

---

# Resource-Efficient Processing of Large Data Volumes

---

**Dissertation**

zur Erlangung des Grades eines

Doktors der Ingenieurwissenschaften

der Technischen Universität Dortmund  
an der Fakultät für Informatik

von

**Stefan Noll**

Dortmund

2021

Tag der mündlichen Prüfung: 17. Dezember 2020  
Dekan/Dekanin: Prof. Dr.-Ing. Gernot A. Fink  
Gutachter/Gutachterinnen: Prof. Dr. Jens Teubner  
Prof. Dr. Jana Giceva

*To my family and my girlfriend  
for supporting me all the way.*



# Abstract

Nowadays, nearly every enterprise, organization, or institution processes large volumes of digital information to create value from data. To process data efficiently, they employ *database management systems* that promise to deliver high throughput, low latency, predictable performance, and scalability. However, meeting all four requirements at the same time poses major challenges for systems. To make matters worse, systems are expected to meet all four requirements while utilizing *resources efficiently*. However, the complex system environment of data processing applications, i.e., the combination of complex software architectures, workloads, and hardware setups makes it very challenging to achieve high resource efficiency. In this thesis, we address resource-efficient data processing by focusing on three scenarios that are relevant—but not limited—to database management systems. Our goal is to develop solutions that improve resource efficiency at multiple system levels.

First, we address the challenge of understanding complex systems. In particular, we focus on *memory tracing*, which allows us to analyze memory access characteristics, and thus enables us to *identify* problems of inefficient memory usage. The problem is, however, that available tools for memory tracing suffer from a large runtime overhead, which makes memory tracing very expensive in practice. To address the problem, we develop an efficient implementation of memory tracing using hardware-based sampling. We demonstrate that our approach enables us to analyze the runtime characteristics of complex systems with a low runtime overhead. It reveals, e.g., access patterns, access statistics, and data and query skew for individual data structures—at byte level. Consequently, our approach opens up new possibilities for optimizing resource usage, especially memory and cache usage.

Second, we demonstrate how we can leverage information about memory access characteristics to optimize the cache usage of algorithms at hardware level. In particular, we address the problem of

*cache pollution* within a multicore processor. Cache pollution can hurt performance, especially in concurrent workloads. To address cache pollution, we apply hardware-based *cache partitioning*. We derive a cache partitioning scheme that we deliberately keep simple: We restrict memory-intensive operators that do not reuse data, such as column scans, to a small portion of the last-level cache. Furthermore, we demonstrate how to integrate cache partitioning into the execution engine of an existing database system with low engineering costs. In our evaluation we show that our approach effectively avoids cache pollution: It may improve but never degrades system performance for arbitrary workloads containing scan-intensive, cache-polluting operators.

Third, after optimizing resource usage within a multicore processor, we optimize resource usage across multiple computer systems. In particular, we address the problem of resource contention for a typical application: *bulk loading*, i.e., ingesting large volumes of data into the system. When bulk loading runs *in parallel* to query processing, both operations compete for processor cores, network bandwidth, and I/O bandwidth, which causes poor and unpredictable performance. Our analysis shows that resource contention occurs due to expensive *data transformations* during bulk loading. To address the problem of resource contention, we exploit the given hardware setup: a distributed environment consisting of the database server and one or more client machines holding the input data. We develop a distributed bulk loading mechanism, *Shared Loading*, which enables *dynamically offloading* parts of the bulk loading pipeline, i.e., deserialization and data transformation, to the client. In our evaluation we demonstrate that *Shared Loading* utilizes the available network bandwidth and the combined compute power of client and server more efficiently. It increases bulk loading throughput, improves a query workload's tail latency, and also works well with additional compression methods.

Ultimately, we claim that the contributions of this thesis have an *impact on real systems*: We implement memory tracing for the Linux kernel, we integrate our cache partitioning mechanism into (a prototype version of) a commercial database system, and we design *Shared Loading* based on the architecture of a commercial system. In our evaluations, we demonstrate that our approaches improve a database system's resource efficiency.

# Acknowledgments

Completing this thesis has been a journey—a long and energy-sapping, but also an instructive and enlightening journey. I am grateful that along the way I had the privilege to work with, meet, and be supported by many people—supervisors, colleagues, friends, and family. They were instrumental in completing this thesis. Hence, I owe them a great debt of gratitude.

First and foremost, I would like to thank my advisor *Jens Teubner*. He gave me the freedom to explore my own ideas, offered guidance when needed, and, most importantly, encouraged me to believe in myself and my work. He introduced me to the SAP HANA team and gave me the opportunity to do research in collaboration with an industry partner. The publication of my papers and this dissertation would not have been possible without him.

Next, I would like to give many thanks to *Norman May* for his constant support and for being the driving force behind the collaboration with SAP. His immense expertise and constructive feedback greatly improved my papers and this dissertation. His constant encouragement kept me developing new ideas. In addition, I am grateful to *Alexander Böhm* for his comments and our collaboration.

Working on a research project together with an industry partner has been a great experience. I would like to thank *SAP* and especially the *SAP HANA group* for funding my PhD project. In addition, I want to thank *Arne Schwarz* for facilitating the collaboration, for creating a welcoming and distraction-free working environment, and for maintaining the SAP HANA Campus.

In particular, I would like to thank my fellow PhD students—past and present—of the SAP HANA Campus: *Thomas Bach, Tiemo Bang, Michael Brendle, Jonas Dann, Matthias Hauck, Axel Hertzschuch, Lucas Lersch, Mehdi Moghaddamfar, Ismail Oukid, Georgios Psaropoulos, Robin Rehrmann, Frank Tetzl, and Florian Wolf*. I enjoyed the countless hours we

spent together—on and off work—discussing ideas, sharing knowledge, giving feedback, encouraging one another, or having fun. I am especially grateful to Thomas, Tiemo, Michael, Lucas, Ismail, Georgios, Robin and Frank for giving me feedback on an early draft of this thesis or my papers as well as to Thomas, Michael, Georgios, and Frank for being good friends and for entertaining me with numerous board game nights, hiking trips, or visits to the pool. I would also like to thank *Max Wildemann*, *Michael Rudolf*, *Elena Vasilyeva*, and *Marios Kardaras* for welcoming me at SAP.

Moreover, I would like to thank *Thomas Willhalm* and *Roman Dementiev* for giving me access to the latest Intel hardware and for discussing ideas. I am especially grateful to Roman for exchanging ideas, which kicked off my work on memory tracing.

I would also like to thank my colleagues from TU Dortmund University, *Henning Funke* and *Jan Mühlig*, for their feedback and our collaboration.

Furthermore, I would like to thank *Jana Giceva*, *Johannes Fischer*, and *Jian-Jia Chen* for taking an active part in my PhD thesis committee as jury members. I am especially grateful to Jana Giceva for taking on the role of co-examiner.

I would also like to thank my long-time friend *Alexander* for his emotional support and for giving me feedback on an early draft of this dissertation.

Last but not least, I would like to especially thank my mother *Claudia* and her husband *Michael*, my siblings *Florian* and *Lena*, as well as my girlfriend *Sarah* for their emotional support, constant encouragement, and unconditional love.

*Sandhausen, October 3, 2020*

Stefan Noll



# Contents

<b>Abstract</b>	<b>v</b>
<b>Acknowledgments</b>	<b>vii</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Resource Efficiency . . . . .	3
1.2 Motivation . . . . .	4
1.3 Challenges . . . . .	5
1.4 Contributions and Outline . . . . .	8
<b>2 Preliminaries</b>	<b>13</b>
2.1 Memory Hierarchy . . . . .	14
2.1.1 Compute Components . . . . .	15
2.1.2 Memory Components . . . . .	17
2.1.3 Experimental Analysis of Cache Performance . . . . .	19
2.2 Main-Memory Database Systems . . . . .	25
2.2.1 Order-Preserving Dictionary Compression . . . . .	27
2.2.2 Buffered Updates . . . . .	28
<b>3 Memory Tracing</b>	<b>31</b>
3.1 Introduction . . . . .	32
3.2 Background: Profiling & PEBS . . . . .	33
3.3 Implementation . . . . .	36
3.4 Use Cases . . . . .	37
3.4.1 Detecting Access Patterns . . . . .	38
3.4.2 Access Counting at Byte level . . . . .	40
3.4.3 Hot Working Set Size . . . . .	41
3.4.4 Table Partitioning . . . . .	42
3.5 Overhead . . . . .	45
3.6 Related Work . . . . .	47
3.7 Conclusion . . . . .	49

<b>4</b>	<b>CPU Cache Partitioning</b>	<b>51</b>
4.1	Introduction . . . . .	52
4.2	Query Execution in SAP HANA . . . . .	54
4.2.1	Data Structures . . . . .	54
4.2.2	Operations . . . . .	55
4.3	Analysis of LLC Usage . . . . .	56
4.3.1	Experimental Setup . . . . .	57
4.3.2	Column Scan . . . . .	59
4.3.3	Aggregation With Grouping . . . . .	60
4.3.4	Foreign Key Join . . . . .	64
4.3.5	Discussion . . . . .	65
4.4	Cache Partitioning in SAP HANA . . . . .	66
4.4.1	Cache Partitioning With CAT . . . . .	66
4.4.2	Cache Partitioning Scheme . . . . .	68
4.4.3	Integration Into SAP HANA . . . . .	69
4.5	Evaluation . . . . .	71
4.5.1	Experimental Setup . . . . .	71
4.5.2	Column Scan & Aggregation With Grouping . . . . .	72
4.5.3	Aggregation With Grouping & FK Join . . . . .	75
4.5.4	Column Scan & TPC-H Queries . . . . .	78
4.5.5	Column Scan & OLTP Query . . . . .	79
4.5.6	Discussion . . . . .	80
4.6	Related Work . . . . .	82
4.7	Conclusion . . . . .	85
<b>5</b>	<b>Shared Loading</b>	<b>87</b>
5.1	Introduction . . . . .	88
5.2	Cost Analysis of Bulk Loading Pipeline . . . . .	90
5.2.1	Overview of Loading Steps . . . . .	90
5.2.2	Costs of Loading Steps . . . . .	91
5.3	Shared Loading . . . . .	92
5.3.1	Client-Centric Loading . . . . .	93
5.3.2	Server-Centric Loading . . . . .	96
5.3.3	Dynamic Offloading . . . . .	97
5.3.4	Implementation . . . . .	98
5.4	Evaluation . . . . .	99
5.4.1	Setup . . . . .	100
5.4.2	Loading in Isolation . . . . .	102
5.4.3	Loading With Concurrent Queries . . . . .	106
5.4.4	Robustness of Parameters . . . . .	108
5.4.5	Additional Compression . . . . .	110

<i>CONTENTS</i>	xi
5.4.6 Discussion . . . . .	116
5.5 Related Work . . . . .	117
5.6 Conclusion . . . . .	120
<b>6 Conclusion</b>	<b>121</b>
6.1 Summary . . . . .	122
6.2 Future Directions . . . . .	124
6.3 Discussion . . . . .	128
<b>Bibliography</b>	<b>131</b>
<b>List of Figures</b>	<b>157</b>
<b>List of Tables</b>	<b>159</b>
<b>List of Listings</b>	<b>161</b>



# 1

## Introduction

Nowadays nearly every enterprise, organization, or institution processes digital information with the goal to create value from data. For example, companies store and retrieve data to promote products, to process orders, or to offer services. They often employ data analytics to extract business intelligence, which allows them to place targeted advertisements, to optimize production processes, or to tailor products and services to the need of their customers. Government agencies analyze data to detect tax fraud, provide weather forecasts, or control road traffic to avoid accidents. Research institutions process data to analyze the human gene [65] or to answer fundamental open questions in physics [35]. The common, major challenge is to handle *large volumes* of data.

Typical data sets consist of product information, orders, financial transactions, geographical data, or personal information. We refer to a collection of related data sets as a *database* [58]. By structuring the data, e.g., by dividing the data into tables and applying relational schemas [43], a database simplifies data processing. In particular, it allows software to manage different databases with similar methods. A software system that lets end users and application programs efficiently and conveniently create, read, update, and delete databases is called a *database management system* (DBMS) [184].

Database management systems provide the basis for performance-critical applications that rely on data processing. Examples include manufacturing, telecommunication, web services, finance, retail, scientific computing, social media, or navigation systems. A major advantage

of a DBMS is that it hides the complexity of data processing from its users. To this end, a DBMS provides a common interface to data management functionality independent of the underlying hardware.

The user typically interacts with a DBMS by expressing queries using a domain-specific language such as SQL. To answer queries, the DBMS performs the necessary work in the background: It handles concurrent accesses from multiple users, enforces constraints, provides the concept of a transaction, i.e., a single logical unit of work that encapsulates multiple database operations, guarantees secure storage, and safely recovers from failures.

At the same time, a user expects a high quality of service from the DBMS. First, the system needs to *scale* with the amount of data as the application grows and evolves over time. Second, it needs to achieve high *throughput*, i.e., to be able to process a large number of queries per second for a given workload. Third, it needs to achieve low *latency*, i.e., to return the answer to a query in a minimum of time. Finally, it needs to achieve *predictable performance*, i.e., to achieve a constant throughput or latency even at peak workload periods.

Meeting all four requirements at the same time poses a major challenge for a database system. To make matters worse, a user typically expects a DBMS to meet all four requirements while utilizing *resources efficiently*, i.e., to minimize resource consumption and to maximize resource utilization. The problem is that, due to the rising complexity of the system environment, it is very challenging for database systems to achieve high resource efficiency.

For example, in the past, advances in microprocessor technology, such as higher clock frequencies or memory speeds, directly resulted in performance improvements—without any software changes. However, today’s microprocessors are increasingly complex. Software engineers need to consider a processor’s architecture, such as multiple cores, multi-level caches, or SIMD instructions, to further improve performance. In other words, software *needs to be* resource-efficient to improve performance on modern hardware.

We address the challenge of optimizing resource efficiency in this thesis. In particular, we research and develop methods that improve resource efficiency in the context of database systems. In the following, we start by introducing the term resource efficiency. Afterwards, we motivate why it is important to achieve a high resource efficiency. Then, we describe the challenges and problems. Finally, we detail our contributions and the outline of this thesis.

## 1.1 Resource Efficiency

In this thesis we address the challenge of *resource efficiency* for data processing. We refer to resource efficiency<sup>1</sup> as the ratio of *useful work* to *resources expended*, i.e.,

$$\text{RESOURCE EFFICIENCY} = \frac{\text{USEFUL WORK}}{\text{RESOURCES EXPENDED}}. \quad (1.1)$$

In the context of database systems, an example of useful work is data and query processing. We quantify the amount of useful work by the number of queries or by the amount of data that the system processes. Typical examples of resources are time, money, energy, hardware such as computer systems, and any physical or virtual component of a computer system including processor cores, processor caches, main memory, network, secondary storage, GPU memory, programmable logic blocks of an FPGA, threads, mutexes, or locks. Resources have a limited quantity, availability, capacity, or throughput.

Equation 1.1 implies that we can increase resource efficiency in two different ways: by *minimizing resource consumption* or by *maximizing useful work*. Minimizing resource consumption to increase resource efficiency means that, e.g., a system performs the same amount of useful work with less resources. However, this approach requires that we can reduce *available* resources. Otherwise, we waste resources. For example, using fewer processors to complete a task increases resource efficiency of the resource “processor” only if we can reduce the total amount of processors to the minimum needed quantity, e.g., by choosing an appropriate configuration from a cloud provider. If we cannot reduce the number of processors, e.g., due to a given on-premise hardware setup, processors are not utilized. In other words, we waste hardware resources.

Maximizing useful work to increase resource efficiency means that, e.g., a system uses the same amount of resources but performs more useful work. This approach is feasible especially if we are unable to reduce available resources. To avoid wasting a given amount of resources, we need to *maximize resource utilization*.

---

<sup>1</sup>The general definition of *efficiency* refers to the “ratio of the effective or useful output to the total input in any system” [11].

## 1.2 Motivation

Achieving a high resource efficiency is crucial for a number of reasons. First, solving a problem with less resources also reduces *costs*. These costs include purchase costs and operational costs: A lower resource consumption makes it possible to use fewer or cheaper hardware. Using fewer or cheaper hardware, on the other hand, typically results in lower energy consumption. Especially for data centers energy costs are a key challenge [160]. To make matters worse, the rapidly growing demand for computing power [53] causes electricity costs to rise and electricity usage to increase which further aggravates the issue [49]. Moreover, consuming less resources to cut costs is not only important for an on-premise infrastructure but also for an infrastructure based on cloud services. Even though a cloud platform enables purchasing resources on-demand with a pay-as-you-go model, which makes it easy to adapt resource capacity to resource consumption, the consumer or cloud provider still needs to efficiently utilize cloud resources and to reduce resource consumption to ensure cost-effectiveness.

In addition, a high resource efficiency is crucial to avoid wasting resources. Underutilizing resources often indicates missed optimization opportunities which ultimately means losing *performance*. Similarly, a high resource efficiency is key to avoid overutilization. Overutilizing resources results in contention. This is especially true for *shared* resources when, e.g., multiple applications run on the same machine and compete for the same resources. Conflicts and contention may create choke points which result in poor *predictability* of an application's performance characteristics. In the context of database systems, predictable performance is particularly important for mission-critical systems that need to meet stringent service-level agreements. Overloaded or poorly optimized systems that are unable to answer requests in time at peak periods cause companies to lose money and customers [50, 52]. By efficiently utilizing resources which are wasted or cause conflicts otherwise, systems can increase the amount of useful work without requiring additional resources. Ultimately, efficient resource usage may enable systems to meet service-level agreements or to answer requests in time at peak periods.

Furthermore, a high resource efficiency may open up new possibilities. For example, solving a problem with less on-chip cache or memory allows hardware designers to shrink caches or memory which frees up transistors. In return, hardware designers may choose to invest



the freed-up transistors into dynamically customizable logic or even application-specific compute logic. The updated hardware design may remove bottlenecks and improve performance. Additionally, it may be an option to decrease the transistor count to reduce costs or power consumption.

Moreover, being resource efficient, i.e., requiring less resources, reduces the environmental impact. A smaller environmental impact increases sustainability and may improve an organization's reputation.

Last but not least, having an *awareness* for resource efficiency, i.e., having a clear understanding of how many resources are used, what the resource utilization is, and thus being able to judge the resource efficiency of a process, is key in many situations. Being resource-aware is important for, e.g., query optimization, query execution, scheduling, hardware provisioning and sizing, or algorithm engineering.

In summary, to cut costs, to improve performance and predictability, to open up new solutions, and to reduce the environmental impact, it is very important to achieve a high resource efficiency. Beyond achieving a high resource efficiency, being resource-aware, i.e., being aware of resource consumption and resource utilization, is important in many situations.

### 1.3 Challenges

Achieving a high resource efficiency when processing large volumes of data is very challenging. Many challenges arise from the *complexity* of the system environment, i.e., from the combination of (i) complex software architectures, (ii) complex workloads, and (iii) complex hardware setups.

First, researchers and engineers require in-depth knowledge of the software architecture. A database system, for example, has a large and complex software architecture. It consists of numerous, interconnected components, supports a wide range of different features, implements hardware-conscious, low-level optimizations, and uses fine-tuned data structures and algorithms. The challenge is that new approaches that aim to optimize resource efficiency need to be *integrated into* the architecture of an *existing system*. New approaches must match a system's interface, and they must not interfere with other mechanisms. Avoiding *interference* is especially challenging when different mechanisms compete for shared resources and requires balancing resource usage carefully.

Second, researchers and engineers require in-depth knowledge of complex workloads. Database systems, for example, need to process different types of queries such as short-running transactional queries or long-running analytical queries. At the same time, database systems need to handle query and data skew. The reason why complex workloads add additional challenges is that *resource characteristics* such as quantity, availability, capacity, or throughput may *change dynamically*. In particular, dynamic workloads have different resource requirements due to changes of, e.g., the data distribution, data access pattern, type of operations, number of concurrent users, or number of requests. Frequently changing workloads require software to be *flexible*. A main challenge is to *adapt resource allocations* on the fly without disrupting execution and without failing to meet service-level agreements.

Third, researchers and engineers require in-depth knowledge of complex hardware setups. While newer hardware generations, such as newer processor models or microarchitectures, are getting more and more powerful, they are also getting more and more complex. To maximize hardware utilization and to avoid wasting hardware resources, software needs to exploit hardware characteristics, i.e., engineers need to implement *hardware-conscious* algorithms and designs. They need to minimize *communication*, e.g., to minimize data movement and synchronization depending on the memory and compute topology, or they need to leverage specific hardware functionality, e.g., to make use of SIMD instructions and different processing units. Ignoring hardware characteristics often results in resource contention or underutilization as well as poor performance.

In summary, to develop methods for achieving high resource efficiency, researchers and engineers need to deal with the complexity of the system environment. The complexity of the system environment makes it hard to *identify* problems of inefficient resource usage. In particular, it is very challenging to *analyze* the system environment, i.e., the software architecture, workloads, and hardware setups. Without the possibility to inspect and understand the system behavior in detail, problems such as memory hotspots, poor data layouts, insufficient parallelism, or inefficient algorithms remain hidden. In addition, the complexity makes it ultimately very challenging to *solve* a problem of inefficient resource usage after it is exposed.

To achieve high resource efficiency, database architects and researchers have developed a plethora of concepts, architectures, and algorithms for data processing and data management in the context of

database systems. Examples include hardware-conscious algorithms [19, 140, 205] and cost models [126], close collaboration between DBMS and operating system [73], data placement and task scheduling strategies [72, 164], optimized usage of shared caches [111], co-processing with accelerators [96, 125], hot and cold data management [67, 116, 145], or different query execution strategies [30, 144]. However, many problems remain unsolved. In this thesis, we address resource-efficient data processing by focusing on three scenarios. Our goal is to develop solutions that improve resource efficiency at multiple system levels. We give an overview of the three scenarios and their specific challenges in the following.

**Memory Tracing.** To be able to identify problems of inefficient resource usage, we need detailed insights into the hard- and software of the system. Thus, we first address the challenge of analyzing complex systems. In particular, we focus on analyzing and understanding system behavior with *memory tracing*. Memory tracing allows analyzing memory access characteristics. Detailed information about memory accesses enable optimizations of memory usage, e.g., optimized memory provisioning, memory access patterns, or data layouts. However, available tools for memory tracing suffer from a large runtime overhead.

To address this problem, we study how we can leverage *hardware-based sampling*, a functionality provided by modern processors, to collect memory traces. In particular, we focus on the following questions: How can we leverage hardware-based sampling to implement efficient memory tracing? What are possible use cases for memory tracing using hardware-based sampling and what insights can we gain from such an approach? What are the performance characteristics and the tracing overhead when profiling complex systems, such as database systems?

**CPU Cache Partitioning.** Subsequently, we demonstrate how we can leverage information about memory access patterns to optimize the resource usage of algorithms at hardware level. In particular, we address the problem of resource contention within a multicore processor. Within a multicore processor, concurrently running tasks with different memory access patterns compete for cache capacity causing *cache pollution*. Cache pollution occurs whenever a task evicts cache lines from a shared cache that are frequently accessed by another task. Cache pollution causes cache misses and can hurt performance, especially in

concurrent workloads if multiple queries or operations compete for the last-level cache of a processor.

To address the problem of cache pollution, we study how we can leverage *hardware-based cache partitioning*, a functionality provided by modern processors, to utilize processor caches more efficiently. In particular, we focus on the following questions: To what degree suffer concurrent database workloads from cache pollution? How much processor cache is needed by individual database operations? How can we utilize hardware-based cache partitioning in the context of database systems? What are the performance characteristics of integrating cache partitioning into the system? What is the integration effort?

**Bulk Loading and Query Processing.** Finally, after optimizing resource usage within a multicore processor, we optimize resource usage across multiple machines. In particular, we address the problem of resource contention and underutilization for a typical database application, *bulk loading*, where a client loads large volumes of data into a database system running on a server. When bulk loading runs *in parallel* to query processing, both operations compete for resources causing resource contention and underutilization of the processor cores, network bandwidth, and I/O bandwidth. The result is poor and unpredictable performance of both bulk loading and query processing.

To address the problem of poor and predictable performance, we study how we can leverage the compute power of a *client* machine to avoid resource contention. We study how to exploit a database system's native compression format in a distributed environment in order to reduce load on the server machine and to utilize available network bandwidth more efficiently. In particular, we focus on the following questions: What processing steps are most expensive during bulk loading? Where can we improve resource efficiency? How can we *distribute work efficiently* between client and server machines? What are the performance characteristics of an approach that distributes work *dynamically* between client and server? How does such an approach impact the performance characteristics of query processing running in parallel to bulk loading?

## 1.4 Contributions and Outline

In this thesis we research problems of inefficient resource usage. Our goal is to develop new methods for optimizing performance and re-

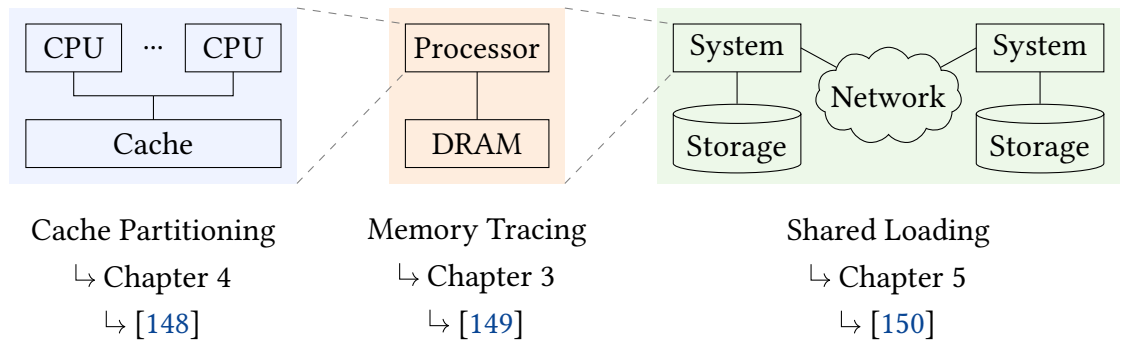


Figure 1.1: Areas of the main contributions of this thesis.

source efficiency in the context of database systems. To that end, we research resource efficiency in three scenarios that span across different levels of the system architecture. The three areas of the main contributions of this thesis are illustrated in Figure 1.1: We analyze and optimize a processor’s cache usage via CPU cache partitioning, a machine’s memory usage via efficient memory tracing, and a distributed system’s compute and network usage via efficient bulk loading.

We start off the thesis in Chapter 2 with preliminary information. We illustrate characteristics and challenges of the memory and compute hierarchy. To motivate efficient cache usage, we present experimental results that illustrate how the working set size, the memory access pattern, and resource contention impact memory latency. In addition, we introduce design and workload characteristics of main-memory database systems using the example of SAP HANA.

In Chapter 3, we present our first contribution: an efficient memory tracing implementation. We leverage a hardware mechanism of modern processors to profile the memory access characteristics of database systems with a low runtime overhead. The memory trace allows us to determine access patterns, access statistics and working set sizes of any (database) workload and of different instances of a data structure. We obtain insights that enable various optimizations of resource usage such as tailoring memory capacity to the actual working set size or partitioning data based on access statistics.

We published parts of Chapter 3 in:

- [149] Stefan Noll, Jens Teubner, Norman May, and Alexander Böhm. *Analyzing Memory Accesses with Modern Processors*. Proceedings of the 16th International Workshop on Data Management on New Hardware, 2020.

Motivated by the results of efficient memory tracing, we present our second contribution in Chapter 4: techniques for improving cache efficiency of concurrent database workloads via CPU cache partitioning. We analyze cache requirements for database operators, and we discuss and evaluate how cache partitioning can be retrofitted into an existing DBMS with low engineering costs. Our approach reduces cache pollution, i.e., resource contention of the last-level cache, improves cache efficiency, and increases throughput of concurrent query execution.

We published parts of Chapter 4 in:

- [148] Stefan Noll, Jens Teubner, Norman May, and Alexander Böhm. *Accelerating Concurrent Workloads with CPU Cache Partitioning*. Proceedings of the 34th IEEE International Conference on Data Engineering, 2018.

We change our focus on the memory and computer hierarchy in Chapter 5. Instead of optimizing resource efficiency within a single machine, we optimize resource efficiency across multiple machines as part of a distributed setup. In particular, we present our third contribution: efficient bulk loading into the optimized storage of a database system. We propose to dynamically offload data transformations to a client machine that ingests data into a DBMS running on another machine. Our approach accelerates bulk loading by using the available network bandwidth more efficiently and improves query performance by balancing the load between client and server machine efficiently.

We published parts of Chapter 5 in:

- [150] Stefan Noll, Jens Teubner, Norman May, and Alexander Böhm. *Shared Load(ing): Efficient Bulk Loading into Optimized Storage*. Proceedings of the 10th Conference on Innovative Data Systems Research, 2020.

Chapter 6 concludes the thesis.

## **Additional Work**

In the course of this thesis, the author of this thesis contributed additional work which is not included in this thesis. This includes the following work:

- [146] Stefan Noll, Henning Funke, Jens Teubner. *Energy Efficiency in Main-Memory Databases*. Datenbank-Spektrum, no. 3, 2017.

- [68] Henning Funke, Sebastian Breß, Stefan Noll, Volker Markl, Jens Teubner. *Pipelined Query Processing in Coprocessor Environments*. Proceedings of the 2018 International Conference on Management of Data, 2018.
- [147] Stefan Noll, Norman May, Alexander Böhm, Jan Mühlig, Jens Teubner. *From the Application to the CPU: Holistic Resource Management for Modern Database Management Systems*. IEEE Data Engineering Bulletin, no. 1, 2019.

### **The Author's Contributions**

In accordance with §10(2) of the doctoral regulations of the department of computer science of TU Dortmund University from August 29, 2011, we describe the author's contributions to the results and publications of this thesis in the following.

The author of this thesis is the principal author of [148, 149, 150] and of all its results used in the chapters throughout this thesis. In particular, he contributed the concepts, implementation, and evaluation of the proposed techniques and analyses. Similarly, the author of this thesis is the principal author of [146]. Furthermore, the author of this thesis co-authored [68]. He contributed minor parts of the implementation and evaluation. Finally, the author of this thesis co-authored [147], where he contributed Section 3 and Section 4.1 as well as substantial parts of the remaining sections.





# 2

## Preliminaries

In this thesis we research methods for resource-efficient data processing in the context of database systems. To study the resource efficiency of database systems, we require detailed knowledge about the system environment of database systems, i.e., detailed knowledge about the hardware setup, the software architecture, and the workloads.

We give preliminary information about the system environment in this chapter. In particular, we provide background information as well as motivation for the contributions presented in the remaining chapters of this thesis.

**Outline.** In Section 2.1, we give an overview of the hardware setup. Database systems are data-intensive by nature. Therefore, characteristics of the memory subsystem may have a significant effect on system performance. To this end, we present the memory components of computer systems and highlight their performance characteristics. In particular, we illustrate the effect of processor caches and introduce the problem of cache pollution, which we pick up again in Chapter 4. In Section 2.2, we give an overview of main-memory database systems. We highlight important characteristics of the system architecture and describe the characteristics of typical workloads. In addition, we present two concepts of main-memory database systems, order-preserving dictionary compression and buffered updates, which we frequently refer to in the remaining parts of this thesis. We explain the two concepts using the example of the commercial database system SAP HANA, which we use in the evaluation of this thesis.

## 2.1 Memory Hierarchy

One of the most important hardware resources for memory-intensive data processing applications, such as database systems, is *memory*. Patterson and Hennessy put it in a nutshell [156, p.12]: “Programmers want memory to be fast, large, and cheap, as memory speed often shapes performance, capacity limits the size of problems that can be solved, and the cost of memory today is often the majority of computer cost.” However, due to conflicting design goals, hardware architects are unable to optimize memory for maximum speed, maximum capacity, and minimal costs at the same time. Thus, hardware architects are required to make trade-offs when designing memory.

As early as 1995, Wulf and McKee [207] pointed out the increasing disparity between processor and memory speed. They observed that while processors continue to get faster and faster, memory speed does not improve at the same rate. As a result, the performance gap between processors and memory constantly widens. They concluded that if the trend continues, processing speed will eventually hit a wall—the *memory wall*—where system performance is entirely determined by memory speed. However, various advances in software and hardware technologies have mitigated the effects of the memory wall until now [83].

A central role in mitigating the disparity between processor speed and memory speed plays the *memory hierarchy* [83]. The memory hierarchy introduces multiple levels, where each level closer to the processor is typically faster, but also smaller and more expensive per byte than the adjacent level farther from the processor. In general, data moves only between two adjacent levels of the memory hierarchy. By introducing multiple levels, the memory hierarchy takes advantage of the locality of memory accesses, i.e., spatial and temporal locality. Spatial locality states that data objects whose memory addresses are near one another are likely to be accessed together in time. Temporal locality states that recently accessed data objects are likely to be accessed soon [83].

Figure 2.1 illustrates the memory hierarchy of a typical hardware setup for database systems<sup>1</sup>. The figure also visualizes how the memory hierarchy spans across different compute components. In the following, we first give an overview of the compute components. Afterwards, we describe the memory components of the memory hierarchy.

---

<sup>1</sup>The figure is based on a figure from Müller [138, p.14, Figure 3].

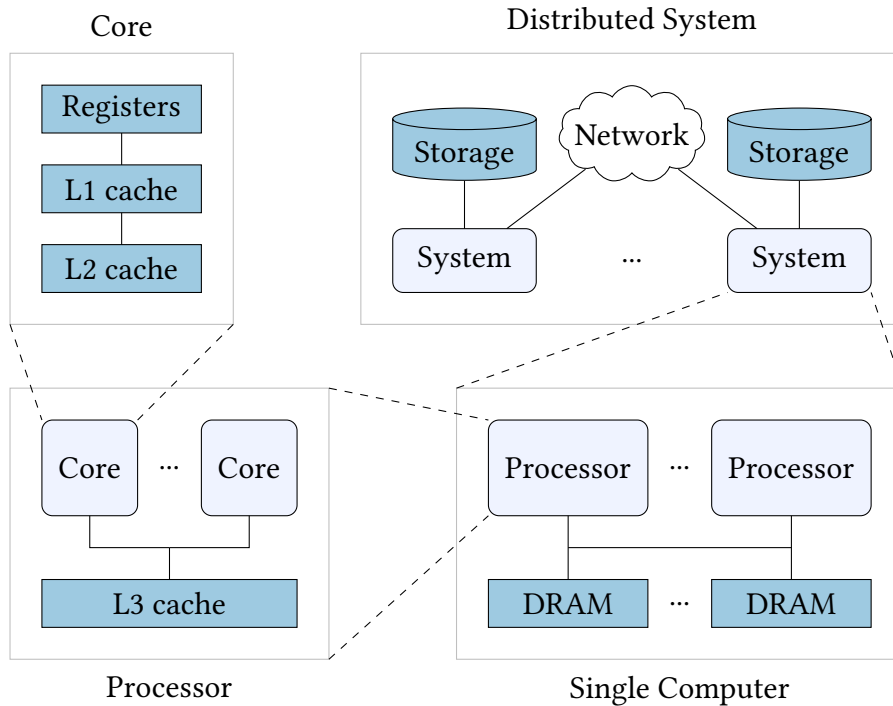


Figure 2.1: Memory hierarchy across different compute components.

### 2.1.1 Compute Components

Today’s computer architecture features a hierarchy of compute components. At the heart is the *central processing unit* (CPU), or processor, which carries out arithmetic and logical operations. One of the main indicators of a processor’s speed is the clock frequency, i.e., the number of clock cycles per second. Figure 2.2 illustrates how the clock frequencies of microprocessors evolved over time. It includes trend data of microprocessors from 1972 to 2020.

More than five decades since Gordon Moore made his famous and often cited observation about growing transistor densities, “Moore’s Law” [136] continues to prevail [53, 64]. It has become notoriously difficult, however, to turn the increasing number of transistors into further improvements in application performance. Figure 2.2 shows that while the number of transistors of processors continues to grow, the clock frequency of processors does not increase significantly anymore. In fact, the key limitation in today’s microprocessor design is the *dissipation of heat* [31]. To prevent overheating, hardware manufacturers are unable to further increase clock frequencies as faster speeds would cause more heat dissipation than chips could withstand. Similarly, to

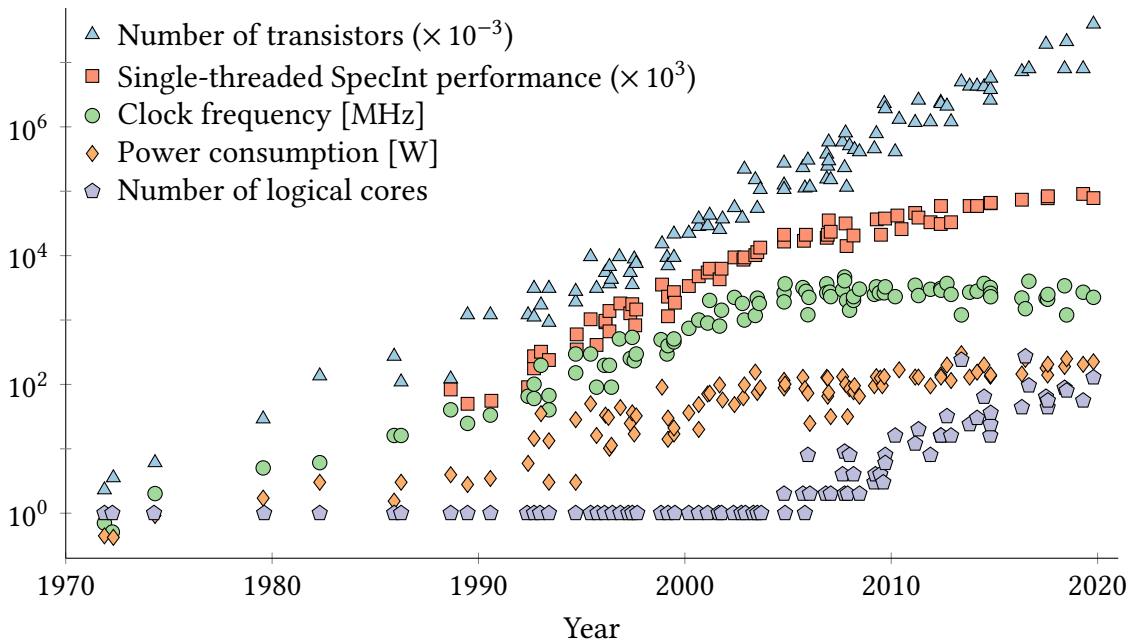


Figure 2.2: Trend data of microprocessors from 1972 to 2020 [175]. While the number of transistors of processors continues to grow, the clock frequency of processors does not increase significantly anymore. Instead, hardware manufacturers use the growing number of transistors to increase the number of logical cores of a processor.

prevent overheating, not all transistors in a modern chip can be active at the same time, an effect also called *dark silicon* [59].

In order to continue to improve throughput while staying within the power budget, hardware manufacturers build microprocessors with an increasing number of *cores* (i.e., a microprocessor consisting of multiple processors) per chip. In addition, a single core may support *simultaneous multithreading* (SMT) to execute multiple hardware threads concurrently. It allows the processor to execute instructions from more than one thread at a time which may increase the utilization of a processor's resources. For example, SMT may be used to hide the latency of memory accesses by executing useful work with one hardware thread while the other is waiting on memory requests.

The number of hardware threads varies across processor models and manufacturers. For example, as of 2020 server processors from Intel or AMD support up to 2 hardware threads [9, 91], whereas IBM Power processors support up to 8 hardware threads [40] per core. To contrast them to *physical* cores of the processor, hardware threads may also be called *logical* cores.

Component	Capacity	Random Read Latency	Read Bandwidth
Registers	$\leq 40 \cdot 16 \cdot 8 \text{ B}$	$\leq 1 \text{ ns}$	$\leq 20 \cdot 140 \text{ GB/s}$
L1 cache	$20 \cdot 32 \text{ KiB}$	$\approx 2 \text{ ns}$	$\leq 20 \cdot 140 \text{ GB/s}$
L2 cache	$20 \cdot 256 \text{ KiB}$	$\approx 6 \text{ ns}$	$\leq 20 \cdot 70 \text{ GB/s}$
L3 cache	$50 \text{ MiB}$	$\approx 15 \text{ ns}$	$\leq 20 \cdot 30 \text{ GB/s}$
DRAM	$256 \text{ GiB}$	$\approx 110 \text{ ns}$	$\approx 80 \text{ GB/s}$
SSD	$1 \text{ TiB}$	$\approx 120 \mu\text{s}$	$\approx 530 \text{ MB/s}$
1-Gbit Ethernet	—	$> 1 \mu\text{s}$	$\leq 125 \text{ MB/s}$
10-Gbit Ethernet	—	$> 1 \mu\text{s}$	$\leq 1250 \text{ MB/s}$

Table 2.1: Capacity, random read latency, and read bandwidth of different memory components for a computer system with a single Intel Xeon E7-8870 v4 processor [90] (20 physical cores, 2.1 GHz), DDR4 SDRAM (1600 MHz), a Micron M600 [194] SSD, and an 1-Gbit and a 10-Gbit network interface controller.

In addition, a *computer system* may consist of multiple multicore processors to further increase the compute power of a single computer system (cf. Figure 2.1). Making use of the different levels of hardware parallelism (SMT, multiple cores, multiple processors) is crucial to achieve high performance for data processing applications. Moreover, multiple computer systems may be connected over the network to form a *distributed system* which further increases the amount of available resources and the complexity of the system environment.

One of the most common forms of network communication uses the TCP/IP [192] protocol family in combination with the Ethernet [77] standard for the physical data transfer between systems. For a distributed system, network bandwidth and network latency are often equally or even more important than memory speed. Consequently, network optimizations may rely on *remote direct memory access* (RDMA) [21, 117, 135, 211, 216], which allows directly accessing the memory of one computer system from another computer system. RDMA reduces latency and CPU time by bypassing the network stack and not involving the operating system.

### 2.1.2 Memory Components

Today's computer architecture does not only feature a hierarchy of compute components for processing data, but it also features a hierarchy of memory components for storing data. Table 2.1 gives an overview of

the performance characteristics of different memory components. The table shows the capacity, latency, and bandwidth<sup>2</sup> of memory components for a hardware setup using an Intel Xeon E7-8870 v4 processor. We give an overview of the different memory components using the example of Table 2.1 in the following.

Closest to the compute logic inside a core of a multicore processor are the *registers*, the smallest yet fastest memory. A core has a private *first-level (L1) cache* as well as a private *second-level (L2) cache*. They feature increased capacity, but decreased speed compared to registers. These caches are only shared by the core's hardware threads. All cores of a multicore processor share the same *third-level (L3) cache*. For typical server processor of Intel or AMD, the L3 cache is the last on-chip cache<sup>3</sup>. Hence, it is also referred to as the *last-level cache (LLC)*.

The closest off-chip memory to a processor is *dynamic random access memory (DRAM)* which serves as main memory. However, processor caches and main memory are volatile—they lose all data when cut off from power. Therefore, further away off-chip memory is typically non-volatile. Non-volatile memory of a computer system may be magnetic storage, i.e., a *hard-drive disk (HDD)* consisting of several spinning magnetic disks. Other non-volatile memory may be based on flash memory, such as a *solid-state disk (SSD)*. To improve latency and bandwidth, modern SSDs transfer data over the PCI Express (PCIe) bus using the NVM Express (NVMe) interface specification [206], are combined into arrays [79], or allow fine-tuned modifications to the firmware [159] to optimize latency and throughput.

A single computer system, consisting of multiple processors, typically has distributed shared DRAM (cf. Figure 2.1) with *non-uniform memory access (NUMA)*—in contrast to *symmetric multiprocessing (SMP)* where all processors have one shared DRAM. Distributing main memory among the processors increases the memory bandwidth and reduces the memory latency if individual processors access their local DRAM modules. Otherwise, accessing remote DRAM, i.e., accessing another

---

<sup>2</sup>Note that the listed values for the bandwidth of the processor caches are only outside estimates based on official specifications [90]. Many factors impact cache bandwidth, such as conflicts due to competing loads from hardware prefetching or other cores, the complex interplay between loads and stores, (instruction-level) parallelism, or the clock frequency. An overview of memory and cache bandwidth for Intel processors is available at <https://software.intel.com/content/www/us/en/develop/articles/memory-performance-in-a-nutshell.html>.

<sup>3</sup>Other processors may have a different cache hierarchy. For example, IBM's POWER8 processor has a fourth-level cache as last-level cache [34].

processor's DRAM modules, increases latency and lowers bandwidth. Thus, memory access speed depends on the location of data in main memory. Such setups require optimizing data placements and access locality [19, 112, 164].

**Challenges.** The separation into a hierarchy of different compute and memory components, creates many challenges that software needs to address. Two of the main challenges are efficient *communication* and *resource sharing*. For a distributed system, i.e., multiple computer systems connected over a network (cf. Figure 2.1), communication is particularly important to efficiently distribute work and optimize throughput. However, communication bandwidth quickly becomes a bottleneck. In Chapter 5, we present an approach for efficient bulk loading into a database system by optimizing communication: We exploit data locality and dynamic data transformations to reduce network transfers.

For a multicore processor, efficient resource sharing is key. To motivate the efficient usage of the shared last-level cache, we perform an experimental analysis of the performance characteristics of processor caches in the following.

### 2.1.3 Experimental Analysis of Cache Performance

To process data, a processor moves data at the granularity of *cache lines* (64 B)<sup>4</sup> automatically between registers and DRAM within the cache hierarchy. To place a new cache line into a cache, the processor needs to evict an existing cache line from the cache. Processors use a particular replacement policy to decide what cache line shall be evicted. An example is the (pseudo) least-recently used replacement policy, where the processor evicts the cache line that has been least recently accessed.

A processor's caches may be inclusive by keeping copies of cache lines from the adjacent cache closer to the processor or exclusive by not keeping copies of cache lines from the adjacent cache closer to the processor. Furthermore, caches are *set associative*. A set is a group of cache lines. To fill the cache, the processor first maps a cache line to a set and then places the cache line anywhere in the set. If a set can hold  $n$  cache lines, the cache is called *n-way set associative* [83]. For example, an Intel Xeon E7-8870 v4 processor has an 8-way set associative L1

---

<sup>4</sup>While processors from Intel [90] or AMD [9] typically have a cache line size of 64 B, processors from IBM [34, 40] have a cache line size of 128 B.

cache, an 8-way set associative L2 cache, and a 20-way set associative L3 cache [90]. The number of ways not only affects the probability of conflict misses but also limits the number of partitions for a way-based implementation of cache partitioning [89] (cf. Chapter 4).

A processor performs aggressive *prefetching*: The processor's hardware prefetcher automatically loads data into the caches by predicting future memory accesses based on a software's memory access pattern. This means that an easily predictable access pattern, such as a sequential memory access pattern, typically results in many cache hits and thus greater throughput. Unpredictable (random) memory access pattern, however, typically cause cache misses and thus poor throughput.

Another issue of unpredictable access pattern is that the hardware prefetcher may mispredict accesses, and thus fills the cache with cache lines that are not needed. In doing so, the processor may evict cache lines that are needed in the near future. Consequently, memory accesses need to be analyzed and optimized in order to utilize caches efficiently. One method for analyzing the memory access pattern of an application is to collect memory traces which we study in Chapter 3.

**Access Patterns And Working Set Sizes.** To illustrate the effects of different memory access patterns and different working set sizes on cache usage and cache performance, we run an experiment with a pointer-chasing microbenchmark—similar to the work of Drepper [54] and the work of Manegold et al. [128].

The benchmark allocates an array of elements, called *links*, in contiguous memory. Each link is aligned to a cache line (64 B) and has a pointer to another link in the array. We either connect the links *sequentially* by setting each pointer to the adjacent link or connect the links *randomly* by setting the pointers based on a uniform random permutation of the array. Then, the benchmark traverses the pointer chain (multiple times) resulting in either a sequential or a random memory access pattern.

We execute the benchmark using a single thread on an Intel Xeon E7-8870 v4 processor. The benchmark performs memory accesses to local DRAM only and uses transparent huge pages [122] to minimize TLB misses. We measure elapsed time and count the number of accessed links to compute the average access latency for a single link of the pointer chain. Figure 2.3 visualizes the results.

We observe that as long as the entire working set, i.e., the pointer chain, fits into the *L1 cache* of the processor, the memory access latency



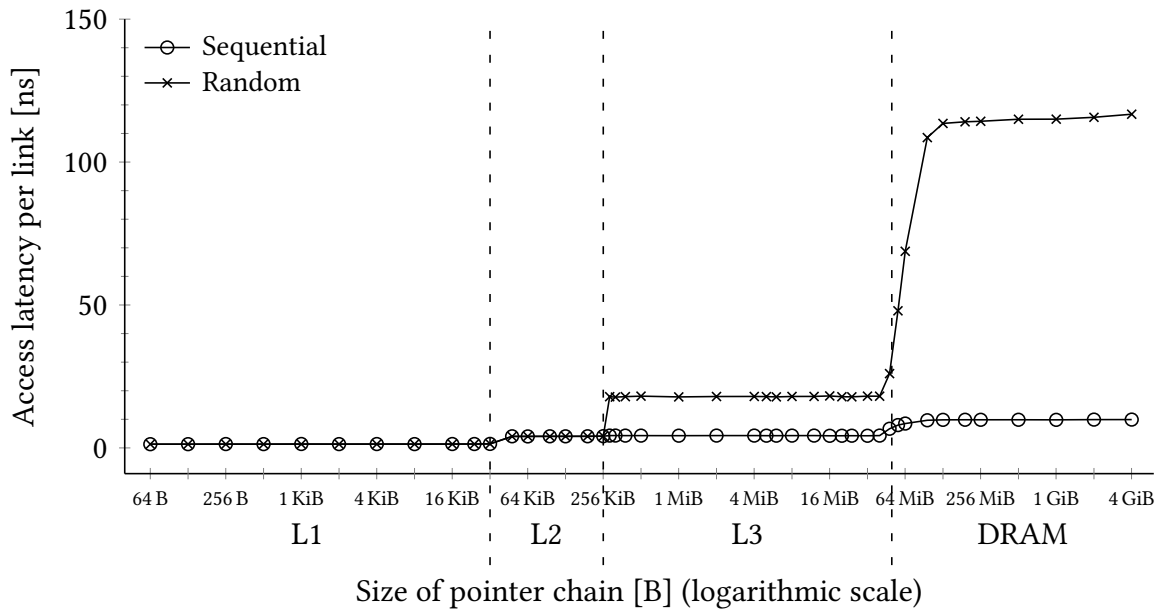


Figure 2.3: Access latency per link (64 B) during the traversal of the pointer chain. We vary the size of the chain and connect the links either sequentially or randomly resulting in a sequential or random memory access pattern during traversal. The results illustrate how the working set size and the access pattern impact cache performance.

per link is less than 2 ns for both memory access patterns. Similarly, as long as the pointer chain exceeds the L1 cache but still fits into the L2 cache, the access latency equals 4 ns for both access patterns. As soon as the pointer chain exceeds the L2 cache, however, the access pattern significantly impacts the access latency. For the L3 cache, we observe a latency of 4 ns for a sequential access and a latency of 18 ns for a random access. The difference is a factor of 4.5. For DRAM, we observe a latency of 10 ns for a sequential access and a latency of 116 ns for a random access. The difference is a factor of 11.6.

The results illustrate that optimizing algorithms to exploit sequential memory access is key to achieve high performance. However, random memory accesses cannot always be avoided, e.g., when using a hash table or data structures using pointers such as trees. In this case, it is crucial to keep the working size small such that frequently accessed data fits at least into the L3 cache. In fact, the results highlight the performance difference between accessing data from the L3 cache (18 ns) and accessing data from DRAM (116 ns): Latency increases by a factor of 6.4.

**Cache Pollution.** However, even if frequently accessed data fits completely into the L3 cache, memory access latency can still degrade. In a multicore processor, (all) cores share an L3 cache. This means that cores compete for resources, in particular, for L3 cache capacity. Whenever a core fills the shared cache with data, it may cause the eviction of cache lines that are frequently accessed by another core. Especially, if the core continuously evicts cache lines without reusing cached data, it causes *cache pollution* for other cores.

We illustrate the problem of L3 cache pollution by running the pointer-chasing microbenchmark on multiple physical cores on a single Intel Xeon E7-8870 v4 processor. On one core, we run a random traversal of the pointer chain for varying chain sizes similar to Figure 2.3. On four other cores, we run a sequential traversal of a second pointer chain with a size of 2 GiB that is shared between the four cores. We execute all traversals at the same time and for the same amount of time to potentially cause resource contention. We show the average access time per link for the single-threaded, random traversal. As a baseline, we include the results without a sequential traversal running in parallel from Figure 2.3. Figure 2.4 visualizes the results.

**Default Setup.** We observe that as long as the entire pointer chain of the random traversal fits into either the *L1 cache* or *L2 cache* of the processor, the memory access latency per link does not change compared to running the traversal in isolation. Indeed, there is no resource contention between the random traversal and any of the sequential traversals because they operate on different data sets and because a physical core has its own private L1 and L2 cache. In addition, the results demonstrate that as long as the size of the pointer chain is less than or equal to 4 MiB, i.e., the chain occupies up to 8% of the *L3 cache* (50 MiB), access latency slightly increases from 18 ns to 20 ns.

As soon as the pointer chain exceeds 4 MiB, however, the sequential traversals running in parallel significantly impact the access latency of the random traversal: Latency increases by a factor of 6.5, from 18 ns to 117 ns. In fact, accessing a link of the pointer chain takes as much time as reading data from *DRAM* (cf. Figure 2.3). Thus, most memory requests do not hit the L3 cache and require the processor to load data from main memory instead.

The random traversal and the sequential traversals compete for shared resources of the processor—especially for the L3 cache and the memory bus. The processor’s L3 cache consists of multiple slices

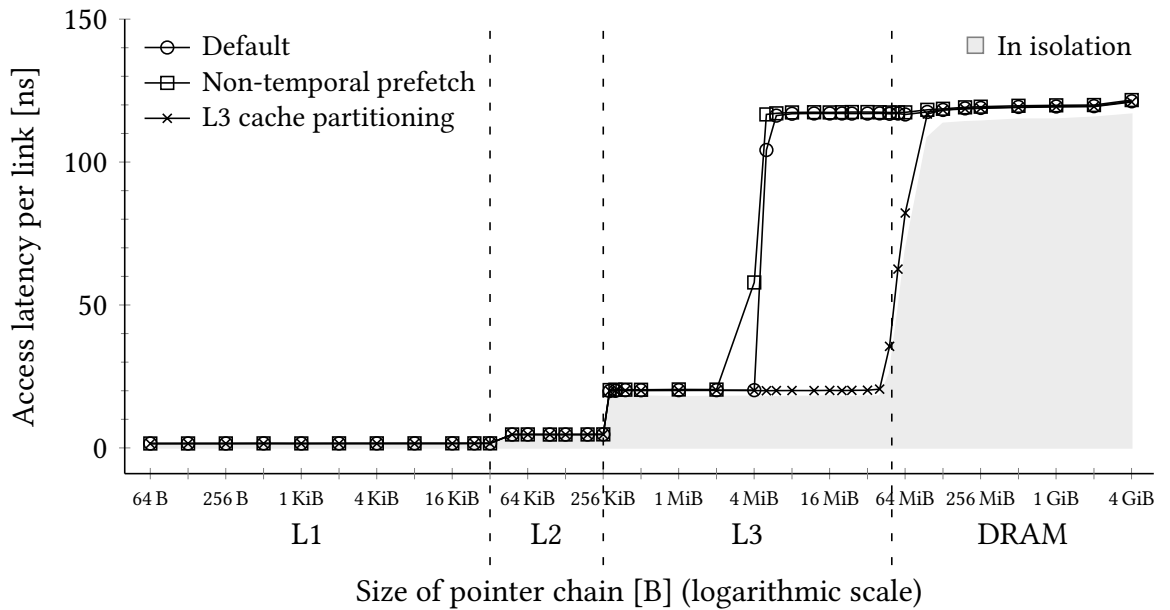


Figure 2.4: Access latency per link (64 B) during the random traversal of the pointer chain. We vary the size of the randomly connected chain and execute a sequential traversal (2 GiB) on 4 cores in parallel. We compare the default setup against using non-temporal prefetch instructions and using cache partitioning. As a baseline, we include the results without a sequential traversal running in parallel from Figure 2.3 (□).

that are connected to a ring bus—one slice per core with a size of 2.5 MiB [90, p. E-7, Figure E-3]. This may explain why access latency increases significantly for working set sizes exceeding 2 MiB: The core that executes the random traversal needs to access other L3 slices using the bus and may suffer from the memory traffic caused by the sequential traversals.

Ultimately, the results demonstrate the effects of resource contention for L3 cache capacity and illustrate L3 *cache pollution*. In the following, we evaluate two possible mitigations for L3 cache pollution: the *non-temporal prefetch* instruction and *cache partitioning*.

**Non-Temporal Prefetching.** We use the non-temporal prefetch instruction (`prefetchnta`)<sup>5</sup> [90] to give a hint to the processor that the

<sup>5</sup>The instruction set architecture of the processor provides also a non-temporal load instruction (`movntdqa`). However, this instruction is primarily used for reading from *write-combining* memory, e.g., memory-mapped I/O regions; it is not used for reading from *write-back* memory, which is available to applications [91].

4 cores executing the sequential traversal do *not reuse* data. By using the non-temporal prefetch instruction, we ask the processor to start loading the data of link  $i + n$  right before processing link  $i$ . Note that implementing prefetching is challenging [18, 81, 97, 187], e.g., due to complex access patterns.

For the sequential traversal, the access pattern is simple. The challenge is to choose the optimal prefetch distance  $n$ . If the distance is too small, we do not avoid memory stalls, or the data may already be loaded into the L3 cache by the hardware prefetcher [90]. If the distance is too big, the cache line may be evicted from the caches again before the processor needs the data.

To determine the optimal prefetch distance for this benchmark, we run an additional experiment with varying prefetch distances (not shown). The results show that prefetching the 11<sup>th</sup> next link ( $n = 11$ ) minimizes access latency of a sequential traversal.

Adding the prefetch instruction reduces the access latency of the *sequential* traversal from 11 ns to 9 ns (not shown). This demonstrates that manually optimizing prefetching increases performance compared to relying only on the hardware prefetcher of the processor. The results also reveal, however, that using the prefetch instruction with a non-temporal hint<sup>6</sup> does not improve access latency of the *random* traversal. The results are surprising because Intel’s documentation [90] clarifies that the instruction causes a cache line to be loaded into the L1 cache, skips the L2 cache, and is loaded into the L3 cache marked for a fast replacement<sup>7</sup>. Thus, we would expect that the data of the sequential traversal occupies only a small portion of the L3 cache and evicts only a small portion of the data of the random traversal. However, the results expose that the random traversal still suffers from cache pollution caused by the sequential traversal.

**Cache Partitioning.** By employing L3 cache partitioning [89], we can restrict a processor core to evict cache lines only from a subset of the L3 cache, i.e., limit write access per core. Cores can still read the entire cache. In this experiment, we allocate only 10 % of the L3 cache to the four cores executing the sequential traversal and 100 % of L3 cache to the core executing the random traversal of the pointer chain.

<sup>6</sup>We verify the use of the instruction `prefetchnta` by inspecting the program code.

<sup>7</sup>Intel’s optimization manual states that, for the non-temporal prefetch instruction, “the fill into the L3 cache or Snoop Filter may not be placed into the Most Recently Used positioned and may be chosen for replacement faster than a regular cache fill” [90, p.280, Table 9-1].

The results show that cache partitioning improves access latency by a factor of 5.8. Latency improves from 117 ns to 20 ns for data sizes exceeding the L2 but fitting into the L3 cache. In fact, we observe that with cache partitioning access latency equals the latency of the baseline, i.e., the random traversal running in isolation. This demonstrates that L3 cache partitioning can mitigate the effects of cache pollution. In this example, it negates the negative impact on performance entirely.

We evaluate the usage of L3 cache partitioning again in Chapter 4. Instead of running a microbenchmark, we analyze the impact of cache pollution on the end-to-end performance of database workloads, and we present an approach for accelerating concurrent workloads by integrating last-level cache partitioning into the execution engine of the database system.

## 2.2 Main-Memory Database Systems

**Overview.** Traditionally, database management systems stored all data on *disk*. In addition, they only had a small amount of main memory available in the past. To process data, disk-based systems use a buffer manager to copy a small subset of the data into main memory and to copy modified data back to disk [169, 184]. However, as DRAM density increased and DRAM costs decreased over time, it became eventually economical to utilize large amounts of main memory. As a result, database systems store more and more data in *main memory*.

Due to the superior latency and bandwidth of DRAM compared to disks, memory-resident systems can achieve a higher performance than only disk-based systems, especially for ad-hoc queries. An additional advantage is that memory-resident systems can avoid the overhead of a buffer manager. They avoid frequently copying data from and to disk or structuring data into (buffer) pages. As a consequence of performing fewer accesses to disks, memory-resident systems are no longer constrained by traditional bottlenecks, such as disk I/O. Instead, memory-resident systems are bound by the bandwidth and the access latency of main memory. To avoid the main-memory bottleneck, it becomes crucial to optimize memory accesses. Hence, systems need to focus on optimizing the usage of processor *caches* to avoid DRAM accesses whenever possible.

Database systems that heavily rely on main memory, i.e., primarily use main memory—not disks—as storage, are referred to as *main-memory database systems* [61]. To implement recovery mechanisms

(in order to guarantee durability [80]), main-memory database systems still need to use non-volatile memory, e.g., for writing checkpoints and a log. This may include magnetic or flash storage as well as emerging storage technologies, such as *non-volatile random access memory* (NVRAM) [155]. In addition, since DRAM is more expensive than disk storage, a memory-resident system may choose to store rarely accessed data, so-called *cold data*, on secondary storage to save costs, while keeping frequently accessed data, the *hot data*, in main memory.

**Workloads.** Database systems run various types of workloads. In the context of this thesis, two types of real-time workloads are most relevant: *online transaction processing* (OLTP) and *online analytical processing* (OLAP). An OLTP workload consists of simple, short-running queries where lots of users concurrently and frequently access or update a relatively small amount of data. Database systems typically choose a row-oriented storage layout for OLTP workloads. Examples include the processing of financial transactions in banking or the processing of orders and sales in retail. In contrast, an OLAP workload consists of complex, long-running queries where a small number of users infrequently analyzes a large volume of data. Database systems typically choose a column-oriented storage layout for OLAP workloads. Examples include the creation of regular reports, audits, and forecasts for decision support.

Due to the different workload characteristics of OLTP and OLAP, users traditionally employ two separate systems: one OLTP-optimized and one OLAP-optimized database system [188]. This makes it necessary to regularly extract, transform, and load (ETL) data from the OLTP database to the OLAP database. However, over the years a new class of database systems for *hybrid online transaction and analytical processing* (HTAP) emerged [26]. Such systems allow applications and users to concurrently execute transactional *and* analytical workloads on the *same* data copy.

**SAP HANA.** An example system supporting HTAP workloads is SAP HANA [13, 62, 102, 183], a commercial, main-memory database system developed by SAP. In this thesis, we evaluate methods for optimizing performance and resource consumption either directly on a prototype version of SAP HANA (Chapter 3 and Chapter 4) or on an independent prototype based on SAP HANA's architecture (Chapter 5). We briefly introduce two concepts of SAP HANA's architecture that are relevant

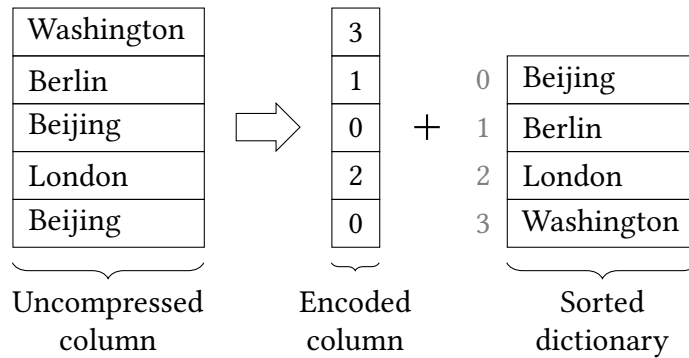


Figure 2.5: Example of applying order-preserving dictionary compression to a column.

for the following chapters: *order-preserving dictionary compression* and *buffered updates*. Note that many other main-memory database systems employ very similar approaches [14, 29, 78, 88, 104–106, 108, 109, 161, 171]. Hence, the concepts presented in this thesis apply to other systems as well. We use the example of SAP HANA in this thesis to demonstrate how our contributions impact the resource efficiency of a commercial database system.

### 2.2.1 Order-Preserving Dictionary Compression

A major problem of memory-resident database systems is the main-memory bottleneck. In particular, the problem is that the latency and bandwidth of DRAM limits the performance of memory-intensive workloads. To overcome the main-memory bottleneck, memory-resident database systems typically employ data compression. The reason is that a reduced data size not only saves memory space but memory bandwidth as well. Due to a reduced data size, the processor needs to load fewer cache lines when processing data. As a result, the system utilizes memory and cache bandwidth more efficiently which improves the performance of memory-intensive workloads.

For example, the main-memory database system SAP HANA makes heavy use of *order-preserving dictionary compression* [1, 114] in its read-optimized storage. Many other systems employ similar ideas [29, 88, 104–106, 108, 109, 161, 171]. Figure 2.5 illustrates a short example of applying order-preserving dictionary compression to a column of a table: An ordered dictionary maps domain values to a dense set of consecutive numbers. Instead of the actual value of the columns, the

engine stores the typically much smaller index of the dictionary entry. The system stores the encoded column sequentially in a vector in main memory using contiguous memory. The compression works especially well for large string columns with a low amount of distinct values. Such data is very common in real-world business applications [24, 139].

In addition, depending on the attribute type, a column or a dictionary can be further compressed using additional compression schemes. A default scheme is bit packing, where the engine stores the integers of the encoded column using the least number of bits. Additional compression schemes include run-length, prefix, cluster, indirect, or sparse encoding.

*Order-preserving* dictionary compression enables the execution engine to determine the order of values during comparison operations without decompressing data. As a result, it allows not only efficiently evaluating equality predicates but range predicates as well. For example, efficient implementations of columns scans with support for range predicates exploit *single instruction, multiple data* (SIMD) capabilities of modern processors to process multiple encoded values at once [204, 205]. The compression improves memory bandwidth utilization and the SIMD implementation improves throughput. Other operations, such as aggregations, may require decompressing data on the fly by looking up the actual value of a dictionary code in the dictionary.

While dictionary compression exploits information redundancy to reduce the memory footprint, it also creates at least one more level of indirection: Accessing the dictionary causes cache misses when the dictionaries cannot be kept in the cache of the processor. Thus, it becomes crucial to optimize the cache usage of database operations that perform many dictionary accesses, i.e., by reserving a greater portion of the available cache for this class of operations (cf. Chapter 4).

### 2.2.2 Buffered Updates

SAP HANA aims to efficiently perform write-intensive transaction processing as well as read-only analytical processing on the same data copy. The system achieves this by storing the table data of the column store in two different physical representations: a write-optimized storage and a read-optimized storage [182]. Others use similar ideas [14, 78, 104, 105, 170, 171].

Figure 2.6 illustrates the storage layout of SAP HANA's column store. The read-optimized storage uses the format described in the previous section that is optimized for low memory consumption and processing large volumes of read-only data: It uses a vector in (contiguous) main



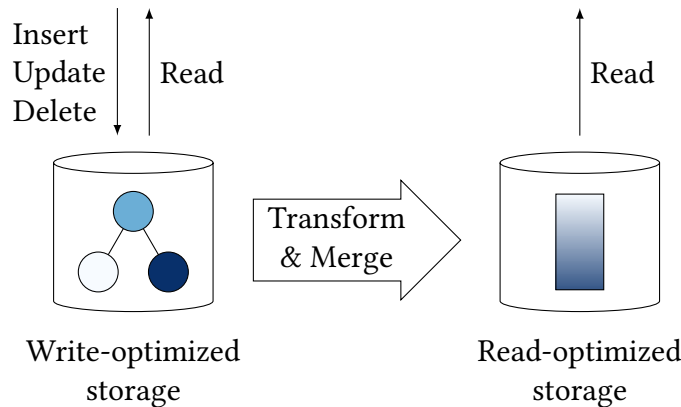


Figure 2.6: Buffered updates in SAP HANA. Records migrate from write- to read-optimized storage. The write-optimized storage supports efficient transaction processing, while the read-optimized storage supports efficient analytical processing.

memory that is compressed using order-preserving dictionary compression with additional compression schemes. The write-optimized storage uses a layout that is optimized for frequent inserts, updates, and deletes. Values are dictionary-compressed without preserving the order in the encoding by assigning dictionary codes on a first-come, first-served basis. A cache-conscious B+-tree (CSB+-tree) [172] provides a sorted view of the unsorted dictionary to accelerate accesses. In addition, transactional operations are logged.

To facilitate data ingestion into optimized storage, SAP HANA transforms new data gradually, migrating records from write- to read-optimized storage as shown in Figure 2.6. New records are first appended to a write-optimized column store. Eventually, a merge operation merges the data of the write-optimized column store with the data of the read-optimized column store. The merge operation transforms the data of the write-optimized storage from a dictionary compression without ordering to an order-preserving dictionary compression. To that end, it rebuilds the data structures of the read-optimized storage. Subsequently, it resets the write-optimized storage for new data ingestion. In addition, the operation may initiate the creation of a savepoint that is written to persistent storage. The merge operation starts either periodically as a background task or upon a user request. As the final step of the merge, the system atomically switches from the old optimized-storage to the new optimized-storage without interrupting or blocking query processing.

In this thesis, we focus on query processing on read-optimized storage (Chapter 3 and Chapter 4). However, the proposed techniques apply to write-optimized storage as well. Moreover, we demonstrate how to efficiently perform bulk loading in a distributed setup by directly writing to read-optimized storage without writing to write-optimized storage first (Chapter 5).

# 3

## Memory Tracing

Analyzing and optimizing complex system environments, i.e., the combination of complex software architectures, workloads, and hardware setups, is very challenging. Often, it is not enough to use common profiling tools because such tools identify the machine instruction rather than the instance of a data structure that causes a performance problem. This leaves a problem's root cause such as memory hotspots or poor data layouts hidden. The state-of-the-art solution is to augment classical profiling with a *memory trace*. However, current approaches for collecting memory traces are not usable in practice due to their large runtime overhead.

In this chapter, we present an approach for efficient memory tracing that utilizes hardware-based sampling. To implement hardware-based sampling, we exploit the debugging and profiling features of *commodity* processors. We evaluate our approach using a commercial and an open-source database system running the JCC-H benchmark. Moreover, we demonstrate that our approach is practical due to its low runtime overhead, and we illustrate how memory traces uncover new insights into the memory access characteristics of complex system environments using the example of database systems.

In particular, we make the following contributions: (i) we present a practical implementation of memory tracing based on Intel's PEBS mechanism; (ii) we evaluate our approach using both a commercial and an open-source database system running the JCC-H benchmark; (iii) we demonstrate and discuss practical use cases; and (iv) we analyze the runtime overhead.

**Outline.** We introduce the problem in Section 3.1. In Section 3.2, we introduce key profiling capabilities of modern processors. We present our implementation of memory tracing in Section 3.3. In Section 3.4, we evaluate the implementation and discuss practical use cases. We study the runtime overhead in Section 3.5. We cover related work in Section 3.6 and we conclude the chapter in Section 3.7.

Parts of this chapter are published in [149].

## 3.1 Introduction

Today’s database management systems are increasingly complex [199]. They offer numerous features, configuration parameters, and low-level optimizations for various hardware setups. This complexity makes it difficult to inspect and analyze system behavior and to identify new optimization and tuning opportunities during development.

To gain insights into the execution engine and data flow, engineers rely on general-purpose profiling tools such as `perf` [121] or VTune Profiler [94], or on custom profiling mechanisms implemented directly into the DBMS [179]. Common profiling tools pinpoint the machine *instruction* (and the source code line) where CPU time is spent or where hardware events such as cache misses occur. However, profiling tools often fail to identify the *instance of a data structure* that causes a problem, which makes a root cause analysis very challenging. In fact, whenever the same program code, such as `hash_table.lookup(key)`, is executed for different instances of a hash table, it may be impossible to detect the instance (e.g., hash table used in hash join of  $R$  and  $S$ ) that causes problems—even with detailed call stack information. In addition, performance problems might only occur when accessing some parts of a data structure, e.g., due to skew caused by data distribution or query predicates.

To identify the root cause of a performance problem, others [95, 103, 131, 157, 189] propose to combine profiling information with a *memory trace*, i.e., all memory addresses the system accesses during runtime. Figure 3.1 illustrates an example. By assigning profiling metrics such as CPU time or cache misses to memory addresses rather than machine instructions, we can identify specific instances or parts of a data structure that cause performance problems.

However, collecting memory traces with tools such as Valgrind [143] or Intel’s Pin [124] incurs a high overhead: They slow down the applica-

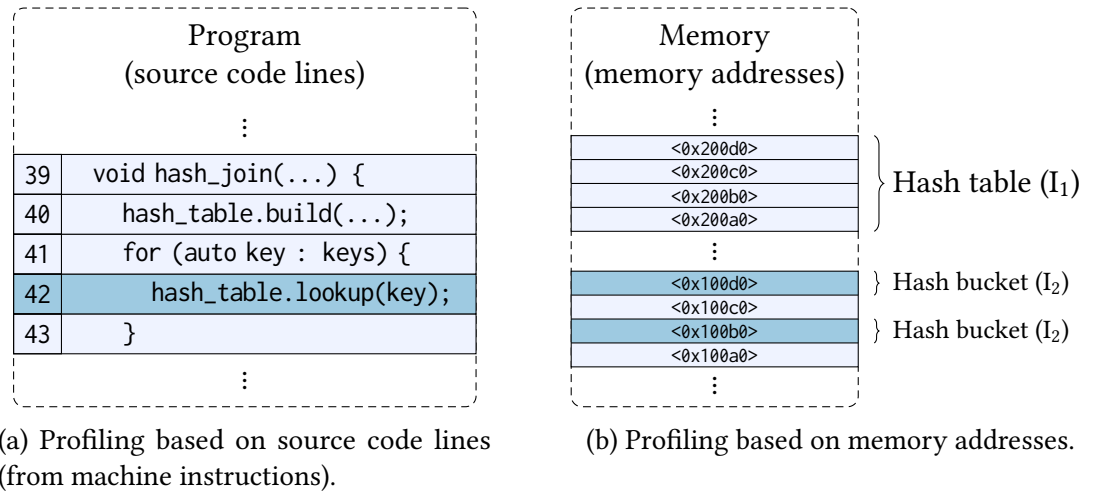


Figure 3.1: Difference between instruction-based and memory-based profiling. Mapping hardware events such as cache misses to machine instructions allows identifying *source code lines* (e.g., source code line 42) (a). In contrast, mapping hardware events to memory addresses allows identifying *instances* of a data structure (e.g., instance I<sub>2</sub>) as well as parts of a data structure at *byte level* (e.g., two hash buckets of I<sub>2</sub>) (b).

tion by more than an order of magnitude. This makes them unusable for profiling complex applications such as database systems—especially for analyzing issues that only occur in production. The good news is that modern processors feature powerful profiling capabilities via *precise event-based sampling* (PEBS) [91] that potentially allows overcoming these restrictions.

In particular, we demonstrate that collecting memory traces with PEBS on recent Intel processors is feasible in practice. We show in a comprehensive experimental evaluation that memory traces provide detailed information about how a database system accesses memory. We analyze the access frequency and the access pattern of memory accesses to reveal skew, hot data structures, or implementation and algorithmic details of the execution engine. In addition, we demonstrate that our implementation collects a memory trace with a low runtime overhead.

## 3.2 Background: Profiling & PEBS

**Profiling.** Common profiling tools such as perf [121] or VTune Profiler [94] allow analyzing detailed performance characteristics of ap-

plications. They incur almost no slowdown. This makes them usable everywhere—even in production environments. In addition to detecting where CPU time is spent, they also provide microarchitectural insights by collecting events that are exposed by modern processors via hardware performance counters [91]. These allow the user to measure cache misses, stalled cycles, memory bandwidth, non-uniform memory accesses, TLB misses, and many more events [91]. Profilers can report the machine instruction (and source code line) that was executed when an event occurred. They do *not* reveal what and how *data* was accessed (cf. Figure 3.1).

VTune Profiler takes a step into this direction. It features a profiling mode that maps certain events such as cache misses to memory objects by instrumenting memory allocations and deallocations [95]. However, VTune Profiler does not provide a memory trace, which is necessary to reveal detailed memory access statistics and access patterns.

Other tools, such as Valgrind [143] or Intel’s Pin [124], allow collecting a complete memory trace, i.e., all memory addresses the system or the application accesses during runtime. However, they incur a high runtime overhead. To trace the memory accesses of an application, the tools use *binary instrumentation*. They dynamically inject additional machine instructions into the binary, i.e., into the existing compiled program code, of an application. For example, for every load instruction the tools will add a new function call or a sequence of new instructions that take the memory address used by the load instruction and write it to a file or memory buffer. However, by adding a function call to every simple load instruction, these tools add *many expensive* instructions to the binary of an application. The added instructions slow down the application by more than an order of magnitude, which makes tools that use binary instrumentation unsuitable for profiling complex applications such as database systems.

**Precise Event-Based Sampling.** We use a processor’s hardware performance counters with support for *precise event-based sampling* (PEBS)<sup>1</sup> [91] to implement efficient memory tracing. PEBS is available

---

<sup>1</sup>Intel calls the mechanism “precise” because it reduces or entirely removes the timing error of hardware performance counters without PEBS: When the processor captures an hardware event, it may report a timestamp (or program counter) that is different from the actual time (or program counter) the event occurred [91]. The wrong timing information associated with an event is called “skid”.

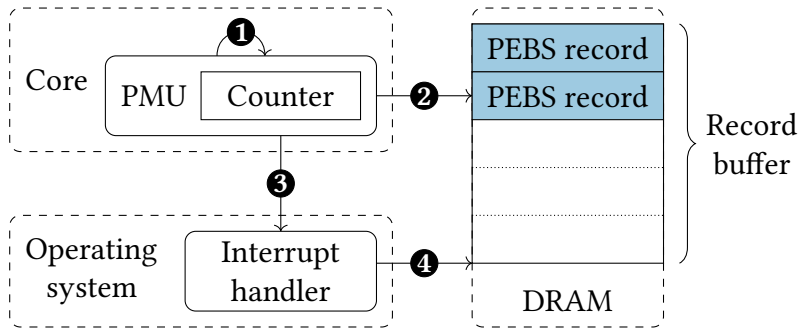


Figure 3.2: Overview of the PEBS mechanism. The core’s PMU counts an event such as L2 cache misses **1**. It writes a record to DRAM when the “Counter” reaches a threshold **2**. It sends an interrupt when the buffer is full **3**. Then, the interrupt handler of the operating system drains the buffer **4** and processes the records.

on modern Intel<sup>2</sup> processors and currently supports a subset of the events such as L1, L2, L3 cache misses or cache hits, all memory reads, or all memory stores<sup>3</sup>. PEBS enables writing debug and profiling information associated with an event to a memory resident buffer. In addition to a precise instruction pointer, this information includes, e.g., copies of general-purpose registers, latency information or the accessed data address. Figure 3.2 illustrates the mechanism.

The operating system configures the *performance monitoring unit* (PMU) of a processor’s core to count an event such as an L2 cache miss. It specifies the sampling rate by setting a threshold and creates a buffer for PEBS records in memory. Then, the PMU counts the specified event and, when the counter reaches the threshold, the hardware automatically writes a record with debug and profiling information to the buffer. When the buffer is full, the PMU sends an interrupt. The interrupt triggers the interrupt handler of the operating system. The interrupt handler drains the buffer and processes the records. Afterwards, it resets the counter and the PMU starts counting again. The advantage is that the *hardware* writes the information to memory without involving the operating system. When the operating system is needed, the *buffer mechanism* amortizes the cost of executing the interrupt handler.

<sup>2</sup>Other manufactures may implement similar functionality. AMD, for example, offers a similar feature called “lightweight profiling” [9].

<sup>3</sup>Future generations of Intel processors are expected to offer more hardware performance counters with support for PEBS and a configurable record format to allow smaller record sizes [91].

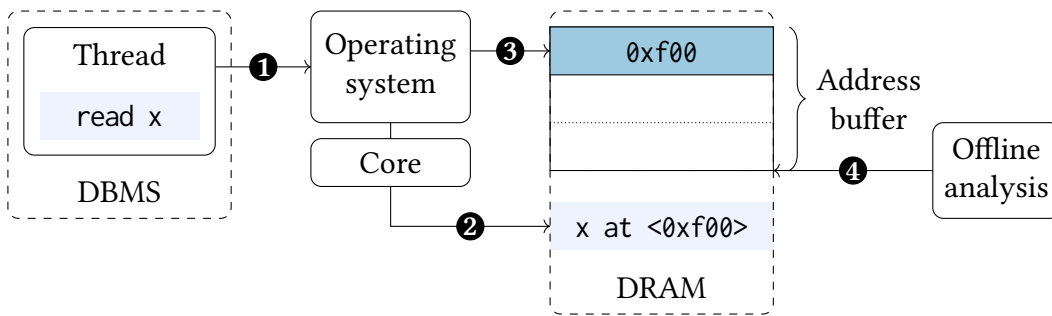


Figure 3.3: Our memory tracing implementation. When a thread accesses a data structure ❶, we may collect the virtual memory address of the accessed data ❷ using PEBS (cf. Figure 3.2) and store the address in a per-core buffer in DRAM ❸. We analyze the addresses offline ❹.

### 3.3 Implementation

To minimize the runtime overhead of memory tracing, we leverage the PEBS mechanism of modern processors (cf. Section 3.2) in our implementation of memory tracing. The implementation of memory tracing based on the PEBS mechanism is the first contribution of this chapter.

The PEBS mechanism allows us to sample the memory address associated with a specific hardware event. In the evaluation presented in this chapter, we focus on the event `mem_load_uops_retired.all_loads`, which occurs whenever the CPU *reads* data from memory [91]. This includes both cache hits and cache misses to the L1, L2 and last-level cache. Note that depending on the use case, we could use other events to collect memory addresses, e.g., associated with all memory *writes*, only last-level *cache misses*, or cache misses where the cache line was modified by another core (possibly indicating *false sharing* [131]).

Figure 3.3 gives an overview of our approach. We assume that the PEBS mechanism of the PMU is already configured to write a record every  $n$ -th occurrence of the monitored event. For each logical core, we create a buffer for storing the address samples. When a worker thread of the DBMS accesses memory, the core’s PMU may write a record with the event’s debug information to the record buffer (cf. Figure 3.2). When the record buffer is full, the PMU triggers the operating system which executes a custom interrupt handler to process the collected records. It extracts only the field with the virtual memory address associated with the event and stores the address into the buffer of the logical core. After running a workload, we analyze the address data.



To enable memory tracing in performance-critical environments, we implement memory tracing by modifying the Linux kernel (version 5.1) and by adding a custom kernel module<sup>4</sup>. Our implementation has ~1000 lines of code. In particular, we leverage the extensive, tested functionality of the perf subsystem [202] of the Linux kernel to program a core's PMU, to set up PEBS, to register an interrupt handler, and to configure the scope of profiling, e.g., to trace memory accesses only for user-space or kernel-space code, or to trace memory accesses only for particular processes or threads. Moreover, we can start and configure memory tracing using the perf tool from user space.

To improve scalability, we modify the interrupt handler to place the sampled memory addresses into per-core buffers instead of the global ring buffer used by the perf subsystem. The kernel module acts as an interface for managing the buffers from user space. In particular, the kernel module enables us read the memory addresses from the buffers, to delete the buffers, or to decrease or increase buffer sizes.

Modifying the perf subsystem is necessary in order to reduce the runtime overhead of collecting memory addresses with a high sample frequency. That is because the perf subsystem collects extensive metadata for a single sample. Processing this metadata wastes memory and compute time. In contrast, our implementation is very efficient: It extracts and stores only the addresses needed for memory tracing.

## 3.4 Use Cases

To demonstrate practical uses cases for database systems, we evaluate our memory tracing implementation with state-of-the-art benchmarks and database systems running queries single-threaded as well as multi-threaded. The evaluation of our implementation and the demonstration of practical uses cases are the second and third contribution of this chapter. Note that we expect more use cases than the ones presented, e.g., when focusing on other parts of the system such as intermediate results or when using other hardware events.

**Experimental Setup.** We use the JCC-H benchmark [28], an extension of the TPC-H benchmark [197] with skewed data and query predicates, with a scale factor of 10. We execute custom queries as well

---

<sup>4</sup>The source code is available at <http://dbis.cs.tu-dortmund.de> and at <https://github.com/stefannoll/mat>.

```

SELECT o_totalprice, o_orderdate, o_shippriority
FROM orders WHERE o_orderstatus = '0'
ORDER BY o_totalprice;

```

Listing 3.1: SQL query executed with DuckDB on the JCC-H data set.

as a complete workload with 200 random queries of the JCC-H benchmark (excluding Q21). We run SAP HANA, a commercial, main-memory database management system (cf. Section 2.2). SAP HANA makes heavy use of order-preserving dictionary compression (cf. Section 2.2.1). In addition, we run experiments with DuckDB [167], an open-source, embedded, analytical database system. We execute queries with DuckDB by using its Python interface. Our test machine has two Intel Xeon E5-2670 v3 processors and 256 GB of main memory. While DuckDB executes queries with a single thread, SAP HANA executes queries with multiple threads using up to 48 logical cores.

**Processing of Memory Traces.** Due to the sampling mechanism we do not trace every memory load, especially if data is accessed only once during profiling. To compensate for missed accesses due to sampling and to improve visualization, we *group addresses into buckets* of a fixed size. We denote the bucket size in each figure. This means that, for each memory address, we report a data access of, e.g., 4 KiB instead of 8 bytes<sup>5</sup>. We assign an address to a bucket of size, e.g., 4 KiB by ignoring the least significant  $\log_2(4096) - 1 = 11$  bits of the address. We explain the visualization of a memory trace using the example of Figure 3.4 in Section 3.4.1.

### 3.4.1 Detecting Access Patterns

To illustrate how memory traces reveal access patterns or other implementation and algorithmic details of the execution engine, we analyze the execution of a custom query with DuckDB. The SQL statement is shown in Listing 3.1. Figure 3.4 shows the memory trace. The memory trace visualizes how DuckDB accesses memory over time. The x-axis shows the samples ordered by their sampling time. The y-axis represents the virtual memory address of the samples sorted by address

<sup>5</sup>The actual data size depends on the load instruction associated with the sampled memory address: e.g., 8 B for 64-bit or 4 B for 32-bit operations.

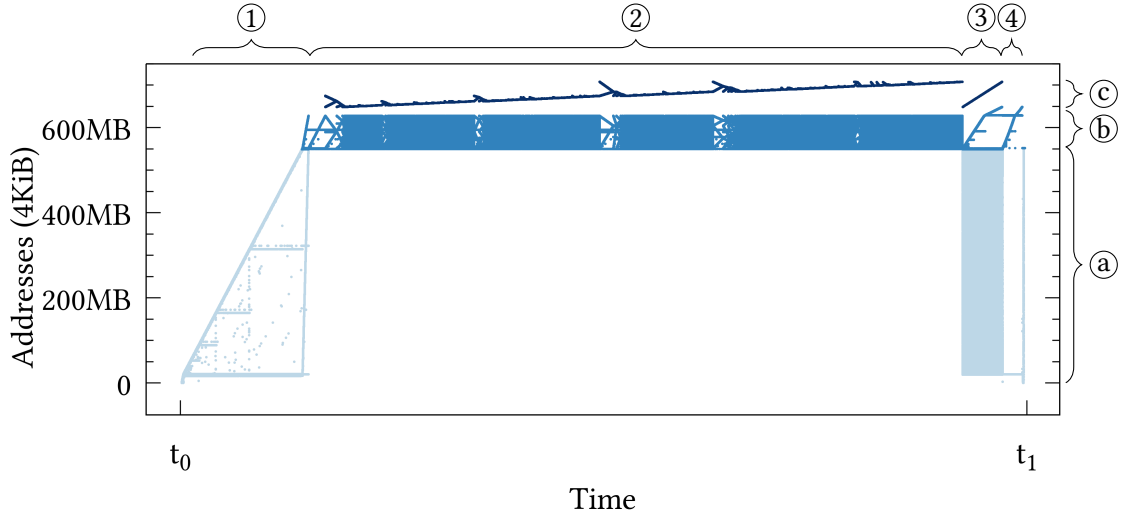


Figure 3.4: The trace illustrates the access patterns of DuckDB’s execution engine. In particular, it reveals how the operators for filtering ①, sorting ②, materializing ③, and assembling the result ④ access important data structures: the table  $\blacksquare$  (a), the filtered column  $\blacksquare$  (b), and the position list  $\blacksquare$  (c).

in ascending order. We illustrate the size of the accessed memory. In particular, we visualize each sampled address as an access to a bucket of size 4 KiB.

Figure 3.4 shows that scanning the table (a) while applying the filter predicate ① reads memory *sequentially*. Afterwards, DuckDB sorts ② the data. The trace reveals that the sort operator accesses one data structure *sequentially* and the other *randomly*. Note that DuckDB’s implementation uses the quicksort algorithm and that it sorts a position list (c) instead of sorting the data in-place. To compare a position  $p$ , it accesses the filtered column (b) indirectly by fetching the value with `column[p]`, which may be increasingly random as the order of the position list changes. The trace illustrates both access patterns: The quicksort algorithm splits the position list recursively and traverses each sublist from the start and the end simultaneously; for the comparison, it indirectly accesses the column. In the next phase, DuckDB materializes ③ the projected columns. It reads the sorted position list *sequentially* and accesses the table *randomly*. Finally, the Python interface of DuckDB transforms the result ④ into Python data structures.

Note that the memory trace allows us to break down the individual phases/operators of the query execution. We can infer the different

```
SELECT AVG(l_extendedprice) FROM lineitem;
```

Listing 3.2: SQL query executed with SAP HANA on the JCC-H data set.

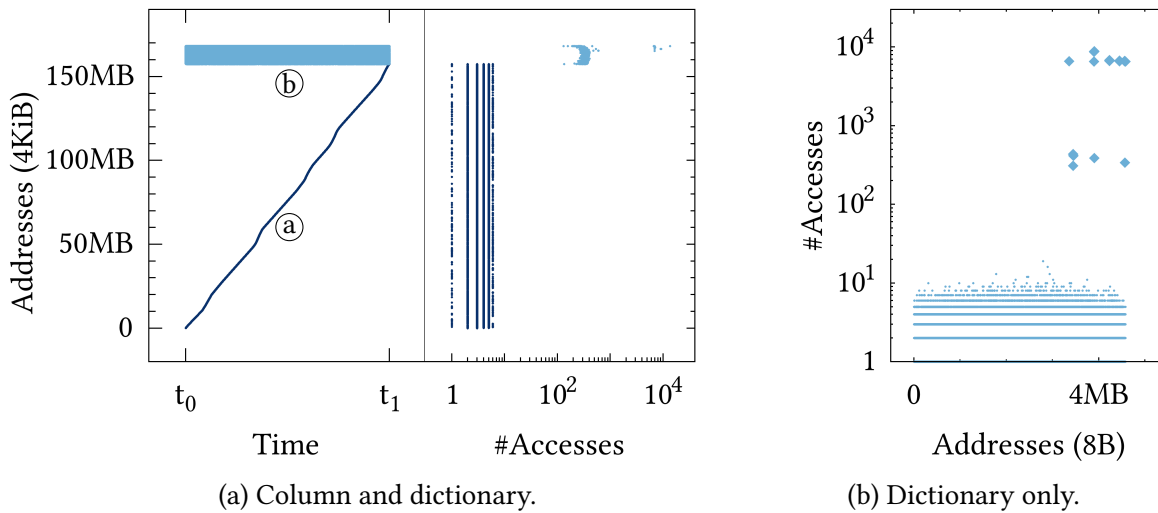


Figure 3.5: The aggregation operator of SAP HANA accesses the encoded column ■ (a) *sequentially*. Due to the data distribution, it accesses the dictionary ■ (b) *randomly* (a). The trace reveals skew at the granularity of individual dictionary entries (◆) (b).

memory access patterns from the visualization alone. We do *not* require the source code to collect the trace. Knowing the memory addresses of data structures by tracking allocations or knowing the implemented algorithms helps, however, to explain the trace: In order to visualize different data structures with different colors, we tracked memory allocations in DuckDB.

### 3.4.2 Access Counting at Byte level

To demonstrate how memory traces allow us to collect detailed access statistics at byte level and to reveal skew, we analyze the execution of a custom query with SAP HANA. The SQL query is shown in Listing 3.2. We limit SAP HANA to execute the query with two threads. We show only the results of the first thread (the results of the other thread are very similar).

Figure 3.5 illustrates the memory trace of the encoded column and the dictionary of `l_extendedprice`. Figure 3.5a shows the memory

accesses over time (left) and the total number of accesses as a histogram (right). Figure 3.5b visualizes the total number of accesses to only the dictionary at byte level. The samples are sorted by address in ascending order.

The aggregation operator *sequentially* reads the encoded column <sup>Ⓐ</sup>. Due to the dictionary encoding, it needs to decode each reference in the column by looking up its value in the dictionary. The data distribution causes these accesses to be *random* <sup>Ⓑ</sup>. In addition, we observe that the dictionary is accessed more frequently (l\_extendedprice contains 97.77 % duplicates). This demonstrates that memory traces show both the access pattern and the access frequency in detail.

When we look only at the dictionary, the trace reveals that the dictionary accesses are heavily skewed: 20 entries are accessed more frequently than others (by several orders of magnitude). Note that this property of the JCC-H benchmark becomes easily observable with the memory trace: We are able to identify “hot” data at the granularity of *memory loads*—not only at the granularity of pages [67]. By tracking memory allocations in SAP HANA, we know the memory address range of the dictionary. This allows us to identify, for example, that the dictionary entry at position 997959 (with the value 55740.45) has the most accesses.

### 3.4.3 Hot Working Set Size

To demonstrate how memory traces enable us to estimate a workload’s “hot” working set size, we execute a workload with 200 random queries of the JCC-H benchmark with SAP HANA. To evaluate a smaller data set, we also run a modified version of the workload with only 85 out of 200 queries that do not reference the orders table. SAP HANA executes the workloads on all 48 logical cores. Figure 3.6a visualizes the memory trace as a histogram, where we sort the sampled addresses by how often they occur in the trace.

We observe that the complete workload accesses table data (encoded columns and dictionaries) with a size of 1.8 GB. In contrast, the total size of the table data of all tables amounts to 3.8 GB in main memory. This demonstrates that the memory trace allows us to quantify the *working set size*, i.e., the size of the table data that is *actually* accessed during the execution of the benchmark. Additionally, we can measure how much data the workload accesses with a specific frequency, i.e., the “hot” data. We discover, for example, that the system accesses table data with a size of 600 MB more frequently.

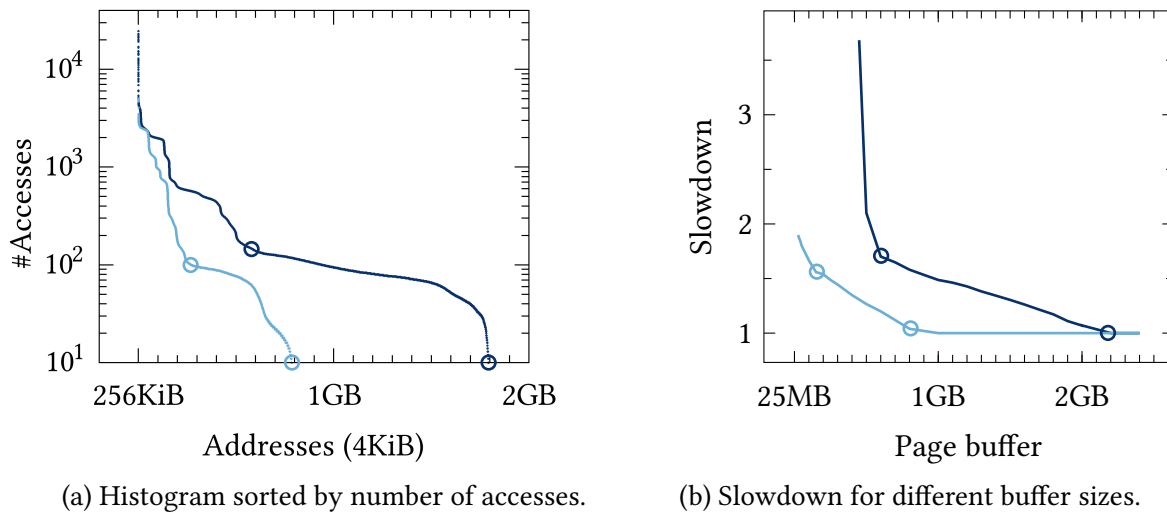


Figure 3.6: Workload consisting of 200 queries of the JCC-H benchmark referencing all tables ■ and a workload consisting of 85 queries that do not reference the orders table ■. Their working set size associated with a specific access frequency (a) matches the performance characteristics of executing the workload with a specific page buffer size (b), highlighted with ○.

**Deriving a Buffer Size.** We can use this information to derive a buffer size for SAP HANA when we execute the workload with page-loadable columns [182], i.e., using a page buffer to hold only a subset of the table data—at page granularity—in memory. Figure 3.6b shows how the size of the page buffer impacts execution time: It illustrates the relative slowdown compared to the execution time when all data fits in memory. If we compare the results to the working set size of a specific access frequency, shown in Figure 3.6a, we observe a strong similarity (highlighted in the figures with ○). The same holds true for the workload without the orders table.

We argue that the traces could help to determine the optimal buffer size for disk-based systems [145] or help to size DRAM when using NVRAM as main memory and DRAM only as a cache—referred to by Intel as “memory mode” [90].

### 3.4.4 Table Partitioning

To demonstrate how memory traces allow us to analyze table partitioning, we use again the JCC-H workload consisting of 200 queries. SAP HANA executes the query workload on all 48 logical cores. Figure 3.7

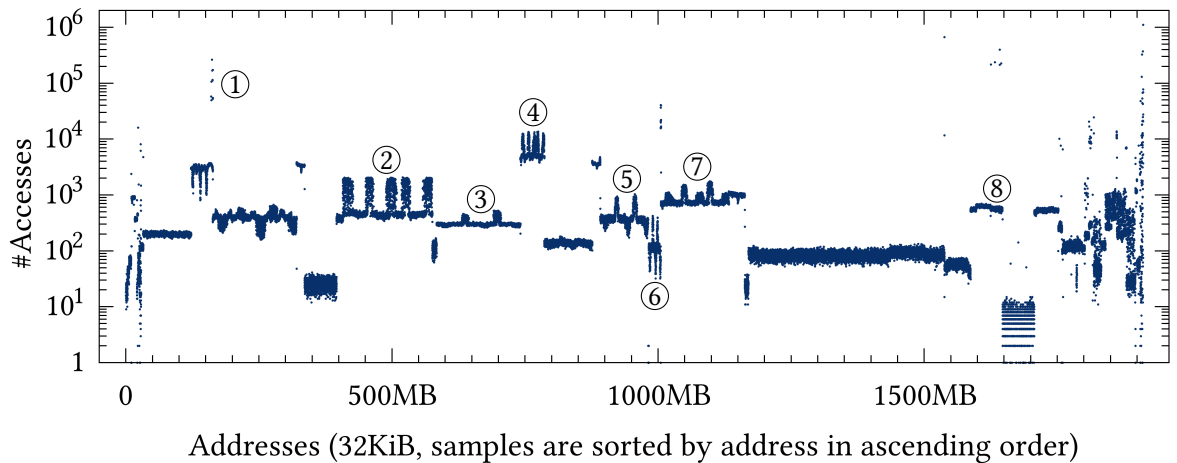


Figure 3.7: JCC-H benchmark with 200 random queries. The memory trace details the access pattern of the table data and reveals skew. We detect the 5 “populous orders”, e.g., for the encoded columns `l_orderkey` ② and `l_quantity` ④. We also detect filter skew that causes 2 of the populous orders to be accessed more frequently, e.g., for the encoded columns `l_partkey` ③, `l_shipdate` ⑤, `l_shipmode` ⑥, and `l_suppkey` ⑦.

shows the memory trace as a histogram, where the samples are sorted by address in ascending order. The memory trace illustrates the access pattern of the encoded columns and dictionaries.

The memory trace reveals *data skew* [28]. The skew becomes visible in the access pattern of the encoded column `l_orderkey` ② and `l_quantity` ④, where 5 parts of the columns (corresponding to 5 *special orders*<sup>6</sup>) are accessed more frequently. In addition, the trace also reveals *filter skew* [28]. In the access pattern of the encoded columns `l_partkey` ③, `l_shipdate` ⑤, `l_shipmode` ⑥, as well as `l_suppkey` ⑦, we observe that only 2 parts of the column (2 of the 5 special orders) are accessed more frequently. The reason is that query predicates include the years 1993 and 1994 more often, resulting in more accesses to the 2 special orders of the two years. The trace also highlights the data skew of the encoded column `l_extendedprice`: 20 distinct values of the dictionary ① are accessed more frequently (cf. Section 3.4.2).

**Impact of Partitioning.** We can use the trace to analyze the impact of table partitioning. In particular, we compare no partitioning

<sup>6</sup>The JCC-H benchmark features data skew: 25% of the rows of the *lineitem* table correspond to only 5 rows of the *orders* table. Boncz et al. call these special orders the 5 “populous orders” because each of the orders consists of a lot of line items. A special order occurs only once per year in the data set.

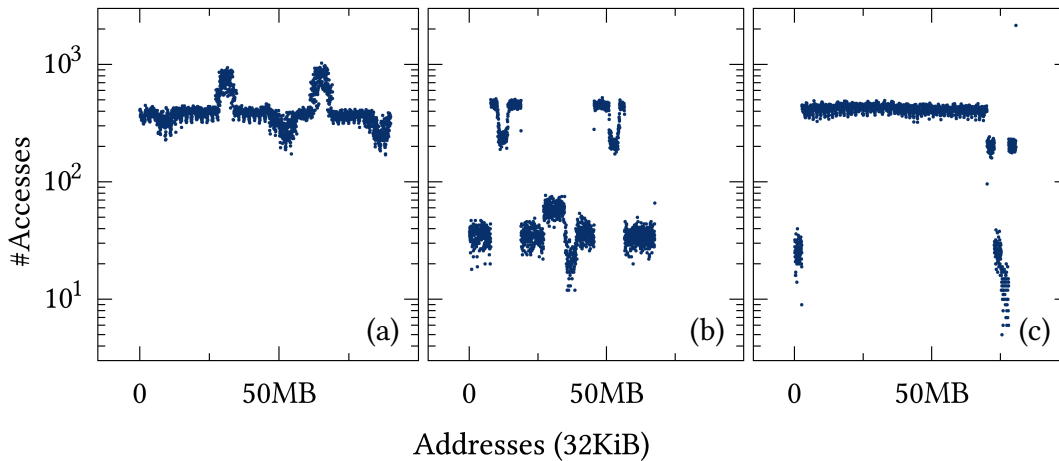


Figure 3.8: Impact of partitioning on `l_shipdate`. Comparison of no partitioning (a), partitioning per year (b) and partitioning per special order (c).

(previously shown in Figure 3.7) to the partitioning used by Microsoft SQL Server 2017 for the TPC-H benchmark [42]. They recommend a range partition on the columns `o_orderdate` and `l_shipdate`. Thus, we perform a range partition per *year* on the two columns which splits the tables `lineitem` and `orders` in 7 partitions each. Furthermore, we compare to a range partition on the column `l_orderkey`, which splits the `lineitem` table in 6 partitions: a partition per one of the 5 *special orders* and one partition holding the remaining rows. Figures 3.8, 3.9, and 3.10 visualize how the different partitioning schemes change the access pattern of the encoded columns `l_shipdate` ⑤, `l_orderkey` ②, and `o_orderdate` ⑧.

We observe that the partitioning causes some parts of the columns to be accessed rarely, i.e., they become “colder”. Instead of focusing only on execution time, the memory trace enables us to evaluate the partitioning schemes by *quantifying* the accessed data volume: To estimate the data volume, we multiply the number of accesses per bucket with the bucket size. The results show that the partitioning per *year* decreases the accessed data volume for `l_shipdate`, `l_orderkey`, and `o_orderdate` by 72 %, 5 %, and 93 %. The partitioning per *special order* decreases the accessed data volume by 19 %, 15 %, and 0 %, respectively.

While the partitioning per year allows for partition pruning whenever a filter predicate selects only some years, the partitioning per special order increases the access locality of the `lineitem` table for special orders. We observed similar effects for other columns (not shown). In addition, the traces illustrate skew: For the column `o_orderdate`, the



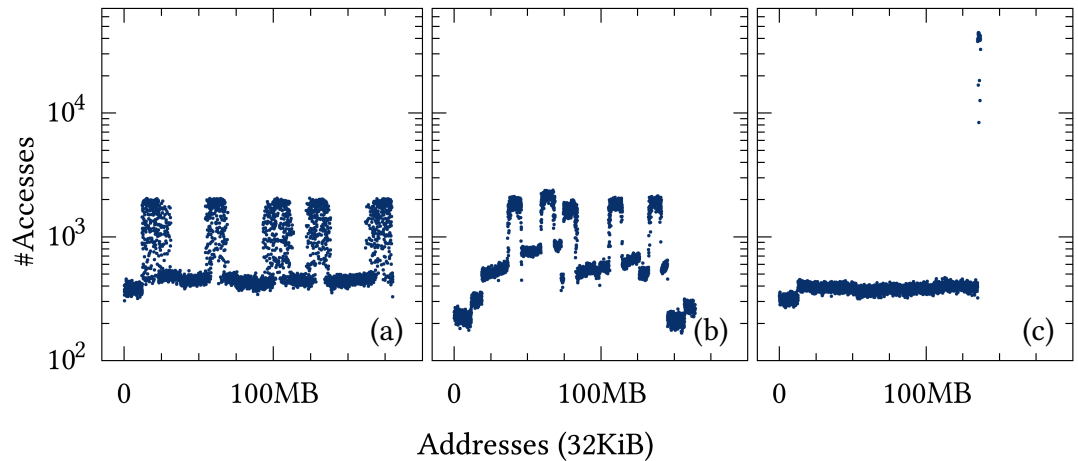


Figure 3.9: Impact of partitioning on `l_orderkey`. Comparison of no partitioning (a), partitioning per year (b) and partitioning per special order (c).

memory traces of no partitioning and partitioning per special order report more than  $10^5$  accesses to the rows corresponding to the 5 special orders. In contrast, the traces report less than  $10^3$  accesses to the remaining rows.

### 3.5 Overhead

The advantage of using Intel’s PEBS mechanism for memory tracing is that the *hardware* writes sampled memory addresses automatically to a memory-resident buffer; software needs to drain the buffer only when it is full. The PEBS mechanism allows the user to configure the sampling rate by setting the threshold that controls after how many events the CPU generates a PEBS record (cf. Section 3.2). By configuring the threshold, the user can manually adjust the trade-off between runtime overhead and precision.

For our final contribution in this chapter, we analyze the runtime overhead and the memory footprint of our implementation using the PEBS mechanism with different thresholds. Table 3.1 shows how the threshold impacts the execution time of the JCC-H workload of 200 queries running on SAP HANA on all 48 logical cores. In addition, the table highlights how the threshold impacts the size of the complete memory trace as well as the size of the filtered memory trace containing only memory addresses of table data.

The results show that as we lower the threshold the runtime over-

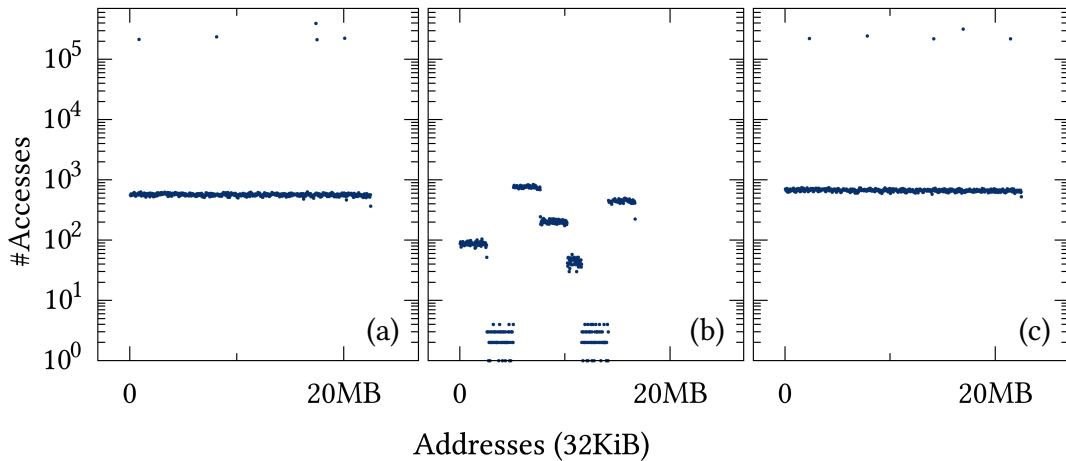


Figure 3.10: Impact of partitioning on `o_orderdate`. Comparison of no partitioning (a), partitioning per year (b) and partitioning per special order (c).

head and the memory footprint increase proportionally. We observe that, for a threshold of 200, our implementation increases runtime by a factor of 2.3. In contrast, for a threshold of 1000, our implementation increases runtime by a factor of 1.27, i.e., by 27 %.

We use a threshold of 200 in the experiments shown in Section 3.4.1 and Section 3.4.2 to illustrate how a high sampling rate (i.e., a low threshold) allows us to observe, e.g., access patterns in great detail. We use a threshold of 1000 in the experiments shown in Section 3.4.3 and Section 3.4.4 to illustrate that even a lower sampling rate allows us to collect detailed access statistics to determine the working set size or to inspect and analyze table partitioning. To choose an appropriate sampling rate, the user needs to decide how much runtime overhead is acceptable and how precise the access statistics need to be. As a rule of thumb, we propose to choose a higher sampling rate for analyzing memory access patterns, i.e., how data structures are accessed over time (cf. Section 3.4.1 and Section 3.4.2) and a lower sampling rate for analyzing total access statistics (cf. Section 3.4.3 and Section 3.4.4).

In summary, the results demonstrate that our implementation, that collects a memory trace via hardware-based sampling, is more than an order of magnitude faster than approaches such as Valgrind [143] or Intel’s Pin [124], that collect a full memory trace via binary instrumentation. Thus, our implementation makes memory tracing practical for complex systems such as database systems. In addition, we argue that the low runtime overhead enables profiling with memory traces even in production environments.

Threshold	Slowdown	Trace [GB]	Trace* [GB]
200	2.30×	45.192	1.518
400	1.67×	22.999	0.855
600	1.45×	15.393	0.548
800	1.34×	11.585	0.375
1000	1.27×	9.285	0.308
2000	1.13×	4.656	0.134
4000	1.05×	2.324	0.064

Table 3.1: Tracing overhead for different thresholds for the PEBS mechanism. The table shows the relative slowdown of the execution time, the size of the complete trace, and the size of the trace containing only addresses of table data (\*), i.e., of the encoded columns and the dictionaries.

Moreover, a user can also lower the runtime overhead further by monitoring different hardware events. In our experiments, we use the hardware performance counter `mem_load_uops_retired.all_loads` to capture cache hits *and* cache misses. Collecting memory traces for an event which captures *only* cache misses, would reduce the runtime overhead significantly because cache misses usually occur less frequent than cache hits.

## 3.6 Related Work

### Tracing Methods

Related work from the systems community explores various approaches for collecting memory traces. They investigate memory tracing via hardware emulation [22], by passing all memory access through an FPGA [118], or by using custom hardware to snoop the memory bus of DRAM DIMMs [20]. In addition, related work uses binary instrumentation [32, 33, 56, 124, 143] to trace memory accesses. Binary instrumentation modifies the program code of the application by injection additional tracing code that captures the memory addresses of, e.g., load and store instructions. These approaches allow tracing *all* memory accesses at the cost of slowdowns by more than an order of magnitude or require additional hardware.

Others propose to use performance monitoring units of modern processors from AMD [103], Intel [180], or IBM Power [191] to trace memory accesses via hardware-based *sampling*. The integration effort or the runtime overhead of employing such an approach in practice remains unclear, however. Other related work [5, 151] studies PEBS parameters such as the size of the record buffer and the sampling rate—but not for an end-to-end database workload. To the best of our knowledge, we are the first to explore memory tracing via PEBS running (a complex workload on) a commercial database system.

### Use Cases

Related work from the database community explores approaches for managing hot (frequently accessed) and cold (rarely accessed) data in tiered storage architectures. While one side advocates the use replacement policies for buffering and caching mechanism for managing hot and cold data [51, 113], others utilize access statistics: For example, Funke et al. [67] collect access statistics by regularly checking flags of the memory management unit to identify rarely accessed virtual memory pages. They use the access statistics to compact cold OLTP data at the granularity of a virtual memory page. Levandoski et al. [57, 116] propose to log (a sample of) record accesses to estimate record access frequencies with an offline analysis. They migrate cold OLTP data at the granularity of rows to secondary storage. Boissier et al. [27] get access statistics of a column from the query optimizer. They use a hybrid table layout where they move individual columns of a table to secondary storage.

We argue that memory tracing may be used to collect access statistics at byte granularity and with a low runtime overhead. We demonstrate that our approach is feasible by tracing memory accesses of a commercial, main-memory database system, as shown example in Section 3.4.2. However, when a system frequently loads and unloads data from secondary storage, we may need to additionally protocol whenever the virtual memory address of table data changes, e.g., by tracking memory allocations or the assignment of buffer pages.

While we use memory tracing to count accesses, to detect skew, to study access patterns, and to analyze table partitioning, related work explores many other use cases. Tözün et al. [195, 196] use memory traces together with hardware simulation to map cache misses back to database operators and to components of the storage manager for OLTP workloads. They propose a transaction scheduling mechanism that uses

memory traces to maximize instruction cache locality. Others use memory traces to build cache miss ratio curves to quantify an application's cache usage [191] or to derive cache partitioning schemes [208], to detect memory errors such as buffer overflows or use-after-frees [185], to detect false sharing [178], to optimize data placement in NUMA systems [48, 130], to pinpoint performance bottlenecks related to cache usage [157], to remove redundant memory loads [189], to learn memory access patterns [81, 85, 119], or to optimize software-based prefetchers [16].

Applying these methods on top of our memory tracing implementation could provide new insights into database systems and possibly reveal optimization opportunities that are hard to discover with current profiling approaches.

### 3.7 Conclusion

The state-of-the-art solution for identifying the root cause of performance problems related to memory accesses is to augment classical profiling with a memory trace. However, current approaches for memory tracing are not usable in practice due to their large runtime overhead.

In this chapter, we present and evaluate an implementation for collecting memory traces via *hardware-based sampling* that leverages Intel's PEBS mechanism. In our experiments using the JCC-H benchmark, SAP HANA, and DuckDB, we illustrate for various use cases that memory traces enable us to analyze the runtime characteristics of a database system. We demonstrate that memory traces reveal access patterns and access statistics for individual data structures and database operators, detect skew at byte level, and allow us to estimate the working set size of a workload as well as to analyze the impact of table partitioning. Deriving the working set size of a workload allows us, e.g., to size the page buffer of a database system.

In addition, we demonstrate that our implementation has a low runtime overhead: For a threshold of 1000 it increases runtime overhead by 27%. This makes it possible to trace memory accesses of complex systems, such as database systems, even in production environments. In summary, our approach opens up new possibilities to inspect and analyze complex software systems and to optimize resource usage, especially memory and cache usage.



# 4

## CPU Cache Partitioning

Modern microprocessors include a sophisticated hierarchy of caches to hide the latency of memory access and thereby speed up data processing. However, multiple cores within a processor usually share the same last-level cache (cf. Section 2.1). A shared cache can hurt performance, especially in concurrent workloads whenever a query suffers from cache pollution caused by another query running on the same processor.

In this chapter, we confirm that this particularly holds true for the different operators of an in-memory DBMS: The throughput of cache-sensitive operators can degrade by more than 50 % for concurrent analytical workloads. To remedy this issue, we devise a cache allocation scheme from an empirical analysis of different operators and integrate a cache partitioning mechanism into the execution engine of a commercial DBMS. Finally, we demonstrate that our approach improves the overall system performance by up to 38 %.

In particular, we make the following contributions: *(i)* we empirically analyze the cache requirements of key DBMS operators by varying available cache sizes; *(ii)* we derive cache partitioning schemes to reserve cache capacity for cache-sensitive operators and queries; *(iii)* we discuss how cache partitioning support can be retrofitted into an existing DBMS with low engineering costs using SAP HANA as an example; and *(iv)* we evaluate the practical benefits of the proposed techniques using both standard benchmarks and queries from a modern HTAP business application (S/4 HANA).

**Outline.** We introduce the problem in Section 4.1. In Section 4.2, we introduce key data structures and database operators of SAP HANA relevant for this chapter. We analyze the cache usage of database operators in Section 4.3. In Section 4.4, we discuss our approach of integrating cache allocation control into an existing system and derive cache partitioning schemes from our analysis. Subsequently, we evaluate our cache partitioning approach in Section 4.5. We cover related work in Section 4.6 and we conclude the chapter in Section 4.7.

Parts of this chapter are published in [148].

## 4.1 Introduction

The key limitation in all of today’s microprocessor designs is the dissipation of heat [31]. To prevent overheating, not all transistors in a chip can be active at the same time, an effect also called dark silicon [59]. In fact, hardware manufacturers dedicate large fractions of the available chip space of mainstream processors to on-chip *caches*. Since caches consume less power than processing logic, using transistors to build large caches is one way to stay within reasonable power budgets [31].

Manufactures of mainstream processors optimize cache sizes for the entirety of applications that may run on these processors. They do not optimize cache sizes for a single application. Hence, it remains unclear if the selected cache sizes are optimal for a specific scenario such as a database system running a specific workload. In particular, it is unclear how much CPU cache individual database operators of a query execution plan need to meet performance guarantees.

Modern database systems allow applications and users to concurrently execute transactional and analytical workloads on the same data set (cf. Section 2.2). Concurrent queries usually have different resource requirements, e.g., depending on the workload, the number of records accessed, as well as the data structures and algorithms being used. In particular, some operations are highly sensitive to the available amount of CPU cache (e.g., random accesses to a small hash table), contrary to cache-insensitive operations such as a sequential memory scan.

Consider the example of the mixed workload illustrated in Figure 4.1. Using SAP HANA, We execute an OLTP query either isolated, concurrently to an OLAP query, or concurrently to an OLAP query with cache partitioning applied. Our measurements show that the throughput of the OLTP query *degrades* significantly when executed *concurrently* to



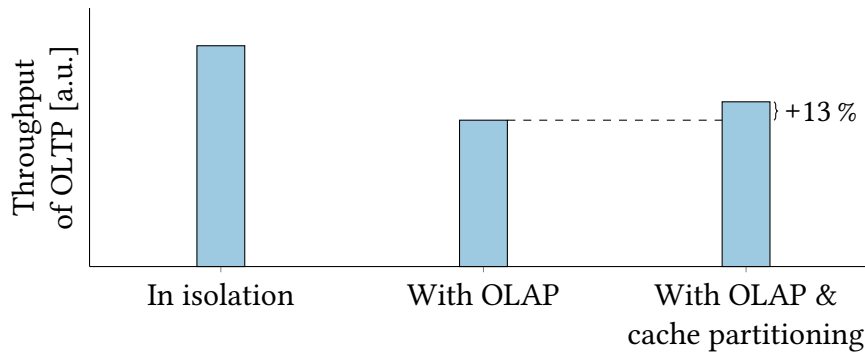


Figure 4.1: Throughput of an OLTP query running either isolated, concurrently to an OLAP query, or concurrently to an OLAP query with cache partitioning applied. Restricting the LLC for the OLAP query by partitioning the cache avoids cache pollution and improves performance of the OLTP query.

the OLAP query. Performance degrades because the queries compete for shared resources such as the processor’s *last-level cache* (cf. Section 2.1) causing *cache pollution*.

Figure 4.2a visualizes cache pollution. Cache pollution occurs whenever an operation evicts cache lines from a shared cache that are frequently accessed by another operation. In the example shown in Figure 4.1, the OLAP query *pollutes* the cache by frequently loading data from DRAM into the LLC, thus evicting data from the cache needed by the OLTP query. As a result, the performance of the OLTP query degrades significantly.

For system designers, the good news is that hardware manufacturers allow fine-grained control of cache allocation by offering mechanisms such as Intel’s *Cache Allocation Technology* (CAT) [89]. It allows avoiding cache pollution by employing cache partitioning (cf. Figure 4.2b). However, besides some isolated analysis of individual algorithms and data structures, it is still unclear how easy it is to apply these mechanisms in the context of real-world systems, and whether the corresponding integration effort pays off.

To fill this gap, we study the impact of the CPU cache size on various query workloads and analyze the effect of careful cache partitioning on the overall system performance, we propose to integrate cache partitioning into a DBMS by restricting the LLC for scan-intensive operators that cause cache pollution, and we show that our approach can improve performance significantly without introducing regressions.

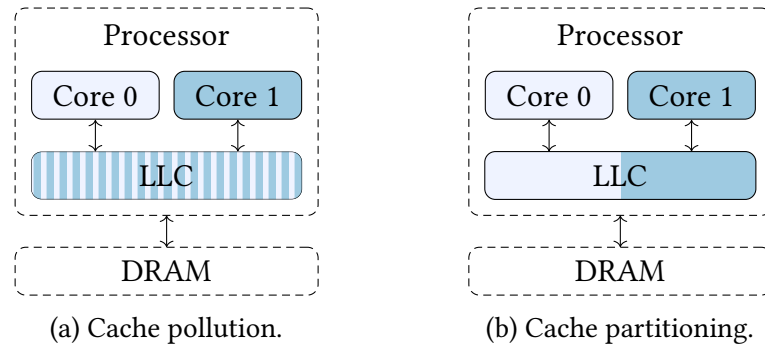


Figure 4.2: Database operations may cause *cache pollution*. By accessing data from main memory, an operation (on core 0) may evict cache lines from the shared LLC of the processor that are frequently accessed by another operation (on core 1) (a). *Cache partitioning* can eliminate cache pollution by controlling how cores access the shared LLC (b).

## 4.2 Query Execution in SAP HANA

As a poster child for our analysis of cache usage, we use the in-memory database system SAP HANA [62] (cf. Section 2.2). To better interpret the experiments described later in this chapter, we give a brief overview of the most relevant implementation details of SAP HANA’s query execution engine. First, we present key data structures that are commonly used in the execution engine. Then, we briefly describe some of the engine’s operations and algorithms that utilize these data structures.

### 4.2.1 Data Structures

The execution engine of SAP HANA uses tailor-made, cache-optimized data structures. For the scope of our work in this chapter, three data structures are most relevant: (i) *dictionaries*, which help to compress columnar data in SAP HANA, but also speed up value comparisons; (ii) *hash tables*, which are commonly used in *aggregation with grouping*; and (iii) *bit vectors*, which accelerate the processing of *foreign key joins*. These data structures are used throughout SAP HANA’s query processing engine (not just for the types of queries we study in this work).

**Dictionary.** Dictionaries play a significant role in the compression-optimized execution engine of SAP HANA. The ordered dictionary

maps a column's domain values to a dense set of consecutive numbers. Instead of storing the actual value in the columns of a table, the storage engine of SAP HANA stores the typically much smaller number referencing an entry in the dictionary (cf. Section 2.2.1). If data needs to be decompressed during query processing, e.g., for projection or intermediate result construction, the dictionary is accessed frequently to look up the actual value.

**Hash Table.** Hash tables are a prominent example in the context of cache-sensitive data structures and operations. By nature, they are typically accessed in a random-access fashion, which can be very expensive when the hash table does not fit into the CPU caches. In SAP HANA, individual algorithms such as *grouped aggregation* use hash tables, e.g., to store temporary results for different groups. They are used both locally per worker thread and globally to merge thread-local results. Characteristic is their very frequent access during query processing.

**Bit Vector.** Bit vectors accelerate, e.g., the evaluation of *foreign key joins* in the OLAP-optimized join algorithms of the execution engine of SAP HANA. The bit vectors map the primary key range to a highly compact representation, which can be kept in CPU caches even for a large key range. Using bit vectors is known to reduce memory loads and CPU costs since the CPU can perform the same operation on multiple elements of a bit vector at once [2, 162]. Mapping tables of this kind is also known to be sensitive to caches [129].

### 4.2.2 Operations

SAP HANA's query execution engine employs many different algorithms and operators. We describe the three core operations that are most relevant in this chapter: the *column scan*, *aggregation with grouping*, and the *foreign key join*.

**Column Scan.** The operator evaluates a range predicate. It reads a column of a table sequentially and identifies all rows that satisfy the predicate expression. The operator works on compressed data. It uses SIMD instructions to process multiple encoded values at once, which significantly improves performance [204, 205].

Note that the *column scan* operator reads data from DRAM only once. It exploits data locality by processing each byte of a cache line. Thus, it profits from the hardware prefetcher, i.e., the CPU can load cache lines into the cache before they are requested. The *column scan* does not depend on any of the data structures mentioned in the previous section.

**Aggregation With Grouping.** The operator aggregates columns while grouping the aggregated values by the contents of other columns. The algorithm proceeds as follows: First, it distributes its input among a set of worker threads. Then, each worker thread collects aggregates locally for its partition. After all threads have finished aggregating, the algorithm merges the local results to build the global result for the next operator of the query plan.

The *aggregation with grouping* operator decompresses the input data to compute the aggregate [198]. As a result, the operator performs many random accesses to the dictionary. Furthermore, the algorithm uses hash tables to store intermediate, pre-aggregated results for every group and to store the merged results globally—similar to work from Ye et al. [210]. Thus, accessing the hash tables results in additional random memory accesses.

**Foreign Key Join.** The *join* operator is optimized for OLAP workloads and exploits the fact that a foreign key maps to exactly one primary key. In a first step, the *join* algorithm creates a very compact representation of the primary keys by mapping the keys to a bit vector. If the primary keys range from 1 to  $N$ , the algorithm creates a bit vector of length  $N$  and sets the  $i$ -th bit if the query's predicate evaluates to true for the row of primary key  $i$ . The resulting bit vector usually fits in the CPU caches even for a large number of keys. During the next step, the algorithm performs a look-up in the bit vector for each foreign key to check if it matches a primary key. In addition, it aggregates the matches.

### 4.3 Analysis of LLC Usage

To motivate why cache partitioning can improve the performance of concurrent workloads, we first analyze the cache usage of individual database algorithms. The empirical analysis is our first contribution of this chapter. Our goal is to determine how much last-level cache a

```
-- (1) Column Scan
SELECT COUNT(*) FROM A WHERE A.X > ?;

-- (2) Aggregation With Grouping
SELECT MAX(B.V), B.G FROM B GROUP BY B.G;

-- (3) Foreign Key Join
SELECT COUNT(*) FROM R, S WHERE R.P = S.F;
```

Listing 4.1: The three SQL queries executed in the experimental analysis. Each query focuses on a specific database operator.

database operator needs to reach the best performance. We also study the impact of data distribution on an operator’s cache usage. The results of our analysis allow us to derive cache partitioning schemes for the concurrent execution of these operators.

First, we describe the experimental setup. Subsequently, we present the results of our evaluation. We analyze the cache usage of the *column scan* operator, *aggregation with grouping*, and the *foreign key join* operator. Finally, we summarize and discuss the results of our analysis.

### 4.3.1 Experimental Setup

We detail the experimental setup of our analysis in the following. We describe the SQL queries, the SQL schema of the tables and their data distribution, the hardware platform, and the measurement method.

#### Queries

Since we want to see the effects of CPU caches on end-to-end performance, we express our benchmark queries on the SQL level and measure full query execution times. The three queries used in our experiments are listed in Listing 4.1. We keep the queries deliberately simple so that each query is dominated by the execution of a specific database operator: (1) *column scan*, (2) *aggregation with grouping*, and (3) *foreign key join*. The first query evaluates a range predicate on a column using the *column scan* operator. We use the parameter “?” to vary the selectivity of the predicate. The second query computes the maximum value of a column grouped by another column. The third query executes a foreign key join between two tables.

```
-- Schema for Column Scan
CREATE COLUMN TABLE A( X INT );

-- Schema for Aggregation With Grouping
CREATE COLUMN TABLE B( V INT, G INT );

-- Schema for Foreign Key Join
CREATE COLUMN TABLE R( P INT, PRIMARY KEY(P));
CREATE COLUMN TABLE S( F INT );
```

Listing 4.2: The different SQL table schemata used in the experimental analysis.

### Data Sets

Listing 4.2 illustrates the SQL schemata of the column tables used in the experiments. We fill the table with generated data (no null values) and vary the distribution of the data to study its impact on the operators' cache usage.

**Column Scan.** The input data for the first query is a table consisting of one column with  $10^9$  integers. We generate random numbers between 1 and  $10^6$  with a uniform distribution. While the integers initially have a size of 32 bits, SAP HANA applies compression to store each integer using  $\lceil \log_2(10^6) \rceil = 20$  bits. To vary the selectivity of the predicate, we set the parameter “?” to a new random integer between 1 and  $10^6$  for every execution of the query.

**Aggregation With Grouping.** The input data of the second query is a table consisting of two columns with  $10^9$  integers. The first column *V* is used for aggregating while the second column *G* is used for grouping. We vary the number of distinct values by randomly picking integers from 1 to  $N$  (uniform distribution). For column *V* we vary  $N$  between  $10^6$  and  $10^8$ , which changes the size of the dictionary, and for column *G* we vary  $N$  between  $10^2$  and  $10^6$ , which changes the number of groups and thus impacts the size of the hash tables used by the algorithm.

**Foreign Key Join.** The input data of the third query consists of two tables with one column each. Column *P* of the first table contains distinct integers that form a primary key ranging from 1 to  $N$ . We vary

$N$  between  $10^6$  and  $10^9$ , which impacts the number of matches and the size of the bit vector used by the *join* algorithm. Column F of the second table contains  $10^9$  integers referencing the primary key of the first table. We generate the foreign keys by randomly picking numbers from column P.

### Hardware Platform

We perform the experiments with a prototype of SAP HANA running on a single socket system with 128 GiB of main memory. We use SUSE Linux Enterprise Server 12.1 as the operating system, but update the Linux kernel to version 4.10. The system features an Intel Xeon E5-2699 v4 processor with 22 cores. With simultaneous multithreading enabled, the processor can execute 44 threads in parallel.

Using the Intel Memory Latency Checker [200], we determine that DRAM has a memory read bandwidth of 64 GB/s and an access latency of 80 ns. The shared LLC has a size of 55 MiB. It stores both data and instructions and it is inclusive: This means the LLC contains all the information stored in the other caches of the cache hierarchy.

### Measurement Method

To analyze the cache usage of SAP HANA and to determine the impact of a smaller cache on an operator's performance, we limit the size of the available LLC [89]. Thus, the entire instance of SAP HANA can only allocate data into a limited size of the LLC. We state the available size of the cache in *mebibyte* (MiB). 1 MiB equals  $2^{20}$  bytes.

We execute SQL queries with SAP HANA and measure end-to-end response time, i.e., the total execution time including parsing, optimizing, query execution and result transfer. Note that we set the concurrency limit of an SQL statement to the number of physical cores of the system. Consequently, a query is potentially executed on all available cores of the processor. We normalize a query's throughput to its maximum throughput when using the entire cache. In addition, we measure the LLC hit ratio and the LLC misses per instruction using Intel's Processor Counter Monitor [203].

#### 4.3.2 Column Scan

Figure 4.3 shows the impact of the cache size on the throughput of the *column scan* operator. We observe that the throughput remains

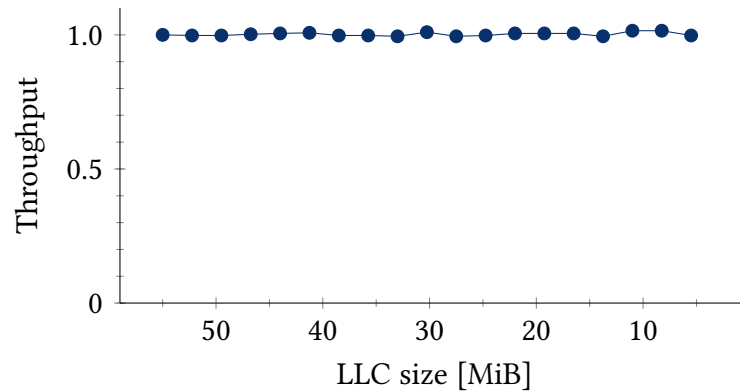


Figure 4.3: Normalized throughput of the *column scan* at varying LLC sizes. The operator is hardly sensitive to the size of the cache.

unaffected by the cache size. In addition, we measure that the LLC hit ratio is below 0.08, while the LLC misses per instruction amount to  $1.9 \cdot 10^{-2}$ , independent of the cache size. Thus, we determine that the *scan* is not sensitive to the cache size.

The results do not come as a surprise because the *column scan* reads data from DRAM only once without reusing it. Moreover, the sequential memory access pattern of the *scan* operator features strong data locality. Therefore, it profits from the hardware prefetcher of the CPU. Furthermore, the *column scan* operator takes advantage of the fact that it can process compressed data. It does not need to perform lookups into a column’s dictionary. This is possible because SAP HANA’s dictionary encoding is *order preserving*. It is sufficient to map the query parameter “?” to its dictionary code, then execute the query entirely on compressed data [204, 205]. That means that during the execution of the *column scan*, the CPU does not need to hold the dictionary in the cache.

### 4.3.3 Aggregation With Grouping

The evaluation of the *aggregation with grouping* operator is split into three different experiments. We vary the number of distinct values in the column *V* to change the size of the dictionary to 4 MiB, 40 MiB and 400 MiB. Then, in each experiment we alter the number of groups (column *G*) in addition to changing the size of the LLC.



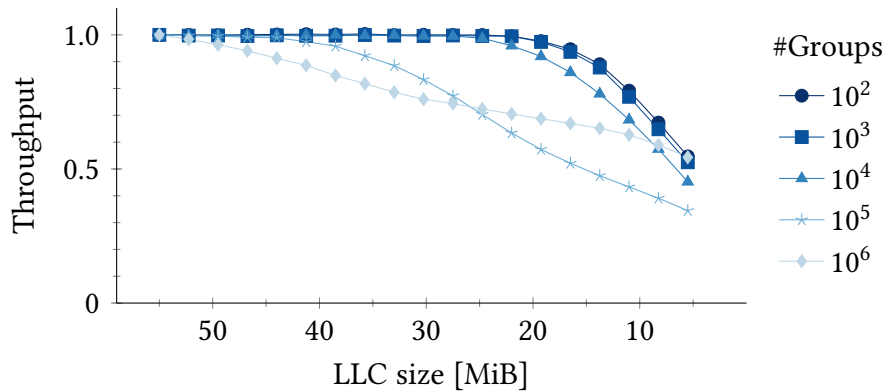


Figure 4.4: Normalized throughput of *aggregation with grouping* at varying LLC sizes and varying number of groups. We set the dictionary size of the column that is aggregated to 4 MiB: The operator is slightly sensitive to the size of the cache for smaller groups and highly sensitive for larger groups.

#### Dictionary Size of 4 MiB

Figure 4.4 visualizes the results of *aggregation with grouping* using a data set with  $10^6$  distinct values in the column  $V$ . This results in a dictionary size of approximately 4 MiB. Thus, the dictionary fits completely in the LLC (55 MiB) but exceeds the size of a single L2 cache (256 KiB).

The results show that for a group size of  $10^2$ ,  $10^3$  and  $10^4$  throughput degrades as soon as query execution is forced to use less than 20 MiB of the cache. We notice that throughput degrades by more than 46 % if we limit the size of the cache to approximately 5 MiB. In addition, we observe the strongest throughput degradation with  $10^5$  groups: The curve breaks at a cache size of less than 40 MiB; throughput degrades by 67 %. If we increase the number of groups to  $10^6$ , throughput degrades less strongly. If we limit the cache size to 25 MiB, throughput decreases by 28 %. If we reduce the cache size even further, throughput degrades by 46 %.

We explain the different performance characteristics by the size of the hash table, which is decided by the number of groups. In case of  $10^5$  different groups, the hash table occupies all the LLC. Thus, if we change the size of the available cache, we observe the most significant impact on performance. If the number of groups is smaller, the hash table is so small that even a small portion of the cache is enough to store it entirely. If the number of groups is bigger, the size of the hash table exceeds the size of the LLC. As a result, the hash table does not

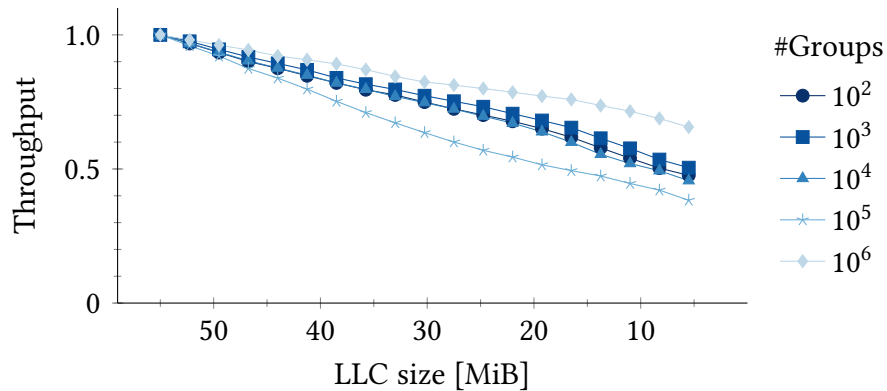


Figure 4.5: Normalized throughput of *aggregation with grouping* at varying LLC sizes and varying number of groups. We set the dictionary size of the column that is aggregated to 40 MiB: The operator is highly sensitive to the size of the cache for all group sizes.

completely fit in the cache and the algorithm suffers from cache misses even if the entire cache is used. Reducing the cache size results in an increasing number of cache misses and further degrades performance.

Similarly, we measure that the LLC hit ratio and the LLC misses per instruction decline significantly between group sizes  $10^2$  to  $10^5$  and group size  $10^6$  when the size of the hash table exceeds the size of the LLC. The LLC hit ratio drops from more than 0.9 to less than 0.6, while the LLC misses per instruction increase by an order of magnitude.

### Dictionary Size of 40 MiB

Figure 4.5 illustrates the results of *aggregation with grouping* using a data set with  $10^7$  distinct values in the column  $V$ . This results in a dictionary size of approximately 40 MiB. Thus, the dictionary can occupy a large portion of the LLC (55 MiB).

We observe that for group sizes  $10^2$  to  $10^5$  throughput drops significantly by up to 62% as we lower the size of the available LLC. In contrast, we observe that for the largest group size the impact on performance is less significant. The results show that throughput degrades by up to 34%. Moreover, if we compare the results of this experiment to the previous experiment with a dictionary size of 4 MiB (cf. Figure 4.4), we notice that throughput degrades steadily for all group sizes—even for large cache sizes.

We explain these results by the increased size of the dictionary. In contrast to the first experiment, the dictionary has a size of more

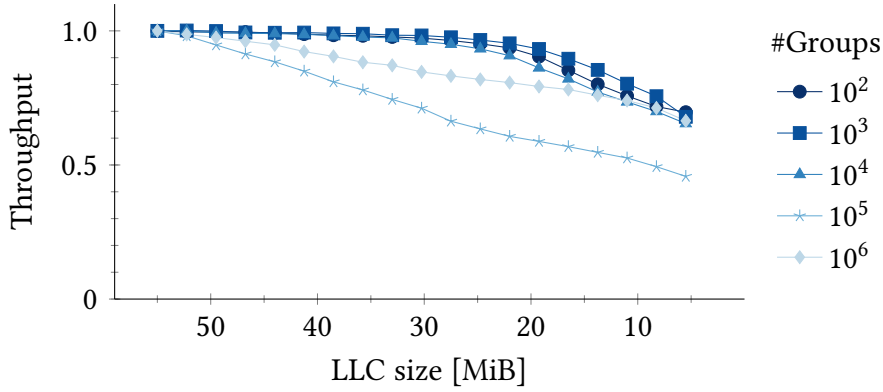


Figure 4.6: Normalized throughput of *aggregation with grouping* at varying LLC sizes and varying number of groups. We set the dictionary size of the column that is aggregated to 400 MiB: The operator is slightly sensitive to the size of the cache for smaller groups and increasingly sensitive for larger groups.

than half of the LLC. During the execution of the *aggregation*, the algorithm performs lots of random memory accesses to the dictionary to decompress the encoded values of the column before aggregating them. Therefore, the execution time of the operator is dominated by accesses to DRAM, as soon as the dictionary size exceeds the LLC size. The results show that the dictionary cannot be held in the LLC either if the number of groups is large ( $10^6$ ) or if we limit the size of available cache to equal to or less than 45 MiB.

#### Dictionary Size of 400 MiB

Figure 4.6 displays the results of *aggregation with grouping* using a data set with  $10^8$  distinct values in the column  $V$ . This results in a dictionary size of 400 MiB. Thus, the size of the dictionary exceeds the size of the LLC (55 MiB) by far.

The results reveal that the cache size impacts throughput less compared to the second experiment (cf. Figure 4.5). We observe that, as we limit the size of the cache, throughput degrades by more than 31%. For group sizes  $10^2$  to  $10^4$ , the curve breaks at a cache size of less than 30 MiB, while for a group size of  $10^6$  the curve breaks earlier at a cache size of less than 50 MiB. If the algorithm aggregates over  $10^5$  groups, throughput degrades by up to 54%.

The results are similar to the results of the first experiment (cf. Figure 4.4) if we compare where the curves break. The reason for

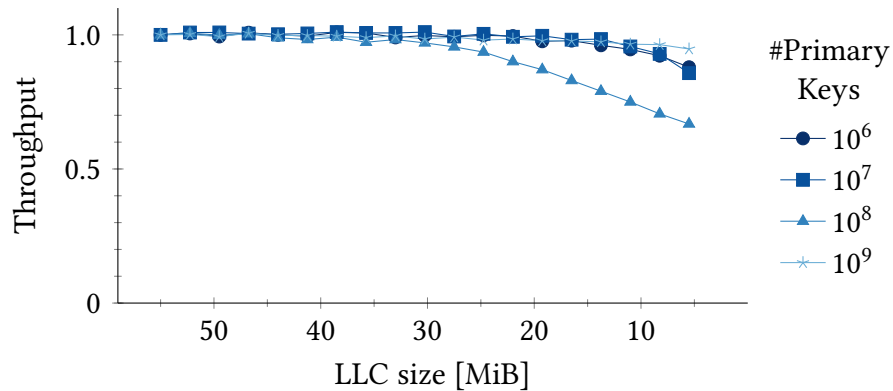


Figure 4.7: Normalized throughput of the *foreign key join* at varying LLC sizes. We vary the number of primary keys ( $P$ ). The operator is sensitive to the size of the cache only for  $10^8$  primary keys when the size of the bit vector is comparable to the size of the LLC.

this runtime behavior is again the increasing size of the hash table. This time, however, the dictionary exceeds by far the size of the LLC. Consequently, the algorithm suffers from lots of cache misses. We notice that compared to the first experiment, the cache hit ratio drops by at least 10 % to 20 %. Consequently, the overall performance degradation is less in comparison to the first experiment. However, the size of the hash tables still impacts the cache sensitivity of the algorithm significantly.

#### 4.3.4 Foreign Key Join

Figure 4.7 illustrates the throughput of the *foreign key join* with varying cache sizes. The OLAP-optimized *join* operator uses a bit vector to represent the primary keys (cf. Section 4.2). The results show that throughput worsens by only 5–14 % for  $10^6$ ,  $10^7$  and  $10^9$  primary keys. For  $10^8$  primary keys, however, throughput degrades by up to 33 %. We observe that performance deteriorates if the size of the LLC is less than 35 MiB.

We attribute the cache sensitivity of the algorithm to the size of the bit vector used to identify the matching primary keys. To represent  $10^8$  distinct keys ranging from 1 to  $10^8$ , the algorithm uses a bit vector with a size of  $10^8$  bit = 11.92 MiB. Thus, the bit vector easily fits in the LLC. In all other cases the bit vector either exceeds the LLC or fits even in the L2 cache of a processor core.

### 4.3.5 Discussion

**Summary.** Our measurements show that the *column scan* operator is hardly sensitive to the size of the cache. A *column scan* does not benefit from a large portion of the LLC and runs well with a small cache configuration (e.g., 10%). This observation does not come as a surprise because, by nature, *scans* read data exactly once from DRAM without reusing the data.

An *aggregation*, by contrast, can be highly sensitive to the size of the cache. The *aggregation with grouping* operator that we consider is based on hashing and is most cache-sensitive whenever the size of the hash tables is comparable to the configured size of the LLC. If the hash table is either very small or very large, cache sensitivity becomes less significant.

The cache sensitivity of the *foreign key join* depends on the cardinality of the primary keys: If the size of the bit vector is comparable to the size of the LLC size, the operator becomes cache-sensitive. Otherwise, the operator does not profit from a large LLC.

**Interpretation.** We interpret our results in different ways. Awareness of the characteristics of an operator’s cache usage may open up new opportunities to improve the *scheduling* mechanisms in database engines. Our observations are consistent with the findings of Lee et al. [111] who contrasted cache demand and cache usage within PostgreSQL. They propose a scheduler that could make its decisions based on each operator’s cache usage pattern. It could, for instance, pay off to co-schedule operators that have the same (or different) cache usage behavior such that *interactions* between operators could be minimized. Such scheduling mechanisms will depend on appropriate means to *describe* the cache usage pattern of a database operator. Therefore, the analysis of *cache miss rates* might be reinvestigated, ideally also combined with analytical models matching the different *cache miss rates* of individual operators [126].

Moreover, our insights align with observations made by Borkar and Chien [31], who point out that the design of newer processors has to change because too large caches might not be energy proportional. Since caches consume less power than processing logic, integrating a larger cache allows the processor to stay within its energy and power envelope. Our experimental analysis shows that the current size of the LLC might already be too large for some of the in-memory database algorithms used in SAP HANA. Thus, in order to optimize hardware for

database workloads, it may pay off to invest available transistors, e.g., in more heterogeneity or specific accelerators for individual database operations (instead of larger caches).

**Cache Pollution.** Ultimately, the DBMS should be aware of the cache sensitivity of an operation. If the DBMS knows how much cache an operation needs to reach good performance, it could not only influence the scheduling of concurrent running operations, but also carefully manage a shared cache to avoid *cache pollution*. Our results suggest that scan-intensive operators, such as the *column scan* operator or the *foreign key join* operator (for a small bit vector), may cause cache pollution for cache-sensitive operators, such as the *aggregation with grouping* operator, if they are executed concurrently. The awareness of these different cache usage characteristics allows us to classify database operators based on their cache usage and use *cache partitioning* to manage the shared LLC of a processor more efficiently for concurrent workloads.

We study cache pollution and the effect of careful *cache partitioning* in the following. In fact, we validate whether the simple approach of restricting scan-intensive operators to a minimum portion of the cache—while allowing a cache-sensitive operator to use the entire cache—can avoid cache pollution for concurrent workloads and improve performance.

## 4.4 Cache Partitioning in SAP HANA

In this section, we present the cache partitioning feature of current Intel processors, and we describe how we implement cache partitioning in the execution engine of a prototype version of SAP HANA. By integrating cache partitioning into a database system, we can avoid cache pollution and thereby improve the performance of concurrent query execution.

### 4.4.1 Cache Partitioning With CAT

Traditionally, the user has little control over the cache, as it is entirely managed by hardware. Techniques such as *page coloring* [111] offer the possibility of partitioning the cache by allocating memory in specific memory pages, known to map to a specific portion of the cache [193]. However, the use of page coloring in commercial systems is limited.

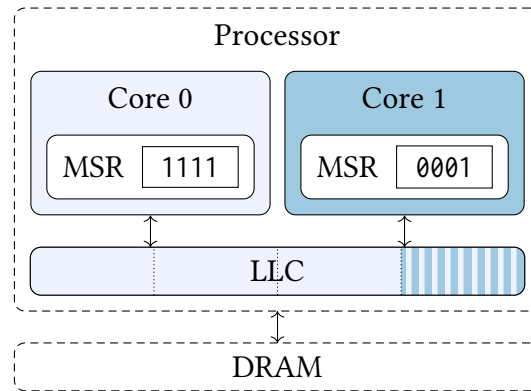


Figure 4.8: Simplified example of using Intel’s Cache Allocation Technology to partition the last-level cache (LLC). Setting a bitmask in a core’s machine-specific register (MSR) controls the cache allocation of the core. In the example, the first core can evict cache lines from the entire LLC. The second core can evict cache lines from only 25 % of the LLC. (Both cores can *read* cache lines from the entire LLC.)

Page coloring requires significant changes to the operating system and to the application, resulting in poor usability and maintainability. In addition, page coloring is less flexible because re-partitioning the cache dynamically at runtime requires copying the allocated data [120, 213].

With the “Haswell” microarchitecture Intel<sup>1</sup> introduced the possibility to partition the *last-level cache* of a processor, thereby giving the user more control over the CPU cache. Intel refers to this hardware feature as *Cache Allocation Technology* (CAT) [89]. It allows the user or the operating system to dynamically control from which portion of the last-level cache an individual (logical) core can evict a cache line in order to replace it with a new one. Figure 4.8 illustrates the feature using a simplified example.

The user partitions the cache by writing a bitmask of  $N$  bits to a *machine-specific register* (MSR) of a core, where  $N$  depends on the processor model. Setting the bit at the  $i$ -th position of the bitmask means that the core can evict cache lines from the  $i$ -th portion of the LLC, while unsetting the bit at the  $i$ -th position means that the core never evicts cache lines from the  $i$ -th portion of the cache. By choosing distinct bitmasks, the user allows cores to evict portions of the cache exclusively. Bitmasks can be dynamically changed at runtime.

<sup>1</sup>Other manufacturers may implement similar functionality. AMD, for example, proposes a mechanism for controlling and monitoring the L3 cache [10] as well.

For example, the Intel Xeon E5-2699 v4 processor used in our experiments has a 20-way associative LLC with a size of 55 MiB. The bitmask for controlling the cache partitioning feature has a size of 20 bits. As a result, one portion of the cache equals  $55 \text{ MiB} / 20 = 2.75 \text{ MiB}$ . This means that setting, e.g., two bits in the bitmask, corresponds to a portion with a size of 5.5 MiB. Note that the processor allows up to 16 different bitmasks to be active at the same time [91].

The Linux kernel supports CAT since version 4.10 [93]. The extension allows the user to specify each core's bitmask used for cache partitioning by reading and writing to the pseudo file system `sysfs` instead of writing directly to an MSR of the processor. Furthermore, instead of specifying a bitmask for a core, the user has the option to specify a bitmask for a *process id* (PID) or a *thread id* (TID). This allows mapping a portion of the cache to an individual process or thread. During a context switch, the scheduler of the kernel is responsible for updating the bitmask of the core on which the process or thread is currently running.

#### 4.4.2 Cache Partitioning Scheme

To avoid cache pollution and to improve the performance of concurrent workloads with cache partitioning, we need to decide *how much cache* we need to allocate to an operator or a query, respectively. To that end, we can derive a cache partitioning scheme from the results of Section 4.3. The proposed cache partitioning scheme is the second contribution of this chapter.

**Column Scan.** Figure 4.3 shows that the performance of the *column scan* does not depend on the size of the cache because it does not reuse data and does not need to access the dictionary. However, the operation evicts lots of cache lines by continuously loading data from DRAM. Thus, we conclude that *column scans* will cause cache pollution for co-running queries. To avoid cache pollution, we give the *column scan* operator the smallest amount of cache (without reducing performance): 10 % of the LLC using the bitmask “0x3”.

**Aggregation With Grouping.** The results from Section 4.3.3 illustrate how *aggregation with grouping* can be highly sensitive to the size of the available cache, because it frequently accesses the dictionary and the hash table. Thus, we do not restrict the access to the LLC. We give



the *aggregation with grouping* operator the entire cache: 100 % of the LLC using the bitmask “0xffff”.

**Foreign Key Join.** Figure 4.7 demonstrates that the *foreign key join* is sensitive to the size of the cache depending on the cardinality of the primary keys: It causes cache pollution if the size of the bit vector is not comparable to the size of the cache; otherwise it becomes cache-sensitive. Consequently, we restrict the *foreign key join* operator to 10 % of the LLC using the bitmask “0x3” in the first case and to 60 % (throughput degrades below 35 MiB) using the bitmask “0xfff” in the other case. As a simple heuristic, we decide based on the size of the bit vector whether the operator is cache-sensitive or not.

**Bitmask “0x1”.** Note that we also evaluated the use of the bitmask “0x1” to restrict a scan-intensive operator. We observed, however, that this configuration degrades performance severely (not shown in Figure 4.3 to 4.7)—even for the *column scan*. We explain this behavior with the current implementation of CAT: Restricting access to a very small number of ways for an  $N$ -way set associative cache may result in significant contention.

### 4.4.3 Integration Into SAP HANA

The integration of a cache partitioning mechanism into a commercial database system is our third contribution of this chapter. To integrate cache partitioning into the execution engine of a prototype version of SAP HANA, we argue to leverage the Linux kernel interface of CAT. Directly using the hardware interface would require to either pin threads to cores with specific bitmasks or to track which thread is running on which core and to manually update a core’s bitmask upon thread migration. This would limit flexibility especially for changing workloads. A schematic overview of how we integrate cache partitioning into the execution engine of a prototype version of SAP HANA is illustrated in Figure 4.9.

The execution engine of SAP HANA uses a thread pool of worker threads called *job workers* to execute *jobs* [165]. A job encapsulates a single operator or—together in a group of jobs—a parallelized operator. Thus, a job represents one operator at the maximum. We implement cache partitioning for jobs to allow the engine fine-grained control over

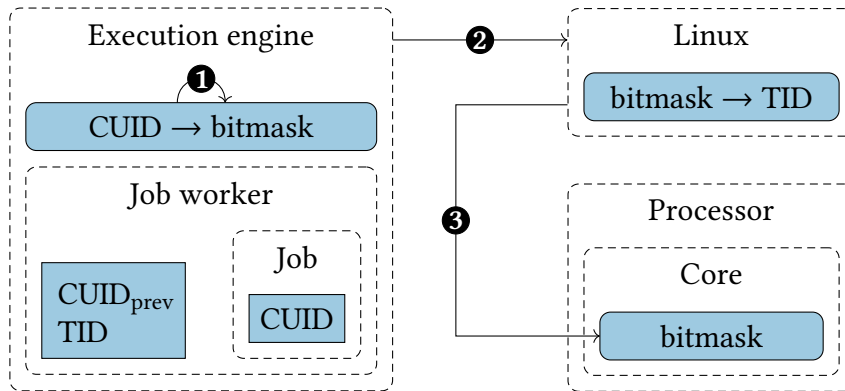


Figure 4.9: Schematic overview of the interaction between the execution engine of SAP HANA, the Linux kernel and the processor. The execution engine maps *jobs* to cache partitions by associating the *cache usage identifier* (CUID) of a *job* with a *bitmask* ①. Then, it passes the *thread id* (TID) of the *job worker* and the *bitmask* to the kernel ②. The kernel interacts with the processor to partition the cache ③.

the mechanism. This enables cache optimizations, e.g., *per operator*—similar to existing NUMA optimizations [165].

In fact, we annotate a job with information of its cache usage by associating it with a *cache usage identifier* (CUID). We currently distinguish between three categories: (i) jobs which are not cache-sensitive and pollute the cache such as the *column scan*; (ii) jobs which are cache-sensitive and profit from the entire cache such as *aggregation with grouping* (for most cases); and (iii) jobs such as the *foreign key join* which can be both cache-polluting and cache-sensitive depending on query predicates or table data. By default, a job belongs to (ii) to avoid performance regressions. The execution engine maps the CUID to a bitmask, following the heuristics described in Section 4.4.2 (“0x3” for (i); “0xfffff” for (ii); and “0x3” or “0xfff” for (iii)). The execution engine then passes the bitmasks to the Linux kernel.

Interacting with the Linux kernel to associate a thread with a new CAT bitmask might incur an overhead. Therefore, our implementation always compares old and new CUIDs and only interacts with the kernel if the job worker’s current CUID differs from the previous CUID. In practice, however, the overhead is negligible at least for OLAP scenarios. We benchmarked our test system and measured an overhead of less than 100  $\mu$ s. If at all, only short-running OLTP queries might see a small performance penalty due to the interaction with the kernel. However,

SAP HANA handles such queries in a dedicated thread pool anyway. That thread pool has always access to the entire cache.

## 4.5 Evaluation

The evaluation of the cache partitioning mechanism is the final contribution of this chapter. To evaluate the integration of cache partitioning into the system, we run experiments with workloads consisting of concurrent queries. We compare the performance with and without using cache partitioning. First, we describe the experimental setup. Then, we present the results of running the following workloads: We start by executing the same SQL queries as in our analysis of cache usage (cf. Section 4.3) using the same data sets. In particular, we run the *column scan* together with *aggregation with grouping*, and we run *aggregation with grouping* together with the *foreign key join*. Afterwards, we evaluate the cache partitioning feature using the TPC-H [197] benchmark and a mixed workload with a query extracted from a real-world SAP S/4HANA application.

### 4.5.1 Experimental Setup

We use the same hardware platform that we introduce in our analysis of cache usage (cf. Section 4.3). We implemented the cache partitioning feature in the execution engine of a prototype version of SAP HANA as described in Section 4.4.3.

**TPC-H.** We run each query of the TPC-H benchmark concurrently with the *column scan* using a generated data set of scale factor 100. Our goal is to study how a scan-intensive OLAP query (column scan), which causes cache pollution, impacts the performance of an individual TPC-H query and how cache partitioning can improve performance.

**SAP S/4HANA Workload.** SAP S/4HANA is an enterprise resource planning application commercialized by SAP. The *Universal Journal Entry Line Items* table ACDOCA is one of the central data stores for processing core financial aspects and is heavily used in both OLTP and OLAP query processing. Reflecting complex business logic, ACDOCA is a wide table with 336 attributes of type NVARCHAR (285) and DECIMAL (51). The instance of ACDOCA used in our experiments has 151 million rows and was extracted from a real customer system together with the most

frequent OLTP query: This query is executed more than 10 million times a week.

**Measurement Method.** In each experiment, we execute all SQL queries repeatedly for 90 seconds. This assures that each query is possibly affected by another query for the same time. For each query, we report the throughput of the query, when running concurrently to another query. We normalize the throughput of a query running concurrently to another query to the throughput of the query running in isolation. If not stated otherwise, we restrict the query causing cache pollution such as the *column scan* to 10 % of the LLC, while the other query can access the entire cache. Note that we tune the system for best throughput. This means that the memory-intensive workloads are limited by the available memory bandwidth of our test machine.

### 4.5.2 Column Scan & Aggregation With Grouping

We evaluate the impact on the throughput of the *column scan* and *aggregation with grouping* when we run both queries concurrently. We run three experiments to evaluate the impact of different dictionary sizes of the aggregated column. In each experiment, we vary the number of groups. We allocate 10 % of the cache to the *column scan* and 100 % to *aggregation*.

#### Dictionary Size of 4 MiB

Figure 4.10 visualizes the results for a dictionary size of 4 MiB. We observe that the throughput degrades with increasing group sizes. If we increase the group size from  $10^4$  to  $10^5$ , throughput of the *aggregation* query drops from 80 % to 66 %. If we increase the group size from  $10^5$  to  $10^6$ , the throughput of the *scan* query drops from 89 % to 69 %. As the number of groups increases, *aggregation with grouping* uses larger hash tables to store temporary results. This heavily impacts the cache usage of the operator. Up to a group size of  $10^4$ , the hash table has the size of only a fraction of the LLC. It mostly fits in the L2 cache. Thus, the *aggregation* is not sensitive to the capacity of the LLC and cache pollution is not a problem.

If we increase the number of groups to  $10^5$ , the size of the hash table is comparable to the size of the LLC. The *column scan* evicts an increasing number of cache lines used by the *aggregation*, thereby causing cache pollution. When aggregating over  $10^6$  groups, the situation

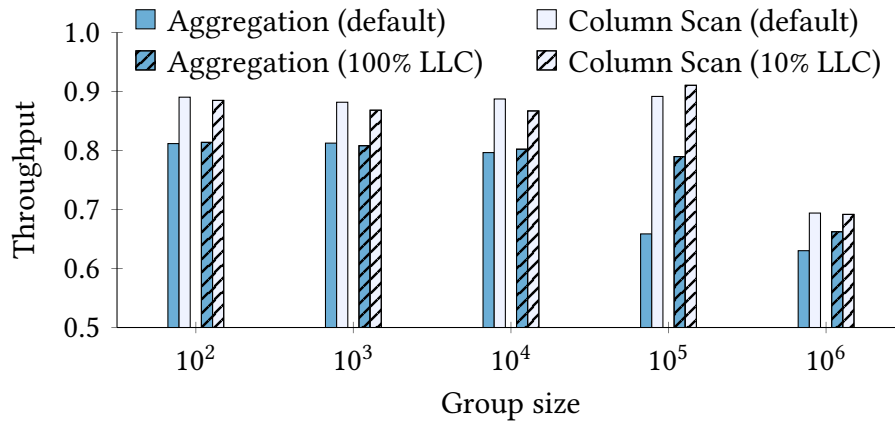


Figure 4.10: Normalized throughput of the *column scan* and *aggregation with grouping* when executed concurrently. We set the dictionary size of the aggregated column to 4 MiB and use cache partitioning: Throughput degrades with increasing group sizes. For 10<sup>5</sup> groups cache partitioning significantly improves throughput.

changes: The size of the hash table exceeds the size of the LLC. The *aggregation* query performs more DRAM accesses and uses more memory bandwidth. As a result, both queries increasingly compete for memory bandwidth, which explains why the throughput of the *scan* query degrades more strongly.

The results show that enabling the cache partitioning feature of the execution engine significantly improves performance of *aggregation with grouping* when the algorithm becomes sensitive to the capacity of the LLC (for a group size of 10<sup>5</sup>). By giving the *aggregation* the entire cache and the *column scan* only a small portion of the cache, we improve throughput by 20%. At the same time, the throughput of the *column scan* improves by 3%.

The performance improvement also correlates with hardware metrics, which we collected by sampling hardware performance counters for the entire system. The cache hit ratio increases from 0.78 to 0.82, while the LLC misses per instruction improve from  $2.86 \cdot 10^{-3}$  to  $2.32 \cdot 10^{-3}$ . Thus, partitioning the cache avoids cache pollution and improves the overall cache efficiency of the workload.

### Dictionary Size of 40 MiB

Figure 4.11 visualizes the results for a dictionary size of 40 MiB. The results show that the throughput of the *aggregation* query drops below

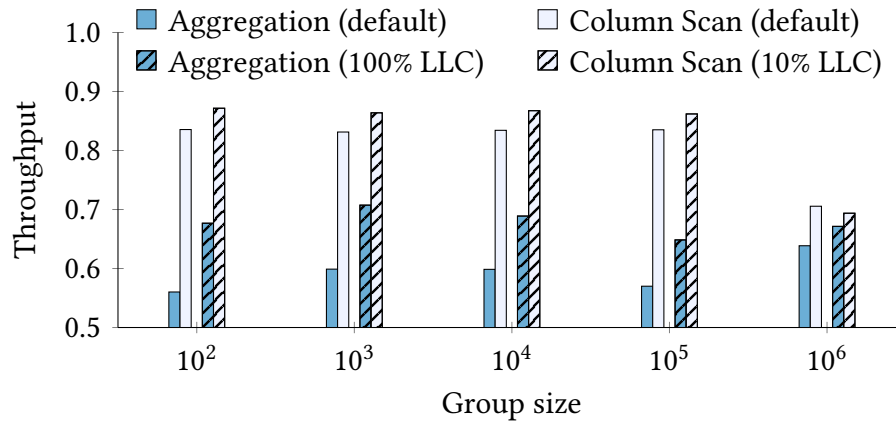


Figure 4.11: Normalized throughput of the *column scan* and *aggregation with grouping* when executed concurrently. We set the dictionary size of the aggregated column to 40 MiB and use cache partitioning: Throughput of *aggregation with grouping* degrades significantly. Cache partitioning improves throughput by up to 21 %.

60 % for up to 10<sup>5</sup> groups. At the same time, the throughput of the *column scan* query drops to 84 %. If we increase the number of groups from 10<sup>5</sup> to 10<sup>6</sup>, the throughput of the *aggregation* degrades less, but the throughput of the *column scan* degrades more.

By utilizing cache partitioning, we improve the throughput of the *aggregation* query by up to 21 %. At the same time, we improve the throughput of the *column scan* by up to 6 %. Reserving 90 % of the cache exclusively for the *aggregation* query allows the entire dictionary to be kept in the cache, as long as the hash table does not exceed the size of the LLC (i.e., up to 10<sup>5</sup> groups). Otherwise, the dictionary and the hash table compete for cache capacity. In case of 10<sup>6</sup> groups the improvements from cache partitioning are significantly smaller, because the capacity of the cache is not enough to hold the dictionary and the large hash table.

We determine that the overall cache hit ratio increases and that the LLC misses per instruction decrease because the *aggregation* has to perform fewer accesses to main memory. In addition, the *column scan* gets more memory bandwidth. By partitioning the cache, the database system uses hardware resources more efficiently and executes both queries faster.

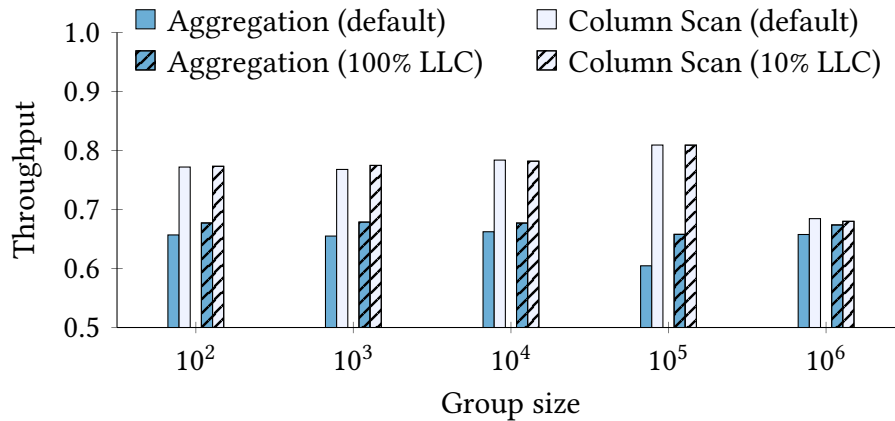


Figure 4.12: Normalized throughput of the *column scan* and *aggregation with grouping* when executed concurrently. We set the dictionary size of the aggregated column to 400 MiB and use cache partitioning: Throughput of both queries degrades significantly. Cache partitioning improves throughput by up to 9%.

### Dictionary Size of 400 MiB

Figure 4.12 visualizes the results for a dictionary size of 400 MiB. We observe that, when the dictionary is several times larger than the cache, the throughput of the *aggregation* query decreases to 60–66%. At the same time, the throughput of the *column scan* query decreases to 68–81%, which is more significant than in the previous two experiments.

This illustrates that both queries compete less for the LLC but more for memory bandwidth. The dictionary and the hash table cannot be kept in the cache at the same time. Thus, the *aggregation* algorithm performs more memory accesses to DRAM—independent of the group size. It consumes more memory bandwidth and impacts the *column scan* query more strongly. At the same time, the *aggregation* is less sensitive to the cache size, which explains why cache partitioning improves the throughput of the *aggregation* query only by 3–9%.

### 4.5.3 Aggregation With Grouping & FK Join

We evaluate the impact on the throughput of *aggregation with grouping* and the *foreign key join* when we run both queries concurrently. We run two experiments with a different number of primary keys in the data set for the *foreign key join*: This changes the size of the bit vector used by the algorithm. In each experiment, we vary the number of

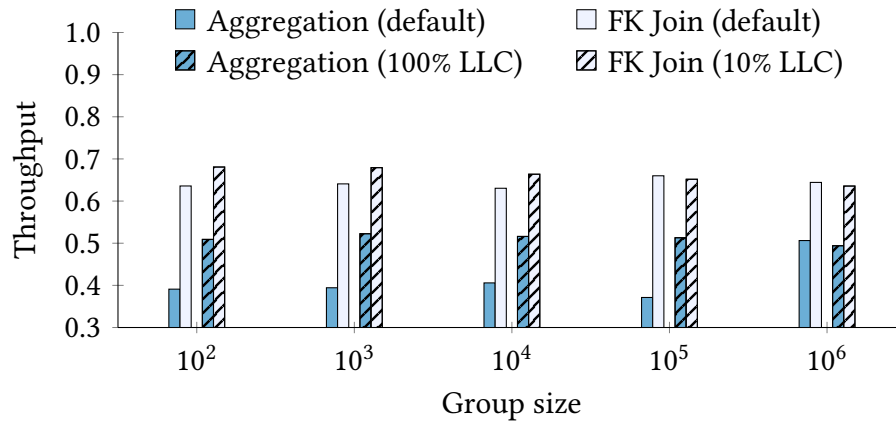


Figure 4.13: Normalized throughput of *aggregation with grouping* and the *foreign key join* when executed concurrently. We set the dictionary size of the aggregated column to 40 MiB, set the number of primary keys to 10<sup>6</sup>, and use cache partitioning: Partitioning improves throughput by up to 38 %.

groups for *aggregation with grouping*. We evaluate two different cache partitioning schemes: First, we allocate 10 % of the cache to the *foreign key join*. Second, we allocate 60 % of the cache to the *foreign key join*. We always allocate 100 % of the LLC to *aggregation with grouping*.

### 10<sup>6</sup> Primary Keys

Figure 4.13 visualizes the results for 10<sup>6</sup> primary keys. We observe that for group sizes below 10<sup>6</sup> the throughput of the *aggregation* query drops to 41 %, while the throughput of the *join* query drops to 63 %. If we increase the group size to 10<sup>6</sup>, the throughput of the *aggregation* decreases to 51 %. The results match previous results presented in Section 4.5.2, but in this workload the performance of both queries suffers more.

The results show that enabling cache partitioning improves the throughput of the *aggregation* query by up to 38 %. At the same time, the throughput of the *join* query improves by up to 7 %. Note that the cache hit ratio increases from 0.55 to 0.67, while the LLC misses per instruction improve from  $2.26 \cdot 10^{-3}$  to  $1.93 \cdot 10^{-3}$  for, e.g., a group size of 10<sup>3</sup>. Thus, we conclude that partitioning the cache improves the overall cache efficiency of the workload. If the group size is 10<sup>6</sup>, both operators are not limited by LLC contention. Consequently, partitioning the cache does not improve performance.



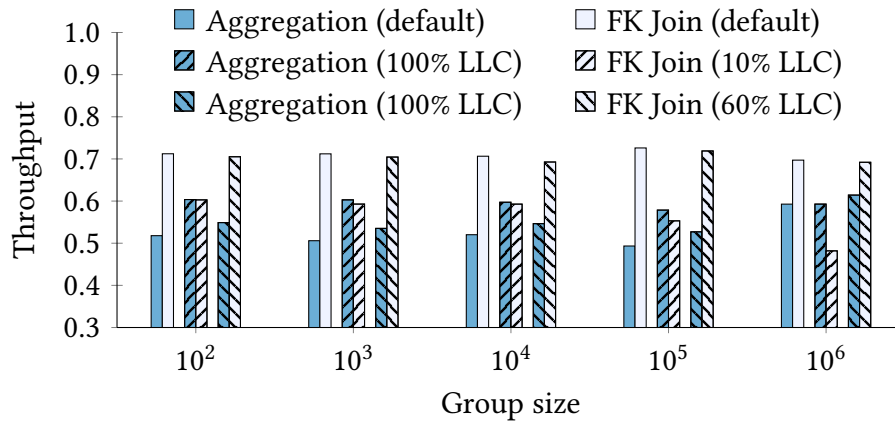


Figure 4.14: Normalized throughput of *aggregation with grouping* and the *foreign key join* when executed concurrently. We set the dictionary size of the aggregated column to 40 MiB, set the number of primary keys to  $10^8$ , and use cache partitioning: Restricting the *foreign key join* to 10 % of the LLC does *not* improve performance, while restricting the operator to 60 % improves throughput by up to 8 %.

### 10<sup>8</sup> Primary Keys

Figure 4.14 visualizes the results for  $10^8$  primary keys. We observe that the throughput of the *aggregation* query drops to 49 %, while the throughput of the *join* query drops to 70 %. By partitioning the cache with the configuration that gives *aggregation with grouping* the entire cache and the *foreign key join* only 10 % of the cache, we improve the throughput of *aggregation* by up to 19 %. However, the throughput of the *join* query worsens by 15–31 %. In total, we lose more than we gain from applying this cache partitioning scheme.

This observation is consistent with the results from Section 4.3.4: The throughput of the *join* query decreases if the cache size falls below 35 MiB. Thus, we need to partition the cache differently. We allow the *aggregation* query to evict cache lines from the entire cache, while we restrict the *join* query to 60 % of the cache. This means that we allocate 40 % exclusively to the *aggregation* query, while 60 % of the cache is shared between both queries. We prioritize the *aggregation* operator over the *join* operator because it needs more cache and suffers more from cache conflicts.

The results show that this configuration improves the throughput of the *aggregation* query by up to 9 %. At the same time, the throughput of the *join* query varies around 2 %. When we consider the combined

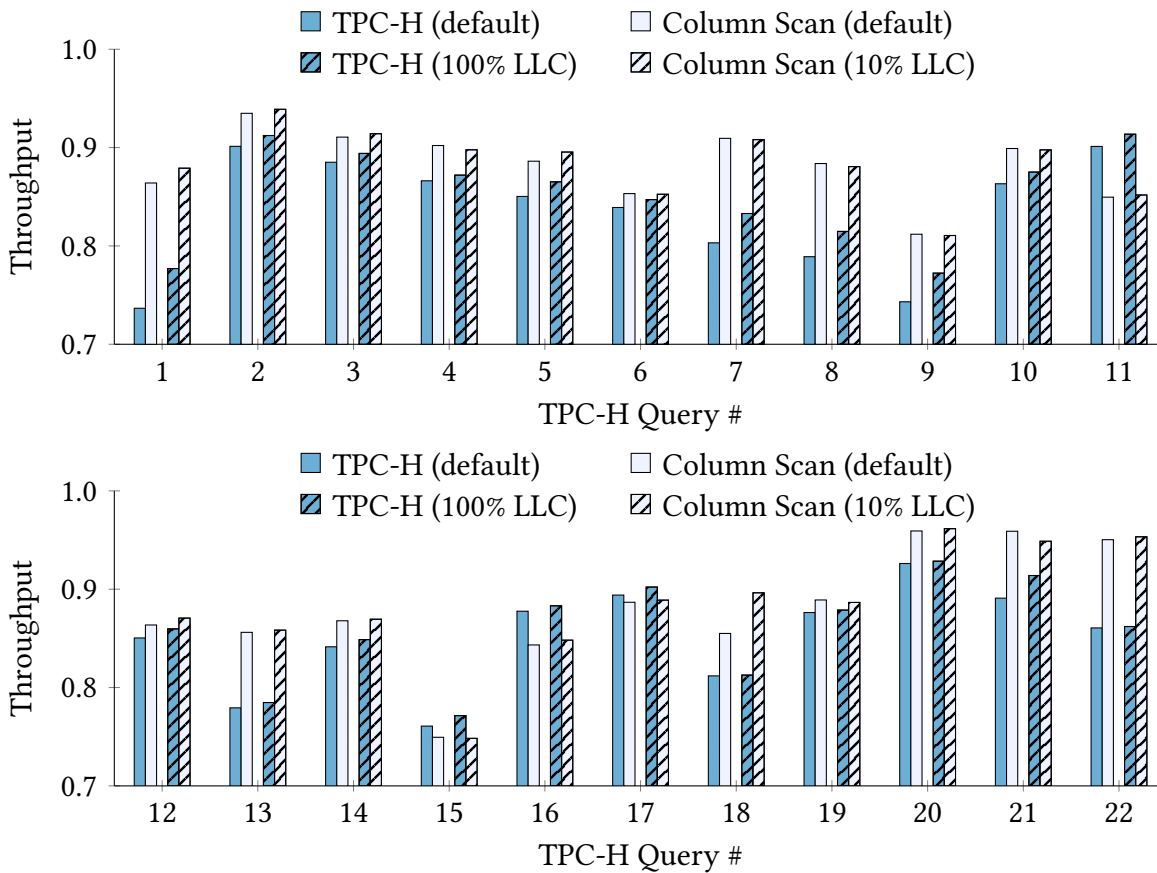


Figure 4.15: Normalized throughput of the *column scan* and each TPC-H query when executed concurrently. We disable or enable cache partitioning: Cache pollution impacts especially TPC-H Queries 1, 7, 8 and 9, while TPC-H Queries 10 to 22 suffer less from cache pollution. Partitioning the cache improves throughput by up to 5%.

throughput, the second cache configuration improves the overall performance of the workload.

#### 4.5.4 Column Scan & TPC-H Queries

We evaluate the impact on the throughput of the *column scan* and on the throughput of each query of the TPC-H benchmark when we run both queries concurrently. We allocate 10% of the cache to the *column scan* and 100% of the LLC to a query of the TPC-H benchmark. Figure 4.15 visualizes the results.

We observe that the impact of the *column scan* on a co-running query varies significantly with the TPC-H queries. The throughput of

the TPC-H queries degrades to 74–93 %, while the throughput of the *column scan* query degrades to 65–96 %. If we enable cache partitioning, we improve the throughput of the TPC-H queries by up to 5 %. The results show that for TPC-H Queries 1, 7, 8 and 9 the cache partitioning approach improves the overall performance of the workload. For other queries the improvements are less noticeable.

This shows that only some queries depend on the LLC. That is because the columns of the TPC-H data, which are aggregated, feature comparatively small dictionaries. Furthermore, grouping usually uses only a relatively small number of groups. Thus, most of the frequently accessed data structures are small enough to fit in L2 caches or in a small portion of the LLC.

One exception is the column `l_extendedprice` with a dictionary size of approximately 29 MiB, which is frequently accessed during the execution of, e.g., TPC-H Query 1. The query aggregates the column causing lots of accesses to the dictionary, which explains why reducing cache pollution through cache partitioning improves its performance.

Interestingly, the avoidance of cache pollution sometimes reflects back on the *column scan* operator. Faced with fewer cache misses, the co-running TPC-H query takes away less bandwidth from the bandwidth-sensitive *scan*, resulting in a throughput increase of up to 5 % for the *column scan* query (e.g., with TPC-H Query 18 co-running).

#### 4.5.5 Column Scan & OLTP Query

We evaluate the impact on the throughput of the *column scan* and on the throughput of an *OLTP* query from the SAP S/4HANA workload when we run both queries concurrently. We use the original query, which contains a projection to 6 columns with small dictionaries. To analyze the impact of the dictionary sizes, we use a modified version of the *OLTP* query, which contains a projection to 13 columns with big dictionaries. We allocate 10 % of the cache to the *column scan* and 100 % of the LLC to the *OLTP* query. Figure 4.16 visualizes the results.

The results show that the throughput of the *OLTP* query drops to 66 % (projection of 13 columns) and 68 % (projection of 6 columns). The throughput of the *column scan* decreases to only 95 % and 96 %, respectively. Restricting the *column scan* to 10 % of the cache improves the throughput of the *OLTP* query by 13 % and 9 %, respectively.

During the processing of the *OLTP* query, the engine accesses the inverted index of five columns that are part of a primary key. Afterwards, it projects the selected rows to 13 (or 6) columns causing accesses to

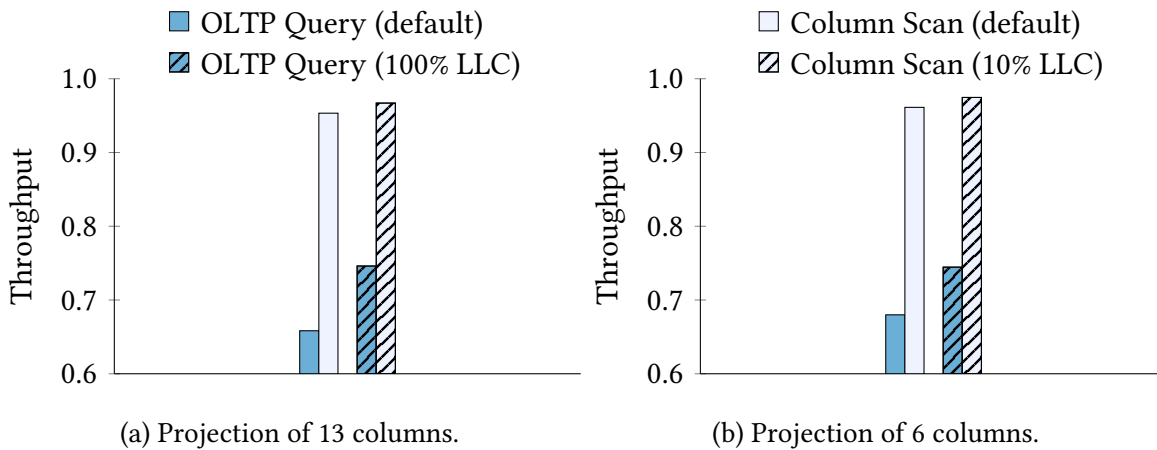


Figure 4.16: Normalized throughput of the *column scan* and an *OLTP* query from the S/4HANA workload when executed concurrently. We vary the number of projected columns for the *OLTP* query and disable or enable cache partitioning: Partitioning improves the throughput of the *OLTP* query by 13 % and 9 %, respectively.

the dictionaries of the columns. We argue that, for a larger number of projected columns, cache partitioning improves the performance more significantly because more dictionaries need to be kept in the cache to avoid cache misses.

An additional experiment (not shown) further demonstrates that the size of the working set, i.e., the size of the dictionaries and indices, affects the cache-sensitivity of the *OLTP* query: We varied the number of projected columns from 2 to 13 (featuring the biggest dictionaries) and observed that the throughput degrades with an increasing number of projected columns. Restricting the *column scan* to 10 % of the cache improves the throughput of the *OLTP* query by 8 % to 13 %.

#### 4.5.6 Discussion

Our evaluation confirms that *aggregations* are sensitive to *cache pollution* either caused by *column scans* or *joins*. *Aggregations* are most sensitive to cache pollution whenever the size of their performance-critical data structures is comparable to the size of the LLC. The *column scan* operator always pollutes the cache because it continuously evicts cache lines and does not reuse data. The *join* operator, on the other hand, only causes cache pollution whenever its frequently accessed data structures either fit in the L2 cache or exceed the LLC. Otherwise, the *join* operator is cache-sensitive.

In the case the *column scan* or the *join* operator cause cache pollution, we can eliminate cache pollution and significantly improve performance by restricting the *column scan* or *join* to a small portion (10 %) of the LLC. In addition, the *column scan* operator profits from the fact that *aggregation* consumes less memory bandwidth: The throughput of the *column scan* increases, too.

If the size of the data structures used by a *join* is comparable to the size of the LLC, the *aggregation with grouping* operator and the *join* operator compete for cache capacity. Thus, we restrict the *join* to 60 % of the cache, but observe that performance improves only slightly. Generally, the search for the “best” partitioning in any given situation will depend on accurate *result size estimates*.

Furthermore, we performed experiments using the *TPC-H benchmark* to evaluate whether limiting the LLC for *scans* improves the performance of OLAP workloads. Our measurements show that the performance of TPC-H Queries 1, 7, 8 and 9 improves from partitioning the cache because these queries frequently access a column with a large dictionary. The performance of the other queries, however, does not improve noticeably. This demonstrates that not all queries are sensitive to cache pollution: The size of the working set, affecting, e.g., dictionary and hash table sizes, determines if the performance of an operator depends on the size of the available LLC.

Finally, we ran experiments with the *column scan* (OLAP) and an OLTP query from a real-world application. The results demonstrate that the performance of the OLTP query degrades significantly in the base configuration. This is because OLTP queries tend to use dictionaries aggressively, which the OLAP query evicts from the cache. By using cache partitioning to restrict the OLAP query to a small portion (10 %) of the cache, we avoid the eviction of dictionaries and improve the performance of the OLTP query.

The results illustrate that our simple approach for avoiding *cache pollution* is effective in improving overall system throughput. Thus, we propose to restrict scan-intensive operators which do not profit from using the LLC, such as the *column scan*, to a minimum portion of the cache. This approach has the advantage that it can improve the performance of *any* concurrent workload containing a scan-intensive operator. It does not depend on further knowledge of the workload. In addition, our results demonstrate that we can apply cache partitioning without introducing performance regressions.

## 4.6 Related Work

### Cache-Aware & Cache-Oblivious Algorithms

A plethora of research efforts from the database community focuses on developing and optimizing *cache-aware* database algorithms [19, 127, 140], which exploit the cache hierarchy of modern processors. They usually depend on setting the correct hardware parameters. Others propose *cache-oblivious* algorithms [23, 45, 66, 82], which achieve cache efficiency independent of specific hardware parameters. However, by design, both groups of algorithms are sensitive to the cache and thus sensitive to cache pollution. Thus, we expect our approach to benefit both groups of algorithms.

### Scheduling

Zhuravlev et al. [215] and Lee et al. [111] propose to mitigate contention for shared resources on multicore processors via thread scheduling. They schedule tasks on the same core (or a group of cores) if they do not cause conflicts. Their approach allows an application to avoid cache *thrashing* caused by frequent context switches and cache *pollution* caused by other tasks. However, their approach may cause scheduling delays or underutilization of processor cores if the workload does not consist of a sufficient number of tasks. In particular, thread scheduling cannot avoid cache pollution if all tasks need to run concurrently on multiple cores sharing a cache. To avoid cache pollution not only for the last-level cache but also for the L1 and L2 cache of a processor, one could combine their approach and our approach to avoid cache thrashing of the L1 and L2 caches via scheduling and cache pollution of the L3 cache via cache partitioning.

### Page Coloring

Plenty of work from the systems community focuses on (software-based) cache partitioning based on *page coloring* [39, 191, 213]. The goal is to either statically or dynamically partition the cache among competing threads to improve resource utilization or guarantee quality of service. Among others, Soares et al. [186] specifically aim to avoid *cache pollution*. They propose a dynamic mechanism, which first characterizes an application's cache behavior using hardware performance counters. Then, it maps the memory pages of applications with

high cache miss rates to dedicated cache sets to avoid polluting an application's memory pages with low cache miss rates.

We derive the cache usage of the database system's operators from an experimental evaluation. However, we argue that determining cache usage based on hardware performance counters, could also be applied to cache partitioning with CAT. To map memory access statistics back to machine instruction, i.e., database operators, efficient memory tracing (cf. Chapter 3) may be used.

Lee et al. [111] build on the results from the systems community and present a method for minimizing last-level cache conflicts for PostgreSQL. They demonstrate a cache allocation mechanism based on *page coloring* to avoid cache capacity conflicts and classify queries based on their data locality and cache sensitivity. They mainly focus on a hash join and an index nested loop join. In addition, they evaluate a disk-based DBMS, which uses a memory buffer pool to keep a portion of the data in main memory. For such a system, allocating and copying memory—necessary to (re-)partition the cache via *page coloring*—is potentially less performance-critical compared to an in-memory database system.

Wang et al. [201] make use of page coloring to optimize the cache usage of key-value stores by storing keys and values separately and with different page colors in memory. They claim that their approach has low re-partitioning costs. However, they focus on relatively stable workloads, where frequent re-partitioning is not needed. In addition, they are evaluating a mixture of different optimizations, which makes it difficult to judge how cache partitioning improves performance.

### Cache Partitioning With Hardware Support

In contrast to software-based cache partitioning, Chiou et al. [38] or Qureshi et al. [166] propose cache partitioning with hardware support by restricting cache line replacement to a certain way for an  $n$ -way associative cache. Similarly, Herdrich et al. [84] introduced two technologies addressing quality of service on Intel's multicore server platforms: Cache Monitoring Technology (CMT) and Cache Allocation Technology (CAT). They highlight the benefit of partitioning the LLC using the SPEC CPU2006 benchmark, network communications, and the STREAM benchmark.

Lo et al. [123] use cache partitioning based on CAT, among other mechanisms for resource isolation, to manage resource sharing between latency-critical services and best-effort batch tasks in workloads from

Google. They claim to increase hardware utilization without violating service-level objectives.

We use cache partitioning based on CAT as well. However, we analyze the cache usage of database operations, we demonstrate how to integrate the mechanism into a commercial database system, and we analyze whether the integration effort pays off.

### Deriving a Cache Partitioning Scheme

While we derive the cache partitioning scheme from an experimental analysis, the application of existing characterization methods for *describing* the cache usage pattern of a database operator could be investigated. For instance, Chou and DeWitt [41] propose the query locality set model based on the knowledge of the various patterns of queries to allocate buffer pool memory efficiently. Others propose cache miss ratio (curves) as an online model for characterizing workloads or operators [126, 191, 214]. Combining these techniques with efficient memory tracing (cf. Chapter 3) could help to determine an application's cache characteristics—similar to work from Xiang et al. [208].

### Hardware Design

Borkar and Chien [31] analyze the performance growth of microprocessors and predict future trends for processor designs. They highlight that the performance growth driven by Moore's Law will slow down in the future because of diminishing transistor-speed scaling and practical energy limits. In particular, they argue that hardware manufactures have to adapt their architectures by investing in heterogeneity, application-specific hardware and dynamically customizable logic, while software developers have to exploit parallelism and utilize new hardware more carefully. They point out the design decision of increasing cache sizes and question if increasing caches sizes can still improve performance.

Our results show that some database operations, i.e., the *column scan* and the *foreign key join*, do not profit from a large LLC. Thus, application-specific hardware or dynamically customizable logic, such as an FPGA [99, 110], could be built for these types of operations with a smaller cache to save costs and power, or to invest in more compute logic instead.



## 4.7 Conclusion

In modern microprocessors, multiple processor cores share the same last-level cache. Conflicts over the shared cache can significantly degrade performance of concurrent workloads, whenever the execution of one operation suffers from *cache pollution* caused by the execution of another operation.

In this chapter, we confirm through an experimental analysis that important in-memory database operators exhibit different performance characteristics depending on the available cache size. We analyze the cache usage of the *column scan*, *aggregation with grouping*, and the *foreign key join* operator with different data sets. Based on the results of the analysis, we derive a cache partitioning scheme that we deliberately kept simple: We restrict memory-intensive operators that do not reuse data to a small portion of the cache.

Furthermore, we demonstrate how to integrate cache partitioning into the execution engine of an existing DBMS with low engineering costs. Our evaluation shows that our approach avoids cache pollution and significantly reduces cache misses. We demonstrate that, by partitioning the cache for concurrent queries, we can improve the overall system performance by up to 38%. In particular, we showcase improvements for custom queries targeting *column scans*, *aggregations*, and *joins*, as well as for the TPC-H benchmark and an OLTP query of a modern HTAP business application. Ultimately, our results show that integrating cache partitioning into a DBMS is worth the effort: It may improve but never degrades performance for arbitrary workloads containing scan-intensive, cache-polluting operators.



# 5

## Shared Loading

Bulk loading large volumes of data into the optimized storage of a database system is a performance-critical task. Depending on the storage layout, bulk loading may entail complex data transformations, which makes bulk loading an expensive task that can disturb other workloads running in parallel.

In this chapter, we demonstrate that data transformations dominate the cost of bulk loading by using the example of SAP HANA, a commercial, in-memory columnar system with a compression-optimized storage. We show that data transformations may cause resource contention on a stressed system, resulting in poor and unpredictable performance for both bulk loading and query processing. To mitigate this problem, we propose *Shared Loading*, a distributed bulk loading mechanism that enables dynamically offloading deserialization and data transformation to the machine where the input data resides. In the evaluation we demonstrate that, for different network bandwidths and data sets, *Shared Loading* accelerates bulk loading into compression-optimized storage and improves the performance and predictability of queries running concurrently.

In particular, we make the following contributions: (i) we analyze where time is lost in a complete bulk loading pipeline; (ii) we present the architecture of the distributed bulk loading mechanism *Shared Loading*, which can dynamically offload work to the client machine; and (iii) we evaluate the performance characteristics of our approach by studying whether our approach improves bulk loading throughput or the tail latency of queries running in parallel to bulk loading.

**Outline.** We introduce the problem in Section 5.1. In Section 5.2, we analyze the costs of a complete bulk loading pipeline. We present the architecture of our distributed bulk loading mechanism *Shared Loading* in Section 5.3. In Section 5.4, we present and discuss the results of the experimental evaluation. We cover related work in Section 5.5 and we conclude the chapter in Section 5.6.

Parts of this chapter are published in [150].

## 5.1 Introduction

In today's heterogeneous system landscape, an ever-growing volume of data is available in plain text files. Such files are widely supported because they are readable by both machines and humans. Popular formats include delimiter-separated values files, such as CSV, fixed-width values files, JSON, or XML. Plain text files are frequently used to transfer scientific data sets [190] or business data, to facilitate replication and system integration, or to migrate to a new system. In the latter case, customers of SAP state that the bulk loading of text files can quickly become the bottleneck in mission-critical migration processes. Thus, fast and efficient bulk loading is imperative.

Related work [3, 6, 37, 100, 153] assumes that data resides on local storage where the DBMS is installed on. However, files are often stored close to the *client* machine, where an application produces data or data is preprocessed, and not close to the *server* machine running the DBMS. Thus, data needs to be transferred over the *network*. This can be a fast, internal network, in case of a cloud-only or an on-premise scenario, or a slow Internet connection, e.g., when loading data from an on-premise setup into the cloud.

Additionally, direct access to the server might be impossible due to security policies or conditions of use. This is often the case in a cloud environment. For example, Amazon's database web service grants only limited access because the entire system is maintained by Amazon. Such a situation makes it very difficult to load local files from the server. For instance, Amazon recommends users to load data into MySQL over the network via CSV files and to split large files manually, to stop all applications, and to consider disabling automatic backups for better performance [8].

Furthermore, related work [6, 137] primarily focuses on optimizing parsing and the creation of an index during bulk loading. They conclude

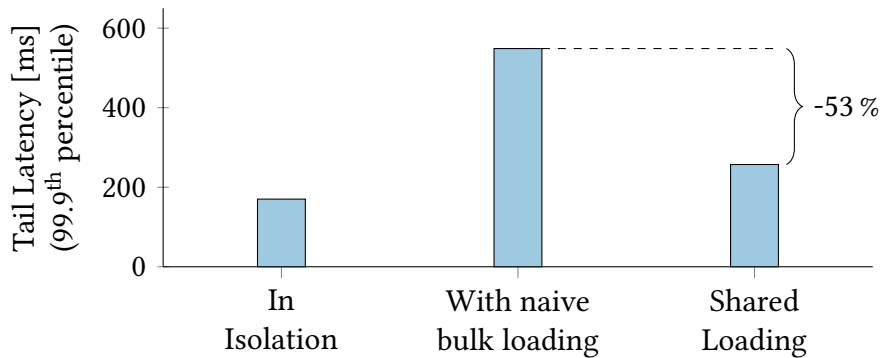


Figure 5.1: Tail latency of query workload when executed in isolation, in parallel to naive bulk loading, and in parallel to bulk loading with *Shared Loading*. Tail latency degrades by a factor of 3.2 under load. *Shared Loading* improves tail latency by 53 %.

that most time is lost on deserialization. However, modern systems [29, 62, 105, 109, 161] employ a (highly) compressed storage. In such a system, it is also a challenge to *transform* data because compression is resource-intensive. In fact, we demonstrate in our analysis of a commercial database system that most time is lost on data transformation—not on deserialization. In addition, such transformations may take away precious CPU cycles from queries running in parallel.

Figure 5.1 illustrates this problem. We execute a simple analytical workload both in isolation and in parallel to the bulk loading of the `lineitem` table of the TPC-H benchmark. Our results demonstrate that the query performance and predictability of the analytical workload degrades significantly when the query workload is executed in parallel to bulk loading: Tail latency degrades by a factor of 3.2.

To address these issues, we study bulk loading in a distributed environment and its impact on query workloads running in parallel. Ultimately, we develop a new mechanism for efficient bulk loading into optimized storage. Our mechanism accelerates bulk loading and improves the performance and predictability of concurrent query processing. Its architecture allows dynamically adapting data transformations and shifting work between client(s) and a server *at loading time*—without the need for the user to partition the input data or to manually parallelize bulk loading [8, 133, 154].

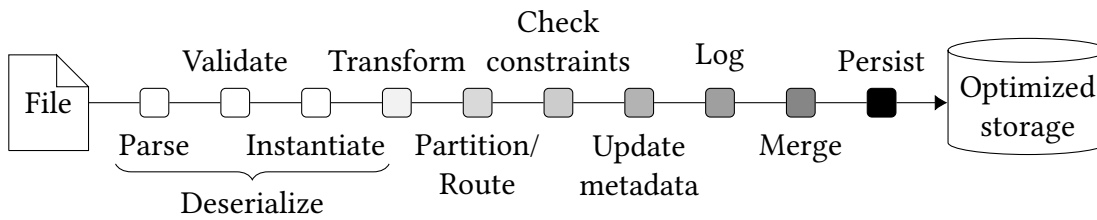


Figure 5.2: Processing steps for bulk loading data from a file into optimized storage.

## 5.2 Cost Analysis of Bulk Loading Pipeline

To identify optimization opportunities, we analyze where time is lost in a complete bulk loading pipeline using SAP HANA as an example. The cost analysis is the first contribution of this chapter. In particular, we break down the processing time. Our goal is to identify the individual steps that are the most time consuming. First, we give an overview of the processing steps of bulk loading. Then, we demonstrate where time is lost during bulk loading.

### 5.2.1 Overview of Loading Steps

Figure 5.2 presents the individual processing steps of the bulk loading pipeline. Note that steps might be interleaved, ordered differently, or omitted depending on the actual implementation used by the DBMS. Conceptually, the bulk loading operation involves the following steps:

Bulk loading starts with the *deserialization* of the file. For a delimiter-separated values file, it *parses* the file contents in search for symbols marking the end of a row (e.g., `'\n'`) or the end of a column (e.g., `'|'`) in order to split the character stream into individual fields of the input table. It *validates* whether a value conforms to the specification of the SQL type given by the schema. If the validation succeeds, the operation creates an *instance* of the SQL data type in memory; otherwise it handles the error, e.g., by collecting discarded rows or by aborting.

In addition, the system *transforms* new data into the physical storage layout, such as a row-based or a columnar representation. This might involve complex restructuring and different compression methods. In SAP HANA, new records migrate from a compressed, write-optimized storage to a compressed, read-optimized storage (cf. Section 2.2.2). If the target table is partitioned, each row needs to be assigned to its correct *partition*. If the system is distributed, it might be necessary to *route* data over the network to another node. The system also checks

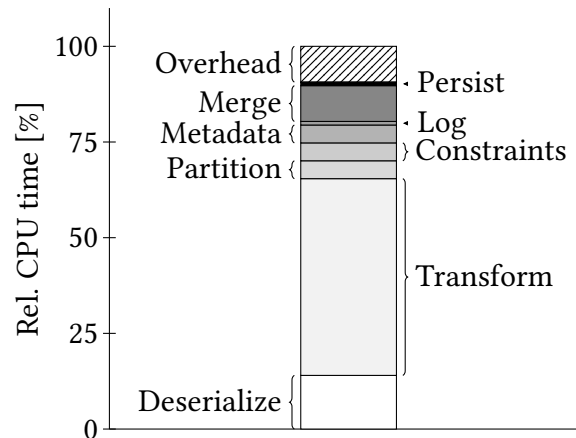


Figure 5.3: Cost analysis of bulk loading into SAP HANA. Most CPU time is spent on data transformation.

*constraints*, such as the absence of null values, uniqueness and primary key constraints, or the referential integrity of foreign keys, and *updates metadata*, such as indices or statistics. Ultimately, it writes a *log* to persistent storage to assure durability. Systems maintaining a read-optimized storage additionally *merge* new data (periodically) into their optimized storage. To speed up recovery, the optimized storage may be written to *persistent* storage.

### 5.2.2 Costs of Loading Steps

To analyze where time is lost, we bulk load the `lineitem` table of the TPC-H benchmark from a local solid-state drive into SAP HANA. The experimental setup is described in Section 5.4.1. In particular, we leverage SAP HANA’s built-in statement for bulk loading CSV files “`IMPORT * FROM CSV [...]`”. The experimental results are shown in Figure 5.3. The figure illustrates the relative CPU time of each processing step of the complete bulk loading pipeline. While the results are specific to the data set and the implementation of SAP HANA, we expect similar results for other systems with (complex) data transformations—especially for systems with a compressed storage. In addition, we expect similar results for other plain text file formats because the file format affects only the parsing step.

The results demonstrate that data transformations consume 55% CPU time. This includes the time it takes to insert new rows into the write-optimized storage and to compute a compressed, columnar in-

memory representation. Deserialization consumes around 15 % CPU time. Checking constraints, partitioning the table, or updating meta data requires only a small amount of CPU time. Merging the write-optimized storage into the read-optimized storage consumes 10 % CPU time. Logging and persisting consume a negligible amount of CPU time due to asynchronous I/O. The remaining 10 % are overhead from the transaction manager, lock handling, and memory management.

In summary, our results differ from previous results [6, 37, 137] that attribute the highest cost for bulk loading to deserialization. For compression-optimized systems, such as SAP HANA, the cost of *transforming* data dominates computing time and outweighs the cost of deserialization by a factor of 3.7. Due to the high cost of compression, we expect that the cost of deserialization outweighs the cost of transforming data also in other systems that apply compression not only at column, but at row or page level.

### 5.3 Shared Loading

Bulk loading and concurrently running queries compete for hardware resources. The result is poor loading throughput *and* poor query performance. However, we can exploit that the input data of bulk loading is often stored close to the *client* machine and not close to the *server* machine running the DBMS. Thus, to address the problem of resource contention, we propose to *offload* part of the bulk loading to the client machine. In particular, the client provides additional hardware resources and allows accessing data more efficiently because it is closest to the data source. The advantages are twofold: By distributing the load efficiently, we avoid wasting resources that may be idle otherwise, and we mitigate a high system load on the server. The design of the bulk loading architecture *Shared Loading* is the second contribution of this chapter.

We argue that offloading work needs to be done *dynamically* depending on, e.g., the input data, the compute power of client and server, or the available network bandwidth. Our cost analysis of the bulk loading pipeline identifies which steps are worth offloading: deserialization and data transformation. To that end, we propose the architecture of a distributed bulk loading mechanism that enables offloading deserialization and data transformation to the client *at loading time*.

We assume that the input file is a delimiter-separated values file such as CSV. However, by adjusting the parsing step of the deserialization of



the file, our approach may support other flat file formats. We use order-preserving dictionary compression in a column store (cf. Section 2.2.1) as an example. The concept of dynamically offloading deserialization and data transformation may be applicable to other storage formats and compression techniques as well. In addition, we propose that the client transforms data *towards* the storage format—not into a data format that can directly be written to disk. This allows the server to ingest data with little effort, but the server still needs to validate all client data.

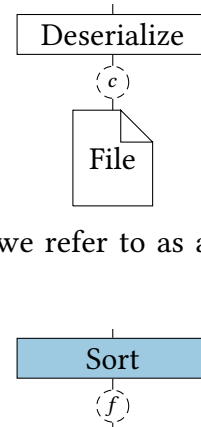
Figure 5.4 gives an overview of the data flow and the processing steps on client and server. First, we introduce client-centric loading shown in Figure 5.4a. Afterwards, we present server-centric loading shown in Figure 5.4b. Subsequently, we describe how we can combine both approaches dynamically. Finally, we highlight implementation details.

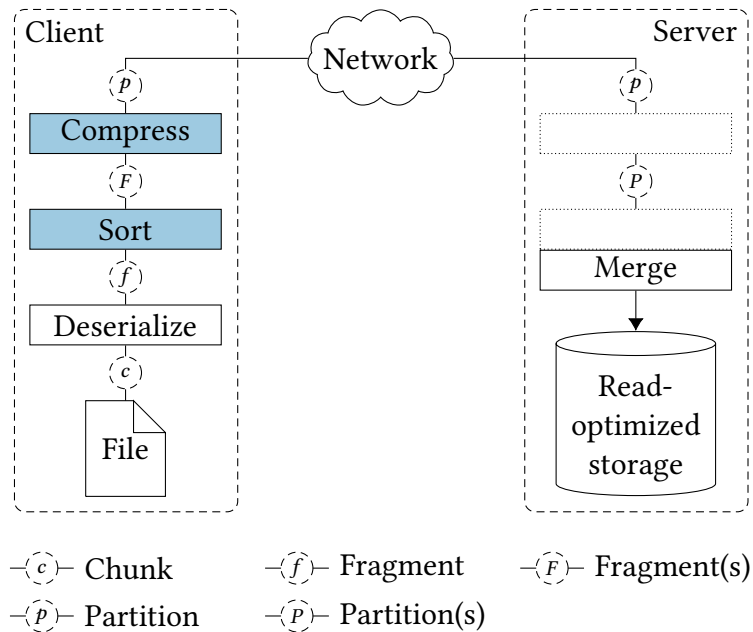
### 5.3.1 Client-Centric Loading

**Client Component.** The client component transforms data by pushing file chunks through a processing pipeline to enable a high degree of parallelism. It produces horizontal *partitions* of the input table as shown in Figure 5.5. When we shift data transformations to the client machine, the client component of *Shared Loading* produces a dictionary-compressed, columnar partition and sends it to the server machine. This allows the DBMS to merge a partition efficiently into its optimized storage. We describe the individual steps in the following.

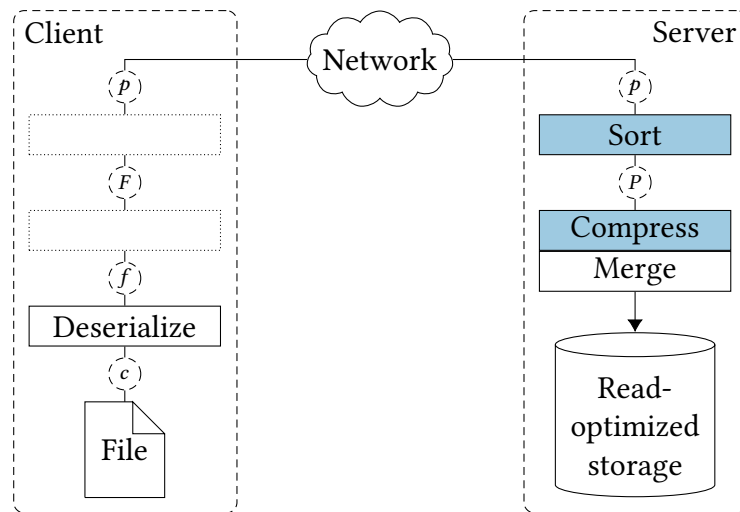
The *deserialization* step converts a file chunk to an in-memory instantiation of the data. It parses the chunk to identify delimiter symbols, validates fields, and instantiates data types in memory according to the schema of the table it gets from the server. Finally, the deserialization step assembles all rows of the chunk into a columnar in-memory representation, which we refer to as a fragment.

The *sort* step adds a temporary dictionary to a fragment's column. For each column of the fragment, it creates a copy, sorts the copy, and removes duplicates. The temporary dictionary facilitates dictionary encoding in the next step.





(a) Client-centric bulk loading into optimized storage.



(b) Server-centric bulk loading into optimized storage

Figure 5.4: Architectural overview of *Shared Loading*. Computational work (gray) can dynamically shift between client (a) and server (b): At loading time, we can decide for a *fragment's column* (cf. Figure 5.5) where to compute its data transformation.

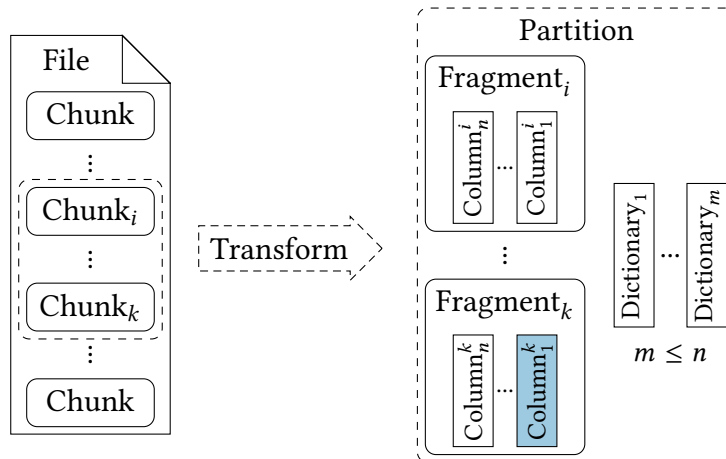
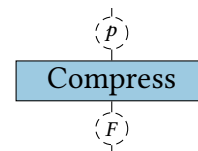


Figure 5.5: *Shared Loading* transforms a file chunk into a columnar in-memory fragment. Multiple fragments form a logical partition. A fragment's column (gray) may be dictionary-compressed.

The *compression* step logically assembles multiple fragments into a horizontal partition shown in Figure 5.5. In addition, it (physically) merges all temporary dictionaries of a column into a single dictionary. This means that a partition's dictionary is sorted locally. It is not sorted globally with respect to the entire table, previous partitions or preexisting data on the server to avoid the need for synchronization. We assure, however, that the order of rows matches their original occurrence in the input file. Thus, the DBMS can exploit the order of presorted data. Afterwards, it uses the dictionary to encode the columns of the fragments.

Finally, the client component sends a partition to the server. It transfers fragment after fragment and subsequently the dictionaries. This enables the server component to process a fragment before the partition is transferred completely. The dictionary compression reduces the transfer size: We show in the evaluation in Section 5.4.2 that the dictionary compression reduces the data size of the warehouse data set by 56 % compared to the original file size.

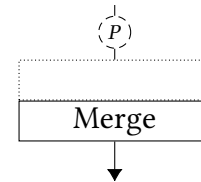
**Server Component.** The server component is designed to be part of the database system with internal access to the storage engine. When we shift data transformations to the client, it receives a dictionary-



compressed, columnar partition. This allows the DBMS to merge the partition efficiently into its read-optimized storage by updating the dictionary encoding (instead of creating the dictionary encoding from scratch). In addition, the compressed columnar format may speed up the network transfer of the partition.

The *merge* step merges partitions into the read-optimized storage. It merges all partitions available since the last merge operation. For each column of the partitions, it merges the dictionaries with the corresponding dictionary of the optimized storage. First, the merge step creates mappings from the dictionaries of the partitions to the new dictionary. Afterwards, it uses the mappings to update the optimized storage and to update the data of the partitions, which is then appended to the optimized storage.

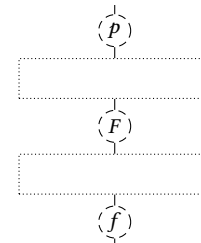
Note that we could merge incoming data into a specific partition of the target table or create a new partition to avoid updating the dictionary compression. Furthermore, we could apply additional compression methods such as bit packing during the merge.



### 5.3.2 Server-Centric Loading

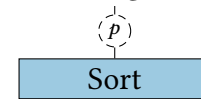
**Client Component.** When we shift only deserialization to the client, but transform data on the server, the client component of *Shared Loading* produces an uncompressed, columnar partition (see Figure 5.4b).

The *deserialization* step produces a fragment just as in the case of client-centric loading, described in Section 5.3.1. Afterwards, the client groups fragments logically into a horizontal partition. In particular, it does *not* sort fragments and does *not* apply dictionary compression as illustrated by the empty, dotted boxes in Figure 5.4b. Subsequently, the client transfers a partition to the server by sending fragment after fragment over the network.



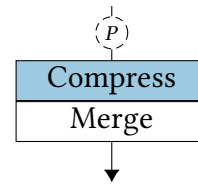
**Server Component.** The server component of *Shared Loading* receives an uncompressed, columnar partition. It needs to transform the partition, before it can merge the partition into optimized storage.

The input for the *sort* step is a partition consisting of multiple fragments. This allows the server to process each fragment independently as soon as a frag-




ment arrives. For each column of a fragment, it creates a temporary dictionary—similar to the sort step of the client component.

The *merge* step merges all partitions available since the last merge operation into the read-optimized storage. For each column of the partitions, it first merges the temporary dictionaries with the corresponding dictionary of the optimized storage. Afterwards, the merge step maps the dictionaries of the optimized storage to the merged dictionaries. It uses the mappings to update the optimized storage and the merged dictionaries to compress the partitions, which are then appended to the optimized storage.



### 5.3.3 Dynamic Offloading

The architecture of *Shared Loading* combines client- and server-centric loading. It allows deciding whether to transform a *fragment's column*, shown in Figure 5.5, on the client or on the server. The decision can be made *at loading time*. To that end, *Shared Loading* can use heuristics during the sort step at the client. It either creates a temporary dictionary and performs client-centric loading for a fragment's column, or it omits the creation of the temporary dictionary and performs server-centric loading.

The remaining steps of the bulk loading pipeline adapt to the decision. This is illustrated by the dotted boxes  in Figure 5.4. The compression step at the client only compresses a fragment's column if it has a temporary dictionary. Otherwise, the fragment's column remains uncompressed in the partition (see Figure 5.5). The sort step at the server checks if a fragment's column is not already dictionary-compressed. Only if that is the case, it creates a temporary dictionary. Thus, the sort step produces a partition where a fragment's column either is dictionary-compressed or has a temporary dictionary. The merge step either updates the dictionary compression of a fragment's compressed column or it encodes a fragment's uncompressed column when writing to optimized storage.

**Heuristics.** The architecture of *Shared Loading* enables the use of different heuristics. For instance, we use a heuristic for *minimizing* the amount of data sent over the network. In particular, we estimate the number of unique values in a column using the HyperLogLog algorithm [63] with HIP estimator [44, 152] *at loading time*. The algorithm allows us to estimate the total memory size of the dictionary-compressed

column and the corresponding dictionary. If we estimate the memory size to be smaller than the uncompressed column, we transform the column at the client; otherwise we delegate the transformation to the server. Using the example of this heuristic, we evaluate if dynamically offloading data transformations is feasible with *Shared Loading*. We also demonstrate that the heuristic produces indeed the smallest size in our evaluation.

Note that other heuristics could decide to shift data transformations based on the server's utilization, the client's and the server's compute power, or the network bandwidth. In particular, *Shared Loading* allows heuristics to use information that is only available at runtime, e.g., to implement a feedback loop.

### 5.3.4 Implementation

Our C++ implementation of *Shared Loading* exploits independent work whenever possible to achieve a high degree of parallelism. The implementation is independent of SAP HANA's code base but simulates major characteristics of SAP HANA's architecture.

We execute each step of the data processing pipeline in at least one thread. In addition, we use the thread pool provided by Intel Thread Building Blocks [92] to implement fine-grained task parallelism: We sort, compress and merge columns in parallel, and we implement parallel algorithms for compressing and merging, e.g., a parallel merge algorithm similar to P-MERGE [47, p. 800].

We use different sorting algorithms that are optimized for specific data types. We use a radix sort implementation for integers and dates, `boost::string_sort` [173, 174] for strings, and `pdqsort` [158] for the remaining types. To facilitate in-place sorting, we represent variable-sized strings of type `VARCHAR(N)` as *fixed-sized* strings of length `N` in partitions sent by the client, while the server stores variable-sized strings. Note that we avoid the increased memory and transfer size of fixed-sized strings by employing dictionary compression. To implement asynchronous network communications, we use the library `boost::asio` [101].

Currently, we do not support delimiter symbols which mark, e.g., the end of a column or the end of a row, if they are enclosed in quotes without being escaped by a preceding backslash. Handling this special case during parsing is orthogonal to our approach and only affects the deserialization step of bulk loading. To identify delimiter symbols that are not escaped by a preceding backslash, others propose, e.g.,

to perform an additional scan over the input file [137] or to employ speculative parsing techniques [70].

We additionally annotate each chunk/fragment with an identifier and assure that the order of the rows in a partition corresponds to the order of the rows in the file. This allows the database system to exploit the fact that rows in a file might be sorted, e.g., by the primary key to accelerate index creation<sup>1</sup>. Note that we do not exploit presorted data in our experiments. Furthermore, we reduce the memory footprint of the merge operation by merging at most two columns in parallel. This is SAP HANA’s default configuration for systems that have the same size as the machine used in our evaluation. This restriction reduces the impact of the merge operation on other workloads.

Our implementation has two configuration parameters: the size of a file chunk and the size of a partition. In our configuration, we set the size of a file chunk to 10 MiB, and we group 50 fragments to a partition. This means that a partition corresponds to 500 MiB of the input file. We experimentally confirm that both parameters are robust in Section 5.4.4. The chunk size needs to be big enough to contain multiple rows and to amortize the parallelization overhead (e.g., 10 KiB) and small enough to allow a high degree of parallelism (e.g., 100 MiB). Similar arguments apply to the partition size. In addition, the partition size impacts the compression ratio because we compute the dictionary compression per partition. We determine that a partition should contain between 10 and 100 chunks.

## 5.4 Evaluation

To analyze how *Shared Loading* can offload work during bulk loading and how it impacts throughput and query performance and predictability, we evaluate *Shared Loading* and state-of-the-art bulk loading architectures without and with concurrent query processing. The evaluation is the final contribution of this chapter.

First, we describe the experimental setup. Afterwards, we evaluate *Shared Loading* and state-of-the-art architectures in isolation and with concurrent query processing. Subsequently, we investigate how robust the chunk size and the partition size are. Then, we demonstrate that

---

<sup>1</sup>For example, the documentation of Amazon’s relational database service for administrating MySQL suggests creating sorted flat files: “Whenever possible, order the data by the primary key of the table being loaded. This drastically improves load times and minimizes disk storage requirements.” [8]

our approach works well with additional compression methods. Finally, we summarize and discuss results.

### 5.4.1 Setup

**Data Set.** We evaluate two data sets: the `lineitem` table of the TPC-H benchmark [197] and the warehouse table, which was extracted from the data warehouse of a customer offering telecommunication services. Both data sets are available in the file format of the TPC-H benchmark, i.e., a plain text file in a delimiter-separated values format: Columns are separated by the character “|” and rows are separated by a newline character. We use the `lineitem` table with a scale factor of 10. The file has a size of 7.24 GiB, close to  $6 \cdot 10^7$  rows, and 16 columns. On average, 94.5 % of the values in a column are duplicates. The file of the warehouse table has a size of 17.57 GiB, around  $12 \cdot 10^6$  rows, and 155 columns: 66 decimal columns, 20 integer columns, and 68 variable-sized string columns with sizes ranging from 1 to 255. On average, 97 % of the values in a column are duplicates.

**Hardware.** Our system has 128 GiB of DRAM and two Intel Xeon E5-2660 v3 processors with 10 physical cores each. We enable simultaneous multithreading. The client process and the server process run on different sockets of the same machine. We allocate 10 physical cores to the server and vary the number of cores for the client from 2 to 8. The text files reside on a local, commodity Micron M600 [194] SSD. We measured a sequential read bandwidth of up to 530 MB/s using `fiio` [15]. Note that in case of a significantly slower storage device, such as a single HDD, a weak client could compute all data transformations without being compute-bound. The SSD is only used for reading. We do not evaluate persisting and logging because we assume the server’s storage to be more powerful than the client’s. We clear the page cache of the Linux kernel (LTS version 4.4) before every run.

**Network.** We use the `tc` utility to emulate different network bandwidths in our setup—similar to work from Raasveldt et al. [168]. TCP/IP messages are sent to the localhost address. Note that we profiled the transfer of a file both between two machines in a local network and between two processes via the localhost address on the same machine, where we emulated the same bandwidth: We did not observe a difference in execution time nor in CPU time.



```
Q1. SELECT SUM(l_extendedprice)
      FROM lineitem;

Q2. SELECT COUNT(*) FROM lineitem
      WHERE l_shipdate BETWEEN
            '1994-1-1' AND '1995-1-1';
```

Listing 5.1: Analytical queries inspired by the TPC-H benchmark.

We evaluate a network bandwidth of 1 and 10 Gbit/s because these represent 69% of the market share of sold Ethernet switches [87]. 1 Gbit/s represents the maximum Internet bandwidth when loading data from an on-premise solution into the cloud. 10 Gbit/s, on the other hand, represents a typical sizing option within the cloud [7, 75] when performing a cloud-internal bulk loading operation.

**Configurations.** We evaluate *Shared Loading* in three different configurations: CLIENT, HEURISTIC, and SERVER. CLIENT corresponds to Figure 5.4a, i.e., client-centric bulk loading: The *client* transforms all data into dictionary-compressed partitions. HEURISTIC uses the heuristic to *minimize data size*: The client *dynamically* decides for each column of a fragment to compress it either on the client or on the server. SERVER corresponds to Figure 5.4b, i.e., server-centric bulk loading: The *server* transforms all data into dictionary-compressed partitions, while the client only deserializes the file.

In addition, we compare *Shared Loading* against two state-of-the-art approaches: CHUNKS and FILE. They represent the bulk loading mechanism of current systems, e.g., using terminal-based commands. CHUNKS means that the client sends file chunks to the server, *while* the server ingests the data. FILE means that the client first transfers the entire file. *Afterwards*, the server ingests all the data at once. Note that in all configurations we bypass the write-optimized storage of the system and ingest data directly into the read-optimized storage.

**Query Workload.** We use two analytical queries that are based on the TPC-H benchmark to evaluate the impact of concurrent query processing. Listing 5.1 shows the SQL statements. Queries run against a second instance of the `lineitem` table with a scale factor of 10. Thus, query processing is independent of bulk loading. We execute each query ten times in a batch, wait for all queries to finish and then execute the

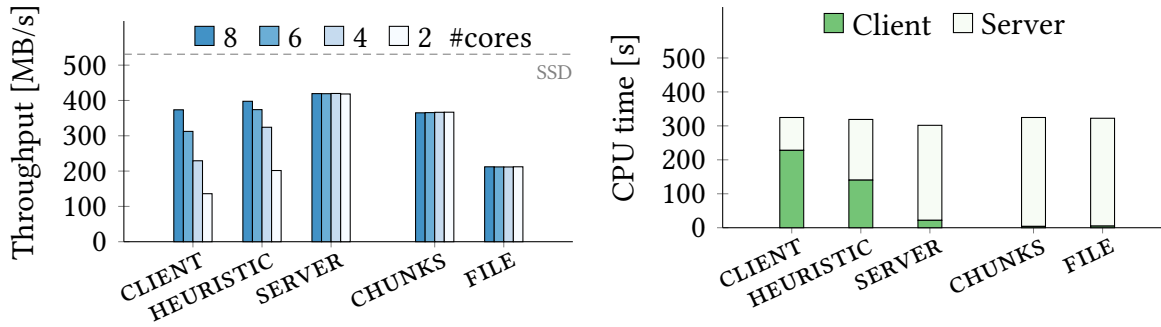


Figure 5.6: Results of bulk loading the lineitem table over a 10-Gbit network without query processing.

next batch. This way, we induce a constant query load from the time the client starts reading the file until the server has merged all data into its read-optimized storage. For a given data set and network bandwidth, we base the duration of the query workload on the maximum loading time among all configurations.

**Measurement Method.** We measure *throughput*, i.e., the size of the file divided by the execution time of the bulk loading. In addition, we measure the *CPU time* of the bulk loading to quantify computational work. CPU time is the total time which processor cores spent on executing instructions. This does not include idle time, i.e., the time cores spend waiting for asynchronous I/O operations.

To evaluate the impact of query processing, we measure *tail latency*—a performance metric for mission-critical systems with stringent service-level agreements [50, 52, 76]. In particular, we focus on the maximum response time of 99.9% of the requests, i.e., the 99.9<sup>th</sup> latency percentile. According to employees of SAP, customers often prefer predictable response times over peak performance.

## 5.4.2 Loading in Isolation

To analyze the maximum throughput and CPU time, we evaluate bulk loading in isolation. We present the results for a 10-Gbit network connection, followed by the results for a 1-Gbit network connection.

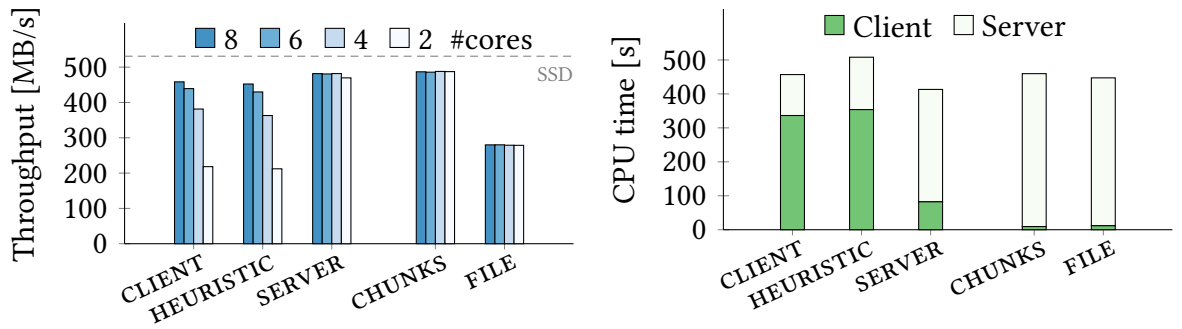


Figure 5.7: Results of bulk loading the warehouse table over a 10-Gbit network without query processing.

### 10-Gbit Network

Figure 5.6 and Figure 5.7 show the results for a network bandwidth of 10 Gbit/s. The results demonstrate that *Shared Loading* can offload a large amount of work to the client (independent of the number of cores allocated to the client). For the `lineitem` table, our approach can shift 71 % (CLIENT), 44 % (HEURISTIC), and 7 % (SERVER) of the total CPU time to the client. For the warehouse table, we are able shift 74 %, 69 %, and 20 % to the client, respectively.

We observe that, for the dynamic configuration of *Shared Loading* (HEURISTIC), the amount of CPU time shifted to the client varies due to the heuristic: For the `lineitem` table, the client compresses 10 out of 16 columns; for the warehouse table, the client compresses 140 out of 155 columns. In particular, the `lineitem` table contains more columns that do not profit from dictionary compression, such as the foreign key columns `l_orderkey` and `l_partkey` or the decimal column `l_extendedprice`, which the heuristics recognize. In addition, the results demonstrate that the server component of *Shared Loading* always consumes less CPU time than the state-of-the-art bulk loading architectures `CHUNKS` and `FILE`.

The throughput of *Shared Loading* is comparable to the state of the art. For the `lineitem` table, the throughput exceeds in some cases (SERVER and HEURISTIC) the state-of-the-art architectures—by up to 15 %. For the warehouse table, the throughput of *Shared Loading* and the state-of-the-art architectures get close to the maximum read bandwidth of the SSD. We observe that shifting only deserialization to the client (SERVER) results in the highest throughput. The results come as no surprise because the bulk loading is limited by the SSD’s read bandwidth

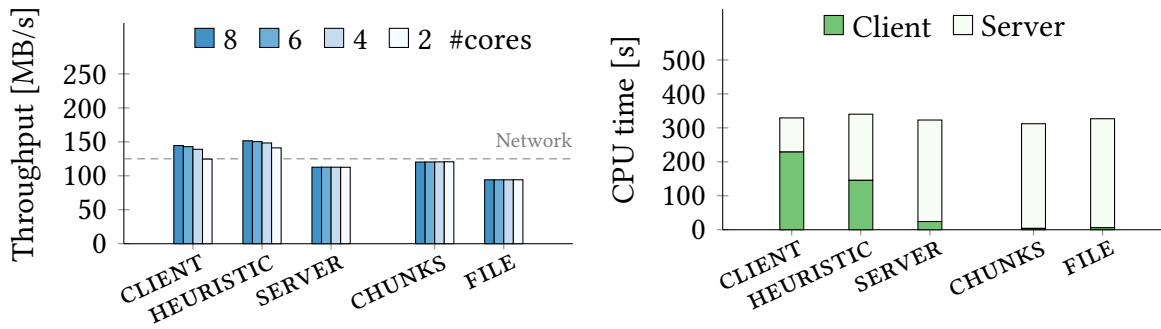


Figure 5.8: Results of bulk loading the `lineitem` table over a 1-Gbit network without query processing.

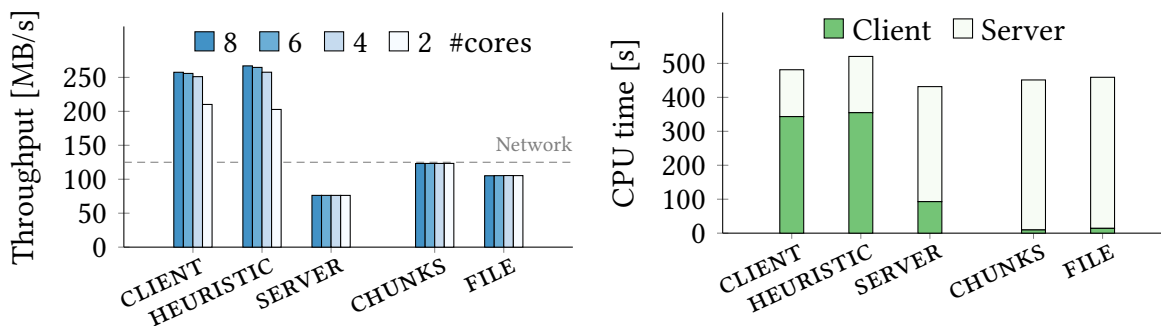


Figure 5.9: Results of bulk loading the `warehouse` table over a 1-Gbit network without query processing.

and the client’s compute power—not by the network bandwidth. For a client with at least 6 cores, it can be feasible to shift part of the data transformations to the client (HEURISTIC).

### 1-Gbit Network

Figure 5.8 and Figure 5.9 illustrate the results for a network bandwidth of 1 Gbit/s. We observe that the measured CPU time resembles the results for a network bandwidth of 10 Gbit/s. The asynchronous network transfer avoids any impact of the network transfers on CPU time: Processor cores enter a power-saving mode whenever there is not enough data to process; they wake up as soon as data is available.

In addition, the results demonstrate that throughput is network-bound, and that throughput differs significantly between configurations. We attribute this to the amount of data, shown in Table 5.1, that the

Configuration	Transfer Size [GiB]	
	lineitem	warehouse
CLIENT	5.80	8.06
HEURISTIC	5.58	7.76
SERVER	7.65	28.33
CHUNKS	7.24	17.57
FILE	7.24	17.57

Table 5.1: The total amount of data (in GiB, per data set) a bulk loading configuration transfers over the network.

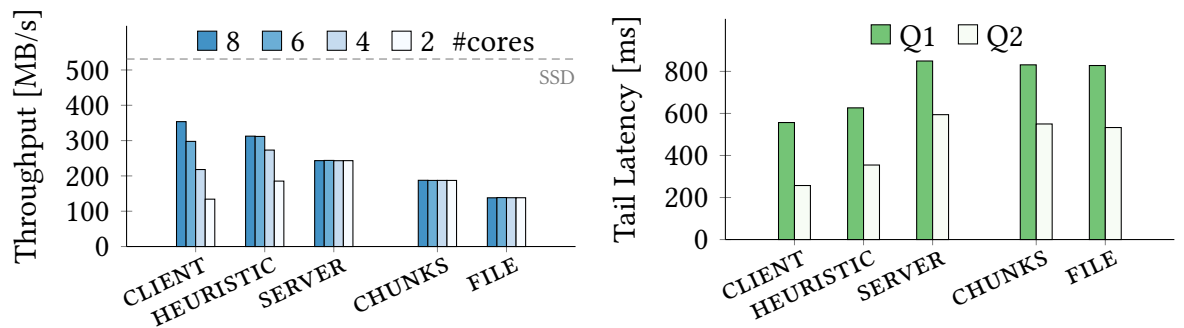


Figure 5.10: Results of bulk loading the lineitem table over a 10-Gbit network with concurrent query processing.

client transfers to the server depending on the configuration. This is especially notable for the warehouse table, where dictionary compression achieves a compression ratio of 2.3. Note that for server-centric bulk loading (SERVER) without dictionary-compressed network transfers, the transfer size increases compared to the original file because we transfer all data types, especially strings, with a fixed size (cf. Section 5.3.4).

We observe that in slower network environments *Shared Loading* performs best when the client selectively transforms data in order to minimize the transfer size (HEURISTIC). For the lineitem table, HEURISTIC reduces the transfer size to 77 % of the size of the input file and it increases throughput by 26 % compared to CHUNKS. For the warehouse table, HEURISTIC transfers data with 44 % of the file size and it increases throughput by 117 % compared to CHUNKS. Thus, using the heuristic results indeed in the smallest transfer size.

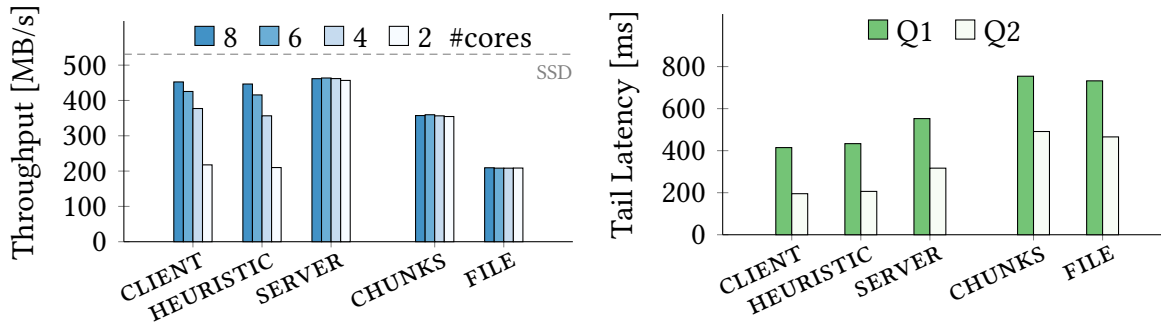


Figure 5.11: Results of bulk loading the warehouse table over a 10-Gbit network with concurrent query processing.

### 5.4.3 Loading With Concurrent Queries

To analyze throughput and query performance when the server is stressed, we evaluate bulk loading with queries running in parallel. We present the results for a 10-Gbit network connection, followed by the results for a 1-Gbit network connection.

#### 10-Gbit Network

Figure 5.10 and Figure 5.11 show the results for a network bandwidth of 10 Gbit/s. We notice that the results differ from previous results without query processing. For the `lineitem` table, we observe that *Shared Loading* increases throughput as more and more work is offloaded from the server to the client (`SERVER`  $\rightarrow$  `HEURISTIC`  $\rightarrow$  `CLIENT`).

Compared to the configuration `CHUNKS`, the configuration `CLIENT` increases throughput by 89%. For the warehouse table, `CLIENT` improves throughput by 27% compared to `CHUNKS`. This demonstrates that, by shifting all transformations including compression to the client (`CLIENT`), *Shared Loading* can maintain a high loading rate even when the server is stressed—unlike state-of-the-art bulk loading approaches `CHUNKS` and `FILE`.

In addition, *Shared Loading* leaves the server more resources for query processing by offloading transformations: When we compare `CHUNKS` with `CLIENT`, tail latency improves by 33% for Q1 and 53% for Q2 for the `lineitem` table. For the warehouse table, tail latency improves by 45% and 60%, respectively. Thus, *Shared Loading* can improve query performance and predictability by reducing server load in peak load situations.

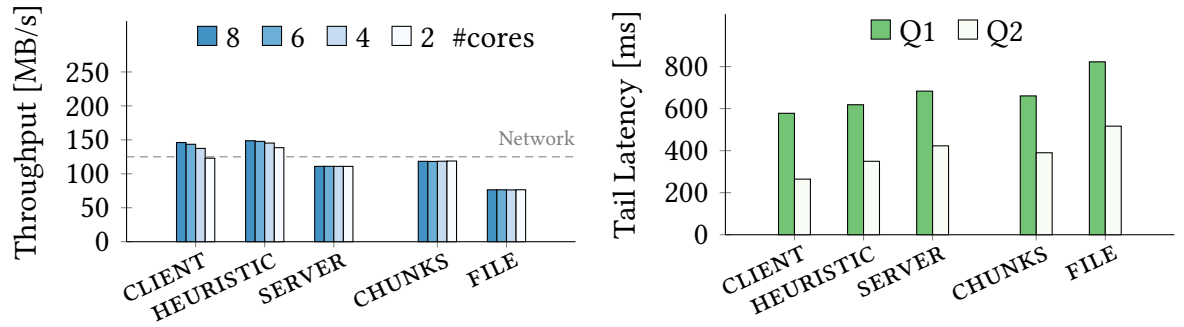


Figure 5.12: Results of bulk loading the lineitem table over a 1-Gbit network with concurrent query processing.

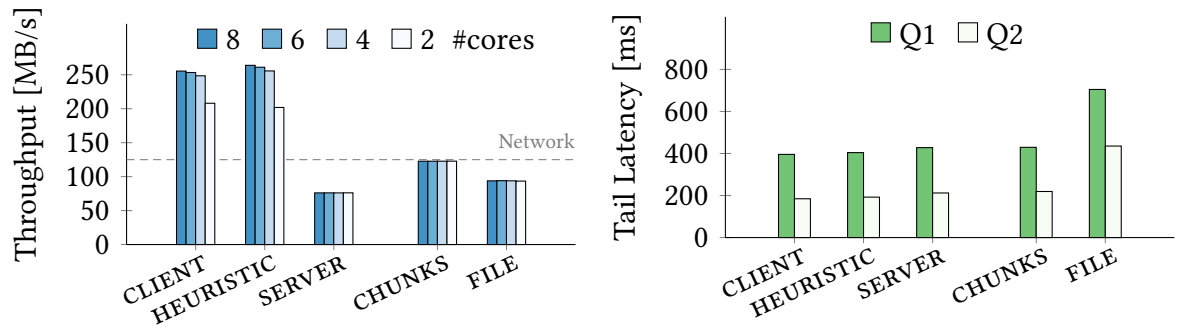


Figure 5.13: Results of bulk loading the warehouse table over a 1-Gbit network with concurrent query processing.

The client can be relatively weak: The results show that 4 cores suffice to transform and compress all data on the client while achieving a higher throughput than the state of the art. Average response times (not shown) improve by 19% and 47% for the lineitem table and by 13% and 33% for the warehouse table when comparing CLIENT with CHUNKS. Note that if the server is under heavy load, job scheduling cannot effectively reduce resource contention. *Shared Loading* mitigates a high system load by leveraging the additional hardware resources of the client.

### 1-Gbit Network

Figure 5.12 and Figure 5.13 show the results for a network bandwidth of 1 Gbit/s. We observe that the results on throughput resemble the ones without query processing: Bulk loading is network-bound. HEURISTIC

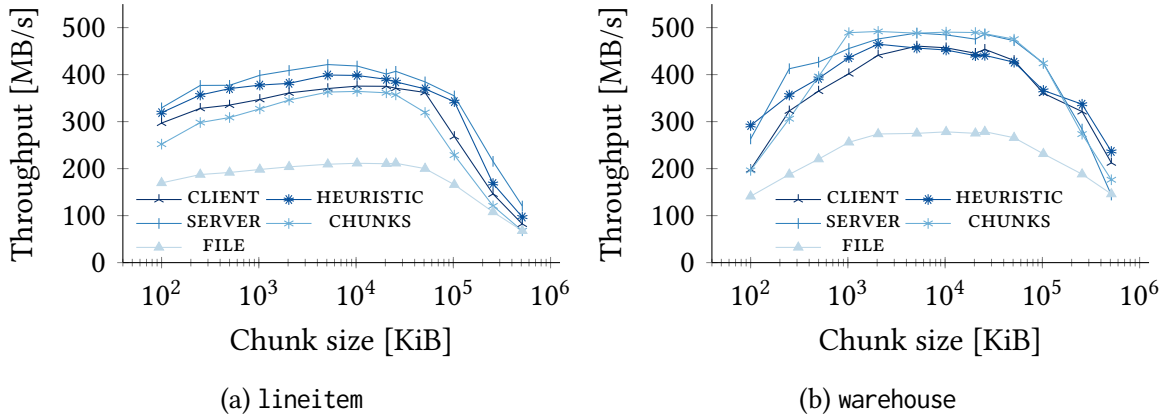


Figure 5.14: Results of bulk loading the lineitem (a) and warehouse (b) table over a 10-Gbit network with varying chunk sizes.

improves throughput by up to 26 % and 116 % compared to CHUNKS. In particular, we observe that throughput does not degrade even though the server is stressed. We attribute this to the low network bandwidth: It gives the server more time to process incoming partitions and makes merging the data into optimized storage less vulnerable to the contention of CPU resources.

Offloading all transformations to the client (CLIENT) improves tail latency by 12 % for Q1 and by 32 % for Q2 compared to CHUNKS for the lineitem table. For the warehouse table, tail latency only improves by 8–16 %. This demonstrates that the transformations of the warehouse table cause fewer load spikes than the lineitem table. *Shared Loading* primarily improves throughput due to the reduced transfer size, while its efficient offloading never degrades query processing on the server.

#### 5.4.4 Robustness of Parameters

In our implementation of *Shared Loading*, we choose two parameters: the chunk size and the partition size. To demonstrate how robust these parameters are, we evaluate *Shared Loading* for varying parameter values. We additionally compare against state-of-the-art bulk loading architectures: We modify the architectures to support different parameter values<sup>2</sup>.

<sup>2</sup>In the previous experiments, CHUNKS has a chunk size of 50; FILE has the maximum chunk size, i.e., only a single chunk. The partition size of both configurations equals 1.



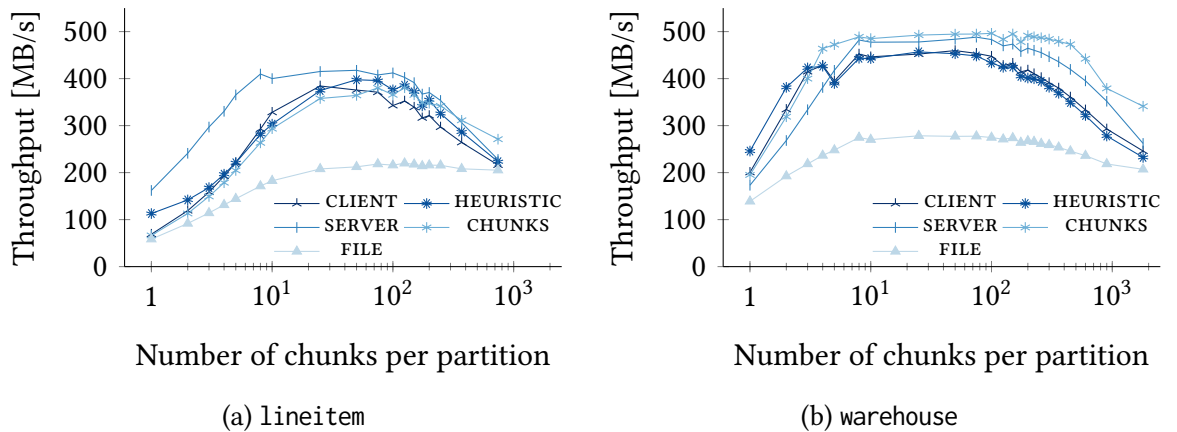


Figure 5.15: Results of bulk loading the lineitem (a) and warehouse (b) table over a 10-Gbit network with varying partition sizes.

**Chunk Size.** The chunk size determines into how many chunks we split an input file. It affects the level of parallelism because a chunk is the input for an independent work unit in our architecture. To study the impact on throughput, we vary the chunk size but keep the partition size fixed to 500 MiB. The network bandwidth equals 10 Gbit/s. Figure 5.14 visualizes the results.

Throughput decreases for chunk sizes below 1 MiB and for chunks sizes above 50 MiB. For example, a chunk size of 10 MiB results in the maximum throughput. Consequently, we choose a chunk size of 10 MiB in our implementation. However, the results also demonstrate that the chunk size is relatively robust. The size needs to be big enough to contain multiple rows and to amortize the parallelization overhead. In addition, it needs to be small enough to allow a high degree of parallelism. Note that the parameter is also robust for a network bandwidth of 1 Gbit/s (not shown). Selecting a chunk size between 100 KiB and 100 MiB results in maximum throughput.

**Partition Size.** The partition size determines the number of partitions the client sends over the network to the server. It affects the level of parallelism because the server merges at least one partition. In addition, it affects the compression ratio because *Shared Loading* computes the dictionary compression per partition. To study the impact on throughput, we vary the partition size but keep the chunk size fixed to 10 MiB. The network bandwidth equals 10 Gbit/s. Figure 5.15 visualizes the results.

We observe that *Shared Loading* achieves the highest bulk loading throughput if 25 to 100 chunks form a partition. This demonstrates that the parameter is robust. Creating partitions with less than 25 chunks, e.g., with only one chunk results in 742 partitions for the `lineitem` table and 1800 partitions for the warehouse table. Too many partitions significantly increase the overhead of the processing pipeline resulting in poor performance. On the other hand, creating partitions with more than 100 chunks limits the parallelism. In our implementation, we choose to group 50 chunks into a partition. This results in 15 partitions for the `lineitem` table and in 36 partitions for the warehouse table.

Choosing larger partitions with 200 or more chunks results in poor performance because of the low number of partitions. The total number of partitions limits both the client and the server in how many partitions they can process in parallel. In particular, a low number of partitions causes the server to run idle for long periods of time between merge operations and increases the amount of data that the server needs to process once the client has finished processing partitions. Note that the results are very similar for a network bandwidth of 1 Gbit/s (not shown).

### 5.4.5 Additional Compression

To further improve throughput for slow network environments, we can combine *Shared Loading* with state-of-the-art compression algorithms. As an example, we evaluate the use of LZ4 [46], a fast, lossless compression algorithm. We modify *Shared Loading* and the state-of-the-art bulk loading architectures such that the client compresses network transfers with LZ4. This means that the server needs to decompress data before processing it. Furthermore, we compare the experimental results to the results without LZ4 compression presented in Section 5.4.2.

#### Loading in Isolation (LZ4)

To analyze maximum throughput and CPU time, we evaluate bulk loading in isolation. We present the results for a 10-Gbit network connection, followed by the results for a 1-Gbit network connection. Network transfers are compressed with LZ4.

**10-Gbit Network.** Figure 5.16 and Figure 5.17 show the results for a network bandwidth of 10 Gbit/s. The results demonstrate that throughput degrades by up to 31 % for state-of-the-art bulk loading architec-

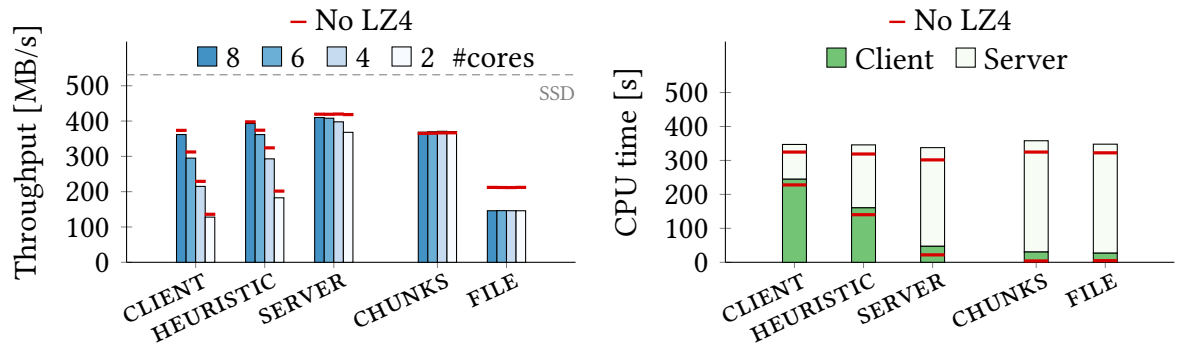


Figure 5.16: Results of bulk loading the `lineitem` table over a 10-Gbit network without query processing and with LZ4 compression. For reference, we include the results without LZ4 compression of Figure 5.6.

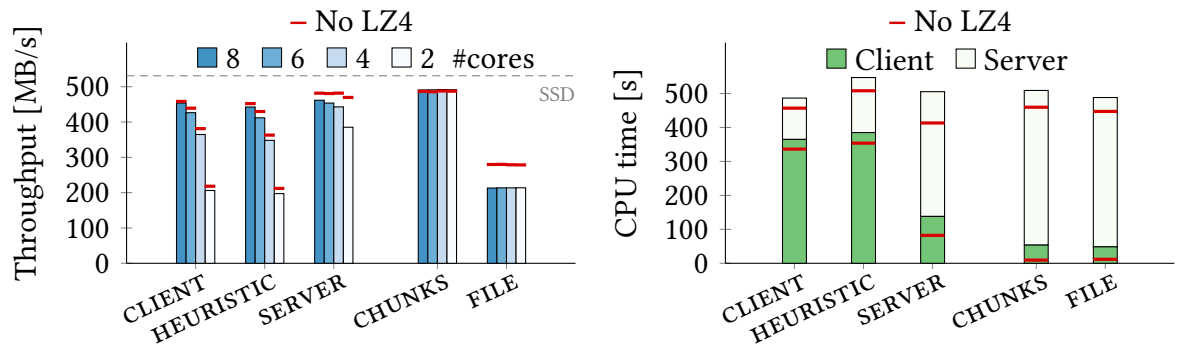


Figure 5.17: Results of bulk loading the `warehouse` table over a 10-Gbit network without query processing using LZ4 compression. For reference, we include the results without LZ4 compression of Figure 5.7.

tures (`lineitem`, `FILE`) and by up to 18 % for *Shared Loading* (`warehouse`, `SERVER`, client with 2 cores) compared to the results without LZ4 compression. In addition, we observe that total CPU time increases by up to 11 % for the state of the art (`warehouse`, `CHUNKS`) and by up to 22 % for *Shared Loading* (`warehouse`, `SERVER`). In fact, the CPU time of the client increases more than the CPU time of the server, because compression is more expensive than decompression.

The results are not surprising. We already confirm in Section 5.4.2 that bulk loading is not network-bound for 10-Gbit network environments. Thus, applying LZ4 compression degrades performance due to the additional (de)compression costs.

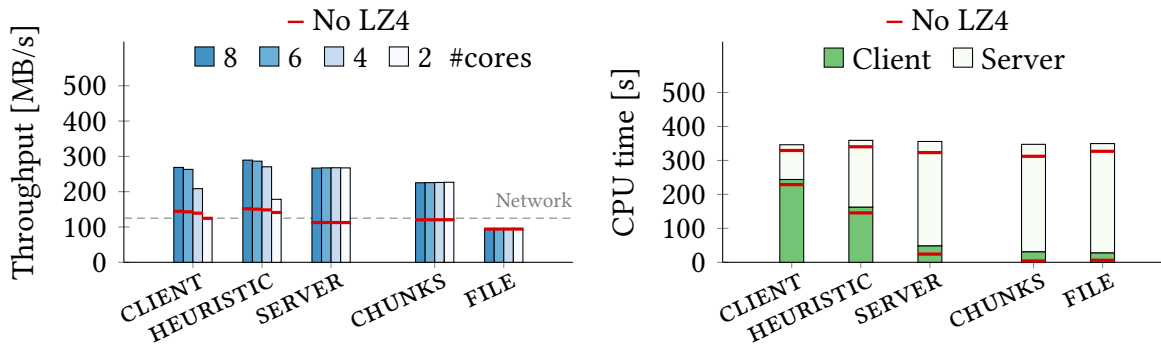


Figure 5.18: Results of bulk loading the `lineitem` table over a 1-Gbit network without query processing using LZ4 compression. For reference, we include the results without LZ4 compression of Figure 5.8.

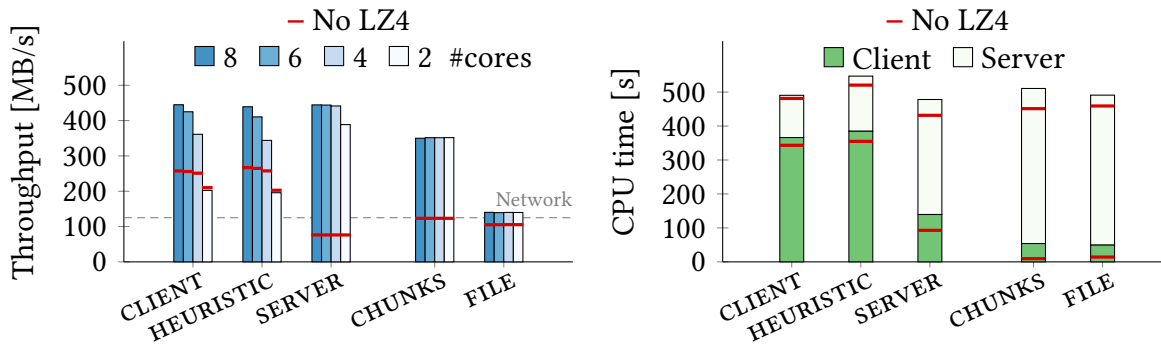


Figure 5.19: Results of bulk loading the `warehouse` table over a 1-Gbit network without query processing using LZ4 compression. For reference, we include the results without LZ4 compression of Figure 5.9.

**1-Gbit Network.** Figure 5.18 and Figure 5.19 show the results for a network bandwidth of 1 Gbit/s. The results show that the measured CPU times resemble previous results for a 10-Gbit network—due to the asynchronous network transfers (cf. Section 5.4.2). More importantly, the results show that, for the `lineitem` table, the use of LZ4 compression increases throughput of *Shared Loading* by 86 % (CLIENT), by 91 % (HEURISTIC), and by 137 % (SERVER). The throughput of the state-of-the-art architecture `CHUNKS` increases by up to 87 %. For the `warehouse` table, throughput of *Shared Loading* increases by 73 % (CLIENT), by 65 % (HEURISTIC), and by 483 % (SERVER). The throughput of the state-of-the-art architecture `CHUNKS` increases by up to 184 %. Compared to `CHUNKS`, *Shared Loading* increases throughput by up to 28 %. This

Configuration	Transfer Size [GiB]			
	lineitem		warehouse	
	with LZ4	<i>without LZ4</i>	with LZ4	<i>without LZ4</i>
CLIENT	2.67	<i>5.80</i>	3.06	<i>8.06</i>
HEURISTIC	2.56	<i>5.58</i>	2.89	<i>7.76</i>
SERVER	2.97	<i>7.65</i>	4.38	<i>28.33</i>
CHUNKS	3.72	<i>7.24</i>	6.00	<i>17.57</i>
FILE	3.72	<i>7.24</i>	6.00	<i>17.57</i>

Table 5.2: The total amount of data (in GiB, per data set) a bulk loading configuration transfers over the network with LZ4 compression. For reference, we show transfer sizes without LZ4 compression (from Table 5.1, *italicized*).

shows that our approach also outperforms the state-of-the-art bulk loading architectures if we compress network transfers with LZ4.

In addition, we observe that the throughput of all three configurations of *Shared Loading* reaches a similar level. We attribute this to the size of the network transfers, shown in Table 5.2. For the `lineitem` table, the additional compression reduces transfer sizes by up to 61 % for *Shared Loading* and by 49 % for `CHUNKS`. For the `warehouse` table, LZ4 compression reduces transfers sizes by up to 85 % and by 66 %, respectively.

In fact, the results demonstrate that the combination of selectively applying dictionary compression based on the heuristic to minimize data size (`HEURISTIC`) and LZ4 compression results in the smallest transfer size. Thus, the physical reorganization of the data due to dictionary compression enables the LZ4 algorithm to achieve a higher compression rate. It shows that using LZ4 compression *together with* dictionary compression works well with *Shared Loading*.

### Loading With Concurrent Queries (LZ4)

To analyze throughput and query performance under load, we evaluate bulk loading with queries running in parallel. We present the results for a 10-Gbit network connection, followed by the results for a 1-Gbit network connection. Network transfers are compressed with LZ4.

**10-Gbit Network.** Figure 5.20 and Figure 5.21 show the results for a network bandwidth of 10 Gbit/s. Similar to the results without con-

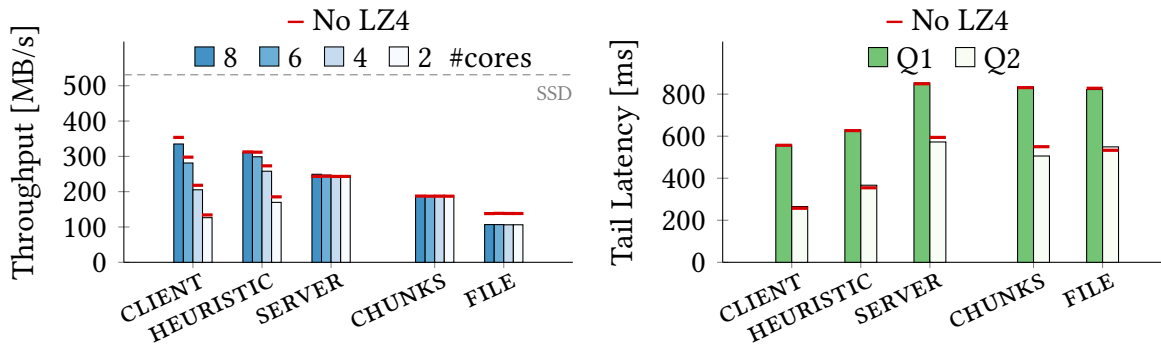


Figure 5.20: Results of bulk loading the lineitem table over a 10-Gbit network with concurrent query processing using LZ4 compression. For reference, we include the results without LZ4 compression of Figure 5.10.

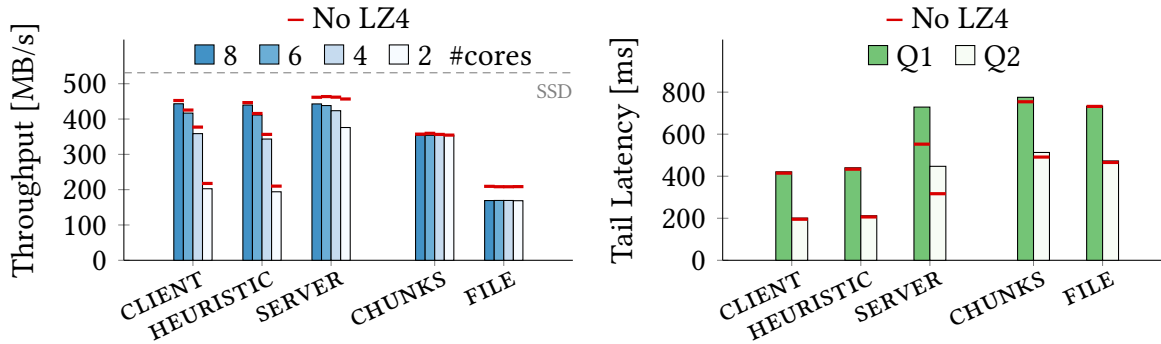


Figure 5.21: Results of bulk loading the warehouse table over a 10-Gbit network with concurrent query processing using LZ4 compression. For reference, we include the results without LZ4 compression of Figure 5.11.

current query processing, we observe that LZ4 compression does not improve throughput. In fact, throughput degrades by up to 18% for *Shared Loading* and by 23% for *FILE*. Nevertheless, *Shared Loading* improves throughput by up to 61% (lineitem, CLIENT, client with 8 cores) compared to state-of-the-art architectures (CHUNKS) if network transfers are compressed with LZ4.

In addition, we observe that the use of LZ4 compression worsens tail latency for SERVER (warehouse) by up to 41%. Similar to the results without concurrent query processing, we observe that *Shared Loading* improves tail latency by up to 62% (CLIENT) compared to the state of the art (CHUNKS).

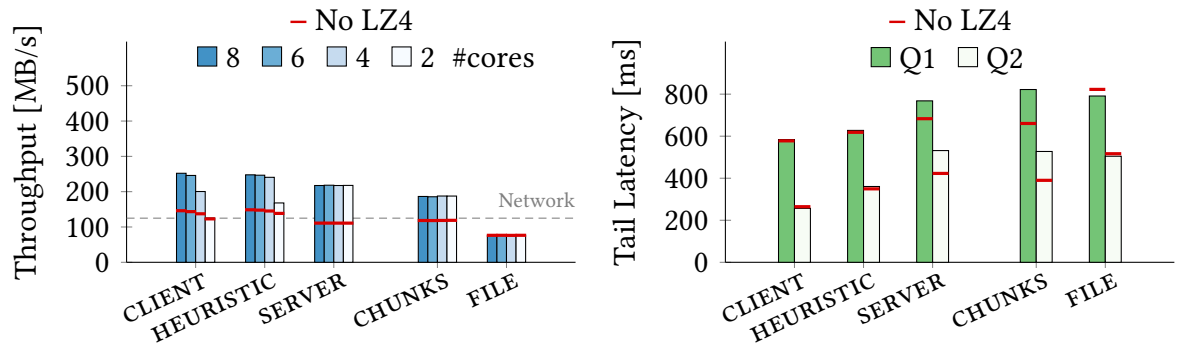


Figure 5.22: Results of bulk loading the `lineitem` table over a 1-Gbit network with concurrent query processing using LZ4 compression. For reference, we include the results without LZ4 compression of Figure 5.12.

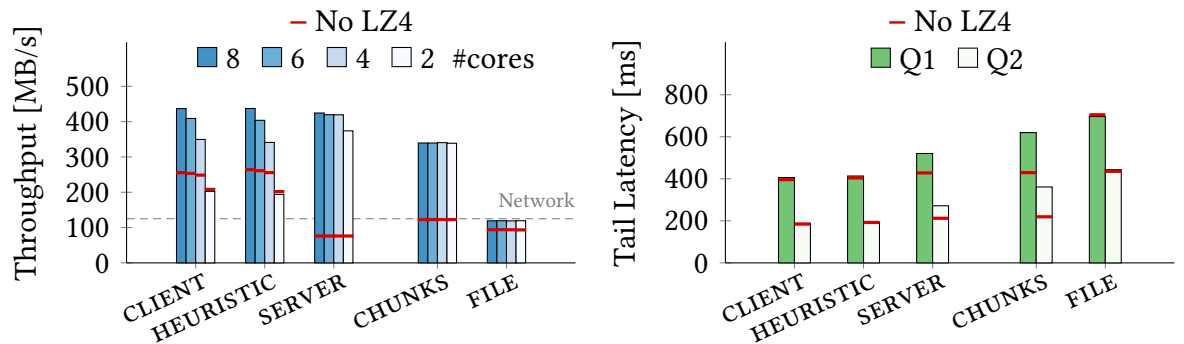


Figure 5.23: Results of bulk loading the `warehouse` table over a 1-Gbit network with concurrent query processing using LZ4 compression. For reference, we include the results without LZ4 compression of Figure 5.13.

**1-Gbit Network.** Figure 5.22 and Figure 5.23 show the results for a network bandwidth of 1 Gbit/s. We observe that using LZ4 compression improves throughput. For the `lineitem` table, *Shared Loading* (CLIENT, client with 8 cores) increases throughput by up to 35 % compared to `CHUNKS`. For the `warehouse` table, it improves throughput by up to 29 %, respectively.

Furthermore, the results show that the use of LZ4 compression worsens tail latency by up to 65 % (`CHUNKS`). However, *Shared Loading* improves tail latency by up to 48 % (CLIENT) compared to the best-performing state-of-the-art bulk loading architecture `CHUNKS`. This demonstrates that even if bulk loading is more expensive because of LZ4 compression, *Shared Loading* can keep the tail latency low by

offloading data transformation and deserialization. It increases both performance and predictability.

### 5.4.6 Discussion

Our evaluation demonstrates that *Shared Loading* performs up to  $2\times$  better than state-of-the-art architectures in 1-Gbit environments due to the compressed network transfer. In addition, the approach is very robust. Throughput never degrades. The performance advantage in 10-Gbit environments becomes clear once the server is under load. *Shared Loading* increases throughput by up to 89%. At the same time, tail latencies of the query workload improve by up to 60%. Offloading data transformations to the client reduces CPU contention on the server which benefits query processing *and* bulk loading.

The results also demonstrate why work needs to be shifted *dynamically*: Different configurations of *Shared Loading* perform best depending on network bandwidth, server load, and compute power of the client: In fast network environments, when loading without query processing or when the client is weak, it is best to only deserialize on the client; in slow network environments, it is best to selectively transform data on the client to minimize transfer size; and when loading with concurrent query processing, it is best to transform all data on the client.

To further improve throughput for slow networks, we can combine *Shared Loading* with additional compression methods, such as LZ4 [46]. Combining *Shared Loading* with LZ4 compression increases throughput by up to 91% compared to using only dictionary compression based on our heuristic. Tail latency improves by up to 48%. This demonstrates that *Shared Loading* works well with other compression methods. In particular, the combination of dictionary compression *and* LZ4 compression results in the smallest transfer size. Note that we also evaluated other compression methods (not shown): the (de)compression algorithms of zlib [4], snappy [74], and zstd [60]. While the heavy-weight compression of zlib results in a better compression ratio, bulk loading throughput and query latency degrade due to the high CPU costs. Both snappy and zstd give similar results as LZ4 compression.

Ultimately, we envision the client component of *Shared Loading* to be part of a lightweight SQL client, shipped with the database. We argue that the complexity of the client components remains low because it performs only deserialization and data transformation—no query processing or extensive validation. In addition, *Shared Loading* consumes a



low amount of memory due to its architecture. In our implementation, the client buffers only one partition in-between processing steps resulting in 7 partitions in total. If we assume the chunk and partition sizes discussed in Section 5.3.4, the total memory consumption will not exceed the in-memory equivalent of  $7 \cdot (50 \cdot 10 \text{ MiB}) \approx 3.5 \text{ GiB}$  of file data for any table. The low resource consumption makes *Shared Loading* also a good candidate for implementing bulk loading in a cloud-native database. When loading a large volume of data, the system could start a (small) instance running the client component of *Shared Loading* to ensure elasticity and reduce costs.

## 5.5 Related Work

### Bulk Loading

Database systems offer a range of interfaces for bulk loading external files. We group existing approaches into three categories.

First, various systems offer a *command* for the terminal-based front-end of the DBMS. The user either manually transfers the file to the DBMS server or it is transferred during the loading operation. All data processing is done by the DBMS server. Examples include the commands `IMPORT FROM` of IBM Db2 [86] and `SAP HANA` [176], `BULK INSERT` of Microsoft SQL Server [134], `LOAD DATA INFILE` of MySQL [141], or `COPY` of PostgreSQL [163].

Second, systems may support parameterized SQL queries with *array bindings*. In contrast to submitting a query for every row of a table, it allows batching multiple rows in a query with a single `INSERT INTO` statement. The database system may support array bindings directly by processing a single SQL statement per batch or it may emulate array bindings by processing a prepared statement per row of a batch. Most systems with support for ODBC [71] or JDBC [12] support array bindings.

Third, some vendors provide *dedicated tools* for bulk loading external files. Some of these tools use array bindings, such as `bcp` from Microsoft [134], while other tools, such as `SQL*Loader` from Oracle [154] or the `LOAD` utility from Db2 [86], write data blocks “directly to the database”. Their documentation does not detail, however, what is computed by the client and by the server.

Dziedzic et al. [55] analyze data loading with terminal-based commands for different database systems. They assume that data already

resides on the server. They corroborate our results by showing that bulk loading can be slow and expensive, especially when loading data into compression-optimized systems. In fact, we close the gap discussed in their work: Our dynamic loading mechanism accelerates bulk loading and improves query performance by taking the load off the DBMS server.

### **In-Situ Query Processing**

Query processing on files [3, 6, 25, 37, 70] focuses on analyzing large files with the goal to minimize initial response times. Thus, first loading the entire file into a DBMS incurs a high initial cost and is usually avoided. In addition, it may suffice to read only a subset of the data for answering a query. This allows, e.g., to push down predicate evaluation into parsing. Related work identifies the repeated parsing, tokenizing, and data type conversion as a performance bottleneck but does not consider network transfers or complex transformations.

While approaches such as NoDB [6] allow (partially) loading and directly querying files, a user that requires, e.g., ACID properties or the interoperability and security of a feature-rich system typically chooses to load data into a DBMS. Similar arguments apply to approaches based on external tables. For example, Oracle Database supports external tables [17], or MySQL offers a dedicated CSV storage engine [142]. However, their goal is data integration, not data ingestion. Hence, their query performance is limited because they have no or reduced support for caching, indices, or statistics.

### **Fast Parsing and Ingestion**

Mühlbauer et al. [137] use SIMD instructions to accelerate deserialization and propose mergeable indices for bulk loading. More recently, Langdale et al. [107] present techniques, e.g., for identifying escaped quote characters and converting digits to integer values using recent SIMD instructions. Xie et al. [209] propose a storage layer for analytics that combines parsing of selected fields and a dynamic, hash-based subset index for querying data during ingestion. They store data in an immutable, row-based log. They do not consider data transformations such as compression or a columnar storage layout.

We do not focus on optimizing parsing and deserialization because our analysis shows that these operations consume less than 20 % CPU time in a commercial database system (cf. Section 5.2). Besides, optimiz-

ing parsing and deserialization by applying SIMD techniques and using just-in-time generated glue code is orthogonal to our approach. We focus on a distributed environment where our approach can dynamically shift data transformations between client(s) and a server to bring data into the optimized storage format of the database system.

### Network Serialization

Raasveldt et al. [168] analyze data *export* from various database systems. They conclude that protocols suffer from a per-row overhead and expensive (de)serialization. They propose to transfer data in column-major chunks with variable-sized strings as well as to employ columnar compression techniques.

In fact, our network protocol resembles their design. However, we transfer larger chunks of data, i.e., 500 MiB not 1 MB, and we choose to represent strings fixed-sized instead of variable-sized to simplify in-place sorting. We reduce transfer volume by employing dictionary compression. While we focus on efficient data ingestion by offloading expensive transformation to the client, a variant of our approach could also help to extract data *from* the system. We expect, however, a smaller benefit because transforming data into, e.g., a dictionary-compressed format, is often more expensive than decompressing the data again.

### Dictionary Compression

The in-memory database system SAP HANA [62] makes heavy use of *order-preserving* dictionary compression in its read-optimized storage—other systems employ similar ideas to varying degrees [29, 105, 109, 161] (cf. Section 2.2.1). In our approach, we focus on order-preserving dictionary compression. Applying additional compression such as bit packing or prefix encoding is orthogonal. We observed in experiments (not shown) that they can be combined efficiently by exploiting the sorting and the known distinct count of dictionary compression.

### Buffered Updates

To facilitate data ingestion into optimized storage, SAP HANA transforms new data gradually, by migrating records from write- to read-optimized storage (cf. Section 2.2.2). Our approach bypasses the write-optimized storage and merges new data directly into the read-optimized storage—similar to work from Lamb et al. [105]. However, our approach enables offloading data transformations at loading time to the client.

## 5.6 Conclusion

In today’s heterogeneous system landscape, bulk loading plain text files is a performance-critical task for data analysis, replication, system integration, and migration. However, for systems that employ a (highly) compressed storage, bulk loading can stress the system significantly. In particular, data *transformation* during bulk loading can be very expensive and negatively impact workloads running in parallel.

We analyze the costs of bulk loading into a commercial in-memory database system with a compression-optimized storage. Our analysis shows that most processing time is spent on transforming data into a compressed format—not on deserializing the input file. Moreover, we confirm that state-of-the-art bulk loading significantly degrades the tail latency of queries running in parallel. At the same time, the performance of bulk loading suffers as well.

To address this problem, we exploit the hardware setup: a distributed environment consisting of the DBMS server and one or more client machines. In particular, we leverage the compute power of the client to reduce server load and to produce a compressed storage format that utilizes available network bandwidth more efficiently. We propose the distributed bulk loading mechanism, *Shared Loading*, which enables *dynamically offloading* deserialization and data transformation—per column fragment—to the client holding the input file. Our evaluation using the lineitem table of the TPC-H benchmark and a real-world data set determines that *Shared Loading* increases bulk loading throughput especially in slower network environments or when the DBMS is stressed. Throughput increases by up to 117%. At the same time, it can significantly improve tail latency of a query workload to enable efficient bulk loading into compression-optimized storage without sacrificing query performance and predictability. Tail latency improves by up to 60%. Moreover, we demonstrate that *Shared Loading* works well with additional compression methods such as LZ4: It improves throughput by up to 61% and tail latency by up to 62%.

# 6

## Conclusion

Resource-efficient data processing can cut costs, increase performance, and improve predictability. However, the complex system environment of data processing applications, i.e., the combination of complex software architectures, workloads, and hardware setups makes it very challenging to identify and solve problems of inefficient resource usage. In this thesis, we contribute to the research area of resource-efficient data processing by focusing on three problems that are relevant—but not limited—to database systems:

- (i) we address the problem of analyzing complex system behavior with memory tracing, which has been considered challenging because of its prohibitive runtime overhead;
- (ii) we address the problem of cache pollution within a microprocessor that hurts performance especially in concurrent workloads; and
- (iii) we address the problem of resource contention during bulk loading and query processing across multiple machines which results in poor performance and predictability.

Our contributions of this thesis include analyses and methods for maximizing hardware utilization. In particular, we demonstrate that our approaches improve resource efficiency at different system levels.

In this chapter we summarize our contributions, give future directions, and discuss the results of this thesis.

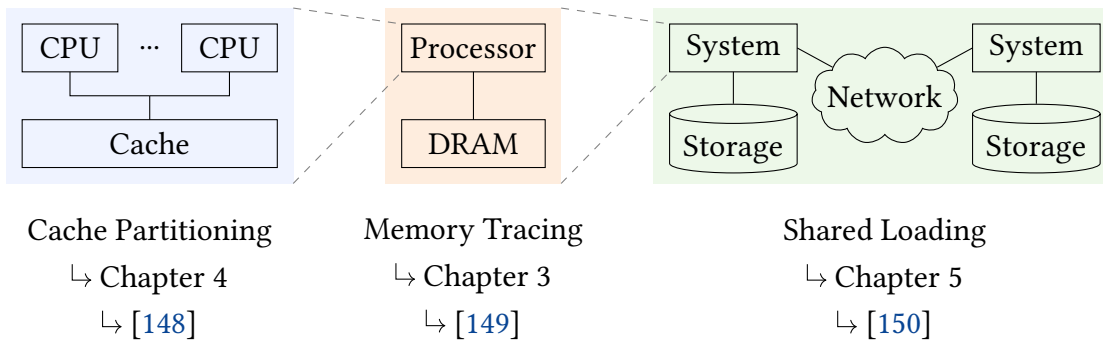


Figure 6.1: Areas of the main contributions of this thesis.

## 6.1 Summary

In this thesis we develop optimizations for resource-efficient data processing. Our three main contributions are (i) efficient memory tracing, (ii) CPU cache partitioning, which we integrate into the execution engine of a database system, and (iii) efficient bulk loading into the compressed storage of a database system. In particular, our three main contributions cover different levels of the memory and compute hierarchy as shown in Figure 6.1 (the same figure as Figure 1.1):

- (i) at the level of a *single machine*, efficient memory tracing allows us to analyze memory accesses;
- (ii) at the level of a *single processor*, CPU cache partitioning allows us to utilize the last-level cache more efficiently; and
- (iii) at the level of *multiple machines* in a distributed environment, efficient bulk loading via *Shared Loading* allows us to dynamically balance the computational work of expensive data transformations between nodes.

### Memory Tracing

Memory tracing allows inspecting and analyzing the memory access characteristics of software. Detailed insights into memory accesses enable various optimizations of memory usage. However, available tools for memory tracing suffer from a large runtime overhead. To address this problem, we develop an efficient mechanism for memory tracing via *hardware-based sampling* that leverages Intel’s PEBS mechanism. We illustrate that memory traces enable us to analyze the

runtime characteristics of a database system: Memory traces reveal access patterns and access statistics for individual data structures and database operators, detect skew at byte level, and allow us to derive the working set size of a workload as well as to analyze the impact of table partitioning. For example, deriving the working set size of a workload allows us to size the page buffer of a database system.

In addition, we demonstrate that our approach has a low runtime overhead. For example, sampling every 1000<sup>th</sup> memory access increases runtime overhead by 27 %. In summary, our approach makes it possible to analyze complex systems—even in production environments—and opens up new possibilities for optimizing resource usage, especially memory and cache usage.

## CPU Cache Partitioning

Conflicts over a shared cache can cause cache pollution and hurt performance. We demonstrate that this problem occurs especially in concurrent workloads whenever multiple queries or operations compete for the last-level cache of a processor. To address this problem, we confirm through an experimental analysis that important in-memory database operators exhibit different performance characteristics depending on the available cache size. Subsequently, we derive a cache partitioning scheme that we deliberately keep simple: We restrict memory-intensive operators that do not reuse data to a small portion of the cache.

Furthermore, we demonstrate how to integrate cache partitioning into the execution engine of a commercial DBMS with low engineering costs. In our evaluation we show that our approach avoids cache pollution, significantly reduces cache misses, and improves the overall system performance by up to 38 %. In summary, our results show that integrating cache partitioning into a DBMS is worth the effort: It may improve but never degrades performance for arbitrary workloads containing scan-intensive, cache-polluting operators.

## Shared Loading

Bulk loading large volumes of data can stress the system significantly. When query processing and bulk loading run *in parallel*, both operations compete for resources causing resource contention and underutilization of the processor cores and network bandwidth. In a first step, we analyze the costs of bulk loading into a commercial, in-memory database system with a compression-optimized storage. Our analysis shows that most

processing time is spent on transforming data into the compressed storage format of the database system. Additionally, we confirm that, if state-of-the-art bulk loading and query processing run in parallel, both operations suffer from poor and unpredictable performance.

To address this problem, we exploit the given hardware setup, i.e., a distributed environment consisting of the DBMS server and one or more client machines holding the input data. Our distributed bulk loading mechanism, *Shared Loading*, leverages the compute power of the client to transform the input data *towards* the compressed storage format of the database system. In particular, it enables *dynamically offloading* parts of the bulk loading pipeline, i.e., deserialization and data transformation, at the granularity of column fragments. In our evaluation we determine that *Shared Loading* increases bulk loading throughput by up to 117 %, especially in slower network environments or when the DBMS is stressed. At the same time, it reduces a query workload’s tail latency by up to 60 %. Moreover, we demonstrate that *Shared Loading* works well with additional compression methods such as LZ4: It improves throughput by up to 61 % and tail latency by up to 62 %. In summary, *Shared Loading* enables efficient bulk loading by utilizing the available network bandwidth and the combined compute power of client and server more efficiently.

## 6.2 Future Directions

We identify several opportunities for extending the work of this thesis.

### Memory Tracing

Our memory tracing implementation minimizes runtime overhead. Another challenge is the large size of the trace data. While our solution can restrict memory tracing to specific processes or threads or to kernel-level or user-level program code, one might want to limit memory tracing to specific memory regions, such as heap memory, to reduce the size of the trace data. In addition, one might explore the pre-aggregation of recurring addresses<sup>1</sup>, lightweight compression techniques that exploit the similarity of memory addresses, or fast storage for moving data asynchronously from memory to storage. How and to what degree

---

<sup>1</sup>Note that aggregating recurring addresses might make it challenging to track access patterns.



this is possible without incurring a significant runtime overhead is an interesting question for future work.

We mainly focus on read-only data structures, e.g., on the encoded columns and dictionaries of a main-memory database system. Typically, objects such as columns or dictionaries have the same memory addresses for a long time. Other objects such as an operator's intermediate results or paged table data that is loaded from a buffer manager may change memory addresses frequently during runtime. Keeping track of frequently changing mappings between objects and their memory addresses is another challenge. Future work may develop efficient methods for tracking mappings as well as for post-processing mappings and trace data.

We demonstrate several use cases for memory tracing in our work. However, we argue that we merely scratch the surface of what is possible with efficient memory tracing. There are many features of the PEBS mechanism that we did not make use of, such as hardware performance counters for distinguishing accesses to DRAM and NVM DIMMs or the samples' timestamps to better analyze memory access patterns across different processor cores. Additionally, future hardware generations are expected to provide additional features: Variable-sized PEBS records may lower the overhead further; and extended PEBS support for all available hardware performance counters may allow memory tracing for a wide range of events related to, e.g., transactional memory operations or TLB misses [91].

Memory tracing reveals memory access statistics such as the quantity, frequency, and locality of memory accesses. One might use that information in multiple ways. A table's access statistics could be used to derive table partition schemes, to classify hot and cold data, or to choose the storage layout. A column's access statistics could be used to decide what type of index or what kind of compression should be used. An operator's access pattern could be used to decide how to partition the cache or how to co-schedule an operator with other operators.

In addition, memory tracing may open up new possibilities not only for software optimizations but also for hardware optimizations. Memory traces may guide optimizations for NUMA systems or systems with NVRAM together with a DRAM cache. Another use case could be self-tuning systems [36, 199, 212], where detailed access statistics could open up new possibilities to automatically adapt the system to hardware or workload characteristics. Furthermore, the low overhead of our memory tracing implementation might give incentive to revisit

related work [16, 57, 189, 195]. Last but not least, future work could explore whether lightweight memory tracing can run as a background task to provide access statistics during runtime.

## CPU Cache Partitioning

In our work we limit scan-intensive operations to a small portion of the last-level cache of a processor. We classify an operation as scan-intensive based on its memory access characteristics that we know at compile-time or that we can derive using a simple heuristic at runtime. It is a challenge to classify operations that have different memory access characteristics for different parameters. For example, access characteristics may depend on user input, such as user defined functions, or on the workload, e.g., on the data distribution or the number of concurrent users. How to classify operations that depend on various parameters and to what extent cache partitioning may improve performance of workloads with these types of operations is a possible question for future work. For a classification based on access characteristics, one might rely on memory tracing.

Since newer generations of processors feature an increasing number of cores sharing a cache [83], we expect that the negative effects of contention and cache pollution will continue to be a problem. Newer generations of processors may also suffer increasingly from cache pollution or cache trashing due to larger L1 or L2 caches shared by an increasing number of logical processor cores. Future work may explore how to address cache pollution across all cache levels, e.g., by leveraging hardware-based cache partitioning for the last-level cache and scheduling mechanisms [111, 215] for the remaining cache levels of the processor.

The idea of allocating resources or isolating resource usage for specific operations may be applicable in many other situations to improve quality of service. For example, it may pay off to restrict the DRAM bandwidth of memory-intensive background tasks, such as transforming data from write- to read-optimized storage. There are processors that already support hardware-based memory bandwidth allocation [91]. Similarly, it may pay off to reserve I/O quota exclusively for some operations, such as logging. Reserving I/O quota is already possible, e.g., with support from the operating system [132].

## Shared Loading

In our work we focus on loading *all* input data into the database system. However, a user may want to load only a *subset*, e.g., specific rows and columns. Related work on in-situ query processing [6, 100] proposes to push down predicate evaluation into the parsing and deserialization to skip individual rows or columns early. Skipping individual rows or columns on the client during bulk loading would significantly reduce network transfers. How and to what degree one might allow (lightweight) query processing on an external, untrustworthy client is a question for future work. Similarly, being able to evaluate a data partitioning strategy, such as range partitioning, on the client would make it possible to route data directly to the correct node of a distributed system.

We assume that the client holding the input data is not trustworthy from the perspective of the database system. If the assumption is relaxed, it would open up new possibilities. In addition to offloading deserialization and data transformation, one might explore the dynamic offloading of (partially) building indices, collecting statistics, or checking constraints without the need for expensive verification. Additionally, a trustworthy client could further optimize network communications by using RDMA operations to write data directly to the database server's main memory.

While we focus on loading data into a system, one could use a modified version of *Shared Loading* to extract data from a database system. Transforming the compressed storage format into the target format on the client could reduce network transfers and offload transformation costs. Future work could analyze the costs of exporting data from a database system and explore ways to dynamically offload expensive operations to the client or server.

In our work we leverage a heuristic that minimizes data size to illustrate how *Shared Loading* can decide on offloading at runtime. The heuristic causes the client to *partially* compute data transformations which improves throughput especially if network bandwidth is limited. However, our evaluation also shows that, in situations where network bandwidth is limited and the server is stressed significantly, it is best to *fully* transform data on the client. Finding a robust and precise mechanism that incorporates runtime information about the data, the available network bandwidth, the server's load factor, and possibly other indicators to decide the offloading dynamically is an interesting question for future work.

Ultimately, our approach of dynamically offloading work to different nodes matches the general tendency of exploiting network environments for resource management. A related research area is *in-network computing*, which aims at leveraging existing hardware of a distributed system, such as switches and network interface controllers, to execute programs *within* the network. Related work are in-network caching as a key-value store [98], in-network data aggregation with map-reduce jobs [177], and initial explorations of in-network, analytical query processing [115]. Another related research area is *resource disaggregation* for system architectures [69, 181]. The idea is to have a pool of standalone resource blades, containing, e.g., only processors, only memory, or only storage devices, that are interconnected using a network fabric—in contrast to grouping resources together in a single machine. Similarly, future work may explore network-centric system architectures and leverage network-centric data processing to distribute as many database operations as possible—in addition to bulk loading—across the network.

### 6.3 Discussion

Our contributions advance the field of resource-efficient data processing in multiple ways. First, our memory tracing implementation improves the ability to *understand complex systems*: It allows analyzing the software, i.e., the memory access pattern of algorithms for specific instances of data structures; it allows analyzing the workload, i.e., the quantity, frequency, or distribution of data accesses, which makes it possible to detect data and query skew; and it allows analyzing the hardware setup, i.e., it maps memory addresses to hardware performance counters that indicate, e.g., cache hits or misses or local or remote DRAM accesses. Having access to these detailed memory access characteristics provides new means to identify inefficient resource usage in complex system environments.

Second, we design and evaluate two approaches for avoiding resource contention and *maximizing resource utilization*: CPU cache partitioning and *Shared Loading*. Within a single processor, we avoid cache pollution. Our approach improves performance for concurrent workloads consisting of scan-intensive and cache-sensitive operations. Across multiple machines, we avoid resource contention of processor cores and network bandwidth. Our approach improves performance and predictability of bulk loading and query processing if both operations

compete for processor cores or if network bandwidth limits throughput. CPU cache partitioning and *Shared Loading* offer orthogonal solutions. They address different problems of inefficient resource usage and target different levels of the memory hierarchy. We argue that they could be combined to further increase resource efficiency, e.g., for concurrent workloads running in parallel to bulk loading.

Third, we argue that our approaches enable *efficient communication*. Efficient communication is particularly important with respect to the development of computer architectures. While newer generations of microprocessors have more and more compute power, memory and network bandwidth does not grow at the same rate (cf. Section 2.1). The gap between processor, memory, and network speed makes resource-efficient data processing necessary: To improve performance, we need to optimize data transfers. Our approaches optimize data transfers and improve performance by improving cache, memory, and network utilization. In particular, we demonstrate that resource-efficient data processing pays off: We achieve significant performance gains with reasonable effort. In addition, we argue that our work will have a lasting impact because efficient communication will continue to be important in the future.

Fourth, we claim that the contributions of this thesis have an *impact on real systems*. Our memory tracing implementation relies on the Linux kernel. Our evaluation of the implementation uses (a prototype version of) a commercial database system. In addition, we integrate CPU cache partitioning into (a prototype version of) a commercial database system and evaluate end-to-end performance characteristics. Similarly, the design and implementation of *Shared Loading* closely matches the architecture of a commercial database system. Thus, we demonstrate that the methods proposed in this thesis indeed improve the resource efficiency of database systems.



# Bibliography

- [1] Daniel J. Abadi, Samuel R. Madden, and Miguel Ferreira. “Integrating Compression and Execution in Column-oriented Database Systems”. In: *Proc. SIGMOD*. ACM, 2006, pp. 671–682. ISBN: 1-59593-434-0. DOI: [10.1145/1142473.1142548](https://doi.org/10.1145/1142473.1142548) (page 27).
- [2] Daniel J. Abadi, Samuel R. Madden, and Nabil Hachem. “Column-Stores vs. Row-Stores: How Different Are They Really?” In: *Proc. SIGMOD*. ACM, 2008, pp. 967–980. ISBN: 978-1-60558-102-6. DOI: [10.1145/1376616.1376712](https://doi.org/10.1145/1376616.1376712) (page 55).
- [3] Azza Abouzied, Daniel J. Abadi, and Avi Silberschatz. “Invisible Loading: Access-Driven Data Transfer From Raw Files Into Database Systems”. In: *Proc. EDBT*. 2013, pp. 1–10. ISBN: 978-1-4503-1597-5. DOI: [10.1145/2452376.2452377](https://doi.org/10.1145/2452376.2452377) (pages 88, 118).
- [4] Mark Adler and Jean-loup Gailly. *zlib Data Compression Library*. URL: <https://www.zlib.net/> (visited on 08/21/2019) (page 116).
- [5] Soramichi Akiyama and Takahiro Hirofuchi. “Quantitative Evaluation of Intel PEBS Overhead for Online System-Noise Analysis”. In: *Proc. ROSS*. ACM, 2017. ISBN: 9781450350860. DOI: [10.1145/3095770.3095773](https://doi.org/10.1145/3095770.3095773) (page 48).
- [6] Ioannis Alagiannis, Renata Borovica, Miguel Branco, Stratos Idreos, and Anastasia Ailamaki. “NoDB: Efficient Query Execution on Raw Data Files”. In: *Proc. SIGMOD*. ACM, 2012, pp. 241–252. ISBN: 978-1-4503-1247-9. DOI: [10.1145/2213836.2213864](https://doi.org/10.1145/2213836.2213864) (pages 88, 92, 118, 127).
- [7] Amazon. *EC2 Instance Types*. URL: <https://aws.amazon.com/ec2/instance-types/> (visited on 08/21/2019) (page 101).
- [8] Amazon. *Importing Data From Any Source to a MySQL or MariaDB DB Instance*. URL: <https://docs.aws.amazon.com/AmazonRDS/latest/UserGuide/MySQL.Procedural.Importing.AnySource.html> (visited on 08/21/2019) (pages 88, 89, 99).

- [9] AMD. *AMD64 Architecture Programmer's Manual: Volumes 1–5*. Apr. 2020. URL: <https://developer.amd.com/resources/developer-guides-manuals/> (visited on 05/01/2020) (pages 16, 19, 35).
- [10] AMD. *AMD64 Technology Platform Quality of Service Extensions*. Aug. 2018. URL: <https://developer.amd.com/wp-content/resources/56375.pdf> (visited on 03/31/2020) (page 67).
- [11] Editors of the American Heritage Dictionaries. *The American Heritage Dictionary of the English Language, Fifth Edition: Fiftieth Anniversary Printing*. Boston, MA, USA: Houghton Mifflin Harcourt, 2018. ISBN: 9781328841698 (page 3).
- [12] Lance Andersen. *JDBC 4.3 API Specification*. Oracle Corporation, July 2017 (page 117).
- [13] Mihnea Andrei, Christian Lemke, Günter Radestock, et al. “SAP HANA Adoption of Non-Volatile Memory”. In: *Proc. VLDB* (2017), pp. 1754–1765. ISSN: 2150-8097. DOI: [10.14778/3137765.3137780](https://doi.org/10.14778/3137765.3137780) (page 26).
- [14] Joy Arulraj, Andrew Pavlo, and Prashanth Menon. “Bridging the Archipelago between Row-Stores and Column-Stores for Hybrid Workloads”. In: *Proc. SIGMOD*. ACM, 2016, pp. 583–598. ISBN: 9781450335317. DOI: [10.1145/2882903.2915231](https://doi.org/10.1145/2882903.2915231) (pages 27, 28).
- [15] Jens Axboe. *fio – Flexible IO Tester*. URL: <http://git.kernel.dk/?p=fio.git> (visited on 08/21/2019) (page 100).
- [16] Grant Ayers, Heiner Litz, Christos Kozyrakis, and Parthasarathy Ranganathan. “Classifying Memory Access Patterns for Prefetching”. In: *Proc. ASPLOS*. ACM, 2020, pp. 513–526. ISBN: 9781450371025. DOI: [10.1145/3373376.3378498](https://doi.org/10.1145/3373376.3378498) (pages 49, 126).
- [17] Hermann Baer, Andy Witkowski, and Allen Brumm. *Performant and Scalable Data Loading with Oracle Database 12c*. Oracle White Paper, 2014 (page 118).
- [18] Jean-Loup Baer and Tien-Fu Chen. “Effective Hardware-Based Data Prefetching for High-Performance Processors”. In: *IEEE Trans. Comput.* 44.5 (May 1995), pp. 609–623. ISSN: 0018-9340. DOI: [10.1109/12.381947](https://doi.org/10.1109/12.381947) (page 24).



- [19] Cagri Balkesen, Gustavo Alonso, Jens Teubner, and M. Tamer Özsu. “Multi-Core, Main-Memory Joins: Sort vs. Hash Revisited”. In: *Proc. VLDB* (2013), pp. 85–96. ISSN: 2150-8097. DOI: [10.14778/2732219.2732227](https://doi.org/10.14778/2732219.2732227) (pages 7, 19, 82).
- [20] Yungang Bao, Mingyu Chen, Yuan Ruan, Li Liu, Jianping Fan, Qingbo Yuan, Bo Song, and Jianwei Xu. “HMTT: A Platform Independent Full-System Memory Trace Monitoring System”. In: *Proc. SIGMETRICS*. ACM, 2008, pp. 229–240. ISBN: 9781605580050. DOI: [10.1145/1375457.1375484](https://doi.org/10.1145/1375457.1375484) (page 47).
- [21] Claude Barthels, Simon Loesing, Gustavo Alonso, and Donald Kossmann. “Rack-Scale In-Memory Join Processing Using RDMA”. In: *Proc. SIGMOD*. ACM, 2015, pp. 1463–1475. ISBN: 9781450327589. DOI: [10.1145/2723372.2750547](https://doi.org/10.1145/2723372.2750547) (page 17).
- [22] Fabrice Bellard. “QEMU, a Fast and Portable Dynamic Translator”. In: *Proc. ATEC*. USENIX Association, 2005, p. 41 (page 47).
- [23] Michael A. Bender, Erik D. Demaine, and Martin Farach-Colton. “Cache-Oblivious B-Trees”. In: *SIAM J. Comput.* 35.2 (2005), pp. 341–358. DOI: [10.1137/S0097539701389956](https://doi.org/10.1137/S0097539701389956) (page 82).
- [24] Carsten Binnig, Stefan Hildenbrand, and Franz Färber. “Dictionary-Based Order-Preserving String Compression for Main Memory Column Stores”. In: *Proc. SIGMOD*. 2009, pp. 283–296. ISBN: 978-1-60558-551-2. DOI: [10.1145/1559845.1559877](https://doi.org/10.1145/1559845.1559877) (page 28).
- [25] Spyros Blanas, Kesheng Wu, Surendra Byna, Bin Dong, and Arie Shoshani. “Parallel Data Analysis Directly on Scientific File Formats”. In: *Proc. SIGMOD*. 2014, pp. 385–396. ISBN: 978-1-4503-2376-5. DOI: [10.1145/2588555.2612185](https://doi.org/10.1145/2588555.2612185) (page 118).
- [26] Alexander Böhm, Jens Dittrich, Niloy Mukherjee, Ippokratis Pandis, and Rajkumar Sen. “Operational Analytics Data Management Systems”. In: *Proc. VLDB* (2016), pp. 1601–1604. ISSN: 2150-8097. DOI: [10.14778/3007263.3007319](https://doi.org/10.14778/3007263.3007319) (page 26).
- [27] Martin Boissier, Rainer Schlosser, and Matthias Uflacker. “Hybrid Data Layouts for Tiered HTAP Databases with Pareto-Optimal Data Placements”. In: *Proc. ICDE*. IEEE Computer Society, 2018, pp. 209–220. DOI: [10.1109/ICDE.2018.00028](https://doi.org/10.1109/ICDE.2018.00028) (page 48).

- [28] Peter A. Boncz, Angelos-Christos Anatiotis, and Steffen Kläbe. “JCC-H: Adding Join Crossing Correlations with Skew to TPC-H”. In: *Performance Evaluation and Benchmarking for the Analytics Era*. Springer International Publishing, 2018, pp. 103–119. ISBN: 978-3-319-72401-0 (pages 37, 43).
- [29] Peter A. Boncz, Martin L. Kersten, and Stefan Manegold. “Breaking the Memory Wall in MonetDB”. In: *Commun. ACM* 51.12 (2008), pp. 77–85. ISSN: 0001-0782. DOI: [10 . 1145 / 1409360 . 1409380](https://doi.org/10.1145/1409360.1409380) (pages 27, 89, 119).
- [30] Peter A. Boncz, Marcin Zukowski, and Niels Nes. “MonetDB/X100: Hyper-Pipelining Query Execution”. In: *Proc. CIDR*. 2005, pp. 225–237. URL: <http://cidrdb.org/cidr2005/papers/P19.pdf> (page 7).
- [31] Shekhar Borkar and Andrew A. Chien. “The Future of Microprocessors”. In: *Commun. ACM* 54.5 (May 2011), pp. 67–77. ISSN: 0001-0782. DOI: [10.1145/1941487.1941507](https://doi.org/10.1145/1941487.1941507) (pages 15, 52, 65, 84).
- [32] Derek L. Bruening. “Efficient, Transparent, and Comprehensive Runtime Code Manipulation”. AAI0807735. PhD thesis. USA: Massachusetts Institute of Technology, 2004 (page 47).
- [33] Derek L. Bruening, Timothy Garnett, and Saman Amarasinghe. “An Infrastructure for Adaptive Dynamic Optimization”. In: *Proc. CGO*. IEEE Computer Society, 2003, pp. 265–275. ISBN: 076951913X (page 47).
- [34] Alexandre Bicas Caldeira, Bartłomiej Grabowski, Volker Haug, Marc-Eric Kahle, Andrew Laidlaw, Cesar Diniz Maciel, Monica Sanchez, and Seulgi Yoppy Sung. *IBM Power Systems S814 and S824 Technical Overview and Introduction*. IBM, Aug. 2014. URL: <https://www.redbooks.ibm.com/redpapers/pdfs/redp5097.pdf> (pages 18, 19).
- [35] S. Chatrchyan, G. Hmayakyan, V. Khachatryan, et al. “The CMS Experiment at the CERN LHC”. In: *Journal of Instrumentation* 3.08 (Aug. 2008), S08004. DOI: [10.1088/1748-0221/3/08/s08004](https://doi.org/10.1088/1748-0221/3/08/s08004) (page 1).
- [36] Surajit Chaudhuri and Vivek Narasayya. “Self-Tuning Database Systems: A Decade of Progress”. In: *Proc. VLDB*. 2007, pp. 3–14. ISBN: 9781595936493 (page 125).

- [37] Yu Cheng and Florin Rusu. “Parallel In-situ Data Processing with Speculative Loading”. In: *Proc. SIGMOD*. 2014, pp. 1287–1298. ISBN: 978-1-4503-2376-5. DOI: [10.1145/2588555.2593673](https://doi.org/10.1145/2588555.2593673) (pages 88, 92, 118).
- [38] Derek Chiou, Prabhat Jain, Larry Rudolph, and Srinivas Devadas. “Application-Specific Memory Management for Embedded Systems Using Software-Controlled Caches”. In: *Proc. DAC*. ACM, 2000, pp. 416–419. ISBN: 1-58113-187-9. DOI: [10.1145/337292.337523](https://doi.org/10.1145/337292.337523) (page 83).
- [39] Sangyeun Cho and Lei Jin. “Managing Distributed, Shared L2 Caches Through OS-Level Page Allocation”. In: *Proc. MICRO*. IEEE Computer Society, 2006, pp. 455–468. ISBN: 0-7695-2732-9. DOI: [10.1109/MICRO.2006.31](https://doi.org/10.1109/MICRO.2006.31) (page 82).
- [40] Young Hoon Cho, Gareth Coates, Bartłomiej Grabowski, and Volker Haug. *IBM Power Systems S922, S914, and S924: Technical Overview and Introduction*. IBM, July 2018. URL: <http://www.redbooks.ibm.com/redpapers/pdfs/redp5497.pdf> (pages 16, 19).
- [41] Hong-Tai Chou and David J. DeWitt. “An Evaluation of Buffer Management Strategies for Relational Database Systems”. In: *Proc. VLDB*. 1985, pp. 127–141 (page 84).
- [42] Cisco. *TPC Benchmark H Full Disclosure Report for Cisco UCS C480 M5 Rack-Mount Server using Microsoft SQL Server 2017 Enterprise Edition and Red Hat Enterprise Linux 7.6*. 2019. URL: <http://www.tpc.org/3337> (page 44).
- [43] Edgar F. Codd. “A Relational Model of Data for Large Shared Data Banks”. In: *Commun. ACM* 13.6 (June 1970), pp. 377–387. ISSN: 0001-0782. DOI: [10.1145/362384.362685](https://doi.org/10.1145/362384.362685) (page 1).
- [44] Edith Cohen. “All-Distances Sketches, Revisited: HIP Estimators for Massive Graphs Analysis”. In: *Proc. PODS*. 2014, pp. 88–99. ISBN: 978-1-4503-2375-8. DOI: [10.1145/2594538.2594546](https://doi.org/10.1145/2594538.2594546) (page 97).
- [45] Richard Cole and Vijaya Ramachandran. “Resource Oblivious Sorting on Multicores”. In: *ACM Trans. Parallel Comput.* 3.4 (Mar. 2017), 23:1–23:31. ISSN: 2329-4949. DOI: [10.1145/3040221](https://doi.org/10.1145/3040221) (page 82).
- [46] Yann Collet, et al. *LZ4*. URL: <http://www.lz4.org/> (visited on 08/21/2019) (pages 110, 116).

- [47] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. 3rd ed. The MIT Press, 2009. ISBN: 9780262033848 (page 98).
- [48] Mohammad Dashti, Alexandra Fedorova, Justin Funston, Fabien Gaud, Renaud Lachaize, Baptiste Lepers, Vivien Quema, and Mark Roth. “Traffic Management: A Holistic Approach to Memory Placement on NUMA Systems”. In: *Proc. ASPLOS*. ACM, 2013, pp. 381–394. ISBN: 9781450318709. DOI: [10.1145/2451116.2451157](https://doi.org/10.1145/2451116.2451157) (page 49).
- [49] Miyuru Dayarathna, Yonggang Wen, and Rui Fan. “Data Center Energy Consumption Modeling: A Survey”. In: *IEEE Commun. Surv. Tutorials* 18.1 (2016), pp. 732–794. DOI: [10.1109/COMST.2015.2481183](https://doi.org/10.1109/COMST.2015.2481183) (page 4).
- [50] Jeffrey Dean and Luiz André Barroso. “The Tail at Scale”. In: *Commun. ACM* 56.2 (2013), pp. 74–80. ISSN: 0001-0782. DOI: [10.1145/2408776.2408794](https://doi.org/10.1145/2408776.2408794) (pages 4, 102).
- [51] Justin DeBrabant, Andrew Pavlo, Stephen Tu, Michael Stonebraker, and Stanley B. Zdonik. “Anti-Caching: A New Approach to Database Management System Architecture”. In: *Proc. VLDB* (2013), pp. 1942–1953. DOI: [10.14778/2556549.2556575](https://doi.org/10.14778/2556549.2556575) (page 48).
- [52] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. “Dynamo: Amazon’s Highly Available Key-Value Store”. In: *Proc. SOSP*. 2007, pp. 205–220. ISBN: 978-1-59593-591-5. DOI: [10.1145/1294261.1294281](https://doi.org/10.1145/1294261.1294281) (pages 4, 102).
- [53] Peter J. Denning and Ted G. Lewis. “Exponential Laws of Computing Growth”. In: *Commun. ACM* 60.1 (Dec. 2016), pp. 54–65. ISSN: 0001-0782. DOI: [10.1145/2976758](https://doi.org/10.1145/2976758) (pages 4, 15).
- [54] Ulrich Drepper. *What Every Programmer Should Know About Memory*. 2007. URL: <https://people.freebsd.org/~lstewart/articles/cpumemory.pdf> (visited on 09/01/2020) (page 20).
- [55] Adam Dzierdzic, Manos Karpathiotakis, Ioannis Alagiannis, Raja Appuswamy, and Anastasia Ailamaki. “DBMS Data Loading: An Analysis on Modern Hardware”. In: *In Proc. ADMS/IMDM*. 2016, pp. 95–117. ISBN: 978-3-319-56111-0. DOI: [10.1007/978-3-319-56111-0\\_6](https://doi.org/10.1007/978-3-319-56111-0_6) (page 117).

- [56] S. Economo, D. Cingolani, A. Pellegrini, and F. Quaglia. “Configurable and Efficient Memory Access Tracing via Selective Expression-Based x86 Binary Instrumentation”. In: *Proc. MASCOTS*. 2016, pp. 261–270. doi: [10 . 1109 / MASCOTS . 2016 . 69](https://doi.org/10.1109/MASCOTS.2016.69) (page 47).
- [57] Ahmed Eldawy, Justin J. Levandoski, and Per-Åke Larson. “Trekking Through Siberia: Managing Cold Data in a Memory-Optimized Database”. In: *Proc. VLDB (2014)*, pp. 931–942. doi: [10.14778/2732967.2732968](https://doi.org/10.14778/2732967.2732968) (pages 48, 126).
- [58] Ramez Elmasri and Shamkant B. Navathe. *Fundamentals of Database Systems*. 7th ed. London, England: Pearson, 2016. ISBN: 9780133970777 (page 1).
- [59] Hadi Esmaeilzadeh, Emily Blem, Renee St. Amant, Karthikeyan Sankaralingam, and Doug Burger. “Dark Silicon and the End of Multicore Scaling”. In: *SIGARCH Comput. Archit. News* 39.3 (June 2011), pp. 365–376. ISSN: 0163-5964. doi: [10.1145/2024723.2000108](https://doi.org/10.1145/2024723.2000108) (pages 16, 52).
- [60] Facebook. *Zstandard – Fast Real-Time Compression Algorithm*. URL: <http://www.zstd.net/> (visited on 08/21/2019) (page 116).
- [61] Franz Färber, Alfons Kemper, Per-Åke Larson, Justin J. Levandoski, Thomas Neumann, and Andrew Pavlo. “Main Memory Database Systems”. In: *Found. Trends Databases* 8.1-2 (2017), pp. 1–130. doi: [10.1561/1900000058](https://doi.org/10.1561/1900000058) (page 25).
- [62] Franz Färber, Norman May, Wolfgang Lehner, Philipp Große, Ingo Müller, Hannes Rauhe, and Jonathan Dees. “The SAP HANA Database – An Architecture Overview”. In: *Data Eng. Bull.* 35.1 (2012), pp. 28–33. URL: <http://sites.computer.org/debull/A12mar/hana.pdf> (pages 26, 54, 89, 119).
- [63] Philippe Flajolet, Éric Fusy, Olivier Gandouet, and Frédéric Meunier. “Hyperloglog: The Analysis of a Near-Optimal Cardinality Estimation Algorithm”. In: *Proc. AOFA*. 2007 (page 97).
- [64] Jessie Frazelle. “Chipping Away at Moore’s Law”. In: *Queue* 18.1 (Feb. 2020), pp. 5–15. ISSN: 1542-7730. doi: [10.1145/3387945.3388515](https://doi.org/10.1145/3387945.3388515) (page 15).
- [65] Kelly A. Frazer, Dennis G. Ballinger, David R. Cox, et al. “A Second Generation Human Haplotype Map of Over 3.1 Million SNPs”. In: *Nature* 449.1 (2007), pp. 851–861. doi: [10.1038/nature06258](https://doi.org/10.1038/nature06258) (page 1).

- [66] Matteo Frigo, Charles E. Leiserson, Harald Prokop, and Sridhar Ramachandran. “Cache-Oblivious Algorithms”. In: *Proc. FOCS*. IEEE Computer Society, 1999, pp. 285–. ISBN: 0-7695-0409-4 (page 82).
- [67] Florian Funke, Alfons Kemper, and Thomas Neumann. “Compacting Transactional Data in Hybrid OLTP&OLAP Databases”. In: *Proc. VLDB* (2012), pp. 1424–1435. ISSN: 2150-8097. DOI: [10.14778/2350229.2350258](https://doi.org/10.14778/2350229.2350258) (pages 7, 41, 48).
- [68] Henning Funke, Sebastian Breß, Stefan Noll, Volker Markl, and Jens Teubner. “Pipelined Query Processing in Coprocessor Environments”. In: *Proc. SIGMOD*. ACM, 2018, pp. 1603–1618. DOI: [10.1145/3183713.3183734](https://doi.org/10.1145/3183713.3183734) (page 11).
- [69] Peter X. Gao, Akshay Narayan, Sagar Karandikar, Joao Carreira, Sangjin Han, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. “Network Requirements for Resource Disaggregation”. In: *Proc. OSDI*. USENIX Association, 2016, pp. 249–264. ISBN: 9781931971331 (page 128).
- [70] Chang Ge, Yinan Li, Eric Eilebrecht, Badrish Chandramouli, and Donald Kossmann. “Speculative Distributed CSV Data Parsing for Big Data Analytics”. In: *Proc. SIGMOD*. 2019, pp. 883–899. ISBN: 978-1-4503-5643-5. DOI: [10.1145/3299869.3319898](https://doi.org/10.1145/3299869.3319898) (pages 99, 118).
- [71] Kyle Geiger. *Inside ODBC*. Redmond, WA, USA: Microsoft Press, 1995. ISBN: 1-55615-815-7 (page 117).
- [72] Jana Giceva, Gustavo Alonso, Timothy Roscoe, and Tim Harris. “Deployment of Query Plans on Multicores”. In: *Proc. VLDB* (2014), pp. 233–244. DOI: [10.14778/2735508.2735513](https://doi.org/10.14778/2735508.2735513) (page 7).
- [73] Jana Giceva, Tudor-Ioan Salomie, Adrian Schüpbach, Gustavo Alonso, and Timothy Roscoe. “COD: Database / Operating System Co-Design”. In: *Proc. CIDR*. 2013. URL: [http://cidrdb.org/cidr2013/Papers/CIDR13%5C\\_Paper71.pdf](http://cidrdb.org/cidr2013/Papers/CIDR13%5C_Paper71.pdf) (page 7).
- [74] Google. *Snappy, a Fast Compressor/Decompressor*. URL: <https://github.com/google/snappy> (visited on 08/21/2019) (page 116).
- [75] Google. *Virtual Private Cloud Resource Quotas*. URL: <https://cloud.google.com/vpc/docs/quota> (visited on 08/21/2019) (page 101).
- [76] Brendan Gregg. *Systems Performance: Enterprise and the Cloud*. 1st. Prentice Hall Press, 2013. ISBN: 9780133390094 (page 102).



- [77] 802.3 WG – Ethernet Working Group. *802.3-2018 – IEEE Standard for Ethernet*. 2018. URL: [https://standards.ieee.org/standard/802\\_3-2018.html](https://standards.ieee.org/standard/802_3-2018.html) (page 17).
- [78] Martin Grund, Jens Krüger, Hasso Plattner, Alexander Zeier, Philippe Cudre-Mauroux, and Samuel R. Madden. “HYRISE: A Main Memory Hybrid Storage Engine”. In: *Proc. VLDB* (2010), pp. 105–116. ISSN: 2150-8097. DOI: [10.14778/1921071.1921077](https://doi.org/10.14778/1921071.1921077) (pages 27, 28).
- [79] Gabriel Haas, Michael Haubenschild, and Viktor Leis. “Exploiting Directly-Attached NVMe Arrays in DBMS”. In: *Proc. CIDR*. 2020. URL: <http://cidrdb.org/cidr2020/papers/p16-haas-cidr20.pdf> (page 18).
- [80] Theo Haerder and Andreas Reuter. “Principles of Transaction-Oriented Database Recovery”. In: *ACM Comput. Surv.* 15.4 (Dec. 1983), pp. 287–317. ISSN: 0360-0300. DOI: [10.1145/289.291](https://doi.org/10.1145/289.291) (page 26).
- [81] Milad Hashemi, Kevin Swersky, Jamie A. Smith, Grant Ayers, Heiner Litz, Jichuan Chang, Christos Kozyrakis, and Parthasarathy Ranganathan. “Learning Memory Access Patterns”. In: *Proc. ICML*. 2018, pp. 1924–1933. URL: <http://proceedings.mlr.press/v80/hashemi18a.html> (pages 24, 49).
- [82] Bingsheng He and Qiong Luo. “Cache-Oblivious Query Processing”. In: *Proc. CIDR*. 2007, pp. 44–55. URL: <http://cidrdb.org/cidr2007/papers/cidr07p05.pdf> (page 82).
- [83] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. 6th ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2017. ISBN: 0128119055 (pages 14, 19, 126).
- [84] A. Herdrich, E. Verplanke, P. Autee, R. Illikkal, C. Gianos, R. Singhal, and R. Iyer. “Cache QoS: From Concept to Reality in the Intel Xeon Processor E5-2600 v3 Product Family”. In: *Proc. HPCA*. IEEE Computer Society, Mar. 2016, pp. 657–668. DOI: [10.1109/HPCA.2016.7446102](https://doi.org/10.1109/HPCA.2016.7446102) (page 83).
- [85] Jason Hiebel, Laura E. Brown, and Zhenlin Wang. “Machine Learning for Fine-Grained Hardware Prefetcher Control”. In: *Proc. ICPP*. ACM, 2019. ISBN: 9781450362955. DOI: [10.1145/3337821.3337854](https://doi.org/10.1145/3337821.3337854) (page 49).

- [86] IBM. *IBM Db2 Version 11.1 Data Movement Utilities and Reference*. URL: [https://www.ibm.com/support/knowledgecenter/SSEPGG\\_11.1.0/com.ibm.db2.luw.admin.dm.doc/com.ibm.db2.luw.admin.dm.doc-gentopic1.html](https://www.ibm.com/support/knowledgecenter/SSEPGG_11.1.0/com.ibm.db2.luw.admin.dm.doc/com.ibm.db2.luw.admin.dm.doc-gentopic1.html) (visited on 08/21/2019) (page 117).
- [87] IDC. *Worldwide Ethernet Switch and Router Trackers*. May 2019. URL: <https://www.idc.com/getdoc.jsp?containerId=prUS45119319> (visited on 08/21/2019) (page 101).
- [88] Stratos Idreos, Fabian Groffen, Niels Nes, Stefan Manegold, K. Sjoerd Mullender, and Martin L. Kersten. “MonetDB: Two Decades of Research in Column-oriented Database Architectures”. In: *Data Eng. Bull.* 35.1 (2012), pp. 40–45. URL: <http://sites.computer.org/debull/A12mar/monetdb.pdf> (page 27).
- [89] Intel. *Improving Real-Time Performance by Utilizing Cache Allocation Technology*. White Paper, Apr. 2015. URL: <https://www.intel.com/content/www/us/en/communications/cache-allocation-technology-white-paper.html> (pages 20, 24, 53, 59, 67).
- [90] Intel. *Intel 64 and IA-32 Architectures Optimization Reference Manual*. May 2020. URL: <https://software.intel.com/en-us/articles/intel-sdm> (pages 17–20, 23, 24, 42).
- [91] Intel. *Intel 64 and IA-32 Architectures Software Developer’s Manual*. May 2020. URL: <https://software.intel.com/en-us/articles/intel-sdm> (pages 16, 23, 33–36, 68, 125, 126).
- [92] Intel. *Threading Building Blocks*. URL: <https://github.com/01org/tbb> (visited on 08/21/2019) (page 98).
- [93] Intel. *User Interface for Resource Allocation in Intel Resource Director Technology*. Documentation of the Linux Kernel, 2017. URL: [https://www.kernel.org/doc/Documentation/x86/intel\\_rdt\\_ui.txt](https://www.kernel.org/doc/Documentation/x86/intel_rdt_ui.txt) (visited on 10/01/2017) (page 68).
- [94] Intel. *VTune Profiler*. URL: <https://software.intel.com/vtune/> (visited on 03/31/2020) (pages 32, 33).
- [95] Intel. *VTune Profiler: Memory Access Analysis*. URL: <https://software.intel.com/en-us/vtune-help-memory-access-analysis> (visited on 03/31/2020) (pages 32, 34).
- [96] Zsolt István, David Sidler, and Gustavo Alonso. “Caribou: Intelligent Distributed Storage”. In: *Proc. VLDB* (2017), pp. 1202–1213. ISSN: 2150-8097. DOI: [10.14778/3137628.3137632](https://doi.org/10.14778/3137628.3137632) (page 7).



- [97] Akanksha Jain and Calvin Lin. “Linearizing Irregular Memory Accesses for Improved Correlated Prefetching”. In: *Proc. MICRO*. ACM, 2013, pp. 247–259. ISBN: 9781450326384. DOI: [10.1145/2540708.2540730](https://doi.org/10.1145/2540708.2540730) (page 24).
- [98] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. “NetCache: Balancing Key-Value Stores with Fast In-Network Caching”. In: *Proc. SOSP*. ACM, 2017, pp. 121–136. ISBN: 9781450350853. DOI: [10.1145/3132747.3132764](https://doi.org/10.1145/3132747.3132764) (page 128).
- [99] Kaan Kara, Jana Giceva, and Gustavo Alonso. “FPGA-Based Data Partitioning”. In: *Proc. SIGMOD*. ACM, 2017, pp. 433–445. ISBN: 9781450341974. DOI: [10.1145/3035918.3035946](https://doi.org/10.1145/3035918.3035946) (page 84).
- [100] Manos Karpathiotakis, Ioannis Alagiannis, and Anastasia Ailamaki. “Fast Queries over Heterogeneous Data Through Engine Customization”. In: *Proc. VLDB (2016)*, pp. 972–983. ISSN: 2150-8097. DOI: [10.14778/2994509.2994516](https://doi.org/10.14778/2994509.2994516) (pages 88, 127).
- [101] Christopher M. Kohlhoff. *Asio C++ Library*. URL: <https://github.com/chriskohlhoff/asio> (visited on 08/21/2019) (page 98).
- [102] Yong Sik Kwon, Joo Yeon Lee, Juchang Lee, Chulwon Lee, Christian Bensberg, Michael Muehle, Franz Färber, Wolfgang Lehner, and Arthur H. Lee. “SAP HANA Distributed In-Memory Database System: Transaction, Session, and Metadata Management”. In: *Proc. ICDE*. IEEE Computer Society, 2013, pp. 1165–1173. ISBN: 9781467349093. DOI: [10.1109/ICDE.2013.6544906](https://doi.org/10.1109/ICDE.2013.6544906) (page 26).
- [103] Renaud Lachaize, Baptiste Lepers, and Vivien Quéma. “MemProf: A Memory Profiler for NUMA Multicore Systems”. In: *Proc. ATC*. USENIX Association, 2012, p. 5 (pages 32, 48).
- [104] Tirthankar Lahiri, Shasank Chavan, Maria Colgan, et al. “Oracle Database In-Memory: A Dual Format In-Memory Database”. In: *Proc. ICDE*. 2015, pp. 1253–1258. DOI: [10.1109/ICDE.2015.7113373](https://doi.org/10.1109/ICDE.2015.7113373) (pages 27, 28).
- [105] Andrew Lamb, Matt Fuller, Ramakrishna Varadarajan, Nga Tran, Ben Vandiver, Lyric Doshi, and Chuck Bear. “The Vertica Analytic Database: C-store 7 Years Later”. In: *Proc. VLDB (2012)*, pp. 1790–1801. ISSN: 2150-8097. DOI: [10.14778/2367502.2367518](https://doi.org/10.14778/2367502.2367518) (pages 27, 28, 89, 119).

- [106] Harald Lang, Tobias Mühlbauer, Florian Funke, Peter A. Boncz, Thomas Neumann, and Alfons Kemper. “Data Blocks: Hybrid OLTP and OLAP on Compressed Storage Using Both Vectorization and Compilation”. In: *Proc. SIGMOD*. 2016, pp. 311–326. DOI: [10.1145/2882903.2882925](https://doi.org/10.1145/2882903.2882925) (page 27).
- [107] Geoff Langdale and Daniel Lemire. “Parsing Gigabytes of JSON per Second”. In: *VLDB Journal* 28.6 (Dec. 2019), pp. 941–960. ISSN: 0949-877X. DOI: [10.1007/s00778-019-00578-5](https://doi.org/10.1007/s00778-019-00578-5) (page 118).
- [108] Per-Åke Larson, Cipri Clinciu, Campbell Fraser, et al. “Enhancements to SQL Server Column Stores”. In: *Proc. SIGMOD*. ACM, 2013, pp. 1159–1168. ISBN: 9781450320375. DOI: [10.1145/2463676.2463708](https://doi.org/10.1145/2463676.2463708) (page 27).
- [109] Per-Åke Larson, Cipri Clinciu, Eric N. Hanson, Artem Oks, Susan L. Price, Srikumar Rangarajan, Aleksandras Surna, and Qingqing Zhou. “SQL Server Column Store Indexes”. In: *Proc. SIGMOD*. 2011, pp. 1177–1184. ISBN: 978-1-4503-0661-4. DOI: [10.1145/1989323.1989448](https://doi.org/10.1145/1989323.1989448) (pages 27, 89, 119).
- [110] Donghun Lee, Andrew Chang, Minseon Ahn, et al. “Optimizing Data Movement With Near-Memory Acceleration of In-memory DBMS”. In: *Proc. EDBT*. 2020, pp. 371–374. DOI: [10.5441/002/edbt.2020.35](https://doi.org/10.5441/002/edbt.2020.35) (page 84).
- [111] Rubao Lee, Xiaoning Ding, Feng Chen, Qingda Lu, and Xiaodong Zhang. “MCC-DB: Minimizing Cache Conflicts in Multi-Core Processors for Databases”. In: *Proc. VLDB (2009)*, pp. 373–384. ISSN: 2150-8097. DOI: [10.14778/1687627.1687670](https://doi.org/10.14778/1687627.1687670) (pages 7, 65, 66, 82, 83, 126).
- [112] Viktor Leis, Peter A. Boncz, Alfons Kemper, and Thomas Neumann. “Morsel-Driven Parallelism: A NUMA-Aware Query Evaluation Framework for the Many-Core Age”. In: *Proc. SIGMOD*. ACM, 2014, pp. 743–754. ISBN: 9781450323765. DOI: [10.1145/2588555.2610507](https://doi.org/10.1145/2588555.2610507) (page 19).
- [113] Viktor Leis, Michael Haubenschild, Alfons Kemper, and Thomas Neumann. “LeanStore: In-Memory Data Management beyond Main Memory”. In: *Proc. ICDE*. IEEE Computer Society, 2018, pp. 185–196. DOI: [10.1109/ICDE.2018.00026](https://doi.org/10.1109/ICDE.2018.00026) (page 48).

- [114] Christian Lemke, Kai-Uwe Sattler, Franz Färber, and Alexander Zeier. “Speeding up Queries in Column Stores: A Case for Compression”. In: *Proc. DaWaK*. Springer-Verlag, 2010, pp. 117–129. ISBN: 3642151043 (page 27).
- [115] Alberto Lerner, Rana Hussein, and Philippe Cudré-Mauroux. “The Case for Network Accelerated Query Processing”. In: *Proc. CIDR*. 2019. URL: <http://cidrdb.org/cidr2019/papers/p142-lerner-cidr19.pdf> (page 128).
- [116] Justin J. Levandoski, Per-Åke Larson, and Radu Stoica. “Identifying Hot and Cold Data in Main-Memory Databases”. In: *Proc. ICDE*. IEEE Computer Society, 2013, pp. 26–37. DOI: [10.1109/ICDE.2013.6544811](https://doi.org/10.1109/ICDE.2013.6544811) (pages 7, 48).
- [117] Feng Li, Sudipto Das, Manoj Syamala, and Vivek R. Narasayya. “Accelerating Relational Databases by Leveraging Remote Memory and RDMA”. In: *Proc. SIGMOD*. ACM, 2016, pp. 355–370. ISBN: 9781450335317. DOI: [10.1145/2882903.2882949](https://doi.org/10.1145/2882903.2882949) (page 17).
- [118] Letitia W. Li, Guillaume Duc, and Renaud Pacalet. “Hardware-Assisted Memory Tracing on New SoCs Embedding FPGA Fabrics”. In: *Proc. ACSAC*. ACM, 2015, pp. 461–470. ISBN: 9781450336826. DOI: [10.1145/2818000.2818030](https://doi.org/10.1145/2818000.2818030) (page 47).
- [119] Shih-wei Liao, Tzu-Han Hung, Donald Nguyen, Chinyen Chou, Chiaheng Tu, and Hucheng Zhou. “Machine Learning-Based Prefetch Optimization for Data Center Applications”. In: *Proc. SC*. ACM, 2009. ISBN: 9781605587448. DOI: [10.1145/1654059.1654116](https://doi.org/10.1145/1654059.1654116) (page 49).
- [120] Jiang Lin, Qingda Lu, Xiaoning Ding, Zhao Zhang, Xiaodong Zhang, and P. Sadayappan. “Gaining Insights into Multicore Cache Partitioning: Bridging the Gap Between Simulation and Real Systems”. In: *Proc. HPCA*. IEEE Computer Society, Feb. 2008, pp. 367–378. DOI: [10.1109/HPCA.2008.4658653](https://doi.org/10.1109/HPCA.2008.4658653) (page 67).
- [121] Linux Kernel Developers. *Perf*. URL: <https://perf.wiki.kernel.org/> (visited on 03/31/2020) (pages 32, 33).
- [122] Linux Kernel Developers. *Transparent Hugepage Support*. Documentation of the Linux Kernel. URL: <https://www.kernel.org/doc/Documentation/vm/transhuge.txt> (visited on 09/01/2020) (page 20).

- [123] David Lo, Liqun Cheng, Rama Govindaraju, Parthasarathy Ranganathan, and Christos Kozyrakis. “Heracles: Improving Resource Efficiency at Scale”. In: *Proc. ISCA*. ACM, 2015, pp. 450–462. ISBN: 9781450334020. DOI: [10 . 1145 / 2749469 . 2749475](https://doi.org/10.1145/2749469.2749475) (page 83).
- [124] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. “Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation”. In: *SIGPLAN Not.* 40.6 (June 2005), pp. 190–200. ISSN: 0362-1340. DOI: [10.1145/1064978.1065034](https://doi.org/10.1145/1064978.1065034) (pages 32, 34, 46, 47).
- [125] Clemens Lutz, Sebastian Breß, Steffen Zeuch, Tilmann Rabl, and Volker Markl. “Pump Up the Volume: Processing Large Data on GPUs with Fast Interconnects”. In: *Proc. SIGMOD*. ACM, 2020, pp. 1633–1649. ISBN: 9781450367356. DOI: [10 . 1145 / 3318464 . 3389705](https://doi.org/10.1145/3318464.3389705) (page 7).
- [126] Stefan Manegold, Peter A. Boncz, and Martin L. Kersten. “Generic Database Cost Models for Hierarchical Memory Systems”. In: *Proc. VLDB*. 2002, pp. 191–202. DOI: [10.5555/1287369.1287387](https://doi.org/10.5555/1287369.1287387) (pages 7, 65, 84).
- [127] Stefan Manegold, Peter A. Boncz, and Martin L. Kersten. “Optimizing Main-Memory Join on Modern Hardware”. In: *Trans. Know. and Data Eng.* 14.4 (July 2002), pp. 709–730. ISSN: 1041-4347. DOI: [10.1109/TKDE.2002.1019210](https://doi.org/10.1109/TKDE.2002.1019210) (page 82).
- [128] Stefan Manegold, Peter A. Boncz, and Martin L. Kersten. “What Happens During a Join? Dissecting CPU and Memory Optimization Effects”. In: *Proc. VLDB*. 2000, pp. 339–350. ISBN: 1558607153 (page 20).
- [129] Stefan Manegold, Peter A. Boncz, Niels Nes, and Martin Kersten. “Cache-Conscious Radix-Decluster Projections”. In: *Proc. VLDB*. 2004, pp. 684–695. ISBN: 0-12-088469-0. DOI: [10.5555/1316689.1316749](https://doi.org/10.5555/1316689.1316749) (page 55).
- [130] Jaydeep Marathe, Vivek Thakkar, and Frank Mueller. “Feedback-Directed Page Placement for CcNUMA via Hardware-Generated Memory Traces”. In: *J. Parallel Distrib. Comput.* 70.12 (Dec. 2010), pp. 1204–1219. ISSN: 0743-7315. DOI: [10.1016/j.jpdc.2010.08.015](https://doi.org/10.1016/j.jpdc.2010.08.015) (page 49).

- [131] Joe Mario. *C2C – False Sharing Detection in Linux Perf*. Sept. 2016. URL: <https://joemario.github.io/blog/2016/09/01/c2c-blog/> (visited on 03/31/2020) (pages 32, 36).
- [132] Paul Menage, Paul Jackson, and Christoph Lameter. *Control Groups*. Documentation of the Linux Kernel. URL: <https://www.kernel.org/doc/Documentation/cgroup-v1/cgroups.txt> (visited on 09/01/2020) (page 126).
- [133] Microsoft. *Microsoft SQL Server 2008: The Data Loading Performance Guide*. 2009. URL: [https://docs.microsoft.com/en-us/previous-versions/sql/sql-server-2008/dd425070\(v=sql.100\)](https://docs.microsoft.com/en-us/previous-versions/sql/sql-server-2008/dd425070(v=sql.100)) (visited on 08/21/2019) (page 89).
- [134] Microsoft. *Microsoft SQL Server 2017 Documentation*. URL: <https://docs.microsoft.com/en-us/sql/sql-server/sql-server-technical-documentation?view=sql-server-2017> (visited on 08/21/2019) (page 117).
- [135] Radhika Mittal, Alexander Shpiner, Aurojit Panda, Eitan Zahavi, Arvind Krishnamurthy, Sylvia Ratnasamy, and Scott Shenker. “Revisiting Network Support for RDMA”. In: *Proc. SIGCOMM*. ACM, 2018, pp. 313–326. ISBN: 9781450355674. DOI: [10.1145/3230543.3230557](https://doi.org/10.1145/3230543.3230557) (page 17).
- [136] Gordon E. Moore. “Cramming More Components onto Integrated Circuits, Reprinted from *Electronics*, Volume 38, Number 8, April 19, 1965, pp.114 ff.” In: *IEEE Solid-State Circuits Society Newsletter* 11.3 (Sept. 2006), pp. 33–35. ISSN: 1098-4232. DOI: [10.1109/N-SSC.2006.4785860](https://doi.org/10.1109/N-SSC.2006.4785860) (page 15).
- [137] Tobias Mühlbauer, Wolf Rödiger, Robert Seilbeck, Angelika Reiser, Alfons Kemper, and Thomas Neumann. “Instant Loading for Main Memory Databases”. In: *Proc. VLDB* (2013), pp. 1702–1713. ISSN: 2150-8097. DOI: [10.14778/2556549.2556555](https://doi.org/10.14778/2556549.2556555) (pages 88, 92, 99, 118).
- [138] Ingo Müller. “Engineering Aggregation Operators for Relational In-Memory Database Systems”. PhD thesis. Karlsruhe Institute of Technology, Germany, 2016 (page 14).
- [139] Ingo Müller and Cornelius Ratsch and Franz Färber. “Adaptive String Dictionary Compression in In-Memory Column-Store Database Systems”. In: *Proc. EDBT*. 2014, pp. 283–294. DOI: [10.5441/002/edbt.2014.27](https://doi.org/10.5441/002/edbt.2014.27) (page 28).

- [140] Ingo Müller, Peter Sanders, Arnaud Lacurie, Wolfgang Lehner, and Franz Färber. “Cache-Efficient Aggregation: Hashing Is Sorting”. In: *Proc. SIGMOD*. ACM, 2015, pp. 1123–1136. ISBN: 978-1-4503-2758-9. DOI: [10.1145/2723372.2747644](https://doi.org/10.1145/2723372.2747644) (pages 7, 82).
- [141] MySQL. *MySQL 8.0 Reference Manual: LOAD DATA Statement*. URL: <https://dev.mysql.com/doc/refman/8.0/en/load-data.html> (visited on 08/21/2019) (page 117).
- [142] MySQL. *MySQL 8.0 Reference Manual: The CSV Storage Engine*. URL: <https://dev.mysql.com/doc/refman/8.0/en/csv-storage-engine.html> (visited on 08/21/2019) (page 118).
- [143] Nicholas Nethercote and Julian Seward. “Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation”. In: *SIGPLAN Not.* 42.6 (June 2007), pp. 89–100. ISSN: 0362-1340. DOI: [10.1145/1273442.1250746](https://doi.org/10.1145/1273442.1250746) (pages 32, 34, 46, 47).
- [144] Thomas Neumann. “Efficiently Compiling Efficient Query Plans for Modern Hardware”. In: *Proc. VLDB* (2011), pp. 539–550. ISSN: 2150-8097. DOI: [10.14778/2002938.2002940](https://doi.org/10.14778/2002938.2002940) (page 7).
- [145] Thomas Neumann and Michael J. Freitag. “Umbra: A Disk-Based System with In-Memory Performance”. In: *Proc. CIDR*. 2020. URL: <http://cidrdb.org/cidr2020/papers/p29-neumann-cidr20.pdf> (pages 7, 42).
- [146] Stefan Noll, Henning Funke, and Jens Teubner. “Energy Efficiency in Main-Memory Databases”. In: *Datenbank-Spektrum* 17.3 (2017), pp. 223–232. DOI: [10.1007/s13222-017-0262-9](https://doi.org/10.1007/s13222-017-0262-9) (pages 10, 11).
- [147] Stefan Noll, Norman May, Alexander Böhm, Jan Mühlig, and Jens Teubner. “From the Application to the CPU: Holistic Resource Management for Modern Database Management Systems”. In: *Data Eng. Bull.* 42.1 (2019), pp. 10–21. URL: <http://sites.computer.org/debull/A19mar/p10.pdf> (page 11).
- [148] Stefan Noll, Jens Teubner, Norman May, and Alexander Böhm. “Accelerating Concurrent Workloads with CPU Cache Partitioning”. In: *Proc. ICDE*. 2018, pp. 437–448. DOI: [10.1109/ICDE.2018.00047](https://doi.org/10.1109/ICDE.2018.00047) (pages 9–11, 52, 122).
- [149] Stefan Noll, Jens Teubner, Norman May, and Alexander Böhm. “Analyzing Memory Accesses with Modern Processors”. In: *Proc. DaMoN*. ACM, 2020. ISBN: 9781450380249. DOI: [10.1145/3399666.3399896](https://doi.org/10.1145/3399666.3399896) (pages 9, 11, 32, 122).



- [150] Stefan Noll, Jens Teubner, Norman May, and Alexander Böhm. “Shared Load(ing): Efficient Bulk Loading into Optimized Storage”. In: *Proc. CIDR*. 2020. URL: <http://cidrdb.org/cidr2020/papers/p2-noll-cidr20.pdf> (pages 9–11, 88, 122).
- [151] Aleix Roca Nonell, Balazs Gerofi, Leonardo Bautista-Gomez, Dominique Martinet, Vicenç Beltran Querol, and Yutaka Ishikawa. “On the Applicability of PEBS Based Online Memory Access Tracking for Heterogeneous Memory Management at Scale”. In: *Proc. MCHPC*. ACM, 2018, pp. 50–57. ISBN: 9781450361132. DOI: [10.1145/3286475.3286477](https://doi.org/10.1145/3286475.3286477) (page 48).
- [152] Hideaki Ohno. *C++ Implementation of HyperLogLog Algorithm and HIP (Historic Inverse Probability) Estimator*. URL: <https://github.com/hideo55/cpp-HyperLogLog> (visited on 08/21/2019) (page 97).
- [153] Matthaios Olma, Manos Karpathiotakis, Ioannis Alagiannis, Manos Athanassoulis, and Anastasia Ailamaki. “Slalom: Coasting Through Raw Data via Adaptive Partitioning and Indexing”. In: *Proc. VLDB (2017)*, pp. 1106–1117. ISSN: 2150-8097. DOI: [10.14778/3115404.3115415](https://doi.org/10.14778/3115404.3115415) (page 88).
- [154] Oracle. *Oracle Database 18c Database Utilities: Part II SQL\*Loader*. URL: <https://docs.oracle.com/en/database/oracle/oracle-database/18/sutil/oracle-sql-loader.html> (visited on 08/21/2019) (pages 89, 117).
- [155] Ismail Oukid, Daniel Booss, Wolfgang Lehner, Peter Bumbulis, and Thomas Willhalm. “SOFORT: A Hybrid SCM-DRAM Storage Engine for Fast Data Recovery”. In: *Proc. DaMoN*. ACM, 2014. ISBN: 9781450329712. DOI: [10.1145/2619228.2619236](https://doi.org/10.1145/2619228.2619236) (page 26).
- [156] David A. Patterson and John L. Hennessy. *Computer Organization and Design: The Hardware/Software Interface*. 5th ed. The Morgan Kaufmann Series in Computer Architecture and Design. Academic Press, 2013. ISBN: 9780124077263 (page 14).
- [157] Aleksey Pesterev, Nickolai Zeldovich, and Robert T. Morris. “Locating Cache Performance Bottlenecks Using Data Profiling”. In: *Proc. EuroSys*. ACM, 2010, pp. 335–348. ISBN: 9781605585772. DOI: [10.1145/1755913.1755947](https://doi.org/10.1145/1755913.1755947) (pages 32, 49).
- [158] Orson Peters. *Pattern-Defeating Quicksort*. URL: <https://github.com/orlp/pdqsort> (visited on 08/21/2019) (page 98).

- [159] Ivan Luiz Picoli, Niclas Hedam, Philippe Bonnet, and Pinar Tözün. “Open-Channel SSD (What is it Good For)”. In: *Proc. CIDR*. 2020. URL: <http://cidrdb.org/cidr2020/papers/p17-picoli-cidr20.pdf> (page 18).
- [160] Meikel Poess and Raghunath Othayoth Nambiar. “Energy Cost, the Key Challenge of Today’s Data Centers: A Power Consumption Analysis of TPC-C Results”. In: *Proc. VLDB* (2008), pp. 1229–1240. ISSN: 2150-8097. DOI: [10.14778/1454159.1454162](https://doi.org/10.14778/1454159.1454162) (page 4).
- [161] Meikel Poess and Dmitry Potapov. “Data Compression in Oracle”. In: *Proc. VLDB*. 2003, pp. 937–947. ISBN: 0-12-722442-4 (pages 27, 89, 119).
- [162] Orestis Polychroniou and Kenneth A. Ross. “Vectorized Bloom Filters for Advanced SIMD Processors”. In: *Proc. DaMoN*. ACM, 2014, 6:1–6:6. ISBN: 978-1-4503-2971-2. DOI: [10.1145/2619228.2619234](https://doi.org/10.1145/2619228.2619234) (page 55).
- [163] PostgreSQL. *PostgreSQL 11 Documentation: COPY*. URL: <https://www.postgresql.org/docs/11/sql-copy.html> (visited on 08/21/2019) (page 117).
- [164] Iraklis Psaroudakis, Tobias Scheuer, Norman May, Abdelkader Sellami, and Anastasia Ailamaki. “Adaptive NUMA-Aware Data Placement and Task Scheduling for Analytical Workloads in Main-Memory Column-Stores”. In: *Proc. VLDB* (2016), pp. 37–48. ISSN: 2150-8097. DOI: [10.14778/3015274.3015275](https://doi.org/10.14778/3015274.3015275) (pages 7, 19).
- [165] Iraklis Psaroudakis, Tobias Scheuer, Norman May, Abdelkader Sellami, and Anastasia Ailamaki. “Scaling Up Concurrent Main-Memory Column-Store Scans: Towards Adaptive NUMA-Aware Data and Task Placement”. In: *Proc. VLDB* (2015), pp. 1442–1453. ISSN: 2150-8097. DOI: [10.14778/2824032.2824043](https://doi.org/10.14778/2824032.2824043) (pages 69, 70).
- [166] Moinuddin K. Qureshi and Yale N. Patt. “Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches”. In: *Proc. MICRO*. IEEE Computer Society, 2006, pp. 423–432. ISBN: 0-7695-2732-9. DOI: [10.1109/MICRO.2006.49](https://doi.org/10.1109/MICRO.2006.49) (page 83).
- [167] Mark Raasveldt and Hannes Mühleisen. “Data Management for Data Science – Towards Embedded Analytics”. In: *Proc. CIDR*. 2020. URL: <http://cidrdb.org/cidr2020/papers/p23-raasveldt-cidr20.pdf> (page 38).



- [168] Mark Raasveldt and Hannes Mühleisen. “Don’t Hold My Data Hostage: A Case for Client Protocol Redesign”. In: *Proc. VLDB* (2017), pp. 1022–1033. ISSN: 2150-8097. DOI: [10.14778/3115404.3115408](https://doi.org/10.14778/3115404.3115408) (pages 100, 119).
- [169] Raghu Ramakrishnan and Johannes Gehrke. *Database Management Systems*. 3rd ed. McGraw-Hill, 2003. ISBN: 978-0-07-115110-8 (page 25).
- [170] Ravishankar Ramamurthy, David J. DeWitt, and Qi Su. “A Case for Fractured Mirrors”. In: *Proc. VLDB*. 2002, pp. 430–441 (page 28).
- [171] Vijayshankar Raman, Gopi Attaluri, Ronald Barber, et al. “DB2 with BLU Acceleration: So Much More than Just a Column Store”. In: *Proc. VLDB* (2013), pp. 1080–1091. ISSN: 2150-8097. DOI: [10.14778/2536222.2536233](https://doi.org/10.14778/2536222.2536233) (pages 27, 28).
- [172] Jun Rao and Kenneth A. Ross. “Making B+-Trees Cache Conscious in Main Memory”. In: *Proc. SIGMOD*. 2000, pp. 475–486. ISBN: 1-58113-217-4. DOI: [10.1145/342009.335449](https://doi.org/10.1145/342009.335449) (page 29).
- [173] Steven J. Ross. *C++ Implementation of Templated Hybrid String Sort Algorithm in Boost Framework*. URL: [https://www.boost.org/doc/libs/1\\_69\\_0/libs/sort/doc/html/boost/spreadsort/string\\_sort\\_idp52153312.html](https://www.boost.org/doc/libs/1_69_0/libs/sort/doc/html/boost/spreadsort/string_sort_idp52153312.html) (visited on 08/21/2019) (page 98).
- [174] Steven J. Ross. “The Spreadsor High-Performance General-Case Sorting Algorithm”. In: *Proc. PDPTA*. CSREA Press, 2002, pp. 1100–1106. ISBN: 1-892512-89-0 (page 98).
- [175] Karl Rupp, M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten. *Microprocessor Trend Data*. 2020. URL: <https://github.com/karlrupp/microprocessor-trend-data> (visited on 09/15/2020) (page 16).
- [176] SAP. *SAP HANA SQL and System Views Reference for SAP HANA Platform 2.0 SPS 04*. URL: <https://help.sap.com/viewer/4fe29514fd584807ac9f2a04f6754767/2.0.04/en-US/b4b0eec1968f41a099c828a4a6c8ca0f.html> (visited on 08/21/2019) (page 117).
- [177] Amedeo Sapio, Ibrahim Abdelaziz, Abdulla Aldilaijan, Marco Canini, and Panos Kalnis. “In-Network Computation is a Dumb Idea Whose Time Has Come”. In: *Proc. HotNets*. ACM, 2017, pp. 150–156. ISBN: 9781450355698. DOI: [10.1145/3152434.3152461](https://doi.org/10.1145/3152434.3152461) (page 128).

- [178] Muhammad Aditya Sasongko, Milind Chabbi, Palwisha Akhtar, and Didem Unat. “ComDetective: A Lightweight Communication Detection Tool for Threads”. In: *Proc. SC*. ACM, 2019. ISBN: 9781450362290. DOI: [10.1145/3295500.3356214](https://doi.org/10.1145/3295500.3356214) (page 49).
- [179] Tobias Scheuer, Norman May, Alexander Böhm, and Daniel Scheibli. “JexLog: A Sonar for the Abyss”. In: *Proc. VLDB (2016)*, pp. 1493–1496. DOI: [10.14778/3007263.3007292](https://doi.org/10.14778/3007263.3007292) (page 32).
- [180] Harald Servat, Germán Llort, Juan González, Judit Giménez, and Jesús Labarta. “Low-Overhead Detection of Memory Access Patterns and Their Time Evolution”. In: *Euro-Par 2015: Parallel Processing*. Springer Berlin Heidelberg, 2015, pp. 57–69 (page 48).
- [181] Yizhou Shan, Yutong Huang, Yilun Chen, and Yiyang Zhang. “LegoOS: A Disseminated, Distributed OS for Hardware Resource Disaggregation”. In: *Proc. OSDI*. USENIX Association, 2018, pp. 69–87. ISBN: 9781931971478 (page 128).
- [182] Reza Sherkat, Colin Florendo, Mihnea Andrei, et al. “Native Store Extension for SAP HANA”. In: *Proc. VLDB (2019)*, pp. 2047–2058. ISSN: 2150-8097. DOI: [10.14778/3352063.3352123](https://doi.org/10.14778/3352063.3352123) (pages 28, 42).
- [183] Vishal Sikka, Franz Färber, Wolfgang Lehner, Sang Kyun Cha, Thomas Peh, and Christof Bornhövd. “Efficient Transaction Processing in SAP HANA Database: The End of a Column Store Myth”. In: *Proc. SIGMOD*. ACM, 2012, pp. 731–742. ISBN: 9781450312479. DOI: [10.1145/2213836.2213946](https://doi.org/10.1145/2213836.2213946) (page 26).
- [184] Avi Silberschatz, Henry F. Korth, and S. Sudarshan. *Database System Concepts*. 7th ed. McGraw-Hill Book Company, 2020. ISBN: 9780078022159 (pages 1, 25).
- [185] Sam Silvestro, Hongyu Liu, Tong Zhang, Changhee Jung, Dongyoon Lee, and Tongping Liu. “Sampler: PMU-Based Sampling to Detect Memory Errors Latent in Production Software”. In: *Proc. MICRO*. IEEE Press, 2018, pp. 231–244. ISBN: 9781538662403. DOI: [10.1109/MICRO.2018.00027](https://doi.org/10.1109/MICRO.2018.00027) (page 49).
- [186] L. Soares, D. Tam, and M. Stumm. “Reducing the Harmful Effects of Last-Level Cache Polluters with an OS-Level, Software-Only Pollute Buffer”. In: *Proc. MICRO*. Nov. 2008, pp. 258–269. DOI: [10.1109/MICRO.2008.4771796](https://doi.org/10.1109/MICRO.2008.4771796) (page 82).

- [187] Stephen Somogyi, Thomas F. Wenisch, Anastasia Ailamaki, and Babak Falsafi. “Spatio-Temporal Memory Streaming”. In: *SIGARCH Comput. Archit. News* 37.3 (June 2009), pp. 69–80. ISSN: 0163-5964. DOI: [10.1145/1555815.1555766](https://doi.org/10.1145/1555815.1555766) (page 24).
- [188] Michael Stonebraker and Ugur Cetintemel. ““One Size Fits All”: An Idea Whose Time Has Come and Gone”. In: *Proc. ICDE*. IEEE Computer Society, 2005, pp. 2–11. ISBN: 0769522858. DOI: [10.1109/ICDE.2005.1](https://doi.org/10.1109/ICDE.2005.1) (page 26).
- [189] Pengfei Su, Shasha Wen, Hailong Yang, Milind Chabbi, and Xu Liu. “Redundant Loads: A Software Inefficiency Indicator”. In: *Proc. ICSE*. IEEE Press, 2019, pp. 982–993. DOI: [10.1109/ICSE.2019.00103](https://doi.org/10.1109/ICSE.2019.00103) (pages 32, 49, 126).
- [190] Alex Szalay, Ani R. Thakar, and Jim Gray. “The sqlLoader Data-Loading Pipeline”. In: *Computing in Science and Engg.* 10.1 (2008), pp. 38–48. ISSN: 1521-9615. DOI: [10.1109/MCSE.2008.18](https://doi.org/10.1109/MCSE.2008.18) (page 88).
- [191] David K. Tam, Reza Azimi, Livio B. Soares, and Michael Stumm. “RapidMRC: Approximating L2 Miss Rate Curves on Commodity Systems for Online Optimizations”. In: *Proc. ASPLOS*. ACM, 2009. ISBN: 978-1-60558-406-5. DOI: [10.1145/1508244.1508259](https://doi.org/10.1145/1508244.1508259) (pages 48, 49, 82, 84).
- [192] Andrew S. Tanenbaum and David Wetherall. *Computer Networks*. 5th ed. Pearson Education, 2010. ISBN: 978-0-13-212695-3 (page 17).
- [193] George Taylor, Peter Davies, and Michael Farmwald. “The TLB Slice—a Low-Cost High-Speed Address Translation Mechanism”. In: *Proc. ISCA*. ACM, 1990, pp. 355–363. ISBN: 0-89791-366-3. DOI: [10.1145/325164.325161](https://doi.org/10.1145/325164.325161) (page 66).
- [194] Micron Technology. *M600 2.5-Inch SATA NAND Flash SSD*. 2014. URL: [https://media-www.micron.com/-/media/client/global/documents/products/data-sheet/ssd/m600\\_2\\_5\\_ssd.pdf?rev=ead5eb20949d47fcbbeb52e56ace0297](https://media-www.micron.com/-/media/client/global/documents/products/data-sheet/ssd/m600_2_5_ssd.pdf?rev=ead5eb20949d47fcbbeb52e56ace0297) (visited on 09/01/2020) (pages 17, 100).
- [195] Pinar Tözün, Islam Atta, Anastasia Ailamaki, and Andreas Moshovos. “ADDICT: Advanced Instruction Chasing for Transactions”. In: *Proc. VLDB* (2014), pp. 1893–1904. ISSN: 2150-8097. DOI: [10.14778/2733085.2733095](https://doi.org/10.14778/2733085.2733095) (pages 48, 126).

- [196] Pinar Tözün, Brian Gold, and Anastasia Ailamaki. “OLTP in Wonderland: Where Do Cache Misses Come from in Major OLTP Components?” In: *Proc. DaMoN*. ACM, 2013. ISBN: 9781450321969. DOI: [10.1145/2485278.2485286](https://doi.org/10.1145/2485278.2485286) (page 48).
- [197] Transaction Processing Performance Council (TPC). *TPC Benchmark H (Decision Support) Standard Specification Revision 2.17.1*. 2018. URL: [http://www.tpc.org/tpc\\_documents\\_current\\_versions/pdf/tpc-h\\_v2.17.1.pdf](http://www.tpc.org/tpc_documents_current_versions/pdf/tpc-h_v2.17.1.pdf) (pages 37, 71, 100).
- [198] F. Transier, C. Mathis, N. Bohnsack, and K. Stammerjohann. “Aggregation in Parallel Computation Environments with Shared Memory”. US Patent App. 12/978,194. 2012 (page 56).
- [199] Dana Van Aken, Andrew Pavlo, Geoffrey J. Gordon, and Bohan Zhang. “Automatic Database Management System Tuning Through Large-Scale Machine Learning”. In: *Proc. SIGMOD*. ACM, 2017, pp. 1009–1024. ISBN: 9781450341974. DOI: [10.1145/3035918.3064029](https://doi.org/10.1145/3035918.3064029) (pages 32, 125).
- [200] Vish Viswanathan, Karthik Kumar, Thomas Willhalm, Patrick Lu, and Blazej Filipiak. *Intel Memory Latency Checker v3.3*. Apr. 2017. URL: <https://software.intel.com/en-us/articles/intel-memory-latency-checker> (visited on 10/01/2017) (page 59).
- [201] Kefei Wang, Jian Liu, and Feng Chen. “Put an Elephant into a Fridge: Optimizing Cache Efficiency for In-Memory Key-Value Stores”. In: *Proc. VLDB (2020)*, pp. 1540–1554. ISSN: 2150-8097. DOI: [10.14778/3397230.3397247](https://doi.org/10.14778/3397230.3397247) (page 83).
- [202] Vince Weaver. *Linux Programmer’s Manual – perf\_event\_open*. URL: [http://man7.org/linux/man-pages/man2/perf\\_event\\_open.2.html](http://man7.org/linux/man-pages/man2/perf_event_open.2.html) (visited on 03/31/2020) (page 37).
- [203] Thomas Willhalm, Roman Dementiev, and Patrick Fay. *Intel Performance Counter Monitor*. Jan. 2017. URL: [www.intel.com/software/pcm](http://www.intel.com/software/pcm) (visited on 10/01/2017) (page 59).
- [204] Thomas Willhalm, Ismail Oukid, Ingo Müller, and Franz Färber. “Vectorizing Database Column Scans with Complex Predicates”. In: *ADMS*. 2013, pp. 1–12 (pages 28, 55, 60).
- [205] Thomas Willhalm, Nicolae Popovici, Yazan Boshmaf, Hasso Plattner, Alexander Zeier, and Jan Schaffner. “SIMD-Scan: Ultra Fast in-Memory Table Scan Using on-Chip Vector Processing Units”. In: *Proc. VLDB (2009)*, pp. 385–394. ISSN: 2150-8097. DOI: [10.14778/1687627.1687671](https://doi.org/10.14778/1687627.1687671) (pages 7, 28, 55, 60).

- [206] NVM Express Workgroup. *NVM Express Specifications*. URL: <https://nvmexpress.org/specifications/> (visited on 09/01/2020) (page 18).
- [207] William A. Wulf and Sally A. McKee. “Hitting the Memory Wall: Implications of the Obvious”. In: *SIGARCH Comput. Archit. News* 23.1 (Mar. 1995), pp. 20–24. ISSN: 0163-5964. DOI: [10.1145/216585.216588](https://doi.org/10.1145/216585.216588) (page 14).
- [208] Yaocheng Xiang, Xiaolin Wang, Zihui Huang, Zeyu Wang, Yingwei Luo, and Zhenlin Wang. “DCAPS: Dynamic Cache Allocation with Partial Sharing”. In: *Proc. EuroSys*. Porto, Portugal: ACM, 2018. ISBN: 9781450355841. DOI: [10.1145/3190508.3190511](https://doi.org/10.1145/3190508.3190511) (pages 49, 84).
- [209] Dong Xie, Badrish Chandramouli, Yinan Li, and Donald Kossman. “FishStore: Faster Ingestion With Subset Hashing”. In: *Proc. SIGMOD*. 2019, pp. 1711–1728. ISBN: 978-1-4503-5643-5. DOI: [10.1145/3299869.3319896](https://doi.org/10.1145/3299869.3319896) (page 118).
- [210] Yang Ye, Kenneth A. Ross, and Norases Vesdapunt. “Scalable Aggregation on Multicore Processors”. In: *Proc. DaMoN*. ACM, 2011, pp. 1–9. ISBN: 978-1-4503-0658-4. DOI: [10.1145/1995441.1995442](https://doi.org/10.1145/1995441.1995442) (page 56).
- [211] Erfan Zamanian, Xiangyao Yu, Michael Stonebraker, and Tim Kraska. “Rethinking Database High Availability with RDMA Networks”. In: *Proc. VLDB (2019)*, pp. 1637–1650. ISSN: 2150-8097. DOI: [10.14778/3342263.3342639](https://doi.org/10.14778/3342263.3342639) (page 17).
- [212] Ji Zhang, Yu Liu, Ke Zhou, et al. “An End-to-End Automatic Cloud Database Tuning System Using Deep Reinforcement Learning”. In: *Proc. SIGMOD*. ACM, 2019, pp. 415–432. ISBN: 9781450356435. DOI: [10.1145/3299869.3300085](https://doi.org/10.1145/3299869.3300085) (page 125).
- [213] Xiao Zhang, Sandhya Dwarkadas, and Kai Shen. “Towards Practical Page Coloring-Based Multicore Cache Management”. In: *Proc. EuroSys*. ACM, 2009, pp. 89–102. ISBN: 978-1-60558-482-9. DOI: [10.1145/1519065.1519076](https://doi.org/10.1145/1519065.1519076) (pages 67, 82).
- [214] Pin Zhou, Vivek Pandey, Jagadeesan Sundaresan, Anand Raghuraman, Yuanyuan Zhou, and Sanjeev Kumar. “Dynamic Tracking of Page Miss Ratio Curve for Memory Management”. In: *Proc. ASPLOS*. ACM, 2004, pp. 177–188. ISBN: 1-58113-804-0. DOI: [10.1145/1024393.1024415](https://doi.org/10.1145/1024393.1024415) (page 84).

- [215] Sergey Zhuravlev, Sergey Blagodurov, and Alexandra Fedorova. “Addressing Shared Resource Contention in Multicore Processors via Scheduling”. In: *Proc. ASPLOS*. ACM, 2010, pp. 129–142. ISBN: 978-1-60558-839-1. DOI: [10.1145/1736020.1736036](https://doi.org/10.1145/1736020.1736036) (pages 82, 126).
- [216] Tobias Ziegler, Sumukha Tumkur Vani, Carsten Binnig, Rodrigo Fonseca, and Tim Kraska. “Designing Distributed Tree-Based Index Structures for Fast RDMA-Capable Networks”. In: *Proc. SIGMOD*. ACM, 2019, pp. 741–758. ISBN: 9781450356435. DOI: [10.1145/3299869.3300081](https://doi.org/10.1145/3299869.3300081) (page 17).

# List of Figures

1.1	Areas of the main contributions of this thesis. . . . .	9
2.1	Memory hierarchy across different compute components. . . . .	15
2.2	Trend data of microprocessors from 1972 to 2020. . . . .	16
2.3	Access latency of a pointer-chasing benchmark. . . . .	21
2.4	Access latency of a pointer-chasing benchmark suffering from cache pollution. . . . .	23
2.5	Order-preserving dictionary compression. . . . .	27
2.6	Buffered updates in SAP HANA. . . . .	29
3.1	Difference between instruction-based and memory-based profiling. . . . .	33
3.2	Overview of the PEBS mechanism. . . . .	35
3.3	Overview of our memory tracing implementation. . . . .	36
3.4	Memory trace of running DuckDB. . . . .	39
3.5	Memory trace of running SAP HANA's aggregation operator. . . . .	40
3.6	Analysis of the working set size of the JCC-H benchmark running on SAP HANA. . . . .	42
3.7	Memory trace of running the JCC-H benchmark with SAP HANA. . . . .	43
3.8	Impact of partitioning on l_shipdate. . . . .	44
3.9	Impact of partitioning on l_orderkey. . . . .	45
3.10	Impact of partitioning on o_orderdate. . . . .	46
4.1	Impact of cache pollution on an OLTP query. . . . .	53
4.2	Overview of cache partitioning. . . . .	54
4.3	Normalized throughput of the column scan at varying LLC sizes. . . . .	60
4.4	4 MiB dictionary: normalized throughput of aggregation with grouping at varying LLC sizes. . . . .	61

4.5	40 MiB dictionary: normalized throughput of aggregation with grouping at varying LLC sizes. . . . .	62
4.6	400 MiB dictionary: normalized throughput of aggregation with grouping at varying LLC sizes. . . . .	63
4.7	Normalized throughput of the foreign key join at varying LLC sizes. . . . .	64
4.8	Simplified example of using Intel's Cache Allocation Technology. . . . .	67
4.9	Integration of cache partitioning into SAP HANA. . . . .	70
4.10	4 MiB: normalized throughput of the column scan and aggregation with grouping with and without cache partitioning. . . . .	73
4.11	40 MiB: normalized throughput of the column scan and aggregation with grouping with and without cache partitioning. . . . .	74
4.12	400 MiB: normalized throughput of the column scan and aggregation with grouping with and without cache partitioning. . . . .	75
4.13	10 <sup>6</sup> primary keys: normalized throughput of aggregation with grouping and the foreign key join with and without cache partitioning. . . . .	76
4.14	10 <sup>8</sup> primary keys: normalized throughput of aggregation with grouping and the foreign key join with and without cache partitioning. . . . .	77
4.15	Normalized throughput of the column scan and each TPC-H query with and without cache partitioning. . . . .	78
4.16	Normalized throughput of the column scan and an OLTP query with and without cache partitioning. . . . .	80
5.1	Impact of bulk loading on query processing. . . . .	89
5.2	Processing steps for bulk loading data from a file into optimized storage. . . . .	90
5.3	Cost analysis of bulk loading into SAP HANA. . . . .	91
5.4	Architectural overview of <i>Shared Loading</i> . . . . .	94
5.5	Storage layout of <i>Shared Loading</i> . . . . .	95
5.6	Results of bulk loading the lineitem table over a 10-Gbit network without query processing. . . . .	102
5.7	Results of bulk loading the warehouse table over a 10-Gbit network without query processing. . . . .	103
5.8	Results of bulk loading the lineitem table over a 1-Gbit network without query processing. . . . .	104



5.9	Results of bulk loading the warehouse table over a 1-Gbit network without query processing. . . . .	104
5.10	Results of bulk loading the lineitem table over a 10-Gbit network with concurrent query processing. . . . .	105
5.11	Results of bulk loading the warehouse table over a 10-Gbit network with concurrent query processing. . . . .	106
5.12	Results of bulk loading the lineitem table over a 1-Gbit network with concurrent query processing. . . . .	107
5.13	Results of bulk loading the warehouse table over a 1-Gbit network with concurrent query processing. . . . .	107
5.14	Results of bulk loading the lineitem and warehouse table over a 10-Gbit network with varying chunk sizes. . . . .	108
5.15	Results of bulk loading the lineitem and warehouse table over a 10-Gbit network with varying partition sizes. . . . .	109
5.16	Results of bulk loading the lineitem table over a 10-Gbit network without query processing and with LZ4 compression. . . . .	111
5.17	Results of bulk loading the warehouse table over a 10-Gbit network without query processing using LZ4 compression. . . . .	111
5.18	Results of bulk loading the lineitem table over a 1-Gbit network without query processing using LZ4 compression. . . . .	112
5.19	Results of bulk loading the warehouse table over a 1-Gbit network without query processing using LZ4 compression. . . . .	112
5.20	Results of bulk loading the lineitem table over a 10-Gbit network with concurrent query processing using LZ4 compression. . . . .	114
5.21	Results of bulk loading the warehouse table over a 10-Gbit network with concurrent query processing using LZ4 compression. . . . .	114
5.22	Results of bulk loading the lineitem table over a 1-Gbit network with concurrent query processing using LZ4 compression. . . . .	115
5.23	Results of bulk loading the warehouse table over a 1-Gbit network with concurrent query processing using LZ4 compression. . . . .	115
6.1	Areas of the main contributions of this thesis. . . . .	122



# List of Tables

2.1	Capacity, latency, and bandwidth of memory components.	17
3.1	Tracing overhead for different thresholds for the PEBS mechanism. . . . .	47
5.1	The total amount of data a bulk loading configuration transfers over the network. . . . .	105
5.2	The total amount of data a bulk loading configuration transfers over the network with LZ4 compression. . . .	113



# List of Listings

3.1	SQL query executed with DuckDB. . . . .	38
3.2	SQL query executed with SAP HANA. . . . .	40
4.1	The operator-specific SQL queries of the experimental analysis. . . . .	57
4.2	The different SQL table schemata used in the experi- mental analysis. . . . .	58
5.1	Analytical queries inspired by the TPC-H benchmark. .	101

