NVM (main fragments) or DRAM (delta fragments) while less frequently data is stored on disk and accessed via the buffer cache. Hyrise, on the other hand, primarily stores data in DRAM and fluently moves it to lower tiers as appropriate.

3.1.5 Leanstore, Umbra, and Mosaic

A different approach to achieving the performance of in-memory databases while benefiting from the capacity of disk-based databases has been chosen by researchers at TU Munich. *Leanstore* [148] is a new storage engine that implements a buffer cache in a way that avoids the bottlenecks of traditional implementations. *Umbra* [177] is a DBMS that combines the Leanstore storage layer with an execution engine that is the "*spiritual successor of HyPer*" [120]. Like HyPer before, it is used as a vehicle for several research projects [31, 118, 121, 254]. Finally, *Mosaic* [243] is an alternative buffer manager for Umbra that focuses on optimizing scan-heavy workloads on SSDs and HDDs. We group the three research projects both because of the overlap in the development team and because of their shared technical ancestry.

Leanstore

Instead of removing the buffer cache, Leis et al. [148] aim at replacing those components of traditional buffer caches that cause them to be inefficient for DRAM-resident data. Traditional buffer caches use a single hash table to store the state of the buffer cache. If a page is loaded into DRAM, an entry is added; if a page is evicted, the entry in the hash table is removed. When referencing a page, its identifier first needs to be translated into a virtual memory offset. Again, this entails an access to the hash table. Because accesses to this hash table have to be synchronized, an increasing degree of parallelism results in high contention. Leanstore addresses this bottleneck by replacing the central hash table with a decentralized bookkeeping method.

The core idea is for each page to be owned by exactly one *swip*. A swip is a reference to a page and can take one of two forms, depending on whether the page is stored on DRAM or on disk. For an in-memory page, the swip holds its virtual memory address; for a disk-resident page, the swip holds a page identifier that can be used to retrieve the page from disk. The two states are encoded in a 1-bit flag as part of the 64-bit swip. This approach is called *pointer swizzling* [251].

In Leanstore, the swip itself stores the information on whether the data is disk- or memory-resident. This is only possible because there is exactly one swip per page. Because an in-memory swip already holds the virtual address of the page, no hash table lookup is necessary. This means that the access to DRAM-resident data is, except for a single *if* statement to check the flag, as fast as a regular pointer access. When disk-resident data is accessed, the page is loaded into the buffer cache and the owning swip is *swizzled*, i.e., replaced with a pointer to the virtual memory address. For eviction, Leanstore uses a speculative second chance algorithm. Pages are randomly marked as *cooling* by unswizzling their swip without evicting them and placing them into a FIFO queue. In this state, their content is still DRAM-resident and accessing them swizzles them again and removes them from the FIFO queue. If, however, they reach the head of the FIFO queue, they are evicted to disk.

The swizzling mechanisms and the one-swip-per-page policy place requirements on data structures used within Leanstore. First, each data structure has to be implemented in a way that stores inter-page references as swips. Second, these inter-page references have to form a tree-like structure. Even for trees, this becomes a challenge, as pointers between the leaf nodes are prohibited [177]. This means that Leanstore cannot easily replace the memory management component of an existing storage layer. Instead, most data structures need to be adapted to Leanstore's ownership principle.

Van Renen et al. [240] propose how NVM could be included in a system like Leanstore. They compare a traditional IMDB with all data on DRAM to (1) an IMDB that primarily stores data on NVM, (2) a system in which DRAM serves as a buffer for data on NVM and SSD, and (3) a system in which NVM is an additional caching layer between DRAM and SSD. To reference data across different tiers, they use the same swizzling approach that Leanstore uses. As a result, the discussions about Leanstore's architectural challenges apply similarly.

Umbra

With Umbra [177], Neumann and Freitag present a DBMS that builds on Leanstore as a storage layer but improves it by allowing for mixed-size pages. Leanstore, as many buffer managers before it, requires that all pages are of the same size. This significantly simplifies the handling of the page cache, as each loaded page can be evicted and replaced by an equally-sized previously evicted page. Umbra relaxes this requirement and allows for multiple page size classes, which are multiples of the system page size and whose size is exponentially growing with a growth factor of 2. The operating system's mapping between virtual and physical pages is used to prevent fragmentation. The advantage of mixed-size pages is that larger data structures can be stored within one page: "*If we can rely upon the fact that a dictionary is stored consecutively in memory, decompression is just as simple and fast as in an in-memory system*" [177].

To reference data, Umbra uses the same pointer swizzling / swip concept used by Leanstore. It continues to require all data to be organized in a tree-like structure. To satisfy this requirement, Umbra stores tuples in the form of B+-trees. These tuples, in turn are stored in a PAX-like layout [6]. For evictions, Umbra adopts Leanstore's page replacement method, which is based on a second-chance FIFO algorithm.

Together with Leanstore's page management approach, Umbra also inherits what we consider to be a disadvantage for the flexibility of the DBMS: all data structures have to be re-implemented to allocate memory on a per-page basis and to use swips for inter-page references. This is in contrast to our PMR-based implementation (cf. Section 5.3), which allows us to include arbitrary data structures in our tiering concept.

Above the storage layer, Umbra shares many concepts with HyPer [120]. Similarly to the latter, it uses just-in-time compilation for query execution. However, instead of compiling the plan into a single binary, Umbra disassembles the plan into multiple steps that can be executed in parallel. Umbra also inherits HyPer's *morsel* concept, which Hyrise used as an inspiration for the chunk concept (cf. Section 4.3.1). We compare the performance of Hyrise and Umbra in Section 4.6.

Mosaic

Mosaic [243] by Vogel et al. is an alternative storage layer for Umbra. Its goal is to support diverse storage devices and to distribute data across the devices in a way in which the performance of the system is maximized. Mosaic focuses on the spectrum between the fastest SSDs and the slowest HDDs. As such, Mosaic solves a problem that is different from the systems described before and the approach presented by us. We include Mosaic in the discussion of related work because its placement strategy works under assumptions similar to our work and is more sophisticated than the solution presented in Chapter 7.

Mosaic is specifically designed for scan-heavy workloads whose performance is bound by the available I/O bandwidth. Unlike existing approaches, Vogel et al. present a *tierless* storage engine. While other systems, including Hyrise, try to maximize the utilization of the faster tiers before resorting to lower tiers, Mosaic distributes data across all devices in a way that maximizes the cumulative bandwidth.

Data is *moved* with the granularity of *column chunks*, which are comparable to the segments used in Hyrise (cf. Section 4.3.1). For Hyrise, we use the same granularity for our migrations and accesses. Mosaic only *tracks* access patterns on the granularity of columns and considers columns to be atomic for the purpose of the placement algorithm. As such, while column chunks could be moved independently, the access skew across column chunks is ignored. This decision was made to reduce the complexity of the linear programming problem.

Mosaic can additionally consider several compression schemes and seek a balance between the I/O bandwidth saved by the compression and the CPU cost of decompressing the data. For Hyrise, such a multi-faceted optimization is subject to future work. A second aspect in which Mosaic's decision making is ahead of our current implementation is that its placement decision uses a more sophisticated cost model that incorporates the seek time and the bandwidth of the devices as well as the compression ratio. As such, even though it solves a slightly different problem, Mosaic gives valuable input for our future work.

3.1.6 Further Approaches

The concept of evicting data from DRAM to lower tiers is not limited to relational databases. Jibril et al. [115] describe how a graph database can profit from NVM and how query compilation can be adapted to hide the increased latency of NVM. Panthera [246] is a storage abstraction for Spark [262] that identifies data access patterns and moves data between DRAM and NVM accordingly. Li and Li propose data spilling for Flink [38]. They follow the anti-caching approach [54] and evict cold data from DRAM to disk. Hershcovitch [102] implemented an NVM-aware storage engine for MongoDB.

Several Projects explored how a DBMS can store all data on NVM and deliver on the vision of resolving the "*dichotomy between memory and durable storage in database management systems*" [11]. Götze et al. [88] and Kuznetsov [135] summarize the challenges and existing approaches. In several publications, Alruraj revisited the architecture of traditional buffer managers and logging components [10, 11, 13, 14]. Oukid presented SOFORT [184, 186], a DBMS that removes the traditional logging component and persistently stores data on NVM. This allows for almost-instant recovery. Similar work has been done in our group on the previous version of Hyrise [220, 223].

If NVM is used as a primary means for persistency and no traditional logging is used, a failure of NVM could result in data losses. To guarantee high availability and allow for disaster recovery, the data has to be stored redundantly. Zarubin et al. [263] present a node-local mechanism that duplicates data across multiple NVM DIMMs in the same system. Mojim [266] uses RDMA to replicate NVM-resident data across independent nodes. Vilamb [119] is a user-space library that also duplicates data across multiple nodes but allows for the replication to occur asynchronously.

3.2 Tiering Outside of Database Systems

Instead of implementing tiering in the DBMS, as described in the previous section, other approaches move the responsibility of identifying access patterns to other layers, including the operating system or additional abstraction layers below the DBMS. Pupykina and Agosta surveyed several of these techniques [197].

3.2.1 Paging

One of the most important features of an operating system is to make DRAM available to different applications. For this, virtual memory management (VMM) is used. The available physical memory is subdivided into pages that are mapped into the virtual address space of the processes. Supported by the CPU, an application's access to such a virtual address is mapped to the corresponding physical address. This allows multiple processes to be executed in isolation. Modern operating systems allow for the available memory to be overcommitted. If more memory is allocated than is available, pages that have not been recently used are moved to lower tiers. This process is known as paging. If the virtual addresses assigned to these pages are later accessed, they need to be brought back into the physical memory. We discuss this process in greater detail in Section 5.1.

Conceptually, paging appears to address a similar issue, namely, to allow for allocations beyond the DRAM limits. It is, however, not suitable for IMDBs in general and not a solution to our problem in particular. The first issue is the selection of the pages that are evicted. Graefe et al. [89] show that "*the OS VM layer is particularly poor at making eviction decisions for transactional workloads*". Stoica and Ailamaki [230] find that "*default OS paging [causes] unpredictable performance degradation*". Leis et al. [148] cite DBMS manuals that explicitly instruct DBAs to prevent paging. One of the reasons for OS paging being unsuitable is said to be the lack of any pattern identification (e.g., large sequential reads) in the OS paging algorithms.

Stoica and Ailamaki [230] propose to assist the operating system in identifying those pages that should be evicted. For this, they track accesses on the tuple level and divide tuples into *hot* and *cold* regions. The pages in the hot region are then excluded from paging by calling `mlock`. As a result, only pages in the cold region are evicted when the memory pressure increases.

In addition to the operating system making inefficient placement decisions, a second factor that makes OS paging less suitable for answering our research questions is that it is only designed for two tiers. Pages are either in-memory or moved into the page file. Neither multi-tier memories (DRAM and NVM), nor multi-tier storage hierarchies (SSDs and HDDs) are considered.

Influencing the kernel's paging decisions is difficult. Some publications propose modifications to the kernel's memory management module [77, 83]. These require additional kernel modules or even entirely custom-built kernels. For security and stability reasons, kernel modifications are generally avoided in productive settings. An alternative to this is to use user-space libraries. These are discussed in the next subsection.

3.2.2 Library Support

Memkind [215] is a heap manager that allows developers to allocate space on DRAM, NVM, and high-bandwidth memory using a shared interface. Instead of calling malloc, developers call `memkind_malloc` and pass a memory *kind*. This kind can either be one of the listed hardware resources, or a policy such as *highest capacity* or *lowest latency*. In Section 5.2, we explain how Hyrise uses Memkind to allocate space on NVM. Umpire [23] is similar to Memkind in that it provides a homogenous interface to memory operations on heterogenous hardware. It bears some resemblance to the Polymorphic Memory Resources used in Section 5.3 as it supports multiple *memory resources* and *pools* that modify the allocation behavior. Hexe [181] and the SharP Unified Memory Allocator [2] follow an approach similar to Memkind. Unlike Memkind, they allow for data to be split across multiple devices using one of four data-splitting policies. X-Mem [69] uses a two-pass approach in which the application is profiled in a first pass, after which a static "*relative priority of data structures*" is computed. During regular execution (the second pass), this priority is used to allocate data accordingly. To map allocations between the two runs, all allocations have to be statically tagged by the programmer. As noted by the authors, this "*assumes homogeneous behavior within a data structure*", which is incompatible to the access skew seen, e.g., in ERP systems.

None of these libraries provides any migration capabilities. Once space has been allocated, it is the developer's job to identify access patterns and to migrate data accordingly. Several libraries can be used to help with this identification [101, 250]. The lack of transparent and automatic migration capabilities in these libraries can be explained easily: from the perspective of the application, a pointer to a data structure (i.e., a virtual address) must remain valid throughout the migration process. The pointer itself can usually not be modified because the tiering library does not know where the pointer is stored and if there are any aliasing copies of the pointer.

An exception to this is Unimem [255]. While Unimem allows data to be migrated, it heavily restricts where memory can be allocated and how pointers can be copied. Additionally, it requires the "*main loop*", i.e., the most-compute intensive part of the program to be programmatically defined. As such, Unimem requires the application to be built around its memory management paradigm.

A second way to allow for transparent migration would be to employ the virtual memory abstraction, i.e., to migrate entire pages between different tiers. This comes close to the paging approaches presented in the previous subsection. We discuss the advantages and disadvantages of page-based migration in Section 5.4, explain why we decided against such an implementation, and present how Hyrise solves the issue of transparent migrations on the layer above the allocator.

3.2.3 Data Structures and Algorithms

For a long time, data structures and algorithms have been designed to take the characteristics of different memory and storage tiers into account. Well-known data structures, such as the B-Tree family [22, 200] have been designed for a system in which DRAM is a scarce resource. They aim to minimize the number of reads from disk, or more precisely, the number of *pages* read from disk. Traditional disk-based database systems use such disk-optimized data structures to make optimal use of the disk bandwidth and minimizing the read latency. Similarly, special implementations of database operators have been developed to process data larger than what fits into DRAM [52]. Again, these optimize for the hardware characteristics of disks.

With in-memory databases, the surrounding conditions changed. Their assumption is that all data can be stored and processed in DRAM. Previous constraints, such as having to read data with a page-level granularity, disappeared. This prompted the design of new, IMDB-focused data structures and algorithms. In the case of our examples, B+-Trees were adapted to optimize their use of the CPU caches [202], and hash joins were implemented to utilize cache-friendly hash tables with vectorized accesses [205, 267].

In Section 2.2.4, we have already described the additional steps needed to persistently store data on NVM. Again, these requirements resulted in newly developed additions to the B-Tree family [9, 156, 185]. But even if NVM is only used for its additional capacity, it is worthwhile to be considerate of its performance characteristics. Aspects such as NVM's asymmetric read/write latencies, its preference for 256 B block reads, and its higher susceptibility to write contention [112, 260] mean that not everything that was beneficial for DRAM works equally well for NVM.

Daase et al. [50] analyzed how these characteristics affect the development of DBMS algorithms and proposed seven *best practices* for the use of NVM by OLAP database systems. As part of their work, they executed the Star Schema Benchmark [180] both with traditional database operators and with hand-crafted, NVM-optimized algorithms. In both cases, all temporary data structures were stored on NVM. As a baseline, they used Hyrise and our polymorphic memory resources (cf. Section 5.3) to move the joins' temporary data to NVM). They found that the joins in Hyrise took 5.3x longer if all data, including temporary data, was stored on NVM instead of DRAM. By using Dash [155], a hashmap specifically built for NVM, they could reduce the impact to 1.66x. From this, they draw the conclusion that "*future research on random access data structures and operations is essential to achieve a good OLAP performance*". Their work complements our work: we show how stored table data can be transparently moved from DRAM to lower tiers. While this reduces the DRAM pressure, it does not eliminate the possibility of operators requiring more temporary memory than available. Developing join algorithms that can gracefully handle these situations by spilling to NVM would further improve the flexibility of our system.

3.3 Summary

We have presented existing approaches for automatic tiering across memory and storage tiers. The approach that comes closest to our implementation is that of Microsoft SQL Server, which utilizes cursors for transparent data accesses across tiers and leverages multiversion concurrency control for transactional safety. Differences to the existing approaches can be found in the way that accesses are tracked and the granularity of the migrations. Our work is the first to enable autonomous and transparent multi-level tiering for column-oriented in-memory databases and to track accesses and access patterns across both the row and column dimension.

Besides comparing different DBMS tiering implementations, we have discussed approaches that do not requires database-specific knowledge. Neither OS paging, nor application library or tiering-specific data structures provide the functionality needed for efficient data tiering in in-memory databases.

4 The Research DBMS Hyrise

In this chapter, we present the research DBMS Hyrise, which is used as the foundation for our work on automatic tiering. We describe its fundamental architecture and dive into selected implementation details. The focus is on areas in which the architecture and implementation are relevant for the following discussion of our tiering approach.

A first version of Hyrise was developed by Grund et al. as a hybrid database system that combines the benefits of row- and column-oriented approaches [91]. By storing data that is frequently accessed together (e.g., an item's price and the currency), it reduced the number of CPU cache misses. In addition to being used to evaluate these hybrid layouts, Hyrise was used by various other research projects in our group. These focused on other aspects of in-memory database management, including "*work on data compression [25], secondary indexes [76], multi-version concurrency control [221], different replication schemes [222], and non-volatile memories for instant database recovery [223]*" [66]. Over the time, however, it became more and more apparent that the DBMS originally built for research on hybrid row/column layouts was not suited for future research projects. In our previous work, we have identified four key limitations [66]:

- "*Data layout abstractions were resolved at runtime and incurred costs that sometimes had a disproportional overhead.*
- *Prototypical components have been implemented to work in isolation, but did not interact well with other components.*
- *The lack of SQL support required query plans to be written by hand and made executing standard benchmarks tedious.*
- *Accumulated technical debt made it difficult to understand the code base and to integrate new features.*"

As such, we needed a new platform for our research. In 2016, we started a complete rewrite of Hyrise[1], taking our experiences with us but leaving the technical debt behind. In our 2019 EDBT paper, we describe the motivation for this rewrite and the lessons learned in greater detail [66].

One might question why it was necessary to develop a new DBMS instead of using one of the existing systems. Given the existing collaborations between our research group and SAP, the most obvious choice would have been SAP HANA. Unlike most other researchers, we are allowed to access their code repository and have access to their compilation infrastructure. This access has been previously used for the research on a caching structure for aggregates [168]. However, the aggregate cache project was accompanied by SAP colleagues without whom the modifications to HANA would have

[1] For the remainder of this thesis, unless explicitly referring to *the previous version of Hyrise*, we use the name *Hyrise* as a reference to this rewritten platform. We are well aware of the ambiguities this causes for people who discuss Hyrise and, in retrospective, should have chosen a different name.

been difficult. Even with this support, we found that the implementation of hardware-level optimizations [64, 65] became too difficult for a quick evaluation. Furthermore, some concepts of SAP HANA that we wanted to challenge, for example the limitation to two levels of partitioning [225], were too deeply engrained in HANA's architecture for us to experiment with different approaches. Finally, using a proprietary DBMS would have made it difficult to publicly share our results in a reproducible way. Still, Hyrise shares many design principles with SAP HANA. This allows us to continue the fruitful exchange of ideas and results with the SAP teams.

4.1 Requirements

Having decided to develop a new research DBMS, we needed to identify the fundamental requirements that would guide our architecture. In this section, we list the functional and non-functional requirements and describe how we have fulfilled these. For the distinction between functional and non-functional requirements, a number of definitions exist [84]. Only few of these definitions are applicable to a research DBMS that is intended to be used for benchmarking different implementations and not for actually being used in production. The definition for non-functional requirements that we found most useful for our purposes has been given by Kotonya and Sommerville [131]:

"*Requirements which are not specifically concerned with the functionality of a system. They place restrictions on the product being developed and the development process, and they specify external constraints that the product must meet.*"

4.1.1 Functional Requirements

Vice versa, functional requirements are those that define the functionality of our DBMS. The basic functionality is the same for all databases, namely, to store data and to return it when queried. In our case, we broaden the definition of functional requirements to include those features that are required in order to use Hyrise as a platform for our research projects.

F1 - Relational, In-Memory, and Column-Oriented

Considering our group's involvement in the early steps of the HANA project, it should not come as a surprise that Hyrise and HANA share fundamental principles. Both are SQL-based relational databases [43]. They primarily store data in main memory but support additional features to include non-DRAM tiers. For HANA, this is the support for Persistent Memory as well as the Native Storage Extension (NSE); for Hyrise it is the automatic tiering presented in this thesis. Furthermore, both systems are primarily column-oriented for the reasons listed in Chapter 2.

Having written tens of papers and a textbook [194] about relational, column-oriented IMDBs gave our group a significant head start for the development of our new version of Hyrise. Second, aligning our requirements with the principles of SAP HANA allows us to build on the past experience of our group allows us to continue the exchange of ideas with the SAP HANA team. We believe that database research profits from a regular exchange between researchers and commercial developers. Doing so improves our understanding of those problems actually seen in productive settings. All too often, these diverge from what is discussed in the research community [244].

At the same time, while sharing the three named principles, we diverge from the design of HANA in a number of ways. This includes our storage concept, the focus on autonomous data management, and the design of the execution engine.

F2 - End-to-end SQL Support

In order for our research results to be transferable into productive settings, our setup has to be as realistic as possible. While some aspects of data management can be evaluated in micro-benchmarks, these have been rightfully criticized as neglecting the larger picture and as showing results that are skewed with regards to local and global costs [160]. Early on, we have decided for end-to-end evaluations and against microbenchmarks in which components are tested in isolation. From a user perspective, this can be described as being able to execute a SQL-based workload end-to-end.

Internally, this brings a number of additional requirements. Supporting SQL does not only include parsing the query string, but more importantly also includes its transformation into a query plan and the optimization thereof. For Hyrise, we have implemented a comprehensive optimizer that features most state-of-the-art optimizations for the TPC-H benchmark as well as additional optimizations [63]. To further increase the degree of realism, Hyrise provides support for the PostgreSQL network layer, which allows us to execute external workloads by connecting to Hyrise via ODBC or JDBC.

Our requirement for enabling end-to-end benchmarks does not mean that we build a feature-complete DBMS. Examples for features that were excluded because they are not in the focus of our research interests are authentication or character sets and collations.

F3 - Flexible Physical Layout

The performance of databases is, to a large extent, influenced by physical design decisions. In our group, we research how better designs can exploit the hardware more efficiently. For example, storing data in a columnar layout and employing dictionary compression make it possible to implement full table scans as vectorized scans on a list of integers. This in turn allows for unparalleled scan performance [67, 252, 253].

To be suitable as a platform for research on such properties, the DBMS has to be flexible enough for explorative changes to its physical layout. The introduction of multiple tiers should not disrupt the architecture of the system. Furthermore, it should be possible to selectively apply different configurations to small parts of the data.

This requirement is shared with other research projects in our group: in the case of automatic compression selection [25], we apply different compression schemes to parts of the data, depending on their data and usage characteristics. In the case of automatic partitioning, we require support for a fine-grained physical layout that can handle multiple partition dimensions with explicit and implicit partitioning criteria.

F4 - Self-Driving Capabilities

The overarching theme of our group's database research projects is *autonomous data management*, or, in more colorful words, the goal of building the *self-driving database*. This goal is motivated by the realization that both database systems and databases themselves become bigger and more complicated. As seen in the case study presented in Section 1.1, database administrators (DBAs) can no longer be expected to know each facet of their workload or each knob in their DBMS. As such, we believe that database systems need to relieve DBAs from most tuning tasks [127, 130]. Our work on automatic tiering, which we present in this thesis, is part of that effort.

To fulfill these tuning tasks, self-driving databases require improved or new features for internal consumption. These include access tracking, cost modeling and benefit estimation, workload modeling and prediction, as well as tuning algorithms [130]. Additionally, for as long as DBAs and self-driving capabilities coexist, the DBMS should provide means for DBAs to monitor, supplement, and override decisions made by the self-driving database [128].

4.1.2 Non-Functional Requirements

In addition to the previous functional requirements, four non-functional requirements have been identified. While the previous requirements described which features the DBMS should support, the following describe what the DBMS must "look like" to be useful as a research platform:

NF1 - Competitive Performance

Hyrise is designed to be a system that enables research on new concepts without having to navigate the complexity of a productive DBMS. As Hyrise is not used as a product itself, the results of our research are only of value if they can be transferred to other systems. If a new algorithm or implementation yields improved results in Hyrise, these results should also be seen when implemented in a similar DBMS. For this, the performance of Hyrise shall be measurably comparable to that of similar database systems.

NF2 - Reproducibility and Comprehensibility

> *"An article about computational science in a scientific publication is not the scholarship itself, it is merely advertising of the scholarship. The actual scholarship is the complete software development environment and the complete set of instructions which generated the figures."*

This ideal, given by Buckheit and Donoho [35], is the distilled version of an article by Claerbout and Karrenback [41]. Following its spirit, we believe that a research DBMS should allow for the published results to be reproduced. Our benchmark tools, which are described in Section 4.6.1, make it possible to execute benchmarks in a single step. Compared to other systems, this relieves the reader from having to generate and load data, starting multiple processes, or even having to execute queries manually. Furthermore, we believe that readers should be empowered to fully dissect the steps that led to these results. To allow for this, we publish both the source code[2] and the modifications made for individual research papers [63].

It is not enough for the source code to be technically accessible, but it needs to be comprehensible, too. For this, we enforce the code quality through automatic code analysis and strict code reviews. While comprehensibility is hard to quantify [36], we track this requirement by using Hyrise not only as a research but also as a teaching platform. Since the rewrite, 93 master's students have participated in four iterations of our project seminar. Most of these students knew about the user-facing principles of databases, but none have developed a DBMS before. In the seminar, they worked on the Hyrise code base and were able to contribute new code within weeks. We take this as an indicator that the code base is comprehensible enough to be understood by outsiders.

NF3 - Modularity

A research platform needs to support experimentation with different components. It should be easy to replace parts of the system in order to evaluate new approaches. For this to be possible, the system must be designed in a modular way and handle modified components gracefully.

In Hyrise, this modularity can be found in many places: in the storage layer, compression algorithms can simply be added. Our compile-time abstraction layer automatically handles the code generation for all database operators. Because of this, no operator code needs to be modified when adding a new compression scheme.

In the optimizer, rules can be added and removed arbitrarily. While some rules perform better if certain plan optimizations have been previously made by other rules, there is no requirement for any rule to be executed before the query can be evaluated. In fact, the entire optimizer can be disabled. A current research project looks at how this flexibility can be used to reduce the operator costs for short-running transactional queries. Similarly, the scheduler can be easily replaced. Hyrise can even run without a scheduler at all, in which case tasks are linearized and executed in an order that is guaranteed to make progress. This modularity allows us to run experiments both with a reduced as well as with the full feature set without any modifications.

NF4 - Decoupling of Research Projects

Closely related to the idea of modularity is the requirement of having research projects decoupled from the core database. Again, this is based on experience gained during the work on the previous version of Hyrise. We found that indiscriminately admitting code to the master branch hinders future research. While this is obvious for code that does not meet code quality criteria, it also applies to code that is fine on its own, but either (1) is relevant to only a minority of users, (2) works only with tightly controlled

[2] `https://github.com/hyrise/hyrise`

parameters, or, (3) even worse, becomes stale. We thus require that individual research projects are separated from the DBMS core project.

Doing so shifts their maintenance burden from the DBMS maintainers to the individual researcher(s). If an individual research project becomes stale, this does not affect the future development of the DBMS core as no project code has to be maintained.

This also applies to our work on automatic tiering. Both the option of having data on NVM in the first place and the algorithm that decides what data should be moved there would not be considered a core functionality of a research DBMS. In Sections 5.3 and 7.1, we describe how these were implemented in a way that fulfills the requirement of decoupling research projects.

4.2 Architecture Overview

Figure 4.1 depicts the high-level architecture of Hyrise. It consists of the *Hyrise Core* and *Plugins*. The Hyrise Core provides the features traditionally expected from a DBMS as well as the foundation for the self-driving capability. The self-driving features, such as the automatic tiering, are encapsulated in plugins. These plugins can be stored in their own code repository and are loaded at run-time. This increases the modularity of the system and decouples individual research projects. For better orientation, we have highlighted and numbered five areas. The components that are not highlighted (SQL Objects and Plan Caches) are not discussed here as they are not relevant for the remainder of the thesis. For these, we refer to our previous work [66, 129].

Starting at the top left, the user has ① three options to communicate with the database. The CLI[3] Console is the easiest way to interact with the database. Not only does it allow for the execution of queries, but it also supports the in-line generation of benchmark data and the visualization of queries. The second option is to use the PostgreSQL-compatible network interface, either via the psql client or via PostgreSQL-compatible libraries. Finally, the benchmark binaries can be used to execute standard benchmarks with a single command and without having to prepare input data or post-process output data.

The ② SQL Pipeline is responsible for executing SQL queries. It describes not just a concept but a class that encapsulates all steps that are necessary to turn a SQL string into a result table. This makes it both easy to execute queries from within Hyrise and to follow the queries' path through the system. In Section 4.4, we describe the pipeline and its steps in greater detail.

Naturally for a relational DBMS, the ③ storage layer is built around the tables. Hyrise horizontally partitions tables into *chunks* that have a fixed number of rows. New rows are inserted into the last chunk. Updates are realized as a combination of invalidations and inserts. For this, the *MVCC information* tracks the visibility of the rows. Besides the primary data (tuples and MVCC information), tables can store secondary data. This includes inverted indexes for faster lookups, filters that make it possible to skip

[3] Command Line Interface

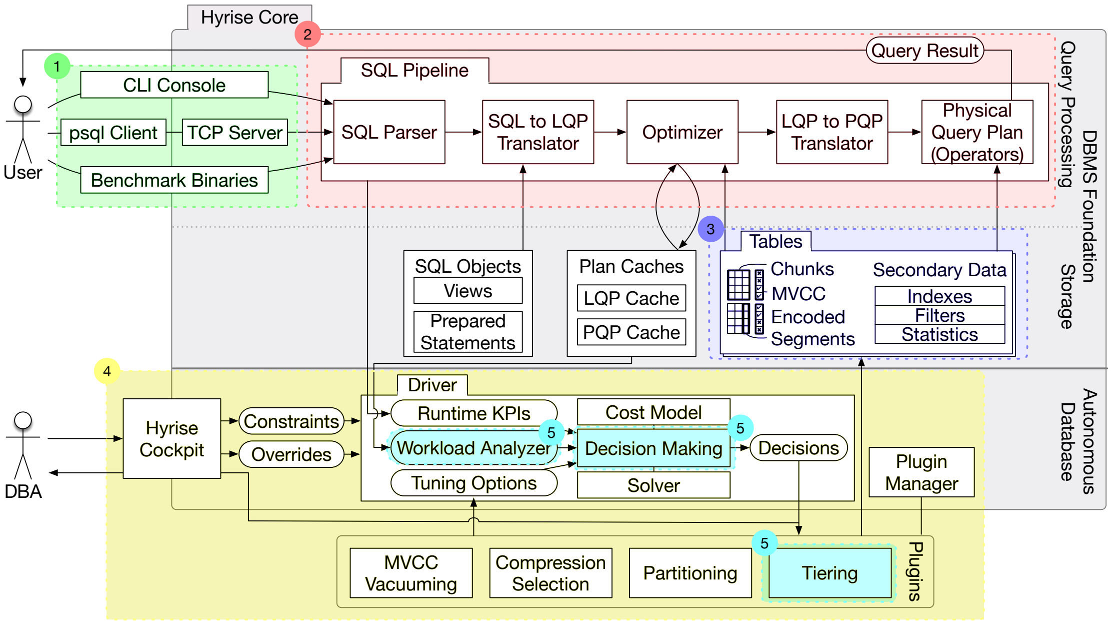


Figure 4.1: Architecture diagram of Hyrise.

chunks during execution, and statistics that are used by the query optimizer and some self-driving components. Section 4.3 describes the physical table layout in greater detail.

Next, we look at the 4 components designed to make Hyrise a self-driving database. Much of this is developed as part of separate PhD theses. As such, it is discussed here for completeness. The center of our approach is the *driver*. It consolidates runtime information from different parts of the system, such as KPIs from the SQL pipeline, or workload statistics from the plan caches. Together with the different tuning options, this information is fed into the decision making algorithm. This algorithm uses an internal cost model to estimate the costs and benefits of a solution and uses an arithmetical solver to reach tuning decisions. These decisions are then applied. Tuning options can be provided by plugins. These are kept separate from the Hyrise core in order to isolate individual research projects. They can be loaded and unloaded on-the-fly via the Plugin Manager. While the system is designed to operate autonomously, DBAs can monitor its decisions and influence the decision making by supplying constraints and overrides via the Hyrise cockpit [128].

Three parts of the self-driving component are highlighted 5. These correspond to three contributions made in this thesis: our memory management framework (Chapter 5) is largely encapsulated in the tiering plugin. As such, it can be individually enabled and is decoupled from the Hyrise core. Next, the access tracking method described in Chapter 6 is part of the Workload Analyzer and can be used by all plugins alike. Finally, the decision making algorithm presented in Chapter 7 interpretes the reports of the access counters and decides which parts of the data are stored on which tier.

4.3 Physical Table Layout

The table layout was designed to fulfill the requirement of a flexible storage layout. Understanding how data is organized in Hyrise is important for the following discussion of our data tiering approach as the architectural decisions made here influence the granularity on which the access tiering and the data migration operate.

4.3.1 Chunks and Segments

In Hyrise, tables are horizontally partitioned into *chunks*. This is displayed in Figure 4.2. A table starts with a single chunk into which data is inserted. Within a chunk, a *segment* exists for each column of the table. Once the chunk reaches a threshold size[4], a new chunk is appended to the table and future inserts are written at the end of that new chunk. Similarly to other database systems that follow the insert-only paradigm [204, 257], data is never modified once written. Instead, updates and deletes mark rows as invalid, which excludes them from future query results. An asynchronous vacuum process later removes those invalid rows.

Chunks can be, and regularly are, compressed. Hyrise supports a number of compression mechanisms, namely dictionary compression [1], run-length compression [206], LZ4, and Frame-of-Reference [86]. In the context of this thesis, we exclusively use dictionary compression for all data. A dictionary-compressed segment consists of a vector that holds all distinct values within that segment in sorted order (the dictionary) and a list of offsets (also called value ids) into that dictionary, called the attribute vector. Not only can dictionary compression reduce the footprint of the segment, but it also greatly improves the performance of scans [67, 253].

The decision to use chunks was inspired by a paper by Leis et al. [147]. They describe *morsel-driven parallelism*, in which tables are subdivided into so-called morsels that operators can work on independently. Depending on the system load, the degree of parallelism can be chosen at run time by varying how many morsels are processed by a single worker. Based on their promising results, we adopted this concept for Hyrise. Besides being useful for multi-threading, we have identified additional benefits of having an architecture-level horizontal subdivision of tables:

- Compression: Instead of requiring an entire column to be dictionary-compressed, different compression methods can be applied to individual segments. This can be used to reduce the data footprint. The decision which method to use can be based on the data distribution or the access pattern. Automating this decision is studied as part of a separate PhD thesis.
- Pruning: By storing statistics on the chunk-level, the query optimizer can identify chunks that do not contain data that is relevant for a given query. Excluding these chunks as part of the optimizer significantly reduces the amount of data accessed in the execution phase.
- Indexing: Instead of requiring a single index over the entire table, Hyrise allows for chunks to be indexed individually. Because all but the last chunk are immutable,

[4] Currently set at $2^{16} - 1 = 65\,535$ rows.

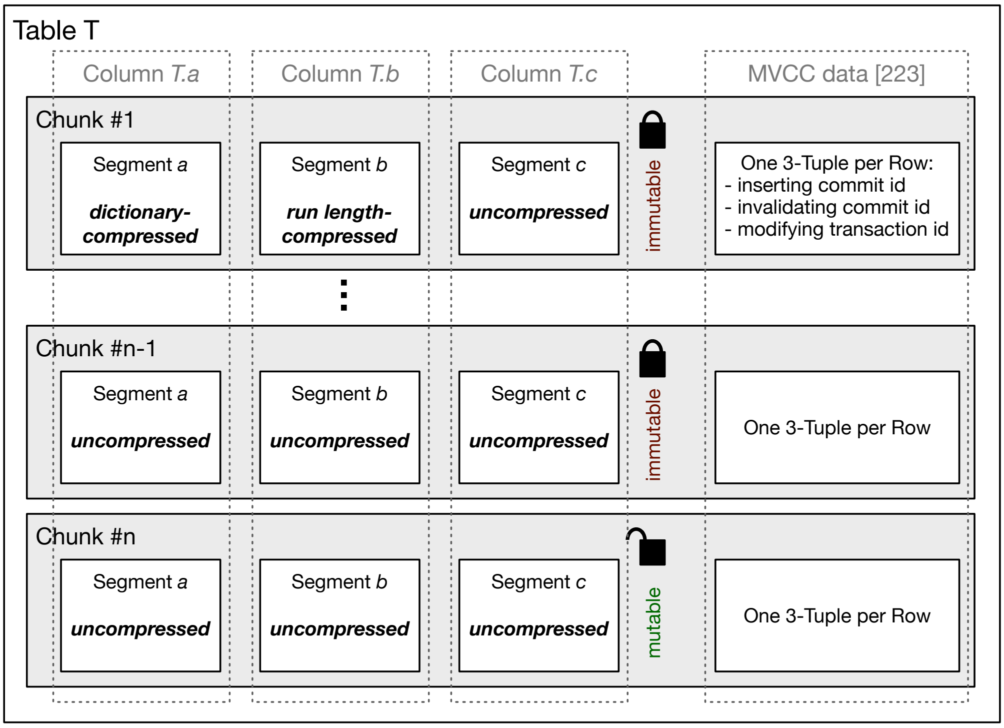


Figure 4.2: Chunks are horizontal partitions of a table. Within a chunk, segments represent the chunk's part of a column. Data is always appended into the last chunk. Once this chunk reaches a certain size, it is marked as immutable and can be asynchronously compressed. [66]. The visibility of rows is controlled by the 3-tuple in the MVCC data vectors. These vectors are always writable, even if the chunk is immutable.

this allows for the corresponding indexes to be immutable, too. Having immutable indexes greatly reduces the index maintenance cost. Together with chunk pruning, immutable indexes also allows for semi-indexed tables, in which only the chunks that are regularly *not* pruned from the result are indexed [249].

- Tiering: By dividing the table horizontally into chunks and vertically into segments, we achieve a great degree of flexibility when it comes to moving data between memory and storage locations. This is used in the following chapters to independently move less-frequently used segments to lower tiers.
- Partitioning: While all but the last chunk are immutable in the sense that no data is ever inserted into them, this does not mean that their data cannot be physically reorganized. This is done as part of the previously mentioned vacuum process, but it can also be used to partition the data by arbitrary criteria.

4.3.2 Iterators

Allowing compression methods to vary on a per-segment basis adds a great degree of flexibility to the database. At the same time, it increases the complexity of accessing data. To fulfill our requirement of modularity, this complexity should be decoupled from most of the code base. For example, for the `Projection` operator, which is responsible for calculating an expression such as `amount * price`, the work is the same no matter whether dictionary or LZ4 compression is used. As such, the compression's details should be hidden from the execution engine. For this, the storage layer has to provide an abstraction layer that allows developers to implement algorithms without being aware of the data's physical representation.

Hyrise provides such an abstraction layer in the form of *iterators* [26, 114]. The key to this approach is its use of static polymorphism, which means that almost all indirections can be resolved at compile time. Boissier et al. showed that this reduces the overhead of the indirection for dictionary segments by more than 3× [25] compared to an object-oriented implementation with dynamic polymorphism.

With iterators, the code needed for an operator to access all values of a segment can be as easy as shown in Listing 4.1. Note that the iterator approach neither requires the data type nor the compression method to be explicitly specified. Instead, they are retrieved from the segment's meta data through compression-specific code that is auto-generated using template metaprogramming.

At the same time, some operators benefit from knowing the physical representation. For example, a full table scan on a dictionary-compressed segment can be performed by first looking up the value id of the search value and then searching for that value id in the attribute vector. By doing so, the segment does not have to be fully decoded. The abstraction layer provides an interface that allows developers to provide specialized implementations for single data types or compression mechanisms. During compilation, these are then compiled into the operator instead of the general-purpose iterators.

The iterators form the basis of our access counters. We revisit the iterators and describe how they enable light-weight tracking in Chapter 6.

```
auto segment = table->get_chunk(0)->get_segment(0);
segment_with_iterators(segment, [](auto it, auto end) {
  for (; it != end; ++it) {
    // decltype(it->value) is deduced based on the segment's
    // data type
    std::cout << it->value << "\n";
  }
});
```

Listing 4.1: Example code for accessing the data stored in a segment through iterators.

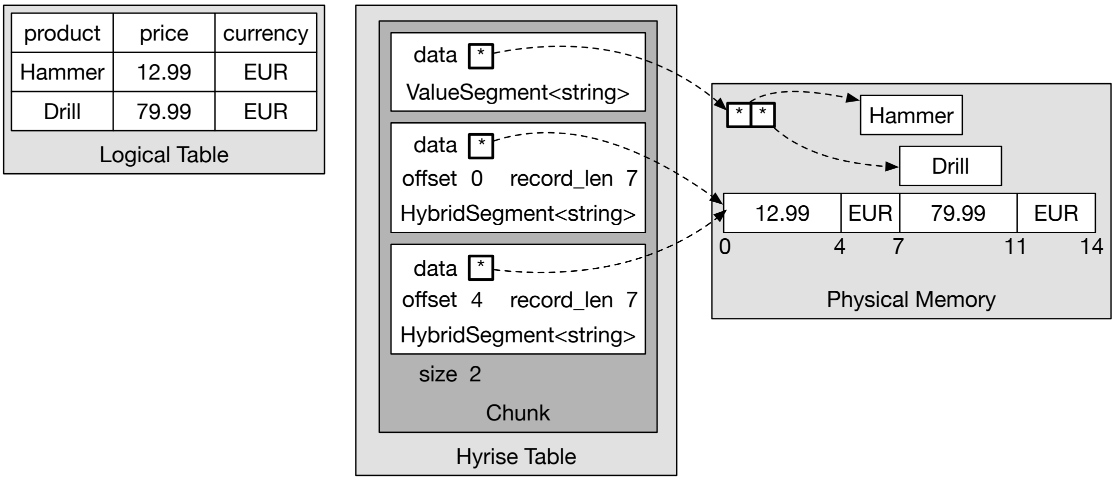


Figure 4.3: The chunk layout is flexible enough to support hybrid row- and column-based layouts. In the example, the price and currency columns are stored in contiguous memory to improve the locality of memory accesses.

4.3.3 Hybrid Row- and Column-Based Layout

The previous version of Hyrise was developed to evaluate hybrid row- and column-based data layouts [91]. Its idea was that columns that are frequently accessed at the same time could be stored together. For example, by storing an item's price and its currency in the same cache line, both can be retrieved at the cost of a single memory access. At the same time, as unrelated columns are stored separately, the hybrid approach still has the column store's benefit of not reading unrelated data when scanning a column.

For the current implementation of Hyrise, this hybrid approach has not yet been implemented for two reasons: first, for most of our group's research projects, the hybrid approach was found to be an orthogonal optimization. We thus did not yet spend the time to re-implement it. Second, while the original paper has often been cited as an example of such a hybrid layout, the concept has not been widely adopted in commercial or open-source database systems. We suspect that this is the case because its advantages do not justify the increased architectural complexity both in the execution engine and the optimizer.

Still, the chunk-based storage layout of Hyrise would allow for hybrid partitioning to be re-introduced. This could be done by allowing multiple segments to share an underlying block of memory, as seen in Figure 4.3. Iterators would take care of accessing the data at the correct offsets, thus encapsulating the hybrid layout in the storage engine and allowing operators to access these hybrid tables without any modifications to the operator code. Again, this highlights the flexibility of the iterator concept.

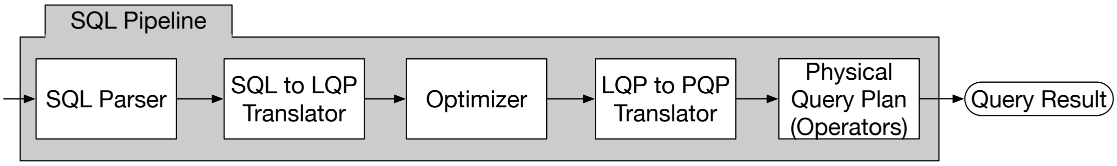


Figure 4.4: Steps involved in the execution of a SQL query (excerpt from Figure 4.1).

4.4 Query Parsing and Execution

Having described how data is organized in memory, we now describe how it is queried. We outline the steps taken from receiving an SQL query to returning a table that contains the query result. The `SQLPipeline` serves as a frontend for this process. It is the key component for fulfilling the requirement of end-to-end SQL execution. Callers, such as the psql server, the console, or unit tests, create a pipeline with an SQL input string and simply call `get_output_table`. This method returns a `Table` object. The pipeline object automatically handles transactional safety, the execution of multiple queries within a single string, and provides methods for inspecting the runtime metrics of the execution. Internally, the `SQLPipeline` performs a number of transformation and optimization steps. The involved components and the intermediary artifacts are shown in Figure 4.4 and are described in further detail in the next sections. We follow an architecture pattern commonly found in other database systems [82, 166]. As such, we discuss most of these components only to the degree to which they are relevant for the following discussions.

4.4.1 SQL Parsing

The first step in executing an SQL query is to parse the incoming SQL string into an abstract syntax tree (AST). We use GNU flex[5] as a lexer generator and GNU Bison[6] as a parser generator. The initial implementation of the SQL parser[7] was the result of a project in our group's database research seminar. At that time, no easy-to-use open-source and stand-alone C or C++ SQL parser was available. The parser has since been extended to support more complex features as needed, e.g., for the TPC-DS benchmark. Even though the parser was not originally planned as a stand-alone component, it has been used externally in research [217], the open-source Envoy MySQL Proxy, and by several other Github users with undisclosed commercial affiliations. It also continues to be the number one Google result for "C++ SQL Parser" and the most-starred repository on GitHub tagged as sql-parser[8].

[5] `https://www.gnu.org/software/flex/`

[6] `https://www.gnu.org/software/bison/`

[7] `https://github.com/hyrise/sql-parser`

[8] `https://github.com/topics/sql-parser`

4.4.2 Translation from Parsed SQL to Logical Query Plans (LQPs)

Once the SQL string has been parsed, the different syntax elements are interpreted semantically. This includes the lookup of identifiers (mostly table and column names) in the catalogue, but also more challenging tasks such as the mapping and resolution of potentially shadowed identifiers. This task is performed by the `SQLTranslator`. It takes the AST generated by the SQL Parser and converts it into a *Logical Query Plan (LQP)*. An LQP consists of edges, which represent intermediary results (i.e., tables), and nodes, which are loosely related to operators in the relational algebra [43]. For example, the `PredicateNode` represents the *restriction* operator.

Most node types have features that go beyond Codd's definition of the relational algebra. This includes outer joins, subqueries, and more complex predicates such as `IN`. Also, unlike Codd's definition, LQPs take the form of a directed, acyclic graph (DAG) instead of forming a tree. This facilitates the reuse of intermediary results.

A number of LQP nodes take *expressions* as parameters. The simplest examples of such expressions are columns or scalar values. These can then be built upon to form more complex expressions, including arithmetic and logical operations, functions, nested logical expressions, as well as correlated and uncorrelated subqueries.

While SQL is a declarative language, the LQP is an imperative description of how to obtain the result table. As such, the order in which joins and predicates occur in the LQP heavily influences the execution cost of the query. Reducing this cost by modifying the LQP is the job of the optimizer.

4.4.3 Optimization

Query optimization has been described as "*absolutely essential for virtually any database system that has to cope with reasonably complex queries*" [176]. The importance of this step can hardly be overstated. In our analysis on the impact of different query optimizations on the TPC-H benchmark [63], we found that several optimization steps have an impact of multiple orders of magnitude.

The Hyrise optimizer consists of a number of rules (17 at the time of writing), which transform the LQP from one valid representation of the original query into a more efficient, but semantically identical representation. This new representation is not necessarily expressible in SQL anymore. An example of this is the use of exclusive *between* operators: while SQL only provides `WHERE x BETWEEN a and b`, which includes both a and b, the *between* expression used in the LQP can also support intervals that are open on either or both sides. As such, a predicate such as `o_orderdate >= '1996-07-01' AND o_orderdate < '1996-10-01'` can be rewritten into a single predicate in which `o_orderdate` is only accessed once. Identifying this optimization opportunity, even if the two predicates are at different positions in the LQP, is the job of the BetweenCompositionRule. Instead of describing all remaining rules in detail, we refer to our previous work [63].

4.4.4 Translation from LQPs to Operators

The individual LQP nodes do not contain any code needed to execute, e.g., a join. This code is part of the individual operators that constitute the execution engine. LQP nodes do not always have a 1:1 mapping into operators. For example, a scan node can be implemented in the form of a full table scan or an index lookup. Translating LQP nodes into operators and, in the bigger context, logical into physical query plans (PQPs), is the job of the `LQPTranslator`.

There are two reasons for this separation: first, it makes for a clear separation of concerns and allows for the logical plan (including its optimizations) and the operators to be tested individually. Second, this approach makes it easy to provide multiple implementations for a single LQP node. As described above, a join can be executed using different algorithms. In Hyrise, (1) hash joins, (2) sort-merge joins, (3) index joins, and (4) nested loop join are supported. The optimizer identifies the most efficient algorithm for a given join and stores a corresponding hint in the `JoinNode` describing which strategy should be used. Based on this hint, the `LQPTranslator` instantiates the appropriate join operator.

4.4.5 Execution Engine

Once it is translated into a physical query plan (PQP), the query can finally be executed. Hyrise follows an operator-at-a-time model, as made popular by MonetDB [28]. In this model, plans are executed bottom-up so that operators (e.g., scan, join) are executed before the succeeding operator is started[9]. This approach was chosen because it enables an easier reuse of intermediary results, is easier to debug, and is better suited for *intra*-operator vectorization. While we have experimented with *inter*-operator vectorization [67], this is currently not part of Hyrise.

Instead of fully materialized results, the output of most operators consists of *position lists*, or short *PosLists*, which are indirections into the original table. Figure 4.5 shows two consecutive scans `colB > 4` and `colA < 10`. The original data is stored in uncompressed `ValueSegments` (VS). A scan of these segments results in `ReferenceSegments`. The table scan operator produces one chunk for each input chunk that contains at least one result. This allows for easy parallelization of the operator. To reduce the cost of producing the PosLists and their memory consumption, PosLists can be shared between `ReferenceSegments` in the same output chunk. The indirection introduced by the ReferenceSegments is resolved as late as possible in order to reduce the number of materializations: when a following operator accesses a value, that value is materialized on-demand.

[9] This still allows for multiple operators to be executed in parallel if their input dependencies are independently satisfied.

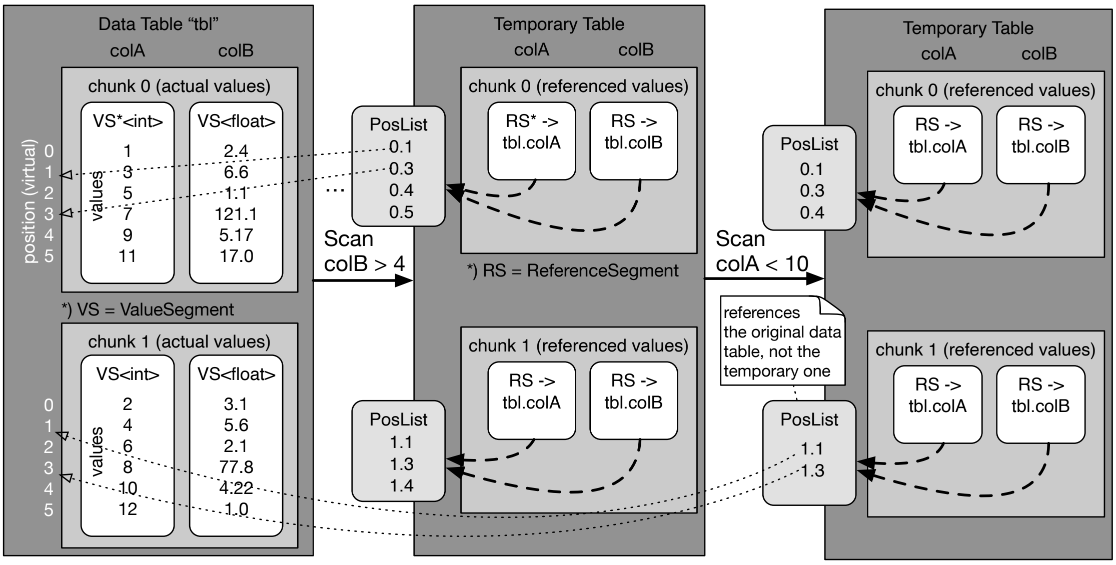


Figure 4.5: Temporary tables as produced by operators contain position lists (short: *PosLists*), which are indirection into the original table.

4.5 Multi-Threading

Hyrise supports both inter- and intra-query parallelism. In case of the former, multiple queries are executed in parallel; for the latter, operators (or parts thereof) within a single query are parallelized. In both cases, the unit of work is a *task*. Tasks can spawn other tasks themselves. For example, the table scan operator spawns one task per scanned chunk so that the table is scanned in parallel.

Tasks are scheduled by adding them to a *task queue*. Workers (one per CPU core) take tasks from their assigned queue and execute them. If the queue is empty, a task from a different queue is taken. This is known as *work-stealing*. Using fewer queues increases the synchronization overhead while having too many queues increases the scheduling overhead. For Hyrise, having one task queue per NUMA node was found to be optimal.

In our research, we find it important to be able to selectively use optimizations, including multi-threading. As such, Hyrise can also run in a single-threaded mode by using a pseudo-scheduler that does not spawn additional threads but instead flattens the task dependency graph to a linear execution order and completes one task after another.

On the concurrency side, Hyrise uses an approach based on Multi-Version Concurrency Control with Group Commit as originally developed for the first version of Hyrise [221]. This approach has been found to be one of the more scalable approaches out of all concurrency methods analyzed by Wu et al. [257]. Compared to the originally described approach, we have added some further optimizations that make use of the chunk-based layout of the new version of Hyrise. These skip the validation of chunks that are entirely visible or invisible to the current transaction. Especially in analytical workloads, in which old data is rarely, if ever, updated, this significantly improves the query performance.

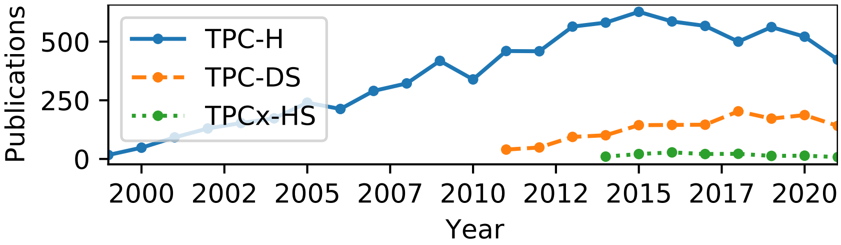


Figure 4.6: Number of publications indexed on Google Scholar referencing "TPC-H", "TPC-DS", or "TPCx-HS", starting with the year of the benchmark's publication [63].

4.6 Evaluation

Having introduced the key concepts of our architecture, we now present a performance evaluation of Hyrise. In this section, we look at the core DBMS without any tiering-specific modifications. This evaluation is done to establish that the baseline performance provided by Hyrise is comparable to that of other research database systems. It also gives some insight into the strengths and weaknesses of Hyrise. At the same time, it is not the purpose of this evaluation to advertise Hyrise as being superior to any of the compared systems. Too many factors play into these benchmarks for the comparison to be used as any type of advertisement [198].

4.6.1 Benchmarking and Reproducibility

Hyrise natively supports a number of publicly available benchmarks. In our research and development, we use these to measure the impact of code modifications on the query latency and throughput. The *TPC-H Benchmark* [239], introduced in 1999, was one of the first extensive OLAP benchmarks. It models a decision support system based around customers, orders, parts, and suppliers. 22 queries are defined and filled with random parameters.

The TPC-H benchmark has been criticized as being partially unrealistic because of its linear and homogeneous data distribution, the use of 3NF instead of a star schema [173], and the lack of complexity in its queries compared to real-world workloads [244]. Still, it is among the most popular benchmarks among academic researchers, as shown in Figure 4.6.

Other benchmarks that are natively supported in Hyrise are JCC-H (a modification of TPC-H that introduces data and access skew) [27], the Join-Order Benchmark [149], and the transactional TPC-C benchmark [237]. The decision support benchmark TPC-DS [238] is partially supported (62 of 103 query variants).

In Section 4.1.2, the second non-functional requirement called for a benchmark setup that makes it easy to reproduce benchmark results. This includes the generation of the data and the queries, as well as the execution and the reporting of the results. In Hyrise, these are combined in the `hyriseBenchmarkTPCH` binary and corresponding binaries for the other benchmarks. These binaries perform all necessary steps for running the benchmark and return the result both in a text and a JSON format. The latter contains not only the runtime of each individual query execution, but also information about the code version that was benchmarked and the chosen benchmark parameters, allowing users to later re-run the benchmark with the same configuration. Further, it is the data basis for most visualizations of benchmark results.

4.6.2 Single-Threaded TPC-H Performance

For the first benchmark, we execute the TPC-H benchmark on MonetDB [28, 100, 109], DuckDB [199], HyPer [80, 120, 147], Umbra [118, 177, 254], and Hyrise. These systems were chosen because they have already been extensively studied by the research community. In the first benchmark, we execute all benchmarks in a single-threaded configuration. This is done because it helps to isolate the performance characteristics of the individual operators. While the multi-threaded performance is at some point capped by the available memory bandwidth, a single-threaded execution cannot reach this cap. Furthermore, it allows us to distinguish the performance of the query execution engine from that of the scheduler.

Benchmark Setup

This benchmark is executed on a 2017 Fujitsu Primergy RX4770 M4 with four Xeon 8180 CPUs (2.5 GHz base, 3.8 GHz turbo, 28 physical cores) and a total of 1.5 TB RAM. The database systems are configured as follows:

- The **MonetDB** team provides a collection of scripts online[10]. We start the server with `set gdk_nr_threads=1` and use their `horizontal_run.sh` script to execute the queries. MonetDB version 11.37.7 is installed using from the official packet sources.
- **DuckDB** (commit 2fa2e62) does not explicitly provide a benchmark facility with tunable parameters. We use a modified internal benchmark script to generate TPC-H data with a scale factor of 10, disable the result validation[11], and run a single query per template.
- For **Hyrise**, the integrated benchmark binary `hyriseBenchmarkTPCH` is used with the `-r 1` parameter to limit the number of query executions and `-s 10` for a scale factor of 10. We present two entries, **Hyrise Paper** and **Hyrise WIP**. The former represents formerly published and peer-reviewed results [63], the latter is based on a yet unpublished version of Hyrise. This work-in-progress version includes the changes since the paper has been published as well as some optimizations that have not yet been merged into the master. This is because the corresponding code reviews and the development of additional tests has not been completed at the time of writing.

[10] `https://github.com/MonetDBSolutions/tpch-scripts`

[11] `https://github.com/hyrise/tpch_paper/blob/master/3_experimental_setup/duckdb.diff`

DBMS	HyPer	Umbra	MonetDB	DuckDB	Hyrise Paper	Hyrise WIP
Total [ms]	19747	16696	27750	94063	73978	39753
rel. to Umbra	1.18	1	1.66	5.63	4.43	2.38

Table 4.1: Summed latency of all TPC-H queries (similar to TPC-H Power Metric).

- Because **HyPer** is not publicly available, we instead cite previously published numbers [72]. Essertel et al. used a benchmark environment that is very comparable to the one that we use for our benchmarks. Most importantly, the used processors are very similar (Xeon 8168 vs. Xeon 8180). The presented numbers use a scale factor of 10, which we thus use for the other systems, too.
- In April 2020, we were kindly provided with a version of **Umbra**. TPC-H is executed by setting the `PARALLEL` environment variable to `off` and running `demo-tpch`. This script executes 22 hard-coded queries once each, so the results are comparable to HyPer in terms of the number of executions.

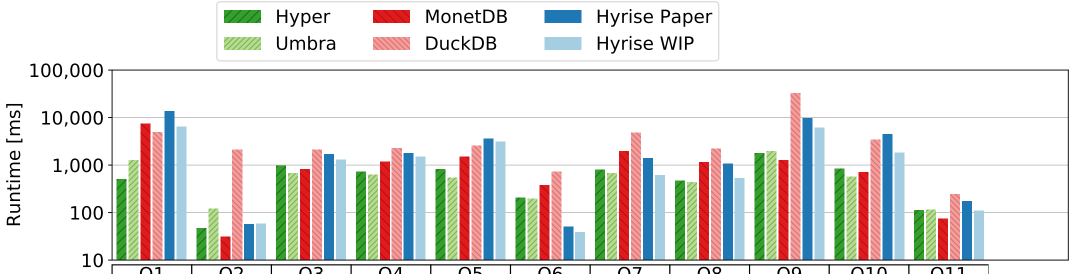
One might argue that instead of executing only a single query per TPC-H query template, we should execute multiple instances of these. Doing so would allow the systems to warm up and cache reoccurring query plans. Indeed, we found that the performance of Query 2 on Hyrise improved significantly after several executions. However, in order to avoid tainting the comparability to the HyPer and DuckDB results, we decided against modifying the developer-provided benchmark scripts or tuning their parameters.

In prior work, we have compared Hyrise with two more database systems [66], namely Quickstep [187] and Peloton [188]. Both systems have since then been discontinued, which is why we exclude them from the comparison. For all queries supported by Peloton, and all but two queries supported by Quickstep, Hyrise is now faster. We further do not publish any results from commercial database systems because of the infamous DeWitt clause [203], which puts researchers publishing benchmark results in legal jeopardy.

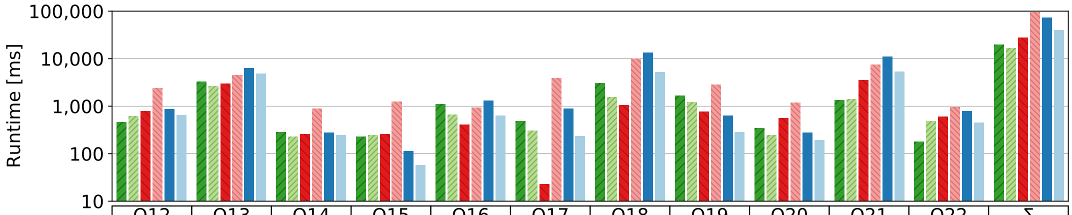
Results

Figure 4.7 shows the result of the single-threaded execution of the TPC-H queries. We show the runtimes of the individual queries as well as the overall sum, i.e., the time to execute each query once. For five out of 22 queries, Hyrise is the fastest out of the five evaluated systems. At the same time, there are two queries for which Hyrise is slower than all other systems.

When **summing the execution time** of all queries (Table 4.1), we find that Hyrise takes 2.38× longer to execute the TPC-H benchmark than the fastest measured DBMS, Umbra. Because both database systems were fully memory-resident, we attribute this not to the storage layer of the systems, but to Umbra's execution engine. While this certainly shows room for improvement, we also believe that it allows us to claim that the single-threaded performance of Hyrise is at least in the same ballpark as other research database systems. It is important to note that in its current version, Hyrise does not use any primary key indexes, which could bring an additional boost to its join performance.



	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10	Q11
Hyper	**498**	47	969	725	821	207	804	467	1,782	846	111
Umbra	1,258	119	**676**	**629**	**536**	196	665	**435**	1,938	**574**	116
MonetDB	7,488	**31**	816	1,185	1,505	372	1,965	1,136	**1,252**	708	**74**
DuckDB	4,874	2,083	2,080	2,283	2,533	723	4,832	2,217	32,681	3,377	241
Hyrise Paper	13,559	57	1,719	1,796	3,579	50	1,404	1,078	9,740	4,496	173
Hyrise WIP	6,376	59	1,293	1,499	3,084	**39**	**609**	525	6,105	1,815	109



	Q12	Q13	Q14	Q15	Q16	Q17	Q18	Q19	Q20	Q21	Q22	Σ
Hyper	**460**	3,307	283	227	1,112	485	3,061	1,666	345	**1,347**	**177**	19,747
Umbra	617	**2,645**	**229**	245	653	301	1,538	1,216	247	1,386	477	**16,696**
MonetDB	786	2,970	258	259	**403**	**23**	**1,054**	765	560	3,535	605	27,750
DuckDB	2,363	4,466	887	1,232	925	3,837	10,039	2,807	1,192	7,452	939	94,063
Hyrise Paper	882	6,388	282	113	1,318	894	13,628	642	282	11,104	794	73,978
Hyrise WIP	651	4,913	249	**58**	629	235	5,207	**287**	**193**	5,366	452	39,753

Figure 4.7: Latencies for a single-threaded execution of TPC-H on different systems.

4.6.3 Multi-Threaded TPC-H Performance

We now look at the multi-threading performance of Hyrise. Unfortunately, the benchmark facilities provided by the systems used for comparison are not flexible enough to execute different types of multi-threaded benchmarks. Most importantly, they do not provide options to run multiple clients (also known as query streams) in parallel. We thus wrote a Python-based benchmark suite to measure the multi-threaded performance.

Benchmark Setup

The multi-threaded benchmark uses the same setup as used in the single-threaded benchmark. For MonetDB, we load the TPC-H data into the database using MonetDB's test scripts and use the `pymonetdb` package to connect to the database. For Hyrise we use `psycopg2`, which is a library that connects to the PostgreSQL-compatible network interface of Hyrise. Umbra has no public Python interface and HyPer was not available to us. For these reasons, the multi-threaded benchmark is limited to MonetDB and Hyrise.

The benchmark executes a shuffled run, in which one or multiple parallel client(s) execute(s) all queries in a random order for one hour. Only full runs are counted and long-running queries have a higher relative impact on the result than smaller queries.

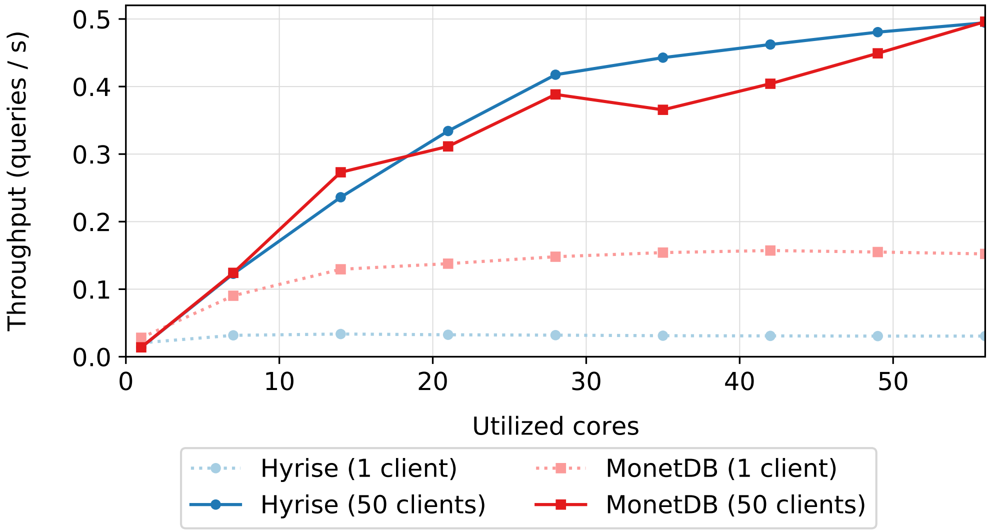


Figure 4.8: Multi-threaded TPC-H results for Hyrise and MonetDB with a varying number of CPU cores and either 1 or 50 concurrent clients (aka. query streams).

To show how the systems perform on a varying number of CPU cores, we limit the number of cores available to the DBMS using `numactl(8)`. This tool takes a range of logical core IDs that a child process should be constrained to. On the test system, logical core IDs are first grouped by the Hyper-Threading sibling id, then by the socket number, and then by the physical core number. As such, when executing a benchmark with a core count varying from 1 to 56, cores 1 to 28 are independent physical cores, while 29 to 56 are the corresponding Hyper-Threading siblings. While MonetDB observes the CPU affinity set by `numactl`, it continues to spawn as many threads as there are cores in the system, which is why we continue to use `set gdk_nr_threads`.

Results

The results of this experiment are shown in Figure 4.8. It shows measurements both for a single client and for 50 clients. With a **single client**, the query throughput of Hyrise does not significantly benefit from a higher number of threads. Reasons for this include the aggregate operator in Hyrise, which has not yet been parallelized, as well as the lack of concurrent bloom filters in the hash join, which results in a sequential filter merge step. MonetDB is slightly more effective when it comes to intra-query parallelism, but also needs more than a single client to reach its full potential.

Looking at the **multi-client** benchmark, the benefits of having multiple cores become more pronounced. Here, Hyrise and MonetDB reach almost the same performance. Both reach their peak performance when all 56 cores are used, with MonetDB maxing out at 0.496 iter/s (with one iteration being a full execution of all 22 queries) and Hyrise reaching 0.494 iter/s.

Our experiment shows that not only the single-threaded but also the multi-threaded performance of Hyrise is competitive for the TPC-H benchmark. In fact, while Hyrise is still slower than MonetDB by 43% in the single-threaded benchmark, it can show its strengths in the multi-threaded benchmark, in which both systems are on eye level. As such, we conclude that Hyrise as a research platform exhibits a reasonable approximation of a productive system's performance.

4.7 Summary

In this chapter, we have presented Hyrise, which is the open-source in-memory DBMS that is used as the experimental platform for our automatic tiering implementation. In Hyrise, tables are organized in horizontal partitions, which are called *chunks*. Each chunk consists of several *segments*, one segment for each column of the table. These segments can be individually compressed. They are accessed via iterators, which serve as an abstraction that hides the compression details of the segment from the database operators. To execute SQL queries, Hyrise translates the queries into logical query plans (LQPs), optimizes these plans, transforms the LQPs into individual operators that are part of physical query plans (PQPs) and executes these operators. Most operators produce *ReferenceSegments*: instead of copying all input values into the operator output, the values in the original table are referenced. This reduces the amount of data that has to be written. In single-threaded and multi-threaded benchmarks, Hyrise can compete with other database systems.

5 Multi-Tier Memory Management and Data Migration

The first pillar of automatic tiering is the handling of different memory and storage types and the migration of data between these. Our goal is for this to be implemented in a (1) transparent and (2) extensible manner that adds (3) little overhead. In this context, (1) transparency means that the different tiers shall not be exposed to the entire DBMS but be contained to the tiering component. Operators should not have to be modified to profit from different tiers. At the same time, to fulfill the requirement of decoupling the research projects, using non-DRAM tiers shall remain just an option. As such, our functionality shall be easy to disable or even remove as needed. Second, our implementation shall be (2) extensible in a way that does not account for only DRAM and NVM, but also allows for existing tiers like SSD and upcoming tiers like disaggregated memory (cf. Section 8.2). Finally, the (3) performance of the DBMS shall not be substantially affected by any abstraction layers. Accesses to data on DRAM shall not be measurably slower than in a system without tiering support. Similarly, we want to be able to exploit the full performance of byte-addressable NVM. Ideally, we would also exploit the full performance of block-addressed devices, such as SSDs.

In the following, we describe how Hyrise solves these challenges and fulfills the goals of transparency, extensibility, and performance. Our approach integrates existing techniques, including Polymorphic Memory Resources, memkind, and umap, into a novel, comprehensive multi-tier memory management framework. It enables automatic tiering for data structures that have not been developed with tiering in mind and that have so far evaded any evaluation from that perspective. As such, our framework constitutes both a contribution to the goal of this thesis as well as to future research.

The remainder of this section is organized by the following guiding questions:

1. How can NVM and SSD tiers be accessed by data structures that were not developed with different tiers in mind? (Section 5.1)
2. Which allocation primitives are needed to perform fine-grained allocations on these tiers? (Section 5.2)
3. With each tier using a different allocation primitive, how can data structures allocate space in a unified and type-safe manner? (Section 5.3)
4. How can data structures be migrated between tiers without stopping the system and even if the data structure was never designed to be migrated? (Section 5.4)
5. What are the performance implications of the abstractions introduced to solve the aforementioned challenges? (Section 5.5)
6. Can these abstractions be used for other challenges outside of automatic tiering? (Section 5.6)

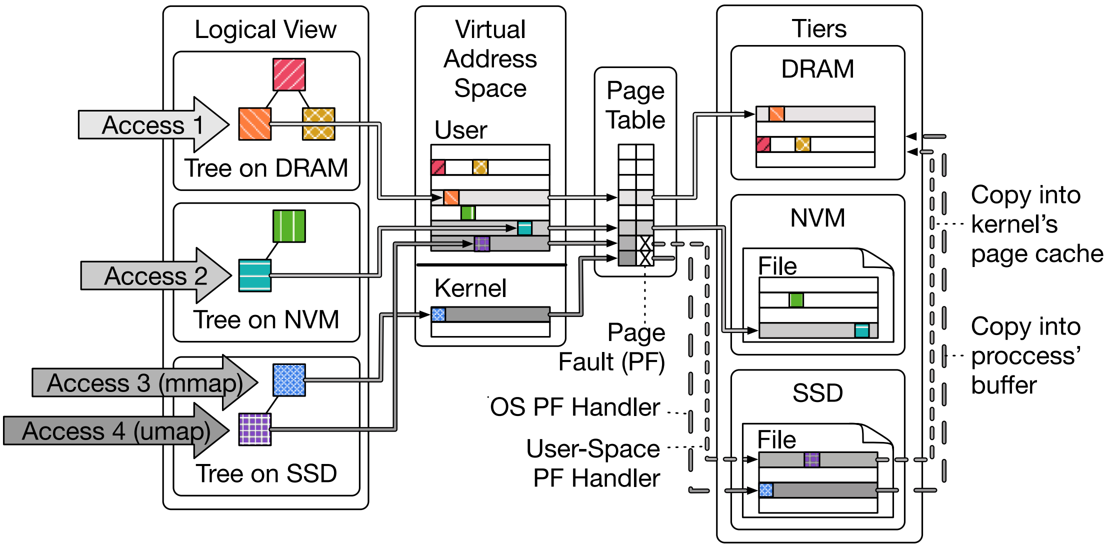


Figure 5.1: Access methods for different memory and storage tiers. The trees, shown on the left, are logical representations of data structures of the physical data stored on DRAM, NVM, or SSD; shown on the right. For DRAM and NVM, accesses only require a translation from a virtual to a physical address. For SSDs, such a physical address does not exist, and a page fault handler has to copy the data block into a DRAM buffer. Optimizations, such as the translation lookaside buffer (TLB) are not displayed.

5.1 Tier-Agnostic Data Access

How can NVM and SSD tiers be accessed by data structures that were not developed with different tiers in mind?

In an in-memory (i.e., DRAM) DBMS, data structures are stored in the virtual address space of a process. Pointers to other virtual addresses are used to connect them and build more complex data structures. When a virtual address is accessed by the CPU, it is transparently translated into a physical address by looking up the corresponding mapping in the page table. The CPU can then access the data at the physical address.

For NVM, which, just like DRAM, is byte-addressable and attached to the CPU's memory controller, data structures are equally accessible by the CPU. This is true even though NVM is usually managed by a filesystem overlay (*fsdax*, cf. Section 2.2.3). Figure 5.1 shows these accesses for DRAM and NVM as *Access 1* and *Access 2*. For the purpose of this thesis, we exclude the steps that NVM requires to perform persistent writes (cf. Section 1.4).

On the other hand, block devices, such as SSDs, cannot be immediately addressed by the CPU. Traditionally, functions like `read` are used to *explicitly* copy (parts of) a file into DRAM, where the data can then be used by the CPU. The explicit call to `read` is part of the data structure's implementation. For example, a tree can hold a file descriptor and use it to `read` the children of a node. This, however, is incompatible with our goal of requiring no changes to the implementation of the data structures.

Instead, we need a mechanism that *implicitly* brings data from block devices into DRAM when accessed. This mechanism exists with the concept of *memory-mapped files*. Such a mapping is established using either the mmap system call or user-defined page fault handling. We first describe mmap with its shortcomings before we come to umap, which we ultimately use for Hyrise. mmap updates the process' page table and assigns a virtual address range to the mapped file. If such a virtual address is accessed, a *page fault* is triggered. The kernel's *page fault handler* copies the data from the block device into the *page cache* and updates the page table. This process is shown as *Access 3* in Figure 5.1. The application can then continue and access the page. The mmap mechanism makes it possible to run address-based functions on data stored in a file without having to adapt the implementation of the function. However, mmap on block devices comes with known drawbacks. Most prominently, Linus Torvalds commented on its performance on the Linux kernel mailing list (LKML) [236]:

> *"It's not the IO on the pages themselves, it's actually the act of populating the page tables that is quite costly. And doing that in the background is basically impossible. [...] And on top of that you still have the actual CPU TLB miss costs etc. Which can often be avoided if you just re-read into the same area instead of being excessively clever with memory management just to avoid a copy. memcpy() (ie 'read()' in this case) is always going to be faster in many cases, just because it avoids all the extra complexity. While mmap() is going to be faster in other cases."*

In our case, in which explicit calls to `read` are not an option, we have to accept a certain overhead for accesses to block-level devices. We quantify that overhead when we evaluate our memory management concept in Section 5.5.

Another disadvantage of mmap is that it holds data in DRAM until the operating system's memory pressure is sufficiently high. Only then do pages get dropped from the OS page cache. While this generally increases the number of cache hits, it also makes the access times to data stored on block-level devices hard to predict. Furthermore, it means that all processes on a machine share the same page cache. In a multi-tenant scenario, this means that the reads of one tenant may evict data from a second tenant and that no guarantees on the individually available cache size can be given. For the sake of performance and robustness, we prefer a method that allows the size of the cache to be restricted and that reliably causes physical reads from these devices instead of cached reads.

Such a method is provided by the umap [191] library. Similarly to mmap, it manipulates the page tables so that virtual addresses are provided for the mapped file. However, accesses to these pages are not handled by the kernel's page fault handler, but by a handler that is implemented in the user space (`userfaultfd`). In Figure 5.1, this is shown as *Access 4*. Pages that are read by this handler are not stored in the OS page cache, but in a umap- and process-internal cache. This mechanism allows umap to provide more fine-grained control over the paging behavior than mmap offers. Most importantly for our use case, it allows us to limit the size of the page cache to a fixed size instead of relying on the variable OS page cache size. This is controlled by the `UMAP_BUFSIZE` environment variable, which defaults to using 90% of the free DRAM. For our purposes, it needs to be set to a lower value. Otherwise, a large buffer cache may prevent reads from actually being served from disk. By choosing a low cache size of 16 MB, we ensure that umap does not act as an additional cache but only as a means to access data from lower tiers.

To summarize, we use direct mapping for byte-addressable memory (DRAM/NVM) and umap for block-level devices (SSDs). These different approaches are necessary because we need to translate between the byte-based accesses of data structures whose code we cannot modify, and the block-based operations of, e.g., SSDs. The presented combination of mmap and umap gives us a unified method to access data on different tiers via virtual addresses without requiring any modifications to the source code of the stored data structures.

5.2 Space Allocation on Different Tiers

Which primitives are needed to perform fine-grained allocations on these tiers?

The next task is to subdivide the fixed-size pages (or blocks) into consumable units. The process of allocating memory on DRAM is usually well-understood: in C, a call to `malloc(n)` returns a virtual address at which n bytes can be used. Because the available physical memory is shared between processes, new memory has to originally be allocated from the kernel. For a variety of reasons, the kernel does not support byte-level allocations but operates on the level of pages. It is thus the job of a dynamic allocator to subdivide the pages. This allocation method needs to be implemented in a way that reduces internal and external fragmentation [229].

For NVM, the allocation process is slightly different. Other than it is the case for DRAM, the kernel does not "own" the NVM physical address space. Instead, NVM is often exposed as a file system (*fsdax*, cf. Section 2.2.3). Memory on NVM can be "allocated" by creating a new file. As with files on traditional storage devices, these files cannot be allocated with byte-level granularity. This again creates the need for a dynamic allocator that subdivides NVM files into smaller allocation sizes. Depending on the use case, different allocators exist: when NVM is used as a persistent data store, the allocator has to be designed accordingly, ensuring ACID[1] criteria even across system failures [218]. For this use case, *PMDK* [214] is commonly used.

In our use case, NVM is used for its higher capacity, not for its persistency. Because files are not re-used when the DBMS process restarts, the challenges of making allocations persistent and non-leaking are not of concern. Instead, the data can be discarded by deleting the file. Thus, no additional requirements exist for the allocator beyond those known from DRAM. This relaxation makes it possible to re-use existing allocators such as jemalloc [73] instead of more expensive persistency-aware allocators such as those included in PMDK. This allocator has already been shown to be highly efficient

[1] Atomicity, Consistency, Isolation, Durability

for in-memory databases [70]. For jemalloc to be used on different tiers, it needs to be modified so that its backing memory is not retrieved from by extending the DRAM map of the process, but by increasing the size of the pool files on the different tiers.

Jemalloc allows developers to define additional memory arenas that draw their backing memory (also called *extents*) not from the standard system calls (`brk` and `mmap`), but by calling a custom extent allocation hook. This hook can then allocate memory on NVM by creating (or enlarging) a file on the NVM file system. The files from which memory is allocated are also called *pools*. Such hooks have been implemented by the memkind library [215], which provides a unified API for allocations on DRAM and NVM. This is shown in Listing 5.1.

```
// Traditional allocation
auto a = malloc(10);

// Allocation on DRAM with memkind:
auto b = memkind_malloc(MEMKIND_REGULAR, 10);

// Preparation for NVM, done only once:
auto nvm_kind = memkind_t{};
memkind_create_pmem("/mnt/nvm/", 0, &nvm_kind);

// Allocation on NVM with memkind:
auto c = memkind_malloc(nvm_kind, 10);
```

Listing 5.1: Allocation using malloc and memkind_malloc primitives.

For SSD and other block devices, we would like to use the same concept. Because memkind does not generally require the pool file to be created on NVM, it could also support other devices. However, as it internally uses mmap, it suffers from the mmap-related problems that were described in the previous section. We thus experimented with a custom jemalloc-based allocator for block devices. Here, we found that jemalloc does not support a separation of meta data and allocated data. Storing the meta data on block devices, such as SSDs, proved to come with an unacceptable overhead. For as long as a separation of the meta data extents [158] is not implemented in jemalloc, we thus use a different approach: we create one file per tiered data structure and subdivide it using a monotonic allocation strategy. Once the data structure is deleted, the file is deleted, too. This is sufficient for our use case in which only immutable data structures are migrated between tiers and no regular re-allocations need to be made.

To summarize, the NVM and SSD tiers require custom dynamic allocators. With memkind, we use an existing library that supports volatile allocations on NVM. For SSDs, we use a custom allocator that is optimized for immutable data structures.

5.3 Use of Allocated Space by Data Structures

With each tier using a different allocation primitive, how can data structures allocate space in a unified and type-safe manner?

Having allocated space from a specific tier, we need to make this space accessible to the data structures used by the DBMS. This is challenging for a number of reasons:

- As described in the previous two subsections, accesses and allocations on DRAM, NVM, and SSD build on different mechanisms. When a data structure requires more space, it needs to call the correct mechanism. All mechanisms must return a virtual address that is properly mapped and that allows transparent access to the allocated space.
- This also applies to nested data structures. An inverted index that holds heap-stored data types (e.g., variable-length strings) should use the pool not only for its own allocations but also for those of the contained data type.
- When the allocated space is no longer needed, it must be returned to the same pool that it was allocated from. For this, the `deallocate` method of the corresponding allocator has to be called.
- The solution to the previous challenges should be applicable to arbitrary data structures without modifications. This is especially relevant for research, as it allows researchers to evaluate third-party data structures without having to change their internals.

5.3.1 Running Example

Before we describe our solution, we introduce a running example. It represents an inverted index on a single chunk. As shown in Figure 5.2, the chunk holds two columns, namely the *double* `a` and the *integer* value `b`. It also contains three rows. The inverted index uses an `std::map`, which is usually implemented as a red-black-tree. We chose this example instead of, e.g., an uncompressed data vector, because it shows that our memory management abstraction can deal even with complicated data structures that include internal pointers. Outside of this example, a more efficient data structure would be chosen as an index data structure. In the following, we describe how this index can be migrated to a different tier and how the aforementioned challenges can be addressed. We limit the discussion to the DRAM and NVM tiers with no loss of generality.

	a	b
0	5	4.2
1	7	1.4
2	2	3.5

```
const auto& segment = chunk.get_column(1);
using InvertedIndex = std::map<double, uint>;
auto index = InvertedIndex{};
for (auto i = 0; i < segment.size(); ++i)
  index->emplace(segment[i], i);
// index == {{1.4, 1}, {3.5, 2}, {4.2, 0}}
```

Figure 5.2: Sample chunk and simplified creation of an inverted index.

5.3.2 Stateful Allocators

> **Excursus** on the term *allocator*: Knuth describes dynamic storage allocation as "*algorithms for reserving and freeing variable-size blocks of space from a larger storage area*" [126]. In contrast, C++ defines allocators as "*class-type objects that encapsulate the information about an allocation model. This information includes [...] the memory allocation and deallocation primitives for it*" [37, p.475]. Here, the traditional task of subdividing a memory or storage pool is called an *allocation primitive.* Luckily, the context in which the term *allocator* is used, together with the awareness of the differing definitions, is usually sufficient to avoid ambiguities.

By default, the red-black-tree created in the running example allocates its space from DRAM. This is governed by one of `std::map`'s implicit template parameters:
`std::map<..., Allocator=std::allocator<std::pair<const Key, T>>>`
Internally, this allocator calls malloc, which is the fundamental way of allocating space in C++. Instead of having the standard allocator allocate space on DRAM, we now want to use NVM. In the previous section, we have shown how memkind can be used to manage and subdivide an NVM pool. It is thus our goal to change the tree from using malloc to `memkind_malloc`. The straight-forward way of doing this is to implement a custom C++ allocator and use it instead of the default allocator. During compile time, the maps's allocation will then automatically be linked to `memkind_malloc`. This is shown in Listing 5.2.

```
template <class T>
class nvm_allocator {
  memkind_t nvm_kind;
  nvm_allocator() {
    memkind_create_pmem("/mnt/pmem/...", 0, &nvm_kind);
  }
  T* allocate(std::size_t n) {
    return (T*) memkind_malloc(nvm_kind, n * sizeof(T));
  }
  // Deallocation and destructor omitted
}

using InvertedIndexDRAM = std::map<double, int>;
using InvertedIndexNVM = std::map<double, int,
  nvm_allocator<std::pair<const double, int>>>;
```

Listing 5.2: Allocating space on NVM using memkind_malloc.

While this `nvm_allocator` allows us to store data on NVM, it has one drawback: the DRAM- and the NVM-based indexes now have two different types. In a strictly typed language such as C++, this means that they are incompatible. They have no common super class and none can be added. As a result, a chunk can either hold a DRAM- or an NVM-based index and this decision has to be made at compile time. To rectify this, we store the information whether DRAM or NVM should be used in the allocator *object* instead of the allocator *type*. This makes the allocator stateful, as shown in Listing 5.3[2].

```
template <class T>
class multitier_allocator {
  struct memkind *kind;

  multitier_allocator(struct memkind *kind) : kind(kind) {}

  T* allocate(std::size_t n) {
    return (T*) memkind_malloc(kind, n * sizeof(T));
  }
}

using InvertedIndex =
  std::map</*[...]*/, multitier_allocator</*[...]*/>>;

auto dram_index = InvertedIndex{dram_kind};
auto nvm_index = InvertedIndex{nvm_kind};
```

Listing 5.3: Allocating space on a memkind that is specified at runtime.

With this modification, the index now draws its space from a memkind (i.e., DRAM or NVM) that is specified at runtime. The allocator's `deallocate` method can be implemented in the same way. Because the compile-time types of the two inverted indexes are identical, they can be used interchangeably. This allows us to store both DRAM and NVM indexes, e.g., in the same vector of inverted indexes. Using stateful allocators fulfils our goals of (1) ensuring that (re-)allocations and deallocations use the correct memory pool and (2) not requiring any modifications to the implementation of the data structures.

Still, this solution is not yet ideal. First, it introduces tight coupling of all tierable data structures to the memkind library. The allocator and with it, the memkind struct, becomes part of every tierable data structure's type. This counteracts our efforts to make automatic tiering transparent not only to the user, but also to other developers. Second, this allocator is only used for the top-level data structure (here: the index) and is not passed down. It is not used for any nested, heap-stored data structures, such as variable-length strings[3]. Finally, this allocator only supports memkind. For SSD and other tiers, we would have to extend the allocator to distinguish between memkind and non-memkind allocations.

[2] Using `std::variant` would also make it possible to mix the two types. As this would make the code more complicated and comes with other drawbacks, we do not further pursue this option.

[3] We have experimented with `scoped_allocator_adaptors`, which could technically address the second problem and have since then been proposed for similar use cases [215]. Compared to polymorphic allocators, which are described next, scoped allocator adaptors have no benefits but require explicitly stating all nested types and hence further increase the coupling.

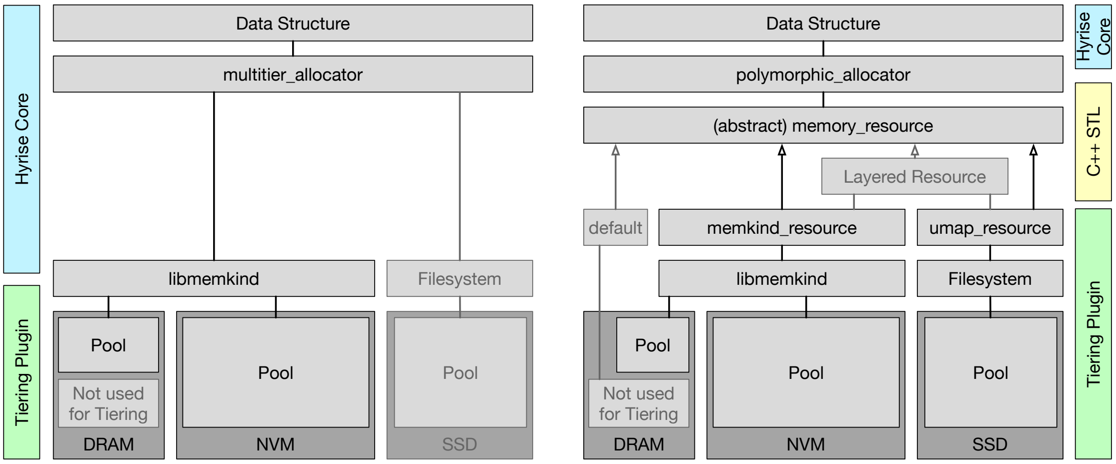


Figure 5.3: Comparison of an approach that uses only stateful allocators (left) and one that includes polymorphic memory resources (right). Both allow the data structure on top to allocate space from one of the DRAM, NVM, or SSD pools. The PMR abstraction helps to decouple the details of the tiering implementation as only the abstract `memory_resource` interface needs to be exposed to the Hyrise Core.

5.3.3 Polymorphic Memory Resources

To quote David J. Wheeler, "*we can solve any problem by introducing an extra level of indirection*" [140]. In this case, this indirection is called *Polymorphic Memory Resources*, or short PMR. With PMR, the allocation behavior is changed from being a compile-time property to being an execution-time property that is resolved using dynamic polymorphism (i.e., inheritance). It introduces a `memory_resource` interface. Implementations of this interface provide methods for allocating and deallocating memory. Data structures that use these memory resources call `(de)allocate` via virtual method calls. Originally, PMR was intended to support "*thoughtfully-chosen local ('arena') memory allocators*", which can "*yield significant (sometimes order-of-magnitude) performance improvements over [...] general-purpose global allocators*" [95]. In other words, it was designed to support different allocation strategies on DRAM. We present a novel approach in which PMR is used to add tiering capabilities to arbitrary data structures.

Figure 5.3 gives an overview of the architecture of the previously described stateful allocators and the PMR improvement presented here. For now, we focus on the components printed with black font. We discuss default and layered resources later. Starting from the top, the data structure uses the `polymorphic_allocator`, a class included in the C++ standard. This allocator is constructed using either the `memkind_resource` or the `umap_resource`. The former allocates memory on DRAM or NVM using memkind, the latter allocates storage space on SSD using our custom monotonic allocator, both of which were described in Section 5.2. An example on how PMR can be used for the running example is given in Listing 5.4.

```
using InvertedIndex =
  std::map</*[...]*/, polymorphic_allocator</*[...]*/>>;

// Allocation on DRAM
auto dram_allocator = polymorphic_allocator(&dram_resource);
auto dram_index = InvertedIndex{dram_allocator};

// Allocation on NVM (using implicit construction)
auto nvm_index = InvertedIndex{&nvm_resource};

// Allocation on DRAM (fallback without explicit resource)
auto dram_index2 = InvertedIndex{};
```

Listing 5.4: Implementation of an inverted index using PMR.

On first sight, this looks similar to the code used for the stateful allocators in Listing 5.2, maybe even more complicated. However, Figure 5.3 reveals some advantages:

- The colored parts on the side of the figure show which components have to be implemented in the Hyrise core, which ones can be encapsulated in the tiering component, and which ones are handled by C++'s standard library (STL). Thanks to PMR, fewer components have to be implemented in the core. Namely, the only necessary modification in the Hyrise core is to add the `polymorphic_allocator` to data structures that should be subject to tiering. Because the polymorphic allocator is part of the C++ library, no additional dependencies are introduced into the Hyrise core.
- PMR has a default memory resource which falls back to the default malloc behavior if no memory resource is specified[4]. As such, both the memkind and umap libraries and the corresponding memory resources can be fully encapsulated within the tiering plugin. Not even their headers have to be included in the Hyrise Core. Only the tiering plugin needs to know about their existence and they can be dynamically loaded together as part of the plugin. If no memory resource is provided, Hyrise falls back to using regular malloc (more specifically, jemalloc).
- Memory resources can be layered, "*i.e., one [resource] provides some memory-management functionality and goes to another backing [resource] when additional memory is needed*" [95]. An example of such a layered resource is the *monotonic buffer resource*, which linearly allocates memory in a contiguous buffer and defers the deallocations to the deletion of the entire buffer. This can be used to remove deallocation costs for very short-lived data structures. By introducing PMR into Hyrise, we can profit from these layers in operators that create numerous temporary objects, such as the hash-based aggregate and join operators.

[4] Reproducibility Notice: The STL's `polymorphic_allocator` calls `get_default_resource`, which is a synchronized method [179, page 219]. In our experiments, we have found this to be a significant performance bottleneck. As such, we use a modified polymorphic allocator, which uses a static `new_delete_resource` if no explicit resource is passed in.

Two more advantages of PMR cannot be visualized in the architecture diagram but are nevertheless relevant:

- Being a C++ library concept, polymorphic memory resources have carefully designed and tested semantics.
- PMR solves the challenge of propagating the memory resource to nested data structures. If a PMR-enabled container (e.g., a variable-length string) is stored in another PMR container, they automatically share the same memory resource.

To summarize, we use polymorphic memory resources as a mean to enable tiering for arbitrary C++ data structures. These data structures do not have to be designed with PMR in mind, but have to conform to C++'s allocation interface. Using PMR instead of regular stateful allocators allows us to decouple the use of these data structures from the different tiering layers. Furthermore, it enables "*rapid prototyping and enhanced predictability*" [95], which has already been named as a "*collateral benefit*" for PMR without tiering. To the best of our knowledge, we are the first to use PMR in the context of data tiering.

5.4 Migration of Data Structures Between Tiers

How can data structures be migrated between tiers without stopping the system and even if the data structure was never designed to be migrated?

By now, all building blocks for storing data structures on different tiers have been presented. To finally put them to use, we need a mechanism to migrate data structures from DRAM to lower tiers and back. Conceptually, there are two approaches: for the first approach, which we call *page migration*, the space on which the data structures are stored is moved from one tier to another. Afterwards, the page table is updated accordingly. With the second approach, which we call *object migration*, the data structures themselves are moved (more exactly: copied) from one space to another. We do this by calling the C++ copy constructor as shown in Listing 5.5.

```
auto nvm_segment = DictionarySegment<T>(dram_segment,
                                        &nvm_resource);

// Post-Conditions: dram_segment == nvm_segment,
//                  &dram_segment != &nvm_segment
```

Listing 5.5: Migration of a DRAM-resident segment to NVM.

Both approaches have their advantages and disadvantages as shown in Table 5.1. After weighing these against each other, we consider the object migration approach to be less intrusive and more maintainable than the page migration approach. The maintainability aspect played a significant role in our decision for the object migration approach. Past experience has shown that overly smart low-level hacks have a tendency to not be maintained once the original developer has passed on the ownership of the component.

Page Migration	Advantages	• Migrating the pages that a data structure is stored on is data structure-agnostic. Because the virtual addresses remain the same, the migration is invisible to both the data structure and its user. • As the migration is completed by atomically updating the page table, read-only data structures (such as immutable segments) can be migrated without any type of synchronization. For mutable data structures, this synchronization can be implemented by write-protecting the page(s) for the duration of the migration. Similar approaches have been successfully used, e.g., for modifying concurrent data structures [58]. • Some systems provide co-processors with a limited instruction set that can be used to move data without blocking a CPU core. For example, the HPE Integrity MC990 X server (formerly SGI UV 300), feature a co-processor called HARP. We have previously shown the value of its `bcopy` primitive for offloading memory operations [64, 65]. This could similarly be used to asynchronously move data between tiers.
	Disadvantages	• To migrate a data structure, the pages that it occupies need to be known. While this can be achieved without modifications to the data structure implementation [62], this tracking adds additional costs. • Pages may contain multiple data structures. While this is only an edge case for data structures that occupy contiguous memory, it is a more common case for data structures with many internal allocations (such as trees). To cleanly migrate individual data structures without affecting others, a 1:n mapping from data structures to pages is needed. This increases the cost of allocations and introduces additional external fragmentation.
Object Migration	Advantages	• Even nested or circular data structures are automatically handled as long as they fulfill the C++ requirements for allocator-aware and copy-constructable classes. This is commonly the case. • Copy construction is more intuitive and, thus, the implementation is easier to understand than manipulations of the page table would be. Even though this component is transparent to most of the code, its simplicity plays a role for the maintainability of the memory management framework itself.
	Disadvantages	• Data structures without a copy constructor cannot be migrated using this approach. Mostly, this applies to data structures that are implemented in plain C. • Invoking the copy constructor is more expensive than performing a binary copy of the underlying data. • Thread-safety in the presence of concurrent writes cannot be guaranteed as easily as with the page migration approach. While parallel reads on the *old* data structure are innocuous, writes can either be lost or lead to a corrupt *new* copy. The absence of such writes thus has to be guaranteed by other mechanisms.

Table 5.1: Advantages and disadvantages of the page and object migration approaches.

When migrating segments, their immutability is guaranteed by the table model and the insert-only properties of Hyrise. As such, two copies can safely co-exist and we do not require additional measures to guarantee the thread-safety of the migration. Without pausing the execution of queries, we (1) copy the segment from the old tier to the new tier, (2) atomically update the chunk's segment pointer from the old to the new copy, and (3) wait for all pending operations on the old segment to complete, at which point the old segment can be destructed and deallocated.

This approach cannot be used for data that is being modified during the migration. In the case of tables, this means that neither mutable segments nor the MVCC information can be migrated. Because there is only a single mutable segment per column and the size of the MVCC information that remains on DRAM is only 12 Byte per row, we consider this limitation to be acceptable.

5.5 Evaluation

What are the performance implications of the abstractions introduced to solve the aforementioned challenges?

So far, we have discussed all components that are needed to store data structures on different tiers and to migrate them between these tiers. We now evaluate these building blocks bottom-up. This includes the access and allocation methods, the PMR-based data structures, and the migration approach.

For the performance of Intel's memory modules alone, and excluding the use of NVM in individual data structures or even entire applications, we are aware of six publications that aim at providing a better understanding of the modules' low-level performance characteristics [92, 103, 112, 190, 242, 260]. It is not the goal of this thesis to supplement these 114 pages of low-level performance evaluations. Instead, we focus on those performance aspects that we found helpful for the development of our automatic tiering approach. At the same time, we reiterate that our architecture is not designed for a specific NVM product and can be used equally for other tiers that exhibit different performance characteristics.

5.5.1 Access Methods

In Section 5.1, we described an abstraction layer that hides the details of different tiers behind the page table mechanism. For DRAM and NVM, this uses 1:1 mappings between logical and physical pages; for SSDs, it uses the umap library. We have already discussed that the page manipulation mechanisms used by mmap and umap come with a certain overhead. This overhead is quantified by comparing the two access methods with traditional `reads`. For the latter, which is not used by Hyrise, we use the *Flexible I/O Tester* fio, "*a tool that [is] able to simulate a given I/O workload without resorting to writing a tailored test case again and again*" [18]. This tool has previously been used in other publications to establish baselines for the performance of different devices [112, 234, 247, 258, 261].

	System A (DRAM and SSDs)	System B (NVM)
CPUs	4x Xeon Platinum 8180 (2.5 GHz base, 3.8 GHz turbo, 28 physical cores)	2x Xeon Gold 5220S (2.7 GHz base, 3.9 GHz turbo, 18 physical cores)
DRAM	2x3x Samsung DDR4-2666 64 GB (M386A8K40BM2-CTD)	2x6x Samsung DDR4-2933 16 GB (M393A2K40CB2-CVF)
OS	Ubuntu 20.10 (Kernel 5.9.10)	Ubuntu 18.04.4 LTS (Kernel 4.15.0)

Table 5.2: Configuration of the benchmark systems.

```
--bs=[variable] --numjobs=[variable] --ioengine=[variable]
--group_reporting --size=2147483648 --offset_increment=134217728
--runtime=10 --gtod_reduce=1 --rw=randread --norandommap=1 --thread=1
--scramble_buffers=0 --time_based
```

Listing 5.6: Parameters used for executing the fio benchmarks.

Benchmark Configuration. The benchmarks are executed with the following devices serving as different memory and storage tiers:

- DRAM: Samsung DDR4-2666 (M386A8K40BM2-CTD), 3 DIMMs per CPU, accessed via `tmpfs`, node locality guaranteed with `mpol=bind:0`
- NVM: Intel Optane DCPMM (NMA1XXD128GPS), 6 DIMMs per CPU, node locality guaranteed by selecting the corresponding `/dev/pmem` device
- SSD 1: Intel 3D XPoint DC P4800X (SSDPED1K375GA), an SSD that uses the same physical storage mechanism as the NVM DIMM, but is attached via PCI/NVMe
- SSD 2: Samsung 850 Pro, a "traditional" NAND-based and SATA-attached SSD

Unfortunately, NVM is only available to us as a shared resource in HPI's Data Engineering Lab. These NVM servers do not have internal SSDs and use an old kernel version that is incompatible with umap. We were unable to overcome these limitations without causing significant disruptions to the service's availability to other users. As such, the NVM and SSD benchmarks have been executed on two different systems: the DRAM and SSD benchmarks use the system that was already used for the benchmarks in Section 4.6. We use two different SSDs, one attached via NVMe and another one via SATA to broaden the scope of the evaluation. For NVM, we use the shared server. The details of both servers are given in Table 5.2. Because we are only benchmarking the raw bandwidth of the systems without producing any compute-intensive load, we consider a comparison across the two systems to be relatively safe.

We use fio in version 3.25-15 and umap in version 2.1.0. All devices were formatted using xfs, NVM was accordingly configured as fsdax. For fio, we use the parameters given in Listing 5.6 and limited the benchmarks to a single NUMA node.

When using fio, the number of threads is controlled via the `numjobs` parameter, not the `thread` parameter. For all but DRAM (for which the parameter is not supported and would not make a difference), we also use the `-direct=1` parameter, which leads to the files being opened with `O_DIRECT`, thus minimizing the system-internal buffering.

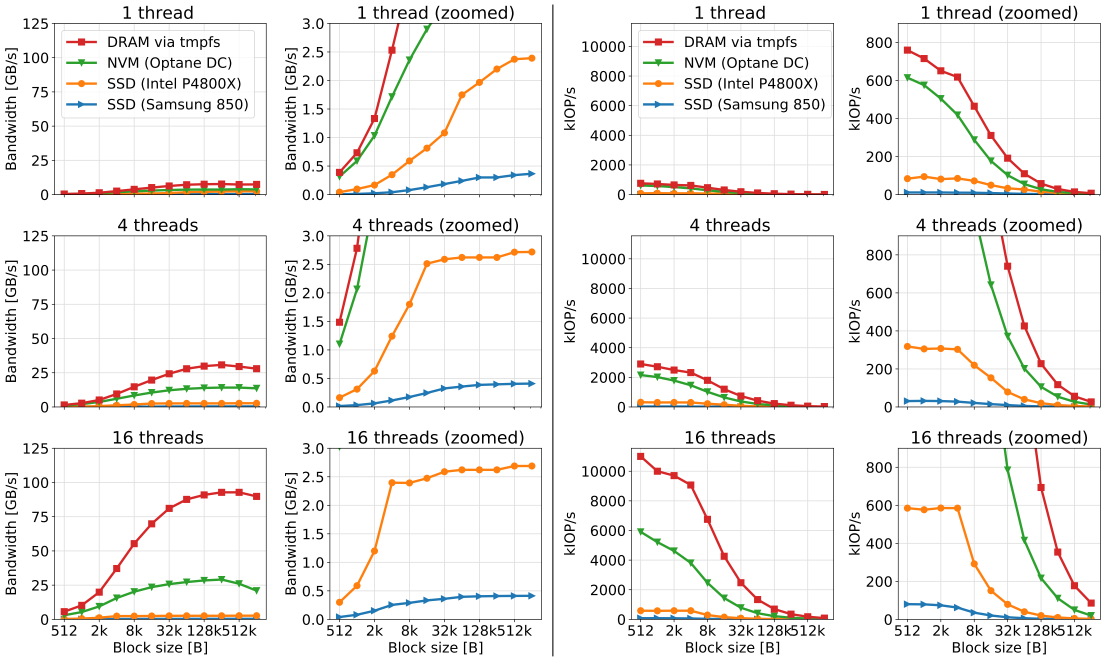


Figure 5.4: Bandwidth (left) and number of operations per second (right) for a random read workload on different devices. These numbers serve as the baseline with which we can compare the performance of our access method in a following experiment.

Findings. Figure 5.4 shows the findings of a benchmark in which random blocks of varying size (x-axis) are read from different devices (different lines) by a varying number of threads (vertical graphs). The left two graphs show the achieved bandwidth on the y-axis; the right two graphs show the number of operations per second. Each data point was obtained by running fio three times for ten seconds each and taking the mean of these runs. The lower end of the block size dimension (x-axis) is 512 Byte, as this is the minimum I/O size of the SSDs and no `O_DIRECT` access is possible below this size. We make the following observations:

1. The memory and storage tiers can be clearly identified. As expected, DRAM is the fastest device regardless of the configuration, followed by NVM, and finally the PCIe and SATA SSDs in that order. The bandwidth of the memory and the storage tiers differs by more than an order of magnitude.
2. For the SSDs, the maximum bandwidth matches the numbers found in the hardware specifications. For DRAM and NVM, fio does not yet reach the maximum bandwidth as measured by other tools like `mlc` and discussed in previous publications [241, 261]. This is because in this experiment, we use the `sync` engine. In a follow-up experiment, we use the mmap engine.
3. In all cases, the maximum bandwidth can only be achieved when multiple threads read data concurrently. This is especially the case for DRAM and NVM, which

cannot be saturated by just four threads.

4. Sequential reads are strongly preferred by all devices. In almost all cases, bigger block sizes (i.e., larger values on the x-axis) lead to a higher bandwidth.

Before we interpret these findings and their relevance for our work, we present a second benchmark. This benchmark uses memory mapping as discussed in Section 5.1. As a microbenchmark, it resembles the access methods that are used by Hyrise.

Benchmark Setup (cont.). For both mmap and umap, the cache size is limited to 16 MB. In the cast of mmap, which does not natively support this limitation, the limit was set via `cgroup`. For umap, we test two configurations: one, in which the cache entry size is 16 KB and one in which it is 64 KB. The cache entry size determines how much data is read on a single (user space) page fault. It is independent of the block size, which describes how much data is actually consumed.

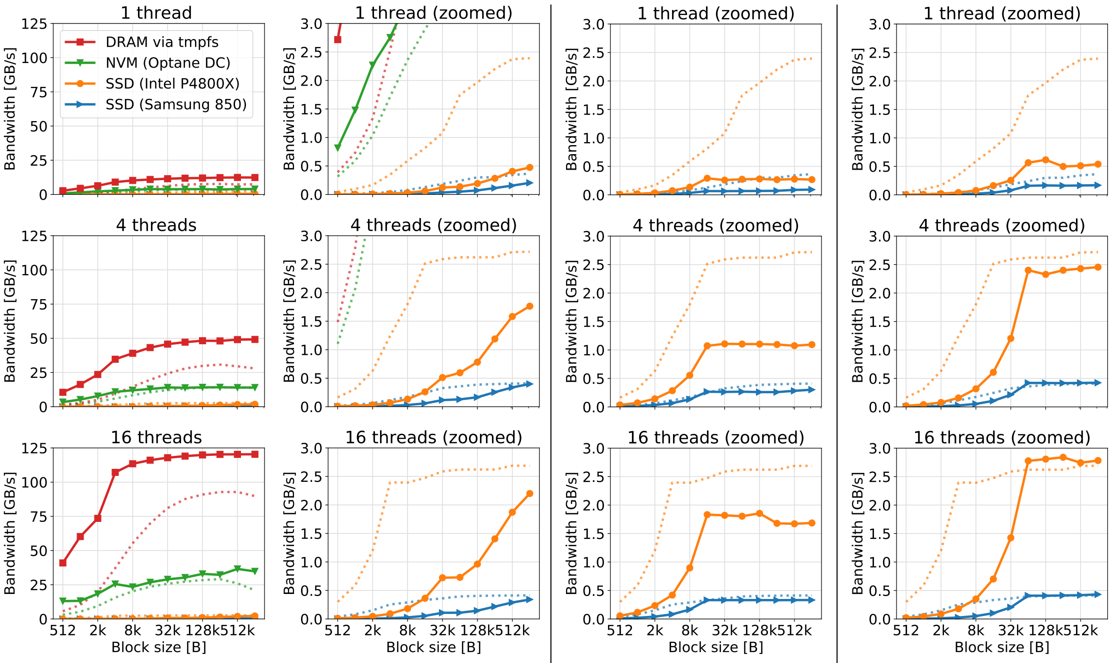
Findings (cont.). Looking at Figure 5.5, we make the following observations:

5. Starting with the mmap measurements (a), we can see a higher bandwidth for both DRAM and NVM than in the fio benchmark with the `sync` engine. Both memory types now reach their maximum bandwidth as reported by `mlc` and previous publications [241, 261].
6. The SSDs, on the other hand, exhibit a consistently lower bandwidth than measured with fio/`sync`. Especially for small block sizes, it is notable that a higher number of threads no longer helps in saturating the potential bandwidth.
7. Now looking at the umap experiments (b and c), the bandwidth is, for the most part, still lower than that measured for the experiment in Figure 5.4. However, compared to mmap, umap shows both a higher average bandwidth and a more consistent distribution across different block sizes.
8. Unsurprisingly, umap performs best when the entire cache entry (16 KB in experiment b and 64 KB in experiment c) are consumed.

We acknowledge that there are further observations that can be made and additional benchmarks that could be conducted. In the case of NVM, entire publications have been written to partially understand the performance characteristics of this new type of memory. We have discussed some of these in Chapter 3. For this thesis, it was necessary to limit the scope of our investigation to those experiments that help us to understand the performance implications of our memory access method. From these experiments, we draw the following conclusions:

Interpretation. The two experiments allow us to draw a number of conclusions for the implementation of our memory management and to identify requirements for the components that build upon it: first of all, DRAM and NVM are faster than SSDs by at least an order of magnitude (Finding 1). This supports our approach of using SSD only as a fallback tier. For DRAM and NVM, mmap is clearly the more efficient access method, as shown by Findings 2 and 5. This is different in the case of the SSDs, for which `read` was more efficient (Finding 6). For our goal of supporting transparent tiering without code modifications, however, using `read` is not an option.

The umap library provides a decent improvement over mmap (Finding 7). This is both because of more efficient page fault handling and because of a customizable cache entry size. Still, umap can only play its strengths when large blocks of data are read (Findings



(a) mmap (4 KB)

(b) umap (16 KB)

(c) umap (64 KB)

Figure 5.5: Bandwidth when accessing random blocks of a given size (x-axis) on different devices using the mmap and umap methods. In all cases, the cache size is 16 MB. In the case of umap, a varying cache entry size of 16 and 64 KB is used. Because umap does not support DRAM/NVM, only the SSDs are shown in the corresponding plots. For comparison, the previous results from the fio-based benchmark Figure 5.4 have been added as dotted lines.

4 and 8). For our tiering concept, this is the most important realization, as it urges us to preferably move those types of data to a lower tier that are predominantly read sequentially. This imposes requirements on both the access tracking method (which needs to identify sequential accesses) and on the decision making algorithm (which needs to take these accesses into account). We discuss how our memory management incorporates this result in Chapters 6 and 7.

Finally, the benchmarks show that to fully utilize the available bandwidth, multiple threads need to access the data (Finding 3). As Hyrise parallelizes both within operators and across queries, this is taken into account by default.

5.5.2 Allocation Primitives

Next, we evaluate the performance of the allocation mechanism. In Section 5.2, we described our use of memkind to allocate space on NVM. As part of the entire memory management component, we need to verify that this abstraction does not introduce any undue overhead. The memkind paper [215] provides some initial benchmarks. Our past

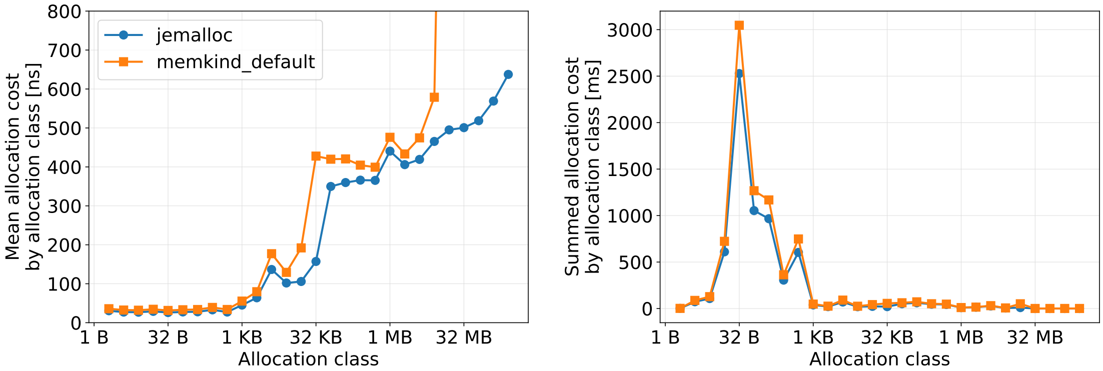


Figure 5.6: Costs when replaying the allocations performed during the execution of the TPC-H benchmark, comparing the default jemalloc allocator with memkind. Allocations are rounded to the next power of two and placed into the corresponding allocation class. The graph on the left shows the average cost per allocation, the one on the right shows the summed cost.

experience, however, has shown that results from general allocator benchmarks do not automatically translate to the allocation patterns used by databases. As such, a custom IMDB-specific benchmark is used to evaluate the overhead of memkind.

Benchmark Configuration. For this, we instrument Hyrise to log its memory allocations into a binary file. We log the size of the allocations, the resulting pointer, and the corresponding deallocations. With this information, we can simulate the `malloc` and `free` calls in chronological order. The trace is generated by running each TPC-H query for ten seconds. It results in a log that contains just over two billion allocations. In a second step, a dedicated microbenchmark reads this file and replays the log using a user-defined allocator. This benchmark design isolates the performance of the allocator from that of Hyrise. In Section 5.5.4, we also show end-to-end benchmarks.

As a baseline, we use jemalloc, which is the default allocator in Hyrise and is used whenever malloc is called. It has also been found to perform best in other database systems, such as Umbra [70]. Next, we run the same workload against memkind in the `MEMKIND_DEFAULT` mode. Internally this also uses jemalloc to manage the different arenas but adds an abstraction layer that deals with the different kinds of memories.

Results. The results are shown in Figure 5.6, which shows a run in which eight threads allocated memory concurrently. Grouped by allocation classes (powers of two), it shows both the mean cost of a single allocation (left) and the aggregate cost over the entire microbenchmark (right). The results for jemalloc and memkind are within close vicinity, suggesting that the allocation costs are very similar. In both cases, the mean cost constantly increases with growing allocation classes. At 32 KB (for memkind) and 64 KB (for jemalloc), the mean allocation cost doubles. Later, at 16 MB, the cost increases even more significantly. However, when looking at the summed allocation costs, the relatively high costs for larger allocations only contributes little to the overall costs.

Instead, the summed allocation costs show that the majority of the allocation costs come from small allocations in the range between 16 and 512 Byte.

In total, jemalloc performs slightly, but noticeably better than memkind. In absolute numbers, jemalloc spends 6.713 seconds on the allocations, while memkind needs 8.172 seconds. This means that for a workload specific to in-memory databases and based on the allocations seen in Hyrise, memkind is 22% slower.

Interpretation. The 22% overhead measured for memkind exceeds the 15% reported in the memkind paper [215]. While 22% might sound like a significant cost, it needs to be seen in the context of the microbenchmark, which mimics the allocation behavior of 220 seconds of TPC-H queries. It does not perform any actual work usually associated with query execution. Relative to those 220 seconds, the allocators' costs correspond to only 3.1% (when Hyrise uses jemalloc) and 3.7% (for memkind). We consider the difference of 0.6 percentage points to be bearable. Still, we cross-validate this result in an end-to-end benchmark in Section 5.5.4.

5.5.3 PMR-based Data Structures

Next, we look into the overhead of the Polymorphic Memory Resources (PMR). Again, we first evaluate this concept in isolation. This shows the worst-case costs of PMR. The PMR concept was described in Section 5.3. To recapitulate, polymorphic allocators differ from `std::allocator` in that they do not call statically malloc to allocate memory but draw the memory from a memory resource that is only known at runtime. Similarly, when the allocated data structure is deallocated, the memory has to be returned to the correct memory resource. The polymorphic allocator handles this by storing a pointer to the resource and calling virtual allocation and deallocation methods on that pointer. As such, the overhead of PMR is (1) the space in the allocator to store the pointer to the memory resource and (2) one virtual method call per allocation or deallocation. Accesses to the allocated space are not affected by PMR.

The increased memory consumption is less of an issue. Each container holds exactly one allocator object. For example, the compressed attribute vector of a dictionary-compressed segment (cf. Section 4.3.1) is stored as an `std::vector<ValueID>`. The memory occupied by a regular vector consists of three pointers plus the space allocated on the heap [179, page 70]. For a segment with the default size of 65 535 that stores 256 distinct values, this results in $3 \times 8\ \mathrm{B} + 65\,535 \times 1\ \mathrm{B} = 65\,669\ \mathrm{B}$. PMR only adds eight additional bytes to this. As such, for most data structures, the memory cost of PMR is negligible. There is one exception to this: for as long as the C++ library implementations do not fully support `ranges`, it is more efficient for strings in Hyrise to be individually placed on the heap instead of using a more compact form of storage. As the `std::string`s containers are subject to tiering, they also hold a PMR pointer. For the time being, this increases the overall footprint for TPC-H scale factor 10 by 7%. Once the `ranges`-based solution can be applied, the footprint increase caused by PMR can be reduced to 124 KB. This corresponds to less than 0.02% for the entire dataset.

Benchmark Configuration. To measure the runtime cost of PMR, we reuse the microbenchmark from the previous subsection. Our goal is to quantify the cost of the

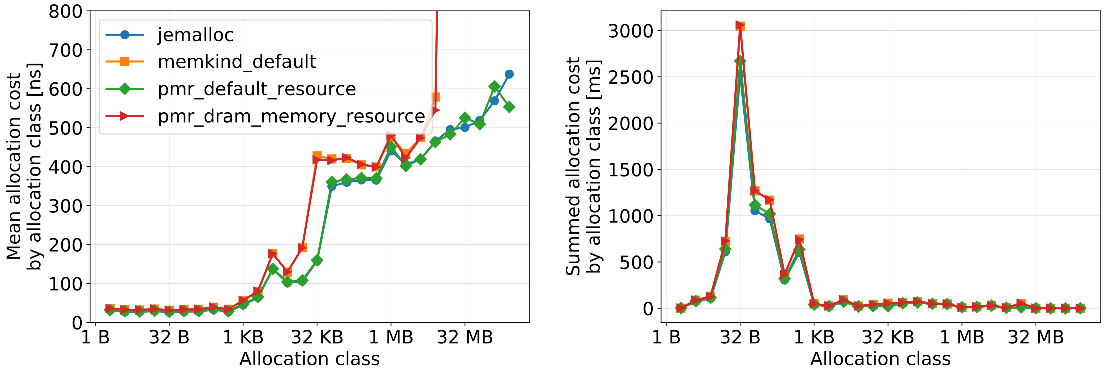


Figure 5.7: Costs for allocations via the PMR abstraction. The non-PMR allocation methods from Figure 5.6 (jemalloc and memkind_default) are included for comparison. For the most part, they overlap with their PMR counterparts.

virtual method calls to the individual allocators. Again, we replay the allocation behavior of Hyrise in a stand-alone binary. This time, we include the PMR indirection: instead of calling malloc or `memkind_malloc` directly, we benchmark memory resources that wrap the corresponding methods. In the case of malloc, this is done by the library's default resource (`pmr::get_default_resource`, which calls jemalloc); for memkind, we use a `memory_resource` wrapper that calls `memkind_malloc`.

Results. Figure 5.7 shows the results of this benchmark and compares them to those of the previous benchmark. Two sets of overlapping lines can be seen: first, jemalloc and its PMR version `default_resource` show a high degree of correlation. Second, `memkind_default` and `dram_memory_resource` are similarly correlated. The correlations can be found in both the graph for the mean allocation cost (left) and that of the summed allocation cost (right).

Table 5.3 shows the total cost of the different allocation methods depending on whether PMR is used or not. Interestingly, the overhead of PMR is higher for jemalloc than it is for memkind. This is reproducible across multiple runs.

Interpretation. Polymorphic Memory Resources add an overhead to the cost of allocations. This is caused by the cost of an additional virtual method call per allocation. Theoretically, we would expect this cost to be the same for both allocation methods. We currently do not have an explanation for the differing PMR overhead between jemalloc and memkind.

	non-PMR	PMR	relative
jemalloc	6.713	7.063	+ 5.2%
memkind	8.172	8.191	+ 0.2%
relative	+ 21.7%	+ 16.0%	

Table 5.3: Summed costs (in seconds) for different allocation methods.

By comparing jemalloc without PMR and memkind with PMR, we can now calculate the absolute overhead of our memory management approach: $(8.191/6.713) - 1 = 22\%$. As explained in Section 5.5.2, the allocations that are replayed in this benchmark correspond to a 220 second execution of the TPC-H benchmark in Hyrise. This puts the absolute overhead of $8.191 - 6.713 = 1.478$ seconds into perspective.

5.5.4 End-to-End Allocation Performance

To conclude the evaluation for the allocation and abstraction layers, we perform an end-to-end benchmark of Hyrise without PMR, with PMR, and with PMR plus memkind. This comparison allows us to evaluate whether we have fulfilled our constraint of not introducing a significant overhead to the traditional DRAM-only code base.

Benchmark Configuration. We run the TPC-H benchmark in the same configuration as used previously in Section 4.6.3. Most importantly, this involves 56 worker threads that execute queries and thus allocate memory in parallel. The benchmark queries are executed by 50 clients in a shuffled order.

We define three configurations as follows: for (1) the benchmark of Hyrise without PMR, we disable PMR by replacing the `polymorphic_allocator` with a regular `std::allocator`. Next, (2) Hyrise with PMR is the unmodified Hyrise master branch. Finally, we measure the (3) overhead of memkind by setting the default memory resource to a memory resource that calls `memkind_alloc` instead of (je)malloc.

Results. Table 5.4 shows the results of this benchmark. For both PMR and memkind, we see a measurably increased latency as well as a reduction in the throughput. The total overhead of PMR + memkind, however, is dominated by memkind.

Interpretation. Compared to the isolated benchmarks in the previous subsection, the overhead of PMR and memkind are significantly lower. This is expected, as the isolated benchmarks performed no work other than allocating memory. For the execution of the TPC-H benchmark, these allocations are only a small, unavoidable step. Because of the different workloads, the relative changes cannot be directly compared between the benchmarks. However, the tendency of memkind being responsible for the biggest share of the performance overhead remains.

We conclude that PMR alone adds an overhead of 0.6% to the DRAM-only codebase. This is low enough not only to justify the flexibility gained by automatic tiering, but also low enough to justify the use of PMR throughout the codebase even if automatic tiering is not used. In the case of data stored on DRAM, we can avoid the additional overhead of memkind by allocating memory directly from (je)malloc. For NVM, for

	Latency [ms]	rel. to (1)	Throughput [Tx/s]	rel. to (1)
(1) jemalloc only	108 421.5		0.4553	
(2) PMR	109 039.5	0.6%	0.4539	−0.3%
(3) PMR + memkind	111 747.3	3.1%	0.4423	−2.8%

Table 5.4: Absolute latency and throughput numbers of different allocation strategies.

which memkind cannot be avoided as easily, we accept an additional overhead of 2.5 percentage points (+3.1% latency instead of +0.6%). This overhead is notable, but not high enough to justify the search for memkind alternatives.

5.5.5 Migration from DRAM to DRAM

Finally, we quantify the cost of migrating data between different tiers. This cost is comprised of two parts, namely the implementation-defined cost of allocating and copying the data structures, and second the hardware-defined cost of reading from and writing to the different tiers. The first is compute-bound, the second is bandwidth-bound. In the first benchmark, we limit the evaluation to the former by migrating data from DRAM to DRAM, thus excluding the additional costs hardware of lower tiers. In the following benchmark, we then include these tiers.

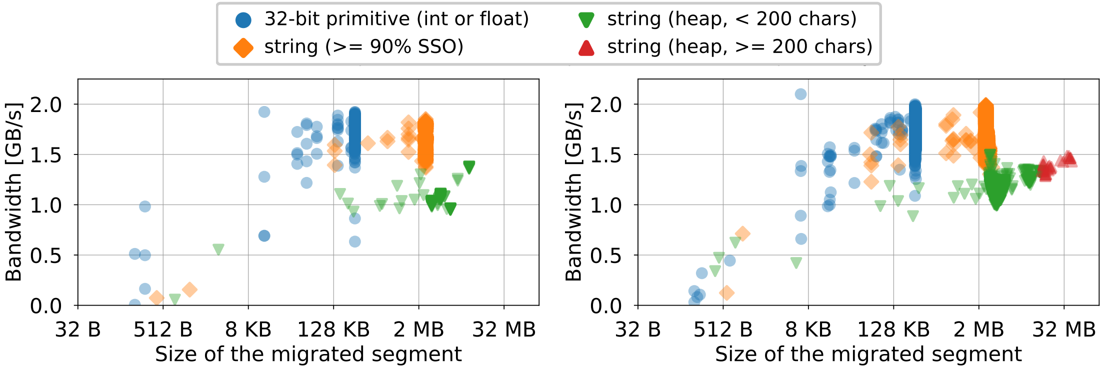
Benchmark Setup. For the benchmark, we migrate data from the TPC-H benchmark as well as from the Join Order Benchmark (JOB) [149]. TPC-H uses uniformly generated, synthetic data, JOB uses data from the Internet Movie Database, i.e., real-world data with realistic value distributions. Unlike the default setting used by Hyrise, we do not use dictionary compression. Using uncompressed data allows us to reason about the impact of different data types more easily, as we do not have to take the compression ratio and the number of bits used for the attribute vectors into account. We re-include dictionary compression in the subsequent benchmark.

The data is fully loaded into DRAM using the Hyrise benchmark binaries. Instead of executing the benchmark queries, the binaries are instrumented to migrate the data to a different DRAM pool and to measure the time elapsed per migrated segment. The migration is performed by a single thread. This matches the usage pattern of our automatic tiering implementation.

Results. Figure 5.8 shows the cost of migrating the two datasets. The x-axis gives the size of the migrated segment, the y-axis the effective bandwidth of the migration. The color and shape of the markers represent the different data types. For the purpose of the migration, `int`s and `float`s are indistinguishable - both are primitive (i.e., trivially copyable [37, page 68]) 32-bit values. For strings, we distinguish short strings for which the small-string optimization (SSO) avoids heap allocations[5] from strings that use heap-allocated memory. We make the following observations:

1. Two vertical clusters of segments with the same size can be identified. These correspond to the 32-bit types and to SSO strings.
2. As the size of the migrated segments grows, so does the migration bandwidth.
3. For strings, the migration of SSO strings results in a higher bandwidth than that of heap-allocated strings. Long heap-allocated strings are migrated with a higher bandwidth than short heap-allocated strings.
4. The bandwidth maxes out at around 2 GB/s. This bandwidth is only reached by primitive data structures and by SSO strings.

[5] In most C++ libraries, the memory allocated for a `basic_string` is used either for the begin, end, and capacity pointers or for storing the string in the object itself. For short strings, this avoids the need of having to allocate heap memory and improves the strings' cache-efficiency [164, page 205].



(a) TPC-H

(b) Join Order Benchmark

Figure 5.8: Migration costs between two DRAM pools. Each marker represents a single segment, its color represents its data type. Segments in which 90% or more of all strings are stored using the small-string optimization (SSO) are placed in a separate bucket.

Interpretation. We interpret the four findings as follows:

1. The two clusters of segments with the same size, i.e., the two vertical lines, can be explained by the chunk size in combination with the data type. With a maximum chunk size of 65 535, a full segment of 32-bit integers consumes 65 535 × 4 B ≈ 262 KB. No segment that contains primitive 32-bit values can be larger. However, there are some smaller segments. These correspond to the last chunks of each table, which do not necessarily reach the maximum chunk size. For SSO strings, the cluster can be explained similarly. Each string object requires 4×8 B, corresponding to the four 64-bit pointers for the begin, end, capacity, and the underlying memory resource. This results in 65 535 × 32 B ≈ 2 097 KB, which is where the cluster is located.
2. When migrating small segments, the fixed part of the costs (allocating memory, atomically replacing the segment) takes a larger share of the overall costs. This causes the migration of small segments to be less efficient.
3. Migrating segments with a large number of heap-allocated strings is more expensive than migrating the same number of SSO strings. This does not come as a surprise, as each migrated string now results in an additional allocation (with the costs as discussed in the previous experiment). These allocation costs are increasingly amortized for longer heap-allocated strings.
4. The maximum bandwidth of 2 GB/s is significantly lower than the maximum DRAM bandwidth of a single core. A comparative benchmark using fio with the `--memcpytest` parameter shows a maximum bandwidth of 8 GB/s. We believe that the discrepancy is caused by the way `libstdc++` implements the copy algorithm for vectors: even for trivially copyable data types, the entries are copied one-by-one instead of copying them en-bloc using SIMD-enabled `memcpy`.

5.5.6 Migration between DRAM and Lower Tiers

Having measured the maximum bandwidth for the migration of a single segment by “migrating” from DRAM to DRAM, we now look into how long it takes to migrate an entire data set from DRAM to a lower tier. It is not our goal to make this step as fast as possible. After all, Hyrise shall migrate data asynchronously. Still, we need to quantify the bandwidth that can be reached and verify that we do not introduce any unexpected bottlenecks.

Benchmark Setup. The setup is similar to that of the previous experiment. Instead of only migrating from DRAM to DRAM, we now use all available tiers. Furthermore, we re-introduce dictionary compression. For both datasets, we then measure the time spent to migrate the segments from DRAM to the lower tier.

Results. Table 5.5 shows the results for the different configurations. For all configurations, the migration to lower tiers takes longer than to higher tiers. The effect of the data size of the segments is visible. For DRAM and the SSDs, a compression factor of 3x (TPC-H) or 2x (Join-Order Benchmark) results in a correspondingly shorter migration cost. For NVM, the difference is less pronounced: when the Join-Order Benchmark data is compressed by 2x, the migration cost is only reduced by 23%.

Interpretation. The migration to lower tiers taking longer matches the expectations. In the case of the two SSDs, the migration benefits from the data first being written to umap’s DRAM buffer cache, before it is written as an entire page to the SSDs. This allows the migration to reach the maximum bandwidth shown in Figure 5.5. For NVM, this is not the case, and the effect of NVM’s read/write asymmetry becomes visible. This causes the migration to NVM to be slightly slower than to DRAM-buffered tiers.

In a future version, this bottleneck could be removed by a custom implementation. Alternatively, more CPU cores could be dedicated to the migration process. Because the system’s architecture calls for the migration to happen asynchronously, we currently do not see any benefit in dedicating additional resources to it.

Dataset	TPC-H		Join-Order Benchmark	
Compression DRAM-to-...	Uncompr. (3.048 GB)	Dictionary (1.015 GB)	Uncompr. (14.946 GB)	Dictionary (6.687 GB)
DRAM	2 137 μs	780 μs	10 063 μs	5 008 μs
NVM	3 094 μs	2 400 μs	10 690 μs	7 757 μs
Intel P4800X	7 191 μs	2 429 μs	33 629 μs	15 487 μs
Samsung 850 Pro	9 796 μs	3 591 μs	46 538 μs	21 489 μs

Table 5.5: Total duration of the migration for different configurations, consisting of different benchmark datasets and compression methods. All data starts on DRAM and is then migrated to the tier listed in the left-most column.

5.5.7 Summary of the Evaluation

In six benchmarks, we have evaluated the performance of the memory management stack. For our access methods mmap and umap, we found that the maximum bandwidth can only be reached when data is accessed sequentially and by multiple threads. This was found to be especially important when data on SSDs is accessed using the umap abstraction. For allocating memory via memkind and PMR, we reported an end-to-end overhead of 0.6% for DRAM and 3.1% for NVM. Finally, for the migration between tiers, we found that these do not yet reach the maximum possible bandwidth but are sufficiently fast enough for our purposes.

5.6 Use Beyond Automatic Tiering

Can these abstractions be used for other challenges outside of automatic tiering?

The concepts described in this section are not limited to automatic tiering. The PMR-based allocation and migration model is beneficial in many cases in which a simple malloc from default memory is not sufficient. An example is the targeted placement of **data on different NUMA nodes**: co-locating data and execution leads to a more efficient use of available hardware resources. In turn, this improves the throughput of both relational [20, 124, 196] and non-relational database systems [90, 265]. Currently, Hyrise allocates memory using the operating system's default policy, which tries to allocate memory on the current node. With our memory management model, we can add a memory resource that makes this node assignment explicit. In turn, this would allow us to migrate data from an overloaded node to another one. This migration would use the same mechanisms that are used for moving data between memory and storage tiers. We have previously investigated this as part of two supervised Master's Theses [175, 210]. While we were able to show benefits especially for the scan operator, the work revealed (and fixed) more pressing multi-threading issues than NUMA placement-specific optimizations. Because of this, NUMA-specific memory resources are currently not used.

In Figure 5.3, we have discussed **layered resources**, which are memory resources that draw their memory from another upstream memory resource. Hyrise uses the `monotonic_buffer_resource`, which is optimized for ephemeral allocations. In our implementation of the hash-based aggregate operator, using this layer is a three-line change and results in a 5% throughput benefit for TPC-H Query 1. A second example is the use of the `unsynchronized_pool_resource` in the hash-based join operator. In this operator, many small position lists are allocated and regularly resized. Using layers allows us to serve these frequent allocations from a thread-local pool.

Another use of layered resources is to **track the semantic context of allocations** in order to understand where memory is allocated. Most allocation profilers track information with a line-of-code granularity. While this helps programmers in understanding where the memory was used, it does not help them in understanding which "*instance of a data structure*" is consuming or accessing the memory [178]. PMR provides a natural of providing such a mapping. By providing information such as the table and column name to the memory resource, allocations could be tracked on a semantic level.

For all use cases, a main benefit is that the static data types remain the same. No code other than the memory resource that is passed into the polymorphic allocator has to be modified. Furthermore, the described layers can be combined. It is not only conceivable, but also reasonable, to have (1) tracked, (2) NUMA-aware allocations (3) on DRAM and NVM, which use (4) their own, optimized allocation method for fine-grained allocations.

5.7 Summary

In this chapter, we have presented our memory management framework. It is used to allocate and access data on different tiers and to migrate data structures between tiers. The design proposed in this chapter answers the first research question:

> *How can data be stored on different memory and storage tiers in a transparent manner that is consistent with the DRAM-first approach of in-memory databases and does not negatively affect the performance when accessing data stored on DRAM?*

To make the handling of multiple tiers as unintrusive to the remaining code basis as possible, we propose a new combination of polymorphic memory resources (PMR) and the memkind and umap libraries. While the implementation of this requires a certain level of C++ proficiency, it makes *using* the abstraction in the rest of the system much easier. We measured an overhead of 0.6% for DRAM and 3.1% for NVM, which we consider to be low enough to be used in productive settings. With this, we enable transparent migrations to lower tiers without sacrificing the DRAM-first approach of IMDBs and its performance benefits. As such, the design proposed in this chapter is a partial answer to our *first* research question:

Furthermore, we have shown how data structures can be migrated across different tiers. This addresses part of our second research question:

> *How can a DBMS automatically [...] [migrate] data without disrupting the continuous operation of the system?*

By performing this migration on immutable chunks, it can be performed without influencing concurrent queries. During the migration, the data exists on multiple tiers, which allows ongoing transactions to complete their work on the old copy of the data. Because the migration is implemented using the C++ copy constructor, arbitrary data structures can be migrated without modifying their internal implementation.

6 Access Tracking

Establishing which data is frequently used and which data is seldom accessed is the second pillar of automated data tiering. If we want the system to make any decisions autonomously, it needs to have data on which to base these decisions. Collecting this data is the job of the access tracking component.

In this chapter, we describe different options for implementing access tracking and discuss their advantages and disadvantages. We describe our choices and evaluate our tracking approach with regards to its information quality and performance impact. Finally, we discuss how having an efficient access tracking mechanism helps with challenges beyond those of automatic tiering.

We contribute a novel access tracking method that allows us to quantify accesses in both the row and column dimensions as well as the distribution of access patterns within consecutive accesses. By reusing the iterator abstraction layer in Hyrise [25], we can design the access counters in a way in which their overhead cannot be identified in benchmarks.

A first version of the access counters has been implemented as part of a previous Master's thesis [78] that was supervised during the work on this thesis. In addition to having guided the original implementation, the contribution of this thesis consists of the full implementation of access counters for all types of data segments, the fine-tuning of the access pattern detection, the differentiation to existing approaches, as well as the TPC-H/JCC-H-based evaluation.

6.1 Possible Approaches

A suitable access tracking method needs to fulfil three criteria. First, it needs to contain the right information for the task at hand. Second, the tracking mechanism needs to be designed in a modular and non-intrusive way; tight coupling between the accessing code and access counters complicates the code. Third, the performance impact needs to be low enough so that it does not outweigh the benefits achieved by whatever builds on the performance counter. This is deliberately phrased vaguely, as tracing components that are enabled by the user (i.e., the DBA) have a higher accepted performance impact than automatic tuning solutions that should be invisible to the user. Bearing these requirements in mind allows us to discuss two key design decisions, namely the granularity of the tracking and the actual tracking method.

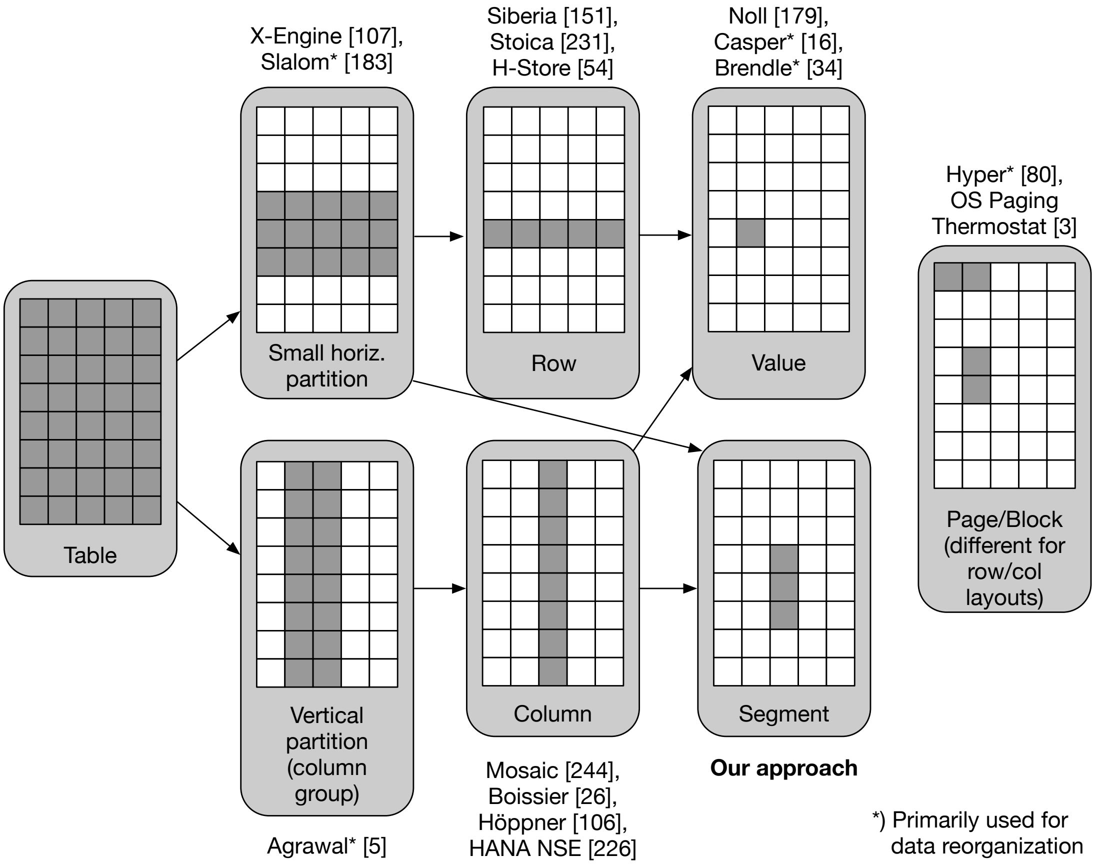


Figure 6.1: Different access tracking granularities and their use in commercial and research database systems. Grey boxes denote the atomic unit of tracking within which accesses are indistinguishable.

6.1.1 Granularity

The granularity describes the unit of data on which accesses are tracked. For the use case of automatic tiering, this depends on the desired tiering dimensions. Row-level counters do not hold any information that allows for identifying unused columns, while column-level counters cannot track any temporal skew in data accesses. The choice of a granularity level is also dependent on the database system's physical layout and additionally desired purposes of the access counters.

Figure 6.1 gives an overview of different tracking granularities used by commercial and research database systems that have been described in previous work. Going beyond the general discussion of these systems in Section 3.1, we now focus on the granularity of their access tracking methods. On the far left, the entire **table** is used for the tracking granularity, which is too coarse to be the basis of meaningful access statistics. Starting from there, the tracking granularity is refined in the row (shown on the top) and the column (bottom) dimensions.

In the row dimension, **small horizontal partitions** are the first possible granularity. In Hyrise, these partitions are called chunks. X-Engine [107] calls them *extents* and tracks their "*access frequency in a recent window*" in order to move "*colder*" extents to lower storage tiers. Slalom [182] is a DBMS that operates on external data that is mostly stored in CSV files. Instead of loading this data into a DBMS-proprietary format, queries are directly executed on that external data. To speed up this process, Slalom automatically builds indexes on heavily accessed horizontal partitions. These are identified using partition-based counters. Tracking accesses on the granularity of small horizontal partitions works well when the access patterns within such a partition are mostly homogeneous.

Choosing a finer granularity leads us to **row-level** access tracking. This is done by Siberia [151], Stoica and Ailamaki [230], and the Anti-Caching approach built for H-Store [54]. Unlike access tracking with a granularity of horizontal partitions, row-level tracking does not assume access correlations between co-located rows. It is thus more accurate. At the same time, it increases costs when updating and storing data. For this reason, some approaches use sampling to track only the statistically relevant rows [230].

In the column dimension, partitions or **column groups** are rarely used as the granularity of access tracking. Unlike rows, whose number may easily reach the billions, tables usually do not exceed a three-digit number of columns. As such, grouping columns does not significantly save tracking and storage costs. A rare exception is Microsoft SQL Server [5], which uses the number of queries that access a column group to optimize the partitioning layout. Even there, we suspect that this metric is actually aggregated from individual column access counts.

Tracking data on a **column level** is the natural granularity for column-oriented systems. This granularity is used by Mosaic [243] (a storage component integrated into Umbra [177]), the previous version of Hyrise [26], SAP HANA as modified by Höppner et al. [106], as well as SAP HANA's Near Storage Extension (NSE) [225]. Höppner et al. found that "*[o]nly about one-third of the columns are accessed in more than 5% of all queries*" [106]. Column-level counters are well suited for identifying these cases. For row stores, on the other hand, column level counters are only of limited use, as even unused columns cannot be separated from the row record. They may, however, still be useful for use cases such as automatic index selection.

The intersection of the row- and column-level tracking approaches is to track accesses on the level of individual **values**. While this provides the most accurate information, it is too expensive for most purposes. Noll et al. [178] propose a tracking approach that works on individual memory addresses and comes closest to tracking accesses on individual values. In their case, this information allows for manual introspection of the system's behavior, for example to analyze access patterns within operators. Casper [16] tracks accesses to single values in a column to establish optimal partition boundaries. Brendle et al. [34] capture value *domain* accesses in addition to accesses on small horizontal partitions in order to identify value boundaries by which a table should be re-partitioned.

In the scope of this thesis, we assume that data has already been partitioned in a way that co-locates data with similar access characteristics (cf. Section 1.4). We use a combination of small horizontal partitions (in our case: chunks) and columns as the granularity for our access tracking. In Section 6.3.1, we show how these **segment-level** counters allow us to capture the access patterns that are relevant for our use case and outperform access counters that are solely on the chunk or column level with regards to their information quality.

We include one more option for the sake of completeness: access tracking on the **page or block level** has been the default for all disk-based systems that use a buffer manager to load and unload pages from disk. There are too many of these to list here. It continues to be an approach used by systems without a buffer cache. HyPer [80] tracks page accesses and maps the virtual addresses to allocated data structures. They use this information to identify hot and cold clusters in the data. While tracking accesses on a page level may result in a higher granularity, they later aggregate the data on a *vector* level, which is the equivalent to segment-level in Hyrise. Next, Thermostat [3] is a DBMS-agnostic kernel extension that identifies "*cold*" pages with more fine-grained information than traditional OS paging but without mapping the accesses to rows or columns. Page-level tracking is also used by traditional OS paging in the case of a memory shortage.

For our work, we have decided for segment-level granularity and against page-level granularity for the following reasons: first, Hyrise does not yet have an explicit concept of pages. Introducing another level of granularity with additional statistics objects below the segment level would complicate both the architecture and the code. Second, not all compression methods store memory in one consecutive block of memory. For example, in the case of dictionary compression, the attribute vector and the dictionary are stored at different memory locations and thus different pages. This further complicates tracking. Third, it would require us to consider the page size in our method, which makes it harder to run Hyrise on different architectures and to compare results.

6.1.2 Tracking Method

Besides the access granularity, the tracking method is another architectural decision. Accesses can, for example, be tracked by incrementing a counter or by reusing the hardware's page bits. Depending on the method, it may or may not be possible to quantify the number of accesses. Knowing the access frequency helps automatic tiering because having a low, but non-zero, number of accesses to lower tier data may be acceptable, while a high number of accesses would require the page to be moved to a higher storage tier. Similarly, some tracking methods allow for the access pattern (sequential, monotonic, or random) to be identified. Again, this information can help in automatic tiering as sequential accesses are significantly cheaper than random accesses on most (if not all) memory and storage devices. An important finding from Section 5.5 was that our tiering approach should take these access types into account.

Different methods have a different overhead on query execution. Depending on the use case, the acceptable cost varies. Some tracking methods have to rely on sampling, as tracking each access would be prohibitively expensive. Table 6.1 shows different tracking methods as used by different systems and analyzes them in these dimensions.

The different systems also use different **approaches** to track the data. Counters and

	Granularity	Approach	Quantitative	Pattern Identif.	Sampling	Overhead / Limitations
Boissier [26]	Column	Plan Cache	Yes	Indirect	No	
Brendle [34]	Value	Unclear	No	No	Future work	14-19%
Casper [16]	Value	Trace	Yes	Yes	Yes	
H-Store [54]	Row	LRU Chain	No	No	Yes	5%
HANA NSE [225]	Column	Counter	Yes	No	No	
Hyper [80]	Page/Block	MMU	No (page bit)	No	No	“virtually no”, kernel only before 2.6.25
Höppner [106]	Column	Plan Cache	Yes	Indirect	No	
Mosaic [243]	Column	Trace	Yes	Focuses on Scans only	No	
Noll [178]	Attribute	Trace	Yes	Yes	Yes	27%-230%
OS Paging	Page/Block	MMU	No (page bit)	No	No	virtually no (MMU internal), no user control
Siberia [151]	Row	Trace	Yes	Unused	Optional	
Slalom [182]	Hor. Part.	Counter	Yes	No *	No	
Stoica [230]	Row	Trace	Yes	Unused	Yes	
Thermostat [3]	Page/Block	MMU	Yes (estim. via TLB misses)	No	Yes	<1%, kernel only
X-Engine [107]	Hor. Part.	Unclear	Yes	No *	No *	
Our approach	**Segment**	**Counter**	**Yes**	**Yes**	**No**	**<1%**

Table 6.1: Access tracking methods compared in different dimensions. Entries marked with a asterisk are not explicitly discussed in the corresponding publication and thus only reflect our understanding of the presenting approach.

traces actively instrument the code and collect data during execution. Counter-based approaches, such as used by Slalom [182], SAP HANA NSE [225], and by our implementation, update one or multiple counters as data on the given granularity level is accessed. Traces (or logs) are buffers that are filled during execution with row identifiers [151, 230], column identifiers [243], or logical addresses [178] of the data that has been accessed. These buffers are then processed by an asynchronous process. To further reduce the performance impact, this work can even be done on external compute units [151, 230]. H-Store [54] uses the uncommon approach of storing an LRU chain, i.e., a double-linked list of recently accessed pages.

Other approaches do not add instrumentation but use already existing data. Boissier et al. [26] (based on the previous work of Hyrise) as well as Höppner et al. [106] use pre-existing information from the query plan cache to identify heavily used columns. HyPer [80] and Thermostat [3] re-use MMU page flags, which are usually used by the operating system's paging algorithms to identify pages to evict. Because these flags are maintained anyway, these approaches usually have a very low overhead.

The question of whether accesses can be **quantified** and whether **access patterns** across multiple accesses can be identified is immediately decided by the chosen approach. Traces can always be analyzed both for the number of accesses and their pattern. For counters, only those patterns that are tracked at collection time can be counted. Plan caches can theoretically derive the access pattern by identifying whether non-order-preserving operators have been executed, but they cannot accurately quantify the number of accesses. As the plan cache is not updated with the actual cardinalities, they have to resort to the optimizer's cardinality estimations, which are known to be inaccurate [149]. Other approaches, such as H-Store's LRU chain or the MMU's page bits hold only binary, thus qualitative, information.

Finally, the need to **sample** accesses is usually established by the overhead of the tracking method if it was unsampled. Some approaches, such as that used by Siberia [151] make sampling optional. This way the allow the consumer of the tracking data to fine-tune the counters' overhead and balance it with the desired accuracy.

Our novel approach combines most benefits of the other solution: we can accurately count the number of accesses and can identify three types of access patterns. This information is available at run-time and does not need to be post-processed. At the same time, our use of the iterator abstraction layer in Hyrise allows us to minimize the tracking costs without resorting to sampling. By tracking accesses on the segment granularity, we achieve a higher precision than solely row- or column-oriented approaches.

6.2 Implementation

We implement access tracking as an addition to the existing segment iterators. These iterators decouple the compression methods from operators that access the data (cf. Section 4.3.2). Building the counters into the iterators has the advantage of being operator-agnostic. No operator code has to be modified to support access counters. This supports our goal of modularity.

Figure 6.2 shows how operators, iterators, and segment access counters interact. It is based on Figure 4.5, which was previously used to introduce the PosList indirection. Two scans are shown: one operates on a data table (i.e., a table as created and stored by the user), the other operates on a temporary table that is created by the first scan. In both cases, iterators are used to access the table data. The number of accessed values can be determined in the iterator's destructor: for an iterator that has reached the end of a data segment, the number of accessed values is equal to the size of the segment. For temporary tables, for which the data has already been filtered or joined, the number of accessed values is equal to the number of values in the PosList that defines the reference segment. If the iterator has not reached the end of the segment (e.g., in the case of a *limit* clause) or the iterator was used in a non-sequential manner (e.g., for binary searchers), the calculation is adapted accordingly.

In addition to the number of accessed values, the segment access counters also track information about the predominant access pattern. Data tables are accessed sequentially except for binary searches on sorted segments. For temporary tables, the access pattern is identified by sampling the values of the PosList. We distinguish between sequential (linear) accesses, monotonic accesses (in which some values are skipped, but the traversal is mono-directional), and random accesses.

Due to the inter- and intra-query parallelism in Hyrise, multiple iterators can access a segment. To synchronize the access to the counters, atomic integers are used. As the counters are only updated once per iterator creation, rather than once per iteration, the number of write accesses is relatively low and does not become a performance factor.

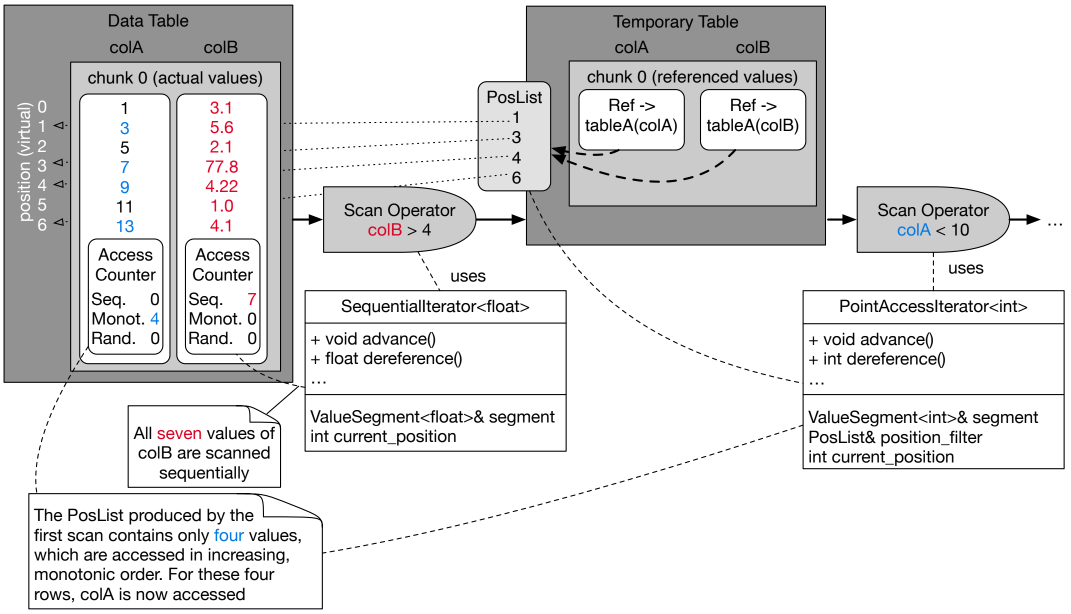


Figure 6.2: A table is scanned by two scan operators. Internally, these operators use iterators to access the raw values of the underlying data table. It is the job of the iterators to update the segment access counters. The first scan sequentially scans the table, accessing seven values in the second column. The second scan only accesses those four rows that were selected in the first scan. Because these are not sequential anymore, they are accessed in a monotonically increasing pattern. The access counter for the first column is updated accordingly.

For a scale factor of 10 and when executing each query once, a total of 421 003 atomic accesses is performed. This translates to roughly 400 atomic accesses per second or 3.2 MB/s. Previous research has found the "bandwidth" for atomic compare-and-swap modifications on contended addresses to be in the range of gigabytes per second even on hardware from 2012 [224]. The experiments in Section 6.3.2 show that the cost of these atomic updates are, indeed, not of relevance.

The counters themselves do not store a temporal component. Instead, consumers of the counters can easily establish the temporal dimension by periodically retrieving the monotonically increasing counter value and comparing it to the value seen in the previous time period. We consider this approach to be superior to one that reports the number of accesses in, e.g., the last minute, as it does not impose any limitations on the temporal dimension's granularity. Instead, every consumer of the counters can retrieve their values at an interval that is independent of other consumers.

Two interfaces expose the access counters to consumers. First, for internal consumers (i.e., those that run in the DBMS process and can use the C++ API), segment counters are directly accessible via the segment itself. For a given segment `s`, `s.access_counter[AccessType::Sequential]` returns how often `s` has been sequentially read since it was created. Second, segment access counters can also be read using our SQL meta table interface. The meta_segments table contains not only information about each segment’s memory consumption and compression method, but also reports the number of accesses to that segment, broken down by access pattern. This allows external consumers, such as the Hyrise Cockpit [128], to visualize this information and make it consumable for, e.g., the database administrator.

6.3 Evaluation

We now evaluate the segment access counters both from an information quality and a performance point of view.

6.3.1 Information Quality

One of our arguments for using segment-level access counters is that they provide more accurate information than column- or row-level counters alone. To evaluate this claim, we again use the TPC-H benchmark. When executing the different benchmark queries, the state of the access counters is automatically logged to a JSON file. This is part of the benchmark binary and can be enabled with the --metrics option. These metrics are then used to plot access heatmaps.

An example of an access heatmap is shown in Figure 6.3. It displays the number of accesses for TPC-H Query 5. On the y-axis, it shows the different tables and their columns. On the x-axis, it shows the chunks. The number of chunks depends on the row count of the table. The color symbolizes the number of accesses, with brighter colors corresponding to a higher number of accesses, and black corresponding to a segment that is never accessed. Additionally, the average number of accesses per chunk and column are plotted.

Out of the 22 chunks in the orders table, five chunks are not accessed at all. This can be explained by a quick look into the SQL query, shown in Listing 6.1. The query filters the orders table for orders in a given year. The TPC-H specification defines this parameter as a “*randomly selected year within [1993 .. 1997]*”. TPC-H, however, contains data for the years 1992 to 1998. Because the orders table is sorted by o_orderkey, the first and last chunks do not contain any orders for the years requested by Query 5. Thanks to the chunk statistics, the optimizer can prune these chunks before any

```sql
SELECT n_name, SUM(l_extendedprice * (1.0 - l_discount))
FROM customer, orders, lineitem, supplier, nation, region
WHERE l_orderkey = o_orderkey AND /* more join criteria */
  AND r_name = 'AMERICA' AND o_orderdate >= '1994-01-01'
  AND o_orderdate < '1995-01-01'
GROUP BY n_name;
```

Listing 6.1: TPC-H Query 5 with sample parameters, slightly condensed.

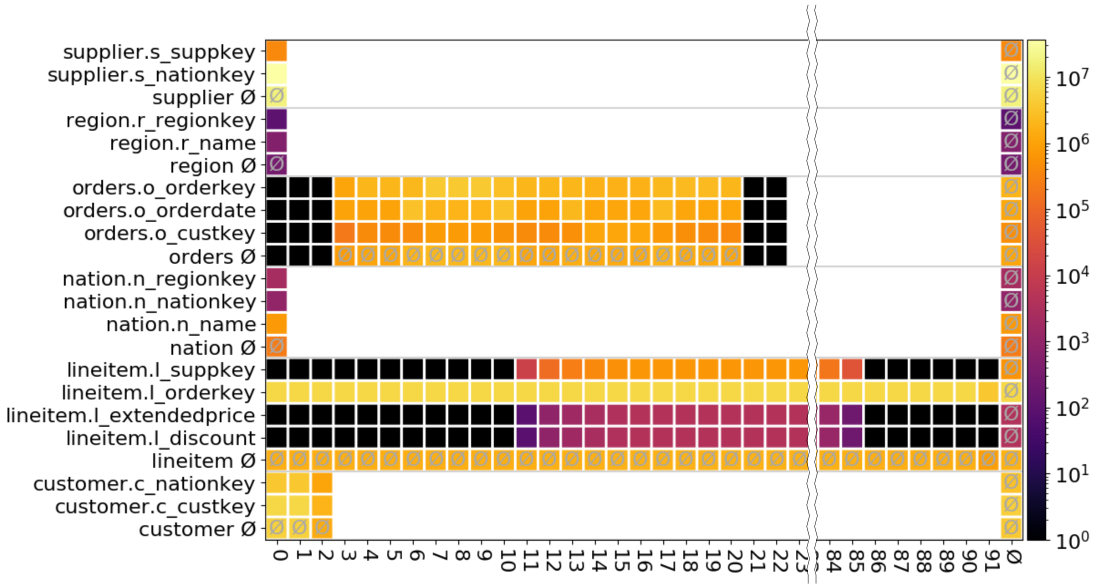


Figure 6.3: Segment accesses for TPC-H Query 5, visualized as a heatmap with logarithmic access frequency-to-color mapping. Segments printed in black are not accessed at all, yellow segments are heavily accessed.

operator is executed. Without pruning, we would expect these chunks to show some accesses in the orders.o_orderdate line, caused by the scans, but no accesses in the other columns.

The heatmap also shows that for 17 of 91 chunks in the lineitem table, some segments are not accessed at all. The reason for this is less obvious as it is for the orders table. Query 5 does not contain any filters on these columns. However, the lack of accesses can be explained by the join between the lineitem and orders table. Both tables are ordered by a date column. In the case of lineitem, this is l_shipdate, for orders, it is o_orderdate. The shipping date and the order date correlate. As such, the lineitem chunks at the beginning and the end of the table contain only line items that belong to orders outside of the queried range. Unlike the orders table, this information is not known at optimization time, because the correlation between l_shipdate and o_orderdate is unknown to the optimizer. As such, we cannot avoid accesses to l_orderkey.

Because the access pattern is skewed on two dimensions, only segment-level counters can identify the two described patterns. This demonstrates the benefit of our approach. We can find further arguments for using segment-level granularity when looking at the JCC-H benchmark [27]. As described in Section 4.6.1, JCC-H uses the same schema and queries as TPC-H but adds additional skew to the data and query parameter generation. Because the entire purpose of the access counters is to identify skews in data accesses, JCC-H gives us an impression of whether this goal has been reached.

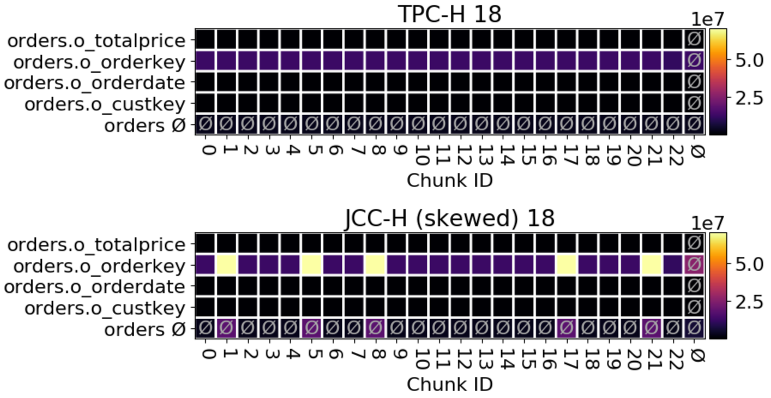


Figure 6.4: Access heatmap for TPC-H (top) and JCC-H (bottom) queries 18, only the `orders` and `lineitem` tables are shown. Unlike Figure 6.3, a linear color map is used to make the identification of absolute values easier.

In Figure 6.4, we can see the effect of this added skew, namely that of "Black Friday": "*JCC-H introduces a so-called Black Friday, which is one day in the year where there are many more orders. [...] [O]n this day, 50% of all orders are placed. [...] However, we do not do this in 2 out 7 the years, namely 1995 and 1996. [...] The reason is that 1995 and 1996 should be sanctuaries from join skew.*" [27]

The `orders` table has 22 chunks, which cover the seven years (1992-1998) for which data is generated. Two years have no Black Fridays. The remaining five Black Fridays can clearly be visually identified in the heatmap. Again, this access pattern would remain undiscovered if only column level access counters were used.

6.3.2 Tracking Cost

Having shown the value of accurate per-segment access counters, we now discuss their performance impact. We show that this impact is less than one percent and can be considered negligible. To reach this conclusion, we compare three configurations: Version 1 uses no access counters, Version 2 uses non-atomic access counters, and Version 3 uses full access counters. Non-atomic access counters are a middle ground, for which writes between multiple threads are not synchronized. In rare cases, when two threads update the same access counter at the same time, one of these updates thus may be lost[1].

Table 6.2 shows the absolute latencies and their relative latency change for the 22 TPC-H queries. The benchmark is executed with a high number of parallel clients (50) in order to stress potential synchronization issues. For non-atomic counters, the latency increases slightly. Across all queries, we report additional costs of 0.9%. For full (i.e., atomic) counters, the latency surprisingly decreases. Compared to the version without counters, the benchmark with full counters is 1.1% faster. This is counter-intuitive,

[1] Technically, non-synchronized accesses are *undefined behavior* in C++. At least on x86, however, the hardware guarantees atomic modifications of the access counters' 64-bit values.

	No Counters	Non-Atomic Counters		Full Counters	
Query	Latency	Latency	Relative	Latency	Relative
1	11 566.2	111 713.8	+1.3%	11 534.6	-0.3%
2	61.4	162.1	+1.1%	61.9	+0.8%
3	4309.5	44 296.6	-0.3%	4270.3	-0.9%
4	4114.3	34 121.9	+0.2%	4069.2	-1.1%
5	10 549.2	110 520.7	-0.3%	10 497.4	-0.5%
6	113.4	2113.9	+0.4%	113.6	+0.2%
7	1794.6	21 805.6	+0.6%	1786.2	-0.5%
8	961.9	1960.7	-0.1%	949.8	-1.3%
9	15 915.8	215 926.6	+0.1%	15 812.8	-0.6%
10	5957.6	65 941.5	-0.3%	5831.9	-2.1%
11	399.8	4404.0	+1.1%	395.7	-1.0%
12	1298.3	11 303.6	+0.4%	1287.2	-0.9%
13	10 436.6	210 611.5	+1.7%	10 444.7	+0.1%
14	764.3	1767.4	+0.4%	753.6	-1.4%
15	374.6	4375.6	+0.3%	371.4	-0.9%
16	2000.8	42 021.3	+1.0%	1975.4	-1.3%
17	292.5	4292.4	+0.0%	290.3	-0.8%
18	26 794.1	527 468.9	+2.5%	26 184.0	-2.3%
19	317.5	7321.9	+1.4%	315.3	-0.7%
20	348.0	5352.2	+1.2%	345.1	-0.8%
21	11 035.2	111 054.9	+0.2%	10 882.7	-1.4%
22	1330.6	21 336.6	+0.5%	1306.1	-1.8%
Sum	110 736.3	111 773.7	+0.9%	109 479.3	-1.1%

Table 6.2: Absolute TPC-H latency numbers and relative changes for non-atomic and full counters. Scale Factor 10, 56 cores, 50 clients.

as the added code has to have some cost. In the best case, this cost would not be measurable, but, at least theoretically, it should not make the overall execution faster. This behavior, however, is something that we have seen and reported before [63]. It is stable for a given code version, which is why additional runs do not change the results. Previous research has identified code alignment and environment changes as possible culprits [170, 171]. For example, instruction code that was added at the initialization site of the iterator could cause an operator's hot loop to be moved further "down" in the binary to a place where it fits on a single page and where the execution causes less cache misses. Unfortunately, existing experimental mitigation strategies [48] cannot be applied to Hyrise as they rely on outdated compiler versions that are not supported by Hyrise.

The goal was to show that segment access counters come with only a minimal, and thus acceptable, overhead. Without claiming that we overachieved this goal, we conclude that the performance impact of the counters is below the analytical limits and thus negligible.

6.4 Use Beyond Automatic Tiering

In addition to being necessary for identifying rarely used data that can be moved to lower memory or storage tiers, segment-level access counters can be used for other purposes, too. In our research group, other research projects also benefit from these counters:

We already explained that Hyrise supports different compression methods (cf. Section 4.3.1). This thesis exclusively uses dictionary-compressed segments. However, there are cases in which other compression algorithms are more beneficial. For example, for data that is locally highly repetitive, such as order dates, run-length compression is more suitable. As a second example, when compressing strings, heavy-weight compression algorithms, such as LZ4, often achieve a better compression rate than dictionary compression. Using a more heavy-weight compression algorithm does, however, result in higher access costs. In our group, we work on **choosing the most efficient compression scheme** based on data characteristics, access patterns, and the available memory budget. Currently, our implementation uses column-level statistics from the query plan cache to estimate the access frequency for different segments [25]. Segment-level access counters make it possible to make better decisions on a more accurate per-segment level.

A recently completed Master's thesis in our group looks at reducing the access costs for indexes. As of now, Hyrise stores indexes on a per-chunk level. While this virtually eliminates maintenance costs for immutable chunks, it means that index lookup costs increase linearly with the number of indexed chunks. This unnecessarily limits the index performance especially for index-based joins. As a solution, we are working on indexes that cover multiple chunks. This raises the question of whether all chunks should be indexed. Knowing which chunks are heavily accessed and which access patterns are predominant can improve the efficiency of **index selection** algorithms.

Finally, our group is looking into partial table replication. Replication is often used to improve the overall availability of the system [145]. Often, the entire database is replicated to a second server of the same size. This is not only useful from a High Availability or Disaster Recovery (HA/DR) aspect but can also be used to improve the performance of the system. Beyond this, additional replicas can be added to further improve the performance. These do not necessarily need to store all data, as the data is already sufficiently distributed from an HA/DR perspective. Such *heterogeneous replicas*, which only store parts of the data, can be more efficient from a cost perspective [94]. To **decide which data should be replicated** on these heterogeneous replicas, it is beneficial to have accurate access information in both table dimensions. Again, this information can be provided by our segment access counters.

6.5 Summary

In this chapter, we have presented a novel access tracking approach. With this, we can answer part of our second research question:

> *How can a DBMS automatically identify those parts of data that should remain on DRAM for performance reasons [...]?*

We have started by comparing existing approaches and discussing their benefits and drawbacks. From there, we have weighed different design options to find the solution that works best with our automatic tiering concept but can also be used for other projects in the realm of self-driving databases. We propose to track data with a segment-level granularity, which combines the advantages of tracking small horizontal partitions [107, 182] with those of column-level tracking [26, 106, 225, 243]. To this, we add the identification of access patterns, which most existing approaches do not support. The implemented solution leverages an existing abstraction layer, namely our iterators. Doing so minimizes its architectural impact and its performance costs. Using the TPC-H and JCC-H benchmarks as examples, we have shown how this novel tracking approach helps to identify patterns in data that would otherwise remain hidden. Finally, we have experimentally verified that our access counters do not have a measurable negative impact on the performance of the system.

7 Decision Making

The third, and final, pillar of our automatic tiering solution is the decision making. Its goal is to make efficient use of a given memory budget by keeping frequently used data close to the CPU and moving less frequently used data to lower tiers. The main source of information for these decisions are the segment access counters (cf. Chapter 6). Based on the number of accesses to the individual segments and the predominant access patterns, segments are moved to one of the available tiers by calling the migration methods of our memory management framework (cf. Chapter 5).

Our vision is for Hyrise to make complex autonomous decisions based on the workload, the system utilization, and the tuning options provided by the self-driving plugins. The *driver* is responsible for combining these parameters and choosing the most efficient combination of the possible options (cf. Section 4.2). Centralizing these responsibilities in the driver brings a number of advantages to the individual self-driving plugins:

- A single driver can evaluate the combined consequences of tuning options from different plugins. For example, both the compression and the tiering plugins aim at satisfying a given memory constraint [25]. When executed separately, this can lead to unwanted effects in which the actions of one plugin (e.g., compressing data to reduce the DRAM footprint) cause counteracting actions of the other plugin (increasing the footprint by moving data from NVM to DRAM). A central driver allows us to prevent these undesirable positive feedback loops. Going one step further, it also enables us to exploit synergy effects between different tiering options: when segments are moved to lower tiers, it may be beneficial to use a heavier compression algorithm in order to better exploit the available bandwidth.
- Sharing common subcomponents, like models and solvers, not only improves the code quality, but also allows us to decouple the research and engineering efforts. Instead of implementing workload classification and prediction in each plugin, these features can be independently developed as a part of the driver.
- Finally, having only a single driver makes monitoring and administering the system easier. These efforts include logging the tuning decisions that were made as well as the KPIs of the system. This data can later be used to identify issues in the tuning process. For Hyrise, this is done in the Hyrise Cockpit [128]. This cockpit also enables the DBA to influence the behavior of the system by inputting certain constraints, rolling back or overriding tuning decisions, or even using an emergency stop button for all self-driving features.

For this chapter, we implemented a driver with limited functionality. It allows us to evaluate the benefits of automatic tiering in isolation. At the same time, we are prepared to replace the interim driver and to integrate the tiering functionality into a more sophisticated driver. This driver is part of a separate ongoing research project.

This chapter is organized as follows: we first explain the implementation of our decision making component and discuss its theoretical foundation. Next, we evaluate the decisions made by the component and show the impact of different optimizations. As part of this, we show how our segment-level granularity and the tracking of the access patterns are used to improve the value function used in the optimization process. Finally, we summarize our current decision making approach.

7.1 Implementation

The goal of the decision making component is to find the configuration in which a given DRAM budget is respected and the performance of the system is maximized. Storing a segment on DRAM improves the performance (i.e., it adds value) but consumes part of the DRAM budget (i.e., it has a cost). A possible abstraction for this type of problem is the *knapsack problem*.

When translated into a knapsack problem, storing a segment on DRAM corresponds to packing an item into the knapsack. The segment's memory consumption is defined as its cost; its number of accesses is used as the segment's value. With this, we optimize the profit density: if, across all segments, more accesses can be satisfied from DRAM, the database will spend less time loading data from lower tiers and will thus be faster. In its current form, decision making consists of three steps:

1. *Collecting statistics:* All segments are queried for their access counters. These counters track the number of accesses since the start of the Hyrise process. Because we want the decision making to quickly react to changing workloads, we need to convert these figures into an interval-based metric. This is done by calculating the difference in the number of accesses in two subsequent intervals. The difference is then used as the value for the segment. We also retrieve the memory consumption of the segment and use it as the segment's weight.
2. *Assigning segments to tiers:* In the second step, the knapsack problem is solved. For this, we use a branch-and-bound solver [141], which is implemented as part of Google's Operations Research Tools [192, 193]. This solver was chosen because of its high result quality even with low runtime limits and because it was one of the few C++-based solvers that could be used in the Hyrise project without licensing issues. In Section 7.2.4, we evaluate the quality of its results and show that they are sufficiently accurate.
3. *Reconfiguration:* Once a new tiering configuration has been calculated, it is applied. Segments are migrated using the object migration approach described in Section 5.4. This is done concurrently with the ongoing query execution.

A common challenge of the knapsack problem is that it assumes the linearity of the items: packing or unpacking an item must not influence the value of other items. For other aspects of a self-tuning database, this becomes an issue. For example, in index selection, a multi-attribute index can also be used for single-attribute queries. Packing the former reduces the value added by the latter. Schlosser et al. find that in these cases, the interaction between different items turns the knapsack abstraction into "*an* *oversimplification of the problem*" [216]. In the case of automatic tiering, however, this type of interaction is less of a concern: the cost of reading data from segments on different tiers is linearly composed of the individual access costs. Even though the available bandwidth of a tier is shared between all accesses to that tier, an exhaustion of that bandwidth affects all segments equally and does not systematically change the value of individual segments.

A learning from our low-level hardware benchmarks in Section 5.5 was that while random accesses are slower than sequential reads for all tiers, this effect is more pronounced for lower tiers such as NVM and SSD. We took this as a reason to build access counters that can identify these access patterns (cf. Section 6.2). To incorporate this information in our decision making process, we opted for a weight-based approach: random accesses are multiplied with a factor that corresponds to the additional cost of accessing the data. We discuss the benefit of adding this factor in Section 7.2.

7.1.1 Knapsack for Multiple Tiers

The previous description is limited to a single knapsack for which a segment is stored on DRAM (part of the knapsack) or NVM (not part of the knapsack). It can be generalized to provide solutions for more than two tiers. This generalization of the knapsack problem is called the multilevel generalized assignment problem (MGAP). First described by Glover et al. [85], it has been more succinctly summarized by Öncan as follows [183]:

> *"Given n items and m knapsacks, the Generalized Assignment Problem (GAP) is to find the optimum assignment of each item to exactly one knapsack, without exceeding the capacity of any knapsack. [...] The [Multilevel GAP (MGAP)] deals with the determination of the minimum cost assignment of tasks to agents with varying efficiency levels. The key difference between the MGAP and the classical GAP is that in the former, agents can perform tasks at more than one efficiency level."*

In our case, *tasks* correspond to segments being stored, *agents* to the different tiers, and the *efficiency level* to the performance of these tiers. Based on this definition, we can use MGAP models to identify the optimal placement of data across tiers. While different approaches to solving the MGAP problem have been discussed, there is no open source and ready-to-use implementation of such a solver.

If, instead of requiring the optimal solution, we accept a reasonably good heuristic solution, we can instead also model the different tiers as a series of independent knapsack problems: first, we exhaust the first knapsack's budget by placing the most valuable (i.e., most heavily accessed) segments on DRAM. Next, the NVM budget is exhausted by choosing the most valuable segments from the *remaining* segments. Finally, all segments that have been placed neither on DRAM nor NVM are placed on SSD.

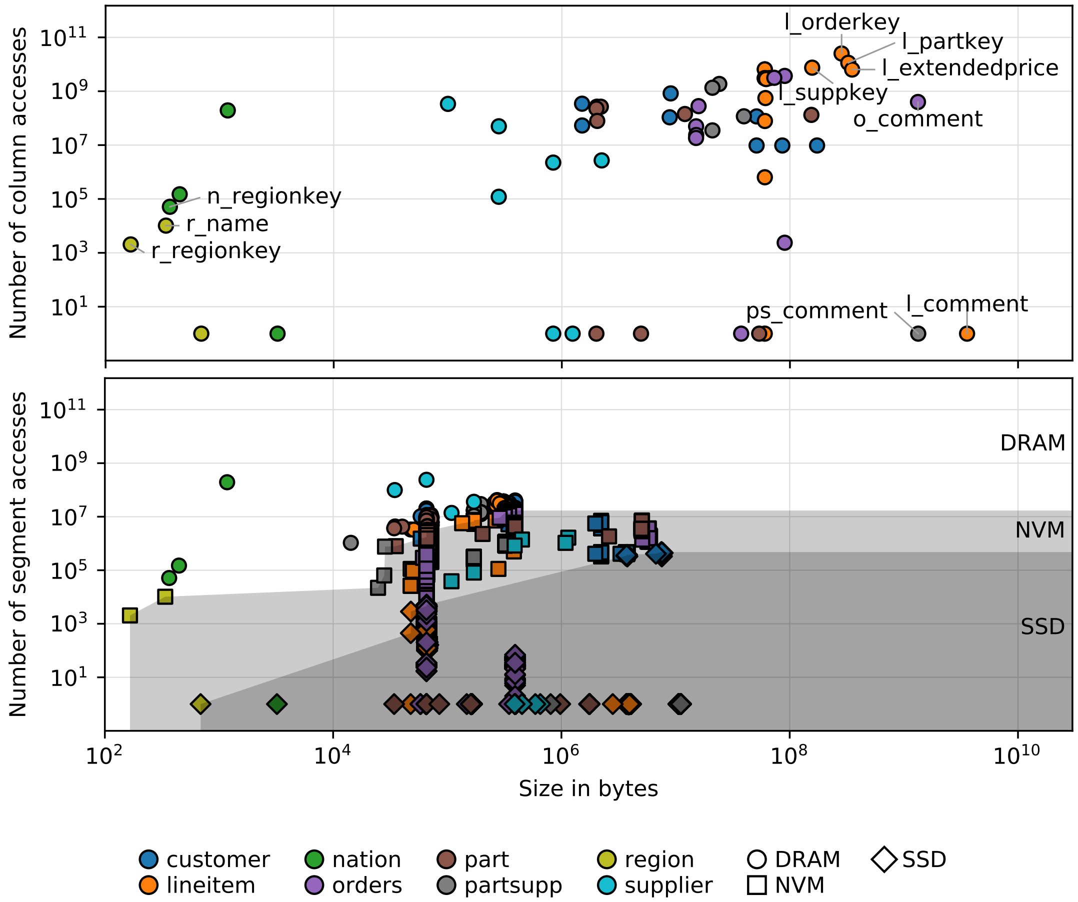


Figure 7.1: Visualization of the knapsack problem and the resulting tiering decision. Each marker in the upper graph corresponds to a *column* in the TPC-H benchmark. Some extremes are labeled with their column names. The lower graph shows the individual *segments*. A sample tiering configuration with a DRAM capacity of 10% and an NVM capacity of 30% is shown.

7.1.2 Visualization of the Decision Making Process

Figure 7.1 shows how the knapsack approach is applied to the TPC-H data. It contains two subfigures, with the first graph showing one marker per *column* of the TPC-H tables and the second graph showing one marker per *segment*. Because most TPC-H tables are stored in multiple chunks, the second graph contains significantly more markers.

The x-axis gives the size of these columns and segments, the y-axis gives the number of accesses to the respective column or segment. Note that both axes are scaled logarithmically. Different colors denote the different TPC-H tables, the shape of the markers in the lower graph visualizes the tier on which the data is stored. The data set has a size of 10 GB. In Section 7.2.3, we show that our approach also works equally well for larger database sizes.

Looking at the different columns of the TPC-H data in the upper graph, we observe that some columns are not being accessed at all[1]. Together, they account for 56% of the data set. The inclusion of unused columns in the benchmark's data set can be explained by the row-oriented history of most traditional analytical database systems. By including a large string in each tuple, the TPC-H authors increased the amount of data that has to be read per tuple. Column-oriented systems, such as Hyrise, store `l_comment` in a separate memory region. These unused columns are thus ignored during the execution of the benchmark and the corresponding memory regions are never touched. As such, they are a great candidate for being migrated to a lower memory tier. However, not all comment columns are unused. For example, `o_comment` from the `orders` table, shown on the top right, is among the most frequently accessed columns.

The lower graph in Figure 7.1 differs from the upper graph in that the individual segments of the columns are shown. For small tables like `nation` and `region`, which fit into a single chunk, this makes no difference. For tables that hold multiple chunks, the markers move towards the bottom left as both the size and the number of accesses is lower for the individual segments. To a certain degree, columns seen in the upper graph can still be identified in the lower graph. For example, the `l_comment` segments are still visible at 3×10^6 B. Also notable are two vertical clusters between 10^4 and 10^6 B. These are caused by the default size of the segments: with a fixed maximum chunk size, many chunks use a similar amount of memory. Smaller variations are introduced by the size of the segments' dictionaries. This is comparable to the pattern previously seen in Figure 5.8, where we used an allocation trace from Hyrise to evaluate the performance of our allocators and identified two clusters in the allocation sizes.

Additionally, the second graph shows an example tiering configuration. The white area denotes segments stored on DRAM and the shaded areas denote segments stored on NVM and SSD, respectively. We allowed for 10% of the data to be stored on DRAM, another 30% on NVM, and store the remaining data on SSD. These budgets are used by the knapsack algorithm and result in an assignment of the segments to the different tiers as shown by the shape of the markers. We also added a skyline polygon to better visualize the boundaries of the tiers. Note that the choice of the axis limits together with the logarithmic scale distorts the visual perception and makes the DRAM look bigger than 10%.

7.2 Evaluation

The decision making component connects the access tracking and the memory management components. Without access tracking, it cannot create the knapsack model; without memory management, the resulting decisions cannot be realized and the performance impact of a good or bad decision cannot be quantified. As such, the decision making component can only be benchmarked in end-to-end experiments. Not only do these experiments allow us to verify and quantify the individual influence factors for a

[1] To include them in a graph with logarithmic scaling, these are plotted with y=1.

close-to-optimal tiering solution, but they also serve as an end-to-end validation of our entire tiering approach.

7.2.1 Benchmark Setup

In our benchmarks, we show how a certain configuration of the decision making algorithm influences the performance of the TPC-H benchmark. For this, we measure the TPC-H query latency under different DRAM budgets. Once our implementation is embedded in a centralized driver for Hyrise, the DRAM budget will be set by that driver. In the absence of budgets provided by the driver, we benchmark pre-defined DRAM budgets, which are defined as a percentage of the entire data set. Every five minutes, a new budget is chosen and the tiering configuration is automatically adapted accordingly. We call this five minute period an *epoch*.

Each individual query execution is written into a JSON result file. This JSON file also contains the start and end timestamps of each epoch as logged by the tiering plugin. These logs are not only visible in the JSON benchmark result, but are also made available to the DBA via the Hyrise Cockpit [128].

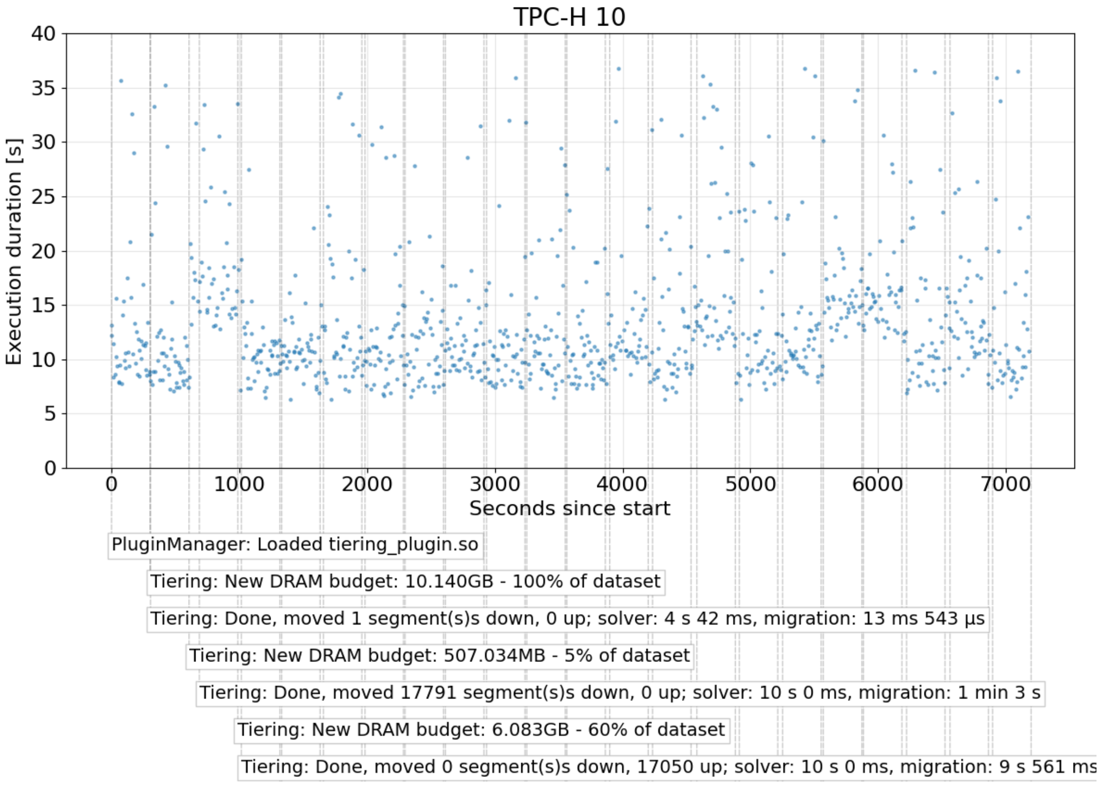
We execute the benchmarks on System B as previously described in Table 5.2. Because of the Linux kernel bug described in Section 5.5 and the configuration of the available benchmark systems, we are unable to benchmark the NVM and SSD tiers simultaneously and limit the discussion in this section to two tiers.

For the benchmarks, we use all available NUMA nodes. The 22 TPC-H queries are executed by 20 clients in parallel. As larger amounts of data result in longer benchmark durations, we use a scale factor of 10 for most experiments. This results in roughly 10 GB, partitioned across 19 142 segments. Each benchmark is executed for at least ten hours to allow for the measurements to stabilize. In Section 7.2.2, we show that the system works equally well for larger amounts of data.

The result of such a benchmark run is shown in Figure 7.2. We show the figure as an intermediary step to explain how we postprocess this raw data. The x-axis shows the progressing runtime of the benchmark, the y-axis represents the latency of a single query execution. While all 22 TPC-H queries were executed, the figure only shows the latencies of Query 10 for the sake of legibility. Each marker represents a single query execution. For example, a marker at $(x{:}900s, y{:}18s)$ means that 15 minutes after the start of the benchmark, TPC-H Query 10 was started and took 18 seconds to execute.

Dashed lines link the x-axis to log entries that were written by the tiering plugin when a different configuration was applied. These entries allow us to map individual query executions to epochs and thus to the DRAM budget that was valid at the given point in time. From the log entries, we know that at $(x{:}900s$, 5% of the data were stored on DRAM.

Some epochs can be clearly identified by the increased runtime of the query. For example, Query 10 takes longer to execute in the second period (spanning from 600 to 1000 seconds) than in the period before. The log messages explain why this is the case: in the second epoch, the DRAM budget was sufficient for only 5% of the dataset, meaning that 95% of the data were stored on NVM. In the first epoch, 100% of the data



```
PluginManager: Loaded tiering_plugin.so
Tiering: New DRAM budget: 10.140GB - 100% of dataset
Tiering: Done, moved 1 segment(s)s down, 0 up; solver: 4 s 42 ms, migration: 13 ms 543 µs
Tiering: New DRAM budget: 507.034MB - 5% of dataset
Tiering: Done, moved 17791 segment(s)s down, 0 up; solver: 10 s 0 ms, migration: 1 min 3 s
Tiering: New DRAM budget: 6.083GB - 60% of dataset
Tiering: Done, moved 0 segment(s)s down, 17050 up; solver: 10 s 0 ms, migration: 9 s 561 ms
```

Figure 7.2: TPC-H Q10 latency over time. Each dot represents a single execution of the query. At pre-defined intervals, the DRAM budget is modified. These intervals are marked and explained in the log entries at the bottom. The log is cropped on the bottom to conserve space. Twelve outliers with a duration of more than 40 seconds are not shown in the graph.

was stored on DRAM. As such, the increase in query latency is the expected effect of data tiering. While this effect is visible when comparing the first two epochs, it is harder to identify for other epochs.

Besides the average latency within a given epoch being hard to read, the graph also shows only a single TPC-H query and does not allow us to compare different benchmark runs. To make the graph more consumable, we perform two transformations: first, all query executions in an epoch are grouped and their mean duration is calculated. The time between the epochs, i.e., the time needed to calculate and apply a new tiering configuration, is excluded.

Second, the data is then pivoted so that the DRAM budget becomes the x-axis and the average runtime of all TPC-H queries becomes the y-axis. Furthermore, we increase the number of measured data points starting at 30% because this is the point at which most queries start to be affected. Doing so allows us to balance the benchmark's degree of detail with its execution duration. The result of these transformations is shown in

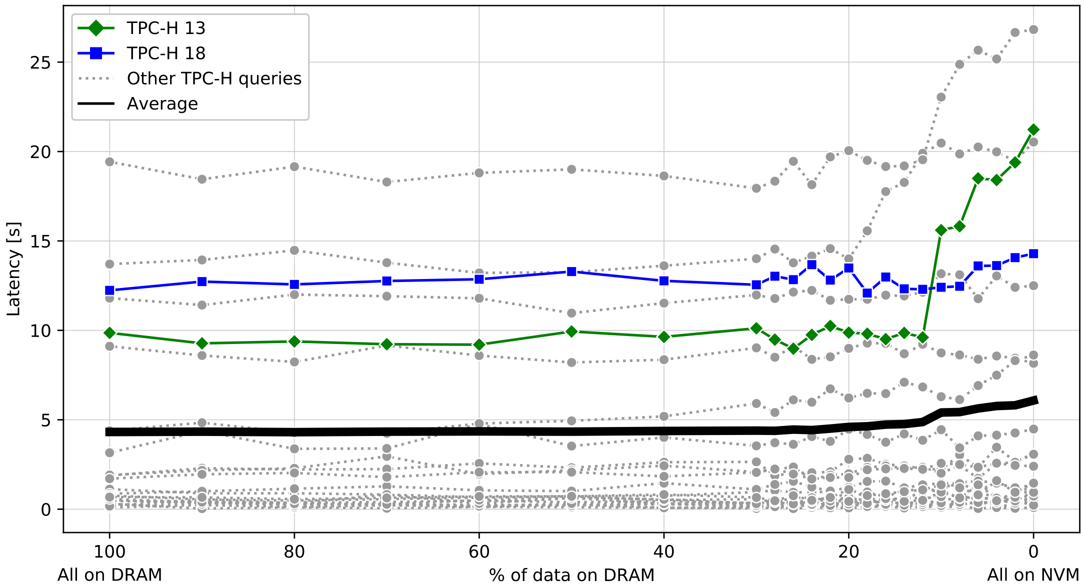


Figure 7.3: TPC-H latency as a function of the amount of data stored on DRAM.

Figure 7.3. Two queries, 13 and 18, will be discussed in detail. These are individually colored in green and blue, respectively. All other queries are not discussed in the text and are plotted in grey. Additionally, the average across all 22 queries is shown by a thick black line.

The far left side of the graph shows the latency at a point at which all data is stored on DRAM; the far right side represents all data being stored on NVM. Thus, the latency increasing towards the right side of the graph is the expected result: as more data is moved to NVM, the queries are increasingly affected by NVM's higher latency and its lower bandwidth.

Not all queries are affected equally. Query 18, plotted in blue, whose most expensive operator performs an unfiltered aggregation of TPC-H's biggest table, performs a number of sequential accesses for the aggregation. Queries with this access pattern generally tolerate higher memory latencies better. This is in line with our previous experiments and can be explained by the prefetcher hiding some of the increased access latencies. Query 13, shown in green, on the other hand, performs a large hash join that produces a large result table with columns that reference the input data in a random order. Resolving these indirections and accessing the underlying tables with a random access pattern is highly susceptible to increased latency, which cannot be hidden by the prefetcher. For Query 13, the latency drastically increases as the DRAM capacity is reduced from 10% to 8%. Here, an often referenced part of the `customer` table is migrated to NVM, causing the corresponding accesses to be slowed down.

The average latency of all queries (i.e., the thick black line) appears to vary only insignificantly. This is visually misleading and only caused by the scale of the y-axis.

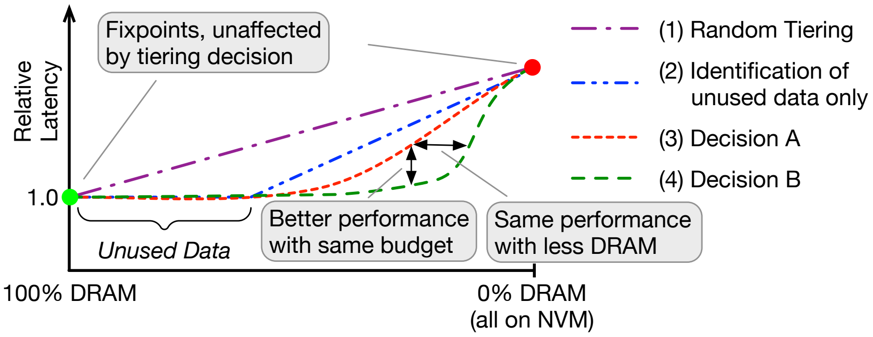


Figure 7.4: Different tiering decisions influencing the performance of the benchmark.

In absolute numbers, the latency increases from 4.3 seconds (all data on DRAM) to 6.1 seconds (all on NVM), which is an increase of 42%. For the following graphs, we focus on this relative change and choose the axes parameters accordingly.

We now run the TPC-H benchmark in different configurations, highlighting some of the workload-aware features enabled by our access counters. Figure 7.4 is a sketch on how the following graphs can be interpreted. Again, the x-axis shows the amount of data stored on DRAM and the y-axis shows the latency. Two fixpoints are marked: the one on the left denotes the latency measured when all data is stored on DRAM. This is the starting point of this thesis and is used as a reference point for the relative latency. The second fixpoint on the far right denotes the latency that is measured if all data is stored on NVM. In both cases, the decision making algorithm has no influence over the latency. Between the two fixpoints, the algorithm is responsible for keeping the latency as low as possible while increasing amounts of data are moved to lower tiers. The figure shows four examples of tiering decision approaches:

(1) With random tiering, segments are randomly moved from DRAM to lower tiers as the DRAM budget is decreased. The 191 000 segments are progressively moved to NVM. While not all of these segments have the same performance impact, the law of large numbers causes the relative latency to change in an approximately linear fashion.

(2) A first, trivial algorithm could simply identify those segments that are never accessed. In Section 7.1.2, we have explained that 55% of the data is unused in the TPC-H benchmark. Moving the corresponding segments to lower tiers does not cause an increase in latency. Once other segments, which are actually accessed, are moved to lower segments in a random order, the latency develops in the same way as it does with the previous approach.

(3/4) With *Decision A* and *B*, two algorithms are sketched that make different decisions about which segments should be moved to NVM first. For example, one algorithm could take the different access patterns into account while the other treats all accesses equally. While even the best algorithm will converge towards the right endpoint, its behavior up to that point determines its quality. An algorithm is *better* if it either results in a higher benchmark performance for a given budget or if it provides the same performance with less consumed DRAM.

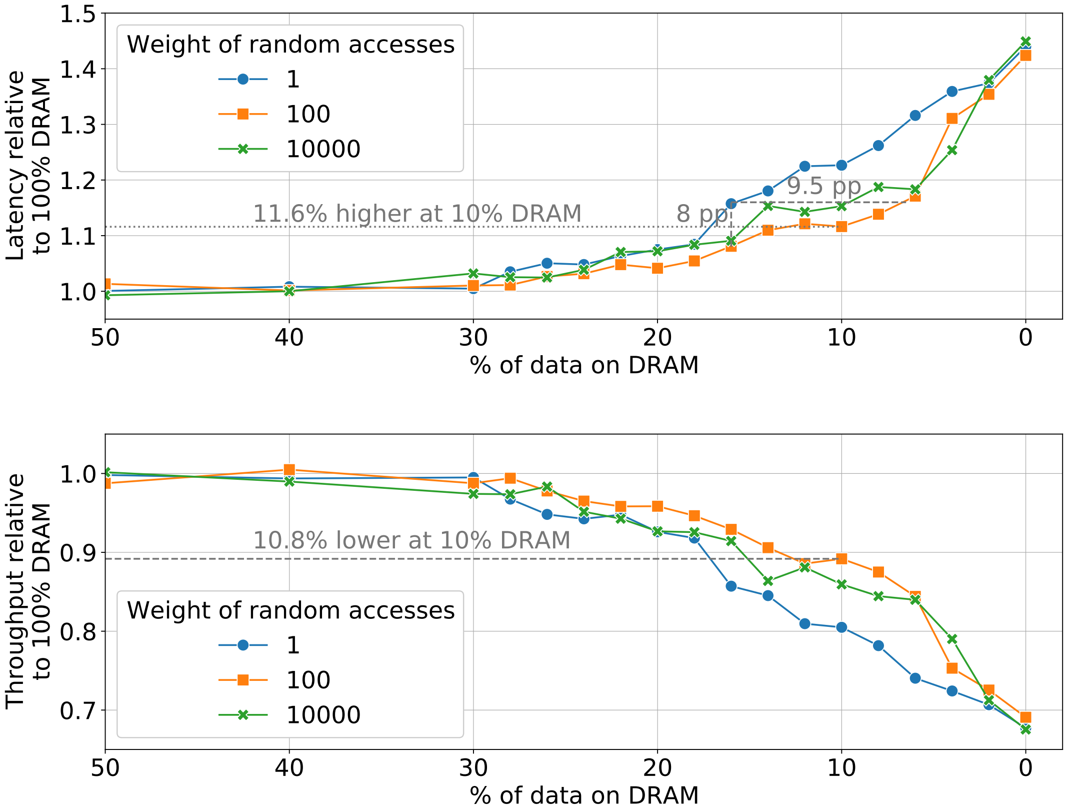


Figure 7.5: Influence of varying random access weights on the TPC-H performance. For a varying DRAM budget and different weights of random accesses, the database's latency and throughput are compared to a DRAM-only database.

7.2.2 Benefits of Weighted Access Patterns

In Section 5.5, we evaluated the costs of sequential and random accesses. When data is moved to lower tiers, the cost of random accesses increases not only in absolute numbers, but those random accesses also become relatively more expensive compared to sequential accesses. In other words, moving a segment to a lower tier hurts the performance disproportionately more if that segment is usually accessed in a random order.

To counteract these effects, our access counters identify these access patterns (cf. Section 6.2). In the decision making algorithm, these random accesses can then be given a greater weight compared to sequential accesses. This artificially increases the value of the segment as used in the knapsack algorithm and makes it more likely that the randomly accessed segment is stored on DRAM. Being able to incorporate the access patterns into the tiering decision is one of the features that sets Hyrise apart from other approaches. We now show how the inclusion of this information into the decision making algorithm improves the efficiency of its decisions.

Figure 7.5 shows three experiments in which the weight of the random accesses is varied between 1 (i.e., no added weight), 100, and 10000. It follows the layout previously explained in Section 7.2.1. We exclude the range between 100% and 50% DRAM as it only includes the previously identified unused data.

The graph shows that applying a weight to random accesses is beneficial for a range of different memory budgets. The impact depends on the chosen DRAM budget. For a fixed budget of 16%, a weight of 100 reduces the relative latency by 8 percentage points (pp). Alternatively, when keeping the relative latency fixed at 16%, applying a weight to the value function makes it possible to further reduce the DRAM budget by 9.5 pp. In other words, compared to an implementation that is unaware of access patterns, our approach allow us to move almost 10% *more* data to NVM without reducing the system's performance.

Besides weights of 1 and 100, we also show a weight of 10 000. The graph shows that this overestimates the costs of random accesses. Still, the decision making algorithm performs better with a weight of 10 000 than if it ignores the access patterns (i.e., uses a weight of 1). Intuitively, one would thus try to find the optimal weight factor. We do not expect for such a single factor to exist. Trying to determine such a single factor would likely cause the parameters to be overfitted a single benchmark. Instead, we propose that this parameter shall be automatically fine-tuned by the driver, which can use a cost model [130] to estimate the physical access costs related to a specific segment.

Compared to a DRAM-only database, the experiment shows that we can move 90% of the data to NVM (i.e., 10% remaining on DRAM) at a cost of increasing the latency by 11.6% and decreasing the throughput by 10.8%.

7.2.3 Benefits of Multi-Dimensional Tracking

In Chapter 6, we argue that accesses should be tracked on a more fine-granular level than just the individual columns. In Hyrise, we use the existing segment-based architecture both as a unit of access tracking and for data migration. In the following experiment, we show that segment-based tiering outperforms column-based and chunk-based tiering. Because the standard TPC-H data is largely homogeneous, we introduce skew into the benchmark by adding multiple tenants to the TPC-H data set. This is similar to a benchmark proposed by Braun et al. [33]. Each of the ten tenants has its own TPC-H data set with a scale factor of 10, resulting in an overall data size of 100 GB. Because the server used for benchmarking has only 192 GB of DRAM and the DBMS needs to allocate temporary memory for the individual operators, this 100 GB data set comes close to the maximum supported by this particular system.

While all tenants store the same amount of data, the number of issued queries varies. We use a binomial distribution with a p value of 0.5 and nine trials. This means that tenant 4 is responsible for approximately 25% of all queries, while tenants 0 and 9 only issue 1% of the queries. The tables are clustered by the tenant id. As a result, the segments storing data from tenant 4 are accessed 25 times more than those of tenants 0 and 9. We expect this pattern to be detected and that incorporating this information makes a material difference in the quality of the tiering decisions.

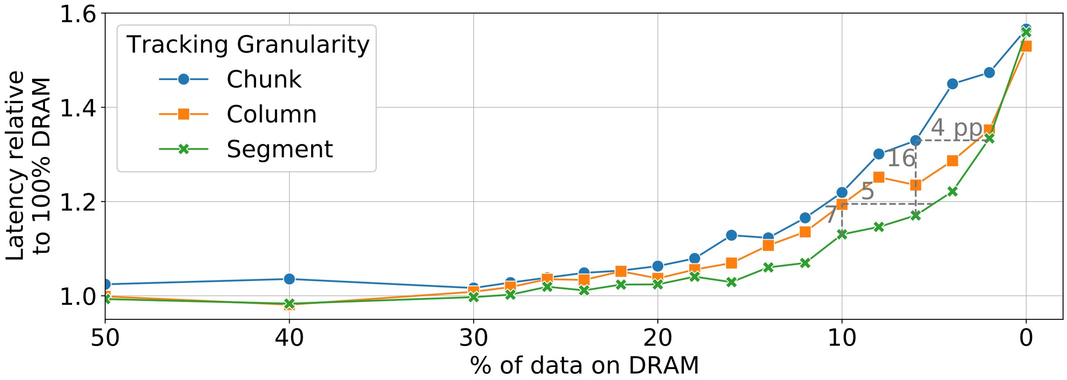


Figure 7.6: Benefit of access tracking and migration on a segment level.

The results of this benchmark are visualized in Figure 7.6. Because the latency and throughput graphs are largely reciprocal (cf. Figure 7.5), we only show one of them, namely the latency graph. Three lines are shown, corresponding to decision making on the chunk, column, and segment granularities. The segment granularity clearly outperforms the approaches in which a homogeneous access pattern across the entirety of a column or a chunk is assumed.

At a DRAM budget of 10%, segment-level tracking reduces the cost of data tiering by 7 percentage points. When optimizing for a minimal DRAM budget with a given acceptable latency increase, segment-level tracking can reduce the needed DRAM budget by 5% of the overall data set's size. Compared to the per-chunk granularity, segment-level tracking performs better because of the unused columns in the TPC-H benchmark. If the column dimension is ignored, these columns are included in the data that is kept on DRAM, where they occupy space that could otherwise be used for data that is actually accessed. For a budget of 6%, making tiering decisions on a segment-level granularity decreases the overhead by 16 percentage points compared to tracking on the chunk-level.

This benchmark shows that the segment granularity correctly identifies parts of the table that are accessed less frequently and that incorporating this information into the decision making algorithm consistently leads to better results.

7.2.4 Solver Efficiency

When describing the implementation of the decision making component in Section 7.1, we explained that we use the branch-and-bound knapsack solver included in Google's or-tools [192, 193]. Besides the aspects of licensing and availability, an important aspect that guides the choice of a solver is its speed and the quality of its solutions. For knapsack problems, for which the goal is to maximize the summed value of all items (the profit) while satisfying the capacity constraint, the quality of a solution can be described as the ratio between the profit of the found solution to that of the optimal solution.

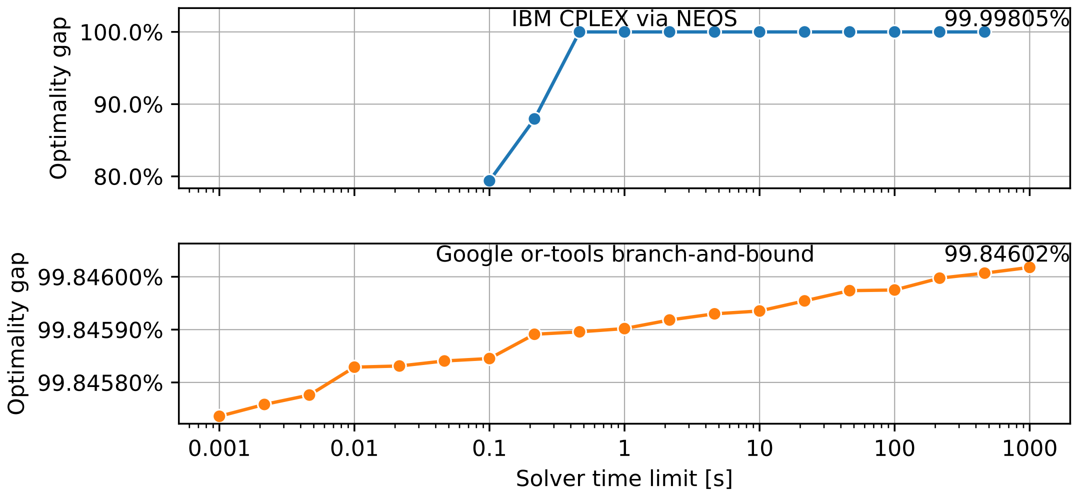


Figure 7.7: Quality of the solutions by CPLEX and or-tools as a function of their time budget. The optimality gap is the distance between the found solution and the best *known* bound.

Calculating the optimal solution of a knapsack problem with many items is infeasible. However, as part of the branch-and-bound algorithm, an *upper bound* is calculated and constantly updated. This upper bound limits the highest possible profit [172]. With an increasing number of iterations, the proposed solution and the upper bound converge. We can use this upper bound to estimate the quality of our solution.

To evaluate the speed and the quality of the solutions found by or-tools, we compare it to IBM ILOG CPLEX Optimization Studio (short CPLEX) [81], a commercial solver. CPLEX is available for free via the public NEOS (Network-Enabled Optimization System) server [49] and can be accessed via the Pyomo library [97]. The data set is taken from the benchmark in Section 7.1.2 and consists of 17 816 individual segments. We vary the time budget given to the solver. Between two powers of ten, we execute four measurements, resulting in data points at, e.g., 1s, 2.15s, 4.64s, and 10s. We track the profit of the proposed solution after this time budget has been exhausted as well as the lowest identified upper bound. The results in Figure 7.7 show the quality of the proposed solution as a percentage of the *globally* lowest upper bound.

CPLEX obtains a result that is within 99% of the optimal solution after 464 ms. Shorter time limits, such as 100 ms, lead to substantially worse results, and even shorter results lead to no result at all. The solver from or-tools, which we use for our tiering implementation, proposes a solution that is within 99.8% of the optimal solution in just 1 ms. At the same time, even after 1000 seconds, it does not reach the maximum result quality obtained by CPLEX: the best result produced by CPLEX has an overall value that is 0.15% higher than the best or-tools results. For Hyrise, we value the availability and speed of or-tools over the minimally higher solution value of CPLEX.

Based on this experiment, we also find that working on a segment granularity does not increase the problem size to a point at which solving the knapsack problem becomes infeasible. Even with low time budgets, solutions can be found that are within 99.8% to the optimal solution. Rather than optimizing for less than one percent, we believe that it makes more sense for future work to further optimize the profit metric.

7.3 Summary

In this chapter, we have described how tiering decisions are made and applied. For this, we have discussed how our decision making component fits into the bigger picture of Hyrise as a self-driving database and how it can interact with optimizations that are developed in other research projects. We have discussed how decision making for tiering can be reduced to a knapsack problem and visualized how this allows us to assign segments to multiple tiers.

The evaluation done for our decision making component also serves as an evaluation of our entire tiering concept. With several benchmarks, we verified the functionality of our implementation, quantified the benefit of fine-grained access counters, and measured the impact of different tiering decisions on end-to-end benchmarks. We report a reduction in the cost of tiering of up to 16 percentage points when accesses are tracked on the two-dimensional segment level compared to one-dimensional tracking. The knowledge of the data access patterns reduces the overhead by 8 percentage points.

Our implementation and evaluation of the decision making component completes our previous answer to the first part of the second research question:

> *How can a DBMS automatically identify those parts of data that should remain on DRAM for performance reasons [...]?*

Segments that are suitable for tiering can be identified by tracking their accesses in both the row and column dimensions as well as their access patterns. For this, we use the access counters presented in Chapter 6. Less frequently segments with predominantly sequential access patterns should be migrated to lower tiers first.

8 Discussion

In this chapter, we first discuss the limitations to the evaluations in this thesis and their potential impact on the generalizability of our contributions. Next, we suggest directions for future work on automatic tiering and potential synergy effects with related areas in database and hardware research. Finally, we conclude this thesis with a summary of the answers to our research questions and of the presented contributions.

8.1 Threats to Validity

Even though TPC-H is a widely accepted benchmark, its accuracy as a simulation for real-world workloads has been repeatedly questioned [173, 244]. We argue that both our memory management (Chapter 5) and our access tracking (Chapter 6) components are too low-level to be severely impacted by the choice of a benchmark. For our decision making component (Chapter 7) on the other hand, we expect different workloads to show different results. More specifically, we expect that the relative NVM overhead for a given memory budget varies with the workload's access patterns. As an example, we have discussed the fact that some *comment* columns in the TPC-H data are entirely unused. With this, the performance impact of tiering only becomes visible once 56% of the data has been moved to lower tiers. In other workloads, we expect less unused data and an earlier onset of tiering-related performance changes. Still, we have no reason to believe that the qualitative conclusions drawn from these experiments would change. Segment-based tiering will continue to outperform concepts in which access skew in the column or row dimensions is ignored. Moving sequentially accessed data to lower tiers will still be more efficient than moving randomly read data.

The evaluation could be improved by using a real-world workload that has been extracted from an existing, sufficiently large in-memory database. We were able to acquire metadata from an SAP customer's ERP system. This data was used for the case study presented in Section 1.1. The aggregated data allowed us to verify our assumptions but was insufficient for an evaluation of our tiering approach. Even if we had a full data export and query trace from that industry partner, the evaluation would run the risk of being overfitted towards the characteristics of that particular database. For a full evaluation, we would thus need to execute benchmarks on multiple data sets.

The problem of a lack of suitable datasets and corresponding workload traces is shared by large parts of the database research community [108, 244]. Some cloud database providers address this challenge by re-executing queries "*using a separate set of resources without impacting the customer's production workload*" and obfuscating the results to guarantee the data's confidentiality [259].

8.2 Future Work

So far, we have applied tiering to move the user-provided tables from DRAM to lower tiers. The **temporary data** generated during query execution remains in DRAM. In our benchmarks, this temporary data accounts for roughly half of the database's memory usage. The exact number varies depending on the number of concurrent queries and the input cardinalities of accumulating operators like joins and aggregates. We expect that not all of this data needs to be continuously stored in DRAM. Instead, we anticipate that some sequentially written and read data structures, such as the position lists generated during a scan (cf. Figure 4.5) or the materializations used for the hash join, could be good candidates for being stored on lower tiers. Daase et al. showed that our memory management framework already makes it possible to migrate temporary data but that further work is needed to do this in a way that does not harm the query performance [50]. Hassan et al. [98] present heuristic data placement for hash joins on hybrid DRAM and NVM memory. We expect that Hyrise can benefit from their contributions.

Our architecture was designed to easily accommodate **additional tiers**. We especially see great potential in the inclusion of disaggregated memory, i.e., memory that is shared across servers as part of an external memory blade [152]. This external memory is connected via fast, coherent interconnects such as Gen-Z [105]. As it is not managed by one of the CPUs' memory controllers, it is not subject to the capacity limits that currently define the upper bound of an in-memory database's size. We have shown that moving large amounts of data to slower memory tiers can be done in a way that strikes a balance between the DRAM savings and the access overhead. Depending on the performance of disaggregated memory, we anticipate similar results for this new type of hardware. Disaggregated memory is especially promising in combination with cloud-based in-memory databases: instead of sizing each server to accommodate the maximum expected workload, memory could be dynamically provisioned from a shared memory blade. Across all servers, this would even out usage peaks and result in a more efficient use of the available memory capacity.

Our research plan for making Hyrise a **self-driving database** involves the combination of different tuning options. Automatic tiering is only one of these options. Others include automatic index selection, automatic data compression, and automatic partitioning. As described in Chapter 7, these topics do not only share requirements in the areas of workload tracking and algorithmic optimization, but they also interact with each other. For example, different compression algorithms perform differently on different memory and storage tiers. Taking these multi-dimensional optimization problems into account as part of a centralized driver will be a challenging but rewarding task.

For an integration of our tiering concept with a future **persistency** layer, the main goal would be to profit from data that is already stored on non-volatile tiers. This data would not have to be recovered from logs but would be immediately available after a DBMS restart, similar to the main-on-NVM concept used by SAP HANA [8]. Special care would have to be given to the persistency of writes to umap-managed tiers, as umap currently only writes to disk as part of its eviction process [191]. We also refer to the remarks given by Graefe et al. on the correctness and performance of persisting memory-mapped data [89]. Besides using the table data for recovery, we suggest that the information from the access counters should be persisted and reused not only for making tiering decisions after a restart, but also to prioritize the recovery of previously DRAM-resident segments.

Most of our previous work focused on analytical queries [63]. For transactional workloads, a key aspect is to improve the handling of **secondary indexes**. A first step towards flexible indexes in Hyrise has been made by Weisgut [249]. He proposes an indexing concept in which a horizontally partitioned table can be partially indexed on a multi-chunk basis and in which these indexes can be updated asynchronously. These partial indexes combine a high hit rate with reduced memory and maintenance costs. For their automatic creation, they can benefit from the presented access counters. Additionally, they can also leverage our flexible memory management concept, which allows the indexes' internal data structures to be placed on arbitrary tiers. Some indexes that might not be desirable on DRAM could become desirable on NVM. We expect that some existing index selection algorithms that already calculate the cost/benefit ratio of DRAM indexes [216] can be reused to perform this calculation across multiple tiers.

8.3 Conclusion

Current in-memory databases are limited by the number and size of DRAM DIMMs that a server can physically support. In recent years, the resulting capacity has stagnated. To further accommodate growing amounts of data, IMDBs thus need to expand beyond DRAM. A promising candidate is non-volatile memory, which quadruples the amount of available memory while still providing access characteristics similar to those of DRAM. As such, NVM fills the gap between DRAM and SSDs.

To exploit this new type of hardware, we enable in-memory databases to use multiple non-DRAM tiers as additional storage in a way that is consistent with the paradigm of in-memory computing and that can be implemented in a way that is transparent to the upper layers of the DBMS as well as to the user.

We present the research DBMS Hyrise, which has been re-engineered and re-written from scratch as a contribution made during the work on this thesis. Hyrise is an open-source, columnar in-memory DBMS that allows for realistic end-to-end benchmarks while providing an infrastructure with a track record of being maintainable and extensible. We further present our implementation of automatic tiering for Hyrise with new approaches for (1) memory management, (2) access tracking, and (3) decision making. This allows us to answer our research questions as follows:

1. *How can data be stored on different memory and storage tiers in a transparent manner that is consistent with the DRAM-first approach of in-memory databases and does not negatively affect the performance when accessing data stored on DRAM?*

   We present a novel way of storing data structures on multiple memory and storage tiers. Polymorphic Memory Resources (PMR) are used as an abstraction mechanism that encapsulates the allocation and access methods of the individual tiers.

All data structures can be accessed via virtual memory addresses and existing code does not have to be modified. Accesses to these virtual addresses are transparently translated into the corresponding byte- or block-level reads and writes. As such, the DRAM-first paradigm is maintained from an engineering perspective.

We also maintain the DRAM-first paradigm in terms of performance: unlike existing approaches in which accesses require a check on whether the data is DRAM-resident, our implementation relies on the CPU's virtual memory management and does not introduce additional steps for accessing data on DRAM. When allocating or deallocating data, PMR requires an additional virtual method call. This results in a 0.6% overhead compared to a DRAM-only DBMS.

2. *How can a DBMS automatically (a) identify those parts of data that should remain on DRAM for performance reasons, and (b) migrate the remainder of the data without disrupting the continuous operation of the system?*

   The access pattern has a significant impact on the cost of reads from different tiers. This impact becomes more pronounced for lower tiers, meaning that random accesses have a disproportionately higher impact if they are performed on NVM or even SSD. As such, there is a need to track both the number of accesses and their access patterns. For part *a* of the question, we present an implementation that tracks access pattern information without introducing a measurable performance cost. This is made possible by extending the existing iterator abstraction and updating the access counters only once per iterator invocation instead of once per individual access. Accesses are also tracked on a per-segment basis, which allows us to identify skewed workloads both in the row and the column dimensions.

   The information from these counters is used by our decision making component, which translates the task of assigning segments to the different tiers into a multi-level knapsack problem. After a tiering decision has been made, we apply it by asynchronously copying data structures to their new location. To address part *b*, we coordinate the migrations with the Multi-Version Concurrency Control mechanism of Hyrise. In short, the insert-only property of an optimistic concurrency algorithm, together with our chunk concept, allows us to consistently operate on two copies of the data and thus allows concurrent queries to be executed without limitations.

   Compared to a pattern-agnostic tracking method, our knowledge of the access patterns allows us to either move an additional 9.5% of the TPC-H dataset to NVM without sacrificing any performance or to reduce the performance impact of storing 85% of the data on NVM by 8 percentage points to under 10%. By making decisions on the segment level instead of on the chunk or column levels, we can further half the performance hit of automatic tiering. In total, this allows us to move 90% of the data to NVM while increasing the query latency by only 11.6%.

With these results, we enable the growth of in-memory databases beyond the limitations of DRAM while maintaining their DRAM-first paradigm, and thus, their performance benefits. ■

List of Figures

List of Tables

List of Code Listings

Acronyms

ADR Asynchronous DRAM Refresh.

CTF Charge-Trap Flash Memory.

DAX NVM Direct Access.

DBA Database Administrator.

DCPMM Intel Optane Data Center Persistent Memory.

devdax →DAX with direct device access.

DSM Decomposition Storage Model.

eADR Extended Asynchronous DRAM Refresh.

EEPROM Electrically Erasable Programmable Read-Only Memory.

ERP Enterprise Resource Planning.

fsdax →DAX with a file-system abstraction.

IMDB In-Memory Database (Management System).

LQP Logical Query Plan.

MVCC Multi-Version Concurrency Control.

NSM N-ary Storage Model.

NVM Non-Volatile Memory.

OOO Out-of-Order Execution.

PCM Phase Change Memory.

PMR Polymorphic Memory Resources.

PQP Physical Query Plan.

SSO Small String Optimization.

WPQ Write-Pending Queue.

Bibliography

[1] D. J. Abadi, S. Madden, and M. Ferreira. Integrating Compression and Execution in Column-Oriented Database Systems. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 671–682, 2006.

[2] F. Aderholdt, M. G. Venkata, and Z. W. Parchman. SharP Unified Memory Allocator: An Intent-Based Memory Allocator for Extreme-Scale Systems. In *Euro-Par 2018: Parallel Processing - 24th International Conference on Parallel and Distributed Computing*, pages 533–545, 2018.

[3] N. Agarwal and T. F. Wenisch. Thermostat: Application-transparent Page Management for Two-tiered Main Memory. In *Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS*, pages 631–644, 2017.

[4] S. Aggarwal. STT–MRAM: High Density Persistent Memory Solution. In *Flash Memory Summit*, 2019.

[5] S. Agrawal, V. R. Narasayya, and B. Yang. Integrating Vertical and Horizontal Partitioning Into Automated Physical Database Design. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 359–370, 2004.

[6] A. Ailamaki, D. J. DeWitt, M. D. Hill, and M. Skounakis. Weaving Relations for Cache Performance. In *International Conference on Very Large Data Bases, VLDB*, pages 169–180, 2001.

[7] K. Alexiou, D. Kossmann, and P. Larson. Adaptive Range Filters for Cold Data: Avoiding Trips to Siberia. *Proceedings of the VLDB Endowment*, 6(14):1714–1725, 2013.

[8] M. Andrei, C. Lemke, G. Radestock, R. Schulze, C. Thiel, R. Blanco, A. Meghlan, M. Sharique, S. Seifert, S. Vishnoi, D. Booss, T. Peh, I. Schreter, W. Thesing, M. Wagle, and T. Willhalm. SAP HANA Adoption of Non-Volatile Memory. *Proceedings of the VLDB Endowment*, 10(12):1754–1765, 2017.

[9] J. Arulraj, J. J. Levandoski, U. F. Minhas, and P. Larson. BzTree: A High-Performance Latch-free Range Index for Non-Volatile Memory. *Proceedings of the VLDB Endowment*, 11(5):553–565, 2018.

[10] J. Arulraj and A. Pavlo. How to Build a Non-Volatile Memory Database Management System. In *ACM SIGMOD International Conference on Management of Data*, pages 1753–1758, 2017.

[11] J. Arulraj, A. Pavlo, and S. Dulloor. Let's Talk About Storage & Recovery Methods for Non-Volatile Memory Database Systems. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 707–722, 2015.

[12] J. Arulraj, A. Pavlo, and P. Menon. Bridging the Archipelago between Row-Stores and Column-Stores for Hybrid Workloads. In *ACM SIGMOD International Conference on Management of Data*, pages 583–598, 2016.

[13] J. Arulraj, A. Pavlo, and K. T. Malladi. Multi-Tier Buffer Management and Storage System Design for Non-Volatile Memory. *CoRR*, 2019. arXiv: 1901.10938.

[14] J. Arulraj, M. Perron, and A. Pavlo. Write-Behind Logging. *Proceedings of the VLDB Endowment*, 10(4):337–348, 2016.

[15] M. Asiatici and P. Ienne. DynaBurst: Dynamically Assemblying DRAM Bursts over a Multitude of Random Accesses. In *International Conference on Field Programmable Logic and Applications, FPL*, pages 254–262, 2019.

[16] M. Athanassoulis, K. S. Bøgh, and S. Idreos. Optimal Column Layout for Hybrid Workloads. *Proceedings of the VLDB Endowment*, 12(13):2393–2407, 2019.

[17] P. Atzeni, C. S. Jensen, G. Orsi, S. Ram, L. Tanca, and R. Torlone. The relational model is dead, SQL is dead, and I don't feel so good myself. *SIGMOD Record*, (2):64–68, 2013.

[18] J. Axboe. fio: Flexible I/O Tester, version 3.25. https://github.com/axboe/fio/.

[19] R. Baker. *DRAM Circuit Design: Fundamental and High-Speed Topics*. Wiley-IEEE Press, 2007.

[20] T. Bang, N. May, I. Petrov, and C. Binnig. The Tale of 1000 Cores: an Evaluation of Concurrency Control on Real(ly) Large Multi-Socket Hardware. In *International Workshop on Data Management on New Hardware, DaMoN*, 3:1–3:9, 2020.

[21] L. Barroso, M. Marty, D. Patterson, and P. Ranganathan. Attack of the Killer Microseconds. *Communications of the ACM*, 60(4):48–54, 2017.

[22] R. Bayer and E. M. McCreight. Organization and Maintenance of Large Ordered Indexes. In *Record of the ACM SIGFIDET Workshop on Data Description and Access*, pages 107–141, 1970.

[23] D. A. Beckingsale, M. McFadden, J. Dahm, R. Pankajakshan, and R. D. Hornung. Umpire: Application-Focused Management and Coordination of Complex Hierarchical Memory. *IBM Journal of Research and Development*, (3/4):00:1–00:10, 2020.

[24] I. Bhati, M. Chang, Z. Chishti, S. Lu, and B. L. Jacob. DRAM Refresh Mechanisms, Penalties, and Trade-Offs. *IEEE Transactions on Computers*, 65(1):108–121, 2016.

[25] M. Boissier and M. Jendruk. Workload-Driven and Robust Selection of Compression Schemes for Column Stores. In *International Conference on Extending Database Technology, EDBT*, pages 674–677, 2019.

[26] M. Boissier, R. Schlosser, and M. Uflacker. Hybrid Data Layouts for Tiered HTAP Databases with Pareto-Optimal Data Placements. In *International Conference on Data Engineering, ICDE*, pages 209–220, 2018.

[27] P. A. Boncz, A. Anatiotis, and S. Kläbe. JCC-H: Adding Join Crossing Correlations with Skew to TPC-H. In *Performance Evaluation and Benchmarking for the Analytics Era - TPC Technology Conference, TPCTC*, pages 103–119, 2017.

[28] P. A. Boncz, M. L. Kersten, and S. Manegold. Breaking the Memory Wall in MonetDB. *Communications of the ACM*, 51(12):77–85, 2008.

[29] P. A. Boncz, S. Manegold, and M. L. Kersten. Database Architecture Optimized for the New Bottleneck: Memory Access. In *International Conference on Very Large Data Bases, VLDB*, pages 54–65, 1999.

[30] P. A. Boncz, T. Neumann, and O. Erling. TPC-H Analyzed: Hidden Messages and Lessons Learned from an Influential Benchmark. In *Performance Characterization and Benchmarking - TPC Technology Conference, TPCTC*, pages 61–76, 2014.

[31] P. A. Boncz, T. Neumann, and V. Leis. FSST: Fast Random Access String Compression. *Proceedings of the VLDB Endowment*, 13(11):2649–2661, 2020.

[32] J. Boukhobza, S. Rubini, R. Chen, and Z. Shao. Emerging NVM: A Survey on Architectural Integration and Research Challenges. *ACM Transactions on Design Automation of Electronic Systems, TODAES*, 23(2):39:1–39:34, 2017.

[33] L. Braun, R. Marroquin, K. Tsay, and D. Kossmann. MTBase: Optimizing Cross-Tenant Database Queries. In *Proceedings of the International Conference on Extending Database Technology, EDBT*, pages 13–24, 2018.

[34] M. Brendle, N. Weber, M. Valiyev, N. May, R. Schulze, A. Böhm, G. Moerkotte, and M. Grossniklaus. Precise, Compact, and Fast Data Access Counters for Automated Physical Database Design. In *Datenbanksysteme für Business, Technologie und Web, BTW*, pages 79–100, 2021.

[35] J. B. Buckheit and D. L. Donoho. *WaveLab and Reproducible Research.* In *Wavelets and Statistics.* 1995, pages 55–81.

[36] R. P. L. Buse and W. Weimer. Learning a Metric for Code Readability. *IEEE Transactions on Software Engineering*, 36(4):546–558, 2010.

[37] C++ Standards Committee. *N4860 (The Final C++ 20 Draft).* International Organization for Standardization, 2020.

[38] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas. Apache Flink: Stream and Batch Processing in a Single Engine. *IEEE Database Engineering Bulletin*, 38(4):28–38, 2015.

[39] K. K. Chang, D. Lee, Z. Chishti, A. R. Alameldeen, C. Wilkerson, Y. Kim, and O. Mutlu. Improving DRAM Performance by Parallelizing Refreshes with Accesses. In *IEEE International Symposium on High Performance Computer Architecture, HPCA*, pages 356–367, 2014.

[40] J. Choe. Intel 3D XPoint Memory Die Removed from Intel Optane PCM (Phase Change Memory). https://www.techinsights.com/blog/intel-3d-xpoint-memory-die-removed-intel-optanetm-pcm-phase-change-memory, 2017.

[41] J. F. Claerbout and M. Karrenbach. Electronic Documents Give Reproducible Research a New Meaning. In *SEG Technical Program Expanded Abstracts*, pages 601–604. Society of Exploration Geophysicists, 1992.

[42] E. F. Codd. A Relational Model of Data for Large Shared Data Banks (Reprint). *Communications of the ACM*, 26(1):64–69, 1983, originally 1970.

[43] E. F. Codd. Relational Completeness of Data Base Sublanguages. *Research Report / RJ / IBM*, 1972.

[44] E. F. Codd. *The Relational Model for Database Management, Version 2*. Addison-Wesley, 1990.

[45] N. Cohen, M. Friedman, and J. R. Larus. Efficient Logging in Non-Volatile Memory by Exploiting Coherency Protocols. *SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA*:67:1–67:24, 2017.

[46] G. P. Copeland and S. N. Khoshafian. A Decomposition Storage Model. *SIGMOD Record*, 14(4):268–279, 1985.

[47] CPP Reference. The as-if rule. `https://en.cppreference.com/w/cpp/language/as_if`.

[48] C. Curtsinger and E. D. Berger. STABILIZER: Statistically Sound Performance Evaluation. In *Architectural Support for Programming Languages and Operating Systems, ASPLOS*, pages 219–228, 2013.

[49] J. Czyzyk, M. P. Mesnier, and J. J. Moré. The NEOS Server. *IEEE Journal on Computational Science and Engineering*, 5(3):68–75, 1998.

[50] B. Daase, L. J. Bollmeier, L. Benson, and T. Rabl. Maximizing Persistent Memory Bandwidth Utilization for OLAP Workloads. In *ACM SIGMOD International Conference on Management of Data*, SIGMOD '21, pages 339–351, 2021.

[51] D. C. Daly, L. C. Fujino, and K. C. Smith. Through the Looking Glass - 2020 Edition: Trends in Solid-State Circuits From ISSCC. *IEEE Solid-State Circuits Magazine*, 12(1):8–24, 2020.

[52] D. L. Davison and G. Graefe. Memory-Contention Responsive Hash Joins. In *International Conference on Very Large Data Bases, VLDB*, pages 379–390, 1994.

[53] J. DeBrabant, J. Arulraj, A. Pavlo, M. Stonebraker, S. B. Zdonik, and S. Dulloor. A Prolegomenon on OLTP Database Systems for Non-Volatile Memory. In *International Workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architectures, ADMS*, pages 57–63, 2014.

[54] J. DeBrabant, A. Pavlo, S. Tu, M. Stonebraker, and S. B. Zdonik. Anti-Caching: A New Approach to Database Management System Architecture. *Proceedings of the VLDB Endowment*, 6(14):1942–1953, 2013.

[55] D. J. DeWitt, R. H. Katz, F. Olken, L. D. Shapiro, M. Stonebraker, and D. A. Wood. Implementation Techniques for Main Memory Database Systems. In *ACM SIGMOD International Conference on Management of Data*, pages 1–8, 1984.

[56] C. Diaconu. Microsoft SQL Hekaton – Towards Large Scale Use of PM for In-memory Databases. In *SNIA Storage Industry Summit*, 2016.

[57] C. Diaconu, C. Freedman, E. Ismert, P. Larson, P. Mittal, R. Stonecipher, N. Verma, and M. Zwilling. Hekaton: SQL Server's Memory-Optimized OLTP Engine. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD*, pages 1243–1254, 2013.

[58] D. Dice, M. Herlihy, and A. Kogan. Fast Non-Intrusive Memory Reclamation for Highly-Concurrent Data Structures. In *ACM SIGPLAN International Symposium on Memory Management*, pages 36–45, 2016.

[59] M. Dimitroc. Redis Enterprise on Optane DC Persistent Memory, 2019. RedisConf. `https://www.youtube.com/watch?v=bvyNBPBBVFM`.

[60] G. Diubin and A. Korbut. The Average Behaviour of Greedy Algorithms for the Knapsack Problem: General Distributions. *Mathematical Methods of Operations Research*, 57:449–479, 2005.

[61] U. Drepper. What Every Programmer Should Know About Memory, 2007. `https://people.freebsd.org/~lstewart/articles/cpumemory.pdf`.

[62] M. Dreseler. Storing STL Containers on NVM. *Persistent Programming in Real Life*, 2019.

[63] M. Dreseler, M. Boissier, T. Rabl, and M. Uflacker. Quantifying TPC-H Choke Points and Their Optimizations. *Proceedings of the VLDB Endowment*, 13(8):1206–1220, 2020.

[64] M. Dreseler, T. Gasda, J. Kossmann, M. Uflacker, and H. Plattner. Adaptive Access Path Selection for Hardware-Accelerated DRAM Loads. In *Databases Theory and Applications - Australasian Database Conference, ADC*, pages 3–14, 2018.

[65] M. Dreseler, T. Kissinger, T. Djürken, E. Lübke, M. Uflacker, D. Habich, H. Plattner, and W. Lehner. Hardware-Accelerated Memory Operations on Large-Scale NUMA Systems. In *International Workshop on Accelerating Analytics and Data Management Systems Using Modern Processor and Storage Architectures, ADMS*, pages 34–41, 2017.

[66] M. Dreseler, J. Kossmann, M. Boissier, S. Klauck, M. Uflacker, and H. Plattner. Hyrise Re-engineered: An Extensible Database System for Research in Relational In-Memory Data Management. In *Advances in Database Technology - International Conference on Extending Database Technology, EDBT*, pages 313–324, 2019.

[67] M. Dreseler, J. Kossmann, J. Frohnhofen, M. Uflacker, and H. Plattner. Fused Table Scans: Combining AVX-512 and JIT to Double the Performance of Multi-Predicate Scans. In *International Conference on Data Engineering Workshops, ICDE Workshops*, pages 102–109, 2018.

[68] K. Drumm. Dynamic Random Access Memory (DRAM). Part 1: Memory Cell Arrays. `https://www.youtube.com/watch?v=I-9XWtdW_Co`.

[69] S. Dulloor, A. Roy, Z. Zhao, N. Sundaram, N. Satish, R. Sankaran, J. Jackson, and K. Schwan. Data Tiering in Heterogeneous Memory Systems. In *European Conference on Computer Systems, EuroSys 2016*, 15:1–15:16, 2016.

[70] D. Durner, V. Leis, and T. Neumann. Experimental Study of Memory Allocation for High-Performance Query Processing. In *International Workshop on Accelerating Analytics and Data Management Systems Using Modern Processor and Storage Architectures, ADMS*, pages 1–9, 2019.

[71] A. Eldawy, J. J. Levandoski, and P. Larson. Trekking Through Siberia: Managing Cold Data in a Memory-Optimized Database. *Proceedings of the VLDB Endowment*, 7(11):931–942, 2014.

[72] G. M. Essertel, R. Y. Tahboub, J. M. Decker, K. J. Brown, K. Olukotun, and T. Rompf. Flare: Optimizing Apache Spark with Native Compilation for Scale-Up Architectures and Medium-Size Data. In *USENIX Symposium on Operating Systems Design and Implementation, OSDI*, pages 799–815, 2018.

[73] J. Evans. A Scalable Concurrent malloc(3) Implementation for FreeBSD. `https://papers.freebsd.org/2006/bsdcan/evans-jemalloc/`, 2006.

[74] F. Faerber, A. Kemper, P.-Å. Larson, J. Levandoski, T. Neumann, and A. Pavlo. Main memory database systems. *Foundations and Trends in Databases*, 8(1-2):1–130, 2017.

[75] F. Färber, N. May, W. Lehner, P. Große, I. Müller, H. Rauhe, and J. Dees. The SAP HANA Database – An Architecture Overview. *IEEE Database Engineering Bulletin*, 35(1):28–33, 2012.

[76] M. Faust, D. Schwalb, J. Krüger, and H. Plattner. Fast Lookups for In-Memory Column Stores: Group-Key Indices, Lookup and Maintenance. In *International Workshop on Accelerating Analytics and Data Management Systems Using Modern Processor and Storage Architectures, ADMS*, pages 13–22, 2012.

[77] V. V. Fedorov, J. Kim, M. Qin, P. V. Gratz, and A. L. N. Reddy. Speculative Paging for Future NVM Storage. In *International Symposium on Memory Systems, MEMSYS*, pages 399–410, 2017.

[78] M. Flüggen. *Anti-Caching for Hyrise.* Master's thesis, Hasso Plattner Institute, University of Potsdam, 2020.

[79] Y. Fujisaki. Review of Emerging New Solid-State Non-Volatile Memories. *Japanese Journal of Applied Physics*, (4R):040001, 2013.

[80] F. Funke, A. Kemper, and T. Neumann. Compacting Transactional Data in Hybrid OLTP&OLAP Databases. *Proceedings of the VLDB Endowment*, 5(11):1424–1435, 2012.

[81] J. M. Garcia-López, K. Ilchenko, and O. Nazarenko. Optimization Lab Sessions: Major Features and Applications of IBM CPLEX. In *Optimization and Decision Support Systems for Supply Chains*, pages 139–150. Springer International Publishing, 2017.

[82] H. Garcia-Molina, J. D. Ullman, and J. Widom. *Database Systems - The Complete Book (Second Edition).* Pearson Education, 2009.

[83] M. Giardino, K. Doshi, and B. H. Ferri. Soft2LM: Application Guided Heterogeneous Memory Management. In *IEEE International Conference on Networking, Architecture and Storage, NAS*, pages 1–10, 2016.

[84] M. Glinz. On Non-Functional Requirements. In *IEEE International Requirements Engineering Conference, RE*, pages 21–26, 2007.

[85] F. Glover, J. Hultz, and D. Klingman. Improved Computer-Based Planning Techniques. Part II. *INFORMS Journal on Applied Analytics*, (4):12–20, 1979.

[86] J. Goldstein, R. Ramakrishnan, and U. Shaft. Compressing Relations and Indexes. In *IEEE International Conference on Data Engineering, ICDE*, pages 370–379, 1998.

[87] V. Gottemukkala and T. J. Lehman. Locking and Latching in a Memory-Resident Database System. In *International Conference on Very Large Data Bases, VLDB*, pages 533–544, 1992.

[88] P. Götze, A. van Renen, L. Lersch, V. Leis, and I. Oukid. Data Management on Non-Volatile Memory: A Perspective. *Datenbank-Spektrum*, 18(3):171–182, 2018.

[89] G. Graefe, H. Volos, H. Kimura, H. A. Kuno, J. Tucek, M. Lillibridge, and A. C. Veitch. In-Memory Performance for Big Data. *Proceedings of the VLDB Endowment*, 8(1):37–48, 2014.

[90] P. M. Grulich, S. Breß, S. Zeuch, J. Traub, J. von Bleichert, Z. Chen, T. Rabl, and V. Markl. Grizzly: Efficient Stream Processing Through Adaptive Query Compilation. In *ACM SIGMOD International Conference on Management of Data*, pages 2487–2503, 2020.

[91] M. Grund. *HYRISE: A Main Memory Hybrid Database Storage Engine.* PhD thesis, Hasso Plattner Institute, University of Potsdam, 2012.

[92] S. Gugnani, A. Kashyap, and X. Lu. Understanding the Idiosyncrasies of Real Persistent Memory. *Proceedings of the VLDB Endowment*, 14(4):626–639, 2021.

[93] P. Guo, A. Sarangan, and I. Agha. A Review of Germanium-Antimony-Telluride Phase Change Materials for Non-Volatile Memories and Optical Modulators. *Applied Sciences*, 9:530, 2019.

[94] S. Halfpap and R. Schlosser. Exploration of Dynamic Query-Based Load Balancing for Partially Replicated Database Systems with Node Failures. In *International Conference on Information and Knowledge Management, CIKM*, pages 3409–3412, 2020.

[95] P. Halpern and J. Lakos. Value Proposition: Allocator-Aware (AA) Software. Technical report P2035R0, ISO C++ Standardization Committee, 2020.

[96] S. Harizopoulos, D. J. Abadi, S. Madden, and M. Stonebraker. OLTP Through the Looking Glass, and What We Found There. In *ACM SIGMOD International Conference on Management of Data*, pages 981–992, 2008.

[97] W. E. Hart, C. D. Laird, J.-P. Watson, D. L. Woodruff, G. A. Hackebeil, B. L. Nicholson, and J. D. Siirola. *Pyomo Optimization Modeling in Python.* Springer Publishing Company, 2nd edition, 2017.

[98] A. Hassan, D. S. Nikolopoulos, and H. Vandierendonck. Fast and Energy-Efficient OLAP Data Management on Hybrid Main Memory Systems. *IEEE Transactions on Computers*, 68(11):1597–1611, 2019.

[99] B. He and Q. Luo. Cache-Oblivious Query Processing. In *Conference on Innovative Data Systems Research, CIDR*, pages 44–55, 2007.

[100] S. Héman, M. Zukowski, A. P. de Vries, and P. A. Boncz. Efficient and Flexible Information Retrieval using MonetDB/X100. In *Conference on Innovative Data Systems Research, CIDR*, pages 96–101, 2007.

[101] O. R. Hernandez, F. Song, B. M. Chapman, J. J. Dongarra, B. Mohr, S. Moore, and F. Wolf. Performance Instrumentation and Compiler Optimizations for MPI / OpenMP Applications. In *OpenMP Shared Memory Parallel Programming - International Workshops, IWOMP*, pages 267–278, 2006.

[102] M. Hershcovitch, R. Eres, and A. J. McPadden. PM Aware Storage Engine for MongoDB. In *ACM International Systems and Storage Conference, SYSTOR*, page 123, 2018.

[103] T. Hirofuchi and R. Takano. A Prompt Report on the Performance of Intel Optane DC Persistent Memory Module. *IEICE Transactions on Information and Systems*, (5):1168–1172, 2020.

[104] J. Hofmann, J. Eitzinger, and D. Fey. Execution-Cache-Memory Performance Model: Introduction and Validation. *CoRR*, 2015. arXiv: `1509.03118`.

[105] S. Hong, W. Kwon, and M. Oh. Hardware Implementation and Analysis of Gen-Z Protocol for Memory-Centric Architecture. *IEEE Access*, 8:127244–127253, 2020.

[106] B. Höppner, A. Waizy, and H. Rauhe. An Approach for Hybrid-Memory Scaling Columnar In-Memory Databases. In *International Workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architectures, ADMS*, pages 64–73, 2014.

[107] G. Huang, X. Cheng, J. Wang, Y. Wang, D. He, T. Zhang, F. Li, S. Wang, W. Cao, and Q. Li. X-Engine: An Optimized Storage Engine for Large-scale E-commerce Transaction Processing. In *ACM SIGMOD International Conference on Management of Data*, pages 651–665, 2019.

[108] K. Huppler. The Art of Building a Good Benchmark. In *Performance Evaluation and Benchmarking, TPC Technology Conference, TPCTC*, pages 18–30, 2009.

[109] S. Idreos, F. Groffen, N. Nes, S. Manegold, K. S. Mullender, and M. L. Kersten. MonetDB: Two Decades of Research in Column-oriented Database Architectures. *IEEE Data Engineering Bulletin*, 35(1):40–45, 2012.

[110] Intel. eADR: New Opportunities for Persistent Memory Applications, 2021. `https://software.intel.com/content/www/us/en/develop/articles/eadr-new-opportunities-for-persistent-memory-applications.html`.

[111] Intel. Intel Xeon Platinum 8380HL Processor Product Specifications. `https://ark.intel.com/content/www/us/en/ark/products/205684/intel-xeon-platinum-8380hl-processor-38-5m-cache-2-90-ghz.html`.

[112] J. Izraelevitz, J. Yang, L. Zhang, J. Kim, X. Liu, A. Memaripour, Y. J. Soh, Z. Wang, Y. Xu, S. R. Dulloor, J. Zhao, and S. Swanson. Basic Performance Measurements of the Intel Optane DC Persistent Memory Module. *CoRR*, 2019. arXiv: `1903.05714`.

[113] S. Jain, A. Ranjan, K. Roy, and A. Raghunathan. Computing in Memory With Spin-Transfer Torque Magnetic RAM. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 26(3):470–483, 2018.

[114] M. Jendruk. *Automatic Clustering in Hyrise*. Master's thesis, Hasso Plattner Institute, University of Potsdam, 2020.

[115] M. A. Jibril, A. Baumstark, P. Götze, and K. Sattler. JIT happens: Transactional Graph Processing in Persistent Memory meets Just-In-Time Compilation. In *International Conference on Extending Database Technology, EDBT*, pages 37–48, 2021.

[116] R. Jones, P. Maniar, R. Moazzami, P. Zurcher, J. Witowski, Y. Lii, P. Chu, and S. Gillespie. Ferroelectric Non-Volatile Memories for Low-Voltage, Low-Power Applications. *International Conference on Metallurgical Coatings and Thin Films*, 270(1):584–588, 1995.

[117] R. Kallman, H. Kimura, J. Natkins, A. Pavlo, A. Rasin, S. B. Zdonik, E. P. C. Jones, S. Madden, M. Stonebraker, Y. Zhang, J. Hugg, and D. J. Abadi. H-Store: A High-Performance, Distributed Main Memory Transaction Processing System. *Proceedings of the VLDB Endowment*, 1(2):1496–1499, 2008.

[118] L. Karnowski, M. E. Schüle, A. Kemper, and T. Neumann. Umbra as a Time Machine: Adding a Versioning Type to SQL. In *Datenbanksysteme für Business, Technologie und Web, BTW*, pages 1–10, 2021.

[119] R. Kateja, A. Pavlo, and G. R. Ganger. Vilamb: Low Overhead Asynchronous Redundancy for Direct Access NVM. *CoRR*, 2020. arXiv: `2004.09619`.

[120] A. Kemper and T. Neumann. HyPer: A hybrid OLTP&OLAP Main Memory Database System Based on Virtual Memory Snapshots. In *International Conference on Data Engineering, ICDE*, pages 195–206, 2011.

[121] T. Kersten and T. Neumann. On Another Level: How to Debug Compiling Query Engines. In *International Workshop on Testing Database Systems, DBTest*, 2:1–2:6, 2020.

[122] A. V. Khvalkovskiy, D. Apalkov, S. Watts, R. Chepulskii, R. S. Beach, A. Ong, X. Tang, A. Driskill-Smith, W. H. Butler, P. B. Visscher, D. Lottis, E. Chen, V. Nikitin, and M. Krounbi. Basic Principles of STT-MRAM Cell Operation in Memory Arrays. *Journal of Physics D: Applied Physics*, 46(7):074001, 2013.

[123] J. S. Kim, M. Patel, A. G. Yağlıkçı, H. Hassan, R. Azizi, L. Orosa, and O. Mutlu. Revisiting RowHammer: An Experimental Analysis of Modern DRAM Devices and Mitigation Techniques. In *International Symposium on Computer Architecture, ISCA*, pages 638–651, 2020.

[124] T. Kissinger, T. Kiefer, B. Schlegel, D. Habich, D. Molka, and W. Lehner. ERIS: A NUMA-Aware In-Memory Storage Engine for Analytical Workload. In *International Workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architectures, ADMS*, pages 74–85, 2014.

[125] R. Klein and W. E. Tchon. Nonvolatile Semiconductor Memory. *Microprocessing and Microprogramming*, 10(2):129–138, 1982. Microelectronics Technology - Current status and trends.

[126] D. E. Knuth. *The Art of Computer Programming, Volume I: Fundamental Algorithms.* Addison-Wesley, 1997.

[127] J. Kossmann. Self-Driving: From General Purpose to Specialized DBMSs. In *PhD Workshop co-located with the 44th International Conference on Very Large Databases, VLDB*, 2018.

[128] J. Kossmann, M. Boissier, A. Dubrawski, F. Heseding, C. Mandel, U. Pigorsch, M. Schneider, T. Schniese, M. Sobhani, P. Tsayun, K. Wille, M. Perscheid, M. Uflacker, and H. Plattner. A Cockpit for the Development and Evaluation of Autonomous Database Systems. In *IEEE International Conference on Data Engineering, ICDE*, pages 2685–2688, 2021.

[129] J. Kossmann, M. Dreseler, T. Gasda, M. Uflacker, and H. Plattner. Visual Evaluation of SQL Plan Cache Algorithms. In *Australasian Database Conference (ADC)*, pages 350–353, 2018.

[130] J. Kossmann and R. Schlosser. Self-Driving Database Systems: A Conceptual Approach. *Distributed Parallel Databases*, 38(4):795–817, 2020.

[131] G. Kotonya and I. Sommerville. *Requirements Engineering - Processes and Techniques*. John Wiley & Sons, 1998.

[132] I. Kouznetsov. Embedded 28-nm Charge-Trap NVM Technology. In *Flash Memory Summit*, 2017.

[133] J. Krüger, C. Kim, M. Grund, N. Satish, D. Schwalb, J. Chhugani, H. Plattner, P. Dubey, and A. Zeier. Fast Updates on Read-Optimized Databases Using Multi-Core CPUs. *Proceedings of the VLDB Endowment*, 5(1):61–72, 2011.

[134] E. Kultursay, M. Kandemir, A. Sivasubramaniam, and O. Mutlu. Evaluating STT-RAM as an Energy-Efficient Main Memory Alternative. In *International Symposium on Performance Analysis of Systems and Software, ISPASS*, pages 256–267, 2013.

[135] S. Kuznetsov. Towards a Native Architecture of In-NVM DBMS. In *Actual Problems of Systems and Software Engineering, APSSE*, pages 77–89, 2019.

[136] T. Kwon, M. Imran, and J. Yang. Cost-Effective Reliable MLC PCM Architecture Using Virtual Data Based Error Correction. *IEEE Access*, 8:44006–44018, 2020.

[137] A. Lacaita. Phase Change Memories: State-of-the-art, Challenges and Perspectives. *Solid-State Electronics*, 50(1):24–31, 2006. Special Issue: Papers selected from the 2005 ULIS Conference.

[138] T. Lahiri, S. Chavan, M. Colgan, D. Das, A. Ganesh, M. Gleeson, S. Hase, A. Holloway, J. Kamp, T. Lee, J. Loaiza, N. MacNaughton, V. Marwah, N. Mukherjee, A. Mullick, S. Muthulingam, V. Raja, M. Roth, E. Soylemez, and M. Zaït. Oracle Database In-Memory: A dual format in-memory database. In *International Conference on Data Engineering, ICDE*, pages 1253–1258, 2015.

[139] T. Lahiri, M. Neimat, and S. Folkman. Oracle TimesTen: An In-Memory Database for Enterprise Applications. *IEEE Database Engineering Bulletin*, 36(2):6–13, 2013.

[140] B. Lampson. *Principles for Computer System Design*. In *ACM Turing Award Lectures*. Association for Computing Machinery, 1993.

[141] A. H. Land and A. G. Doig. An Automatic Method of Solving Discrete Programming Problems. English. *Econometrica*, 28(3):497–520, 1960.

[142] P. Larson, S. Blanas, C. Diaconu, C. Freedman, J. M. Patel, and M. Zwilling. High-Performance Concurrency Control Mechanisms for Main-Memory Databases. *Proceedings of the VLDB Endowment*, 5(4):298–309, 2011.

[143] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger. Architecting Phase Change Memory as a Scalable Dram Alternative. In *International Symposium on Computer Architecture, ISCA*, pages 2–13, 2009.

[144] G. Lee, S. Shin, W. Song, T. J. Ham, J. W. Lee, and J. Jeong. Asynchronous I/O Stack: A Low-latency Kernel I/O Stack for Ultra-Low Latency SSDs. In *USENIX Annual Technical Conference, USENIX ATC)*, pages 603–616, 2019.

[145] J. Lee, H. Lee, S. Ko, K. H. Kim, M. Andrei, F. Keller, and W. Han. Asymmetric-Partition Replication for Highly Scalable Distributed Transaction Processing in Practice. *Proceedings of the VLDB Endowment*, 13(12):3112–3124, 2020.

[146] J. Lee, M. Muehle, N. May, F. Faerber, V. Sikka, H. Plattner, J. Krüger, and M. Grund. High-Performance Transaction Processing in SAP HANA. *IEEE Database Engineering Bulletin*, 36(2):28–33, 2013.

[147] V. Leis, P. A. Boncz, A. Kemper, and T. Neumann. Morsel-Driven Parallelism: A NUMA-aware Query Evaluation Framework for the Many-Core Age. In *ACM SIGMOD International Conference on Management of Data*, pages 743–754, 2014.

[148] V. Leis, M. Haubenschild, A. Kemper, and T. Neumann. LeanStore: In-Memory Data Management beyond Main Memory. In *International Conference on Data Engineering, ICDE*, pages 185–196, 2018.

[149] V. Leis, B. Radke, A. Gubichev, A. Mirchev, P. A. Boncz, A. Kemper, and T. Neumann. Query Optimization Through the Looking Glass, and What We Found Running the Join Order Benchmark. *VLDB Journal*, 27(5):643–668, 2018.

[150] L. Lersch, X. Hao, I. Oukid, T. Wang, and T. Willhalm. Evaluating Persistent Memory Range Indexes. *Proceedings of the VLDB Endowment*, 13(4):574–587, 2019.

[151] J. J. Levandoski, P. Larson, and R. Stoica. Identifying Hot and Cold Data in Main-Memory Databases. In *International Conference on Data Engineering, ICDE*, pages 26–37, 2013.

[152] K. T. Lim, Y. Turner, J. R. Santos, A. AuYoung, J. Chang, P. Ranganathan, and T. F. Wenisch. System-Level Implications of Disaggregated Memory. In *IEEE International Symposium on High Performance Computer Architecture, HPCA*, pages 189–200, 2012.

[153] J. Lindström, V. Raatikka, J. Ruuth, P. Soini, and K. Vakkila. IBM solidDB: In-Memory Database Optimized for Extreme Speed and Availability. *IEEE Data Engineering Bulletin*, 36(2):14–20, 2013.

[154] A. Löser. *Automatic Clustering in Hyrise.* Master's thesis, Hasso Plattner Institute, University of Potsdam, 2020.

[155] B. Lu, X. Hao, T. Wang, and E. Lo. Dash: Scalable Hashing on Persistent Memory. *Proceedings of the VLDB Endowment*, 13(8):1147–1161, 2020.

[156] Y. Luo, P. Jin, Q. Zhang, and B. Cheng. TLBtree: A Read/Write-Optimized Tree Index for Non-Volatile Memory. In *International Conference on Data Engineering, ICDE*, pages 1889–1894, 2021.

[157] B. J. MacLennan. *Principles of Programming Languages - Design, Evaluation, and Implementation*. Holt, Rinehart and Winston, 1987.

[158] S. Mallidi. Storing Allocator Metadata Separate from Address Range, 2019. `https://github.com/jemalloc/jemalloc/issues/1650`.

[159] J. A. Mandelman, R. H. Dennard, G. B. Bronner, J. K. DeBrosse, R. Divakaruni, Y. Li, and C. J. Raden. Challenges and Future Directions for the Scaling of Dynamic Random-Access Memory (DRAM). *IBM Journal of Research and Development*, 46(2-3):187–222, 2002.

[160] S. Manegold and I. Manolescu. Performance Evaluation in Database Research: Principles and Experience. In *International Conference on Extending Database Technology, EDBT*, page 1156, 2009.

[161] T. Mason, T. D. Doudali, M. I. Seltzer, and A. Gavrilovska. Unexpected Performance of Intel Optane DC Persistent Memory. *IEEE Computer Architecture Letters*, 19(1):55–58, 2020.

[162] N. May, W. Lehner, S. H. P., N. Maheshwari, C. Müller, S. Chowdhuri, and A. K. Goel. SAP HANA - From Relational OLAP Database to Big Data Infrastructure. In *International Conference on Extending Database Technology, EDBT*, pages 581–592, 2015.

[163] J. S. Meena, S. M. Sze, U. Chand, and T.-Y. Tseng. Overview of Emerging Nonvolatile Memory Technologies. *Nanoscale Research Letters*, 9(1):526–558, 2014.

[164] S. Meyers. *Effective Modern C++: 42 Specific Ways to Improve Your Use of C++11 and C++14*. O'Reilly Media, Inc., 1st edition, 2014.

[165] R. Micheloni, editor. *3D Flash Memories*. 2016.

[166] G. Moerkoette. *Building Query Compilers (Under Construction)*. 2020. `https://pi3.informatik.uni-mannheim.de/~moer/querycompiler.pdf`.

[167] H. Mühe, A. Kemper, and T. Neumann. Executing Long-Running Transactions in Synchronization-Free Main Memory Database Systems. In *Conference on Innovative Data Systems Research, CIDR*, 2013.

[168] S. Müller, A. Nica, L. Butzmann, S. Klauck, and H. Plattner. Using Object-Awareness to Optimize Join Processing in the SAP HANA Aggregate Cache. In *International Conference on Extending Database Technology, EDBT*, pages 557–568, 2015.

[169] D. Mulnix, A. Rudoff, K. Ananth, and V. Srinivasan. Third Generation Intel Xeon Processor Scalable Family Technical Overview. `https://software.intel.com/content/www/us/en/develop/articles/intel-xeon-processor-scalable-family-overview.html`.

[170] T. Mytkowicz, A. Diwan, M. Hauswirth, and P. F. Sweeney. Producing Wrong Data Without Doing Anything Obviously Wrong! In *International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS*, pages 265–276, 2009.

[171] T. Mytkowicz, A. Diwan, M. Hauswirth, and P. F. Sweeney. We Have It Easy, But Do We Have It Right? In *International Symposium on Parallel and Distributed Processing, IPDPS*, pages 1–7, 2008.

[172] S. Nakariyakul. Suboptimal Branch and Bound Algorithms for Feature Subset Selection: A Comparative Study. *Pattern Recognition Letters*, 45:62–70, 2014.

[173] R. O. Nambiar and M. Poess. The Making of TPC-DS. In *International Conference on Very Large Data Bases, VLDB*, pages 1049–1058, 2006.

[174] D. Narayanan and O. Hodson. Whole-System Persistence. *SIGPLAN Notices*, 47(4):401–410, 2012.

[175] L. Neiss. *Scalability Optimization for In-Memory Databases on Hundreds of Cores*. Master's thesis, Hasso Plattner Institute, University of Potsdam, 2018.

[176] T. Neumann. Engineering High-Performance Database Engines. *Proceedings of the VLDB Endowment*, 7(13):1734–1741, 2014.

[177] T. Neumann and M. J. Freitag. Umbra: A Disk-Based System with In-Memory Performance. In *Conference on Innovative Data Systems Research, CIDR*, 2020.

[178] S. Noll, J. Teubner, N. May, and A. Böhm. Analyzing Memory Accesses with Modern Processors. In *International Workshop on Data Management on New Hardware, DaMoN*, 1:1–1:9, 2020.

[179] A. O'Dwyer. *Mastering the C++17 STL: Make Full Use of the Standard Library Components in C++17*. Packt Publishing, 2017.

[180] P. E. O'Neil, E. J. O'Neil, X. Chen, and S. Revilak. The Star Schema Benchmark and Augmented Fact Table Indexing. In *Performance Evaluation and Benchmarking, TPC Technology Conference, TPCTC*, pages 237–252, 2009.

[181] L. Oden and P. Balaji. Hexe: A Toolkit for Heterogeneous Memory Management. In *IEEE International Conference on Parallel and Distributed Systems, ICPADS*, pages 656–663, 2017.

[182] M. Olma, M. Karpathiotakis, I. Alagiannis, M. Athanassoulis, and A. Ailamaki. Adaptive Partitioning and Indexing for in situ Query Processing. *VLDB Journal*, 29(1):569–591, 2020.

[183] T. Öncan. A Survey of the Generalized Assignment Problem and its Applications. *INFOR: Information Systems and Operational Research*, 45(3):123–141, 2007.

[184] I. Oukid, D. Booss, W. Lehner, P. Bumbulis, and T. Willhalm. SOFORT: a Hybrid SCM-DRAM Storage Engine for Fast Data Recovery. In *International Workshop on Data Management on New Hardware, DaMoN*, 8:1–8:7, 2014.

[185] I. Oukid, J. Lasperas, A. Nica, T. Willhalm, and W. Lehner. FPTree: A Hybrid SCM-DRAM Persistent and Concurrent B-Tree for Storage Class Memory. In *ACM SIGMOD International Conference on Management of Data*, pages 371–386, 2016.

[186] I. Oukid, W. Lehner, T. Kissinger, T. Willhalm, and P. Bumbulis. Instant Recovery for Main Memory Databases. In *Conference on Innovative Data Systems Research, CIDR*, 2015.

[187] J. M. Patel, H. Deshmukh, J. Zhu, N. Potti, Z. Zhang, M. Spehlmann, H. Memisoglu, and S. Saurabh. Quickstep: A Data Platform Based on the Scaling-Up Approach. *Proceedings of the VLDB Endowment*, 11(6):663–676, 2018.

[188] A. Pavlo, G. Angulo, J. Arulraj, H. Lin, J. Lin, L. Ma, P. Menon, T. C. Mowry, M. Perron, I. Quah, S. Santurkar, A. Tomasic, S. Toor, D. V. Aken, Z. Wang, Y. Wu, R. Xian, and T. Zhang. Self-Driving Database Management Systems. In *Conference on Innovative Data Systems Research, CIDR*, 2017.

[189] A. Pavlo. Multi-tier Anti-Caching, 2014. `https://github.com/apavlo/h-store/issues/154`.

[190] I. B. Peng, M. B. Gokhale, and E. W. Green. System Evaluation of the Intel Optane Byte-Addressable NVM. In *International Symposium on Memory Systems, MEMSYS*, pages 304–315, 2019.

[191] I. B. Peng, M. McFadden, E. W. Green, K. Iwabuchi, K. Wu, D. Li, R. Pearce, and M. B. Gokhale. UMap: Enabling Application-driven Optimizations for Page Management. In *IEEE/ACM Workshop on Memory Centric High Performance Computing, MCHPC*, pages 71–78, 2019.

[192] L. Perron. Operations Research and Constraint Programming at Google. In *Principles and Practice of Constraint Programming, CP*, page 2, 2011.

[193] L. Perron and V. Furnon. OR-Tools, version 7.2, Google. `https://developers.google.com/optimization/`.

[194] H. Plattner. *A Course in In-Memory Data Management.* Springer, 2014.

[195] H. Plattner. The Impact of Columnar In-Memory Databases on Enterprise Systems. *Proceedings of the VLDB Endowment*, 7(13):1722–1729, 2014.

[196] I. Psaroudakis, T. Scheuer, N. May, A. Sellami, and A. Ailamaki. Scaling Up Concurrent Main-Memory Column-Store Scans: Towards Adaptive NUMA-aware Data and Task Placement. *Proceedings of the VLDB Endowment*, 8(12):1442–1453, 2015.

[197] A. Pupykina and G. Agosta. Survey of Memory Management Techniques for HPC and Cloud Computing. *IEEE Access*, 7:167351–167373, 2019.

[198] M. Raasveldt, P. Holanda, T. Gubner, and H. Mühleisen. Fair Benchmarking Considered Difficult: Common Pitfalls In Database Performance Testing. In *International Workshop on Testing Database Systems, DBTest*, 2:1–2:6, 2018.

[199] M. Raasveldt and H. Mühleisen. DuckDB: an Embeddable Analytical Database. In *ACM SIGMOD International Conference on Management of Data*, pages 1981–1984, 2019.

[200] F. Ramsak, V. Markl, R. Fenk, M. Zirkel, K. Elhardt, and R. Bayer. Integrating the UB-Tree into a Database System Kernel. In *International Conference on Very Large Data Bases, VLDB*, pages 263–272, 2000.

[201] D. S. Rao, S. Kumar, A. S. Keshavamurthy, P. Lantz, D. Reddy, R. Sankaran, and J. Jackson. System Software for Persistent Memory. In *EuroSys Conference*, 15:1–15:15, 2014.

[202] J. Rao and K. A. Ross. Making B+- Trees Cache Conscious in Main Memory. In *ACM SIGMOD International Conference on Management of Data*, pages 475–486, 2000.

[203] A. G. Read. DeWitt Clauses: Can We Protect Purchasers without Hurting Microsoft. *Review of Litigation*, 25(2):387–422, 2006.

[204] D. P. Reed. *Naming and Synchronization in a Decentralized Computer System.* PhD thesis, Massachusetts Institute of Technology, 1978.

[205] K. A. Ross. Efficient Hash Probes on Modern Processors. In *International Conference on Data Engineering, ICDE*, pages 1297–1301, 2007.

[206] M. A. Roth and S. J. V. Horn. Database Compression. *SIGMOD Record*, 22(3):31–39, 1993.

[207] A. Rudoff. Developer Insights for Intel Optane Persistent Memory Programming. `https://www.intel.com/content/www/us/en/events/memory-and-storage.html?videoId=6215533858001`.

[208] A. Rudoff. Is eADR a reality now? `https://groups.google.com/g/pmem/c/Zm_u6K5KkcU?pli=1`.

[209] A. Rudoff. The Impact of the NVM Programming Model ... and Persistent Memory. *SNIA Storage Developer Conference 2013.*

[210] N. Rzepka. *Optimizing OLTP Performance in In-Memory Databases on NUMA System.* Master's thesis, Hasso Plattner Institute, University of Potsdam, 2017.

[211] SAP SE. Persistent Memory. *Administration Guide for SAP HANA Platform.* `https://help.sap.com/viewer/6b94445c94ae495c83a19646e7c3fd56/2.0.04/en-US/1f61b13e096d4ef98e62c676debf117e.html`.

[212] K. Saranti and S. Paul. *Charge-Trap-Non-volatile Memory and Focus on Flexible Flash Memory Devices.* In *Charge-Trapping Non-Volatile Memories: Volume 2 – Emerging Materials and Structures.* Springer International Publishing, 2017, pages 55–89.

[213] N. Savage. The Power of Memory. *Communications of the ACM*, 57(9):15–17, 2014.

[214] S. Scargall. *PMDK Internals: Important Algorithms and Data Structures.* In *Programming Persistent Memory: A Comprehensive Guide for Developers.* Apress, 2020, pages 313–331.

[215] S. Scargall. *Volatile Use of Persistent Memory.* In *Programming Persistent Memory: A Comprehensive Guide for Developers.* Apress, 2020, pages 155–186.

[216] R. Schlosser, J. Kossmann, and M. Boissier. Efficient Scalable Multi-Attribute Index Selection Using Recursive Strategies. In *International Conference on Data Engineering, ICDE*, pages 1238–1249, 2019.

[217] M. E. Schüle, L. Karnowski, J. Schmeißer, B. Kleiner, A. Kemper, and T. Neumann. Versioning in Main-Memory Database Systems: From MusaeusDB to TardisDB. In *International Conference on Scientific and Statistical Database Management, SSDBM*, pages 169–180, 2019.

[218] D. Schwalb, T. Berning, M. Faust, M. Dreseler, and H. Plattner. nvm_malloc: Memory Allocation for NVRAM. In *International Workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architectures, ADMS*, pages 61–72, 2015.

[219] D. Schwalb, M. Dreseler, M. Uflacker, and H. Plattner. NVC-Hashmap: A Persistent and Concurrent Hashmap For Non-Volatile Memories. In *VLDB Workshop on In-Memory Data Mangement and Analytics, IMDM*, 4:1–4:8, 2015.

[220] D. Schwalb, M. Faust, M. Dreseler, P. Flemming, and H. Plattner. Leveraging Non-Volatile Memory for Instant Restarts of In-Memory Database Systems. In *International Conference on Data Engineering, ICDE*, pages 1386–1389, 2016.

[221] D. Schwalb, M. Faust, J. Wust, M. Grund, and H. Plattner. Efficient Transaction Processing for Hyrise in Mixed Workload Environments. In *VLDB Workshop on In-Memory Data Mangement and Analytics, IMDM*, pages 112–125, 2014.

[222] D. Schwalb, J. Kossmann, M. Faust, S. Klauck, M. Uflacker, and H. Plattner. Hyrise-R: Scale-out and Hot-Standby through Lazy Master Replication for Enterprise Applications. In *VLDB Workshop on In-Memory Data Mangement and Analytics, IMDM*, 7:1–7:7, 2015.

[223] D. Schwalb, G. Kumar, M. Dreseler, Anusha S., M. Faust, A. Hohl, T. Berning, G. Makkar, H. Plattner, and P. Deshmukh. Hyrise-NV: Instant Recovery for In-Memory Databases Using Non-Volatile Memory. In *International Conference on Database Systems for Advanced Applications, DASFAA*, pages 267–282, 2016.

[224] H. Schweizer, M. Besta, and T. Hoefler. Evaluating the Cost of Atomic Operations on Modern Architectures. In *International Conference on Parallel Architectures and Compilation, PACT*, pages 445–456, 2015.

[225] R. Sherkat, C. Florendo, M. Andrei, R. Blanco, A. Dragusanu, A. Pathak, P. Khadilkar, N. Kulkarni, C. Lemke, S. Seifert, S. Iyer, S. Gottapu, R. Schulze, C. Gottipati, N. Basak, Y. Wang, V. Kandiyanallur, S. Pendap, D. Gala, R. Almeida, and P. Ghosh. Native Store Extension for SAP HANA. *Proceedings of the VLDB Endowment*, 12(12):2047–2058, 2019.

[226] R. Sherkat, C. Florendo, M. Andrei, A. K. Goel, A. Nica, P. Bumbulis, I. Schreter, G. Radestock, C. Bensberg, D. Booss, and H. Gerwens. Page As You Go: Piecewise Columnar Access In SAP HANA. In *ACM SIGMOD International Conference on Management of Data*, pages 1295–1306, 2016.

[227] S. Shiratake. Scaling and Performance Challenges of Future DRAM. In *IEEE International Memory Workshop, IMW*, pages 1–3, 2020.

[228] V. Sikka, F. Färber, W. Lehner, S. K. Cha, T. Peh, and C. Bornhövd. Efficient Transaction Processing in SAP HANA Database: the End of a Column Store Myth. In *ACM SIGMOD International Conference on Management of Data*, pages 731–742, 2012.

[229] A. Silberschatz, P. B. Galvin, and G. Gagne. *Operating System Concepts, 10th Edition*. Wiley, 2018.

[230] R. Stoica and A. Ailamaki. Enabling Efficient OS Paging for Main-Memory OLTP Databases. In *International Workshop on Data Management on New Hardware, DaMoN*, pages 1–7, 2013.

[231] M. Stonebraker. Operating System Support for Database Management. *Communications of the ACM*, 24(7):412–418, 1981.

[232] M. Stonebraker, S. Madden, D. J. Abadi, S. Harizopoulos, N. Hachem, and P. Helland. The End of an Architectural Era (It's Time for a Complete Rewrite). In *International Conference on Very Large Data Bases, VLDB*, pages 1150–1160, 2007.

[233] M. Stonebraker and A. Weisberg. The VoltDB Main Memory DBMS. *IEEE Database Engineering Bulletin*, 36(2):21–27, 2013.

[234] P. Subedi, P. E. Davis, J. J. Villalobos, I. Rodero, and M. Parashar. Using Intel Optane Devices for In-situ Data Staging in HPC Workflows. *CoRR*, 2018. arXiv: `1807.09651`.

[235] C. Sun, D. L. Moal, Q. Wang, R. Mateescu, F. Blagojevic, M. Lueker-Boden, C. Guyot, Z. Bandic, and D. Vucinic. Latency Tails of Byte-Addressable Non-Volatile Memories in Systems. In *IEEE International Memory Workshop, IMW*, pages 1–4, 2017.

[236] L. Torvalds. Re: mmap/mlock performance versus read. *Linux Kernel Mailing List*, 2000. `http://lkml.iu.edu/hypermail/linux/kernel/0004.0/0775.html`.

[237] Transaction Processing Performance Council. *TPC Benchmark C - Standard Specification.* 2010. `http://www.tpc.org/tpc_documents_current_versions/pdf/tpc-c_v5.11.0.pdf`.

[238] Transaction Processing Performance Council. *TPC Benchmark DS - Standard Specification.* 2015. `http://www.tpc.org/tpc_documents_current_versions/pdf/tpc-ds_v2.1.0.pdf`.

[239] Transaction Processing Performance Council. *TPC Benchmark H (Decision Support) - Standard Specification.* 1993. `http://www.tpc.org/tpc_documents_current_versions/pdf/tpc-h_v2.18.0.pdf`.

[240] A. van Renen, V. Leis, A. Kemper, T. Neumann, T. Hashida, K. Oe, Y. Doi, L. Harada, and M. Sato. Managing Non-Volatile Memory in Database Systems. In *ACM SIGMOD International Conference on Management of Data*, pages 1541–1555, 2018.

[241] A. van Renen, L. Vogel, V. Leis, T. Neumann, and A. Kemper. Building Blocks for Persistent Memory. *VLDB Journal*, 29(6):1223–1241, 2020.

[242] A. van Renen, L. Vogel, V. Leis, T. Neumann, and A. Kemper. Persistent Memory I/O Primitives. In *International Workshop on Data Management on New Hardware, DaMoN*, 12:1–12:7, 2019.

[243] L. Vogel, A. van Renen, S. Imamura, V. Leis, T. Neumann, and A. Kemper. Mosaic: A Budget-Conscious Storage Engine for Relational Database Systems. *Proceedings of the VLDB Endowment*, 13(11):2662–2675, 2020.

[244] A. Vogelsgesang, M. Haubenschild, J. Finis, A. Kemper, V. Leis, T. Muehlbauer, T. Neumann, and M. Then. Get Real: How Benchmarks Fail to Represent the Real World. In *Proceedings of the Workshop on Testing Database Systems, DBTest*, 1:1–1:6, 2018.

[245] D. Waddington, C. Dickey, L. Xu, T. Janssen, J. Tran, and D. Kshitij. Evaluating Intel 3D-Xpoint NVDIMM Persistent Memory in the Context of a Key-Value Store. In *IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS*, pages 202–211, 2020.

[246] C. Wang, H. Cui, T. Cao, J. Zigman, H. Volos, O. Mutlu, F. Lv, X. Feng, and G. H. Xu. Panthera: Holistic Memory Management for Big Data Processing over Hybrid Memories. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI*, pages 347–362, 2019.

[247] Z. Wang, X. Liu, J. Yang, T. Michailidis, S. Swanson, and J. Zhao. Characterizing and Modeling Non-Volatile Memory Systems. In *IEEE/ACM International Symposium on Microarchitecture, MICRO*, pages 496–508, 2020.

[248] M. Weiland and B. Homölle. Usage Scenarios for Byte-Addressable Persistent Memory in High-Performance and Data Intensive Computing. *Journal of Computer Science and Technology*, 36(1):110–122, 2021.

[249] M. Weisgut. Partial Indexes for Horizontally Partitioned Tables in In-Memory Column Stores. In *GI-Workshop Grundlagen von Datenbanken, GvDB*, 2021.

[250] S. Wen, L. Cherkasova, F. X. Lin, and X. Liu. ProfDP: A Lightweight Profiler to Guide Data Placement in Heterogeneous Memory Systems. In *International Conference on Supercomputing, ICS*, pages 263–273, 2018.

[251] S. J. White and D. J. DeWitt. A Performance Study of Alternative Object Faulting and Pointer Swizzling Strategies. In *International Conference on Very Large Data Bases, VLDB*, pages 419–431, 1992.

[252] T. Willhalm, I. Oukid, I. Müller, and F. Faerber. Vectorizing Database Column Scans with Complex Predicates. In *International Workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architectures, ADMS*, pages 1–12, 2013.

[253] T. Willhalm, N. Popovici, Y. Boshmaf, H. Plattner, A. Zeier, and J. Schaffner. SIMD-Scan: Ultra Fast in-Memory Table Scan using on-Chip Vector Processing Units. *Proceedings of the VLDB Endowment*, 2(1):385–394, 2009.

[254] C. Winter, T. Schmidt, T. Neumann, and A. Kemper. Meet Me Halfway: Split Maintenance of Continuous Views. *Proceedings of the VLDB Endowment*, 13(11):2620–2633, 2020.

[255] K. Wu, Y. Huang, and D. Li. Unimem: Runtime Data Management on Non-Volatile Memory-Based Heterogeneous Main Memory. In *International Conference for High Performance Computing, Networking, Storage and Analysis, SC*, 58:1–58:14, 2017.

[256] K. Wu, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Towards an Unwritten Contract of Intel Optane SSD. In *USENIX Workshop on Hot Topics in Storage and File Systems, HotStorage 2019*, 2019.

[257] Y. Wu, J. Arulraj, J. Lin, R. Xian, and A. Pavlo. An Empirical Evaluation of In-Memory Multi-Version Concurrency Control. *Proceedings of the VLDB Endowment*, 10(7):781–792, 2017.

[258] Y. Wu, K. Park, R. Sen, B. Kroth, and J. Do. Lessons Learned from the Early Performance Evaluation of Intel Optane DC Persistent Memory in DBMS. In *International Workshop on Data Management on New Hardware, DaMoN*, 14:1–14:3, 2020.

[259] J. Yan, Q. Jin, S. Jain, S. D. Viglas, and A. W. Lee. Snowtrail: Testing with Production Queries on a Cloud Database. In *International Workshop on Testing Database Systems, DBTest*, 4:1–4:6, 2018.

[260] J. Yang, J. Kim, M. Hoseinzadeh, J. Izraelevitz, and S. Swanson. An Empirical Guide to the Behavior and Use of Scalable Persistent Memory. In *USENIX Conference on File and Storage Technologies, FAST*, pages 169–182, 2020.

[261] J. Yang, B. Li, and D. J. Lilja. Exploring Performance Characteristics of the Optane 3D Xpoint Storage Technology. *ACM Transactions on Modeling and Performance Evaluation of Computing Systems, ToMPECS*, 5(1):4:1–4:28, 2020.

[262] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M. J. Franklin, S. Shenker, and I. Stoica. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *USENIX Symposium on Networked Systems Design and Implementation, NSDI*, pages 15–28, 2012.

[263] M. Zarubin, T. Kissinger, D. Habich, T. Willhalm, and W. Lehner. Efficient Compute Node-Local Replication Mechanisms for NVRAM-Centric Data Structures. *VLDB Journal*, 29(2-3):775–795, 2020.

[264] H. Zhang, G. Chen, B. C. Ooi, K. Tan, and M. Zhang. In-Memory Big Data Management and Processing: A Survey. *IEEE Transactions on Knowledge and Data Engineering, TKDE*, 27(7):1920–1948, 2015.

[265] S. Zhang, B. He, D. Dahlmeier, A. C. Zhou, and T. Heinze. Revisiting the Design of Data Stream Processing Systems on Multi-Core Processors. In *International Conference on Data Engineering, ICDE*, pages 659–670, 2017.

[266] Y. Zhang, J. Yang, A. Memaripour, and S. Swanson. Mojim: A Reliable and Highly-Available Non-Volatile Memory System. In *International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS*, pages 3–18, 2015.

[267] M. Zukowski, S. Héman, and P. A. Boncz. Architecture-Conscious Hashing. In *Workshop on Data Management on New Hardware, DaMoN*, pages 6–13, 2006.

Eigenständigkeitserklärung
Declaration of Authorship

Hiermit versichere ich an Eides statt, dass die vorliegende Arbeit bisher an keiner anderen Hochschule eingereicht worden ist sowie selbständig und ausschließlich mit den angegebenen Mitteln angefertigt worden ist.

Berlin, den 29. Oktober 2021

Markus Dreseler