Memory Footprint Reduction of Cloud Databases with Automated Physical Database Design

Doctoral thesis for obtaining the academic degree
Doctor of Natural Science (Dr. rer. nat.)

submitted by

Michael Brendle

at the

Faculty of Sciences
Department of Computer and Information Science

Konstanz, 2022

Date of the oral examination: July 29, 2022

1. Reviewer: Prof. Dr. Michael Grossniklaus
2. Reviewer: Prof. Dr. Guido Moerkotte

MICHAEL BRENDLE

MEMORY FOOTPRINT REDUCTION OF CLOUD DATABASES WITH AUTOMATED PHYSICAL DATABASE DESIGN

Michael Brendle: *Memory Footprint Reduction of Cloud Databases with Automated Physical Database Design*

To my family

ABSTRACT

Enterprises increasingly move their application data into the cloud by employing data management offerings of database-as-a-service providers, who specialize in hosting and managing database instances. While being obligated to certain performance commitments stipulated in service-level agreements (SLAs) with the customer, database-as-a-service providers are incentivized to minimize internal costs to enhance profitability. Since database workloads are often skewed and DRAM constitutes the primary driver of hardware costs, internal costs can be reduced by evicting rarely accessed (cold) data to cheaper storage layers. Hence, only frequently accessed (hot) data should remain in DRAM. As most database management systems employ a buffer manager to load and evict data at page granularity from secondary storage to DRAM and vice versa, keeping only disk pages with hot data in DRAM can lead to substantial cost savings. The physical database schema, however, is usually not defined according to the data access pattern. Therefore, cold data may be stored on the same disk page as hot data. As we will elaborate, polluting the buffer pool with cold data wastes DRAM capacities, which offers a largely untapped cost reduction potential to database-as-a-service providers.

In this dissertation, we aim at utilizing the buffer manager more efficiently by recommending a physical database schema in which hot disk pages contain mainly hot data. Our approach is based on the observation that, in most cases, rows of a table are accessed either frequently or rarely according to a value range of a specific column of that table. In order to identify hot and cold data, we introduce a *statistics collector* that gathers accurate workload statistics with low memory and runtime overhead. By exploiting the collected statistics, our *table partitioning advisor* proposes a range partitioning layout by grouping rows into partitions belonging to hot- or cold-classified value ranges. As a result, hot range partitions with a high density of hot data stay in DRAM, whereas cold range partitions are evicted to cheaper storage layers. This prevents the pollution of the buffer pool with cold data. In addition, we periodically optimize the physical schema in light of workload changes over time. We propose a forward-looking approach by developing a *workload predictor*, which forecasts the approximate future workload. The predicted workload is then fed into our table partitioning advisor. Finally, we implement our ideas into a prototype of a commercial database and showcase its applicability by incorporating a real-world database workload. Our experimental evaluation demonstrates a buffer pool size reduction of up to 3.2× compared to related approaches while still adhering to SLAs.

ZUSAMMENFASSUNG

Unternehmen verlagern ihre Anwendungsdaten zunehmend in die Cloud, indem sie Angebote von Dienstleistern nutzen, die sich auf das Bereitstellen und Verwalten von Datenbankmanagementsystemen in der Cloud spezialisiert haben. Diese Dienstleister sind zwar über Vereinbarungen mit den Kunden an bestimmte Leistungsverpflichtungen gebunden, versuchen jedoch innerhalb dieser Rahmenbedingen durch interne Kostenminimierung ihre eigene Rentabilität zu steigern. Da Zugriffe auf Daten in Datenbankmanagementsystemen oft ungleichmäßig verteilt sind und der Hauptspeicher der primäre Kostentreiber von Hardware ist, können die internen Kosten gesenkt werden, indem Daten mit wenigen Zugriffen (kalte Daten) auf kostengünstigere Speichermedien ausgelagert werden. In Folge dessen sollten nur die sogenannten heißen Daten, auf die häufig zugegriffen wird, im Hauptspeicher bleiben. Da die meisten Datenbankmanagementsysteme einen Zwischenspeicher einsetzen, welcher Daten in der Granularität einer Seite vom Sekundärspeicher in den Hauptspeicher lädt, können erhebliche Kosteneinsparungen erzielt werden, indem nur Seiten mit heißen Daten im Hauptspeicher gehalten werden. Jedoch ist das physische Datenbankschema in der Regel nicht entsprechend dem Datenzugriffsmuster definiert. Daher können kalte Daten auf derselben Seite wie heiße Daten gespeichert sein. Somit wird unnötigerweise Hauptspeicher verschwendet, welcher Dienstleistern von Datenbankmanagementsystemen in der Cloud ein weitgehend ungenutztes Kosteneinsparungspotenzial bietet.

In dieser Dissertation ist unser Ziel, den Zwischenspeicher eines Datenbankmanagementsystems effizienter zu nutzen. Wir schlagen dabei ein physisches Datenbankschema vor, in dem heiße Seiten hauptsächlich heiße Daten enthalten. Unser Ansatz basiert auf der Beobachtung, dass in den meisten Fällen auf Zeilen einer Tabelle entweder häufig oder selten zugegriffen wird, entsprechend einem Wertebereich einer bestimmten Spalte dieser Tabelle. Um heiße und kalte Daten zu identifizieren, entwickeln wir einen Statistiksammler, der präzise Statistiken über das Datenzugriffsmuster erfasst und gleichzeitig nur wenige zusätzliche Speicher- und Berechnungsressourcen beansprucht. Unser automatisierter Ratgeber nutzt dann die gesammelten Statistiken und schlägt eine Partitionierung vor, indem Zeilen in Partitionen gruppiert werden, die entweder zu heiß oder kalt klassifizierten Wertebereichen gehören. Dadurch verbleiben Partitionen mit einer hohen Dichte an heißen Daten im Hauptspeicher, während Partitionen mit ausschließlich kalten Daten auf kostengünstigere Speichermedien ausgelagert werden. Dies verhindert das Füllen des Zwischenspeichers mit kalten

Daten. Außerdem optimieren wir das physische Datenbankschema in regelmäßigen Abständen angesichts des sich im Laufe der Zeit ändernden Datenzugriffsmusters. Dabei schlagen wir einen vorausschauenden Ansatz vor, bei dem wir die zukünftigen Anfragen an die Datenbank vorhersagen, welche dann von unserem Tabellenpartitionierungsratgeber verwendet werden. Abschließend implementieren wir unsere Ideen in einen Prototyp eines kommerziellen Datenbankmanagementsystems und evaluieren die praktische Anwendbarkeit mit Hilfe von praxisnahen Benchmark-Tests. Unsere Evaluierung zeigt dabei eine Reduzierung des Zwischenspeichers um bis zu das 3,2-fache im Vergleich zu existierenden Ansätzen bei gleichzeitiger Einhaltung der Leistungsverpflichtungen.

ACKNOWLEDGMENTS

First and foremost, I would like to thank my advisor and mentor, Michael Grossniklaus, for his outstanding support over the last few years. From the moment we first met – I was looking for a Bachelor's Project – a long and enlightening journey has started. Michael constantly pushed my ideas, provided excellent guidance, introduced me to the SAP HANA Database Campus, and motivated me to believe in myself and my work. In addition to productive on-topic discussions, there was always time for private conversations, especially about football games. I am entirely convinced that the next championship for the FC Basel will happen soon.

I would also like to express my gratitude to Guido Moerkotte. Once for taking on the role of co-examiner but, far more importantly, for his incredible interest and passion in my research. I benefited immensely from Guido's experience throughout this journey, from notating definitions to writing concise and precise explanations. Additionally, his ongoing questioning of the benefits and drawbacks of the proposed ideas indescribably improved this work.

Next, I would like to give many thanks to Norman May as the primary research driver behind the collaboration with SAP. Norman showed indescribable encouragement and expertise in supervising my work from both the industry and the research side from the very beginning. In addition to the business aspect, Norman added to this work, his productive and valuable feedback substantially improved the papers and this dissertation.

Besides Norman, I am very thankful to former SAP colleagues and advisors: Robert Schulze, Alexander Boehm, and Ismail Oukid. I always enjoyed the open and fruitful discussions during our meetings. Furthermore, Robert's grateful effort and commitment to discussing how to implement my ideas into a prototype of SAP HANA are indescribable inputs to this work.

Throughout my dissertation, I also had the privilege of supervising three Master's students, Nick Weber, Mahammad Valiyev, and Roman Diachuk, who supported my research. I am very grateful for their outcome [47, 160, 164] and would like to thank Nick for our conservation on a cost model, Mahammad for discussing optimal range partitionings, and Roman for debating detecting workload drift types.

When Michael Grossniklaus introduced me to SAP, a new chapter of my life started in Walldorf. I was welcomed nicely from the beginning, and my special thanks go to Florian Wolf. He was a role model and supervised my Master's thesis on robustness metrics in an outstanding manner, which resulted in my first research paper. In addition, we had

a wonderful time optimizing a research prototype for the SIGMOD programming contest and were invited to Houston as one of the finalists.

I would also like to thank Arne Schwarz as the head of the campus, which made the funding of this dissertation and the collaboration with SAP possible. He also guaranteed that the work-life balance on the campus was not neglected, e.g., through our annual team event.

While working at the campus, I had the privilege to benefit from the experience and discussions with my fellow Ph.D. students: Thomas Bach, Tiemo Bang, Jonas Dann, Matthias Hauck, Axel Hertzschuch, Neetha Jambigi, Hanna Kruppe, Lukas Landgraf, Robert Lasch, Lucas Lersch, Mehdi Moghaddamfar, Stefan Noll, Georgios Psaropoulos, Robin Rehrmann, and Frank Tetzel. I want to thank Jonas, Neetha, Hanna, Robert, Lukas, Mehdi, and Stefan for giving me feedback on an early draft of this dissertation. Further, I am extremely thankful to Tiemo, Stefan, and Frank for being good friends and spending time off work, e.g., having board game evenings, barbecues, or hiking trips.

I would also like to thank my colleagues from the University of Konstanz, Mehmet Aytimur, Johann Bornholdt, Theodoros Chondrogiannis, Manuel Hotz, Silvan Reiner, and Leonard Wörteler, for their collaboration and pleasant time at conferences. In addition, I am grateful to Marc H. Scholl for taking on the role of chairman of the doctoral examination committee.

Although writing a dissertation is an instructive journey, it is also energy-sapping. Therefore, to recharge my batteries, my table tennis teammates at DJK Dossenheim and TV Langenargen made a priceless impact on this dissertation. Especially, I would like to thank Leo Beyer for his indescribable effort while proofreading chapters of this dissertation.

Extraordinary thanks go to my long-time friends Uwe Moser, Joel Scheuner, and Julian Bühler for being there and having a great time together. Particularly, I am grateful to Uwe for his endless encouragement throughout my life, regardless of whether times were good or bad. With Joel, I already had awesome trips to the national parks in the USA and a magic winter wonderland in Lapland. Julian is keen on watching sports like me, and we have already enjoyed fascinating sports events around Europe.

Finally, I would like to thank my parents, Gisela and Franz, for fostering my siblings, Sabine and Manuel, and me during our whole life, encouraging us to pursue our goals, and for their unconditional love. Mum, Dad: We owe you a lot. I am also glad for my niece Emma and nephews Jonas and Matti. I wish you all the best for the future.

CONTENTS

1 INTRODUCTION

Data management is essential in most of today's business processes. Enterprises typically run their data-driven applications on top of a database management system such as IBM DB 2 [33], Microsoft SQL Server [112], Oracle Database [127], SAP HANA [54], or PostgreSQL [155]. Alibaba's e-commerce web application, for instance, manages data about their products, customers, and orders in an optimized version of MySQL [76]. There are numerous benefits of employing a database management system for data-driven applications and end-users in general. First, database management systems promise high performance and scalability. Second, the validity of managed data despite errors and failures can be guaranteed by fulfilling the ACID properties of atomicity, consistency, isolation, and durability [68]. Third, the complexity of data management is hidden from the end-user as they simply formulate their data requests by a declarative language like SQL [32]. Finally, database management systems can extract valuable business intelligence.

Most database management systems employ the *relational model* proposed by E. F. Codd to model the application domain [35]. A single relation is a table (e.g., ordered products), where each row relates to a tuple (e.g., a single order) and each column to an attribute (e.g., the order date). The specification of all tables derived from the application domain and their relationships to each other are defined as the *logical schema*. Database administrators aim at finding a logical schema that eliminates redundancy, e.g., by decomposing each relation into the third-normal form [36]. This is because redundancy causes redundant storage and a loss of data integrity due to data anomalies.

Once the logical schema is chosen, the *physical schema* defines how tables are stored on disk and how data can be accessed. The physical schema considers aspects like compression, partitioning, or the selection of indexes. Although some aspects are fixed by the database (e.g., dictionary compression in SAP HANA [54]), database administrators are responsible for mutable aspects such as selecting indexes or specifying partitionings. Since the physical schema strongly impacts performance and memory characteristics [103], database administrators focus on finding a physical schema that maximizes an objective function, e.g., the number of SQL statements executed per second. Due to the vast solution space, however, finding an optimal physical schema *manually* by the database administrator is, in most cases, complicated or even infeasible. Therefore, academia and industry developed tools for *automated physical database design* [131].

1.1 MEMORY FOOTPRINT REDUCTION OF CLOUD DATABASES

Traditionally, enterprises have run database management systems on their own hardware, i.e., *on-premise*. As hardware was typically purchased for a more extended period of several years, the physical schema was mainly optimized towards achieving the best performance on the dedicated hardware [103]. Nowadays, enterprises are increasingly moving their data into the *cloud* by using data management offerings of database-as-a-service providers like Amazon Redshift [28], Snowflake [39], or SAP HANA Cloud [146]. Database-as-a-service providers specialize in hosting and managing database instances, also called tenants. While offering high-performance data management to their customers is essential, database-as-a-service providers also need to consider their role as economic actors. As such, they are incentivized to minimize their internal costs to enhance profitability. In addition, reduced internal costs may result in more differentiated pricing to acquire new customers and increase market share.

One option to lower internal costs consists of utilizing the available hardware resources more efficiently by placing as many database instances as possible on shared hardware resources to increase the *tenant density*. This is possible because database-as-a-service providers host database instances of up to thousands of customers on shared hardware resources, e.g., shared compute, memory, and storage nodes [71]. Therefore, the virtualized hardware of each database instance can be flexibly adapted by database-as-a-service providers.

Previous work [106] identified the provisioned amount of DRAM as the primary driver of hardware costs. To illustrate, the monthly cost of a memory-optimized Google Cloud [60] instance in 2022 amounts to 18 USD per vCPU and 170 USD per TB of provisioned SSD disk space, whereas DRAM capacity is prized at 2610 USD per TB. Hence, database-as-a-service providers are inclined to reduce the provisioned amount of DRAM of each database instance. Reductions in memory footprint thus translate into substantial hardware cost savings.

Although low internal costs crucially affect the profitability of database-as-a-service providers, service-level agreements (SLAs) with customers are defined such that customers can count on an agreed level of performance. For example, the e-commerce platform Alibaba needs to ensure that up to 491,000 sales per second can be processed on Single's Day [76]. As a result, database-as-a-service providers face the challenge of meeting performance commitments while keeping the memory footprint of hosted database instances as small as possible.

1.2 HOT AND COLD DATA

Database workloads often result in a skewed data access pattern [7, 20, 46, 51, 56, 76, 102, 162]. In particular, data accesses can be skewed over

the domain and over time. A domain skew can be observed in an e-commerce web application, where everyday, low-priced items are more frequently purchased than particularly expensive ones. By contrast, orders of seasonal items illustrate a temporal skew as they are only in high demand at specific time periods, e.g., around Christmas. In summary, frequently accessed data are referred to as *hot data*, whereas rarely accessed data are called *cold data*.

An efficient way to decrease the memory footprint of a database instance while still fulfilling performance commitments assured in SLAs is to leverage the workload's access skew by moving cold data to cheaper storage layers and retaining only hot data in DRAM. In order to load and evict data from secondary storage to DRAM and vice versa, most database management systems employ a buffer manager [70]. Since a buffer manager operates at page granularity (i.e., small chunks of data), the database administrator is inclined to keep the disk pages with hot data in DRAM, e.g., the working set [44, 46, 99, 120, 157]. However, the physical schema is often not defined according to the data access pattern. Thus, cold data can appear on the same disk page as hot data. To give an example, a single disk page may store data about an entire product line, i.e., tuples (in a row store) or values (in a column store) of both rarely accessed, expensive products and frequently accessed, low-priced products. Consequently, no substantial memory footprint reduction can be achieved as all hot disk pages (including the respective cold data) must be kept in DRAM. The reason is that reducing the buffer pool size, in this case, would violate performance commitments as disk pages with hot data would have to be evicted to secondary storage.

The buffer manager can be utilized more efficiently by adjusting the physical schema (e.g., by data reorganization) such that hot disk pages contain mainly hot data. This prevents cold data from polluting the buffer pool, leading to a considerably smaller buffer pool size which fulfills performance commitments. In addition, a smaller buffer pool size can reduce the provisioned amount of DRAM for a hosted database instance. This results in lower internal costs for database-as-a-service providers and thereby enhances their profitability.

1.3 PROBLEM STATEMENT AND CHALLENGES

A skewed data access pattern can often be observed in database workloads. Since the physical schema is usually not defined according to the data access pattern, cold data can appear on the same disk page as hot data. As DRAM constitutes the primary driver of hardware costs, keeping both hot and cold data in the buffer pool wastes expensive DRAM capacities. This offers a largely untapped cost-saving potential to database-as-a-service providers. We point out three challenges that need to be addressed to propose a solution to this problem.

1.3.1 *Accurate Collection of Workload Statistics with Low Overhead*

The first step toward reducing the buffer pool size is identifying hot and cold data at a finer granularity than disk pages. To make this classification, accurate statistics about the database workload are essential. In addition, statistics about the workload should be collected with low overhead to avoid significant implications on currently executed workloads in terms of performance degradation or memory consumption enlargement. Therefore, the first challenge we identify relates to the collection of workload statistics with high precision, low memory consumption, and low runtime overhead.

1.3.2 *Memory Footprint Reduction with Automated Table Partitioning*

After hot and cold data are accurately identified, a physical schema should be proposed by grouping hot data into hot disk pages and cold data into cold disk pages. As we will elaborate on later, a typical workload access pattern can be observed in which rows are either frequently or rarely accessed according to a value range of a specific column. For example, frequently accessed orders may have an order date during the last three days. This observation infers that range partitioning helps prevent the pollution of the buffer pool with cold data. More specifically, rows that correspond to hot-classified value ranges can be grouped into hot range partitions that remain in DRAM. Further, rows that belong to cold-classified value ranges are grouped into cold range partitions that can be evicted to cheaper storage layers. Whereas range partitioning seems promising to separate hot and cold data, hash and round-robin partitioning would do the opposite because hot and cold data are likely distributed evenly across all partitions. Accordingly, the second challenge we identify touches on how to propose a range partitioning layout for each relation in order to minimize the buffer pool size while adhering to SLAs.

1.3.3 *Prediction of the Future Workload*

As workloads are not only skewed over the domain but also over time (i.e., drifting workloads), the proposed physical schema may become outdated and is no longer optimal. This can lead to either a significant increase in memory footprint or a violation of performance commitments. Thus, a straightforward approach is to repeatedly adjust the physical schema based on the observed workload. The proposed physical schema, however, may already be suboptimal when it is proposed as the workload has drifted. Ideally, a physical schema should be proposed which is optimized for the future workload. Hence, the third challenge we identify concerns the prediction of the future workload based on an observed workload.

1.4 CONTRIBUTIONS AND OUTLINE

In this dissertation, we propose the following solution to the problem introduced in Section 1.3. We recommend a range partitioning layout for each relation by grouping tuples into partitions according to hot- or cold-classified value ranges of a specific attribute based on a given workload. This reduces the buffer pool size while still fulfilling performance commitments as the pollution of the buffer pool with cold data is prevented. The proposed range partitioning layout is either based on collected workload statistics of an observed workload or a prediction of the future workload in light of drifting workloads.

We address each of the three introduced challenges with the following contributions made in this dissertation:

- A *statistics collector* that gathers accurate statistics about an executed workload with low memory consumption and low runtime overhead (cf. Chapter 3).
- A *table partitioning advisor* that proposes a range partitioning layout based on a given workload in order to minimize the buffer pool size while agreed-upon performance commitments in SLAs are still fulfilled (cf. Chapter 4).
- A *workload predictor* that forecasts the approximate future workload based on the observed workload (cf. Chapter 5).

Before we elaborate on the contributions in this dissertation, we sketch in Figure 1.1 two use cases that benefit these contributions.

In the first use case, we assume that the workload is only skewed over the domain, e.g., an *analytical workload* with long-running and static queries. Under this assumption, statistics about the workload, executed on the current physical schema, are collected once. The statistics are then used as input to the table partitioning advisor, which proposes a future physical schema to minimize the buffer pool size while adhering to performance commitments.

We assume that the workload is skewed over the domain and over time for the second use case. This assumption is typical for *transactional workloads*, e.g., the order fulfillment process in an e-commerce web application. Therefore, we predict the future workload based on the observed workload, which is then used as input to the table partitioning advisor in this use case. In addition to the predicted workload, the table partitioning advisor also considers the current physical schema because a change in the physical schema shall only be triggered when the expected benefits of workload cost reduction outweigh the cost of changing the physical schema, e.g., due to table repartitioning costs. This is crucial as the physical schema may be adjusted regularly to react to workload drifts. Furthermore, the workload prediction and table partitioning advice phase are repeated periodically to adopt the physical schema in small and cheap adjustments.

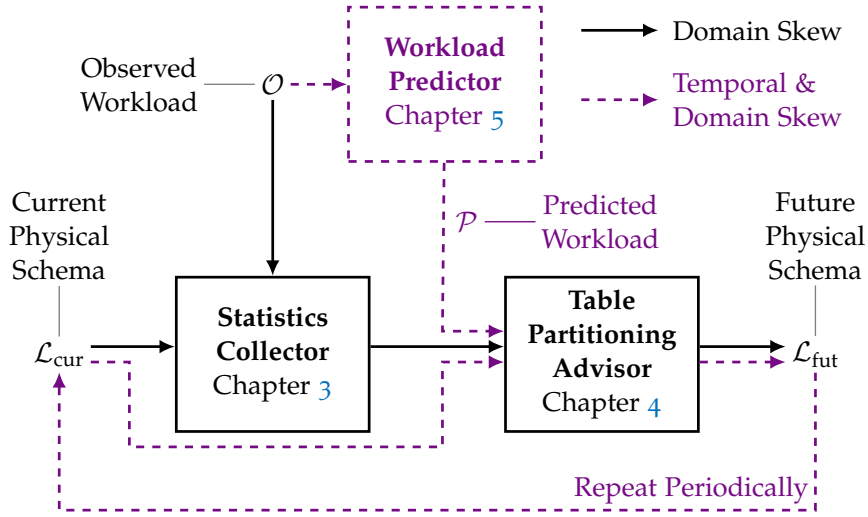


Figure 1.1: Illustration of how the three contributions made in this dissertation address the problem of buffer pool pollution with cold data. In the proposed solution, a table partitioning advisor recommends a range partitioning for each table in which hot and cold data are separated into partitions. The proposed range partitioning is based on collected workload statistics or a predicted workload.

In Chapter 3, we present the statistics collector as our first contribution. We provide a formal description of which workload statistics need to be collected for four physical database design advisors: an index advisor, a data compression advisor, a buffer pool size advisor, and a table partitioning advisor. Subsequently, we introduce low-level data access counters, which collect statistics about an executed workload with high precision, low memory consumption, and low runtime overhead. We then experimentally evaluate our low-level data access counters and demonstrate that they outperform related approaches. Parts of Chapter 3 were previously published in Brendle et al. [24].

The table partitioning advisor presented in Chapter 4 constitutes the second contribution. We provide a formal problem description of minimizing the buffer pool size while fulfilling performance commitments assured in SLAs for range partitioning layouts. Furthermore, we explain how we determine a range partitioning layout based on the collected statistics in Chapter 3. In particular, we propose an optimal range partitioning layout and a heuristic approach to lower optimization time. The optimal approach is based on an estimator for data accesses and storage sizes as well as a cost model that considers both memory footprint and performance, whereas the heuristic approach only utilizes the collected statistics. We also demonstrate that our advisor substantially reduces the buffer pool size while still fulfilling performance commitments compared to related approaches. In Brendle et al. [25], we published parts of Chapter 4.

The third and final contribution of this dissertation is a workload predictor, which is introduced in Chapter 5. We present an approach

that predicts the future statement arrival rate and the future parameterization of SQL statements in two stages. First, we introduce detectors for common workload drift types in both stages and present a classifier to resolve conflicts between multiple detected workload drift types. Second, we describe a predictor for each classified workload drift type that extrapolates the statement arrival rate and parameterization of SQL statements into the future. Third, we present an algorithm that combines the results of both stages to predict the future workload. Finally, we demonstrate how accurately our workload predictor approximates the future workload.

We continue this dissertation in Chapter 2 by providing background information and conclude the dissertation in Chapter 6.

2

BACKGROUND

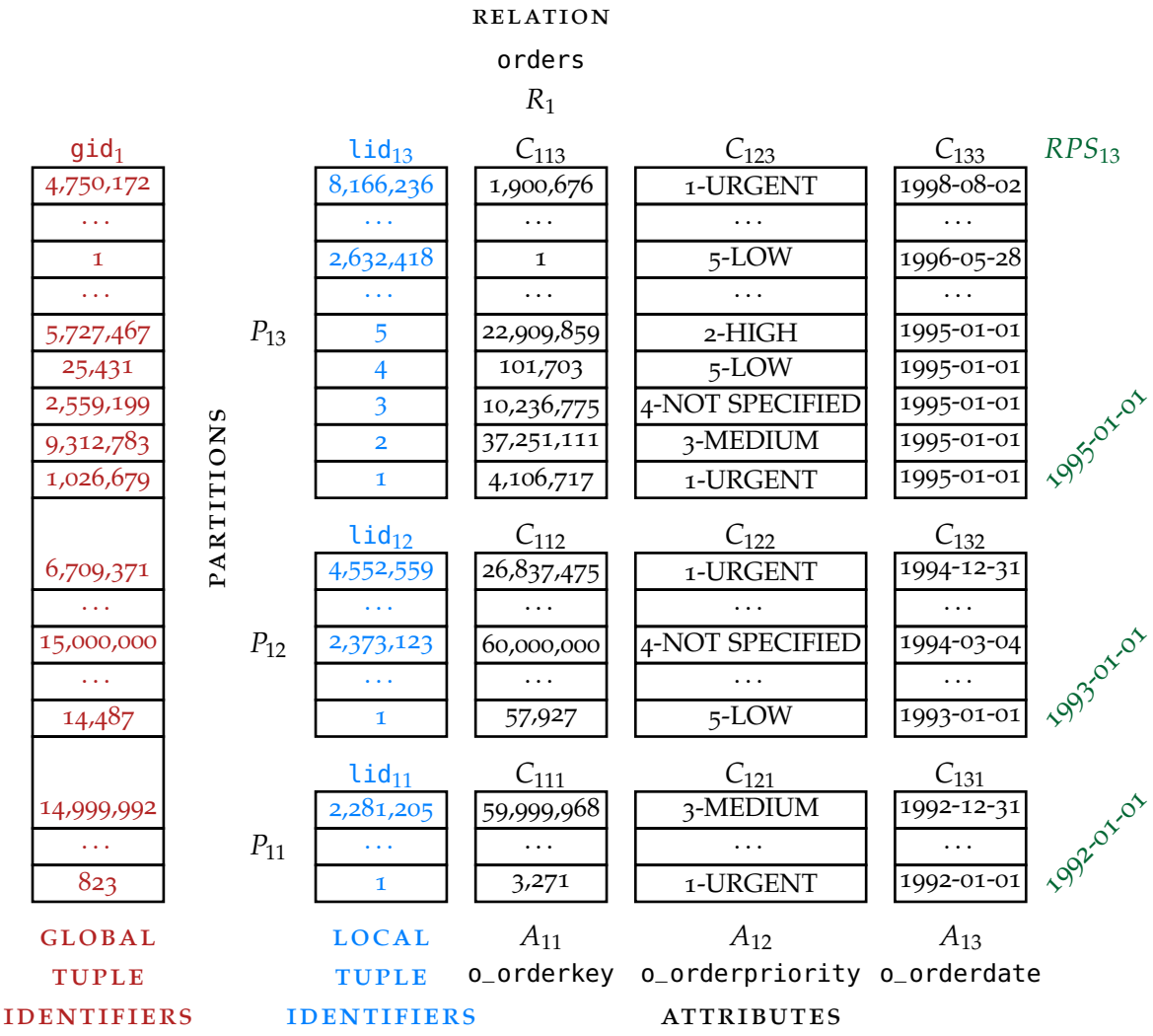
In this chapter, we provide the background information required to understand the following chapters. Throughout this dissertation, we focus on reducing the memory footprint of database management systems by changing their physical schema while still adhering to performance commitments. Therefore, we begin by introducing notation on physical database design and the workload of a database (cf. Section 2.1). As finding an optimal physical schema that meets all requirements (e.g., with respect to memory footprint and performance) is usually a complex and challenging task, academia and industry devised tools for automated physical database design [26, 38, 73, 80, 89, 118, 119, 125, 133, 138, 149, 173, 174]. Hence, we introduce four use cases of automated physical database design: an index advisor, a data compression advisor, a buffer pool size advisor, and a table partitioning advisor (cf. Section 2.2). Afterwards, we describe the architecture of the commercial database management system SAP HANA [54, 97, 109, 153] because we implement and evaluate the effectiveness of our ideas on a prototype of SAP HANA (cf. Section 2.3). Finally, we introduce the five-minute rule published in 1987 by Gray and Putzolu (cf. Section 2.4). We use the five-minute rule to draw a reasonable border between hot and cold data to move cold data to cheaper storage layers.

2.1 NOTATION

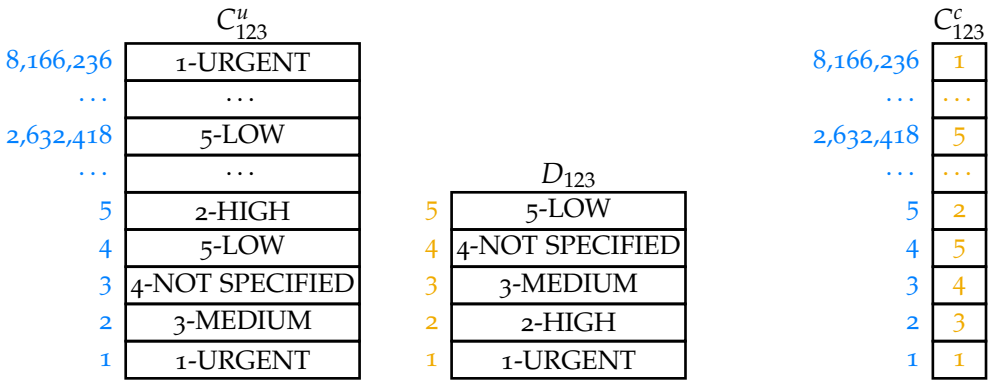
We now introduce notation on three popular aspects of physical database design such as indexing (cf. Section 2.1.1), table partitioning (cf. Section 2.1.2), and data compression (cf. Section 2.1.3). Subsequently, we define the physical schema of the database (cf. Section 2.1.4). We then introduce the notation of a database workload since the effectiveness of a physical schema strongly depends on the executed workload (cf. Section 2.1.5). As an example, performance can be improved if an index is created on a set of attributes that are frequently referenced in selective predicates.

All introduced notation is summarized in Table 2.1. In addition, Figure 2.1 is a running example throughout this chapter, which illustrates a physical schema of the `orders` relation of the JCC-H benchmark [22].

Definition 1 (Relations and Attributes)**.** *We denote by* $\mathcal{R} = \{R_1, \ldots, R_i, \ldots, R_n\}$ *a set of* $n \in \mathbb{N}$ *relations, and by* $\mathcal{A}(R_i) = \{A_{i1}, \ldots, A_{ij}, \ldots, A_{im_i}\}$ *a set of* $m_i \in \mathbb{N}$ *attributes of relation* $R_i \in \mathcal{R}$*. In addition, we denote by* $\text{dom}(A_{ij}) = \Pi^D_{A_{ij}}(R_i)$ *the active domain of attribute* A_{ij}*, and by* $d_{ij} = |\text{dom}(A_{ij})|$ *the number of distinct values of* A_{ij}*.*



(a) A table partitioning layout $\mathcal{T}_1$ for `orders` (R_1), generated by the range partitioning specification RPS_{13}.



(b) Uncompressed column partition C^u_{123} (c) Dictionary D_{123} (d) Compressed column partition C^c_{123}

Figure 2.1: Illustration of a physical schema for relation `orders` (R_1) with three attributes `o_orderkey` (A_{11}), `o_orderpriority` (A_{12}), and `o_orderdate` (A_{13}). The table partitioning layout $\mathcal{T}_1$ is depicted in (a), an example of an uncompressed column partition C^u_{123} in (b), a dictionary D_{123} in (c), and a dictionary-compressed column partition C^c_{123} in (d).

Note that $\Pi^D_{A_{ij}}(R_i)$ is the duplicate elimination projection for values from attribute A_{ij} from tuples in relation R_i. In Figure 2.1, we illustrate `orders` as relation R_1, and `o_orderkey` (A_{11}), `o_orderpriority` (A_{12}), and `o_orderdate` (A_{13}) as three attributes of R_1. The active domain of A_{12} is {1-URGENT, 2-HIGH, 3-MEDIUM, 4-NOT SPECIFIED, 5-LOW}, consisting of 5 distinct values, denoted as d_{12}.

2.1.1 *Index Configuration*

An index is a data structure to speed up data retrieval. In a database management system, a single- or multi-column index can be defined over a set of attributes to improve the performance of selections, joins, or uniqueness checks [4, 26, 80, 89, 118, 125, 174]. In the following, we start to define a single- or multi-column index before specifying an index configuration for a relation.

Definition 2 (Index). *Let $\mathbb{A}_{is} \in \mathcal{P}(\mathcal{A}(R_i))$ be a set of attributes from the power set of all attributes of relation $R_i \in \mathcal{R}$ that is uniquely identified by $s \in [1, |\mathcal{P}(\mathcal{A}(R_i))|]$. We define by $\mathbb{I}_{is}$ a single- or multi-column index over the set of attributes $\mathbb{A}_{is}$.*

For relation `orders` (R_1) in Figure 2.1, for instance, indexes over eight different sets of attributes are possible: $\mathbb{A}_{11} = \varnothing$ (i.e., no index), $\mathbb{A}_{12} = \{A_{11}\}$, $\mathbb{A}_{13} = \{A_{12}\}$, $\mathbb{A}_{14} = \{A_{13}\}$, $\mathbb{A}_{15} = \{A_{11}, A_{12}\}$, $\mathbb{A}_{16} = \{A_{11}, A_{13}\}$, $\mathbb{A}_{17} = \{A_{12}, A_{13}\}$, and $\mathbb{A}_{18} = \{A_{11}, A_{12}, A_{13}\}$. The index $\mathbb{I}_{17}$ is then a multi-column index over the attributes `o_orderpriority` (A_{12}) and `o_orderdate` (A_{13}).

Definition 3 (Index Configuration). *Let $\mathcal{I}_i = \{\mathbb{I}_{is} \mid 1 \leq s \leq |\mathcal{P}(\mathcal{A}(R_i))|\}$ be the set of all possible indexes of relation R_i. We define by $\mathcal{IC}_i \subseteq \mathcal{I}_i$ an index configuration for relation R_i as a subset of $\mathcal{I}_i$.*

To illustrate, for relation `orders` (R_1) in Figure 2.1, an index configuration $\mathcal{IC}_1 = \{\mathbb{I}_{12}, \mathbb{I}_{17}\}$ would create a single-column index over `o_orderkey` (A_{11}) and a multi-column index over `o_orderpriority` (A_{12}) and `o_orderdate` (A_{13}).

2.1.2 *Table Partitioning*

Table partitioning is a technique to divide the data of a relation into disjoint units. This can be done horizontally by dividing the tuples or vertically by dividing the attributes into partitions [103]. A typical use case of table partitioning is a scale-out system, where data are partitioned and distributed across several server nodes (e.g., by hash or round-robin partitioning) to balance the workload across the system [38, 73, 119, 133, 138, 149, 173]. In this work, we focus on horizontal range partitioning, where each partition contains data that belong to a specific value range of a certain attribute of that relation. To

give an example, a range partition contains all tuples of `orders` with an `o_orderdate` in 1992. In addition, we focus on column stores because we implemented our methods into a prototype of SAP HANA's column store (cf. Section 2.3). Note that column stores are already vertically partitioned by definition.

Definition 4 (Partition-Driving and Passive Attributes). *We define $A_{i\lambda} \in \mathcal{A}(R_i)$ as the partition-driving attribute of relation $R_i \in \mathcal{R}$, where $1 \leq \lambda \leq m_i$. Any other attribute $A_{ij} \neq A_{i\lambda}$ is called a passive attribute.*

Definition 5 (Range Partitioning Specification). *We define a range partitioning specification $RPS_{i\lambda} = \{v_{i1}, \ldots, v_{ik}, \ldots, v_{ip_{i\lambda}}\} \subseteq \text{dom}(A_{i\lambda})$ with $v_{i1} < \ldots < v_{ik} < \ldots < v_{ip_{i\lambda}}$ and $v_{i1} = \min(\text{dom}(A_{i\lambda}))$ as a subset of the active domain of the partition-driving attribute $A_{i\lambda}$. Further, we denote by $\mathbb{RPS}_{i\lambda}$ the set of all range partitioning specifications for $A_{i\lambda}$.*

The right side of Figure 2.1 illustrates the range partitioning specification $RPS_{13} = \{1992\text{-}01\text{-}01, 1993\text{-}01\text{-}01, 1995\text{-}01\text{-}01\}$ for the partition-driving attribute `o_orderdate` (A_{13}). The attributes `o_orderkey` (A_{11}) and `o_orderpriority` (A_{12}) are passive attributes.

Definition 6 (Partitioning). *We define a partitioning $\mathbb{P}(RPS_{i\lambda}) = \{P_{i1}, \ldots, P_{ik}, \ldots, P_{ip_{i\lambda}}\}$ as a set of $p_{i\lambda} \in \mathbb{N}$ partitions, generated from a range partitioning specification $RPS_{i\lambda}$ by*

$$P_{ik} := \begin{cases} \sigma_{v_{ik} \leq A_{i\lambda} < v_{i(k+1)}}(R_i) & (k < p_{i\lambda}) \\ \sigma_{v_{ip_{i\lambda}} \leq A_{i\lambda}}(R_i) & (k = p_{i\lambda}) \end{cases}, \quad \text{for all } 1 \leq k \leq p_{i\lambda}.$$

Note that σ is the selection operator, such that a partition P_{ik} is generated by selecting only tuples of relation R_i, where the value of the partition-driving attribute $A_{i\lambda}$ is between two partition boundaries v_{ik} and $v_{i(k+1)}$ of the range partitioning specification $RPS_{i\lambda}$. For example, Figure 2.1 shows the partitioning $\mathbb{P}(RPS_{13}) = \{P_{11}, P_{12}, P_{13}\}$ generated from the range partitioning specification RPS_{13}. Partition P_{11} contains tuples with an `o_orderdate` (A_{13}) between 1992-01-01 and 1993-01-01.

As we focus on column stores, we next define a table partitioning layout as a set of all column partitions.

Definition 7 (Column Partition). *We denote by $C_{ijk} = \Pi_{A_{ij}}(P_{ik})$ a column partition as a projection of attribute A_{ij} in partition P_{ik}.*

In Section 2.1.3, we define two physical representations of a column partition C_{ijk}, either uncompressed or dictionary-compressed.

Definition 8 (Table Partitioning Layout). *We define by*

$$\mathcal{T}_i := \{C_{ijk} \mid 1 \leq j \leq m_i, 1 \leq k \leq p_{i\lambda}\}$$

a table partitioning layout of relation $R_i \in \mathcal{R}$ as a set of $m_i \cdot p_{i\lambda}$ column partitions C_{ijk}.

Figure 2.1 shows the table partitioning layout $\mathcal{T}_1$ for `orders` (R_1), consisting of nine column partitions $C_{111}, \ldots, C_{133}$, generated by the range partitioning specification RPS_{13}.

Definition 9 (Global and Local Tuple Identifiers)**.** *We associate with every tuple in relation R_i a unique global tuple identifier $\texttt{gid}_i \in [1, |R_i|]$, and with every tuple in a partition P_{ik} a unique local tuple identifier $\texttt{lid}_{ik} \in [1, |P_{ik}|]$. Given a partition P_{ik} and a local tuple identifier $\texttt{lid}_{ik}$, the global tuple identifier is retrieved by $P_{ik}[\texttt{lid}_{ik}].\texttt{get\_gid}$.*

We associate conceptual global and local tuple identifiers to identify the same tuple of different partitioning layouts. Note that they do not necessarily need to be stored in a database. The left side of Figure 2.1 shows for each tuple of `orders` (R_1) its global tuple identifier $\texttt{gid}_1 \in [1, 15000000]$. Further, for each tuple of the three partitions P_{11}, P_{12}, and P_{13}, the corresponding local tuple identifier $\texttt{lid}_{1k}$ is depicted. For example, tuples in partition P_{11} have a $\texttt{lid}_{11} \in [1, 2281205]$.

2.1.3 *Data Compression*

Data compression is a technique of encoding data using fewer bits than the original, uncompressed representation. In a database management system, compression of data stored in relations, attributes, partitions, or column partitions can reduce the memory footprint. A popular compression technique, particularly in column stores, is *dictionary compression* [1, 2, 18, 91, 93, 94, 136, 139]. Therefore, we present definitions for dictionary and dictionary-compressed column partitions. In order to compare it with the original representation, we start with the definition of an uncompressed column partition.

Definition 10 (Uncompressed Column Partition)**.** *We define an uncompressed column partition C^u_{ijk} of attribute A_{ij} in partition P_{ik} as a vector of length $|P_{ik}|$ with*

$$C^u_{ijk}[\texttt{lid}_{ik}] = P_{ik}[\texttt{lid}_{ik}].A_{ij}, \qquad \text{for all } 1 \leq \texttt{lid}_{ik} \leq |P_{ik}|,$$

where $P_{ik}[\texttt{lid}_{ik}].A_{ij}$ retrieves the value of attribute A_{ij} in partition P_{ik} for the tuple with the local tuple identifier $\texttt{lid}_{ik}$.

In Figure 2.1b, we illustrate an uncompressed column partition C^u_{123}. It stores all values of attribute `o_orderpriority` (A_{12}) in partition P_{13} in a vector of length 8,166,236. The local tuple identifiers $\texttt{lid}_{13}$ determine the placement of the values inside the vector.

Definition 11 (Dictionary)**.** *Let* $\text{dom}(A_{ij}, P_{ik}) = \Pi^D_{A_{ij}}(P_{ik}) = \{v_{ijk1}, \ldots, v_{ijky}, \ldots, v_{ijkd_{ijk}}\}$ *with* $v_{ijk1} < \ldots < v_{ijky} < \ldots < v_{ijkd_{ijk}}$ *denote the active domain of attribute A_{ij} in partition P_{ik}, where $d_{ijk} = |\text{dom}(A_{ij}, P_{ik})|$ is the number of distinct values of A_{ij} in P_{ik}. We define a dictionary of attribute A_{ij} in partition P_{ik} as a bijection $D_{ijk} = (\texttt{vid}_{ijk} : \text{dom}(A_{ij}, P_{ik}) \rightarrow [1, d_{ijk}])$ with $\texttt{vid}_{ijk}(v_{ijky}) = y$.*

The dictionary D_{ijk} of attribute A_{ij} in partition P_{ik} is a bijection $\texttt{vid}_{ijk}$, where $\text{dom}(A_{ij}, P_{ik})$ is the domain and $[1, d_{ijk}]$ is the image of the function, such that the y-th value of the domain returns number y. As an example, Figure 2.1c shows the dictionary D_{123} of attribute `o_orderpriority` (A_{12}) in partition P_{13} with the five distinct values 1-URGENT, 2-HIGH, 3-MEDIUM, 4-NOT SPECIFIED, and 5-LOW.

Definition 12 (Dictionary-Compressed Column Partition)**.** *We define a dictionary-compressed column partition C^c_{ijk} of attribute A_{ij} in partition P_{ik} as a vector of numbers in $[1, d_{ijk}]$, such that*

$$C^c_{ijk}[\texttt{lid}_{ik}] = \texttt{vid}_{ijk}(P_{ik}[\texttt{lid}_{ik}].A_{ij}), \qquad \text{for all } 1 \leq \texttt{lid}_{ik} \leq |P_{ik}|.$$

The dictionary-compressed column partition C^c_{ijk} stores the numbers returned by the bijection $\texttt{vid}_{ijk}$ of the dictionary D_{ijk} for all values of attribute A_{ij} in partition P_{ik}. The placement of the numbers inside the vector C^c_{ijk} is determined by the local tuple identifiers $\texttt{lid}_{ik}$. For example, Figure 2.1d visualizes the compressed column partition C^c_{123} of attribute `o_orderpriority` (A_{12}) in partition P_{13}.

As the major objective of data compression is reducing the data size in bytes, we next define the number of bytes to store an uncompressed column partition, a dictionary, and a dictionary-compressed column partition. In order to understand these definitions, we need to discuss the architecture of a buffer manager, which most database management systems employ as a data cache to load and evict data from secondary storage to main memory and vice versa [70]. As data on secondary storage is partitioned in disk pages, the buffer manager holds a subset of these disk pages in DRAM, in a so-called buffer pool. To illustrate, a disk page may store for a set of local tuple identifiers of a column partition a set of values. In most cases, the buffer manager only holds a strict subset of all disk pages (e.g., the workload's working set [46]) because it may not be economically feasible to hold all data in main memory as DRAM is the primary driver of hardware costs [106]. For this purpose, buffer managers employ a replacement policy (e.g., LRU-K [128]) to decide which disk page must be evicted when a new disk page is requested that is not yet loaded, and no slots are free in the buffer pool. A page request may come from the execution engine that wants to materialize a tuple.

We now define the storage size of an uncompressed column partition, a dictionary, and a dictionary-compressed column partition. For each physical representation, we define a page size in bytes and the number of disk pages that are required to store the underlying data.

Definition 13 (Disk Pages)**.** *We denote by $\texttt{NUMPAGES}^u_{ijk}$ the number of disk pages to store an uncompressed column partition C^u_{ijk}, where each disk page has a page size of $\texttt{PAGESIZE}^u_{ijk}$ bytes. Further, we denote by $\texttt{NUMPAGES}^d_{ijk}$ and $\texttt{NUMPAGES}^c_{ijk}$ the number of disk pages to store a dictionary D_{ijk} and a dictionary-compressed column partition C^c_{ijk} with page sizes (in bytes) of $\texttt{PAGESIZE}^d_{ijk}$ and $\texttt{PAGESIZE}^c_{ijk}$.*

Definition 14 (Storage Size). *We define by* $||C^u_{ijk}||$, $||D_{ijk}||$, *and* $||C^c_{ijk}||$ *the number of bytes to store all disk pages of an uncompressed column partition* C^u_{ijk}, *a dictionary* D_{ijk}, *and a dictionary-compressed column partition* C^c_{ijk}:

$$||C^u_{ijk}|| := \texttt{NUMPAGES}^u_{ijk} \cdot \texttt{PAGESIZE}^u_{ijk}$$
$$||D_{ijk}|| := \texttt{NUMPAGES}^d_{ijk} \cdot \texttt{PAGESIZE}^d_{ijk}$$
$$||C^c_{ijk}|| := \texttt{NUMPAGES}^c_{ijk} \cdot \texttt{PAGESIZE}^c_{ijk}.$$

In order to choose between no compression or dictionary compression, both the memory footprint and the performance could be considered. The choice may depend solely on the effectiveness of dictionary compression when aiming for minimal memory footprint. However, dictionary compression adds one more level of indirection and may degrade performance. During tuple materialization, for instance, both the dictionary-compressed column partition and the dictionary needs to be accessed. Accessing the dictionary incurs, in general, one additional random memory access, e.g., when the dictionary does not fit into the CPU cache. In addition, the performance of `INSERT` and `UPDATE` statements may degrade as sorted dictionaries and dictionary-compressed column partitions cannot be modified easily to apply the data modifications.

In general, the decision of compressing a column partition or not may depend on the objective function and constraints regarding performance and memory footprint. Furthermore, database management systems generally support several compression techniques. For example, SAP HANA supports prefix, run-length, cluster, sparse, and indirect encoding on top of dictionary compression (cf. Section 2.3.2). In order to cover all possibilities of compression, we introduce a data compression indicator function to decide if and how a column partition should be compressed.

Definition 15 (Data Compression Indicator Function). *We denote by* $\mathbb{C} = \{\textit{uncompressed}, \textit{dictionary-compressed}, \dots\}$ *a set of compression techniques to compress a column partition. To identify the compression technique for a column partition* C_{ijk} *of attribute* A_{ij} *in partition* P_{ik}, *we define by* $\texttt{comp}_{i\lambda} : [1, m_i] \times [1, p_{i\lambda}] \to \mathbb{C}$ *a data compression indicator function.*

We now illustrate an example of a data compression indicator function $\texttt{comp}_{i\lambda}$ that optimizes solely on the memory footprint, i.e., does not consider the aspect of performance during projections or updates. For a set of compression techniques $\mathbb{C} = \{\text{uncompressed}, \text{dictionary-compressed}\}$, a data compression indicator function $\texttt{comp}_{i\lambda}$ can then be defined as

$$\texttt{comp}_{i\lambda}(j,k) := \begin{cases} \text{dictionary-compressed} & \text{if } ||D_{ijk}|| + ||C^c_{ijk}|| < ||C^u_{ijk}|| \\ \text{uncompressed} & \text{otherwise.} \end{cases}$$

The physical representation of a column partition C_{ijk} is then chosen depending on the data compression indicator function $\texttt{comp}_{i\lambda}$:

$$C_{ijk} := \begin{cases} (D_{ijk}, C^c_{ijk}) & \text{if } \texttt{comp}_{i\lambda}(j,k) = \text{dictionary-compressed} \\ C^u_{ijk} & \text{otherwise.} \end{cases}$$

The storage size in bytes of a column partition C_{ijk} is defined as

$$||C_{ijk}|| := \begin{cases} ||D_{ijk}|| + ||C^c_{ijk}|| & \text{if } \texttt{comp}_{i\lambda}(j,k) = \text{dict.-compr.} \\ ||C^u_{ijk}|| & \text{otherwise.} \end{cases}$$

Finally, the storage size in bytes of a table partitioning layout $\mathcal{T}_i$ is defined as the sum of the storage size of all column partitions C_{ijk}:

$$||\mathcal{T}_i|| := \sum_{j}^{m_i} \sum_{k}^{p_{i\lambda}} ||C_{ijk}||.$$

2.1.4 *Physical Schema of a Database*

For each relation, we can define an index configuration, a table partitioning layout, and a data compression indicator function. To define the physical schema of a database, we combine all three aspects of physical database design for each relation.

Definition 16 (Physical Schema of a Database)**.** *We define the physical schema of a database by*

$$\mathcal{L} = \{(\mathcal{IC}_1, \mathcal{T}_1, \texttt{comp}_{1\lambda}), \ldots, (\mathcal{IC}_i, \mathcal{T}_i, \texttt{comp}_{i\lambda}), \ldots, (\mathcal{IC}_n, \mathcal{T}_n, \texttt{comp}_{n\lambda}\},$$

where each triple $(\mathcal{IC}_i, \mathcal{T}_i, \texttt{comp}_{i\lambda}) \in \mathcal{L}$ *consists of an index configuration* $\mathcal{IC}_i$*, a table partitioning layout* $\mathcal{T}_i$*, and a data compression indicator function* $\texttt{comp}_{i\lambda}$ *for relation* $R_i \in \mathcal{R}$*.*

2.1.5 *Workload*

As the effectiveness of indexing, table partitioning, and data compression strongly depends on the database workload, we now introduce its notation. We begin by defining a set of parameterized SQL statements and the host variables of a single statement. Next, we specify a statement instantiation, which assigns parameter values to the host variables of the statement. Afterwards, we define the physical execution plan of a statement instantiation. Finally, a workload is defined as a set of statement instantiations.

Definition 17 (SQL Statements and Host Variables)**.** *We define by* $\mathcal{S} = \{S_1, \ldots, S_q, \ldots, S_u\}$ *a set of* $u \in \mathbb{N}$ *parameterized SQL statements (e.g.,* `SELECT`*,* `INSERT`*,* `UPDATE`*, and* `DELETE` *statements). Each statement* $S_q \in \mathcal{S}$ *contains a vector* $H_q = [h_{q1}, \ldots, h_{qr}, \ldots, h_{qw_q}]$ *of* $w_q \in \mathbb{N}$ *host variables.*

Listing 2.1: Example of the Shipping Priority Query (S_3) from the set of 22 SQL statements of the JCC-H benchmark, consisting of the two host variables `:1` (h_{31}) and `:2` (h_{32}) [22].

```
SELECT   top 10
         l_orderkey,
         sum(l_extendedprice * (1 - l_discount)) as revenue,
         o_orderdate,
         o_shippriority
FROM     customer,
         orders,
         lineitem
WHERE    c_mktsegment = :1
         and c_custkey = o_custkey
         and l_orderkey = o_orderkey
         and o_orderdate < to_date(:2)
         and l_shipdate > to_date(:2)
GROUP BY l_orderkey,
         o_orderdate,
         o_shippriority
ORDER BY revenue desc,
         o_orderdate
```

For example, the JCC-H benchmark consists of a set of 22 SQL statements [22]. Note that the JCC-H benchmark extends the TPC-H benchmark [159] with data and parameter skew. In Listing 2.1, we illustrate statement S_3 of the JCC-H benchmark, which retrieves the 10 unshipped orders with the highest revenue. We observe that the statement contains two host variables `:1` (h_{31}) and `:2` (h_{32}).

Definition 18 (Statement Instantiation and Assignment)**.** *We define the triple* (t, S_q, V_q) *as an instantiation of a statement* $S_q \in \mathcal{S}$ *with a vector* V_q *of* w_q *parameter values at timestamp* $t \in \mathbb{N}$*. A statement instantiation* (t, S_q, V_q) *contains* w_q *assignments, such that parameter value* $V_q[r] \in \text{dom}(h_{qr})$ *is assigned to host variable* h_{qr} *at timestamp* t*, where* $\text{dom}(h_{qr})$ *denotes the domain of the host variable* h_{qr}*.*

To illustrate, statement S_3 of the JCC-H benchmark in Listing 2.1 can be instantiated with the vector $V_3 = [\text{FURNITURE}, 1993\text{-}05\text{-}29]$ at the (`UNIX`) timestamp 1643900400 (i.e., 2022-02-03 16:00:00 GMT). Thus, host variable `:1` (h_{31}) is assigned by the parameter value FURNITURE and host variable `:2` (h_{32}) by the parameter value 1993-05-29.

Definition 19 (Physical Execution Plan)**.** *We define* $T(t, S_q, V_q)$ *as the physical execution plan of a statement instantiation* $(t, S_q, V_q) \in \mathcal{W}$*.*

Figure 2.2 shows the physical execution plan produced by SAP HANA's query optimizer [54, 97, 109, 153] for the instantiation of statement S_3 of the JCC-H benchmark (cf. Listing 2.1) with the vec-

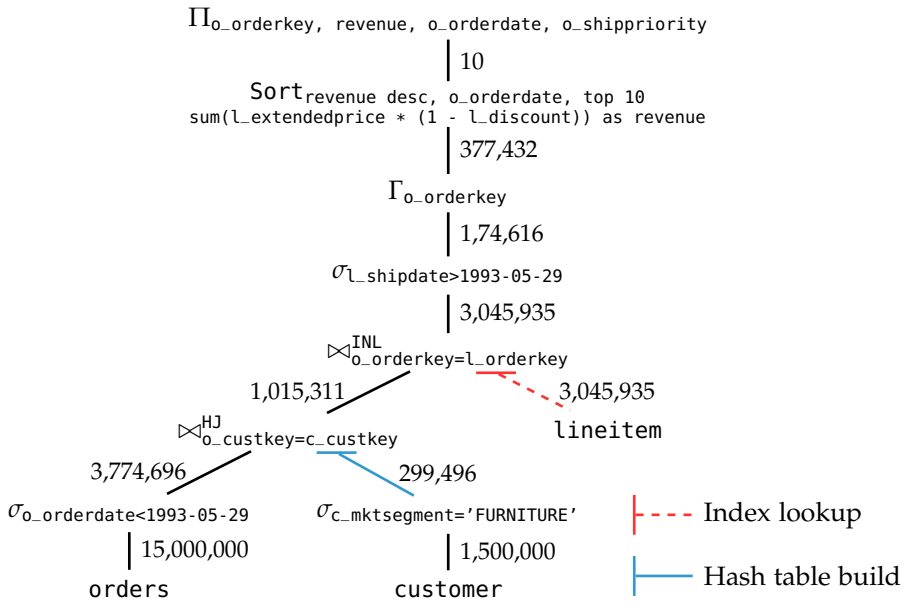


Figure 2.2: Illustration of the physical execution plan produced by SAP HANA's query optimizer [54, 97, 109, 153] for the instantiation of statement S_3 of the JCC-H benchmark with scale factor 10 (cf. Listing 2.1) at timestamp 1643900400 with the vector $V_3 = [\text{FURNITURE}, \text{1993-05-29}]$ of two parameter values.

tor $V_3 = [\text{FURNITURE}, \text{1993-05-29}]$ at timestamp 1643900400, using scale factor 10 for generating the data of JCC-H. In the physical execution plan, every node represents an operator and every edge an intermediate result with the actual output cardinality of the previously executed operator. We observe that the selections on `o_orderdate` and `c_mktsegment` are pushed down to their base relations `orders` and `customer`. For example, the output cardinality of the selection $\sigma_{\texttt{o\_orderdate<1993-05-29}}(\texttt{orders})$ is $3,774,696$. As `orders` contains $15,000,000$ tuples, the selectivity of this selection is 0.25. The join ordering shows that `orders` and `customer` are joined before `lineitem` and `orders`. A hash join (HJ) is used between `orders` and `customer` on `o_custkey=c_custkey`. The hash table is build on `c_custkey` of `customer`. In contrast, an index nested loop join (INL) is executed between `lineitem` and `orders` on `o_orderkey=l_orderkey`. Note that the index over `l_orderkey` of `lineitem` is used for the lookup. Moreover, we observe that the selection on `l_shipdate` is executed after the join between `lineitem` and `orders` and thus is not pushed down to the base relation `lineitem`. Finally, the group by, sorting, and projection operator are executed after all selections and joins.

Definition 20 (Workload). *We define a workload of a database*

$$\mathcal{W} = \{(t_1, S_{q_1}, V_{q_1}), \ldots, (t_z, S_{q_z}, V_{q_z})\}$$

as a set of $z \in \mathbb{N}$ *statement instantiations.*

$\mathcal{R} = \{R_1, \ldots, R_i, \ldots, R_n\}$	A set of n relations ($1 \leq i \leq n$).
$\mathcal{A}(R_i) = \{A_{i1}, \ldots, A_{ij}, \ldots, A_{im_i}\}$	A set of m_i attributes of R_i ($1 \leq j \leq m_i$).
$\text{dom}(A_{ij}) = \Pi^D_{A_{ij}}(R_i)$	The active domain of A_{ij}.
$d_{ij} = \lvert\,\text{dom}(A_{ij})\rvert$	The number of distinct values of A_{ij}.
$\mathbb{A}_{is} \in \mathcal{P}(\mathcal{A}(R_i))$	A set of attributes of R_i that is uniquely identified by $s \in [1, \lvert\mathcal{P}(\mathcal{A}(R_i))\rvert]$.
$\mathbb{I}_{is}$	A single- or multi-column index over $\mathbb{A}_{is}$.
$\mathcal{I}_i = \{\mathbb{I}_{is} \mid 1 \leq s \leq \lvert\mathcal{P}(\mathcal{A}(R_i))\rvert\}$	The set of all possible indexes of relation R_i.
$\mathcal{IC}_i \subseteq \mathcal{I}_i$	An index configuration for relation R_i.
$A_{i\lambda} \in \mathcal{A}(R_i)$	The partition-driving attribute of R_i ($1 \leq \lambda \leq m_i$).
$A_{ij} \neq A_{i\lambda}$	A passive attribute of R_i.
$RPS_{i\lambda} = \{v_{i1}, \ldots, v_{ik}, \ldots, v_{ip_{i\lambda}}\} \subseteq \text{dom}(A_{i\lambda})$	A range partitioning specification for $A_{i\lambda}$.
$\mathbb{RPS}_{i\lambda}$	The set of all range partitioning specifications for a partition-driving attribute $A_{i\lambda}$.
$\mathbb{P}(RPS_{i\lambda}) = \{P_{i1}, \ldots, P_{ik}, \ldots, P_{ip_{i\lambda}}\}$	A set of $p_{i\lambda}$ partitions of R_i with $RPS_{i\lambda}$.
C_{ijk}	A column partition of attribute A_{ij} in partition P_{ik}.
$\mathcal{T}_i = \{C_{ijk} \mid 1 \leq j \leq m_i, 1 \leq k \leq p_{i\lambda}\}$	A table partitioning layout as a set of $m_i \cdot p_{i\lambda}$ column partitions of R_i.
$\texttt{gid}_i \in [1, \lvert R_i\rvert]$	A unique global tuple identifier in R_i.
$\texttt{lid}_{ik} \in [1, \lvert P_{ik}\rvert]$	A unique local tuple identifier in P_{ik}.
$C^u_{ijk} = [P_{ik}[1].A_{ij}, \ldots, P_{ik}[\lvert P_{ik}\rvert].A_{ij}]$	An uncompressed column partition as a vector of length $\lvert P_{ik}\rvert$ with values of A_{ij} in P_{ik}.
$\text{dom}(A_{ij}, P_{ik}) = \{v_{ijk1}, \ldots, v_{ijkd_{ijk}}\}$	The active domain of A_{ij} in P_{ik}.
$d_{ijk} = \lvert\,\text{dom}(A_{ij}, P_{ik})\rvert$	The number of distinct values of A_{ij} in P_{ik}.
$D_{ijk} = (\texttt{vid}_{ijk} : \text{dom}(A_{ij}, P_{ik}) \rightarrow [1, d_{ijk}])$	The dictionary for A_{ij} in P_{ik} as a bijection between $\text{dom}(A_{ij}, P_{ik})$ and numbers in $[1, d_{ijk}]$.
C^c_{ijk}=[$\texttt{vid}_{ijk}(P_{ik}[1].A_{ij})$, ..., $\texttt{vid}_{ijk}(P_{ik}[\lvert P_{ik}\rvert].A_{ij})$]	A compressed column partition as a vector of length $\lvert P_{ik}\rvert$ with numbers in $[1, d_{ijk}]$.
$\lvert\lvert C^u_{ijk}\rvert\rvert, \lvert\lvert D_{ijk}\rvert\rvert, \lvert\lvert C^c_{ijk}\rvert\rvert, \lvert\lvert C_{ijk}\rvert\rvert, \lvert\lvert \mathcal{T}_i\rvert\rvert$	The number of bytes to store C^u_{ijk}, D_{ijk}, C^c_{ijk}, C_{ijk}, $\mathcal{T}_i$.
$\mathbb{C}$={uncompressed,dictionary-compressed,...}	A set of compression techniques.
$\texttt{comp}_{i\lambda} : [1, m_i] \times [1, p_{i\lambda}] \rightarrow \mathbb{C}$	A data compression indicator function.
$\mathcal{L} = \{(\mathcal{IC}_1, \mathcal{T}_1, \texttt{comp}_{1\lambda}, \ldots, (\mathcal{IC}_n, \mathcal{T}_n, \texttt{comp}_{n\lambda})\}$	The physical schema of a database.
$\mathcal{S} = \{S_1, \ldots, S_q, \ldots, S_u\}$	A set of u parameterized SQL statements.
$H_q = [h_{q1}, \ldots, h_{qr} \ldots, h_{qw_q}]$	A vector of w_q host variables of S_q.
V_q	A vector of w_q parameter values.
(t, S_q, V_q), $V_q[r]$ is assigned to h_{qr}	A statement instantiation of S_q with vector V_q of parameter values at timestamp t.
$T(t, S_q, V_q)$	The physical execution plan of an instantiation.
$\mathcal{W} = \{(t_1, S_{q_1}, V_{q_1}), \ldots, (t_z, S_{q_z}, V_{q_z})\}$	A workload as a set of z statement instantiations.

Table 2.1: List of notations for automated physical database design and the database workload.

2.2 AUTOMATED PHYSICAL DATABASE DESIGN

Finding an optimal physical schema with respect to memory footprint or performance is often done automated by physical database design advisors since finding an optimal physical schema manually by database experts is expensive or even infeasible. We argue that automated physical database design tools can be categorized according to their objective function, aiming for minimum memory footprint or maximum throughput. Besides that, advisor tools need to fulfill given constraints, e.g., performance commitments agreed upon in SLAs or a memory budget. In this dissertation, we focus on four use cases of automated physical database design advice. We introduce an index advisor, which focuses on maximum throughput by speeding up query response times of a given workload with a memory budget. Afterwards, we present three tools that minimize memory footprint while fulfilling performance commitments: a data compression advisor, a buffer pool size advisor, and a table partitioning advisor.

2.2.1 *Index Advisor*

Creating a single- or multi-column index over a set of attributes can improve the performance of the database. For example, if the workload includes selective filter predicates, traversing the index is then faster than performing a full-column scan. While performance is the objective function of almost all index advisors, a memory budget is typically assumed as a constraint to create indexes only over those attributes where they yield the largest benefit [4, 26, 80, 89, 118, 125, 174].

Use Case 1 (Index Advisor)**.** *Let $\mathcal{BS} \in \mathbb{N}$ be the buffer pool size in bytes. An index advisor proposes an index configuration $\mathcal{IC}_i \subseteq \mathcal{I}_i$ for each relation R_i such that the execution time $\mathcal{E}$ of a workload $\mathcal{W}$ executed on the physical schema $\mathcal{L}$ with the buffer pool size $\mathcal{BS}$ is minimized while the additional memory consumption $\mathcal{M}$ for all index configurations $\mathcal{IC}_i$ adheres to a given memory budget MB:*

$$\underset{R_i \in \mathcal{R}, \mathcal{IC}_i \subseteq \mathcal{I}_i}{\arg\min} \; \mathcal{E}(\mathcal{W}, \mathcal{L}, \mathcal{BS}) \qquad \text{subject to} \left(\sum_{i=1}^{n} \mathcal{M}(\mathcal{IC}_i) \right) \leq MB.$$

2.2.2 *Data Compression Advisor*

The memory footprint of a database management system can be reduced by compressing its data [1, 2, 18, 19, 40, 91, 93, 94, 100, 136, 139]. However, this is often done at the cost of degrading performance as one or more levels of indirection are introduced. To give an example: during a projection, accessing the dictionary in addition to the dictionary-compressed column partition incurs, in general, an additional random memory access compared to accessing only an

uncompressed column partition. While memory footprint is typically the objective function of a data compression advisor, an SLA may be assumed to ensure robust performance of the database. A column partition, for instance, should only be compressed if the latency of critical SQL statement instantiations does not decline significantly.

Use Case 2 (Data Compression Advisor)**.** *Let $\mathcal{BS} \in \mathbb{N}$ be the buffer pool size in bytes, and $\mathcal{W}_{crit} \subseteq \mathcal{W}$ be a subset of critical SQL statement instantiations in the workload $\mathcal{W}$. A data compression advisor proposes a data compression indicator function $\texttt{comp}_{i\lambda}$ for each relation $R_i \in \mathcal{R}$, such that the storage size in bytes of all relations $||\mathcal{T}_i||$ is minimized, while for each critical statement instantiation $(t, S_q, V_q) \in \mathcal{W}_{crit}$ its execution time $\mathcal{E}$ on the physical schema $\mathcal{L}$ with buffer pool size $\mathcal{BS}$ does not exceed a fixed latency threshold of statement S_q, denoted as SLA_q:*

$$\underset{R_i \in \mathcal{R}, \texttt{comp}_{i\lambda}}{\arg\min} \sum_{i=1}^{n} ||\mathcal{T}_i||$$
$$\text{subject to } \forall (t, S_q, V_q) \in \mathcal{W}_{crit} : \mathcal{E}((t, S_q, V_q), \mathcal{L}, \mathcal{BS}) \leq SLA_q.$$

2.2.3 *Buffer Pool Size Advisor*

Traditionally, disk-based database management systems employed buffer pools to manage data larger than main memory [70, 128]. Since DRAM capacity increased and DRAM costs decreased over time, it became eventually possible to store all data in DRAM and avoid the computation and memory overhead of a buffer manager [69]. Accordingly main-memory database management systems were initially designed without a traditional buffer manager [21, 87, 156]. However, research on buffer managers received renewed attention and showed how modern architectures of buffer managers achieve in-memory speed [99, 120, 151]. Modern designs of buffer managers are especially crucial for database-as-a-service providers, which face the challenge of meeting performance commitments assured in SLAs while minimizing their internal costs to enhance profitability. As database workloads are often skewed [7, 20, 46, 51, 56, 76, 102, 162] and DRAM is the primary driver of hardware costs [60, 106], database-as-a-service providers may employ a buffer pool size advisor to lower their internal costs. A buffer pool size advisor may identify the workload's working set [46] and configures the buffer pool size so that all disk pages with hot data stay in DRAM, whereas disk pages with cold data are moved to cheaper storage layers and are loaded only on demand [157].

Use Case 3 (Buffer Pool Size Advisor)**.** *A buffer pool size advisor proposes a minimal buffer pool size $\mathcal{BS} \in \mathbb{N}$, such that the execution time $\mathcal{E}$ of a workload $\mathcal{W}$ executed on the physical schema $\mathcal{L}$ with buffer pool size $\mathcal{BS}$ does not violate a maximum workload execution time, denoted as SLA:*

$$\underset{\mathcal{BS} \in \mathbb{N}}{\arg\min} \ \mathcal{BS} \qquad \text{subject to } \mathcal{E}(\mathcal{W}, \mathcal{L}, \mathcal{BS}) \leq SLA.$$

2.2.4 *Table Partitioning Advisor*

Existing table partitioning advisors focus on the classical database objective of maximizing performance [38, 73, 119, 133, 138, 149]. In this dissertation, we present a table partitioning advisor that recommends a range partitioning layout for each relation to minimize the buffer pool size while still adhering to performance commitments. Whereas a buffer pool size advisor (cf. Section 2.2.3) is a simple approach to retain data's hot working set in DRAM, its most significant drawback is that mixing hot and cold data within the same disk page pollutes the buffer pool with cold data and works against its effectiveness. This is because data are often not organized according to their access pattern. As a consequence, we group tuples that belong to hot-classified value ranges into hot range partitions, whereas tuples for cold-classified value ranges are gathered into cold range partitions. We then keep only hot-classified column partitions with a high density of hot data in the buffer pool. This avoids polluting DRAM with cold data.

To illustrate the idea of a table partitioning advisor that minimizes memory footprint, let us consider the following query:

```
SELECT o_orderpriority
FROM   orders
WHERE  o_orderdate < 1993-01-01.
```

Assume that `orders` is stored on pages in a disk-based column store with a buffer pool, that `orders` is not clustered by `o_orderdate`, and that `orders` does not have an index over `o_orderdate`. Under this assumption, the whole `o_orderdate` column must be scanned to evaluate the selection predicate. Further, to project on `o_orderpriority`, almost all pages of the `o_orderpriority` column are accessed because the qualifying tuples are likely distributed over all `o_orderpriority` pages. By contrast, the table partitioning layout $\mathcal{T}_1$ for `orders` (R_1) (cf. Figure 2.1), generated by a range partitioning specification $RPS_{13} = \{1992\text{-}01\text{-}01, 1993\text{-}01\text{-}01, 1995\text{-}01\text{-}01\}$, reduces the number of accessed pages and thus the buffer pool size. There are two reasons for this: First, due to partition pruning, only the column partition C_{131} must be scanned to evaluate the selection predicate. Second, because of the correlated storage of the `o_orderpriority` column with the `o_orderdate` column that follows from the range partitioning, only pages from column partition C_{121} are accessed to project on `o_orderpriority`.

Use Case 4 (Table Partitioning Advisor)**.** *A table partitioning advisor proposes for each relation $R_i \in \mathcal{R}$ a range partitioning specification $RPS_{i\lambda}$ to minimize the buffer pool size $\mathcal{BS}$, such that the execution time $\mathcal{E}$ of a workload $\mathcal{W}$ executed on the physical schema $\mathcal{L}$ with the buffer pool size $\mathcal{BS}$ does not violate a maximum workload execution time, denoted as SLA:*

$$\underset{R_i \in \mathcal{R}, RPS_{i\lambda} \in \mathbb{RPS}_{i\lambda}}{\arg\min} \quad \mathcal{BS} \qquad \text{subject to } \mathcal{E}(\mathcal{W}, \mathcal{L}, \mathcal{BS}) \leq SLA.$$

2.3 THE SAP HANA DATABASE

Throughout this dissertation, we present novel ideas in the context of automated physical database design advice. In order to evaluate the effectiveness of those ideas, we implement them into a prototype of the commercial, main-memory database SAP HANA [54, 97, 109, 153]. In this section, we provide an architecture overview of SAP HANA. First, we describe its main and delta fragment, which allows SAP HANA to quickly process transactional and analytical workloads on the same copy of data (cf. Section 2.3.1). Second, we give an overview of the supported compression techniques (cf. Section 2.3.2). Finally, we describe SAP HANA's buffer manager (cf. Section 2.3.3).

2.3.1 *Main and Delta Fragment*

The physical representation of a table partition in SAP HANA is split into a main and a delta fragment to efficiently process both transactional and analytical workloads.

The main fragment is read-optimized storage for analytical workloads. It consists of a dictionary and a dictionary-compressed column partition for each column partition (cf. Section 2.1.3). The dictionary-compressed column partition allows evaluating a selection during a `SELECT` statement only on the integer values in the compressed column partition. This can improve performance compared to an uncompressed partition with long strings, for which a comparison takes much longer than integer comparisons. As the dictionary is order-preserving, this works not only for equality predicates but also for more complex predicates such as range predicates. In addition, the column-scan on the dictionary-compressed column partition enables effective vectorized processing using *single instruction, multiple data* (SIMD) to improve the performance further [166, 167].

In contrast, the delta fragment is write-optimized storage for transactional workloads, i.e., it holds all modified data. The delta fragment consists of an unsorted dictionary filled by values in a first-come-first-served manner from `INSERT` and `UPDATE` statements during the workload execution. A dictionary-compressed column partition then stores the number of the value position from the unsorted dictionary, similar to the dictionary-compressed column partition of the main fragment (cf. Section 2.1.3). To still efficiently access the values of the unsorted dictionary, the delta fragment consists by default of an in-memory cache-conscious B^+ tree (CSB^+ tree) [140] to map values to the value position stored in the dictionary-compressed column partition. Such a search tree is not necessary for the main fragment as the dictionary is sorted, i.e., a binary search is used to return the value position in the dictionary for a given value.

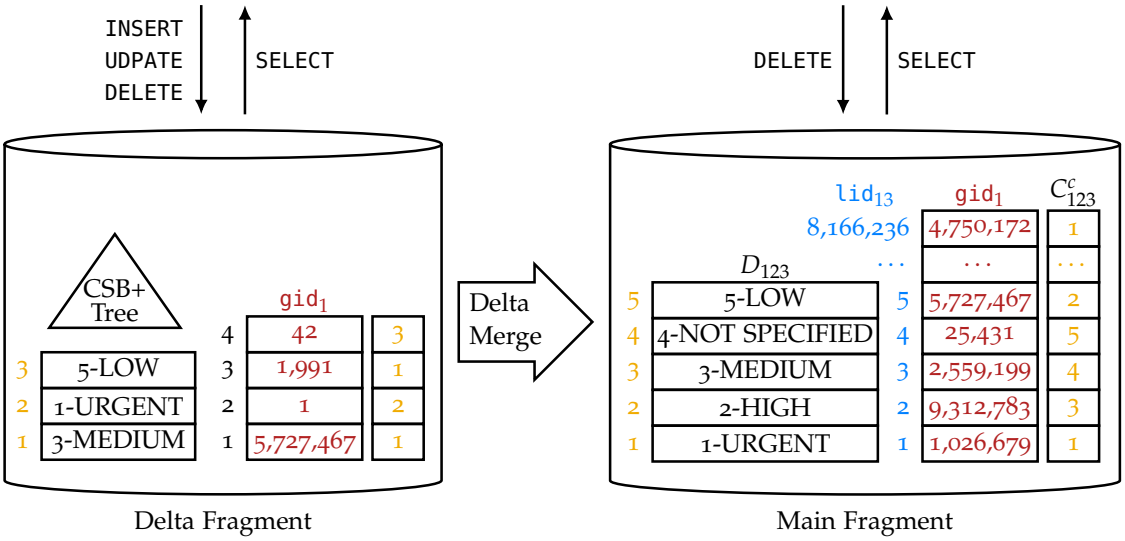
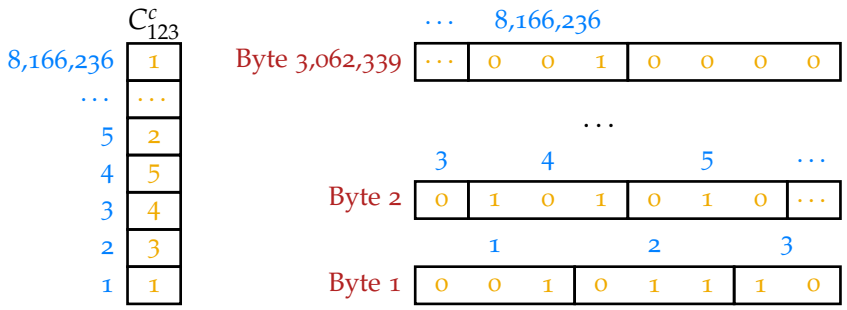


Figure 2.3: Illustration of the main and delta fragment in SAP HANA. The main fragment is read-optimized storage, whereas the delta fragment is write-optimized storage. The delta fragment is then periodically merged into the main fragment.

`DELETE` statements may access both main and delta fragments. The *multiversion concurrency control* (MVCC) manager stores visibility information in delta and main fragments. When a tuple is deleted, the tuple gets invalidated and may no longer be visible. Depending on where the tuple is stored can affect which fragments' visibility information needs to be updated.

Due to data modifications, data in the main fragment get invalidated over time. To still benefit from the read-optimized storage for analytical workloads, the modifications of the delta fragment are periodically merged into the main fragment. A system process called *mergedog* periodically checks whether or not a delta merge is necessary based on user-defined criteria, e.g., the storage size of delta fragment or the duration since the last merge. Both data structures, order-preserving dictionary and dictionary-compressed column partition, are read-optimized and thus cannot be modified easily. As a result, a delta merge can incur that both data structures are created from scratch [90].

Figure 2.3 illustrates the main and delta fragment for partition P_{123} from the table partitioning layout $\mathcal{T}_1$ for `orders` (R_1) (cf. Figure 2.1). For simplicity, we illustrate the data structures for column partition C_{123}. We depict the read-optimized main fragment on the right side, consisting of the dictionary D_{123} and the dictionary-compressed column partition C^c_{123}. The write-optimized delta fragment is shown on the left side and consists of an unsorted dictionary, a CSB$^+$ tree [140], and a dictionary-compressed column partition. In addition, we showcase the global tuple identifier in both main and delta fragments in order to identify the same tuple in both fragments. We observe four data modifications, e.g., the `o_orderpriority` of the tuple with $\texttt{gid}_1 = 5727467$ is updated from 2-HIGH to 3-MEDIUM.



(a) Compressed Column Partition

(b) Compressed Column Partition with Bit Packing

Figure 2.4: Example of applying bit packing on the dictionary-compressed column partition C^c_{123} of attribute `o_orderpriority` (A_{12}) in partition P_{13} (cf. Figure 2.1d).

2.3.2 *Data Compression*

Dictionary-compression is the default data compression technique in both main and delta fragments. Apart from dictionary compression, the compressed column partition of the main fragment is always encoded further with bit packing [165, 167]. More specifically, each number in the compressed column partition C_{ijk} is replaced by $\lceil log_2(d_{ijk}) \rceil$ number of bits, such that all numbers in $[1, d_{ijk}]$ can be represented.

For example, in Figure 2.4a, the compressed column partition C^c_{123} consists of five distinct values. One additional value is in general reserved for the `NULL` value. As a result, only 3 bits ($\lceil log_2(6) \rceil = 3$) are required to represent all numbers using bit-packing. In Figure 2.4b, we illustrate the compressed column partition C^c_{123} using bit packing. To demonstrate the benefit of bit packing for memory footprint reduction, let us calculate their storage sizes. If bit-packing is applied to column partition C^c_{123}, the `o_orderpriority` value of all 8,166,236 tuples can be stored in 748 disk pages of 4096 Bytes each, i.e., $\texttt{NUMPAGES}^c_{123} = 748$ and $\texttt{PAGESIZE}^c_{123} = 4096$ Bytes, resulting in 3.06 MB (cf. Section 2.1.3). In addition, one disk page is required to store the 5 `STRING` values of the dictionary, i.e., $\texttt{NUMPAGES}^d_{123} = 1$ and $\texttt{PAGESIZE}^d_{123} =$ 4096 Bytes. In contrast, without bit packing and assuming 32-bit unsigned integers to store the numbers in the dictionary-compressed column partition, 7984 disk pages of 4096 Bytes each (= 32.7 MB) are required. Therefore, the storage size can be reduced by a factor of 10.67 using bit packing.

Apart from dictionary compression and bit packing, SAP HANA supports optional compression techniques. The dictionary-compressed column partition can be compressed further by prefix, run-length, cluster, sparse, or indirect encoding [100, 101]. Front coding can be used to compress a string dictionary further [95, 116]. On the one hand, further compression may additionally reduce the memory footprint. Run-length encoding, for instance, can be very effective if the column

partition consists of a few distinct values and is sorted on the domain. On the other hand, yet another level of indirection is added, which may degrade performance. To illustrate, a lookup with a given local tuple identifier on a compressed column partition with run-length encoding results in a binary or exponential search because each run may group a different number of tuples. In contrast, a lookup with a given local tuple identifier on a dictionary-compressed column partition without further compression can be done in constant time because the offset in the vector can be calculated statically. Moreover, encoding dictionary-compressed column partitions and dictionaries may also degrade the performance of delta merges. This is because both data structures are read-optimized and cannot be modified easily. As a consequence, it may incur that both data structures are created from scratch. In summary, further encoding dictionary-compressed column partitions or dictionaries must be carefully considered as it is, in general, a trade-off between performance and memory footprint.

2.3.3 *Native Storage Extension*

The architecture of SAP HANA was initially designed so that only the entire column partition could be loaded into DRAM. As a result, accessing only a single value of a column partition (e.g., during tuple materialization) implied that the entire column partition was loaded from secondary storage. This wastes DRAM capacity as a huge amount of non-accessed data are loaded. SAP HANA introduced the *Native Storage Extension* to overcome this drawback, which allows loading a column partition at page granularity through a buffer manager [151, 152]. Hence, accessing only a single value of a column partition may result in accessing only a single disk page instead of the entire column partition. This extension can especially reduce the memory footprint when only a few pages of a column partition contain hot data.

The SAP HANA buffer manager supports the following seven page sizes: 4 KB, 16 KB, 64 KB, 256 KB, 1 MB, 4 MB, and 16 MB. The used page size depends on the column partition data type and their usage. For example, a string dictionary consists of a larger page size compared to a dictionary-compressed column partition with bit packing. The buffer manager consists of a free and an LRU list for each page size. In addition, a hot buffer list is used to keep pages with a high access frequency in DRAM over a more extended period. This is crucial as hot pages should not be flushed during a full-column scan, opposed to an LRU list. Further, a housekeeping thread dynamically grows and shrinks the lists of all page sizes depending on their usage, such that the allocated memory for the buffer manager (i.e., the buffer pool size) is utilized efficiently. Finally, the buffer manager can also prefetch pages before accessing them to reduce the I/O waiting time, e.g., during a full-column scan.

2.4 THE FIVE-MINUTE RULE

In this dissertation, we present novel ideas to reduce the memory footprint of a database management system while still adhering to performance commitments. As DRAM constitutes the primary driver of hardware costs [60, 106], an efficient way to reduce the memory footprint while still fulfilling SLAs is to leverage access skew in the workload [7, 20, 46, 51, 56, 76, 102, 162] by moving cold data to cheaper storage layers and keeping only hot data in DRAM. For this, data need to be classified as hot or cold. In order to draw a reasonable border between hot and cold-classified data, we consider the five-minute rule, published in 1987 by Gray and Putzolu, that states as a rule of thumb: "[d]ata referenced every five minutes should be memory resident" [65]. The argument was based on economic considerations and hardware costs of the 1980s, comparing the cost-performance ratio of DRAM and secondary storage. As prices, capacities, and performance of these two storage tiers evolve at a different pace, the five-minute rule was revisited several times [8, 63, 64], resulting in different thresholds to keep data in DRAM. As the exact threshold is not a constant but depends on the underlying hardware parameters and prices, this dissertation refers to the rule as a π-second rule, where π denotes the break-even point in seconds and is calculated as follows:

$$\pi := \frac{\text{Disk Costs [\$/sec]}}{\text{Disk IOPS [Page/sec]}} \;/\; \text{DRAM Costs [\$/Page/sec]},$$

where

Page [Bytes]	is the transfer unit of one IOP in bytes;
DRAM Costs [\$/Page/sec]	is the costs in USD for one page of main memory per seconds;
Disk Costs [\$/sec]	is the costs in USD for a single disk per seconds;
Disk IOPS [Page/sec]	is the amount of random read IOPs per second (IOPS) of a disk.

In order to illustrate the π-second rule, Table 2.2 shows current DRAM and persistent disk prices, capacities, and performance offered by Google Cloud [60]. For DRAM costs, we consider a *memory-optimized machine family* because it is offered that "[t]hese machine series are well-suited for large in-memory databases such as SAP HANA" [61], which is priced at $2610 per TB of DRAM. As page size we consider 4096 Bytes because it is the minimum page size offered by the buffer manager of SAP HANA (cf. Section 2.3.3). To calculate the π-second rule, we now consider three different types of persistent disks with different prices and performance to illustrate the impact of

Description	Parameter [Unit]	Value
Disk Page Size	Page [Bytes]	4096
Memory-optimized Memory	DRAM Costs [$/TB/month]	2610
	DRAM Costs [$/Page/sec]	$4.12 \cdot 10^{-12}$

(a) Page size and DRAM prizes of a memory-optimized Google Cloud instance.

Description	Parameter [Unit]	Value	π-Second Rule
Zonal standard persistent disks	Disk Costs [$/month] (assuming 1 TB)	40	$\pi = 4983.82$ sec
	Disk Costs [$/sec]	$1.54 \cdot 10^{-5}$	
	Disk IOPS [Page/sec]	750	
Zonal SSD persistent disks	Disk Costs [$/month] (assuming 1 TB)	170	$\pi = 530.74$ sec
	Disk Costs [$/sec]	$6.56 \cdot 10^{-5}$	
	Disk IOPS [Page/sec]	30,000	
Extreme SSD persistent disks	Disk Costs [$/month] (assuming 1 TB)	125	$\pi = 6189.32$ sec
	Prov. IOPS Costs [$/prov. IOPS/month]	0.065	
	Disk Costs (120,000 prov. IOPS) [$/sec]	$3.06 \cdot 10^{-3}$	
	Disk IOPS [Page/sec]	120,000	

(b) The calculated π-second rule for different types of persistent disks offered by Google Cloud, using the page size and DRAM prices from Table 2.2a.

Table 2.2: The π-second rule calculated from current DRAM and persistent disk prices, capacities, and performance offered by Google Cloud [60–62].

the hardware configuration on the π-second rule. Since the π-second rule does not consider the disk capacity and only the disk prices and disk performance, we assume for all disk types that 1 TB of disk space is required, similar to the capacities of disk types considered in the revisited five-minute rule in 2017 [8].

A *standard persistent disk* is the cheapest disk storage offered by Google Cloud and is priced at $40 per TB of provisioned disk space, is backed by standard hard disk drives (HDD), and reaches 750 random read IOPS with a disk size of 1 TB, resulting in $\pi = 4983.82$ seconds. However, the fastest enterprise HDDs spin at 15,000 rpm, such as *Seagate CheetAh* 15*K*.5 [148], resulting in at most 200 random read IOPS. Thus, we are concerned that the 750 random read IOPS for a standard persistent disk is constantly achievable.

In addition, Google Cloud offers *SSD persistent disks* for a "high-performance database [...] that require lower latency" [62]. These are priced at $170 per TB of disk space, backed by solid-state drives (SSD), and offer 30,000 random read IOPS with a disk size of 1 TB, resulting

in $\pi = 530.74$ seconds. Note that the π-second rule is $\approx 10\times$ lower compared to the standard persistent disk because its price is $\approx 4\times$ higher but offers $\approx 40\times$ more IOPS.

Furthermore, Google Cloud offers an *extreme SSD persistent disk*, which is "designed for high-end database workloads, such as Oracle or SAP HANA" [62], and can be provisioned of up to 120,000 IOPS. Contrary to both other disk types, a customer needs to pay per provisioned IOPS per month, too. An extreme SSD persistent disk is priced at $125 per month and one provisioned IOPS costs $0.065 per month. As a result, an extreme SSD persistent disk with 120,000 provisioned IOPS results in $\pi = 6189.32$ seconds. In this case, the π-second rule grows compared to an SSD persistent disk as the disk price increases by a factor of ≈ 50 compared to a $\approx 4\times$ higher IOPS rate.

In summary, we observe that the hardware prices, capacities, and performance significantly impact the π-second rule. One insight is that SSDs offer a better price-performance ratio compared to HDDs. However, high-performance SSDs designed for high-end databases workloads typically do not pay off as the customer needs to pay per provisioned IOPS. Finally, disk storage with a better price-performance ratio result in a lower π-second rule, such that data can be evicted earlier from DRAM to reduce the memory footprint.

3

COLLECTION OF WORKLOAD EXECUTION STATISTICS

In this chapter, we address the challenge of collecting workload execution statistics with high precision, low memory consumption, and low runtime overhead (cf. Section 1.3.1). The availability of accurate statistics about the database workload is essential as it strongly impacts the effectiveness of physical database design advisors, e.g., the quality of the recommended range partitioning layout in Chapter 4. Besides accurate statistics, we should gather statistics with low memory and runtime overhead to avoid a significant impact on performance and memory consumption for currently executed workloads. Therefore, we design and implement low-level data access counters, which collect workload execution statistics precisely, compact, and fast.

3.1 MOTIVATION

As finding an optimal physical schema that meets all requirements (e.g., with respect to memory footprint or performance) is often complex and challenging [131, 132], we introduced in Section 2.2 four use cases of automated physical database design advice: an index advisor, a data compression advisor, a buffer pool size advisor, and a table partitioning advisor. All four advisors require an objective function, e.g., aiming for minimal memory footprint or minimal query response times, while satisfying given constraints like a maximum workload execution time or a memory budget. In order to find an optimal physical schema, each advisor considers a set of physical schema candidates. A table partitioning advisor, for instance, enumerates all range partitioning specifications $\mathbb{RPS}_{i\lambda}$ (cf. Definition 5) for a certain partition-driving attribute $A_{i\lambda}$ (cf. Definition 4). Afterwards, for each physical schema candidate, the advisor calculates a change in the objective function based on the data, the workload, and the current physical schema. During this calculation, accurate statistics about the workload are of particular importance for the effectiveness of the physical database design advice.

In order to demonstrate how accurate statistics about the workload influence the decision of physical database design advisors, let us consider the four advisors introduced in Section 2.2. First, an index advisor (cf. Use Case 1) requires precise knowledge of query predicate selectivities. This is because an index configuration can only improve query response times if it includes an index over a set of attributes that is part of a selective filter predicate. Second, a data compression

advisor (cf. Use Case 2) relies on precise knowledge on how much data are sequentially (e.g., during full-column scans) or randomly accessed (e.g., during tuple materialization). The reason is that compression adds one more level of indirection and may degrade performance. In contrast, compression helps to keep a larger fraction of the working set in DRAM or CPU caches. To illustrate, in a dictionary-compressed column store, a projection accesses at least one page of the dictionary and one page of the dictionary-compressed column partition, whereas only one page of the uncompressed column partition might have been accessed in an uncompressed setting (cf. Section 2.1.3). As a result, compressing a column partition accessed by many projections may be avoided. Third, a buffer pool size advisor (cf. Use Case 3) is based on precise page access statistics to identify the workload's working set, such that the buffer pool size can be configured to hold only pages with hot data in DRAM. Finally, a table partitioning advisor (cf. Use Case 4) needs accurate access statistics at tuple- or value-granularity. A range partition can group data items that belong to either a hot- or cold-classified value range of a certain attribute of that table. To give an example, all tuples of `orders` with an `o_orderdate` in the year 1995, which are never accessed, could be grouped into a range partition that is evicted to secondary storage and are loaded only on demand to reduce the memory footprint of the database.

Apart from describing an executed workload accurately, the statistics should ideally be collected with low memory and runtime overhead on currently executed workloads. This is especially crucial to database-as-a-service providers, which aim at optimizing the physical schema of hosted database instances quickly and without noticeable impacting their current performance and memory consumption. Adjusting the physical schema is important to database-as-a-service providers as internal costs can be reduced to enhance profitability. Whenever the physical schema is defined in a way where hot disk pages contain mainly hot data, the pollution of the buffer pool with cold data can be avoided, and DRAM capacities are not wasted anymore.

Obviously, there is a trade-off between the accuracy of workload execution statistics and their memory and runtime overhead. However, in practice nowadays, workload execution statistics are either gathered offline or with low precision. Offline approaches execute a representative sample of the workload on a separate node [4, 38, 141], analyze log samples [102], or caches runtime access patterns [67]. The drawback of these approaches is that they result in high memory consumption and runtime overhead. Other approaches collect workload execution statistics by tracking coarse-granular access frequencies and combining them with sampling [56, 76, 122]. These approaches have a low runtime and memory overhead but lack in precision. Hence, no current approach collects workload execution statistics with high precision, low memory consumption, and low runtime overhead.

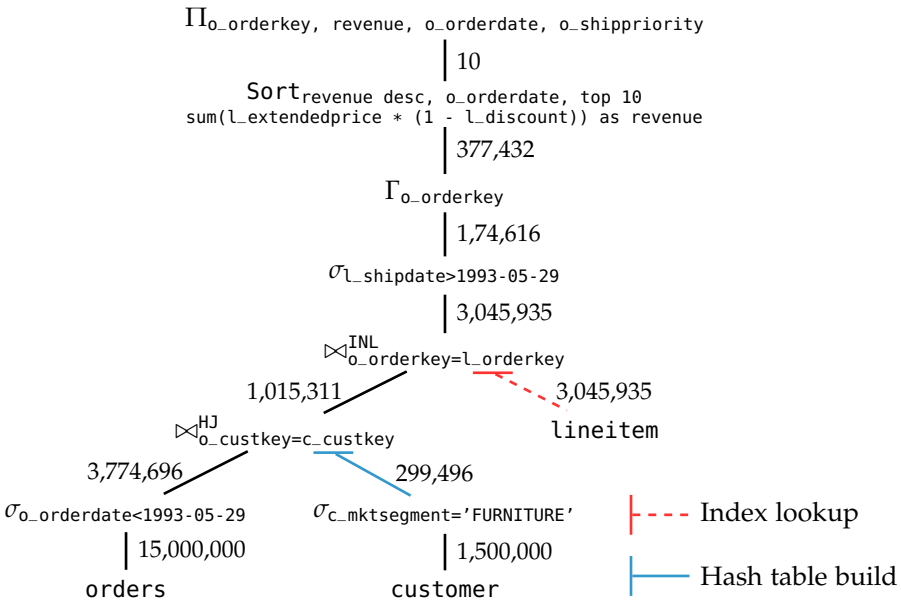


Figure 3.1: Illustration of the physical execution plan produced by SAP HANA's query optimizer [54, 97, 109, 153] for the instantiation of statement S_3 of the JCC-H benchmark (cf. Listing 2.1) with vector $V_3 = [\text{FURNITURE}, \text{1993-05-29}]$ and scale factor 10.

3.2 PROBLEM STATEMENT

In this section, we define the workload execution statistics that need to be collected and formalize the problem of providing workload execution statistics with high precision, low memory consumption, and low runtime overhead as input to physical database design advisors. Throughout this section, we show exemplary the collected workload execution statistics for statement S_3 of the JCC-H benchmark with scale factor 10 (cf. Listing 2.1), instantiated with vector $V_3 = [\text{FURNITURE}, \text{1993-05-29}]$. Figure 3.1 shows the corresponding physical execution plan produced by SAP HANA's query optimizer [54, 97, 109, 153].

Definition 21 (Workload Execution Statistics for an Index Advisor)**.** *Let $|\sigma_p(R_i)|$ be the output cardinality of a selection $\sigma_p(R_i) \in T(t, S_q, V_q)$ in the physical execution plan on a base relation $R_i \in \mathcal{R}$. Further, let $\mathcal{F}(p)$ be the free attributes contained in the selection predicate p. We define by $FStat_{idx}$ the workload execution statistics for an index advisor, which stores a tuple $(|\sigma_p(R_i)|, \mathcal{F}(p))$ for each selection $\sigma_p(R_i) \in T(t, S_q, V_q), (t, S_q, V_q) \in \mathcal{W}$ that consists of an index-SARGable predicate p.*

Table 3.1 shows the workload execution statistics $FStat_{\text{idx}}$ for an index advisor collected during the execution of the physical execution plan in Figure 3.1. We observe collected statistics for two selections. First, the selection $\sigma_{\texttt{o\_orderdate<1993-05-29}}(\texttt{orders})$, where `o_orderdate` is a free attribute, has an output cardinality of 3,774,696, resulting in a selectivity of 0.25 as `orders` contains 15,000,000 tuples. Second,

$\sigma_p(R_i)$	$\lvert\sigma_p(R_i)\rvert$	$\frac{\lvert\sigma_p(R_i)\rvert}{\lvert R_i\rvert}$	$\mathcal{F}(p)$
$\sigma_{\texttt{o\_orderdate<1993-05-29}}(\texttt{orders})$	3,774,696	0.25	{o_orderdate}
$\sigma_{\texttt{c\_mktsegment='FURNITURE'}}(\texttt{customer})$	299,496	0.20	{c_mktsegment}

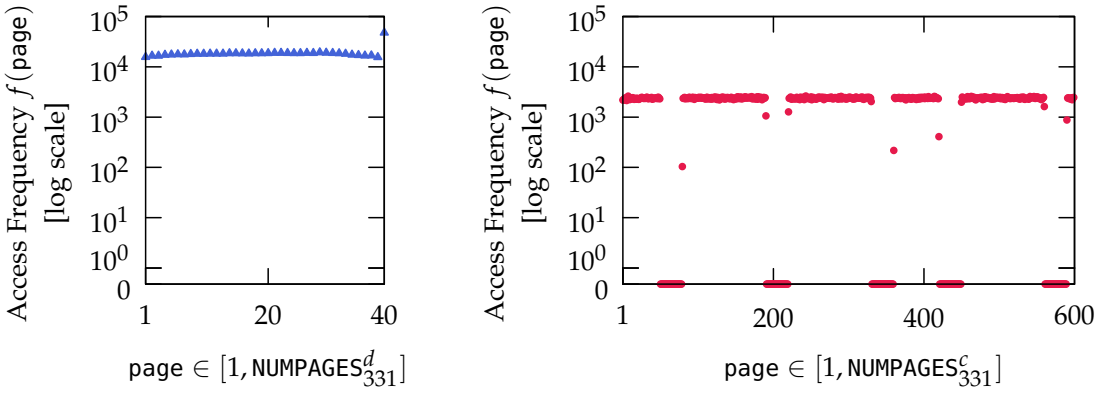
Table 3.1: Collected workload execution statistics $FStat_{\text{idx}}$ for an index advisor while executing the physical execution plan of Figure 3.1. For each selection $\sigma_p(R_i)$, the output cardinality $|\sigma_p(R_i)|$ is stored with the free attributes on which the predicate p is applied for.

the selection $\sigma_{\texttt{c\_mktsegment='FURNITURE'}}(\texttt{customer})$ with c_mktsegment as a free attribute has an output cardinality of 299,496, leading to a selectivity of 0.20 as customers contains 1,500,000 tuples. The selection with l_shipdate as a free attribute is not recorded since it is not performed on a base relation (cf. Figure 3.1). An index advisor might propose first an index over c_mktsegment as it is part of the most selective filter predicate. Moreover, an index over o_orderdate could be recommended depending on the memory budget.

Definition 22 (Workload Execution Statistics for a Data Compression Advisor)**.** *We define by $FStat_{comp}$ the workload execution statistics for a data compression advisor as a pair (s_{ijk}, r_{ijk}) for each column partition C_{ijk}, where s_{ijk} is the number of tuples in C_{ijk} that were sequentially accessed by workload $\mathcal{W}$ (e.g., by a selection $\sigma_p(R_i) \in T(t, S_q, V_q), (t, S_q, V_q) \in \mathcal{W}$, where p contains A_{ij}), and r_{ijk} is the number of tuples in C_{ijk} that were randomly accessed by workload $\mathcal{W}$ (e.g., by a projection $\Pi_{A_{ij}}(e) \in T(t, S_q, V_q), (t, S_q, V_q) \in \mathcal{W}$ on some expression e).*

R_i	Relation	A_{ij}	Attribute	P_{ik}	C_{ijk}	s_{ijk}	r_{ijk}
R_1	orders	A_{11}	o_orderkey	P_{11}	C_{111}	0	1,015,311
		A_{12}	o_orderdate	P_{11}	C_{121}	15,000,000	377,432
		A_{13}	o_custkey	P_{11}	C_{131}	0	3,774,696
		A_{14}	o_shippriority	P_{11}	C_{141}	0	10
R_2	customer	A_{21}	c_custkey	P_{11}	C_{211}	0	299,496
		A_{22}	c_mktsegment	P_{11}	C_{221}	1,500,000	0
R_3	lineitem	A_{31}	l_orderkey	P_{11}	C_{311}	0	3,045,935
		A_{32}	l_discount	P_{11}	C_{321}	0	1,074,616
		A_{33}	l_extendedprice	P_{11}	C_{331}	0	1,074,616
		A_{34}	l_shipdate	P_{11}	C_{341}	0	3,045,935

Table 3.2: Collected workload execution statistics $FStat_{\text{comp}}$ for a data compression advisor during the execution of the physical execution plan of Figure 3.1. For each column partition C_{ijk} of unpartitioned orders, customer, and lineitem tables, the number of tuples that were sequentially (s_{ijk}) and randomly (r_{ijk}) accessed are stored.



(a) Dictionary

(b) Dictionary-Compressed Column Partition

Figure 3.2: Collected workload execution statistics $FStat_{\text{buf}}$ for a buffer pool size advisor during the execution of the physical execution plan of Figure 3.1. For a column partition C_{331} on attribute `l_extendedprice` (A_{33}) of an unpartitioned `lineitem` (R_3) table, the y-axis shows the access frequency $f(\texttt{page})$ to each `page` (x-axis), where in (a) $\texttt{page} \in [1, \texttt{NUMPAGES}^d_{ijk}]$ belongs to the dictionary D_{331} and in (b) $\texttt{page} \in [1, \texttt{NUMPAGES}^c_{ijk}]$ belongs to the dictionary-compressed column partition C^c_{331}.

Table 3.2 shows the collected workload execution statistics $FStat_{\text{comp}}$ for a data compression advisor during the execution of the physical execution plan of Figure 3.1. In this example, the tables of `orders` (R_1), `customer` (R_2), and `lineitem` (R_3) are unpartitioned. We observe that column partition C_{221} that belongs to attribute `c_mktsegment` (A_{22}) exposes only sequential but no random accesses. As a result, a data compression advisor might suggest compression because no performance slow down is expected. A data compression advisor might also propose compression on column partition C_{141} that belongs to attribute `o_shippriority` (A_{14}) because only 10 tuples are randomly accessed. For this column partition, the data compression advisor needs to consider the trade-off between the gain in reducing the memory footprint due to compression and the potential loss of performance due to the 10 random accesses.

Definition 23 (Workload Execution Statistics for a Buffer Pool Size Advisor)**.** *We define by $FStat_{buf}$ the workload execution statistics for a buffer pool size advisor that stores the access frequency $f(\texttt{page})$ by workload $\mathcal{W}$ for each* `page`*, where* $\texttt{page} \in [1, \texttt{NUMPAGES}^u_{ijk}]$ *is defined for an uncompressed column partition C^u_{ijk},* $\texttt{page} \in [1, \texttt{NUMPAGES}^d_{ijk}]$ *for a dictionary, and* $\texttt{page} \in [1, \texttt{NUMPAGES}^c_{ijk}]$ *for a dictionary-compressed column partition.*

Figure 3.2 shows the access frequency $f(\texttt{page})$ (y-axis) to each `page` (x-axis) of the column partition C_{331} that belongs to attribute `l_extendedprice` (A_{33}) of an unpartitioned `lineitem` (R_3) table with dictionary compression. In Figure 3.2a, we illustrate the access frequency to each page of the dictionary D_{331} (40 pages), while in Figure 3.2b, we

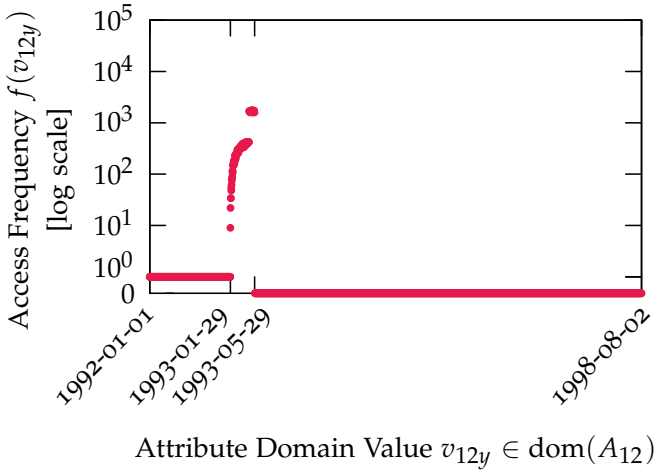


Figure 3.3: Collected workload execution statistics $FStat_{\text{part}}$ for a table partitioning advisor during the execution of the physical execution plan of Figure 3.1. The y-axis shows the access frequency $f(v_{12y})$ to each value v_{12y} of `o_orderdate` (A_{12}) (x-axis).

show the access frequency to each page of the dictionary-compressed column partition C^c_{331} (600 pages). We observe that only ≈75% of the pages of the dictionary-compressed column partition are accessed. A buffer pool size advisor would identify the number of pages with an access frequency above a certain threshold to set the buffer pool size in a way that only the hot-classified pages fit in DRAM.

Definition 24 (Workload Execution Statistics for a Table Partitioning Advisor)**.** *We define by $FStat_{part}$ the workload execution statistics for a table partitioning advisor that stores for each attribute domain value $v_{ijy} \in \text{dom}(A_{ij})$ the access frequency $f(v_{ijy})$ by workload $\mathcal{W}$, where $f(v_{ijy})$ is the sum of sequential accesses to A_{ij} by statement instantiations $(t, S_q, V_q) \in \mathcal{W}$, such that $\exists R_i[\texttt{gid}_i].A_{ij} = v_{ijy}$ that is part of the matching tuples (e.g., by a selection $\sigma_p(R_i) \in T(t, S_q, V_q), (t, S_q, V_q) \in \mathcal{W}$ where p references A_{ij} and v_{ijy} satisfy p), and random accesses to A_{ij} by statement instantiations $(t, S_q, V_q) \in \mathcal{W}$, where $R_i[\texttt{gid}_i].A_{ij} = v_{ijy}$ (e.g., by a projection $\Pi_{A_{ij}}(e) \in T(t, S_q, V_q), (t, S_q, V_q) \in \mathcal{W}$ on some expression e).*

We record only sequential accesses to tuples that match the selection predicate because we assume that a range partition generated for a value v_{ijy} is pruned, i.e., not accessed, if the value does not satisfy the selection predicate (cf. Section 2.2.4).

Figure 3.3 shows for each value of the active domain of `o_orderdate` (A_{12}) (x-axis) the corresponding access frequency (y-axis) after execution of the physical execution plan of Figure 3.1. A table partitioning advisor might propose a hot range partition for tuples with an `o_orderdate` between 1993-01-29 and 1993-05-28 since only those values are accessed frequently. In contrast, tuples with an `o_orderdate` larger than 1993-05-28 are not accessed. As a result, a table partition-

ing advisor might propose a cold table partition for tuples with an `o_orderdate` larger than 1993-05-28 as this range partition will be pruned by the selection $\sigma_{\texttt{o\_orderdate<1993-05-29}}(\texttt{orders})$ (cf. Figure 3.1).

Based on our definitions of workload execution statistics for an index advisor, a data compression advisor, a buffer pool size advisor, and a table partitioning advisor, we now state the problem of providing an approximation of workload execution statistics with high precision, low memory consumption, and low runtime overhead.

Problem 1. *The problem we consider is to provide an approximation of workload execution statistics $FStat_{idx}$, $FStat_{comp}$, $FStat_{buf}$, and $FStat_{part}$, which are precise (i.e., the statistics should be as accurate as possible), compact (i.e., the memory footprint compared to the storage size should be as small as possible), and fast (i.e., the runtime overhead during workload execution should be as low as possible).*

3.3 DATA ACCESS COUNTERS

In this section, we present data structures for collecting precise, compact, and fast workload execution statistics. We begin by describing the data structure for an index advisor (cf. Section 3.3.1), followed by data structures for a data compression advisor (cf. Section 3.3.2), a buffer pool size advisor (cf. Section 3.3.3), and a table partitioning advisor (cf. Section 3.3.4).

3.3.1 *Use Case 1: Index Advisor*

The most popular approach of providing workload execution statistics $FStat_{\text{idx}}$ for an index advisor is to gather the instantiations of SQL statements and feed them into the optimizer's what-if API to evaluate how the workload execution time would change if the index configuration is adopted [4, 141]. This is an offline approach because the optimizer's what-if API is called with a representative sample of the workload on a separate node. It fails to provide accurate statistics as it relies on precise cardinality estimates.

To address these limitations, we track the actual output cardinalities of all selections $\sigma_p(R_i) \in T(t, S_q, V_q), (t, S_q, V_q) \in \mathcal{W}$ contained in the workload $\mathcal{W}$ at execution time. Since tracking the exact output cardinalities $|\sigma_p(R_i)|$ of all selections would consume too much memory, we introduce a threshold parameter $\phi_{idx} \in (0, 1]$ to capture only selections with an output cardinality less than $\phi_{idx} \cdot |R_i|$. The reason is that only selective predicates benefit from indexes [88]. To reduce the memory overhead further, we group the actual output cardinalities into intervals and only count the number of selections per interval:

$$[b^r, b^{r+1}), b \in \mathbb{R}_{>0}, 0 \leq r \leq \lceil log_b(\phi_{idx} \cdot |R_i|) \rceil.$$

The estimated output cardinality for selections that are recorded to the interval $[b^r, b^{r+1})$ is $\sqrt{b^r \cdot b^{r+1}}$. Obviously, the maximum error between the actual and recorded output cardinality is $\sqrt{b}$ for arbitrary complex predicates. In our experiments in Section 3.4, we set the interval base parameter b to 2, such that the actual and recorded output cardinalities differ at most by a factor of $\sqrt{2}$.

Since an index advisor may recommend multi-column indexes, we would need one set of intervals per combination of free attributes per relation R_i with m_i attributes, i.e., in total, $2^{m_i} - 1$ ($= |\mathcal{P}(\mathcal{A}(R_i)) \setminus \{\}|$) set of intervals. Accordingly, the memory consumption of our approach using 32-bit counters for a relation R_i with m_i attributes would be $(\lceil log_b(\phi_{idx} \cdot |R_i|)\rceil + 1) \cdot (2^{m_i} - 1) \cdot 4$ Bytes, which may not fulfill the requirement on compactness for relations with many attributes. As a result, to meet the memory requirements, we propose lazy counters, only created if (i) the corresponding combination of free attributes actually occurred in selection predicates, and (ii) the selectivity of this attribute combination is below ϕ_{idx}. We argue that this number of attribute combinations is significantly smaller than the number of all attribute combinations. For example, creating counters with $b = 2$ for all attribute combinations of `lineitem` of the TPC-H benchmark [159] with 16 attributes and 60,000,000 tuples (scale factor 10) would require 0.32% of the storage size of `lineitem` in SAP HANA (1.90 GB). In contrast, our lazy counters constitute only 0.02% of the storage size. The reasons are twofold. First, in the set of 22 SQL statements of TPC-H, selection predicates contain only a limited number of free attributes. Second, some attributes are never referenced in a selection predicate. Accordingly, there exists a large number of attribute combinations where no counters are created.

Apart from a low memory consumption due to the lazy counters and a high precision as the actual and recorded output cardinality differ at most by a factor of $\sqrt{b}$ for arbitrary complex predicates, Section 3.4 demonstrates that our approach also has a low runtime overhead. We summarize the presented data structure in the following:

Access Counter 1 (Index Advisor)**.**

PHYSICAL ACCESSES: *We consider each selection $\sigma_p(R_i) \in T(t, S_q, V_q)$, $(t, S_q, V_q) \in \mathcal{W}$ consisting of an index-SARGable predicate p, and its actual output cardinality $|\sigma_p(R_i)|$, collected during execution.*

LAZY COUNTERS: *For a base $b \in \mathbb{R}$ and a set of attributes $\mathbb{A}_{is} \in \mathcal{P}(\mathcal{A}(R_i))$, we create and maintain integer counters $X^{idx}_{is0}, ..., X^{idx}_{isr} ..., X^{idx}_{is\lceil log_b(\phi_{idx} \cdot |R_i|)\rceil}$ if there exists a selection $\sigma_p(R_i) \in T(t, S_q, V_q)$, $(t, S_q, V_q) \in \mathcal{W}$ such that $\mathbb{A}_{is} \subseteq \mathcal{F}(p)$ and $|\sigma_p(R_i)| < \phi_{idx} \cdot |R_i|$.*

INTERVAL COUNTING: *A counter X^{idx}_{isr} is incremented by 1 for a selection $\sigma_p(R_i) \in T(t, S_q, V_q), (t, S_q, V_q) \in \mathcal{W}$ if $|\sigma_p(R_i)| > 0$ and $r = \lceil log_b(|\sigma_p(R_i)|)\rceil$ and $|\sigma_p(R_i)| < \phi_{idx} \cdot |R_i|$. For $|\sigma_p(R_i)| = 0$, the counter X^{idx}_{is0} is incremented by 1.*

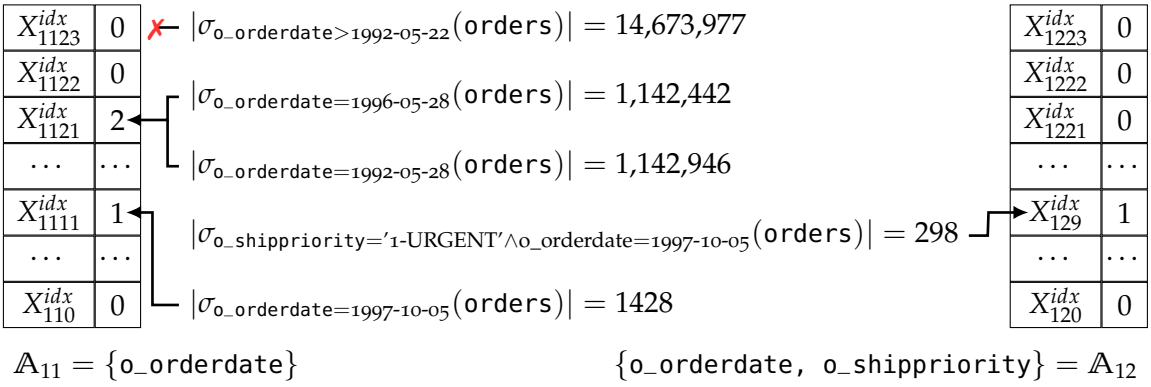


Figure 3.4: Illustration of how Access Counter 1 collects workload execution statistics $FStat_{\text{idx}}$ for an index advisor with high precision, low memory consumption, and low runtime overhead. For each set of attributes $\mathbb{A}_{is} \in \mathcal{P}(\mathcal{A}(R_i))$, a set of lazy counters is maintained.

Figure 3.4 shows how Access Counter 1 with base $b = 2$ collects workload execution statistics $FStat_{\text{idx}}$ for an index advisor for five selections on `orders` (R_1) with scale factor 10 (15,000,000 tuples). We show the access counters for selection predicates containing attribute `o_orderdate` (left), and selection predicates containing `o_orderdate` and `o_shippriority` (right). The first selection on `o_orderdate` matches 14,673,977 tuples. As 14,673,977 is larger than $\phi_{idx} \cdot |R_1|$ for $\phi_{idx} = 0.1$ no counter is updated. The counter X^{idx}_{1121} is updated twice, by the second ($\lceil log_2(1{,}142{,}442) \rceil = 21$) and the third selection ($\lceil log_2(1{,}142{,}946) \rceil = 21$). The fourth selection updates the counter X^{idx}_{129} for the attribute set $\mathbb{A}_{12}$ of `o_orderdate` and `o_shippriority` as 298 tuples match, and both attributes are referenced in the predicate.

We also emphasize that the index advisor will evaluate the ordering of attributes for multi-column indexes. For example, let us assume that our lazy counters only tracked output cardinalities over $\mathbb{A}_{11} = \{\texttt{o\_orderdate}\}$ and $\mathbb{A}_{12} = \{\texttt{o\_orderdate, o\_shippriortiy}\}$, as illustrated in Figure 3.4. An index advisor then may recommend only an index $\mathbb{I}_{12}$ over $\mathbb{A}_{12}$ with the attribute ordering `o_orderdate, o_shippriortiy`. The reason is that this index can also be used for selections that reference only `o_orderdate` as it is ranked first in the ordering of the attributes. In contrast, creating an index $\mathbb{I}_{12}$ over $\mathbb{A}_{12}$ with the attribute ordering `o_shippriortiy, o_orderdate` could not be used for selections that reference only `o_orderdate`. In this case, an additional index $\mathbb{I}_{11}$ over $\mathbb{A}_{11}$ should be created.

Finally, Access Counter 1 can be extended to collect the output cardinality of a join $e \bowtie_{A_{i'j'}=A_{ij}} R_i$, where e is an arbitrary expression (e.g., a selection $\sigma_p(R_{i'})$). The reason is that an index over attribute A_{ij} may improve the performance if the output cardinality of the expression e is small. Traversing the index over A_{ij} is then faster than building a hash table over A_{ij}.

3.3.2 *Use Case 2: Data Compression Advisor*

Existing approaches that collect workload execution statistics $FStat_{\text{comp}}$ for a data compression advisor do not consider the type of access (i.e., sequential vs. random access). This is crucial as compression adds one more level of indirection and thus random accesses (e.g., by tuple materialization) may degrade performance. To address this shortcoming, we propose to count both the number of tuples accessed sequentially and randomly by the workload. Maintaining just two counters per column partition fulfills the space efficiency requirement. In Section 3.4, we show that our approach also achieves a low runtime overhead. We also emphasize that in addition to workload execution statistics $FStat_{\text{comp}}$, data characteristics (e.g., number of distinct values, value distribution, or whether data are sorted) are needed to propose an optimal compression technique (cf. Section 2.2.2). However, modern database management systems provide this information with high accuracy already [40]. Therefore, workload execution statistics are sufficient to propose if and how a column partition should be compressed. We summarize the presented access counter in the following:

Access Counter 2 (Data Compression Advisor)**.**

PHYSICAL ACCESSES: *We consider the physical data accesses during execution of workload $\mathcal{W}$.*

ACCESS TYPE: *For each column partition C_{ijk}, we create and maintain an integer counter X_{ijk}^{seq}, which tracks the number of tuples sequentially accessed in C_{ijk}, and an integer counter X_{ijk}^{rand}, which tracks the number of tuples randomly accessed in C_{ijk}.*

Figure 3.5 shows how Access Counter 2 collects workload execution statistics $FStat_{\text{comp}}$ for a data compression advisor while execution of the physical execution plan of Figure 3.1. In this example, the tables of `orders` (R_1), `customer` (R_2), and `lineitem` (R_3) are unpartitioned. For each accessed column partition C_{ijk}, we show the value of the access counters X_{ijk}^{seq} and X_{ijk}^{rand} after executing the physical execution plan with SAP HANA. Data accesses by an operator in the physical execution plan (bottom) and updating the corresponding access counters X_{ijk}^{seq} and X_{ijk}^{rand} (top) are highlighted using a unique color and number ⓧ to identify the operator in the physical execution plan that making the accesses. For example, the selection $\sigma_{\texttt{o\_orderdate<1993-05-29}}$ makes 15,000,000 sequential tuple accesses to the column partition that belongs to `o_orderdate`. The sorting operator after grouping on `o_orderkey` makes only 377,432 random accesses for materializing tuples of `o_orderdate` but 1,074,616 random accesses for materializing tuples of `l_discount` and `l_extendedprice`. The reason is that an order in JCC-H typically comprises 3 items. Note that in contrast to TPC-H, where an order generally comprises 4 items, in JCC-H, there are 5 special orders, where each comprises 3 million items.

Attribute	C_{ijk}	X_{ijk}^{seq}	X_{ijk}^{rand}
o_orderkey	C_{111}	0	❹ 1,015,311
o_custkey	C_{112}	0	❸ 3,774,696
o_orderdate	C_{113}	❷ 15,000,000	❼ 377,432
o_shippriority	C_{114}	0	❽ 10
c_custkey	C_{211}	0	❸ 299,496
c_mktsegment	C_{212}	❶ 1,500,000	0
l_orderkey	C_{311}	0	❹ 3,045,935
l_discount	C_{312}	0	❼ 1,074,616
l_extendedprice	C_{313}	0	❼ 1,074,616
l_shipdate	C_{314}	0	❺ 3,045,935

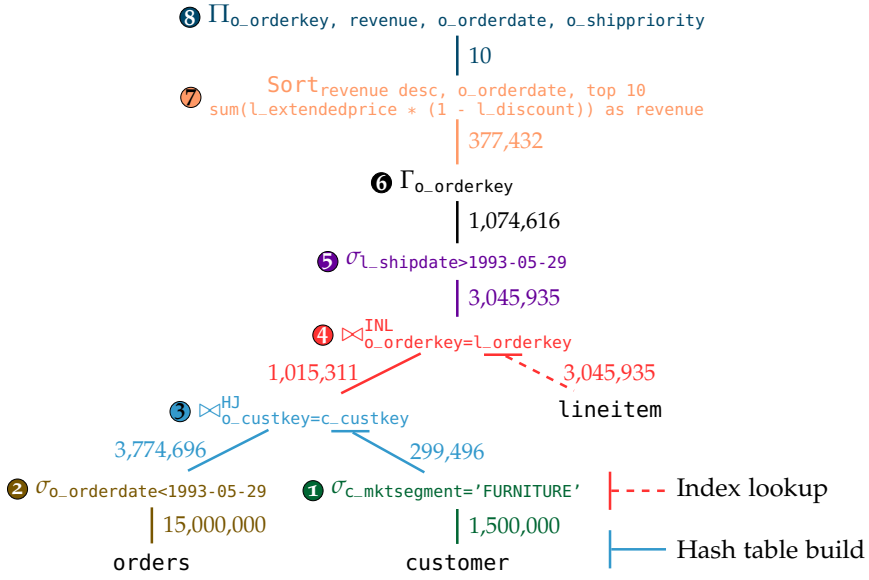


Figure 3.5: Illustration of how Access Counter 2 collects workload execution statistics $FStat_{\text{comp}}$ for a data compression advisor with high precision, low memory consumption, and low runtime overhead. For each column partition C_{ijk}, the number of tuples that were sequentially X_{ijk}^{seq} and randomly X_{ijk}^{rand} accessed are stored. Data accesses by an operator in the physical execution plan and updating the corresponding access counters X_{ijk}^{seq} and X_{ijk}^{rand} are highlighted using a unique color and number ⓧ to identify the operator in the physical execution plan that caused that access.

3.3.3 *Use Case 3: Buffer Pool Size Advisor*

To collect workload execution statistics $FStat_{\text{buf}}$ for a buffer pool size advisor, existing approaches use an integer counter per disk page [56, 76]. This approach provides precise access frequencies of pages but may not fulfill the requirement on the runtime overhead. For example, for an uncompressed column partition C^u_{ijk}, in the worst case, $\texttt{NUMPAGES}^u_{ijk}$-many counters need to be incremented during a full-column scan, i.e., the counters of all disk pages of C^u_{ijk} are updated (cf. Section 2.1.3). Instead of updating the counters of all accessed disk pages individually, we propose to update only the respective start and end page counters. More specifically, if a physical execution plan accesses the pages $[u, w]$, where $u, w \in [1, \texttt{NUMPAGES}^u_{ijk}]$, the corresponding counter to page u is incremented, while the counter of page $w + 1$ is decremented. This enables counter updates in constant time. Since page w is the last accessed page, we decrement the counter of the following page. Thus, in total $\texttt{NUMPAGES}^u_{ijk} + 1$ counters for an uncompressed column partition are needed. After collecting the workload execution statistics, the final page access frequencies are derived by calculating the prefix sum of the counters up to the target page.

Furthermore, we argue that the statistics are updated considerably more frequently than they are read (e.g., after a sampling phase) and that we meet the runtime overhead requirements. In addition, the memory overhead is low because only a single 64-bit signed atomic integer counter per page is stored. The counters are signed as their value may become negative. Since SAP HANA supports page sizes between 4 KB and 16 MB (cf. Section 2.3.3), the memory overhead varies between 0.2% (64 bit/4 KB) and 0.00005% (64 bit/16 MB). Moreover, we achieve high precision as we track all physical page accesses. We present the data structure below:

Access Counter 3 (Buffer Pool Size Advisor)**.**

PHYSICAL ACCESSES: *We consider the physical data accesses during execution of workload $\mathcal{W}$.*

START/END BLOCK COUNTING: *We create and maintain integer counters $X^{buf}_1, \ldots, X^{buf}_{\texttt{NUMPAGES}^u_{ijk}+1}$ for each uncompressed column partition C^u_{ijk}, $X^{buf}_1, \ldots, X^{buf}_{\texttt{NUMPAGES}^d_{ijk}+1}$ for each dictionary D_{ijk}, and $X^{buf}_1, \ldots, X^{buf}_{\texttt{NUMPAGES}^c_{ijk}+1}$ for each compressed column partition C^c_{ijk}. For physical accesses to disk pages in the range $[u, w]$, where $u, w \in [1, \texttt{NUMPAGES}^u_{ijk}]$, $u, w \in [1, \texttt{NUMPAGES}^d_{ijk}]$, or $u, w \in [1, \texttt{NUMPAGES}^c_{ijk}]$, counter X^{buf}_u is incremented by 1, and counter X^{buf}_{w+1} is decremented by 1.*

ACCESS FREQUENCY: *The access frequency $f(\texttt{page})$ for a $\texttt{page}$ is defined as $f(\texttt{page}) = \sum_{u=1}^{\texttt{page}} X^{buf}_u$, where $\texttt{page} \in [1, \texttt{NUMPAGES}^u_{ijk}]$ for C^u_{ijk}, $\texttt{page} \in [1, \texttt{NUMPAGES}^d_{ijk}]$ for D_{ijk}, or $\texttt{page} \in [1, \texttt{NUMPAGES}^c_{ijk}]$ for C^c_{ijk}.*

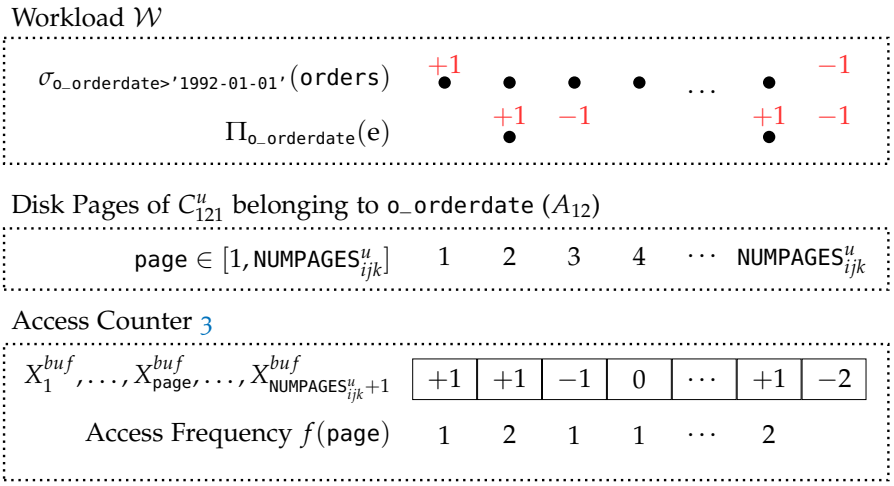


Figure 3.6: Illustration of how Access Counter 3 collects workload execution statistics $FStat_{\text{buf}}$ for a buffer pool size advisor with high precision, low memory consumption, and low runtime overhead. For each page $\in [1, \texttt{NUMPAGES}^u_{ijk}]$ an integer counter $X^{buf}_{\texttt{page}}$ is maintained. The counter is incremented or decremented by 1 depending on the accesses by the workload $\mathcal{W}$.

Figure 3.6 shows how Access Counter 3 collects workload execution statistics $FStat_{\text{buf}}$ for a buffer pool size advisor for a selection and a projection on o_orderdate. In this example, orders (R_1) is unpartitioned. For each page $\in [1, \texttt{NUMPAGES}^u_{ijk}]$ of the uncompressed column partition C^u_{121} that belongs to attribute o_orderdate (A_{12}), we maintain one counter $X^{buf}_{\texttt{page}}$. The projection $\Pi_{\texttt{o\_orderdate}}(\texttt{e})$, after some expression e, increments the counter of the accessed page by 1 and decrements the counter of the following page by 1. In contrast, the selection $\sigma_{\texttt{o\_orderdate>'1992-01-01'}}(\texttt{orders})$ is a full-column scan but updates only the first counter by $+1$ and the last counter by -1. Note that for accesses to the last page, the counter $X^{buf}_{\texttt{NUMPAGES}^u_{ijk}+1}$ is decremented. We compute the prefix sum of the counters up to the target page to obtain the access frequencies of individual pages. For example, the second page has an access frequency of 2 ($= X^{buf}_1 + X^{buf}_2$).

3.3.4 *Use Case 4: Table Partitioning Advisor*

A naïve approach of tracking the access frequencies of values in the active attribute domain is to group values into value ranges and to increment a value range counter by one whenever a value or sub-range of the value range is accessed. With the counter representing the access frequency of each value in the range, frequencies are overestimated substantially. Instead, we propose to increase the value range counter by one when only a single value of the range was accessed (e.g., during tuple materialization) and increase the value range counter

by the number of values in the range (i.e., the block size) when all values of the range were accessed (e.g., during a full-column scan). The access frequency of a value is then obtained by dividing the value range counter by the block size. The calculated access frequencies are nevertheless prone to skewed access patterns. More specifically, access frequencies of heavy hitters are underestimated, whereas frequencies of rarely accessed values (i.e., the long tail) are overestimated.

To improve precision in such situations, we propose employing the space-saving algorithm and its stream-summary data structure [110] to monitor the top-h most frequently accessed values of an attribute. The stream-summary data structure creates and monitors h counters for h values. If a value inserted into the stream-summary is already monitored, the corresponding counter is incremented by 1. Otherwise, the value with the lowest counter is replaced by the inserted value. The counter of the replaced value is incremented by 1 and is then used for the inserted value. Accordingly, values that correspond to the counters with one of the lowest counts in the stream-summary tend to be overestimated as their counters have likely been incremented by many different values during the workload. In contrast, values that correspond to the counters with one of the highest counts in the stream-summary tend to be accurate (or only slightly overestimated) as these values have likely been monitored by the same counter during the entire workload. As a consequence, we need to identify which values monitored in the stream-summary are true heavy hitters. To do this, we consider the counter of the value range where the value belongs. Since we assume that the estimated frequency of a heavy hitter can only be slightly overestimated, we conclude that the estimated frequency of a heavy hitter by the stream-summary is only slightly higher than its corresponding value range counter. Therefore, we introduce a tolerance parameter $\phi_{part} \in \mathbb{R}_{\geq 1}$, such that the estimated access frequency of the stream-summary is only considered if its estimate is at most ϕ_{part}-times larger than its corresponding value range counter.

Access frequencies are computed with a different approach depending on whether they are heavy hitters or not. We start calculating the access frequency of a value by checking the stream-summary if the corresponding value range contains heavy hitters. If this is the case, we subtract their accumulated access count from the value range counter. As heavy hitters can be slightly overestimated, we ensure non-negative values after subtraction. Then, the estimated access frequency of values from the long tail is given by the remaining non-negative block count divided by the number of values from the long tail in the value range. The estimated access frequency of heavy hitters is simply taken from the stream-summary.

Our approach can be tuned to fulfill the space requirement by configuring the block size and the number of heavy hitter candidates tracked by the stream-summary. We show in Section 3.4 that our approach

also achieves high precision while having a low runtime overhead. The presented data structure is summarized in the following:

Access Counter 4 (Table Partitioning Advisor)**.**

BLOCK COUNTING: *For each attribute A_{ij}, we create counters $X^{dom}_{ij0}, \ldots, X^{dom}_{ijb}, \ldots, X^{dom}_{ij(\lfloor d_{ij}/b_{ij} \rfloor)}$, where the block size b_{ij} is the number of values grouped into a block.*

STREAM-SUMMARY: *For each attribute A_{ij}, we create a stream-summary data structure SS^h_{ij} such that $\mathrm{dom}(SS^h_{ij})$ is the domain of the monitored top-h most frequently accessed values. For a value $v_{ijy} \in \mathrm{dom}(A_{ij})$, the estimated access frequency is given by $SS^h_{ij}(v_{ijy})$ if $v_{ijy} \in \mathrm{dom}(SS^h_{ij})$, otherwise 0.*

PHYSICAL ACCESSES: *We consider the physical data accesses during execution of workload $\mathcal{W}$. For a sequential access to A_{ij}, X^{dom}_{ijb} is incremented by the number of values that fall into the given block and have at least one matching tuple. The values are also inserted into SS^h_{ij}. For a random access $R_i[\texttt{gid}_i].A_{ij} = v_{ijy}$, the block counter $X^{dom}_{ij\lceil y/b_{ij} \rceil}$ is incremented by 1, and the value is inserted into SS^h_{ij}.*

ACCESS FREQUENCY: *The estimated access frequency $\widehat{f}(v_{ijy})$ of an attribute domain value $v_{ijy} \in \mathrm{dom}(A_{ij})$ is calculated as follows:*

$$\widehat{f}(v_{ijy}) = \begin{cases} SS^h_{ij}(v_{ijy}) & \text{if } isHH(v_{ijy}) \\ \left\lceil \dfrac{\max\left(0, X^{dom}_{ij\lfloor y/b_{ij} \rfloor} - numHHAcc\right)}{(b_{ij} - numHH)} \right\rceil & \text{otherwise,} \end{cases}$$

where

$$isHH(v_{ijy}) = \begin{cases} 1 \;\; \text{if } v_{ijy} \in \mathrm{dom}(SS^h_{ij}) \\ \quad \wedge\, SS^h_{ij}(v_{ijy}) \leq \phi_{part} \cdot X^{dom}_{ij\lfloor y/b_{ij} \rfloor} \\ 0 \;\; \text{otherwise} \end{cases}$$

$$numHH = \sum_{y'=\lfloor y/b_{ij} \rfloor \cdot b_{ij}}^{\lceil y/b_{ij} \rceil \cdot b_{ij} - 1} isHH(v_{ijy'})$$

$$numHHAcc = \sum_{y'=\lfloor y/b_{ij} \rfloor \cdot b_{ij}}^{\lceil y/b_{ij} \rceil \cdot b_{ij} - 1} isHH(v_{ijy'}) \cdot SS^h_{ij}(v_{ijy'}).$$

Figure 3.7 shows how Access Counter 4 collects workload execution statistics $FStat_{\text{part}}$ for a table partitioning advisor during a selection and a join. We illustrate the block counters and the stream summary data structure for attribute `o_orderkey` (A_{11}). The selection $\sigma_{\texttt{o\_orderkey>30}}(\texttt{orders})$ increments counter X^{dom}_{111} by 1 since only value 32 satisfies the predicate, while all following counters are incremented by the block size $b_{11} = 4$. The join $\bowtie_{\texttt{o\_orderkey=l\_orderkey}}(e)$ after some expression e increments counter X^{dom}_{111} by 1 and counter X^{dom}_{112} by 4 because in total four tuples with value 35 were accessed. Besides, all accesses are inserted into the stream-summary SS^2_{11} with $h = 2$.

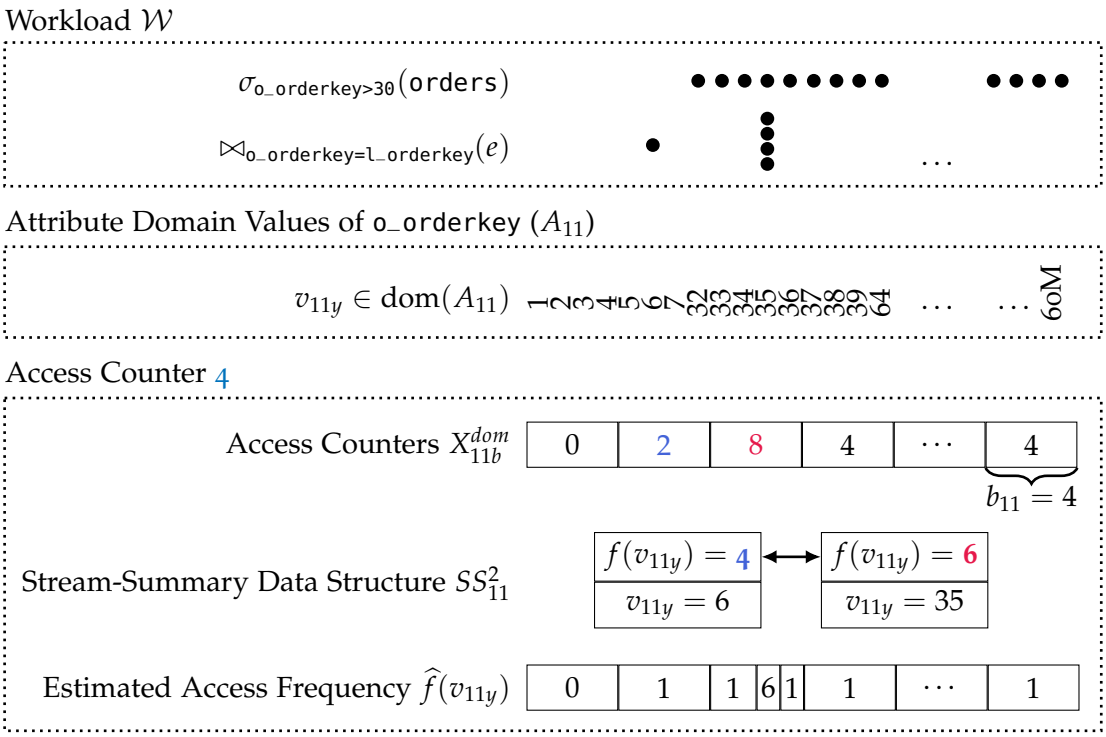


Figure 3.7: Illustration of how Access Counter 4 collects workload execution statistics $FStat_{\text{part}}$ for o_orderkey (A_{11}). A set of block counters tracks accesses to value ranges and a stream-summary monitors the top-h most frequently accessed values.

To compute the access frequencies, both the block counter and the stream-summary data structure are considered. For example, the value 35 is stored in SS^2_{11} as a heavy hitter because 6 is not larger than $\phi_{part} \cdot X^{dom}_{112}$ with $\phi_{part} = 1.2$. Therefore, the counter X^{dom}_{112} is decremented by 6, which results in an estimated access frequency of 1 for the values from the long tail, i.e., 33, 34, and 36. In contrast, value 6 is not classified as a heavy hitter because the estimated access frequency 4 is more than ϕ_{part}-times larger than X^{dom}_{111} with $\phi_{part} = 1.2$.

3.4 EXPERIMENTAL EVALUATION

The evaluation of the presented access counters on their precision, space efficiency, and runtime overhead is the final contribution of this chapter. We implemented our access counters prototypically into SAP HANA (cf. Section 2.3). The experimental setup is discussed in Section 3.4.1. We then evaluate the access counter for an index advisor (cf. Section 3.4.2), a data compression advisor (cf. Section 3.4.3), a buffer pool size advisor (cf. Section 3.4.4), and a table partitioning advisor (cf. Section 3.4.5), using real-world and synthetic benchmarks.

S_q	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33
TPC-H	8	11	11	5	14	16	9	5	8	11	5	11	10	7	12	4	8	8	10	9	9	9											
JCC-H	11	15	7	14	19	9	9	12	–	7	14	10	12	8	9	–	9	14	10	–	–	11											
JOB	5	6	4	4	7	11	4	5	5	3	6	2	9	4	6	11	12	16	17	7	4	9	5	5	5	7	3	8	5	3	5	7	10

Table 3.3: Illustration of the number of occurrences of each SQL statement S_q in each of the three considered workloads, i.e., TPC-H, JCC-H, and Join Order Benchmark.

3.4.1 *Experimental Setup*

Our test system is equipped with an Intel Xeon E7-8870 v4 CPU (4 sockets) and 1 TB DRAM. Secondary storage is provided by a RAID controller of 8 disks of type HGST HUC101812CSS204 HDD with 10k rpm and a SAS 12 Gbit/s interface.

The first workload is the synthetic TPC-H benchmark [159] with scale factor 10, consisting of 22 templated SQL statements. To create a challenging environment for our access counters, we consider as a second workload the JCC-H benchmark [22] (scale factor 10), which extends TPC-H with data and parameter skew. Special shopping events like Black Friday are reflected by spikes in `o_orderdate`. In order to cover the experiments in an acceptable time, we excluded SQL statements S_9, S_{16}, S_{20}, and S_{21} for JCC-H as parameter combinations led to execution times larger than five minutes due to the data and parameter skew. Our third workload is the Join Order Benchmark (JOB) [98]. JOB consists of 33 different SQL statement templates (113 different SQL statements in total) and uses real-world data from IMDb with data skew and correlations that aggravate estimation errors.

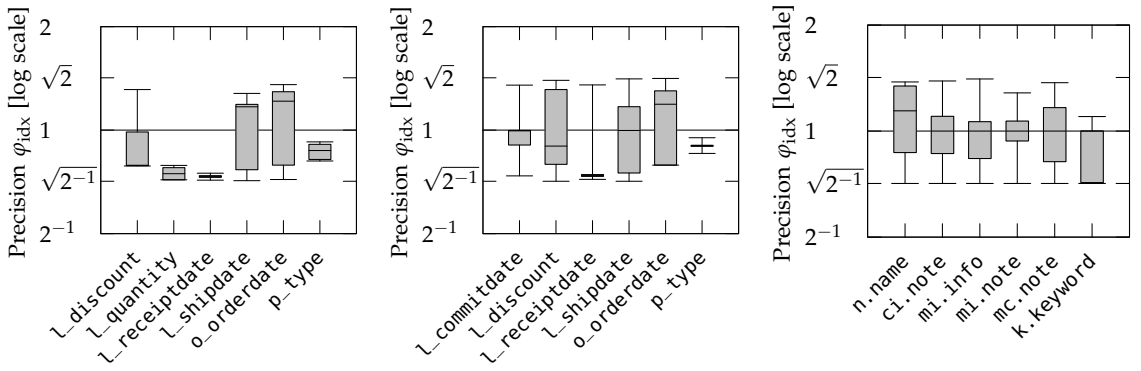
For the evaluation, we randomly generated a workload of 200 statement instantiations for each benchmark. Table 3.3 shows how often each templated SQL statement occurs in each workload. For example, SQL statement S_3 of the JCC-H benchmark is executed seven times with seven different parameter combinations.

3.4.2 *Use Case 1: Index Advisor*

We start by evaluating Access Counter 1 for collecting workload execution statistics $FStat_{\text{idx}}$ for an index advisor. Since we group actual output cardinalities into intervals $[b^r, b^{r+1})$ and count only the number of selections per interval, we calculate the precision φ_{idx} of our approach by dividing the estimated output cardinality $|\widehat{\sigma_p(R_i)}| = \sqrt{b^r \cdot b^{r+1}}$ by the actual output cardinality $|\sigma_p(R_i)|$:

$$\varphi_{\text{idx}} = |\widehat{\sigma_p(R_i)}| / |\sigma_p(R_i)|.$$

We set the interval base parameter b to 2, such that the actual and estimated output cardinalities differ at most by a factor of $\sqrt{2}$.



(a) TPC-H (SF 10) (b) JCC-H (SF 10) (c) JOB

Figure 3.8: Precision φ_{idx}, i.e., the ratio of estimated and actual output cardinalities of selections on base relations, for collecting workload execution statistics $FStat_{\text{idx}}$ for an index advisor using our Access Counter 1 with interval base parameter $b = 2$.

Figure 3.8 shows the precision φ_{idx} for six sets of attributes $\mathbb{A}_{is} \subseteq \mathcal{F}(p), \forall \sigma_p(R_i) \in T(t, S_q, V_q), (t, S_q, V_q) \in \mathcal{W}$, executing the TPC-H, JCC-H, and Join Order Benchmark. Overestimation is shown on the top, underestimation at the bottom. Each boxplot shows the 0.00, 0.25, 0.50, 0.75, and 1.00 percentiles. Although the precision φ_{idx} is at most $\sqrt{2}$ in accordance with our choice of $b = 2$, we observe that estimated output cardinalities for selections in TPC-H are underestimated, e.g., selections with `l_quantity` and `l_receiptdate` as an attribute referenced in the predicate. However, this is not a conceptual problem. In those cases, the root cause for underestimation is that assigned parameter values to host variables that reference `l_quantity` or `l_receiptdate` do not differ substantially. As a result, the collected output cardinalities for those selections are very similar. For example, the host variable that references `l_quantity` in SQL statement S_6 is either assigned with 24 or 25, according to the TPC-H specification [159], whereas selections with `l_quantity` referenced in statements S_{17}, S_{18}, S_{19}, and S_{20} are not executed on base relations. Similar behavior is observable for `l_receiptdate`. In statement S_{12}, two host variables reference `l_receiptdate`. The first host variable is either assigned with 1992-01-01, 1993-01-01, 1994-01-01, 1995-01-01, 1996-01-01, or 1997-01-01, while the second host variable is assigned with a date that is one year later than the date of the first host variable. As data in `lineitem` in TPC-H are not heavily skewed, the output cardinality for all those selections is very similar in each case as precisely one year is selected. Selections with `l_quantity` referenced in SQL statements S_4 and S_{21} are not executed on base relations, e.g., as a post-filter in S_{21}.

Table 3.4 shows the results with respect to precision, space efficiency, and runtime overhead of precise counting (i.e., one counter per output cardinality) and our Access Counter 1 (i.e., lazy counters and interval

	Precise Counting			Our Access Counter 1		
Workload	TPC-H	JCC-H	JOB	TPC-H	JCC-H	JOB
Precision φ_{idx}	1.0	1.0	1.0	$\leq \sqrt{2}$	$\leq \sqrt{2}$	$\leq \sqrt{2}$
Memory Overhead	10.6%	10.6%	8.4%	$< 0.1\%$	$< 0.1\%$	$< 0.1\%$
Runtime Overhead	1.4%	1.5%	1.6%	1.7%	2.6%	3.1%

Table 3.4: Precision, space efficiency, and runtime overhead for collecting workload execution statistics $FStat_{\text{idx}}$ for an index advisor using either our Access Counter 1 with interval base parameter $b = 2$ or precise counting (i.e., one counter per output cardinality).

counting). Precise counting trades off precision for memory overhead. Indeed, its memory overhead varies between 8.4% and 10.6%, which is substantial. Our approach instead still attains reasonably accurate estimates, differing at most by a factor of $\sqrt{2}$. The memory overhead is also negligible due to lazy counting combined with intervals. Both approaches yield a low runtime overhead since only the actual output cardinalities of selections are tracked. We conclude that our access counters are precise, compact, and fast.

3.4.3 *Use Case 2: Data Compression Advisor*

We now evaluate our Access Counter 2 to collect workload execution statistics $FStat_{\text{comp}}$ for a data compression advisor. Table 3.5 shows the results, indicating precision, space efficiency, and runtime overhead. Our approach is 100% precise because the exact number of tuples accessed sequentially and randomly by the workload is counted for each column partition. Maintaining just two 64-bit integer counters per column partition is also space-efficient. For example, for the Join Order Benchmark with 108 attributes in 21 relations and without table partitioning, the total memory consumption is only 1.73 KB ($= 108 \cdot 16$ Bytes). This represents a mere 0.00008% of the storage size in SAP HANA (2.28 GB). As the runtime overhead is also low (between 4.7% and 9.1%), we conclude that our access counters for a data compression advisor are precise, compact, and fast.

Workload	TPC-H	JCC-H	JOB
Precision	100% precise		
Memory Overhead	$< 0.1\%$	$< 0.1\%$	$< 0.1\%$
Runtime Overhead	4.7%	8.3%	9.1%

Table 3.5: Precision, space efficiency, and runtime overhead on workload execution statistics $FStat_{\text{comp}}$ for a data compression advisor using our Access Counter 2.

	Naïve Page Access Counters			Our Access Counter 3		
Workload	TPC-H	JCC-H	JOB	TPC-H	JCC-H	JOB
Precision	100% precise			100% precise		
Memory Overhead	$\leq 0.2\%$	$\leq 0.2\%$	$\leq 0.2\%$	$\leq 0.2\%$	$\leq 0.2\%$	$\leq 0.2\%$
Runtime Overhead	8.3%	13.1%	21.8%	5.2%	9.2%	13.5%

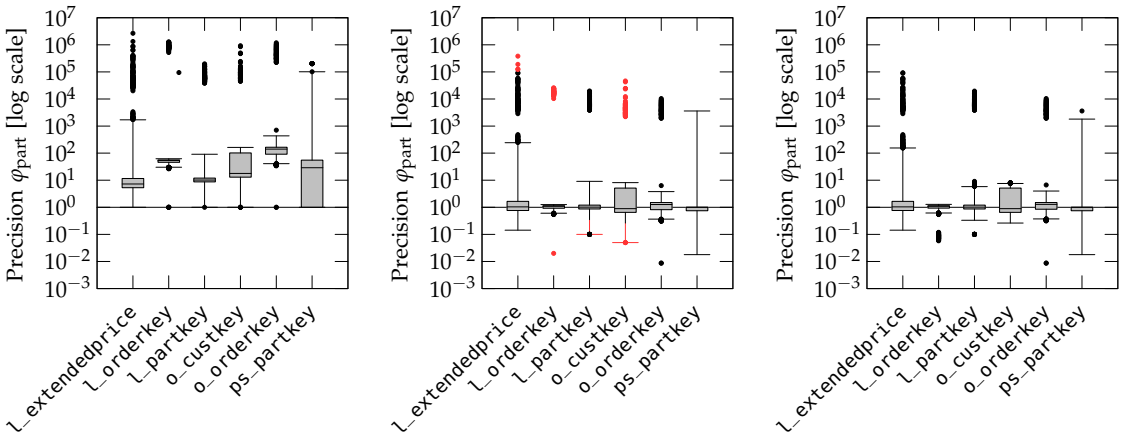
Table 3.6: Precision, space efficiency, and runtime overhead for collecting workload execution statistics $FStat_{\text{buf}}$ for a buffer pool size advisor using either our Access Counter 3 or naïve page access counters (i.e., updating the frequencies of all touched pages).

3.4.4 *Use Case 3: Buffer Pool Size Advisor*

In the third experiment, we evaluate our Access Counter 3 for collecting workload execution statistics $FStat_{\text{buf}}$ for a buffer pool size advisor. Table 3.6 shows the results with respect to the precision, space efficiency, and runtime overhead of naïve page counters (i.e., updating the frequencies of all touched pages) and our approach (i.e., updating only the frequencies of start and end pages). Both approaches are precise because all physical accesses to every page is tracked. We use one signed 64-bit integer counter per disk page as counters may become negative. As a result, the memory overhead is at most 0.2%, given the smallest page size of 4 KB in SAP HANA (cf. Section 2.3.3). The runtime overhead of naïve page counters varies between 8.3% and 21.8%. Our approach has a runtime overhead between 5.2% and 13.5%. This is because multi-page accesses require a constant number of updates to the counters, regardless of the number of pages they span. We conclude that our access counters for a buffer pool size advisor are precise, compact, and fast.

3.4.5 *Use Case 4: Table Partitioning Advisor*

Finally, we evaluate our Access Counter 4 for collecting workload execution statistics $FStat_{\text{part}}$ for a table partitioning advisor. To fulfill the space efficiency requirement, we limit the memory footprint of our access counters to 1% compared to the storage size. For example, for `o_orderdate` (23 MB, 2406 distinct values), we create one counter per domain value, while for `o_orderkey` (105 MB, 15,000,000 distinct values), domain values are grouped into ranges of 115 values each. We also maintain a stream-summary data structure for attributes with a block size larger than one to track the top-100 most frequently accessed values. Finally, we set $\phi_{part} = 1.2$, i.e., a value is classified as a heavy hitter if its access frequency estimated by the stream-summary data structure is at most $1.2\times$ larger than its value range counter. We experimentally determined $\phi_{part} = 1.2$ to be a good choice. In



(a) Naïve Block-Level Counters (b) Access Counter 4 (without SS^h_{ij}) (c) Access Counter 4 (with SS^h_{ij})

Figure 3.9: Precision for collecting workload execution statistics $FStat_{\text{part}}$ for a table partitioning advisor using either our Access Counter 4 (with and without the stream-summary data structure) or naïve block-level access counters (i.e., the block counter is incremented by one whenever a value or sub-range of the block is accessed), executing JCC-H.

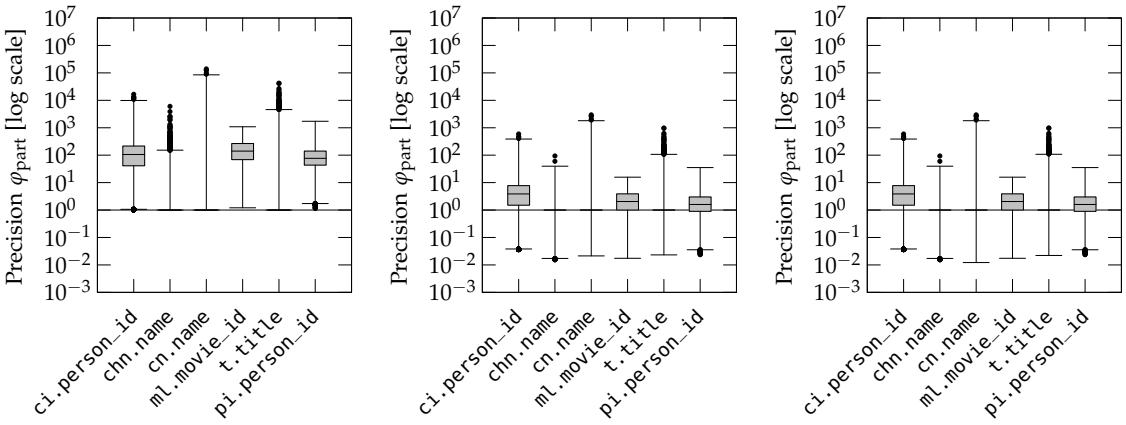
order to evaluate the quality of the estimated access frequency $\widehat{f}(v_{ijy})$ obtained by our access counter for a value $v_{ijy} \in \text{dom}(A_{ij})$, we define a precision metric φ_{part}:

$$\varphi_{\text{part}} = \widehat{f}(v_{ijy}) / f(v_{ijy}).$$

In the JCC-H benchmark, 29 of 61 attributes yield a block size larger than one, i.e., they cannot grant 100% precision within a memory budget of 1% of the storage size of a column partition. Figure 3.9 shows the precision φ_{part} of six representative attributes with a block size larger than one for our Access Counter 4 (with and without the stream-summary data structure) and naïve block-level access counters, executing the JCC-H benchmark. Overestimation is shown at the top, underestimation at the bottom. The boxplot displays the 0.0001, 0.25, 0.50, 0.75, and 0.9999 percentiles. Outliers are highlighted as dots above or below the boxplot.

Figure 3.9a shows the precision of naïve block-level counters, i.e., the block counter is incremented by one whenever a value or sub-range of the block is accessed. The results confirm our claims in Section 3.3.4, stating that access frequencies are overestimated substantially.

Figure 3.9b shows the precision of our approach counting the number of actually accessed block values, while the access frequency of a value is obtained by dividing the total number of accessed values by the block size. We observe that our approach improves precision by several orders of magnitude. In addition, most of the estimates are bounded by a factor of 2. However, heavy hitters are underestimated



(a) Naïve Block-Level Counters (b) Access Counter 4 (without SS^h_{ij}) (c) Access Counter 4 (with SS^h_{ij})

Figure 3.10: Precision for collecting workload execution statistics $FStat_{\text{part}}$ for a table partitioning advisor using either our Access Counter 4 (with and without the stream-summary data structure) or naïve block-level access counters, executing the Join Order Benchmark.

for all six attributes, and rarely accessed values are overestimated (shown on the top and bottom of Figure 3.9b).

Figure 3.9c shows the precision obtained by adding the stream-summary data structure to identify heavy hitters. In order to highlight the benefit of the stream-summary data structure, we mark the over- and underestimated values in Figure 3.9b that are correctly estimated in Figure 3.9c in red. For example, the heavy hitters of `l_orderkey` (shown in red at the bottom in Figure 3.9b) are estimated precisely in Figure 3.9c. Accordingly, rarely accessed values of the corresponding block are overestimated without the stream-summary (shown at the top of Figure 3.9b) but estimated precisely with the stream-summary. We observe similar results for `o_custkey`, `l_partkey`, and `l_extendedprice`.

The TPC-H benchmark's precision is very similar to the JCC-H benchmark without the heavy hitters of Figure 3.9.

In the Join Order Benchmark, 47 of 108 attributes yield a block size larger than one. Figure 3.10 shows the precision φ_{part} for six representative attributes. We again observe that naïve block-level counters overestimate access frequencies substantially (cf. Figure 3.10a), while our approach improves the precision by 1-2 orders of magnitude (cf. Figure 3.10b). However, we do not observe substantial improvement by adding the stream-summary data structure like for the JCC-H benchmark (cf. Figure 3.10c). The reason is that the JCC-H benchmark exhibits heavy hitters by design, while the Join Order Benchmark exposes limited data and parameter skew. Nevertheless, we argue that our approach improves precision by orders of magnitudes compared to naïve block-level access counters.

	Value-Level Access Counters			Block-Level Counters & Access Counter 4 (without SS^h_{ij})			Access Counter 4 (with SS^h_{ij})		
Workload	TPC-H	JCC-H	JOB	TPC-H	JCC-H	JOB	TPC-H	JCC-H	JOB
Memory Overhead	10.80%	10.82%	20.53%	$\leq 1\%$	$\leq 1\%$	$\leq 1\%$	$\leq 1\%$	$\leq 1\%$	$\leq 1\%$
Runtime Overhead	3.9%	14.7%	15.6%	2.1%	9.7%	9.6%	13.8%	22.7%	23.6%

Table 3.7: Space efficiency and runtime overhead for collecting workload execution statistics $FStat_{\text{part}}$ for a table partitioning advisor using either Access Counter 4 (with and without the stream summary data structure), naïve block-level access counters, or value-level access counter (i.e., one counter per value), executing TPC-H, JCC-H, and JOB.

Table 3.7 shows the space efficiency and runtime overhead of value-level access counters (i.e., one counter per value), naïve block-level access counters, and our Access Counter 4, with and without the stream summary data structure. While value-level data access counters are precise, they have high memory and runtime overheads. By contrast, naïve block-level access counters and our approach (without adding the stream-summary data structure) use a fixed memory budget of 1% and achieve low runtime overhead. However, naïve block-level access counters are imprecise, while our approach achieves precise estimates, as demonstrated in Figures 3.9 and 3.10. Furthermore, adding the stream-summary data structure improves the precision (cf. Figure 3.9) at the cost of increasing the runtime overhead. Therefore, our approach (without the stream-summary data structure) is preferred for workloads with critical runtime overhead. The stream-summary data structure may be added to improve the precision by trading off the runtime overhead on currently executed workloads.

3.5 RELATED WORK

We discuss related approaches in the literature, which collect workload execution statistics, too. Our findings are also summarized in Table 3.8.

In order to feed a physical database design advisor with accurate workload execution statistics, previous approaches propose to collect physical accesses per tuple or value [67, 102]. A similar approach is a graph representation of the workload, where each tuple is represented as a node. An edge connect tuples accessed within the same transaction [38, 149]. These approaches yield precise workload execution statistics because each physical access to the data is tracked. However, in Table 3.7, we show that they result in high memory consumption and in a critical runtime overhead due to a more fine-granular access tracking. Therefore, these approaches are unable to collect workload execution statistics precisely, compact, and fast.

Approach for Collecting Workload Execution Statistics	Precise				Compact	Fast
	$FStat_{idx}$	$FStat_{comp}$	$FStat_{buf}$	$FStat_{part}$		
Access counters per tuple/value [67, 102]	✗	✓	✓	✓	✗	✗
Graph representation [38, 149]	✗	✓	✓	✓	●	●
Block-level access counters [56, 76]	✗	✓	✓	●	✓	●
Memory access tracing [122]	✗	✗	✓	✓	✗	✗
SQL statements + What-if API [4, 141]	●	●	●	●	✓	✗
Our approach	✓	✓	✓	✓	✓	✓

Table 3.8: Comparison between different approaches for collecting workload execution statistics $FStat_{idx}$, $FStat_{comp}$, $FStat_{buf}$, and $FStat_{part}$ as input to physical database design advisors with respect to their precision, space efficiency, and runtime overhead.

Instead of tracking accesses fine-granular, block-level access counters were proposed to reduce the memory consumption [56, 76]. However, in Figures 3.9 and 3.10, we show that block-level access counters fail to provide accurate workload execution statistics for a table partitioning advisor in the presence of heavy hitters, e.g., due to events like Black Friday. Furthermore, the runtime overhead of block-level access counters can be critical, e.g., when collecting workload execution statistics for a buffer pool size advisor, as shown in Table 3.6. In the worst-case scenario, i.e., in the presence of a full-column scan, all counters of all accessed disk pages need to be updated.

Contrary to a collection of workload execution statistics inside the database management system, memory access tracing [122] uses the PEBS mechanism of Intel processors to trace memory accesses, which are mapped to the data to determine precise access frequencies of pages and values of the active attribute domain. However, while only single memory accesses are traced, the access granularity and access type cannot be identified. Moreover, since memory traces are logged and analyzed offline, the memory and runtime overhead is high. Hence, this approach is also unable to collect precise and compact workload execution statistics quickly.

All aforementioned approaches collect physical data accesses. In contrast, related approaches to collect workload execution statistics for an index advisor rely on estimates of the query optimizer's what-if API [4, 141]. In particular, they feed instantiations of SQL statements into the what-if API and evaluate how the response time of the statements would change if the index configuration is adopted. However, accurate workload execution statistics cannot be guaranteed as this approach relies on the availability of precise cardinality estimates.

3.6 DISCUSSION

In this chapter, we presented low-level data access counters to four physical database design advisors. Our low-level data access counters collect workload execution statistics with high precision, low memory consumption, and low runtime overhead to address the first challenge raised in this dissertation (cf. Section 1.3.1). Since the workload is skewed over the domain, existing approaches that collect workload execution statistics for a table partitioning advisor with low memory and runtime overhead, e.g., block-level data access counters, lack precision due to the presence of heavy hitters during events like Black Friday. The presented low-level data access counters improve precision in these situations. This is because we employ the space-saving algorithm combined with a stream-summary data structure to cope with heavy hitters and combine the counted accesses from the blocks with the heavy hitters found by the stream-summary data structure. Figures 3.9 and 3.10 demonstrate that our approach achieves precise estimates while still having a low memory and runtime overhead, as demonstrated in Table 3.7. As introduced in the next chapter, our table partitioning advisor utilizes the collected workload execution statistics and proposes a range partitioning layout.

4

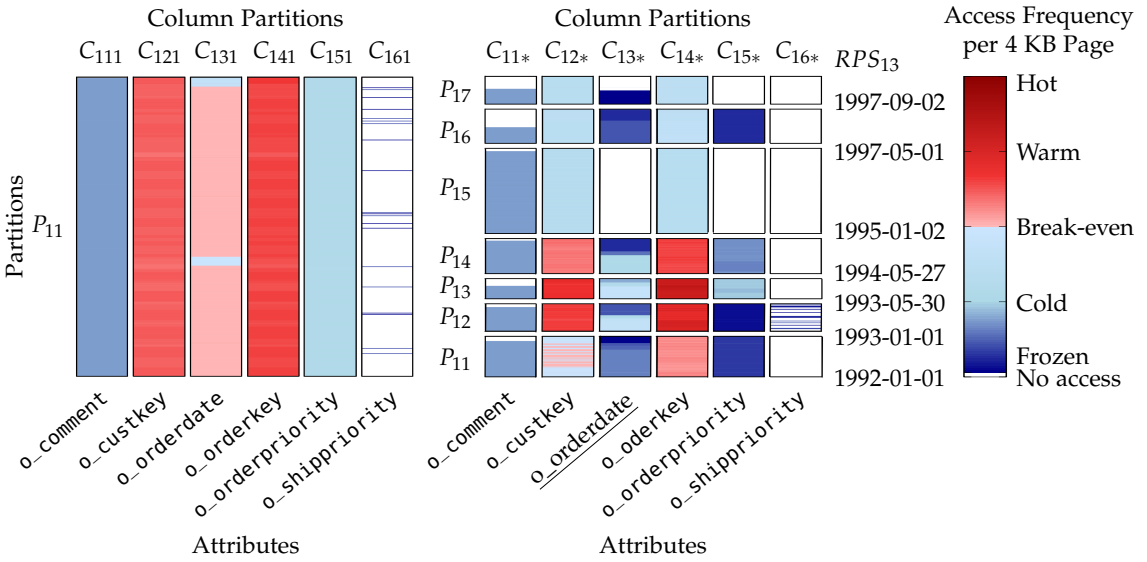
MEMORY FOOTPRINT REDUCTION WITH TABLE PARTITIONING

In the previous chapter, we demonstrated how workload execution statistics could be collected with high precision, low memory consumption, and low runtime overhead. In this chapter, we utilize the collected workload execution statistics for a table partitioning advisor that addresses the second challenge pointed out in this dissertation (cf. Section 1.3.2). The goal is to recommend a range partitioning layout for each relation in order to minimize the buffer pool size of the database management system while still adhering to performance commitments between database-as-a-service providers and customers.

4.1 MOTIVATION

Since DRAM constitutes the primary driver of hardware costs [60, 106], database-as-a-service providers face the challenge of meeting performance commitments assured in SLAs while keeping the memory footprint of hosted database instances as small as possible to lower their internal costs and enhance profitability. A way to decrease the memory footprint while still adhering to SLAs is to employ a buffer pool size advisor (cf. Section 2.2.3) and utilize the workload's access skew by evicting cold disk pages to cheaper storage layers and keeping only hot disk pages in DRAM. Although this approach is simple, its most significant drawback is that mixing hot and cold data within the same page pollutes the buffer cache with cold data and works against its effectiveness. This is because the physical schema is often not defined according to the data access pattern. As a consequence, cold data can be on the same disk page as hot data. Therefore, no substantial reductions in memory footprint are achievable because all hot disk pages (including their cold data) must be kept in DRAM. The reason is that reducing the buffer pool size may violate SLAs since disk pages with hot data would be evicted to secondary storage.

As discussed in Section 2.2.4, a sharper separation of hot and cold data can be achieved by proposing a range partitioning layout for each relation. This is because a typical workload access pattern can be observed in which rows are either frequently or rarely accessed according to a value range of a specific column of that table, as we will see later. Therefore, rows that correspond to hot-classified value ranges into hot range partitions that stay in DRAM, whereas rows that belong to cold-classified value ranges are grouped into cold range partitions that can be evicted to cheaper storage layers and are loaded



(a) Non-partitioned `orders` (R_1). (b) A table partitioning layout $\mathcal{T}_1$ for `orders` (R_1) as chosen by our approach, generated by the range partitioning specification RPS_{13} with `o_orderdate` (A_{13}) as the partition-driving attribute.

Figure 4.1: Example of page accesses to six attributes of `orders` (R_1) for a non-partitioned and a range-partitioned layout. After executing 200 statements of the JCC-H benchmark [22], the pages are classified as hot or cold according to the π-second rule (cf. Section 2.4). The range-partitioned layout proposed by our approach consists of fewer hot pages.

only on demand. As a result, only hot range partitions with a high density of hot data stay in DRAM. This prevents the pollution of the buffer pool with cold data.

In order to demonstrate the benefits of a table partitioning advisor that reduces the number of hot-classified pages compared to a non-partitioned layout, let us consider Figure 4.1. After executing 200 SQL statements of the JCC-H benchmark [22] with scale factor 10, we show the access frequency of each disk page for two partitioning layouts for six attributes of `orders` (R_1). Figure 4.1a shows the non-partitioned layout, while Figure 4.1b depicts a table partitioning layout as chosen by our approach, generated by the range partitioning specification RPS_{13} (cf. Definition 5) with `o_orderdate` (A_{13}) as the partition-driving attribute (cf. Definition 4). Each column partition consists of multiple 4 KB disk pages (cf. Section 2.1.3), i.e., each horizontal line represents a single 4 KB disk page. To each disk page, we count the number of page accesses of all operators by the workload with our Access Counter 3 (cf. Section 3.3.3). To draw a reasonable border between frequently (hot) and rarely (cold) accessed pages, we consider the π-second rule to classify pages as hot or cold (cf. Section 2.4). Hot pages are shown in red, whereas cold pages are white (no access) or blue (at least one access). We observe that the range-partitioned layout proposed by our approach consists of fewer hot pages than the non-partitioned

layout. In particular, only a subset of `o_custkey`, `o_orderdate`, and `o_orderkey` is frequently accessed in the range-partitioned layout compared to the non-partitioned layout. Our approach also identified nine column partitions (boxes in white) that are not accessed at all due to partition pruning. For example, only a single column partition of `o_shippriority` is accessed in the range-partitioned layout. In contrast, all accesses to `o_shippriority` are distributed over the entire attribute in the non-partitioned layout. In summary, the buffer pool size can be reduced with the range-partitioned layout compared to the non-partitioned layout when only the hot-classified pages are kept in DRAM during the entire workload.

While Figure 4.1 demonstrates how a range partitioning specification can reduce the buffer pool size, existing table partitioning advisors [4, 5, 38, 73, 119, 133, 138, 141, 149, 173, 174] focus solely on the classical database objective of maximizing performance and do not consider the aspect of minimizing memory footprint. In particular, Schism [38], Clay [149], Horticulture [133], Mesa [119], Hilprecht et al. [73], and Strife [138] do the exact opposite of what our approach intends to achieve: To balance the load on partitions, they distribute accesses evenly across all partitions and, therefore, generate disk pages with mixed temperatures (i.e., pages with hot and cold data). This pollutes the buffer pool with cold data because all disk pages with hot data (including cold data on these pages) are required to be held in DRAM to maximize performance.

Furthermore, existing table partitioning advisors are mainly designed for row stores [4, 5, 38, 73, 119, 133, 138, 141, 149, 173, 174]. As we focus on column stores in this dissertation, we present instead a table partitioning advisor that is mainly designed for column stores. In addition, we consider the impact of dictionary compression and bit-packing on the memory footprint of table partitioning layouts. This aspect is crucial since many column stores use and benefit from dictionary compression and bit-packing (cf. Section 2.1.3).

4.2 PROBLEM STATEMENT

We present a table partitioning advisor that optimizes each relation independently from other relations. Its objective is to decrease buffer pool pollution and reduce data accesses. We consider derived partitioning of multiple relations as future work. Our table partitioning advisor focuses on range partitioning because hash and round-robin partitioning distribute accesses evenly over all partitions and are thus unsuitable for memory footprint reduction [103]. For large (fact) tables, a multi-level partitioning setup might be preferred, such that hash partitioning can be used for scale-out as a first level and range partitioning for memory footprint reduction as a second level.

Problem 2. *The problem we consider is to find for each relation* $R_i \in \mathcal{R}$ *a partition-driving attribute* $A_{i\lambda}$ *and a range partitioning specification* $RPS_{i\lambda} \in \mathbb{RPS}_{i\lambda}$*, such that the buffer pool size* $\mathcal{BS}$ *is minimized while the execution time* $\mathcal{E}$ *of a workload* $\mathcal{W}$ *executed on the physical schema* $\mathcal{L}$ *with the buffer pool size* $\mathcal{BS}$ *does not violate a maximum workload execution time* SLA *(e.g., an agreement on the maximum workload execution time between the customer and the database-as-a-service provider):*

$$\underset{R_i \in \mathcal{R}, A_{i\lambda}, RPS_{i\lambda} \in \mathbb{RPS}_{i\lambda}}{\arg\min} \quad \mathcal{BS} \qquad \text{subject to } \mathcal{E}(\mathcal{W}, \mathcal{L}, \mathcal{BS}) \leq SLA.$$

4.3 SYSTEM MODEL

We propose a solution to Problem 2 by presenting the table partitioning advisor SAHARA: a **s**torage **a**dvisor based on **h**eavy **a**nd **r**are **a**ccesses. SAHARA proposes a range partitioning specification for each relation in order to minimize the buffer pool size while still adhering to all performance commitments. Figure 4.2 shows the system model of SAHARA. We first collect workload execution statistics on the *current* physical schema $\mathcal{L}_{\text{cur}}$ (cf. Section 4.4), which may contain non-partitioned tables if SAHARA was not applied before. We then enumerate physical schema *candidates*, where each physical schema *candidate* is identified by a partition-driving attribute $A_{i\lambda}$ and a range partitioning specification $RPS_{i\lambda} \in \mathbb{RPS}_{i\lambda}$ for each relation R_i. Section 4.5 presents exact and heuristic enumeration algorithms to determine a range partitioning specification. Next, in Section 4.6, the statistics collected on the

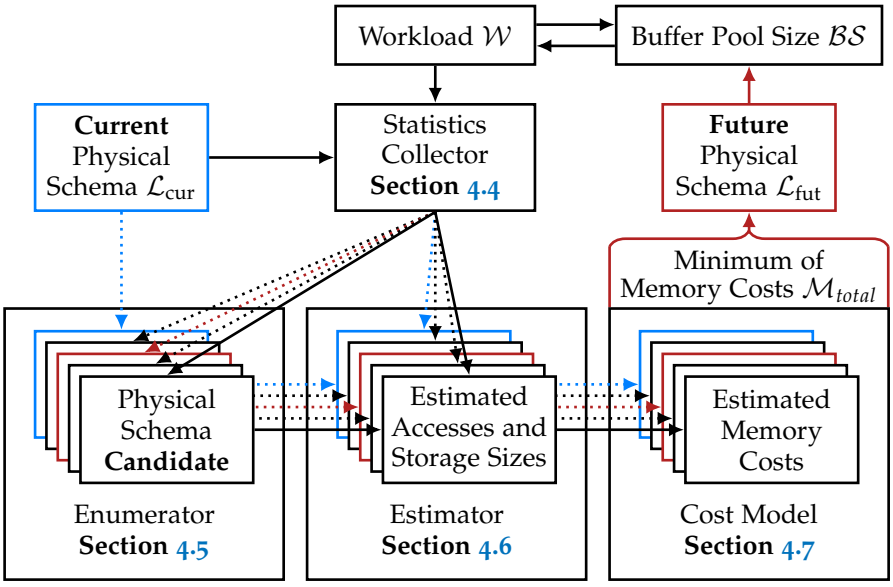


Figure 4.2: The system model of the table partitioning advisor SAHARA consists of four building blocks: a statistics collector, an enumerator for physical schema candidates, an estimator for workload's data accesses and storage sizes, and a cost model for the memory costs.

current physical schema must be transformed into estimates for each physical schema *candidate*. The reason is that a workload's data access may differ for each physical schema *candidate*, e.g., due to partition pruning. In addition, compression ratios can differ due to the number of values replicated into the dictionaries of multiple partitions. Our cost model is then used to calculate the memory costs $\mathcal{M}_{total}$ for each physical schema *candidate* based on the estimates, a given SLA, and the hardware configuration (cf. Section 4.7). A physical schema *candidate* with minimal memory costs $\mathcal{M}_{total}$ is proposed as the future physical schema $\mathcal{L}_{\text{fut}}$. As a consequence, the buffer pool size $\mathcal{BS}$ is adjusted to fulfill the SLA. Finally, our system is evaluated in Section 4.8.

4.4 STATISTICS COLLECTION

In Section 3.3.4, we showed how workload execution statistics for a table partitioning advisor are collected with high precision, low memory consumption, and low runtime overhead. The proposed Access Counter 4 groups domain values into ranges, counts the number of accesses for each value range, and uses a stream-summary data structure to keep track of the most frequently accessed values to cope with heavy hitters. However, the access frequency provided by Access Counter 4 could be dominated by many accesses occurring only during a short period, cached in the buffer pool. As our goal is to propose a physical schema that minimizes the buffer pool size (cf. Problem 2), the distribution of data accesses over time is crucial because it impacts the buffer pool eviction policy (cf. Section 2.3.3). For example, a data item accessed twice within a short period is likely cached in the buffer pool at the second access. Therefore, we modify Access Counter 4 as follows to incorporate the behavior of the buffer pool during statistics collection:

1. For each value range, we use a bitmap instead of a 32-bit integer counter, such that each bit is mapped to a specified time window.
2. The bit is set if at least one value of the value range is accessed during the specified time window.
3. The access frequency of the value range is obtained by the `popcount` of the bitmap.
4. The time window length is set to $\pi/2$ seconds.

There are two reasons why the time window length is set to $\pi/2$: First, according to the π-second rule (cf. Definition 2.4), the time window length should not be substantially smaller than π because two accesses to the same data item within π seconds generally result in at most one page fault. Second, the Nyquist–Shannon sampling theorem proves that a sampling rate of $\pi/2$ is sufficient to achieve precise statistics [150].

The first building block for the statistics collector is to define *a set of time windows* between the smallest and largest timestamp in the workload $\mathcal{W}$ using a time window length of $\pi/2$ seconds.

Definition 25 (Time Windows). *Let $t_{\min}, t_{\max} \in \mathbb{N}$ be two timestamps, such that $\forall (t, S_q, V_q) \in \mathcal{W} : t_{\min} \leq t < t_{\max}$. We define by Ω a set of time windows between $t_{\min}$ and $t_{\max}$ using a time window length of $\pi/2$:*

$$\Omega = \left\{ \left[t_{\min} + \tfrac{\pi}{2} \cdot b, t_{\min} + \tfrac{\pi}{2} \cdot (b+1) \right), b \in \mathbb{N}, t_{\min} + \tfrac{\pi}{2} \cdot b < t_{\max} \right\}.$$

We denote by $\omega_b \in \Omega$ the time window $\left[t_{\min} + \frac{\pi}{2} \cdot b, t_{\min} + \frac{\pi}{2} \cdot (b+1) \right)$.

As a second building block, we define a *workload trace* that contains all accessed data during the execution of workload $\mathcal{W}$.

Definition 26 (Workload Trace). *We define a workload trace Tr as*

$$Tr \subseteq \{ (R_i, \texttt{gid}_i, A_{ij}, (t, S_q, V_q)) \mid 1 \leq i \leq n, 1 \leq \texttt{gid}_i \leq |R_i|, \\ 1 \leq j \leq m_i, (t, S_q, V_q) \in \mathcal{W} \},$$

where each element in Tr denotes an access to attribute A_{ij} of relation R_i for the tuple with the global tuple identifier $\texttt{gid}_i$ during the execution of the statement instantiation (t, S_q, V_q) in the workload $\mathcal{W}$.

As a third building block, we define a *Boolean function* `eval` that evaluates if an attribute domain value $v_{ijy} \in \text{dom}(A_{ij})$ satisfies a predicate in a selection on a base relation in the physical execution plan of a statement instantiation. This is crucial because we assume in Section 4.6 that a range partition generated according to a value range of a partition-driving attribute will be pruned, i.e., not accessed, if the partition-driving attribute is referenced in the selection predicate and the value range does not satisfy the predicate.

Definition 27 (Predicate Evaluation Function). *We define a Boolean function `eval` for an attribute A_{ij} in relation R_i, an attribute domain value $v_{ijy} \in \text{dom}(A_{ij})$, and a statement instantiation $(t, S_q, V_q) \in \mathcal{W}$ as*

$\texttt{eval}(R_i, A_{ij}, v_{ijy}, (t, S_q, V_q)) \quad \Leftrightarrow$ *there exists a selection $\sigma_p(R_i) \in T(t, S_q, V_q), (t, S_q, V_q) \in \mathcal{W}$, where the selection predicate p references attribute A_{ij} and attribute domain value $v_{ijy} \in \text{dom}(A_{ij})$ satisfy p.*

This leads us to the definition of *domain block counters* that collect accesses to value ranges and are maintained per attribute A_{ij} in each relation R_i. The domain block counters will be used in Section 4.5 to enumerate range partitioning specifications. For instance, we present a heuristic approach that groups consecutive value ranges with a similar access pattern to merge hot rows into a hot range partition.

Definition 28 (Domain Block Counters). *For each attribute A_{ij} in relation R_i, we create domain block counters $X^{dom}_{ij0}, \ldots, X^{dom}_{ijz}, \ldots, X^{dom}_{ij(\lfloor d_{ij}/DBS_{ij} \rfloor)}$, where the domain block size DBS_{ij} is the number of consecutive values in the active domain grouped into a block. Each counter X^{dom}_{ijz} consists of a bitmap with $|\Omega|$ bits, where bit $b \in [0, |\Omega|)$ is set as follows:*

$$X^{dom}_{ijz}[b] := \begin{cases} 1 & \exists\, \texttt{gid}_i \in [1, |R_i|], v_{ijy} \in \text{dom}(A_{ij}), (t, S_q, V_q) \in \mathcal{W} : \\ & \quad (R_i, \texttt{gid}_i, A_{ij}, (t, S_q, V_q)) \in Tr \\ & \quad \wedge\, \texttt{eval}(R_i, A_{ij}, v_{ijy}, (t, S_q, V_q)) \\ & \quad \wedge\, R_i[\texttt{gid}_i].A_{ij} = v_{ijy} \\ & \quad \wedge\, \lfloor y / DBS_{ij} \rfloor = z \\ & \quad \wedge\, t \in \omega_b \\ 0 & \textit{otherwise.} \end{cases}$$

The b-th bit of counter X^{dom}_{ijz} is set if the workload trace Tr contains at least one element that accesses attribute A_{ij} within time window ω_b for a tuple with attribute domain value v_{ijy} that falls into the z-th domain block and satisfies the predicate evaluation function `eval`.

Note that domain block counters are created per attribute and not per column partition. The reason is that mapping domain block accesses per column partition to the active attribute domain would deteriorate statistics precision due to missing values between the active domains of the column partitions. In addition, we avoid recording duplicates between different column partitions.

In addition to domain block counters, we maintain *tuple block counters* to capture accesses to each tuple in a column partition C_{ijk} in each relation R_i. Tuple block counters will be used in Section 4.6 to estimate access frequencies of column partitions enumerated by range partitioning specifications of Section 4.5. These estimates are required to calculate the memory costs $\mathcal{M}_{total}$ for physical schema candidates (cf. Section 4.7) and to compute an optimal range partitioning specification from it recursively.

Definition 29 (Tuple Block Counters). *For each column partition C_{ijk} in relation R_i, we create tuple block counters $X^{tuple}_{ijk0}, \ldots, X^{tuple}_{ijkz}, \ldots, X^{tuple}_{ijk(\lfloor |P_{ik}|/TBS_{ijk} \rfloor)}$, where the tuple block size TBS_{ijk} is the number of consecutive local tuple identifiers in partition P_{ik} grouped into a block. Each counter X^{tuple}_{ijkz} consists of a bitmap with $|\Omega|$ bits, where bit $b \in [0, |\Omega|)$ is set as follows:*

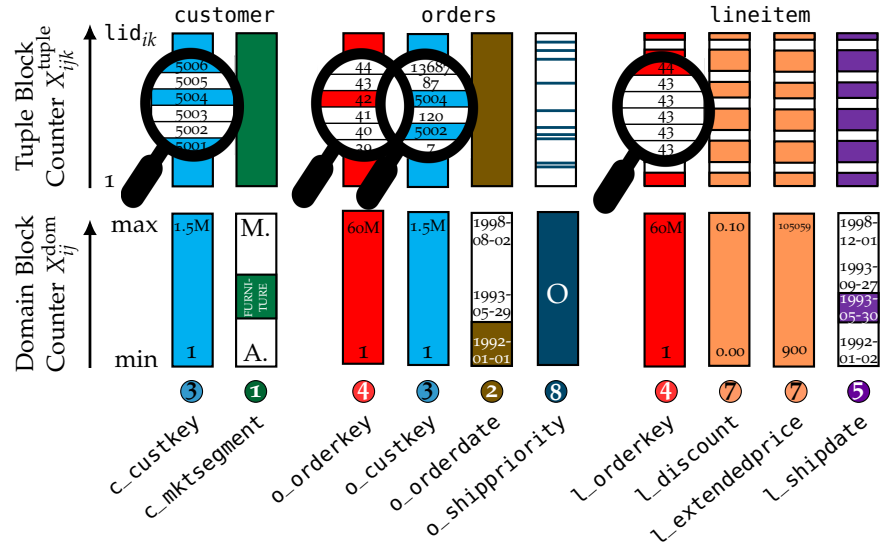
$$X^{tuple}_{ijkz}[b] := \begin{cases} 1 & \exists\, \texttt{gid}_i \in [1, |R_i|], \texttt{lid}_{ik} \in [1, |P_{ik}|], (t, S_q, V_q) \in \mathcal{W} : \\ & \quad (R_i, \texttt{gid}_i, A_{ij}, (t, S_q, V_q)) \in Tr \\ & \quad \wedge\, P_{ik}[\texttt{lid}_{ik}].\texttt{get\_gid} = \texttt{gid}_i \\ & \quad \wedge\, \lfloor \texttt{lid}_{ik} / TBS_{ijk} \rfloor = z \\ & \quad \wedge\, t \in \omega_b \\ 0 & \textit{otherwise.} \end{cases}$$

The b-th bit of counter X^{tuple}_{ijkz} is set if the workload trace Tr contains at least one element that accesses attribute A_{ij} within time window ω_b for a tuple with the global tuple identifier $\texttt{gid}_i$, such that the local tuple identifier $\texttt{lid}_{ik}$ of partition P_{ik} corresponds to the global tuple identifier $\texttt{gid}_i$ of the tuple and falls into the z-th block.

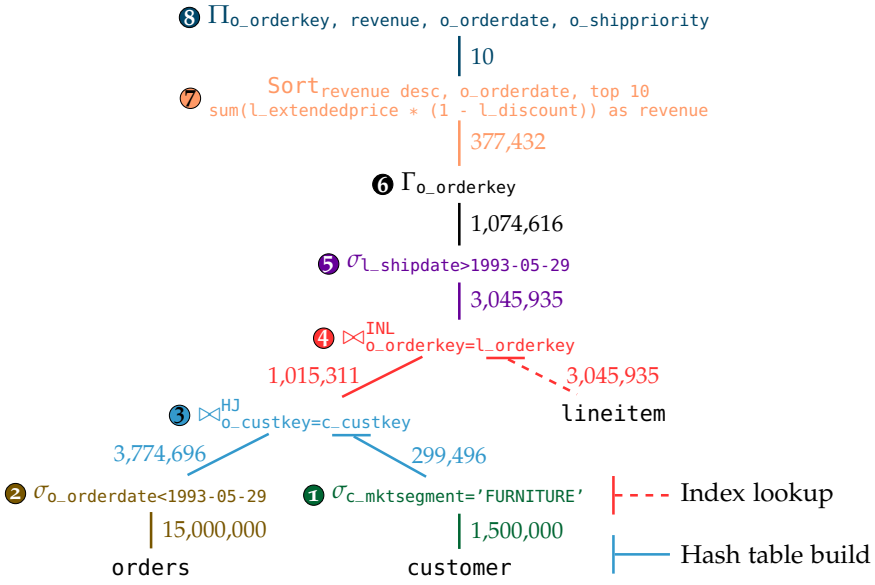
In our experiments in Section 4.8, we set the domain DBS_{ij} and tuple block size TBS_{ijk} such that 1% additional memory is spent on the collection of workload execution statistics compared to the storage size of the physical schema. Note that domain and tuple block counters may have different tuple and domain block sizes for a given memory budget due to varying data type width and compression.

EXAMPLE Figure 4.3 presents the collected workload execution statistics during one time window for the execution of SQL statement S_3 of the JCC-H benchmark (cf. Listing 2.1) with scale factor 10 and initialized with the vector $V_3 = [\text{FURNITURE}, 1993\text{-}05\text{-}29]$. In Figure 4.3a, we illustrate the tuple block counters on the top, whereas the domain block counters are at the bottom. Note that the physical schema of `customer`, `orders`, and `lineitem` are unpartitioned during statistics collection. The physical execution plan produced by SAP HANA's query optimizer [54, 97, 109, 153] is illustrated in Figure 4.3b. Every node represents an operator, and every edge an intermediate result with the actual output cardinality of the previously executed operator. In addition, we highlight the build side of join operators. We show the first access to each block because we only record whether or not a block was accessed during a time window. Accesses of blocks are highlighted using a unique color and number ⓧ to identify the operator in the physical execution plan that caused that access.

The selection operators ❶ and ❷ touch all tuple blocks of `c_mktsegment` and `o_orderdate`, but the respective domain blocks only record if domain values satisfy the selection predicate (cf. Definition 28). Therefore, range partitioning `orders` on `o_orderdate` with [1993-05-29, 1998-08-03) would create a column partition that is never accessed. In particular, whereas all 15 million tuples of `orders` are fetched without partitioning, a range partitioning specification with partition boundary values 1992-01-01 and 1993-05-29 would fetch only 3,204,724 tuples. The subsequent hash join ❸ touches all tuple and domain blocks on the build (`customer`) and the probe side (`orders`). However, only a subset of the tuples is accessed. For example, the customer with `c_custkey` '5004' was filtered out by ❶. Hence, the buffer pool is polluted with cold data since all pages are accessed but not all tuples. Next, an index nested loop join ❹ touches all tuple blocks in `orders` but only ≈75% of the tuple blocks in `lineitem`. For example, the order with `l_orderkey` '43' comprises 3 million items, which spans multiple blocks, but was already filtered out by ❷. The following selection ❺ filters all `l_shipdate` values smaller than 1993-05-30.



(a) Collected workload execution statistics on domain and tuple block counters for a non-partitioned layout of `customer`, `orders`, and `lineitem`. Accessed blocks are shown with a unique color, whereas blocks that are not accessed are white.



(b) The physical execution plan produced by SAP HANA's query optimizer [54, 97, 109, 153]. Every node represents an operator, and every edge an intermediate result with the actual output cardinality of the previously executed operator.

Figure 4.3: Illustration of the collected workload execution statistics during one time window for the execution of statement S_3 of the JCC-H benchmark (cf. Listing 2.1) with scale factor 10 and initialized with the vector V_3 = [FURNITURE, 1993-05-29]. Accesses to domain and tuple blocks are highlighted using a unique color and number ⓧ to identify the operator in the physical execution plan that caused that access.

Values larger than 1993-09-26 are not accessed since the `l_shipdate` of an item is at most 121 days after its `o_orderdate`, and orders with an `o_orderdate` larger than 1993-05-28 were filtered out by ❷. While such constraints are only known to domain experts [23] and cannot be extracted from physical execution plans, domain block counters can provide this insight. The memory footprint can be reduced by creating a range partition with [1993-05-30, 1993-09-27) on `l_shipdate`. In particular, 75% of `lineitem` pages are fetched without partitioning, while a partitioning layout based on the range partitioning specification with partition boundary values 1992-01-01, 1993-05-30, and 1993-09-27 would access only 5% of the pages. While the following group-by operator ❻ on `o_orderkey` does not create new accesses, the sorting operator ❼ additionally accesses `l_discount` and `l_extendedprice`. Finally, the projection ❽ accesses only ten blocks of `l_shippriority` because statement S_3 is limited to the top-10 results. However, the ten (block) accesses are distributed over the entire column partition.

4.5 DETERMINING PARTITIONING LAYOUTS

In this section, we determine a range partitioning specification $RPS_{i\lambda}$ with $A_{i\lambda}$ as a partition-driving attribute for a relation R_i based on the collected workload execution statistics of the previous section. Since any attribute $A_{i\lambda}$ may be the partition-driving attribute, we compute a range partitioning specification $RPS_{i\lambda}$ for each possible $A_{i\lambda}$. We then propose the range partitioning specification that minimizes the memory costs most while fulfilling the maximum workload execution time SLA. The calculation of the memory costs for a range partitioning specification is explained in Section 4.7 based on estimated access frequencies and storage sizes, as discussed in Section 4.6.

We first identify an optimal range partitioning specification $RPS_{i\lambda}$ for a partition-driving attribute $A_{i\lambda}$ (cf. Section 4.5.1). Afterwards, in Section 4.5.2, we present a heuristic that identifies a near-optimal range partitioning specification $RPS_{i\lambda}$ to lower the optimization time.

4.5.1 *Optimal Range Partitioning Specification*

Algorithm 1 finds an optimal range partitioning specification $RPS_{i\lambda}$ for a partition-driving attribute $A_{i\lambda}$ in relation R_i using dynamic programming (DP). The main idea is to calculate the optimal range partitioning specification for d $(1 \leq d \leq d_{i\lambda})$ distinct values of the domain of the partition-driving attribute $A_{i\lambda}$ by using a previously calculated optimal range partitioning specification with $d-1$ or less distinct values, where $d_{i\lambda}$ is the number of distinct values of $A_{i\lambda}$ (cf. Definition 1). We find the optimal range partitioning specification $RPS_{i\lambda}$ for $A_{i\lambda}$ iteratively.

To calculate the optimal range partitioning specification, Algorithm 1 uses two two-dimensional arrays `cost` and `split`. Array `cost`

Algorithm 1: Optimal Range Partitioning Specification

Memory: `cost[d][y]`: optimal memory costs for a range partition of d distinct values starting at attribute domain value $v_{i\lambda y}$
`split[d][y]`: optimal partition border for a range partition of d distinct values starting at attribute domain value $v_{i\lambda y}$

1 **Function** `DP`$(\mathcal{L}_{cur}, R_i, A_{i\lambda}, X^{tuple}, X^{dom})$**:**
2 **for** $1 \leq d \leq d_{i\lambda}$ **do** `// iterate over number of distinct values`
3 **for** $1 \leq y \leq d_{i\lambda} - d + 1$ **do** `// iterate over indexes of start values`
4 $v_{\text{ub}} \leftarrow \infty$ `// initialize upper partition boundary value`
5 **if** $y + d < d_{i\lambda}$ **then** $v_{\text{ub}} \leftarrow v_{i\lambda(\texttt{y+d})}$
6 `// calculate costs for a range partition on` $[v_{i\lambda y}, v_{\text{ub}})$
7 `cost[d][y]` $\leftarrow 0$
8 **for** $1 \leq j \leq m_i$ **do** `// iterate over each attribute index`
9 `// virtually create a column partition to assign statistics`
10 $C_{\text{virt}} \leftarrow$ virtual column part. for a range part. $[v_{i\lambda y}, v_{\text{ub}})$ on A_{ij}
11 `//estimate access freq. and storage size for` C_{virt} `(cf. Sec. 4.6)`
12 $\widehat{f}(C_{\text{virt}}, \mathcal{W}) \leftarrow$ `FreqEst`$(R_i, A_{ij}, A_{i\lambda}, v_{i\lambda y}, v_{\text{ub}}, X^{\text{tuple}}, X^{\text{dom}})$
13 $\widehat{||C_{\text{virt}}||} \leftarrow$ `SizeEst`$(R_i, A_{ij}, A_{i\lambda}, v_{i\lambda y}, v_{\text{ub}})$
14 `// estimate memory costs for` C_{virt} `(cf. Section 4.7)`
15 `cost[d][y]` $\leftarrow$ `cost[d][y]` $+ \mathcal{M}_{total}(C_{\text{virt}}, \mathcal{W}, \mathcal{L}_{\text{cur}})$
16 `split[d][y]` $\leftarrow \infty$ `// no partition border`
17 **for** $1 \leq b < d$ **do** `// iterate over partition borders`
18 `// check if beneficial to have a partition border at` $v_{i\lambda(y+b)}$
19 **if** `cost[b][y] + cost[d-b][y+b] < cost[d][y]`
20 `cost[d][y]` $\leftarrow$ `cost[b][y] + cost[d-b][y+b]`
21 `split[d][y]` $\leftarrow$ `b` `// partition border at` $v_{i\lambda(y+b)}$
22 $RPS_{i\lambda} \leftarrow \{\}$ `// create range partitioning specification`
23 `build`$(d_{i\lambda}, 1, RPS_{i\lambda})$ `// build range partitioning specification`
24 **return** $(\texttt{cost}[d_{i\lambda}][1], RPS_{i\lambda})$
25 **Procedure** `build(d, y, &`$RPS_{i\lambda}$`):` `// pass` $RPS_{i\lambda}$ `as reference`
26 **if** `split[d][y] =` ∞ **then** $RPS_{i\lambda} = RPS_{i\lambda} \cup \{v_{i\lambda y}\}$
27 **else**
28 `build(split[d][y], y,` $RPS_{i\lambda}$`)`
29 `build(d-split[d][y], y+split[d][y],` $RPS_{i\lambda}$`)`

(respectively `split`) stores at position `[d][y]` the optimal memory costs $\mathcal{M}_{total}$ (respectively partition border) for a range partitioning with d distinct values and the y-smallest value $v_{i\lambda y} \in \text{dom}(A_{i\lambda})$ as the lower bound of the range partition.

The first `for` loop (cf. Lines 2 to 21) iterates over the number of distinct values d, while the second `for` loop (cf. Lines 3 to 21) iterates over all possible indexes y of start values $v_{i\lambda y}$ as the lower bound of the range partition. For each combination of d and y, we then consider a *single* range partition on the value range $[v_{i\lambda y}, v_{i\lambda(y+d)})$ (or $[v_{i\lambda y}, \infty)$ for the last value range) with $A_{i\lambda}$ as a partition-driving attribute. We then calculate the memory costs for this range partition and assign it to the `cost` array at position `[d][y]` (cf. Lines 6 to 15). We do this by summing up the memory cost for each column partition that would be generated

for this range partition as we focus on column stores (cf. Lines 8 to 15). In particular, we *virtually* create a column partition C_{virt} on attribute A_{ij} for the range partition on $[v_{i\lambda y}, v_{i\lambda(y+d)})$ (or $[v_{i\lambda y}, \infty)$ for the last value range) with $A_{i\lambda}$ as a partition-driving attribute (cf. Line 10). For C_{virt}, we estimate the access frequency $\widehat{f}(C_{\text{virt}}, \mathcal{W})$ by the workload $\mathcal{W}$ and the storage size $\widehat{||C_{\text{virt}}||}$ based on the collected workload execution statistics of Section 4.4 (cf. Lines 12 and 13). The estimation of the access frequency and storage size is explained in Section 4.6. Both estimates are used to calculate the memory costs $\mathcal{M}_{total}$ for C_{virt} (cf. Line 15). Our cost model is defined in Section 4.7. In addition, we initialize the `split` array with ∞ to indicate that there is no partition border (cf. Line 16).

Afterwards, we check if it is more beneficial to have a partition border at $v_{i\lambda(y+b)}$ $(1 \leq b < d)$ and update `cost` and `split` accordingly (cf. Lines 17 to 21). For this, we combine the previously calculated optimal range partitioning specification for b distinct values starting at $v_{i\lambda y}$ with the previously calculated optimal range partitioning for $d - b$ distinct values starting at $v_{i\lambda(y+b)}$.

Finally, we build and return the optimal range partitioning specification with its memory costs $\mathcal{M}_{total}$ (cf. Lines 22 to 24). Lines 25 to 29 show the recursive build of the range partitioning specification based on the `split` array. The building process terminates once the splitting point is equal to the current number of distinct values d. We then add the lower border of the range partition $v_{i\lambda y}$ to the proposed range partition specification (cf. Line 26). Otherwise, it is recursively called with the range from the start to the splitting point (cf. Line 28) and from the splitting point to the end of the range (cf. Line 29).

COMPLEXITY The complexity of Algorithm 1 is $\mathcal{O}({d_{i\lambda}}^3)$. This is because the three nested `for` loops over the distinct values $d_{i\lambda}$ of the partition-driving attribute $A_{i\lambda}$ are the dominating term of the computational complexity of Algorithm 1. The three for loops start in Lines 2, 3, and 17. All other function calls and `for` loops are asymptotically subsumed by $d_{i\lambda}$. The reasons are threefold. First, the iteration over the number of attributes m_i (cf. Lines 8 to 15) in the second `for` loop over $d_{i\lambda}$ is negligible because the number of attributes is, in general, substantially smaller than $d_{i\lambda}$. Second, the call of `SizeEst` (cf. Line 13), as described in Algorithm 4, can be done in $\mathcal{O}(1)$. Finally, the execution of `FreqEst` (cf. Line 12), as detailed in Algorithm 3, iterates over all domain block counters X^{dom} of $d_{i\lambda}$ and all tuple block counters X^{tuple} of A_{ij} and $A_{i\lambda}$. The number of domain block counters X^{dom} of $d_{i\lambda}$ is, by definition, at most $d_{i\lambda}$ (cf. Definition 28). In addition, the number of tuple block counters is rather small and, in general, does not exceed $d_{i\lambda}$ because tuples are grouped into blocks to employ at most 1% of the storage size for the statistics collection (cf. Section 4.4).

CORRECTNESS We now prove that Algorithm 1 finds the range partitioning specification $RPS_{i\lambda}$ for a partition-driving attribute $A_{i\lambda}$ with minimal memory costs. To prove the correctness of the algorithm, we assume precise estimates on the memory costs $\mathcal{M}_{total}$ for single column partitions (cf. Line 15). We evaluate the precision of the estimates in Section 4.8.4 and the impact of the estimates on the optimality of the proposed range partitioning specification in Section 4.8.5.

Theorem 1. *Algorithm 1 finds an optimal range partitioning specification $RPS_{i\lambda}$ for a partition-driving attribute $A_{i\lambda}$ according to $\mathcal{M}_{total}$.*

Proof. We prove the correctness of Algorithm 1 by induction over the number of distinct values d for value ranges $[v_{i\lambda y}, v_{i\lambda(y+d)})$, respectively, $[v_{i\lambda y}, \infty)$ for the last value range.

Base case ($d = 1$): The only possible range partitioning specification for the value range $[v_{i\lambda y}, v_{i\lambda(y+1)})$ of any starting value $v_{i\lambda y} \in \text{dom}(A_{i\lambda})$ is a single range partition. Algorithm 1 is correct since `cost[1][y]` is initialized with the memory costs of the single range partition on the range $[v_{i\lambda y}, v_{i\lambda(y+1)})$ (cf. Lines 8 to 15), `split[1][y]` with the partition border ∞ (cf. Line 16), and both are not updated in Lines 17 to 21 since the condition of the `for` loop is not satisfied for any b if $d = 1$.

Induction step ($d - 1 \to d$): We now prove that Algorithm 1 finds the optimal range partitioning specification for the value range $[v_{i\lambda y}, v_{i\lambda(y+d)})$ of any starting value $v_{i\lambda y} \in \text{dom}(A_{i\lambda})$ with d distinct values. We assume the induction hypothesis that Algorithm 1 finds the optimal range partitioning specification for a value range with less than d distinct values. First, we have to show that Algorithm 1 considers that the optimal range partitioning specification can be a single range partition on the value range $[v_{i\lambda y}, v_{i\lambda(y+d)})$. This is considered by the initialization of `cost[d][y]` and `split[d][y]` (cf. Lines 8 to 16). Second, we have to show that Algorithm 1 considers that the optimal range partitioning specification can be a combination of optimal range partitioning specifications for the value ranges $[v_{i\lambda y}, v_{i\lambda(y+b)})$ and $[v_{i\lambda(y+b)}, v_{i\lambda(y+d)})$ with a partition border at $v_{i\lambda(y+b)}$ ($1 \leq b < d$). This is considered since Algorithm 1 iterates over all partition borders $v_{i\lambda(y+d)}$ and updates `cost[d][y]` and `split[d][y]` if the sum of the memory costs for the optimal range partitioning specifications on the value ranges $[v_{i\lambda y}, v_{i\lambda(y+b)})$ and $[v_{i\lambda(y+b)}, v_{i\lambda(y+d)})$ is smaller than `cost[d][y]` (cf. Lines 17 to 21). By the induction hypothesis, the memory costs of the optimal range partitioning specification for both value ranges can be fetched from `cost[b][y]` and `cost[d-b][y+b]` since b and $d - b$ are smaller than d. Hence, Algorithm 1 is correct. □

OPTIMIZATION Our experimental evaluation in Section 4.8 uses an optimized version of Algorithm 1. Instead of iterating over the number of distinct values $d_{i\lambda}$ of the partition-driving attribute $A_{i\lambda}$, we prune

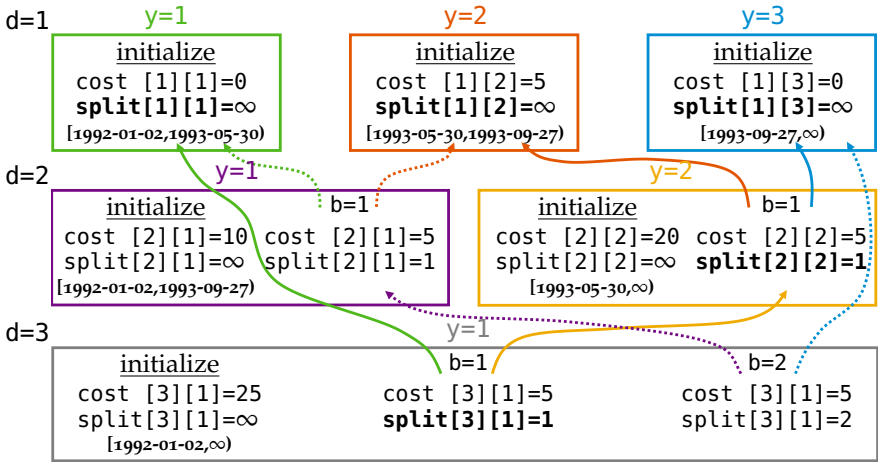


Figure 4.4: Illustration of how the optimized version of Algorithm 1 finds the optimal range partitioning specification for `lineitem` with `l_shipdate` as a partition-driving attribute based on the collected workload execution statistics of Figure 4.3.

the search space: First, we iterate only over the domain blocks $X_{i\lambda}^{\text{dom}}$ (cf. Definition 28) instead of all distinct values $d_{i\lambda}$. Second, we consider only borders between two consecutive domain blocks $X_{i\lambda z}^{\text{dom}}, X_{i\lambda(z+1)}^{\text{dom}}$ as a partition border if at least one timestamp $\omega_b \in \Omega$ is accessed differently, i.e., $\exists b \in [0, |\Omega|) : X_{i\lambda z}^{\text{dom}}[b] \neq X_{i\lambda(z+1)}^{\text{dom}}[b]$. We argue that both pruning strategies have no impact on uncompressed column partitions, i.e., we still find an optimal range partitioning specification by applying both pruning strategies. Otherwise, two partitions with the same temporal access pattern would be generated. Since no dictionary compression is applied, the storage size cannot decrease with partitioning. In contrast, we may not find the optimal range partitioning specification with dictionary compression and bit-packing if both pruning strategies are applied. If some values occur only in a single column partition, the storage size decreases because a dictionary-compressed column partition with bit-packing may require fewer bits to store the number representing the position of the value in the dictionary (cf. Section 2.3.2). The reason is that only a subset of the active domain of the attribute is present in this column partition. Nevertheless, we argue that the performance benefit is superior to the pruning of the search space. However, Algorithm 1 considers all values and finds the optimal range partitioning specification.

EXAMPLE In Figure 4.4, we show exemplarily how the optimized version of Algorithm 1 finds the optimal range partitioning specification for `lineitem` with `l_shipdate` as a partition-driving attribute based on the collected workload execution statistics of Figure 4.3. The domain block counters for `l_shipdate` in Figure 4.3 show only three potential lower bound values of a range partition: 1992-01-02, 1993-05-30, and 1993-09-27. Therefore, Figure 4.4 shows the iteration over d

($1 \leq d \leq 3$) horizontally (cf. Lines 2 to 21) and the iteration over the potential lower bound values $v_{i\lambda y}$ (cf. Lines 3 to 21) vertically. For each combination of d and y, we show the initialize step (cf. Lines 4 to 16) of the `cost` and `split` array at position `[d][y]` for a *single* range partition on the value range $[v_{i\lambda y}, v_{i\lambda(y+d)})$ (or $[v_{i\lambda y}, \infty)$ for the last value range). We also denote each step of the iteration over the partition borders at $v_{i\lambda(y+b)}$ ($1 \leq b < d$) (cf. Lines 17 to 21). To illustrate, the `cost` array for $d = 3$ and $y = 1$ is initialized with memory costs of 25 for the range partition $[1992\text{-}01\text{-}02, \infty)$. The memory costs is reduced, having a partition border at 1993-05-30 ($b = 1$) that uses the sum of the previously calculated optimal range partitioning for $d = 1$ and $y = 1$ and for $d = 2$ and $y = 2$. The recursive build of the optimal range partitioning specification from the `split` array is depicted in bold.

4.5.2 *Heuristic Approach SumMaxMinDiff*

Since Algorithm 1 finds an optimal partitioning but has cubic complexity, we now present a heuristic to lower optimization time. The idea is to leverage the domain block counters of the partition-driving attribute and cluster value ranges with almost identical accesses. On the one hand, we group consecutive domain blocks that were all accessed during the same time window to merge hot data into a single partition. On the other hand, we split domain blocks that were not all accessed during the same time window into partitions to separate hot and cold data. While this might generate a partition for each domain block, we introduce a heuristic that clusters consecutive domain blocks such that the number of time windows that are *partially* accessed is smaller or equal to a tuning parameter $\Delta_{SMMD} \in \mathbb{N}$. We call the *SumMaxMinDiff* as the number of partially accessed time windows for a set of consecutive domain blocks.

In Figure 4.5, we illustrate the calculation of the *SumMaxMinDiff* for two boundaries `l` and `r` based on collected domain blocks on attribute `o_orderdate` of `orders` (y-axis) while executing 200 SQL statements of JCC-H [22] with scale factor 10 during 89 time windows (x-axis). We highlight domain block accesses in red if, for a given time window $\omega_b \in \Omega$, domain blocks between `l` and `r` are fully accessed (i.e., in total 22 time windows). By contrast, we highlight domain block accesses in blue if, for a given time window $\omega_b \in \Omega$, blocks between `l` and `r` are only partially accessed (i.e., in total 16 time windows). Thus, the *SumMaxMinDiff* for the two boundaries `l` and `r` is 16.

Algorithm 2 describes the heuristic for finding a near-optimal range partitioning specification $RPS_{i\lambda}$ for a partition-driving attribute $A_{i\lambda}$ in relation R_i. Given domain block boundaries `l` and `r`, we search for the domain block that was accessed during most time windows, and place it into the current range partition, i.e., the boundaries $\widehat{\texttt{l}}$ and $\widehat{\texttt{r}}$ (cf. Lines 2 to 6). Afterwards, we iteratively extend the current

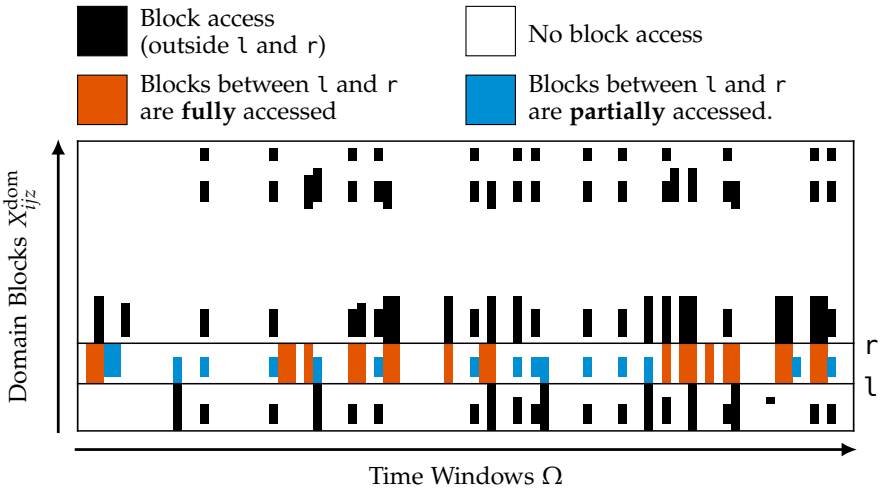


Figure 4.5: Illustration of the calculation of *SumMaxMinDiff* for two boundaries `l` and `r` based on collected domain blocks on attribute `o_orderdate` of `orders` (y-axis) while executing 200 SQL statements of JCC-H [22] during 89 time windows (x-axis).

range partition to the left or right as long as the *SumMaxMinDiff* of the current range partition is smaller or equal than Δ_{SMMD} (cf. Lines 7 to 12). Lines 18 to 26 show the calculation of the *SumMaxMinDiff*, as illustrated in Figure 4.5. For each time window $\omega_b \in \Omega$ (cf. Lines 20 to 25), we loop over all domain block numbers z between `l` and `r` (cf. Lines 22 to 24) and add a time window ω_b to the *SumMaxMinDiff* if at least one (cf. Line 23) but not all domain blocks (cf. Line 24) were accessed during ω_b, i.e., the blocks between `l` and `r` were only partially accessed within ω_b. Next, the heuristic is called recursively on all domain blocks with a smaller domain block number than $\hat{l}$ and on all domain blocks with a larger domain block number than $\hat{r}$ (cf. Lines 14 and 16). We also add the current lower bound value $v_{i\lambda(\hat{l} \cdot \mathrm{DBS}_{i\lambda})}$ as partition border to the range partitioning specification $RPS_{i\lambda}$ (cf. Line 15). To obtain the index of the value in the domain of the partition-driving attribute $A_{i\lambda}$, we multiply the domain block number $\hat{l}$ by the domain block size $\mathrm{DBS}_{i\lambda}$ of $A_{i\lambda}$. Finally, the range partitioning specification $RPS_{i\lambda}$ is returned (cf. Line 17).

The heuristic leverages only domain block counters instead of utilizing the memory cost function. Although the complexity is reduced to $\mathcal{O}({d_{i\lambda}}^2)$, we cannot guarantee an optimal range partitioning specification. Therefore, we evaluate in Section 4.8 the quality of the proposed range partitioning specifications by the heuristic.

To address large domains, e.g., more than 100,000 different accessed domain blocks, we propose a combination of Algorithms 1 and 2. Before running Algorithm 1, the heuristic should be executed with a small Δ_{SMMD} to reduce the number of partition border candidates.

Algorithm 2: Heuristic Approach SumMaxMinDiff

```
1  Function Heuristic(R_i, A_iλ, X_iλ^dom, l, r, Δ_SMMD):
2      hot ← l; f ← 0
3      for l ≤ z < r do // search for hottest domain block
4          f̂ ← popcount(X_iλz^dom)
5          if f̂ > f then hot ← z
6      l̂ ← hot; r̂ ← hot + 1 // initialize range partition
7      while l < l̂ ∨ r > r̂ do // extend range partition
8          Δ_l ← ∞; Δ_r ← ∞
9          if l < l̂ then Δ_l ← SumMaxMinDiff(R_i, A_iλ, X_iλ^dom, l̂ − 1, r̂)
10         if r > r̂ then Δ_r ← SumMaxMinDiff(R_i, A_iλ, X_iλ^dom, l̂, r̂ + 1)
11         if Δ_l > Δ_SMMD ∧ Δ_r > Δ_SMMD then break
12         if Δ_l ≤ Δ_r then l̂ ← l̂ − 1 else r̂ ← r̂ + 1
13     RPS_iλ ← {} // create range partitioning specification
14     if l < l̂ then RPS_iλ ← RPS_iλ ∪ Heuristic(R_i, A_iλ, X_iλ^dom, l, l̂, Δ_SMMD)
15     RPS_iλ ← RPS_iλ ∪ {v_iλ(l̂·DBS_iλ)} // partition border at pos l̂ · DBS_iλ
16     if r > r̂ then RPS_iλ ← RPS_iλ ∪ Heuristic(R_i, A_iλ, X_iλ^dom, r̂, r, Δ_SMMD)
17     return RPS_iλ
18 Function SumMaxMinDiff(R_i, A_iλ, X_iλ^dom, l, r):
19     Diff ← 0
20     for 0 ≤ b < |Ω| do
21         max ← 0; min ← 1
22         for l ≤ z < r do
23             max ← maximum(max, X_iλz^dom[b])
24             min ← minimum(min, X_iλz^dom[b])
25         Diff ← max − min + Diff
26     return Diff
```

OPTIMIZATION In our experimental evaluation in Section 4.8, we use an optimized version of Algorithm 2. During expansion (cf. Lines 7 to 12), we check if a minimal partition cardinality is satisfied all the time, which can lead to a range partition that does not satisfy the *SumMaxMinDiff* constraint. Nevertheless, we argue that this constitutes only a small overhead since we add at most as many values as the minimal partition cardinality. If the partition cardinality is too small, the overhead of scheduling jobs, e.g., scans, as well as opening and closing partitions becomes too large.

4.6 ACCESS FREQUENCY AND STORAGE SIZE ESTIMATOR

We now describe the estimation of the access frequency $\widehat{f}(C_{\text{virt}}, \mathcal{W})$ and the storage size $\widehat{||C_{\text{virt}}||}$ for a column partition C_{virt} that would be generated on attribute A_{ij} for the range partition $[v_{i\lambda y}, v_{i\lambda(y+d)})$ with $A_{i\lambda}$ as the partition-driving attribute. These estimates are required in Algorithm 1 (cf. Line 15) to calculate the memory costs $\mathcal{M}_{total}$ for C_{virt} (cf. Section 4.7) and to compute an optimal range partitioning

Algorithm 3: Column Partition Access Frequency Estimation

```
1  Function FreqEst(R_i, A_ij, A_iλ, v_iλy, v_iλ(y+d), X^tuple, X^dom):
2      B ← initialize bitset with length |Ω| and all bits set to 0
3      for 0 ≤ b < |Ω| do // iterate over all time window indexes
4          // case 1: attribute A_ij is not accessed
5          non ← 0
6          for 1 ≤ k ≤ p_iλ do // iterate over all partition indexes
7              for 1 ≤ z ≤ ⌊|P_ik|/TBS_ijk⌋ do // iterate over all block numbers
8                  non ← non ∨ X^tuple_ijkz[b]
9          if non = 0 then B[b] ← 0; break;
10         // case 2: A_ij is the partition-driving attribute A_iλ
11         acc ← 0
12         for ⌊y/DBS_iλ⌋ ≤ z < ⌈(y + d)/DBS_iλ⌉ do
13             acc ← acc ∨ X^dom_iλz[b]
14         if A_ij = A_iλ then B[b] ← acc; break
15         // case 3: passive attribute A_ij correlates to A_iλ
16         sub ← 1
17         for 1 ≤ k ≤ p_iλ do // iterate over all partition indexes
18             for 1 ≤ lid_ik ≤ |P_ik| do // iterate over all local tuple ids
19                 sub ← sub ∧ (X^tuple_ijk(⌊lid_ik/TBS_ijk⌋)[b] ≤ X^tuple_iλk(⌊lid_ik/TBS_iλk⌋)[b])
20         if sub = 1 then B[b] ← acc; break;
21         // case 4: passive attribute A_ij does not correlate to A_iλ
22         B[b] ← 1
23     return popcount(B)
```

specification from it recursively. The estimations are based on the domain X^{dom} and tuple block counters X^{tuple} of the *current* physical schema $\mathcal{L}_{\text{cur}}$ (cf. Section 4.4). We first describe the estimation of the access frequency (cf. Section 4.6.1) and then the estimation of the storage size (cf. Section 4.6.2).

4.6.1 *Estimation of the Column Partition Access Frequency*

In Algorithm 3, we detail the estimation of the access frequency $\widehat{f}(C_{\text{virt}}, \mathcal{W})$ during workload $\mathcal{W}$ for a column partition C_{virt} that would be generated on attribute A_{ij} for the range partition $[v_{i\lambda y}, v_{i\lambda(y+d)})$ with $A_{i\lambda}$ as a partition-driving attribute. Note that Algorithm 3 estimates the access frequency for column partitions of both partition-driving and passive attributes. We start by initializing a bitmap `B` of length $|\Omega|$ (cf. Line 2). Afterwards, we estimate if the column partition C_{virt} would be accessed for each time window $\omega_b \in \Omega$ (cf. Lines 3 and 22). Whenever the column partition C_{virt} is accessed during a time window ω_b, we set the bit at $\texttt{B}[b]$. We argue that four cases exist to estimate an access during ω_b to column partition C_{virt}:

CASE 1: The attribute A_{ij} was not accessed during ω_b, i.e., all tuple block counters $X^{\text{tuple}}_{ijkz}[b]$ over all partitions ($1 \leq k \leq p_{i\lambda}$) are zero. In this case, we estimate that the column partition C_{virt} is not accessed during ω_b and set $\texttt{B}[b] = 0$ (cf. Lines 4 to 9).

CASE 2: The attribute A_{ij} is the partition-driving attribute $A_{i\lambda}$. In this case, we leverage its domain block counters $X^{\text{dom}}_{i\lambda z}[b]$. The column partition is accessed (i.e., $\texttt{B}[b] = 1$) if the domain block counters record at least one access during ω_b that falls into the partition boundaries $[v_{i\lambda y}, v_{i\lambda(y+d)})$. If no access exists, we assume that the column partition C_{virt} is not accessed during ω_b, e.g., partition pruning is applied, and set $\texttt{B}[b] = 0$ (cf. Lines 10 to 14).

CASE 3: The range partition $[v_{i\lambda y}, v_{i\lambda(y+d)})$ with $A_{i\lambda}$ as the partition-driving attribute influences accesses to the passive attribute A_{ij}, e.g., if partition pruning is applied [103]. This is the case if the set of tuples accessed in the passive attribute A_{ij} during ω_b is a subset of the tuples accessed in the partition-driving attribute $A_{i\lambda}$. In particular, for each local tuple identifier $\texttt{lid}_{ik}$ over all partitions ($1 \leq k \leq p_{i\lambda}$), the tuple block counter of A_{ij} during ω_b is smaller or equal than the tuple block counter of $A_{i\lambda}$ during ω_b. Note that we need to iterate over all local tuple ids instead of all tuple blocks because the tuple block size for the partition-driving attribute $A_{i\lambda}$ can be different than the tuple block size of the passive attribute A_{ij} due to varying data type width and compression (cf. Lines 18 to 19). If the partition-driving attribute $A_{i\lambda}$ influences accesses to the passive attribute A_{ij}, we use the estimation of Case 2, i.e., we set $\texttt{B}[b] = 1$ if $A_{i\lambda}$ is assumed to be accessed; otherwise, we set $\texttt{B}[b] = 0$ (cf. Lines 15 to 20).

CASE 4: The range partition $[v_{i\lambda y}, v_{i\lambda(y+d)})$ with $A_{i\lambda}$ as the partition-driving attribute does not influence accesses to the passive attribute A_{ij}. In this case, we assume that the column partition C_{virt} is accessed during ω_b and set $\texttt{B}[b] = 1$ (cf. Lines 21 and 22).

Finally, we return the `popcount` of the bitmap `B` as estimated access frequency $\widehat{f}(C_{\text{virt}}, \mathcal{W})$ during workload $\mathcal{W}$ for C_{virt} (cf. Line 23).

EXAMPLE Let us consider the collected workload execution statistics in Figure 4.3. We observe that the domain block counters for `o_orderdate` show no access to the value range [1993-05-29, 1998-08-03). Thus, a column partition on `o_orderdate`, generated for the range partition [1993-05-29, 1998-08-03) with `o_orderdate` as the partition-driving attribute, would not be accessed. In addition, the accessed tuples of `o_custkey` are a subset of the accessed tuples of `o_orderdate`. Hence, the column partition of the passive attribute `o_custkey`, generated for the range partition [1993-05-29, 1998-08-03) with `o_orderdate` as the partition-driving attribute, would not be accessed, too.

Algorithm 4: Column Partition Storage Size Estimation

1 **Function** SizeEst($R_i, A_{ij}, A_{i\lambda}, v_{i\lambda y}, v_{i\lambda(y+d)}$):
2 // estimated column partition cardinality provided by the database
3 $\texttt{card} \leftarrow \texttt{DBEstimate}\left(\left|\sigma_{v_{i\lambda y} \leq A_{i\lambda} < v_{i\lambda(y+d)}}(R_i)\right|\right)$
4 // average storage size in bytes of a value $v_{ijy} \in \mathrm{dom}(A_{ij})$
5 $\texttt{val\_size} \leftarrow \texttt{DBEstimate}\left(||v_{ijy}||\right)$ [Bytes]
6 // calculate storage size of an uncompressed column partition
7 $\texttt{size\_unpart} \leftarrow \left\lceil \frac{\texttt{card} \cdot \texttt{val\_size}\ [\text{Bytes}]}{\texttt{PAGESIZE}^u_{\text{virt}}\ [\text{Bytes}]} \right\rceil \cdot \texttt{PAGESIZE}^u_{\text{virt}}\ [\text{Bytes}]$
8 // estimated distinct count provided by the database
9 $\texttt{distcnt} \leftarrow \texttt{DBEstimate}\left(\left|\Pi^D_{A_{ij}}\left(\sigma_{v_{i\lambda y} \leq A_{i\lambda} < v_{i\lambda(y+d)}}(R_i)\right)\right|\right)$
10 // calculate storage size of a dictionary
11 $\texttt{size\_dict} \leftarrow \left\lceil \frac{\texttt{distcnt} \cdot \texttt{val\_size}\ [\text{Bytes}]}{\texttt{PAGESIZE}^d_{\text{virt}}\ [\text{Bytes}]} \right\rceil \cdot \texttt{PAGESIZE}^d_{\text{virt}}\ [\text{Bytes}]$
12 // number of bits to represent all dictionary values and NULL
13 $\texttt{num\_bits} \leftarrow \lceil log_2(\texttt{distcnt} + 1) \rceil$ [Bits]
14 // calculate storage size of a dict.-compressed column partition
15 $\texttt{size\_part} \leftarrow \left\lceil \frac{\texttt{num\_bits}\ [\text{Bits}] \cdot \texttt{card}}{8 \cdot \texttt{PAGESIZE}^c_{\text{virt}}\ [\text{Bytes}]} \right\rceil \cdot \texttt{PAGESIZE}^c_{\text{virt}}\ [\text{Bytes}]$
16 **return** $\min(\mathit{size\_unpart}, \mathit{size\_dict} + \mathit{size\_part})$

4.6.2 *Estimation of the Column Partition Storage Size*

In Algorithm 4, we show the estimation of the storage size $\widehat{||C_{\text{virt}}||}$ for a column partition C_{virt} that would be generated on attribute A_{ij} for the range partition $[v_{i\lambda y}, v_{i\lambda(y+d)})$ with $A_{i\lambda}$ as the partition-driving attribute. Note that Algorithm 4 estimates the storage size of a column partition for both partition-driving and passive attributes.

We start by estimating the cardinality of the column partition, using a cardinality estimate provided by the database management system [37] (cf. Line 3). Afterwards, we estimate the attribute data type size in bytes (cf. Line 5). This estimate is precise for fixed-length data types, whereas for variable-length data types, the data distribution influences the precision of the estimate. As a next step, estimate the storage size of an uncompressed column partition as the product of the estimated cardinality and the attribute data type size (cf. Line 7). The storage size is a multiple of the page size for an uncompressed column partition $\texttt{PAGESIZE}^u_{\text{virt}}$ (cf. Section 2.1.3).

In order to estimate the storage size of a dictionary-compressed column partition, we first estimate the dictionary size, which is influenced by the number of values replicated within the dictionaries of different partitions (cf. Line 9). The database provides the estimated distinct count for the column partition [37]. Afterwards, we multiply the estimated distinct count and the attribute data type size in bytes to calculate the dictionary storage size. Note that we use a

multiple of the dictionary page size $\texttt{PAGESIZE}^{d}_{\text{virt}}$ (cf. Line 11). The estimated storage size of a dictionary-compressed column partition with bit-packing depends on the number of bits needed to represent all values of the attribute's domain within a column partition as described in Section 2.3.2 (cf. Line 13). In general, one additional value is reserved for the `NULL` value. We multiply the number of bits needed by the estimated cardinality and use a multiple of the page size of a dictionary-compressed column partition $\texttt{PAGESIZE}^{c}_{\text{virt}}$ (cf. Line 15).

Finally, we take the minimum of the estimated storage size of the uncompressed column partition and the sum of the storage size of the dictionary with the dictionary-compressed column partition as the estimated storage size $\widehat{||C_{\text{virt}}||}$ (cf. Line 16).

4.7 COST MODEL

We now describe the calculation of the memory costs $\mathcal{M}_{total}$ for a column partition C_{virt} that would be generated on attribute A_{ij} for the range partition $[v_{i\lambda y}, v_{i\lambda(y+d)})$ with $A_{i\lambda}$ as the partition-driving attribute. The calculation is based on the estimated access frequency $\widehat{f}(C_{\text{virt}}, \mathcal{W})$ for column partition C_{virt} during workload $\mathcal{W}$ and the estimated storage size $\widehat{||C_{\text{virt}}||}$ for column partition C_{virt}, as described in the previous section. The estimated memory costs $\widehat{\mathcal{M}}_{total}$ for column partition C_{virt} is then required in Algorithm 1 (cf. Line 15) to compute an optimal range partitioning specification $RPS_{i\lambda}$.

The main idea is that the estimated access frequency $\widehat{f}(C_{\text{virt}}, \mathcal{W})$ for column partition C_{virt} is utilized to classify the column partition as either hot or cold, which then determines its memory costs. As introduced in Section 2.4, we consider the π-second rule to classify a column partition as hot or cold. We then assume that a hot-classified column partition is configured to hold all data in DRAM, such that its memory costs depend on DRAM prices and the storage size of the column partition. In contrast, cold-classified column partitions can be pruned during the evaluation of the selection predicate and, therefore, are not required to be held in DRAM during the entire workload. Since disk I/O is performed instead for each access, their memory costs depend on the number of accessed pages by workload $\mathcal{W}$ and the memory costs per disk IOP. As a consequence, the buffer pool size $\mathcal{BS}$ can be calculated by summing up the storage sizes of all hot-classified column partitions.

Definition 30 (Hot/Cold Classification)**.** *Given an estimated column partition access frequency $\widehat{f}(C_{virt}, \mathcal{W})$ by workload $\mathcal{W}$, a maximum workload execution time SLA in seconds, and π in seconds, a column partition C_{virt} is classified as hot if it is accessed more frequently than every π-seconds:*

$$\texttt{isHot}(C_{virt}, \mathcal{W}) \Leftrightarrow \frac{SLA\ [\text{sec}]}{\widehat{f}(C_{virt}, \mathcal{W})} \leq \pi\ [\text{sec}].$$

We classify a column partition C_{virt} as hot if the fraction of the maximum workload execution time SLA and the estimated number of accesses is smaller or equal to π. A misclassification of a hot column partition as cold induces many expensive disk IOPs, degrades performance, and potentially violates the SLA. By contrast, misclassification of a cold column partition as hot increases the DRAM consumption, resulting in higher internal costs for the database-as-a-service provider. In terms of our cost model, the additional memory footprint for data misclassified as cold is unbounded, whereas data misclassified as hot is bounded. Nevertheless, in Section 4.8.4, we show that our estimates are accurate so that expensive misclassifications of a hot column partition as cold and vice versa are prevented.

The memory costs $\mathcal{M}_{total}$ of a column partition C_{virt} are defined in two steps. As a first step, we define the workload memory costs $\mathcal{M}_{workload}$ of a column partition C_{virt} that occur during the execution of the workload $\mathcal{W}$. As a second step, we define the table repartitioning costs $\mathcal{M}_{repart}$ incurred by generating column partition C_{virt}. We argue that considering table repartitioning costs is crucial because Algorithm 1 should only propose a change in the range partitioning specification $RPS_{i\lambda}$ when the expected benefits of workload cost reductions outweigh the cost of table repartitioning. Finally, we combine the workload and table repartitioning costs into $\mathcal{M}_{total}$.

Definition 31 (Workload Memory Costs)**.** *The the workload memory costs in $ of a column partition C_{virt} during workload $\mathcal{W}$ is defined as*

$$\mathcal{M}_{workload}(C_{virt}, \mathcal{W}) := \begin{cases} \mathcal{M}_{hot}(C_{virt}, \mathcal{W})\ [\$] & \textit{if } \texttt{isHot}(C_{virt}, \mathcal{W}) \\ \mathcal{M}_{cold}(C_{virt}, \mathcal{W})\ [\$] & \textit{otherwise.} \end{cases}$$

We now specify the cost functions $\mathcal{M}_{hot}$ and $\mathcal{M}_{cold}$. The memory costs of a hot-classified column partition are affected by the estimated storage size of the column partition in bytes, the maximum workload execution time in seconds, and the DRAM costs (in $ per byte per second) because all data are held in DRAM.

Definition 32 (Memory Costs of a Hot-Classified Column Partition)**.** *Given the estimated storage size of a column partition $\widehat{||C_{virt}||}$ in bytes, a maximum workload execution time SLA in seconds, and the costs in $ for one byte of main memory per seconds. The memory costs in $ of a hot-classified column partition C_{virt} for workload $\mathcal{W}$ is defined as*

$$\mathcal{M}_{hot}(C_{virt}, \mathcal{W}) := \textit{DRAM Costs} \left[\frac{\$/Byte}{\text{sec}}\right] \cdot \widehat{||C_{virt}||}[Byte] \cdot SLA[\text{sec}].$$

The memory costs of a column partition classified as cold consider the estimated storage size in bytes of the column partition, the column partition access frequency, and the hardware configuration because data are fetched for every access.

Definition 33 (Memory Costs of a Cold-Classified Column Partition)**.** *Given the estimated storage size of a column partition* $\widehat{||C_{virt}||}$ *in bytes, an estimated column partition access frequency* $\widehat{f}(C_{virt}, \mathcal{W})$ *by workload* $\mathcal{W}$*, the page size* $\texttt{PAGESIZE}_{virt}$ *in bytes of* C_{virt}*, the costs in \$ for a single disk per seconds, and the amount of random read IOPs per second (IOPS) of a disk. The memory costs in \$ of a cold-classified column partition* C_{virt} *for workload* $\mathcal{W}$ *is defined as*

$$\mathcal{M}_{cold}(C_{virt}, \mathcal{W}) := \widehat{f}(C_{virt}, \mathcal{W}) \cdot \left\lceil \frac{\widehat{||C_{virt}||}\ [Byte]}{\texttt{PAGESIZE}_{virt}\left[\frac{Byte}{Page}\right]} \right\rceil \cdot \frac{Disk\ Costs[\$/sec]}{Disk\ IOPS\left[\frac{Page}{\text{sec}}\right]}.$$

As a second step, we consider the table repartitioning costs that would be incurred by generating column partition C_{virt}. A straightforward approach would be to calculate table repartitioning costs globally, i.e., at the relation level. However, because Algorithm 1 operates at the level of column partitions to find an optimal range partitioning specification recursively, we need to compute the table repartitioning costs per column partition. Furthermore, we decided to model table repartitioning costs in \$ to be able to compare the table repartitioning costs to the specified workload memory costs (cf. Definition 31).

If a column partition C_{virt} already exists in the table partitioning layout $\mathcal{T}_i$ of the current physical schema $\mathcal{L}_{\text{cur}}$, we set its repartitioning costs to \$0. Otherwise, the column partition C_{virt} is created by splitting or merging column partitions in the table partitioning layout $\mathcal{T}_i$ of the current physical schema $\mathcal{L}_{\text{cur}}$, e.g., by separating hot and cold data of a single range partition into two range partitions. In this case, we make four assumptions:

1. As column stores often employ read-optimized data structures (e.g., compressed dictionaries) that cannot be modified easily, we assume that C_{virt} is created from scratch (cf. Section 2.1.3).

2. Each value written to column partition C_{virt} was read from an existing column partition in the table partitioning layout $\mathcal{T}_i$ of the current physical schema $\mathcal{L}_{\text{cur}}$. Accordingly, we assume that the total amount of data accessed during repartitioning is twice the estimated storage size of the column partition C_{virt}.

3. To reduce the impact on performance commitments, we ensure fast repartitioning times by assuming that all data accessed during repartitioning is held in DRAM.

4. We assume that the system exhibits a repartitioning rate R_{rate} in Byte/ sec for creating column partition C_{virt}, as well as a fixed overhead R_{ovrhd} in sec for initializing C_{virt} (cf. Section 4.8).

Based on these assumptions, we define the table repartitioning costs $\mathcal{M}_{repart}$ that would occur for generating the column partition C_{virt}.

Definition 34 (Table Repartitioning Costs). *Given the estimated storage size of a column partition* $\widehat{||C_{virt}||}$ *in bytes, a repartitioning rate* R_{rate} *in Byte/* sec*, a fixed overhead* R_{ovrhd} *in* sec*, and the costs in $ for one byte of main memory per seconds. The table repartitioning costs* $\mathcal{M}_{repart}$ *in $ for a column partition* C_{ijk} *based on the current physical schema* $\mathcal{L}_{cur}$ *is defined as*

$$\mathcal{M}_{repart}(C_{virt}, \mathcal{L}_{cur}) := \begin{cases} \$0 & \text{if } C_{virt} \in \mathcal{T}_i, (\mathcal{IC}_i, \mathcal{T}_i, \texttt{comp}_{i\lambda}) \in \mathcal{L}_{cur} \\ DRAM\ Costs\left[\frac{\$/Byte}{\text{sec}}\right] \cdot & \text{otherwise.} \\ 2 \cdot \widehat{||C_{virt}||}\,[Byte] \cdot \left(\frac{2 \cdot \widehat{||C_{virt}||}\,[Byte]}{R_{rate}\left[\frac{Byte}{\text{sec}}\right]} + R_{ovrhd}\,[\text{sec}]\right) & \end{cases}$$

We now combine the workload and table repartitioning costs into the total memory cost $\mathcal{M}_{total}$ in $ of a column partition C_{virt}.

Definition 35 (Total Memory Costs). *We define the total memory costs* $\mathcal{M}_{total}$ *in $ for a column partition* C_{virt} *based on the workload* $\mathcal{W}$ *and the current physical schema* $\mathcal{L}_{cur}$ *as*

$$\mathcal{M}_{total}(C_{virt}, \mathcal{W}, \mathcal{L}_{cur}) := \mathcal{M}_{workload}(C_{virt}, \mathcal{W}) + \mathcal{M}_{repart}(C_{virt}, \mathcal{L}_{cur}).$$

Finally, we argue that a minimum partition cardinality as a system-specific restriction exists. The reason is that the overhead of scheduling jobs and opening and closing partitions becomes too large when a large amount of range partitions exists with a small partition cardinality. As a consequence, we assign infinite memory costs to small range partitions, such that Algorithm 1 proposes a range partitioning specification, where the cardinality of each range partition is equal to or larger than the threshold.

4.8 EXPERIMENTAL EVALUATION

The experimental evaluation of SAHARA is the final contribution of this chapter. We implemented SAHARA as a prototype into SAP HANA (cf. Section 2.3). The experimental setup is discussed in Section 4.8.1. We then evaluate the memory footprint reduction achieved by SAHARA (cf. Section 4.8.2), the hardware cost savings (cf. Section 4.8.3), the precision of access frequency, storage size, and memory cost estimations (cf. Section 4.8.4), optimality of range partitioning specifications (cf. Section 4.8.5), and the runtime and memory overhead as well as the optimization time of SAHARA (cf. Section 4.8.6).

4.8.1 *Experimental Setup*

We use the same test system as introduced in Section 3.4.1. As workloads, we consider JCC-H [22] (scale factor 10) and the Join Order Benchmark (JOB) [98] (cf. Section 3.4.1). We argue that both workloads pose a challenging environment for SAHARA as they include data and

parameter skew, as well as data correlation. For both workloads, we randomly sampled 200 SQL statements. In Section 3.4.1, we showed how often each templated SQL statement occurs in each workload.

PARAMETERS We calculate $\pi = 70$ seconds by inserting the prices, capacities, and performance of our hardware into the π-second rule (cf. Section 2.4). As a result, we set the time window length to $\pi/2 = 35$ seconds, such that we fulfill the Nyquist–Shannon sampling theorem (cf. Section 4.4). Furthermore, we set the minimum partition cardinality to 100,000 based on the multi-threading and partitioning capabilities of SAP HANA. The page size varies between 4 KB and 16 MB, depending on the column partition data type and their usage (cf. Section 2.3.3). In order to spend at most 1% additional memory compared to the storage size on collecting workload execution statistics, we set the block size of the tuple and domain block counters as follows: First, we group consecutive logical tuple identifiers of dictionary-compressed column partitions (i.e., value identifiers) into blocks such that the cumulative number of bits of each block with bit-packing equals 4 KB. Second, we limit the number of domain blocks per dictionary to 5000. Finally, we set R_{rate} to 133 MB/ sec and R_{ovrhd} to 0.16 seconds using a q-error approximation [114]. In Section 4.8.4, we analyze the impact of R_{rate} and R_{ovrhd} on the precision of repartitioning costs. Overall, we argue that the parameters are neither workload-specific nor need tuning by a database administrator.

BASELINE AND DATABASE EXPERTS: In order to demonstrate SAHARA's effectiveness, we compare SAHARA against combinations of partitioning layouts and buffer pool sizes. As a baseline, we include the non-partitioned layout. Since related approaches (cf. Section 4.9) optimize performance and, therefore, differ in their objective function to SAHARA, we compare ourselves to carefully hand-optimized partitioning layouts proposed by database experts.

For JCC-H, the table partitioning layouts referred to as *DB Expert* 1 represents the recommendation [45] of hash partitioning the primary key columns of `orders` and `lineitem`. The table partitioning layouts referred to as *DB Expert* 2 represents the recommendation [34] of range partitioning `orders` and `lineitem` on the columns `o_orderdate` and `l_shipdate`. SAHARA also recommends range partitioning `orders` and `lineitem` on `o_orderdate` and `l_shipdate` but with a different range partitioning specification. In particular, the partition boundary values are picked based on the workload's data access pattern. To give an example, a range partition for `orders` with the value range [1995-01-02, 1997-05-01) is created because it will be only rarely accessed (cf. Figure 4.1). The table partitioning layouts from SAHARA and both database experts for JCC-H are illustrated in Table 4.1.

Relation	Partition-Driving Attribute	Range Partitioning Specification
orders	o_orderdate	{ 1992-01-01, 1993-01-01, 1993-05-30, 1994-05-27, 1995-01-02, 1997-05-01, 1997-09-02 }
lineitem	l_shipdate	{ 1992-01-02, 1993-01-01, 1994-01-01, 1995-01-01, 1996-05-21 }

(a) SAHARA

Relation	Partition-Driving Attribute	Number of Hash Partitions
orders	o_orderkey	8
lineitem	l_orderkey	8

(b) DB Expert 1

Relation	Partition-Driving Attribute	Range Partitioning Specification
orders	o_orderdate	{ 1992-01-01, 1993-01-01, 1994-01-01, 1995-01-01, 1996-01-01, 1997-01-01, 1998-01-01 }
lineitem	l_shipdate	{ 1992-01-01, 1993-01-01, 1994-01-01, 1995-01-01, 1996-01-01, 1997-01-01, 1998-01-01 }

(c) DB Expert 2

Table 4.1: Table partitioning layouts from SAHARA and both database experts for JCC-H [22].

To the best of our knowledge, no related work on partitioning the tables of JOB exists. As JOB executes many joins between the foreign key column `movie_id` and the primary key column `id` of table `title`, *DB Expert* 1 might partition on these columns. The physical schema referred to as *DB Expert* 2 creates range partitioning specifications on columns with selective filter predicates, e.g., `production_year` of `title`. SAHARA instead recommends range partitioning specifications for the relations `aka_name`, `cast_info`, `char_name`, `name`, `title`, and `movie_info`. In Table 4.2, we illustrate the table partitioning layouts from SAHARA and both database experts for JOB.

For both JCC-H and JOB, we compare SAHARA against three strategies to configure the buffer pool size. The strategy referred to as *ALL in Memory* denotes the baseline where the buffer pool size is set to the accumulated storage size of all partitions. This yields the best performance but results in a high memory costs. The strategy referred to as *WS in Memory* is a database expert, who profiled the workload accesses and set the buffer pool size to the working set (WS) size, i.e., all accessed data fits into the buffer pool. The strategy referred to as *MIN in Memory (SLA)* represents a database expert, who sets the buffer pool size to the smallest value such that the *SLA* is still fulfilled.

Relation	Partition-Driving Attribute	Range Partitioning Specification
aka_name	person_id	{ 4, 2534118 }
cast_info	note	{ '((as Laura Bauer) (as Laura Cunningham)', '(episode 2 "Total Wedding")', '(location assistant) (as Anna Johansson)', '(printer: SQ Film Laboratories) (as Oscar Camposano)', '(production accountant) (as Jose Lichtig)', '(story "Langes radieux")', '(story) (as Martin Herbert)', '(video "Loshadka" species of fishes remix)', '(written by) (as Frank Wind)' }
char_name	name	{ '!Nanseb', 'Benjamin Verdoodt' 'Dolores LeBeau', 'Frania Caravelle', 'Herself', 'Inserviente piscina', 'Lizuca', 'Mrs. Hanton', 'Playground kid 2', 'Scullen', 'Tinkerbell the Dog' }
name	gender	{ 'f', 'm' }
title	episode_nr	{ 1, 3434 }
movie_info	note	{ '"3 Ways to Kill a Mook" Film Premiere Program', '(interiors: motel)', '<Stever', 'Foreign Policy Association', 'trivia' }

(a) SAHARA

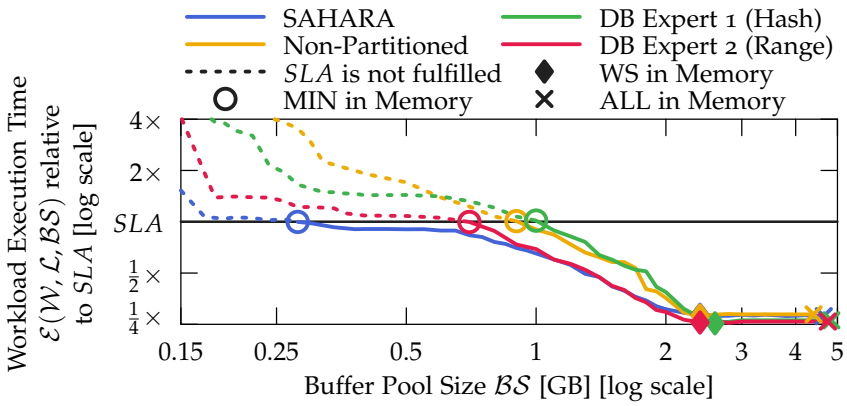
Relation	Partition-Driving Attribute	Number of Hash Partitions
aka_title	movie_id	8
cast_info	movie_id	8
complete_cast	movie_id	8
movie_companies	movie_id	8
movie_info_idx	movie_id	8
movie_keyword	movie_id	8
movie_link	movie_id	8
title	id	8
movie_info	movie_id	8

(b) DB Expert 1

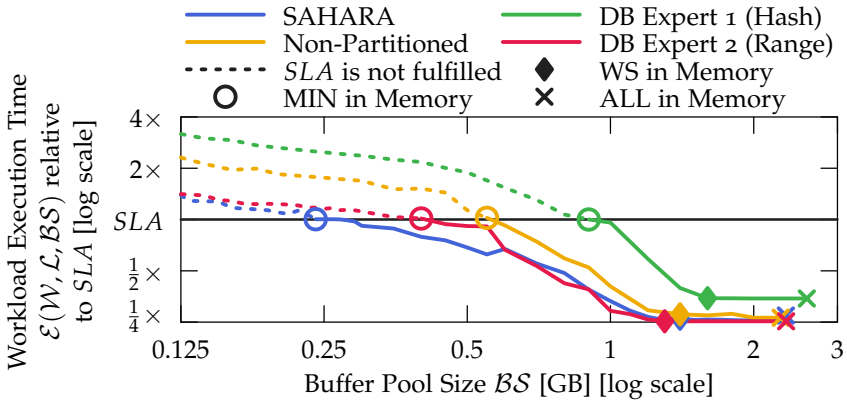
Relation	Partition-Driving Attribute	Range Partitioning Specification
company_name	country_code	{ '[ad]', '[us]', '[uy]' }
name	gender	{ 'f', 'm' }
title	production_year	{ 1, 1950, 1980, 2000, 2005, 2010 }

(c) DB Expert 2

Table 4.2: Table partitioning layouts from SAHARA and both database experts for JOB [98].



(a) JCC-H (scale factor 10)



(b) Join Order Benchmark

Figure 4.6: Comparison of workload execution times $\mathcal{E}(\mathcal{W}, \mathcal{L}, \mathcal{BS})$ relative to the SLA (y-axis) for varying buffer pool sizes $\mathcal{BS}$ (x-axis) between SAHARA and table partitioning layouts proposed by database experts, running the workloads JCC-H and JOB.

4.8.2 *Experiment 1: Memory Footprint Reduction*

The first experiment analyzes the effect of the proposed partitioning specifications on the minimal required buffer pool size $\mathcal{BS}$ that still fulfills a performance commitment, defined as a maximum workload execution time SLA. For illustration purposes, we depict an SLA as a maximum workload execution time that is $4\times$ slower than the in-memory workload execution time on a non-partitioned layout. We observed similar memory footprint reductions for other SLAs.

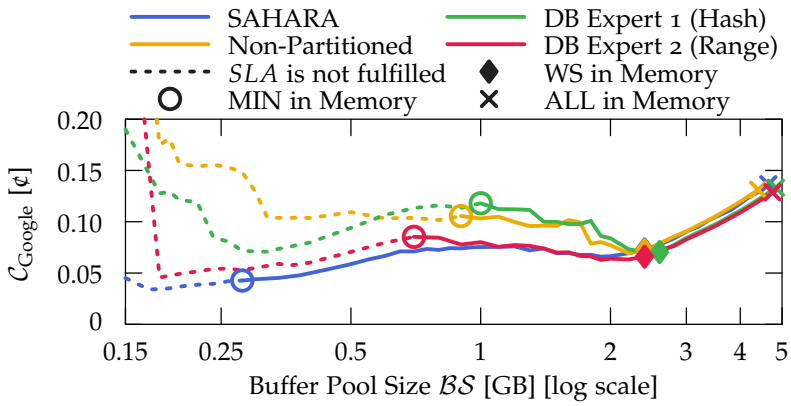
Figure 4.6a shows on the y-axis the relative end-to-end workload execution time for the previously explained table partitioning layouts of JCC-H. The x-axis represents the buffer pool size $\mathcal{BS}$. The storage sizes differ for all table partitioning layouts since the partitioning specification impacts dictionary compression and additional compression techniques such as bit-packing. For instance, hash partitioning

produces many duplicate dictionary entries. The execution times of all table partitioning layouts are approximately equal between the storage size (ALL in Memory) and the size of the accessed data (WS in Memory). In this segment, the buffer pool size may be reduced without increasing execution times. Further lowering the buffer pool size starts to increase the execution time. For the non-partitioned layout, the smallest possible buffer pool size, which still fulfills the SLA, is 900 MB. DB Expert 1 needs a buffer pool size of at least 1000 MB because hash-partitioning does not cluster hot and cold data into separate partitions, while DB Expert 2 can decrease the buffer pool size until 700 MB using range partitioning. The physical schema proposed by SAHARA reduces the buffer pool size to 280 MB while still fulfilling the SLA by separating hot and cold data into disjoint partitions to avoid pollution of the buffer pool with cold data. As a consequence, SAHARA increases the tenant density by 2.5× compared to table partitioning layouts proposed by experts. Compared to the total storage size of each physical schema, SAHARA reduces the memory footprint by a factor of 16.6, whereas the workload execution time increases, according to the SLA, only by a factor of 4. Since SAHARA consistently yields the best performance or comes close to the best performance for all buffer pool sizes, SAHARA reduces the memory footprint for all other possible SLAs, too.

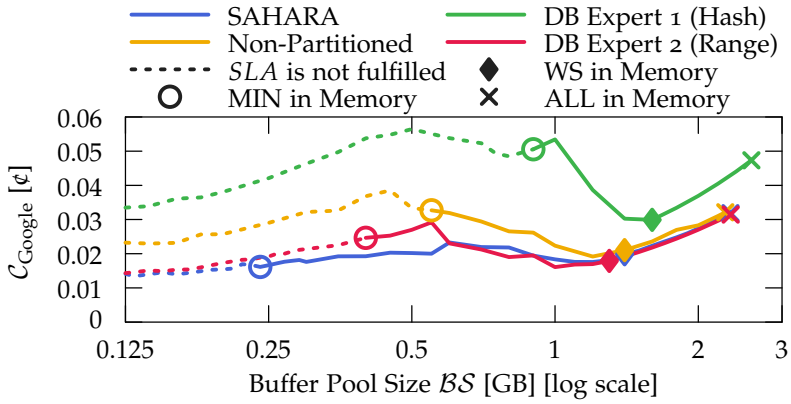
The measurements for JOB in Figure 4.6b show similar effects. SAHARA is again able to run the workload with the smallest buffer pool (240 MB) and increases the tenant density by at least 1.7× compared to database experts and the baseline. DB Expert 1 consumes substantially more memory than other table partitioning layouts due to many duplicate dictionary entries caused by hash partitioning.

4.8.3 *Experiment 2: Hardware Cost Savings*

The second experiment analyzes the hardware cost that a database-as-a-service provider needs to pay for executing the workload $\mathcal{W}$. As SAHARA optimizes the memory costs, we calculate the hardware costs with a fixed number of CPUs. The task of proposing an appropriate number of CPUs [41, 43] is beyond the scope. We run the experiment on the introduced on-premise hardware (cf. Section 3.4.1) but map the provisioned hardware costs to a *memory-optimized* Google Cloud instance, priced at $2610 per TB/month of DRAM and $80.00 per TB/-month for regional standard provisioned disk space, as introduced in Section 2.4. While DRAM and provisioned disk space are billed per GB on Google Cloud, database-as-a-service providers can reduce hardware costs internally on a more fine-granular level by placing multiple database instances on the same node. As a result, we consider memory costs $\mathcal{C}_{\text{Google}}$ of a Google Cloud instance per MB/s in ¢.



(a) JCC-H (scale factor 10)



(b) Join Order Benchmark

Figure 4.7: Comparison of hardware memory costs $\mathcal{C}_{\text{Google}}$ in ¢ on Google Cloud (y-axis) for varying buffer pool sizes $\mathcal{BS}$ (x-axis) between SAHARA and table partitioning layouts proposed by database experts, running the workloads JCC-H and JOB.

Figure 4.7a shows on the y-axis the memory cost $\mathcal{C}_{\text{Google}}$ in ¢ for different table partitioning layouts of JCC-H and on the x-axis the buffer pool size $\mathcal{BS}$. We use the same definition of the *SLA* as in Experiment 1. The costs of all table partitioning layouts decrease from the storage size (ALL in Memory) until the first local minimum close to the size of the accessed data (WS in Memory). By lowering the buffer pool size further, the costs start to increase because increasing execution times impact costs more heavily than reduced buffer pool sizes. Below a buffer pool size of ca. 800 MB, costs for SAHARA and both database experts are reduced since hot data are cached in the buffer pool. While the *SLA* for both experts is no longer fulfilled, SAHARA reduces the costs to 0.04¢ with a buffer pool size of 280 MB and fulfills the *SLA*. For the non-partitioned layout and both experts, the cost-optimal buffer pool size (0.06¢) that fulfills the *SLA* is 2.4 GB. Thus, SAHARA yields the smallest buffer pool size and memory costs.

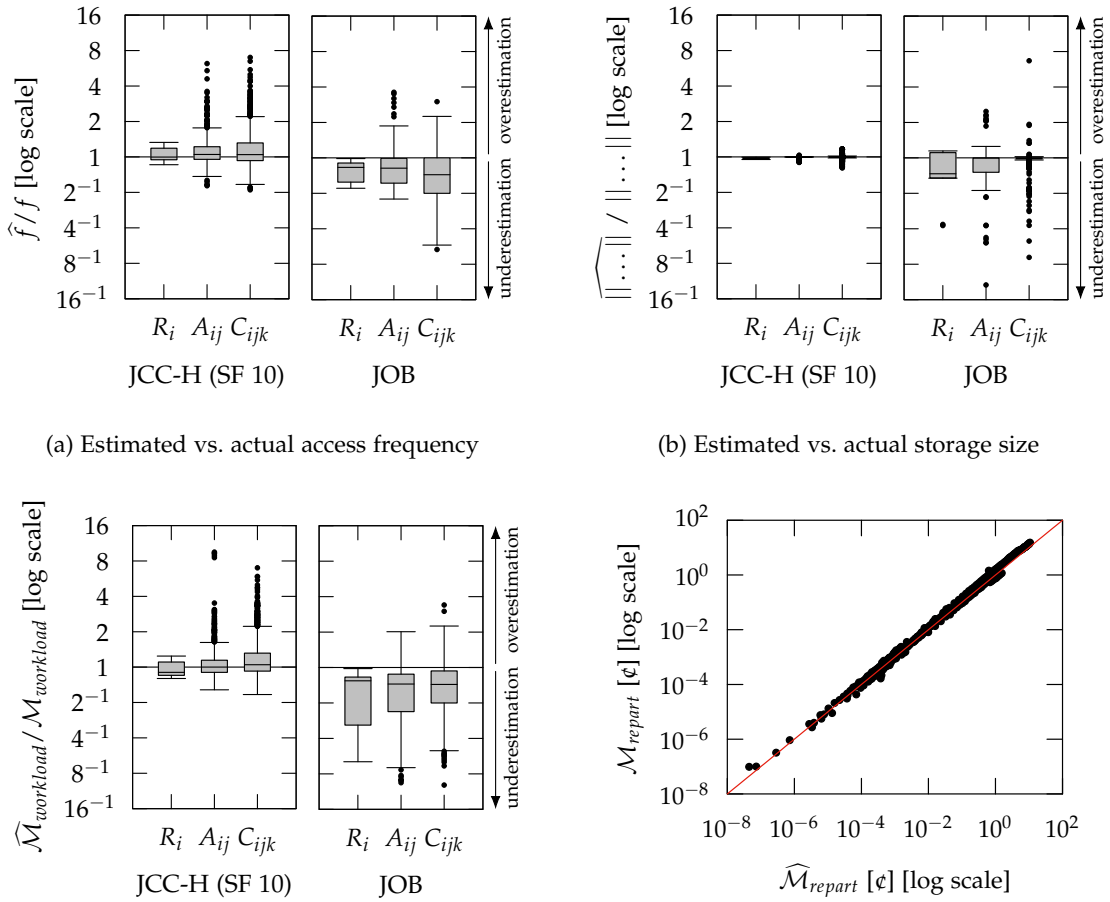
The measurements for JOB in Figure 4.7b show similar behavior. SAHARA achieves a cost-optimal buffer pool size, still fulfilling the *SLA*, at only 240 MB (0.15¢), while other table partitioning layouts require a buffer pool size of at least 1000 MB for minimal costs (0.16¢).

4.8.4 *Experiment 3: Precision of Estimates*

The third experiment evaluates how precisely SAHARA estimates access frequencies, storage sizes, workload memory costs, and table repartitioning costs. We generated for JCC-H 67 and for JOB 37 random range partitioning specifications with a random partition-driving attribute and then compared the estimated and actual values at relation, attribute, and column partition level. In summary, we analyzed 67 estimates at relation, 1030 at attribute, and 5699 at column partition level for JCC-H, and 37 estimates at relation, 310 at attribute, and 2237 at column partition level for JOB. In contrast to estimates for access frequencies, storage sizes, and workload memory costs, table repartitioning costs can only be compared at the relation level. For this purpose, we generated 363 random range partitioning specifications with a random partition-driving attribute for `lineitem` and `orders` of JCCH and `makes`, `servers`, `install_sessions`, `test_profiles`, `test_cases`, `test_case_info`, `test_log_files`, and `single_tests` of the quality assurance database (QADB) of the SAP HANA development project. The QADB will be explained in more detail in Chapter 5.

ACCESS FREQUENCY The y-axis of Figure 4.8a shows the ratio between the estimated and the actual access frequency at relation, attribute, and column partition level using different range partitioning specifications for JCC-H (on the left) and JOB (on the right). Overestimated is shown on the top, underestimation is depicted on the bottom. Since partition pruning impacts the access frequency and SAHARA proposes a new range partitioning specification based on the collected workload execution statistics of the current physical schema, the current range partitioning specification can impact the precision of the estimates (cf. Section 4.6.1). In general, we observe that most estimates are bound by a factor of 4. Therefore, expensive misclassifications of a hot page as being cold and vice versa are prevented.

STORAGE SIZE Figure 4.8b shows the ratio between the estimated and the actual storage size at relation, attribute, and column partition granularity using different range partitioning specifications for JCC-H and JOB. An overestimated storage size is shown on the top, underestimation is depicted on the bottom. We observe that all storage size estimation errors for JCC-H are bounded by a factor of 1.5. Although JCC-H adds skew to the data compared to TPC-H, it is still a synthetic benchmark and, therefore, not a challenging task for the storage size



(a) Estimated vs. actual access frequency

(b) Estimated vs. actual storage size

(c) Estimated vs. actual workload memory costs

(d) Estimated vs. actual table repartitioning costs

Figure 4.8: Precision of access frequency, storage size, workload memory costs, and table repartitioning costs at relation, attribute (except table repartitioning costs), and column partition level (except table repartitioning costs) for random range partitioning specifications with random partition-driving attribute.

estimator. For JOB, SAHARA tends to underestimate the storage size. This is because commercial systems tend to underestimate the cardinality for JOB, as shown by Leis et al. [98], and the cardinality estimate mainly influences the estimated storage size (cf. Algorithm 4). Even so, most estimation errors are still bounded by a factor 2.

WORKLOAD MEMORY COSTS Figure 4.8c shows the estimated-to-actual workload memory cost ratio at relation, attribute, and column partition levels. Again, overestimation is shown on the top, while underestimation is depicted on the bottom. The estimated workload memory cost depends on the estimated access frequency and the estimated storage size. We observe that SAHARA tends to overestimate the workload memory cost for JCC-H because the access frequencies tend to be slightly overestimated, whereas the storage sizes are esti-

mated nearly optimal. The estimated access frequency dominates the estimated workload memory cost for JCC-H. In contrast, estimates for JOB are underestimated because both access frequencies and storage sizes tend to be underestimated.

TABLE REPARTITIONING COSTS In contrast to access frequencies, storage sizes, and workload memory costs, table repartitioning costs can only be analyzed at the relation level as repartitioning is a process that is performed for the entire table. For each generated table repartitioning process, Figure 4.8d shows on the x-axis the estimated table repartitioning costs with R_{rate} set to 133 MB/sec and R_{ovrhd} set to 0.16 sec, while the y-axis then denotes the measured table repartitioning costs. We observe a strong correlation between the estimated and actual repartitioning costs as the maximum q-error [115] is 2.27.

4.8.5 *Experiment 4: Optimality*

The fourth experiment evaluates the impact of the estimates on the optimality of the proposed range partitioning specification. For this experiment, we created range partitioning specifications with the lowest estimated memory costs $\widehat{\mathcal{M}}_{total}$ for all possible partition-driving attributes and number of partitions. We then ran the workload and compared the *actual* memory costs $\mathcal{M}_{total}$ for each created table partitioning layout against the solution proposed by SAHARA, the non-partitioned table partitioning layout, and the table partitioning layouts proposed by database experts.

Figure 4.9 shows on the y-axis the *actual* memory footprint $\mathcal{M}_{total}$ for table partitioning layouts of six different partition-driving attributes of `lineitem`. The x-axis denotes the number of partitions per layout. We also highlight SAHARA, the non-partitioned layout, and the layouts chosen by database experts. As SAHARA estimates are accurate (cf. Section 4.8.4), the proposed table partitioning layout with five partitions and `l_shipdate` as partition-driving attribute is close to the optimum with seven partitions. DB Expert 2 chooses the same partition-driving attribute but has higher memory costs than SAHARA due to a different range partitioning specification. DB Expert 1 picks the wrong partition-driving attribute (`l_orderkey`) and has higher memory costs than most other table partitioning layouts. Note that `l_receiptdate` and `l_commitdate` as partition-driving attributes also have low memory costs due to their correlation with `l_shipdate`. We observed similar behavior for other tables of JCC-H and JOB.

As SAHARA's choice is close to the optimum, it particularly reduces the access frequencies while still considering the impact of range partitioning specification on the storage size. The reason is that an increasing number of partitions would separate hot and cold data better into partitions by reducing the number of accesses and,

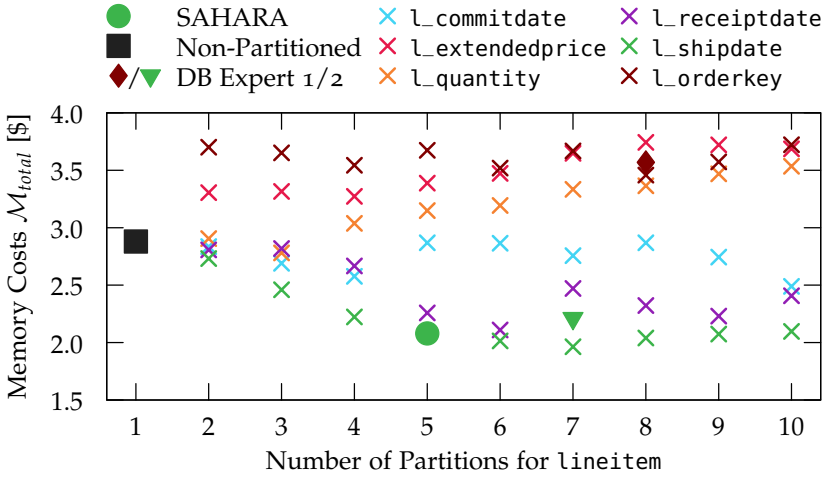


Figure 4.9: Comparison between SAHARA, a non-partitioned layout, table partitioning layouts proposed by database experts, and layouts with the lowest estimated memory costs for all possible partition-driving attributes and number of partitions (x-axis) of `lineitem`. The y-axis shows the *actual* memory costs $\mathcal{M}_{total}$ after running the JCC-H benchmark on all table partitioning layouts.

therefore, reducing the memory costs. However, an increasing number of partitions would also increase the storage size in most cases due to dictionary duplicates and, therefore, increase the memory costs. SAHARA instead balances both. In summary, SAHARA's partitioning layout is close to the optimum, while other partitioning layouts may fail due to the wrong choice of the partition-driving attribute or range partitioning specification.

In Figure 4.9, we showed the output of SAHARA using Algorithm 1. In order to lower the optimization time, we introduced the SumMaxMinDiff heuristic (cf. Algorithm 2). However, the SumMaxMinDiff heuristic does not guarantee an optimal range partitioning specification with respect to estimated access frequencies and storage sizes. When we apply the SumMaxMinDiff heuristic instead of Algorithm 1 for proposing range partitioning specifications, we observe that for JCC-H only the range partitioning specification for `orders` and `lineitem` has higher memory costs $\mathcal{M}_{total}$ compared to the range partitioning specification proposed by Algorithm 1. The memory costs increase by 0.6% for `orders` and by 0.8% for `lineitem`. For JOB, the proposed range partitioning specifications for `aka_name` using the SumMaxMinDiff heuristic results in 0.1% higher memory costs $\mathcal{M}_{total}$ compared to Algorithm 1. For `cast_info` the memory costs $\mathcal{M}_{total}$ increase by 2.9%, for `char_name` by 4.3%, and for `movei_info` by 6.5% compared to Algorithm 1. In summary, the SumMaxMinDiff heuristic provides near-optimal partitioning specifications because the memory costs increase by at most 6.5%.

Workload	JCC-H	JOB
Statistics Collection: Memory Overhead	0.39%	0.28%
Statistics Collection: Runtime Overhead	14.84%	18.74%
Optimization Time: Algorithm 1 (DP)	3.06 sec	1.45 sec
Optimization Time: Algorithm 2 (SumMaxMinDiff)	0.02 sec	0.01 sec

Table 4.3: Memory and runtime overhead for the collection of workload execution statistics and optimization time for determining range partitioning specifications.

4.8.6 *Experiment 5: Overhead and Optimization Time*

The final experiment evaluates the memory (relative to the storage size) and runtime overhead (relative to the in-memory workload execution time of Experiment 1) for collecting workload execution statistics, as well as the optimization time of SAHARA, using either Algorithm 1 (DP) or Algorithm 2 (SumMaxMinDiff).

In Table 4.3, we see that SAHARA has a low optimization time and a low memory overhead. However, the runtime overhead is notable and similar compared to the runtime overhead of Access Counter 4, as evaluated in Section 3.4.5. Nevertheless, we argue that although the runtime overhead is notable, our approach enables substantial memory cost reductions. Furthermore, the workload execution statistics may be collected only periodically or sampling could be applied to reduce the runtime overhead.

4.9 RELATED WORK

We now discuss the novelties of SAHARA compared to related approaches. Our findings are also summarized in Figure 4.10.

The objective function is the main difference between state-of-the-art table partitioning advisors and SAHARA. While all other table partitioning advisors focus on maximizing performance, the function of SAHARA is reducing memory footprint.

Besides SAHARA, Casper [11] is the only table partitioning advisor specifically built for column stores. All other table partitioning advisors are mainly designed for row stores. Besides the different objective function, Casper has the following two shortcomings: First, the partition-driving attribute has to be provided by the database administrator. Second, only selection operators are considered during optimization. Instead, our approach recommends a partition-driving attribute and a range partitioning specification (cf. Section 4.5). Furthermore, we handle all operators of the physical execution plan during statistics collection (cf. Section 4.4) and estimation (cf. Section 4.6), and, therefore, SAHARA can be used for any workload.

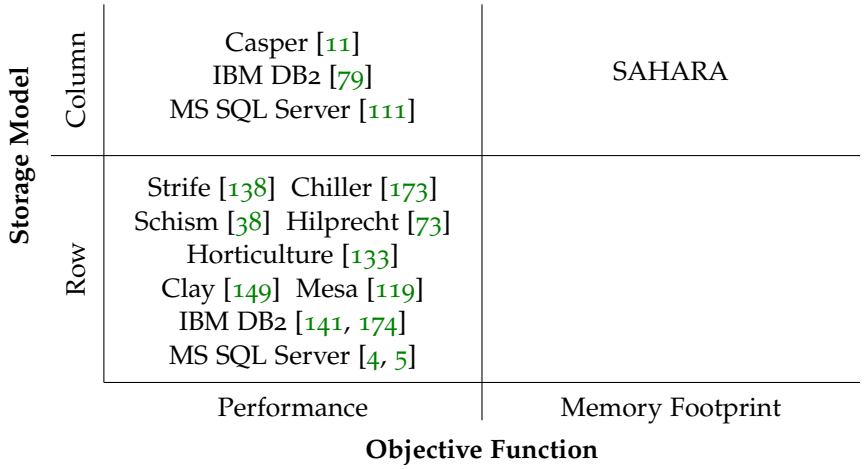


Figure 4.10: Comparison between SAHARA and state-of-the-art table partitioning advisors on their objective function and storage model.

Schism [38], Clay [149], Horticulture [133], Mesa [119], Hilprecht et al. [73], Strife [138], and Chiller [173] are table partitioning advisors for distributed DBMS and designed for row stores. They aim to minimize cross-partition transactions by distributing hot accesses evenly across all server nodes, thus generating pages with mixed temperatures (i.e., hot and cold data). As a result, the buffer pool is polluted with cold data. Our approach does the exact opposite: SAHARA separates hot and cold data into partitions, such that a buffer pool pollution with cold data is avoided, and the memory footprint can be reduced.

Table partitioning advisors in IBM DB2 [79, 141, 174] and Microsoft SQL Server [4, 5, 111] support column stores only partially. For example, IBM DB2 does not support range partitioning specifications for column store tables [78]. Both commercial tools minimize estimated query costs, i.e., query response time, using the optimizer's what-if API. Apart from a different objective function, our approach does not rely on the optimizer's what-if API and is not sensitive to any skew in the distribution of data accesses. We determine a range partitioning specification based on physical collected data accesses (cf. Section 4.4).

As DRAM is an expensive hardware resource [106], the following approaches also focus on identifying hot and cold data, intending to move cold data to secondary storage or compressing cold data with a higher compression ratio. However, SAHARA is the first approach that uses table partitioning as a technique to reduce the memory footprint.

Data skipping [158] and Qd-tree [172] analyze filter predicates and group tuples into pages to minimize I/O cost by routing the statements to the blocks that need to be accessed. Both approaches work on a more fine-granular level and thus can be applied on top of SAHARA.

Project Siberia [7, 51, 102] and X-Engine [76] leverage access frequencies at row, respectively, at extent granularity, to identify tuples that can be moved to secondary storage. While X-Engine does not

disclose details of its temperature metric, Project Siberia collects log samples to estimate the access frequency. SAHARA instead collects all physical data accesses of the workload (cf. Section 4.4) and proposes a range partitioning specification rather than move individual tuples from DRAM to secondary storage.

Anti-Caching [44] and LeanStore [99, 120] utilize replacement policies instead of access counters to identify cold data. Unlike both approaches, SAHARA's uses the π-minute-rule to classify data as hot and cold without additional tuning knobs, based only on the hardware and the workload (cf. Section 4.7).

Hyrise [20] and Mosaic [162] determine hot and cold columns based on a representative workload sample. Since data access patterns are already heavily distorted within a column due to events like *Black Friday* [22, 46, 76], SAHARA classifies hot and cold data at a more fine-granular level and proposes a range partitioning specification.

HyPer [56] uses flags of the CPU's MMU for each virtual memory page to identify cold pages for compression. While HyPer considers only compression and does not change the physical schema, we consider partition pruning (cf. Section 4.6.1) and dictionary compression (cf. Section 4.6.2) while determining an optimal range partitioning specification. Both are excellent opportunities to reduce the memory footprint of database management systems.

4.10 DISCUSSION

In this chapter, we presented a table partitioning advisor that utilizes the collected workload execution statistics from Chapter 3. Our table partitioning advisor focuses on minimizing the buffer pool size while still adhering to performance commitments and, therefore, addresses the second challenge of this dissertation (cf. Section 1.3.2). The proposed solution is based on a typical workload characteristic, where tuples of a relation are either frequently or rarely accessed according to a value range of a specific attribute of that relation. As a consequence, we group tuples that belong to hot-classified value ranges into hot range partitions, whereas tuples for cold-classified value ranges are gathered into cold range partitions. We then keep only hot-classified column partitions with a high density of hot data in the buffer pool. This results in a reduced buffer pool size and avoids polluting DRAM with cold data. In addition, we consider dictionary compression with bit packing and partition pruning during optimization. Both are excellent further opportunities to reduce the buffer pool size and are employed in many column stores. Finally, we integrated SAHARA as a prototype into SAP HANA. Our experimental evaluation (cf. Section 4.8) demonstrates a buffer pool size reduction of up to 3.2× while still adhering to SLAs compared to table partitioning layouts proposed by database experts and the non-partitioned layout.

5 A FRAMEWORK FOR WORKLOAD PREDICTION

In the previous chapter, we demonstrated how a table partitioning advisor utilizes the collected workload execution statistics from Chapter 3 to propose a physical schema that allows reducing the buffer pool size while still fulfilling performance commitments. As real-world applications, however, are characterized by workloads where the arrival rate and parameterization of SQL statements change over time, the proposed physical schema may no longer be optimal. This may lead to an increased buffer pool size, or performance commitments can no longer be fulfilled. Therefore, we now address the third challenge of this dissertation (cf. Section 1.3.3), which concerns predicting the future workload based on an observed workload. The predicted workload can then be used as input to the table partitioning advisor from Chapter 4 or by any other physical database design advisor that relies on workload statistics to optimize the physical schema for the future.

5.1 MOTIVATION

As the physical schema of a database significantly impact its performance and memory footprint, academia and industry developed tools for automated physical database design advice (cf. Section 2.2). Most existing advisor tools, particularly table partitioning advisors [4, 5, 25, 38, 73, 119, 141, 173], focus only on static workloads, i.e., a physical schema is proposed under the assumption that the collected workload execution statistics (cf. Chapter 3) are stable over time. However, workloads for real-world applications are in most cases not stable and thus change over time [49, 74, 75, 107, 134]. More specifically, *workload drifts* can be identified, which are characterized by a temporal change in the arrival rate of SQL statements or in the parameter values assigned to the host variables of a statement. In real-world applications, the five drift types linear, exponential, reoccurring, static, and irregular have been identified [57, 75, 107]. Whenever workload drifts are not addressed timely, the current physical schema may no longer be optimal. As a consequence, the database-as-a-service provider may increase the buffer pool size of hosted database instances to still adhere to performance commitments. This may lead to increased internal costs and lower the profitability of database-as-a-service providers.

A straightforward method to deal with workload drifts is to repeatedly feed the observed workload into a physical database design advisor at fixed intervals [11, 133, 138, 149]. However, such an approach is inherently *backward-looking* as the suggested physical schema

lags behind workload drifts and may already be suboptimal when the data reorganization starts. By contrast, we propose a *forward-looking* approach that determines the new physical schema using a prediction of the future workload. We then feed the predicted workload into a physical database design advisor, such that the physical schema is optimized for the future workload.

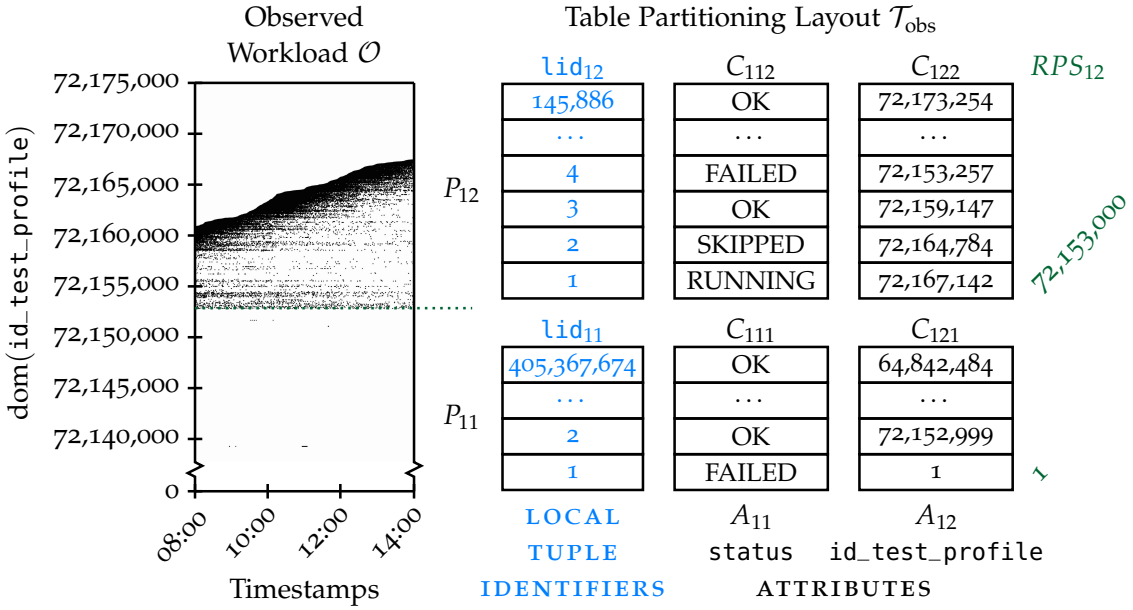
To illustrate the benefits of a forward-looking approach, let us consider the quality assurance database (QADB) of the SAP HANA development project [12, 13] as a real-world scenario for workload drift. The QADB records statistics about more than 30 billion test runs to identify bugs or authorize patches. In QADB, related *test cases* (e.g., TPC-H benchmark queries) are grouped into *test profiles*. Therefore, the following SQL statement returns the status (e.g., OK, running, failed, skipped) of all test cases with the given test profile ID:

```
SELECT status
FROM   test_cases
WHERE  id_test_profile = :1.
```

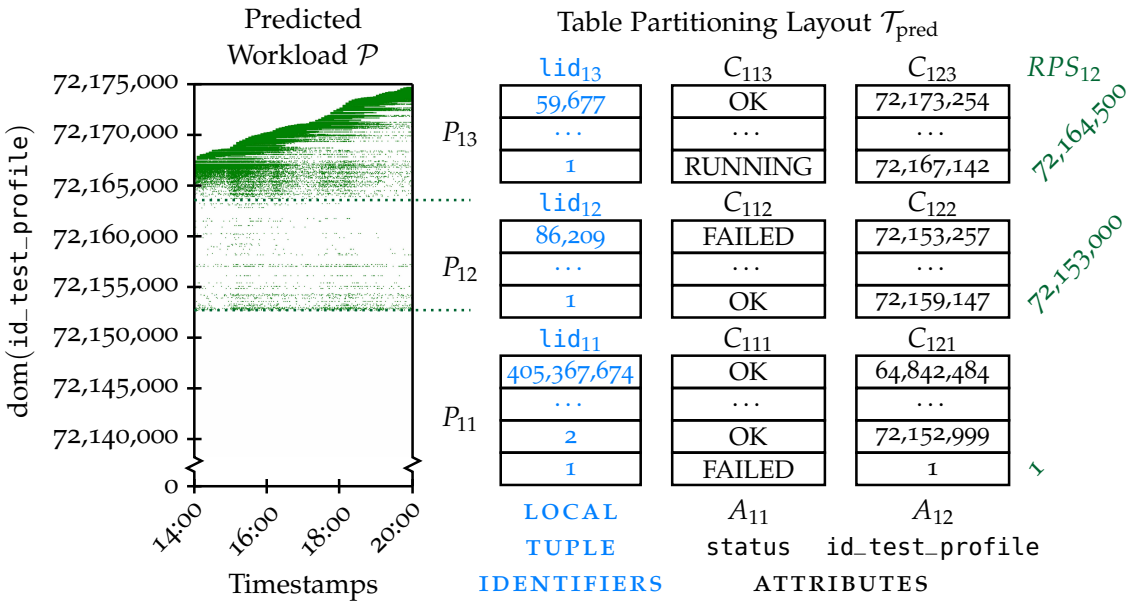
In Figure 5.1a, we depict on the left side the observed workload during a six-hour time window on a regular workday. Each dot represents a statement execution at a specific time (x-axis) with a specific parameter value (y-axis) assigned to the host variable `:1`. We observe that the workload drifts over time as the later the statement is instantiated, the larger the parameter value becomes. Using a backward-looking approach, the table partitioning advisor of Chapter 4 proposes the table partitioning layout $\mathcal{T}_{\text{obs}}$, illustrated on the right side in Figure 5.1a. The table partitioning layout $\mathcal{T}_{\text{obs}}$ is generated by grouping frequently accessed tuples of `test_cases` (R_1) with `id_test_profile` $\geq$ 72,153,000 into a hot partition P_{12} and all other records into a cold partition P_{11}.

Contrary to that, a forward-looking approach first predicts the future workload based on the observed workload and then feeds the predicted workload into a table partitioning advisor. On the left side in Figure 5.1b, we illustrate the predicted workload. The table partitioning advisor of Chapter 4 proposes then a table partitioning layout $\mathcal{T}_{\text{pred}}$, illustrated on the right side in Figure 5.1b, which groups frequently accessed tuples of `test_cases` (R_1) with `id_test_profile` $\geq$ 72,164,500 into a hot partition P_{13}, while rarely accessed tuples with 72,153,000 $\leq$ `id_test_profile` $<$ 72,164,500 are grouped into a cold partition P_{12} and all other tuples with no accesses into another cold partition P_{11}.

In order to demonstrate the benefits of the forward-looking approach in terms of memory footprint, we ran the actual future workload, as illustrated in Figure 5.2, in SAP HANA's column store (cf. Section 2.3) for both partitioning layouts, where each layout is stored on pages, is not clustered by `id_test_profile`, and no index exists on `id_test_profile`. After running the future workload, we observed that the table partitioning layout $\mathcal{T}_{\text{pred}}$, as proposed by the forward-



(a) Observed Workload $\mathcal{O}$ and the proposed table partitioning layout $\mathcal{T}_{obs}$ (backward-looking approach)



(b) Predicted Workload $\mathcal{P}$ and the proposed table partitioning layout $\mathcal{T}_{pred}$ (forward-looking approach)

Figure 5.1: The assignments of parameter values to the host variable `:1` for statement "`SELECT status FROM test_cases WHERE id_test_profile = :1`" of the observed $\mathcal{O}$ and predicted workload $\mathcal{P}$. The backward-looking approach then proposes for relation `test_cases` (R_1) the table partitioning layout $\mathcal{T}_{obs}$ based on the observed workload $\mathcal{O}$, while a forward-looking approach proposes table partitioning layout $\mathcal{T}_{pred}$ based on the predicted workload $\mathcal{P}$.

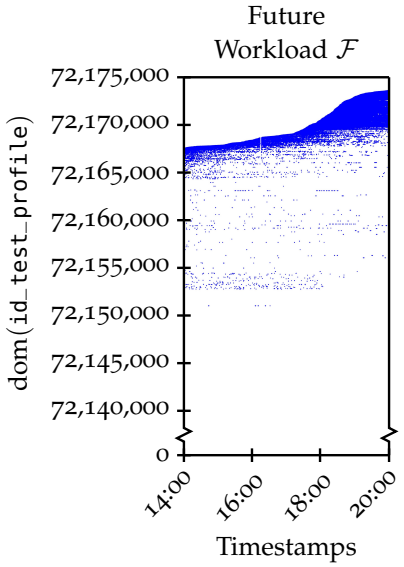


Figure 5.2: The assignments of parameter values to the host variable `:1` for statement "`SELECT status FROM test_cases WHERE id_test_profile = :1`" of the actual future workload $\mathcal{F}$.

looking approach, reduces the number of frequently accessed hot pages held in DRAM by a factor of ≈ 2 compared to the table partitioning layout $\mathcal{T}_{\text{obs}}$ as suggested by the backward-looking approach. There are two reasons for this: First, since the future workload (cf. Figure 5.2) frequently accesses tuples with `id_test_profile` $\geq$ 72,164,500, partition pruning during the evaluation of the selection predicate results only in frequent scans of the small column partition C_{123} in layout $\mathcal{T}_{\text{pred}}$ (59,677 tuples), while in the backward-looking approach the larger column partition C_{122} in layout $\mathcal{T}_{\text{obs}}$ (145,886 tuples) must be scanned. Second, the qualifying tuples to project on `status` are only distributed on the smaller partition P_{13} in layout $\mathcal{T}_{\text{pred}}$ compared to the larger partition P_{12} in layout $\mathcal{T}_{\text{obs}}$. As a consequence, only pages of the small column partition C_{113} in layout $\mathcal{T}_{\text{pred}}$ are accessed during tuple materialization. In contrast, the qualifying tuples in layout $\mathcal{T}_{\text{obs}}$ are likely distributed over all pages of the larger column partition C_{112} since the tuples are not clustered by `id_test_profile`.

While this example demonstrates how a table partitioning layout optimized on the future workload can reduce the memory footprint, existing workload predictors focus solely on forecasting the future statement arrival rate [49, 74, 75, 107, 134]. Consequently, their predictions can only serve as input to index advisors but cannot be used for table partitioning advisors. This is because a table partitioning advisor needs knowledge about future assignments of parameter values in the `WHERE` clause to propose a range partition that can be pruned in the future workload. As we will see later in this chapter, for an individual statement, the arrival rate and the assignments of parameter values to host variables can be affected by different workload drift types,

which can also overlap. Therefore, a workload predictor must be able to handle workload drift types and combinations thereof at the arrival rate and assignment level.

In order to address the shortcomings of existing workload predictors, we present the OUTATIME framework that consists of two phases. While the first phase predicts the future workload, the second phase feeds the predicted workload into a physical database design advisor. Both phases are periodically repeated to ensure a continuous adaption of the physical schema. Within the workload prediction phase, we predict the future parameter values assigned to the host variables and the future arrival rate of statements. Hence, the predicted workload of our approach can be used for any physical database design advisor that relies on workload statistics. Our framework is also extensible enough to support novel workload drift types in the future.

5.2 PROBLEM STATEMENT

In this section, we formalize the problem of predicting the future workload based on the observed workload. We begin by defining the observed and future workload, and subsequently state the problem. All introduced notation of this chapter is summarized in Table 5.1. In addition, Table 5.1 displays all relevant notation from Section 2.1 to understand the definitions of this chapter, e.g., the notation of host variables and statement instantiations.

Definition 36 (Observed and Future Workload)**.** *Let $o_s, o_e, f_s, f_e \in \mathbb{N}$ be four timestamps, such that $o_s < o_e \leq f_s < f_e$. We define an observed workload as $\mathcal{O}$, such that $\forall (t, S_q, V_q) \in \mathcal{O} : o_s \leq t < o_e$, and a future workload as $\mathcal{F}$, such that $\forall (t, S_q, V_q) \in \mathcal{F} : f_s \leq t < f_e$.*

Problem 3 (Workload Prediction)**.** *The problem we consider is to find a predicted workload $\mathcal{P}$ from an observed workload $\mathcal{O}$, where $\forall (t, S_q, V_q) \in \mathcal{P} : f_s \leq t < f_e$, such that the predicted workload $\mathcal{P}$ approximates the future workload $\mathcal{F}$.*

Figure 5.1a shows an observed workload $\mathcal{O}$, where the host variable `:1` is assigned frequently by parameter values between 72,160,000 and 72,167,000. By contrast, in the future workload $\mathcal{F}$ (cf. Figure 5.2), the host variable `:1` is assigned frequently by parameter values between 72,166,000 and 72,174,000. Figure 5.1b shows then the predicted workload $\mathcal{P}$ that approximates the future workload $\mathcal{F}$.

5.3 WORKLOAD DRIFT TYPES

We now discuss the characteristics of the five workload drift types linear, exponential, reoccurring, static, and irregular that have been identified in real-world applications [57, 75, 107]. We illustrate each drift type using an example from SAP HANA's test environment.

$\mathcal{S} = \{S_1, \ldots, S_q, \ldots, S_u\}$	A set of u parameterized SQL statements.
$H_q = [h_{q1}, ..., h_{qr}, ..., h_{qw_q}]$	A vector of w_q host variables of S_q.
V_q	A vector of w_q parameter values.
(t, S_q, V_q)	A statement instantiation of S_q with vector V_q of parameter values at timestamp t, where parameter value $V_q[r]$ is assigned to h_{qr}.
$\mathcal{W}$	A workload as a set of z statement instantiations.
$\mathcal{O}, \mathcal{F}, \mathcal{P}$	The observed, future, and predicted workload.
o_s, o_e	The start and end timestamps of $\mathcal{O}$.
f_s, f_e	The start and end timestamps of $\mathcal{F}$ and $\mathcal{P}$.
δ	A discretization interval between two successive, discrete timestamps.
$\Delta(o_s, o_e)$	A set of equidistant timestamps between o_s and o_e.
$F(t, \mathcal{O}, S_q)$	The observed statement arrival frequency at t for S_q in $\mathcal{O}$.
$SAR(\mathcal{O}, S_q)$	The observed statement arrival rate for S_q in $\mathcal{O}$.
$\rho(SAR(\mathcal{O}, S_q))$	The Pearson correlation coefficient ρ applied on the observed SAR.
$\rho(log(SAR(\mathcal{O}, S_q)))$	The Pearson correlation coefficient ρ on the logarithm of the observed SAR, where log is applied on the frequencies in the observed SAR.
$\varphi_{\text{lin}}, \varphi_{\text{exp}}$	Two thresholds to detect either a linear or an exponential drift.
$DFT(SAR(\mathcal{O}, S_q))$	The discrete Fourier transform on the observed SAR.
ub_q, lb_q	The smallest and largest statement arrival frequency of S_q in $\mathcal{O}$.
θ_q	A threshold for S_q to detect a reoccurring drift based on a fixed φ_{cyc}.
$RMSE(SAR(\mathcal{O}, S_q))$	The root mean square error on the observed SAR.
μ_q	The mean statement arrival frequency for S_q in $\mathcal{O}$.
φ_{static}	A threshold to detect a static workload.
$\alpha(\mathcal{O}, h_{qr})$	The set of all assignments to h_{qr} in $\mathcal{O}$.
$\gamma \cdot \delta$	A threshold interval as a multiple of the discretization interval δ to distinguish time-dependent and time-independent assignments.
$SFA(\mathcal{O}, h_{qr})$, $SFA(\mathcal{P}, h_{qr})$	The observed (predicted) series of fresh assignments of h_{qr} in $\mathcal{O}$ ($\mathcal{P}$).
$\alpha_{sub}(\mathcal{O}, h_{qr}), \alpha_{reg}(\mathcal{O}, h_{qr})$	The set of subsequent (regular) assignments to h_{qr} in $\mathcal{O}$.
$\tau(t, V_q[r])$	The time difference between a subsequent assignment $(t, V_q[r]) \in \alpha_{sub}$ and its corresponding fresh assignment.
$Int(\kappa) = [\kappa \cdot \delta, (\kappa + 1) \cdot \delta)$	A time interval with δ as the interval length.
p_{sub}	A probability mass function for subsequent assignments.
p_{reg}	A probability mass function for regular assignments.
η	The prediction confidence factor.
ω	The observation period.

Table 5.1: Notation for workload prediction.

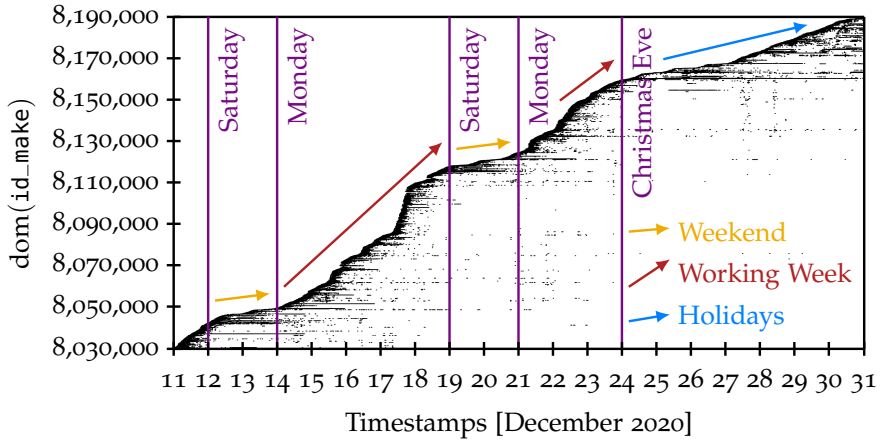


Figure 5.3: Reoccurring workload drift for parameter values of `id_make` in "`INSERT INTO install_sessions (id_make, id_server) VALUES (:1, :2)`". A single dot represents one assignment to `:1` by a certain parameter value (y-axis) at a certain time (x-axis).

5.3.1 *Linear / Exponential Workload Drift*

A linear workload drift is characterized when the statement arrival rate or parameter values assigned to a host variable increase or decrease linearly over time. For example, in Figure 5.1a, we observe linearly increasing parameter values of the domain `id_make` assigned to the host variable `:1`. Further, the statement arrival rate or parameter values assigned to a host variable can also increase or decrease exponentially over time. For example, close to the release of a new SAP HANA version, certain statements are instantiated exponentially more frequently than during regular development phases.

5.3.2 *Reoccurring Workload Drift*

Since databases often interact with humans, workloads may follow reoccurring patterns [74, 107]. For example, to test a specific SAP HANA build, the following SQL statement is executed when a test environment, called *install session*, is created on a dedicated test server:

```
INSERT INTO install_sessions (id_make, id_server)
VALUES      (:1, :2).
```

In Figure 5.3, we show all instantiations of this statement between December 11 and 31 in 2020 at a certain time (x-axis) using a specific build ID (y-axis). We observe a steep increase of `id_make` on weekdays alternated with much slighter increases on weekends, revealing a weekly pattern. In addition, we observe that `id_make` increases more moderately during Christmas than during the rest of the month, indicating an annual pattern. Since `id_make` also grows piecewise linear over time (e.g., between December 19 and 21), Figure 5.3 illustrates an example of two overlapping workload drift types.

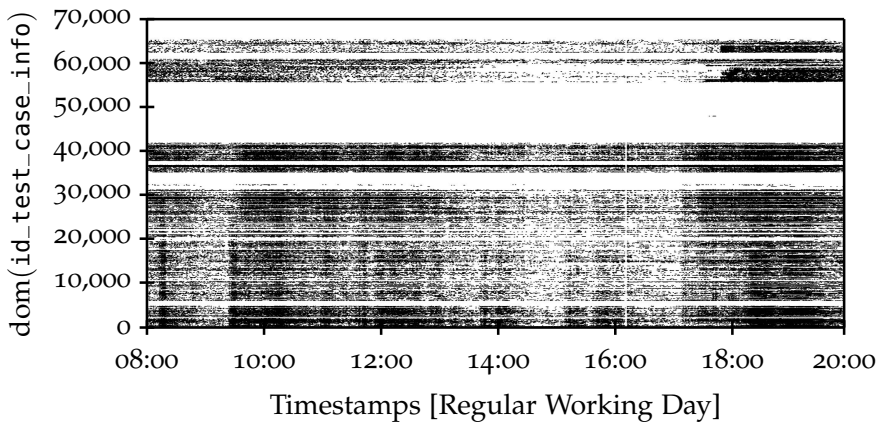


Figure 5.4: Showing a static workload for the statement "`SELECT id FROM test_cases WHERE id_test_case_info = :1`". A single dot is one assignment to the host variable `:1` by a certain parameter value (y-axis) at a certain time (x-axis).

5.3.3 *Static Workload*

A static workload is a workload where no temporal drift can be observed. For example, in SAP HANA's test environment, the dimension table `test_case_info` represents existing tests (e.g., their SQL string), such that the following SQL statement returns all test case executions with the given test case ID from the fact table `test_cases`:

```
SELECT id
FROM   test_cases
WHERE  id_test_case_info = :1.
```

In Figure 5.4, we show all instantiations of this statement between 08:00 and 20:00 on a regular working day at a certain time (x-axis) and using a specific parameter value assigned to the host variable `:1` (y-axis). No change in the parameter values can be observed during the considered time frame. This is expected as the same tests are repeatedly executed over an extended period of time to identify bugs or authorize patches.

5.3.4 *Irregular Workload Drift*

An irregular workload drift is characterized by an abrupt and unexpected change of the statement arrival rate or the parameter values assigned to a host variable [75, 107]. For example, occasional stress test campaigns in addition to regular testing lead to an irregular drift of the statement arrival rate. Unlike other workload drift types, irregular workload drifts can only be detected retrospective (e.g., after a sudden increase of the statement arrival rate) but cannot be anticipated in advance. Nonetheless, the framework presented in this chapter can handle irregular workload drifts, as we will see in Section 5.5.4.

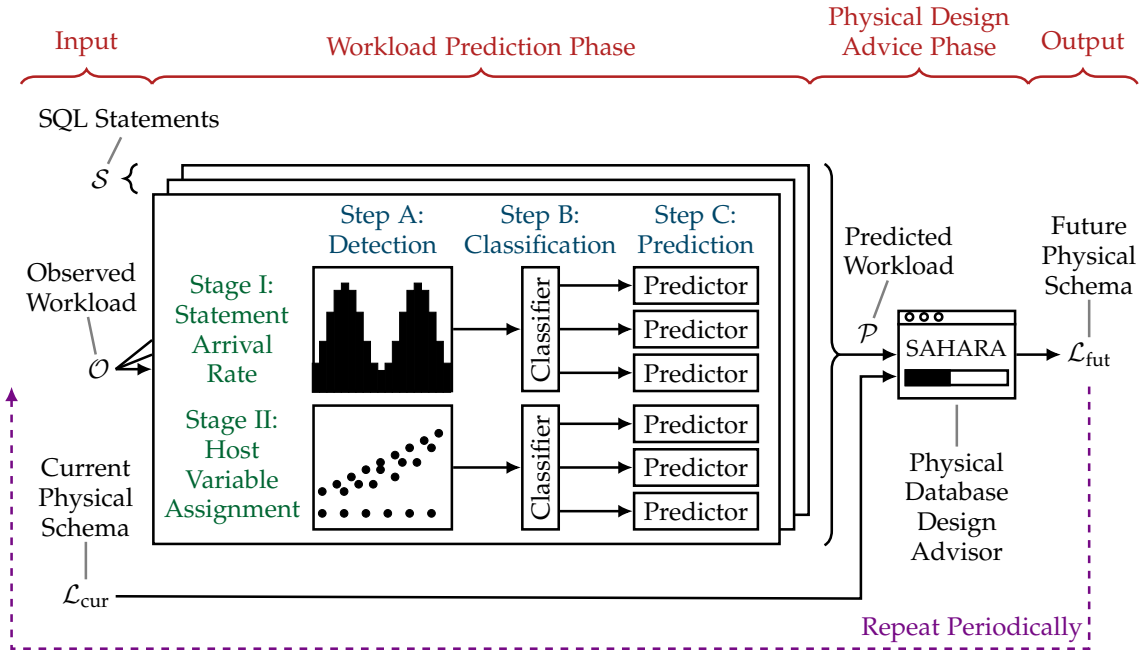


Figure 5.5: Illustration of the proposed OUTATIME framework that consists of a workload prediction and a physical database design advice phase. The prediction phase is split into a statement arrival rate prediction and an assignment prediction stage. Each stage consists of a detection, a classification, and a prediction step.

5.4 FRAMEWORK OVERVIEW

In this section, we provide an overview of our framework for workload prediction and its application to automated physical database design. In Figure 5.5, we illustrate the proposed OUTATIME framework, which consists of a workload prediction and a physical database design advice phase. In the workload prediction phase, we solve Problem 3 by predicting the future workload $\mathcal{P}$ based on the observed workload $\mathcal{O}$ on a per-statement basis. As it may be computationally infeasible to consider all statements in the observed workload for the prediction, we only utilize statements in the SQL plan cache since they represent typically more than 99% of the workload cost [109].

For each SQL statement $S_q \in \mathcal{S}$, the future workload is predicted in two independent stages. Stage I, as described in Section 5.5, predicts the statement arrival rate, whereas Stage II, as described in Section 5.6, predicts the future assignments of parameter values to host variables. We obtain the predicted workload $\mathcal{P}$ by combining the results of both stages, as described in Section 5.7. Stage I and Stage II are each composed of three steps. In Step A, we detect which workload drift types are present in the observed statement arrival rate and in the observed assignments. In Step B, we resolve conflicts between multiple detected workload drift types using a classifier. Finally, in Step C, we extrapolate the statement arrival rate and the assignments into the future using a predictor based on the outcome of the prior classification. Our

framework allows exchanging all detectors, classifiers, and predictors to support workload drifts in novel environments.

The physical database design advice phase of the OUTATIME framework, as described in Section 5.8, proposes the future physical schema $\mathcal{L}_{\text{fut}}$ with the smallest combined workload and data reorganization costs based on the predicted workload $\mathcal{P}$ and the current physical schema $\mathcal{L}_{\text{cur}}$. For example, the cost model of Section 4.7 can be used that combines workload and table repartitioning costs. In case the effort of a design change is not expected to pay off due to high data reorganization costs, the current physical schema $\mathcal{L}_{\text{cur}}$ will continue to be used. Finally, the workload prediction and physical database design advice phase are periodically repeated to adopt the physical schema in small and cheap adjustments.

5.5 STAGE I: PREDICTION OF THE STATEMENT ARRIVAL RATE

In the following, we elaborate on the statement arrival rate prediction, which is Stage I of the workload prediction phase as established in Section 5.4. In particular, we predict the future arrival rate for each statement $S_q \in \mathcal{S}$. As a prerequisite, we first discretize the workload and formally define the observed statement arrival rate (SAR) in Section 5.5.1. The prediction then consists of three steps. As Step A, we introduce conditions to detect different workload drift types in the observed SAR in Section 5.5.2. As Step B, we present a classifier to resolve conflicts between multiple detected drift types in Section 5.5.3. Lastly, as Step C, we predict the future SAR in Section 5.5.4.

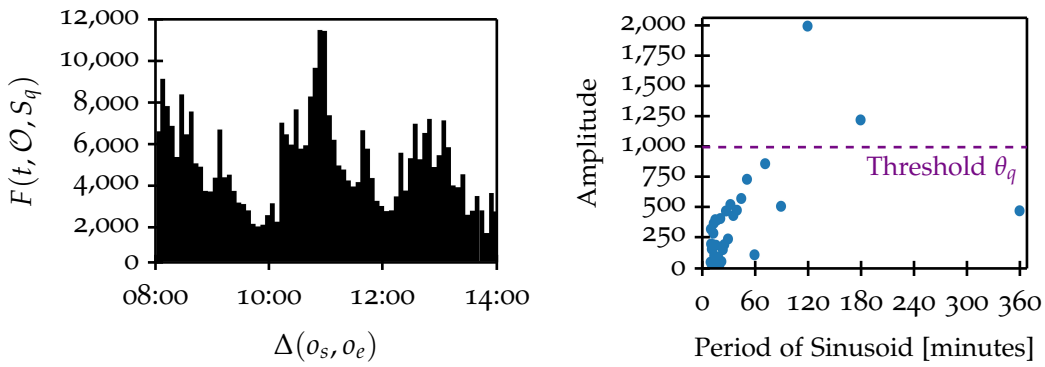
5.5.1 *Discretization*

We start by discretizing the observed workload $\mathcal{O}$ to reduce noise and other short-term fluctuations [86, 104]. For this purpose, we define a set of equidistant timestamps using a *discretization interval* $\delta \in \mathbb{N}$ between two successive, discrete timestamps. For the table partitioning advisor of Chapter 4, the discretization interval δ should be set to the time window length $\pi/2$ of the statistics collector (cf. Section 4.4) to still guarantee an advice with high quality. As our framework can be used for any physical database design advisor that relies on workload statistics, δ might be initialized differently for other advisors. For illustration purposes, we set δ to five minutes in this section.

Definition 37 (Equidistant Timestamps)**.** *We define $\Delta(o_s, o_e)$ as a set of equidistant timestamps between the start and end timestamps $o_s, o_e \in \mathbb{N}$:*

$$\Delta(o_s, o_e) := \{o_s + \kappa \cdot \delta \mid \kappa \in \mathbb{N}, o_s \leq o_s + \kappa \cdot \delta < o_e\}.$$

To calculate the observed statement arrival frequency, we aggregate all statement instantiations in the observed workload $\mathcal{O}$ that fall within interval $[t, t + \delta)$ for a timestamp $t \in \Delta(o_s, o_e)$.



(a) Observed $SAR(\mathcal{O}, S_q)$ (b) $DFT(SAR(\mathcal{O}, S_q))$

Figure 5.6: Illustration of the observed statement arrival rate $SAR(\mathcal{O}, S_q)$ of the observed workload $\mathcal{O}$ of Figure 5.1a and its discrete Fourier transform $DFT(SAR(\mathcal{O}, S_q))$.

Definition 38 (Observed Statement Arrival Frequency). *We define the observed statement arrival frequency $F(t, \mathcal{O}, S_q)$ at timestamp $t \in \Delta(o_s, o_e)$ for statement $S_q \in \mathcal{S}$ in the observed workload $\mathcal{O}$ as:*

$$F(t, \mathcal{O}, S_q) := \left| \left\{ (t', S'_q, V_q) \in \mathcal{O} \mid S'_q = S_q \wedge t \leq t' < t + \delta \right\} \right|.$$

Finally, we define the observed statement arrival rate as a series of consecutive statement arrival frequencies between the start and end timestamps $o_s, o_e \in \mathbb{N}$ in the observed workload $\mathcal{O}$.

Definition 39 (Observed Statement Arrival Rate). *We define the observed statement arrival rate (observed SAR) for statement $S_q \in \mathcal{S}$ in the observed workload $\mathcal{O}$ as:*

$$SAR(\mathcal{O}, S_q) := \left\{ (t, F(t, \mathcal{O}, S_q)) \mid t \in \Delta(o_s, o_e) \right\}.$$

In Figure 5.6a, we show the observed SAR for equidistant timestamps between 08:00 and 14:00 with a discretization interval δ of five minutes, calculated from the observed workload of Figure 5.1a. We observe a wavelike pattern with peaks at 08:30, 11:00, and 13:00.

5.5.2 *Step A: Detection*

The first step to predict the future SAR is to detect which workload drift types are present in the observed SAR. We introduce a detector as a condition based on the observed SAR for each drift type.

To detect a linear workload drift, we use the *Pearson correlation* [135] between the series of discrete timestamps and the series of statement arrival frequencies. Both series are represented in $SAR(\mathcal{O}, q_i)$. The condition is satisfied if the absolute value of the Pearson correlation coefficient ρ is greater than or equal to a threshold $\varphi_{\text{lin}} \in [0, 1]$:

$$SAR(\mathcal{O}, S_q) \text{ is } \textbf{linear} \Leftrightarrow |\rho(SAR(\mathcal{O}, S_q))| \geq \varphi_{\text{lin}}. \tag{5.1}$$

In order to spot exponential workload drifts, we take advantage of the fact that a function grows exponentially if its logarithm grows linearly. Therefore, the condition is satisfied if the absolute value of the Pearson correlation coefficient ρ between the series of discrete timestamps and the series of the natural logarithm of each statement arrival frequency is greater than or equal to a threshold $\varphi_{\text{exp}} \in [0,1]$:

$$SAR(\mathcal{O}, S_q) \text{ is } \textbf{exponential} \Leftrightarrow |\rho(ln(SAR(\mathcal{O}, S_q)))| \geq \varphi_{\text{exp}}, \quad (5.2)$$
where ln is only applied on the frequencies in $SAR(\mathcal{O}, S_q)$.

To detect a reoccurring workload drift, we first compute the *discrete Fourier transform DFT*, i.e., we convert the observed SAR from the time domain into the frequency domain [124, 169]. Afterwards, we evaluate whether a sinusoid with an amplitude greater than or equal to a threshold θ_q is present in the DFT. To specify the threshold θ_q, we first determine the smallest and largest observed statement arrival frequencies lb_q and ub_q of statement S_q in the observed workload. Given a fixed $\varphi_{\text{cyc}} \in [0,1]$, the threshold θ_q is obtained as φ_{cyc} times the largest possible amplitude range but at least φ_{cyc} times the smallest statement arrival frequency:

$$ub_q := \max_{t \in \Delta(o_s, o_e)} F(t, \mathcal{O}, S_q) \quad (5.3)$$

$$lb_q := \min_{t \in \Delta(o_s, o_e)} F(t, \mathcal{O}, S_q) \quad (5.4)$$

$$\theta_q := \varphi_{\text{cyc}} \cdot \max\left(ub_q - lb_q, lb_q\right) \quad (5.5)$$

$$SAR(\mathcal{O}, S_q) \text{ is } \textbf{reoccuring} \Leftrightarrow \exists z \in DFT(SAR(\mathcal{O}, S_q)) : |z| \geq \theta_q. \quad (5.6)$$

In Figure 5.6b, we illustrate the DFT of the observed SAR of Figure 5.6a. Each dot represents a sinusoid with a certain period (x-axis) and amplitude (y-axis). We also depict the threshold $\theta_q = 999$ calculated using the smallest (1498) and largest (11491) statement arrival frequencies and $\varphi_{\text{cyc}} = 0.1$. As can be seen, the amplitudes of two sinusoids with periods of 120 and 180 minutes exceed the threshold, which constitutes a reoccurring workload drift.

In order to identify static workloads, we examine whether the observed SAR is stable, i.e., it fluctuates only within narrow limits around the mean statement arrival frequency μ_q. Given a fixed $\varphi_{\text{static}} \in [0,1]$, the condition is satisfied if the *root mean square error (RMSE)* [9] between the observed statement arrival frequencies and μ_q is less than or equal to φ_{static} times μ_q:

$$\mu_q := \frac{\sum_{t \in \Delta(o_s, o_e)} F(t, \mathcal{O}, S_q)}{|\Delta(o_s, o_e)|} \quad (5.7)$$

$$RMSE(SAR(\mathcal{O}, S_q)) := \sqrt{\frac{\sum_{(t, F(t, \mathcal{O}, S_q)) \in SAR(\mathcal{O}, S_q)} (F(t, \mathcal{O}, S_q) - \mu_q)^2}{|\Delta(o_s, o_e)|}} \quad (5.8)$$

$$SAR(\mathcal{O}, S_q) \text{ is } \textbf{static} \Leftrightarrow RMSE(SAR(\mathcal{O}, S_q)) \leq \varphi_{\text{static}} \cdot \mu_q. \quad (5.9)$$

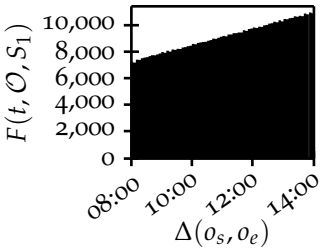
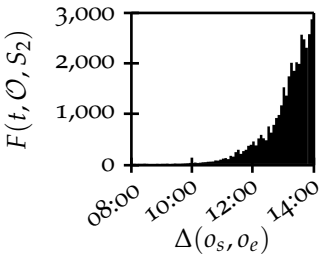
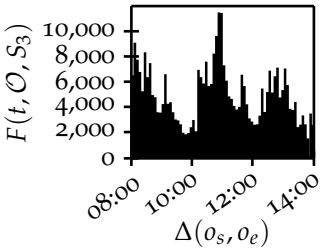
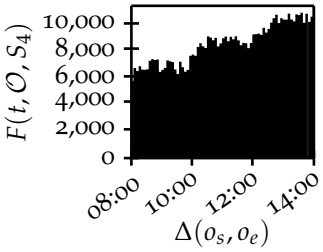
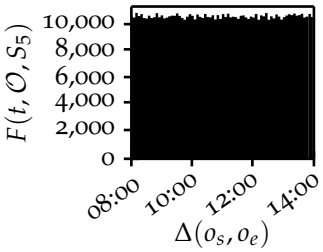
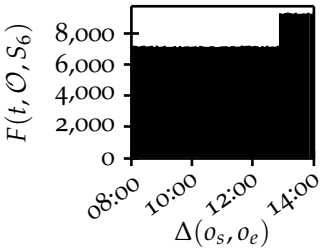
Observed SAR	Steps A & B: Detection & Classification		Step C: Prediction
	Linear (Eq. 5.1)	$\lvert 0.99 \rvert \geq 0.8$ ✓	
	Exponential (Eq. 5.2)	$\lvert 0.98 \rvert \geq 0.8$ ✓	
	Reoccurring (Eq. 5.6)	$\lvert -60 + 574i \rvert \not\geq 700$ ✗	$F(t, \mathcal{P}, S_1) = 13t + 7071$
	Static (Eq. 5.9)	$1051 \not\leq 0.1 \cdot 889$ ✗	
	Classification (Fig. 5.7)	Linear	
	Linear (Eq. 5.1)	$\lvert 0.78 \rvert \not\geq 0.8$ ✗	
	Exponential (Eq. 5.2)	$\lvert 0.99 \rvert \geq 0.8$ ✓	
	Reoccurring (Eq. 5.6)	$\lvert 554 + 598i \rvert \geq 301$ ✓	$F(t, \mathcal{P}, S_2) = e^{0.09t + 1.41}$
	Static (Eq. 5.9)	$816 \not\leq 0.1 \cdot 54$ ✗	
	Classification (Fig. 5.7)	Exponential	
	Linear (Eq. 5.1)	$\lvert 0.24 \rvert \not\geq 0.8$ ✗	$F(t, \mathcal{P}, S_3) = 4954$
	Exponential (Eq. 5.2)	$\lvert 0.23 \rvert \not\geq 0.8$ ✗	$+1216\cos(2\pi t)$
	Reoccurring (Eq. 5.6)	$\lvert -1004 + 1701i \rvert \geq 999$ ✓	$+18\sin(2\pi t)$
	Static (Eq. 5.9)	$2203 \not\leq 0.1 \cdot 495$ ✗	$-1004\cos(2\pi t)$
	Classification (Fig. 5.7)	Reoccurring	$-1701\sin(2\pi t)$
	Linear (Eq. 5.1)	$\lvert 0.95 \rvert \geq 0.8$ ✓	$F(t, \mathcal{P}, S_4) = 61t + 5801$
	Exponential (Eq. 5.2)	$\lvert 0.94 \rvert \geq 0.8$ ✓	$+185\cos(2\pi t)$
	Reoccurring (Eq. 5.6)	$\lvert 124 + 831i \rvert \geq 700$ ✓	$+135\sin(2\pi t)$
	Static (Eq. 5.9)	$1337 \not\leq 0.1 \cdot 796$ ✗	$-239\cos(2\pi t)$
	Classification (Fig. 5.7)	Linear & Reoccuring	$-317\sin(2\pi t)$
	Linear (Eq. 5.1)	$\lvert 0.05 \rvert \not\geq 0.8$ ✗	
	Exponential (Eq. 5.2)	$\lvert 0.05 \rvert \not\geq 0.8$ ✗	
	Reoccurring (Eq. 5.6)	$\lvert -41 + 33i \rvert \not\geq 1000$ ✗	$F(t, \mathcal{P}, S_5) = 10690$
	Static (Eq. 5.9)	$132 \leq 0.1 \cdot 10250$ ✓	
	Classification (Fig. 5.7)	Static	
	Linear (Eq. 5.1)	$\lvert 0.69 \rvert \not\geq 0.8$ ✗	
	Exponential (Eq. 5.2)	$\lvert 0.69 \rvert \not\geq 0.8$ ✗	$F(t, \mathcal{P}, S_6) =$
	Reoccurring (Eq. 5.6)	$\lvert 164 + 597i \rvert \not\geq 700$ ✗	$F(t \bmod 72, \mathcal{O}, S_6)$
	Static (Eq. 5.9)	$829 \not\leq 0.1 \cdot 7457$ ✗	
	Classification (Fig. 5.7)	Irregular	

Table 5.2: The detection and classification of the observed SAR for six SQL statements depend on four conditions and the decision DAG of Figure 5.7. The outcome of the predicted statement arrival frequencies is then based on the classification.

Since irregular workload drifts by definition are abrupt and unexpected, we argue that there can be no condition that handles irregular drifts. As a consequence, in Section 5.5.3, we will classify the workload as irregular if none of the above conditions are met.

To demonstrate how workload drifts can be detected using the conditions we proposed, Table 5.2 shows for six SQL statements the observed SAR (first column) and whether the conditions are satisfied (✓) or not (✗) (second column). For the condition of a reoccurring drift, we present the sinusoid with the largest amplitude. As in our experimental evaluation in Section 5.9, we set the parameters as follows: $\varphi_{\text{lin}} = 0.8$, $\varphi_{\text{exp}} = 0.8$, $\varphi_{\text{cyc}} = 0.1$, and $\varphi_{\text{static}} = 0.1$. We observe that in some cases multiple drift types are detected. For example, statement S_2 fulfills the condition of an exponential and reoccurring drift. Next, we discuss how these cases can be handled.

5.5.3 *Step B: Classification*

The second step towards predicting the future SAR is to resolve conflicts between multiple detected workload drift types, such that the observed SAR is classified as one drift type or a combination thereof. As classifier, we propose the decision DAG shown in Figure 5.7: If *none* of the conditions are satisfied, we classify the observed SAR as irregular (e.g., S_6 in Table 5.2). If *exactly one* drift type is detected, we classify the observed SAR as the detected drift type. If *two or more* drift types are detected, we distinguish compatible and incompatible drift combinations. We consider the combination of linear and reoccurring as compatible (e.g., S_4 in Table 5.2). In contrast, since an exponential drift asymptotically subsumes a linear drift, both are an incompatible combination. In this case, we prefer the drift type with

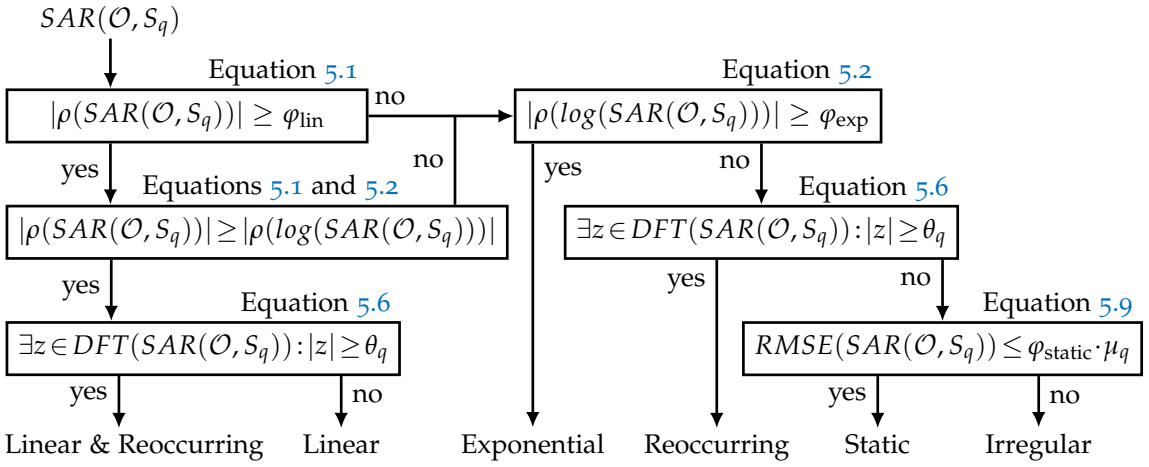


Figure 5.7: A decision DAG as a classifier to determine a workload drift type or combination thereof for an observed SAR. The observed SAR is classified either as linear, exponential, reoccurring, linear and reoccurring, static, or irregular.

the stronger Pearson correlation coefficient ρ. Further, an exponential drift is preferred over a reoccurring drift because the exponential drift may appear as a sinusoid in the DFT (e.g., S_2 in Table 5.2).

5.5.4 *Step C: Prediction*

As the final step, we predict the future SAR using the observed SAR and the outcome of the prior classification. To do this, we calculate the predicted statement arrival frequency $F(t, \mathcal{P}, S_q)$ at future timestamps $t \in \Delta(f_s, f_e)$ for each SQL statement $S_q \in \mathcal{S}$. Similar as in Definition 39, the predicted $SAR(\mathcal{P}, S_q)$ is then obtained as the series of consecutive statement arrival frequencies between f_s and f_e.

The prediction is based on the classification of the observed SAR:

LINEAR: We use a linear approximation function obtained by ordinary least-squares regression on the observed SAR [55].

EXPONENTIAL: We first perform ordinary least-squares regression on the natural logarithm of the observed SAR. The future SAR is calculated by raising e to the power of the prediction produced by the linear approximation function.

REOCCURRING: We convert all sinusoids with an amplitude greater or equal than threshold θ_q (cf. Equation 5.5) into a sum of trigonometric functions in the time domain. Sinusoids with a small amplitude are filtered out because they may result from short-term fluctuations or noise and would therefore lead to overfitting.

STATIC: We use the mean observed statement arrival frequency μ_q (cf. Equation 5.7) to predict future statement arrival frequencies.

IRREGULAR: Since irregular drifts can neither be modeled nor predicted, we cannot pretend to know the future. Instead, we copy the observed SAR and paste it into the future, similar to a backward-looking approach.

LINEAR AND REOCCURRING: We compute a sum of a linear function and trigonometric functions. The linear function is obtained by ordinary least-squares regression on the observed SAR. We then calculate a *normalized* observed SAR by subtracting the linear function from the observed SAR. Finally, we obtain the trigonometric functions by transforming all sinusoids in the DFT of the normalized SAR with an amplitude greater or equal than threshold θ_q (cf. Equation 5.5).

Since the overhead of our workload predictor (cf. Section 5.9) and our table partitioning advisor (cf. Section 4.8) is low, we run the workload predictor and the advisor periodically, as illustrated in

Figure 5.5. As a consequence, our framework responds timely to the *outcome* of an irregular drift and thus is able to handle irregular drifts.

In the last column in Table 5.2, we show the predicted statement arrival frequency at future discrete timestamps $t \in \Delta(f_s, f_e)$ for all six observed SARs. Because of a 6-hour observation window and a choice of 5 minutes as discretization interval δ, the timestamps of $\mathcal{O}$ range between 0 and 71, while timestamps of $\mathcal{P}$ begin at 72.

5.6 STAGE II: PREDICTION OF THE HOST VARIABLE ASSIGNMENT

We now detail the prediction of assignments to all host variables, which is Stage II of the workload prediction phase, as introduced in Section 5.4. For pragmatic reasons, we predict the assignments to one host variable independently of other host variables. To predict the future assignments of parameter values to each host variable, we first define a set of all assignments to a host variable in the observed workload.

Definition 40 (Host Variable Assignments)**.** *We define $\alpha(\mathcal{O}, h_{qr})$ as the set of all assignments to host variable h_{qr} in the observed workload $\mathcal{O}$ by certain parameter values at certain timestamps t:*

$$\alpha(\mathcal{O}, h_{qr}) := \{(t, V_q[r]) \mid (t, S_q, V_q) \in \mathcal{O}, V_q[r] \textit{ is assigned to } h_{qr}\}.$$

To illustrate the main idea of this stage, let us consider Figure 5.8, which is an abstract representation of the observed workload in Figure 5.1a. Each dot represents a single assignment $(t, V_q[r]) \in \alpha(\mathcal{O}, h_{qr})$ of parameter value $V_q[r]$ (y-axis) to host variable h_{qr} at timestamp t (x-axis). For now, it is sufficient to understand that an assignment is classified as UNCERTAIN if it occurs close to the beginning of the observation period, as REGULAR if $V_q[r]$ is regularly assigned to h_{qr}, as FRESH if $V_q[r]$ has not been assigned yet to h_{qr} within a recent time interval, and as SUBSEQUENT otherwise.

In Figure 5.8, the fresh assignments follow a linear drift. Furthermore, subsequent assignments frequently occur shortly after fresh assignments and rarely further on. Finally, the regular assignments are distributed across the entire observed workload.

The prediction of assignments for the future workload consists of the following steps: As a combined Step A and B, we classify each assignment to a host variable in the observed workload as uncertain, regular, fresh, or subsequent (cf. Section 5.6.1). As Step C, we extrapolate the series of fresh assignments into the future, model the distribution of subsequent assignments in relation to their corresponding fresh assignments, and model the distribution of regular assignments (cf. Section 5.6.2). Both models are used in Section 5.7 to predict future subsequent and regular assignments.

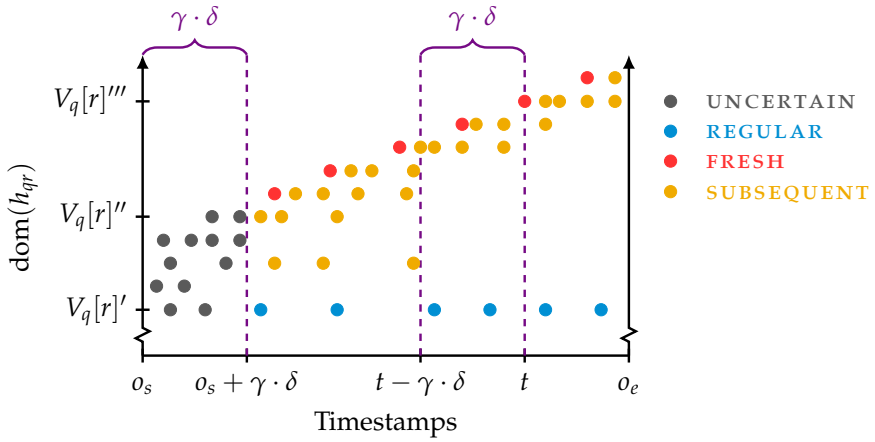


Figure 5.8: Illustration of the classification of assignments $(t, V_q[r]) \in \alpha(\mathcal{O}, h_{qr})$ to host variable h_{qr} in the observed workload $\mathcal{O}$ by certain parameter values $V_q[r]$ (y-axis) at certain timestamps t (x-axis) as uncertain, regular, fresh, and subsequent.

5.6.1 *Steps A and B: Detection and Classification*

In order to classify an assignment $(t, V_q[r]) \in \alpha(\mathcal{O}, h_{qr})$, we pick a *threshold interval* of length $\gamma \cdot \delta$, with $\gamma \in \mathbb{N}$, as a multiple of the discretization interval $\delta \in \mathbb{N}$ (cf. Section 5.5.1). As we will see in a moment, the threshold interval $\gamma \cdot \delta$ is used to distinguish time-dependent and time-independent assignments.

Definition 41 (Assignment Classification)**.** *Given a threshold interval of length $\gamma \cdot \delta$, we classify an assignment $(t, V_q[r]) \in \alpha(\mathcal{O}, h_{qr})$ as*

$$\begin{cases} \text{UNCERTAIN} & \text{if } o_s \leq t < o_s + \gamma \cdot \delta \\ \text{REGULAR} & \text{else if } \forall \widehat{t} \in [o_s, o_e - \gamma \cdot \delta)\ \exists (t', V_q[r]) \in \alpha(\mathcal{O}, h_{qr}) : \\ & \qquad \widehat{t} \leq t' < \widehat{t} + \gamma \cdot \delta \\ \text{FRESH} & \text{else if } \nexists (t', V_q[r]) \in \alpha(\mathcal{O}, h_{qr}) : t - \gamma \cdot \delta \leq t' < t \\ \text{SUBSEQUENT} & \text{else.} \end{cases}$$

We argue that assignments, where the same parameter value is assigned to the same host variable at least once every $\gamma \cdot \delta$ time units, are independent of the time and thus part of the static workload. Such assignments are classified as REGULAR. To illustrate, in Figure 5.8, this is the case for parameter value $V_q[r]'$. In contrast, the absence of a repeated assignment of a parameter value every $\gamma \cdot \delta$ time units implies that the assignment is time-dependent and thus part of the workload drift. If the parameter value in such assignments is first assigned after $\gamma \cdot \delta$ time units, we classify the assignment as FRESH, otherwise (second, third, etc.) as SUBSEQUENT. For example, in Figure 5.8, the assignment of $V_q[r]'''$ at timestamp t is fresh as $V_q[r]'''$ was not assigned in the previous $\gamma \cdot \delta$ time units, whereas later assignments

of $V_q[r]'''$ are subsequent. Finally, we classify assignments in the interval $[o_s, o_s + \gamma \cdot \delta)$ as UNCERTAIN since we cannot exclude the possibility that fresh assignments in this interval are subsequent assignments that follow fresh assignments before o_s. Including potentially subsequent assignments into the extrapolation of fresh assignments (cf. Section 5.6.2.1) would deteriorate the prediction quality.

5.6.2 *Step C: Prediction*

We now predict the future fresh assignments and model the distribution of subsequent and regular assignments. The predicted series of fresh assignments and both models are then used in Section 5.7 to predict the future workload.

5.6.2.1 *Fresh Assignments*

In order to predict the future fresh assignments, we first define a series of all fresh-classified assignments in the observed workload.

Definition 42 (Observed Series of Fresh Assignments)**.** *We define the observed series of fresh assignments (observed SFA) of host variable h_{qr} in the observed workload $\mathcal{O}$ as the set of all fresh assignments in $\alpha(\mathcal{O}, h_{qr})$:*

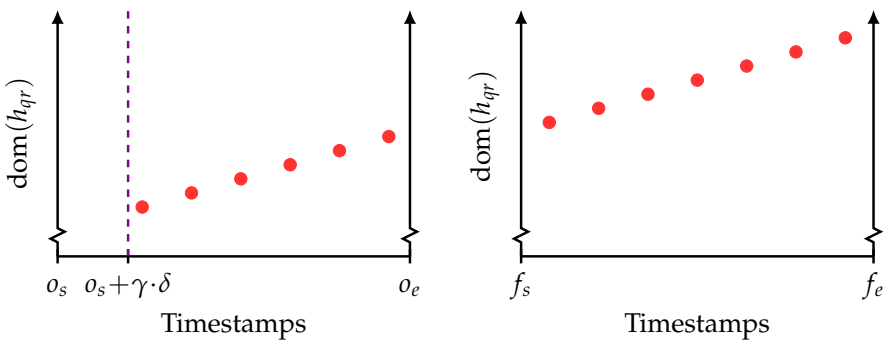
$$SFA(\mathcal{O}, h_{qr}) := \{(t, V_q[r]) \mid (t, V_q[r]) \in \alpha(\mathcal{O}, h_{qr}), (t, V_q[r]) \text{ is fresh}\}.$$

We then predict the series of future fresh assignments as follows: First, we detect which workload drift types are present in the observed SFA, similar to the described techniques in Section 5.5.2. Second, we use our classifier (cf. Figure 5.7) to resolve conflicts between multiple detected drift types, similar to Section 5.5.3. Finally, we predict the future series of fresh assignments $SFA(\mathcal{P}, h_{qr})$ based on the observed series of fresh assignments $SFA(\mathcal{O}, h_{qr})$ and the outcome of the prior classification, like in Section 5.5.4.

To give an example, in Figure 5.9a, we show the observed series of fresh assignments $SFA(\mathcal{O}, h_{qr})$ from the fresh classified assignments of Figure 5.8. We can observe that the fresh assignments follow a linear drift. As a consequence, the future series of fresh assignments $SFA(\mathcal{P}, h_{qr})$ is predicted by a linear approximation function obtained by ordinary least-squares regression on $SFA(\mathcal{O}, h_{qr})$. We depict the predicted series of fresh assignments $SFA(\mathcal{P}, h_{qr})$ in Figure 5.9b.

5.6.2.2 *Subsequent Assignments*

We model the distribution of subsequent assignments of parameter values in relation to their corresponding fresh assignments. In Section 5.7, we use this model to predict future subsequent assignments in relation to their predicted fresh assignments. We now formalize the time difference between a subsequent assignment and its corresponding fresh assignment as the temporal offset between the two.



(a) $SFA(\mathcal{O}, h_{qr})$ (b) $SFA(\mathcal{P}, h_{qr})$

Figure 5.9: Illustration of the observed series of fresh assignments in (a) and the predicted series of fresh assignments in (b).

Definition 43 (Time Difference). *Let $\alpha_{sub}(\mathcal{O}, h_{qr})$ be the set of subsequent assignments to host variable h_{qr} in the observed workload $\mathcal{O}$. The time difference $\tau(t, V_q[r])$ between a subsequent assignment $(t, V_q[r]) \in \alpha_{sub}$ and its corresponding fresh assignment is defined as:*

$$\tau(t, V_q[r]) := \min\left(\left\{t - t' \mid (t', V_q[r]) \in \alpha(\mathcal{O}, h_{qr}), (t', V_q[r]) \text{ is fresh}, t' < t\right\}\right).$$

Note that for the same parameter value, more than one fresh assignment can exist. This may happen for example with a reoccurring drift. In order to handle such ambiguities, we only consider the temporal offset to the most recent corresponding fresh assignment.

Another special case is when no fresh assignment exists for a subsequent assignment because the fresh assignment occurred at the beginning of the observed workload and was classified as uncertain. For example, in Figure 5.8, this happens for $V_q[r]''$. In such cases, we estimate a fresh assignment by extrapolating the observed SFA into the past using the techniques described in Section 5.6.2.1.

We now build a probability mass function that models the probability of time differences between a subsequent assignment and its corresponding fresh assignment. In Section 5.7, this model will be used to predict future subsequent assignments in relation to predicted fresh assignment. We model the time difference in terms of time intervals $Int(\kappa) = [\kappa \cdot \delta, (\kappa + 1) \cdot \delta), \kappa \in \mathbb{N}$. We pick δ as the interval length only for the pragmatic reason that the observed workload has been discretized in steps of length δ (cf. Section 5.5).

Definition 44 (Probability Mass Function for Subsequent Assignments). *We define a probability mass function p_{sub} as the probability that the time difference $\tau(t, V_q[r])$ between a subsequent assignment of parameter value $V_q[r]$ to host variable h_{qr} and its fresh assignment falls into $Int(\kappa)$:*

$$p_{sub}(h_{qr}, \kappa) := \frac{|\{(t, V_q[r]) \mid (t, V_q[r]) \in \alpha_{sub}(\mathcal{O}, h_{qr}), \tau(t, V_q[r]) \in Int(\kappa)\}|}{|\alpha_{sub}(\mathcal{O}, h_{qr})|}.$$

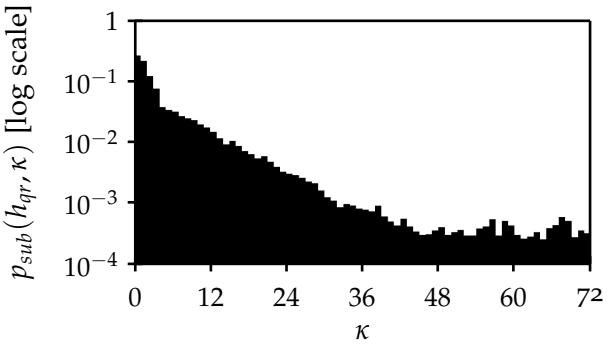


Figure 5.10: Illustration of the probability (y-axis) for time differences between subsequent assignments and their corresponding fresh assignments in the interval $Int(\kappa)$ (x-axis), calculated from assignments in the observed workload in Figure 5.1a.

In Figure 5.10, we illustrate the probability mass function p_{sub} for the observed workload in Figure 5.1a as a bar plot. For time intervals $Int(\kappa), \kappa \in [0, 72)$ (x-axis) the corresponding probability $p_{sub}(h_{qr}, \kappa)$ is depicted on the y-axis. We observe that larger time differences between subsequent assignments and their corresponding fresh assignments are associated with smaller probabilities.

5.6.2.3 *Regular Assignments*

Finally, we model the distribution of regular assignments in terms of a probability mass function. Using this model, we predict the future regular assignments in Section 5.7.

Definition 45 (Probability Mass Function for Regular Assignments)**.** *Let $\alpha_{reg}(\mathcal{O}, h_{qr})$ be the set of regular assignments to host variable h_{qr} in the observed workload $\mathcal{O}$. We define a probability mass function p_{reg} as the probability of a regular assignment of parameter value $V_q[r]$ to host variable h_{qr}:*

$$p_{reg}(h_{qr}, V_q[r]) := \frac{|\{(t, V_q[r]) \mid (t, V_q[r]) \in \alpha_{reg}(\mathcal{O}, h_{qr})\}|}{|\alpha_{reg}(\mathcal{O}, h_{qr})|}.$$

Our model is based on the following two assumptions: First, regular assignments of a parameter value to a host variable in the observed workload will continue in the future workload. Second, regular assignments are uniformly distributed. To motivate both assumptions, let us consider Figure 5.1a where parameter values of `id_test_profile` between 72,154,000 and 72,159,000 belong to background jobs which check the flakiness of tests. The goal of these periodic jobs is to ensure that all tests are stable and produce deterministic results. Finally, our framework is extensible enough to model the distribution of regular assignments with any other parameterized distribution.

5.7 WORKLOAD PREDICTION

In this section, we generate the predicted workload $\mathcal{P}$ by combining the results of the statement arrival rate prediction (Stage I) and the prediction of assignments to each host variable (Stage II). In particular, we do this by describing an algorithm that computes the predicted workload $\mathcal{P}$. The algorithm expects as input for each SQL statement $S_q \in \mathcal{S}$ the predicted statement arrival rate $SAR(\mathcal{P}, S_q)$ (cf. Section 5.5.4), and for each host variable h_{qr} in SQL statement S_q the predicted series of fresh assignments $SFA(\mathcal{P}, h_{qr})$ (cf. Section 5.6.2.1) as well as the probability mass functions $p_{sub}(h_{qr}, \kappa)$ (cf. Section 5.6.2.2) and $p_{reg}(h_{qr}, V_q[r])$ (cf. Section 5.6.2.3).

Algorithm 5 shows the calculation of the predicted workload $\mathcal{P}$. We first initialize $\mathcal{P}$ as an empty set (cf. Line 1). Next, we iterate over all SQL statements $S_q \in \mathcal{S}$ (cf. Line 2) and all future equidistant timestamps $t \in \Delta(f_s, f_e)$ (cf. Line 3). We then derive the number of statement instantiations $F(t, \mathcal{P}, S_q)$ to be predicted in the current interval $[t, t + \delta)$ from the predicted SAR, where $SAR(\mathcal{P}, S_q)[t]$ denotes the statement arrival frequency at timestamp t (cf. Line 4). For each statement instantiation to predict, we draw a random timestamp t_{rand} uniformly from $[t, t + \delta)$ (cf. Lines 5 and 6) and allocate a vector V_q with a capacity of w_q parameter values (cf. Line 7). Finally, we iterate over all host variables h_{qr} to predict their assignment by parameter values $V_q[r]$ (cf. Lines 8 to 20) and add the predicted statement instantiation $(t_{\text{rand}}, S_q, V_q)$ to the predicted workload $\mathcal{P}$ (cf. Line 21).

We assume that the distribution of fresh, subsequent, and regular assignments in the observed and future workloads is identical. Therefore, to predict the future parameter value $V_q[r]$ that is assigned to host variable h_{qr} at timestamp t_{rand}, we first draw an assignment category FRESH, SUBSEQUENT, or REGULAR with the same probabilities as they occur in $\alpha(\mathcal{O}, h_{qr})$ (cf. Line 9). For example, the probability of generating a subsequent assignment is $|\alpha_{sub}(\mathcal{O}, h_{qr})| / (|\alpha(\mathcal{O}, h_{qr})| - |\alpha_{unc}(\mathcal{O}, h_{qr})|)$, where α_{sub} denotes the set of subsequent assignments to host variable h_{qr} in the observed workload $\mathcal{O}$ and α_{unc} is the set of uncertain assignments in $\alpha(\mathcal{O}, h_{qr})$.

The prediction of parameter value $V_q[r]$ that is assigned to host variable h_{qr} at the future timestamp t_{rand} depends on which assignment category is drawn:

FRESH: We generate a fresh assignment with parameter value $V_q[r]$ set to the value of the predicted series of fresh assignments $SFA(\mathcal{P}, h_{qr})$ at timestamp t_{rand} (cf. Line 12).

SUBSEQUENT: The timestamp and parameter value of subsequent assignments are modeled in terms of their corresponding fresh assignments (cf. Definition 44). We generate a subsequent assignment in two steps: In a first step, we draw a time difference t_{diff} between the subsequent assignment to generate at

Algorithm 5: Workload Prediction

Input: For each SQL statement $S_q \in \mathcal{S}$:
– predicted statement arrival rate $SAR(\mathcal{P}, S_q)$
For each host variable h_{qr} in statement S_q:
– predicted series of fresh assignments $SFA(\mathcal{P}, h_{qr})$
– probability mass functions p_{sub} and p_{reg}

1 $\mathcal{P} := \{\}$ // initialize the predicted workload
2 **for** $1 \leq q \leq u$ **do** // iterate over SQL statement indexes
3 **for** $t \in \Delta(f_s, f_e)$ **do** // iterate over future equidistant timestamps
4 $F(t, \mathcal{P}, S_q) := SAR(\mathcal{P}, S_q)[t]$ // get statement arrival frequency
5 **for** $1 \ldots F(t, \mathcal{P}, S_q)$ **do** // iterate over statement instantiations
6 draw a random timestamp $t_{\text{rand}} \in [t, t + \delta)$ // uniform
7 allocate vector V_q with a capacity of w_q parameter values
8 **for** $1 \leq r \leq w_q$ **do** // iterate over host variable indexes
9 draw assignment category FRESH, SUBSEQUENT, or REGULAR according to their distribution in $\alpha(\mathcal{O}, h_{qr})$
10 **switch** *assignment category* **do**
11 **case** FRESH **do** // generate fresh assignment
12 $V_q[r] := SFA(\mathcal{P}, h_{qr})[t_{\text{rand}}]$ // get value from *SFA* at t_{rand}
13 **case** SUBSEQUENT **do** // generate subsequent assignment
14 draw an interval $[\kappa \cdot \delta, (\kappa + 1) \cdot \delta)$ according to p_{sub}
15 draw a random time difference $t_{\text{diff}} \in [\kappa \cdot \delta, (\kappa + 1) \cdot \delta)$
16 $t_{\text{fresh}} := t_{\text{rand}} - t_{\text{diff}}$ // calculate time of fresh assignment
17 $V_q[r] := SFA(\mathcal{P}, h_{qr})[t_{\text{fresh}}]$ // get value from *SFA* at t_{fresh}
18 **case** **regular** **do** // generate regular assignment
19 $V_q[r] :=$ draw parameter value according to p_{reg}
20 insert parameter value $V_q[r]$ into vector V_q
21 $\mathcal{P} = \mathcal{P} \cup \{(t_{\text{rand}}, S_q, V_q)\}$ // add predicted stmt. instantiation

timestamp t_{rand} and its corresponding fresh assignment at t_{fresh}. More specifically, we first draw an interval $[\kappa \cdot \delta, (\kappa + 1) \cdot \delta)$ according to the probability given by p_{sub} (cf. Line 14). Next, we draw the time difference t_{diff} uniformly from this interval (cf. Line 15), and calculate t_{fresh} as difference between t_{rand} and t_{diff} (cf. Line 16). In a second step, we generate a subsequent assignment with parameter value $V_q[r]$ set to the value of the predicted series of fresh assignments $SFA(\mathcal{P}, h_{qr})$ at t_{fresh} (cf. Line 17).

REGULAR: We generate a regular assignment with a parameter value drawn according to the probability mass function p_{reg} (cf. Line 19).

In Figure 5.11, we illustrate the prediction of regular, fresh, and subsequent assignments to host variable h_{qr} at a random future timestamp $t_{rand} \in [t, t + \delta)$. The x-axis represents the future time frame $[f_s, f_e)$ while the y-axis denotes the domain of h_{qr}. Further, we show the predicted series of fresh assignments $SFA(\mathcal{P}, h_{qr})$. We obtain the parameter value of a FRESH assignment $\overline{V_q[r]}$ from the predicted series of fresh assignments at timestamp t_{rand}. In case a SUBSEQUENT assignment is predicted, we illustrate as a first step the time

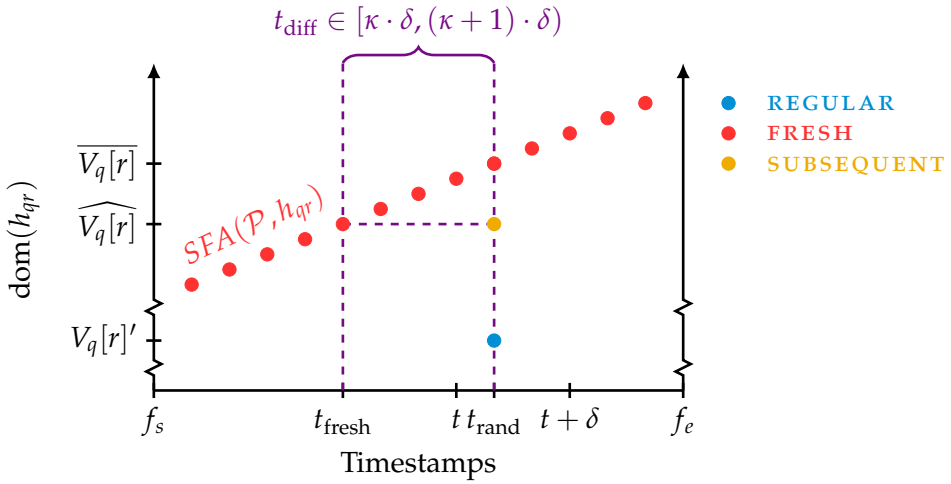


Figure 5.11: Illustration of how Algorithm 5 predicts a parameter value (y-axis) that is assigned to host variable h_{qr} at a random future timestamp t_{rand} (y-axis) for each of the three assignment categories regular, fresh, and subsequent.

difference t_{diff} drawn between t_{fresh} and t_{rand}. As a second step, the parameter value of the subsequent assignment $\widehat{V_q[r]}$ is obtained from the predicted series of fresh assignments at timestamp t_{fresh}. Finally, we show the parameter value $V_q[r]'$ drawn for a REGULAR assignment. Note that the parameter value $V_q[r]'$ is the identical parameter value that is already regularly assigned to h_{qr} in Figure 5.8.

5.8 PHYSICAL DATABASE DESIGN ADVICE PHASE

In the second phase of our framework, as described in Section 5.4, we feed the predicted workload $\mathcal{P}$ from the previous section into a physical database design advisor such as the table partitioning advisor SAHARA from Chapter 4.

As SAHARA minimizes the total memory costs $\mathcal{M}_{total}$ of an arbitrary workload $\mathcal{W}$ (cf. Definition 35), we now feed the predicted workload $\mathcal{P}$ into $\mathcal{M}_{total}$, such that a future physical schema $\mathcal{L}_{\text{fut}}$ is proposed that minimizes the total memory costs $\mathcal{M}_{total}$ of $\mathcal{P}$:

$$\mathcal{M}_{total}(C_{ijk}, \mathcal{P}, \mathcal{L}_{\text{cur}}) := \mathcal{M}_{workload}(C_{ijk}, \mathcal{P}) + \mathcal{M}_{repart}(C_{ijk}, \mathcal{L}_{\text{cur}}).$$

Note that the total memory costs $\mathcal{M}_{total}$ are the combined workload and table repartitioning costs. A natural question arises: *At which timestamp in the future is a table repartitioning beneficial?* In general, table repartitioning costs can be seen as a mortgage that needs to be amortized by the benefit in terms of workload costs in a future physical schema. The longer the duration of the predicted workload (i.e., $f_e - f_s$), the more likely the table repartitioning costs $\mathcal{M}_{repart}$ (cf. Definition 34) are amortized. On the contrary, the longer the prediction

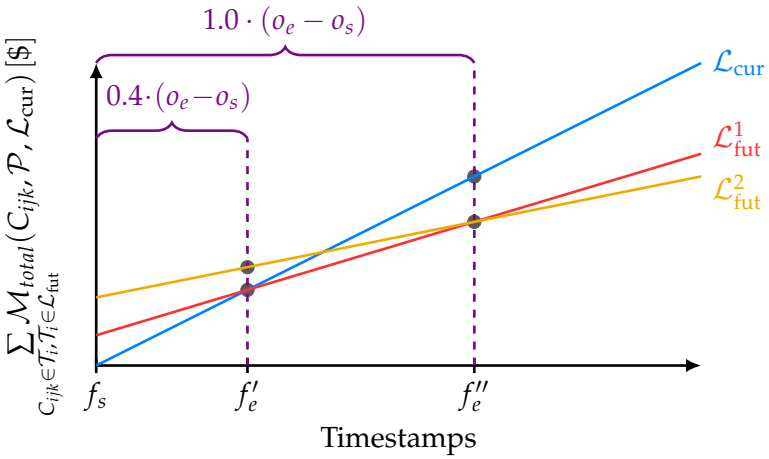


Figure 5.12: Impact of the prediction confidence factor η on the choice of the future physical schema according to $\mathcal{M}_{total}$.

horizon, the less accurate the prediction becomes. Thus, we may avoid table repartitioning if the timestamp at which table repartitioning is beneficial is too far in the future. To resolve this conflict, we calculate the duration of the predicted workload as a *prediction confidence factor* $\eta \in \mathbb{R}$ times an *observation period* $\omega \in \mathbb{N}$ (i.e., $o_e - o_s$):

$$o_s \coloneqq \text{current timestamp} \tag{5.10}$$

$$o_e \coloneqq o_s + \omega \tag{5.11}$$

$$f_s \coloneqq o_e \tag{5.12}$$

$$f_e \coloneqq f_s + \eta \cdot \omega. \tag{5.13}$$

In Figure 5.12, we illustrate the impact of the prediction confidence factor η on the choice of a future physical schema $\mathcal{L}_{fut}$. The x-axis shows a future start timestamp f_s and two future end timestamps f'_e and f''_e depending on the choice of η. The total memory costs in \$ for the current physical schema $\mathcal{L}_{cur}$ and two future physical schemata $\mathcal{L}^1_{fut}$ and $\mathcal{L}^2_{fut}$ are shown on the y-axis. The costs for both future physical schemata are greater than \$0 at the y-axis intercept due to table repartitioning costs $\mathcal{M}_{repart}$. We observe that the current physical schema $\mathcal{L}_{cur}$ is optimal for a short-time prediction as the table repartitioning costs of both future physical schemata are not amortized yet. In contrast, the future physical schema $\mathcal{L}^1_{fut}$, respectively $\mathcal{L}^2_{fut}$, become optimal for a mid-term, respectively long-term prediction. In Section 5.9, we experimentally evaluate how the prediction confidence factor η and observation period ω should be chosen.

We finally describe the integration of the table partitioning advisor SAHARA (cf. Chapter 4) into the proposed OUTATIME framework of this chapter. For this, we assume a fixed observation period $\omega \in \mathbb{N}$ and a fixed prediction confidence factor $\eta \in \mathbb{R}$ as prerequisites. In Algorithm 6, we start by initializing $\mathcal{L}_{cur}$ as the current physical schema and o_s as the current timestamp (cf. Lines 1 and 2). We then start

Algorithm 6: Execution of the OUTATIME framework

Global Variables: – observation period $\omega \in \mathbb{N}$
– prediction confidence factor $\eta \in \mathbb{R}$

1 $\mathcal{L}_{cur} :=$ current physical schema
2 $o_s :=$ current timestamp `// initialize start timestamp of observation`
3 **start** OUTATIME
4 **while** *OUTATIME is executed* **do**
5 $o_e := o_s + \omega$ `// set end timestamp of observation based on` ω
6 wait until timestamp o_e `// wait until end of observation`
7 generate observed workload $\mathcal{O}$ based on o_s and o_e
8 $f_s := o_e$ `// set start timestamp of prediction`
9 $f_e := f_s + \eta \cdot \omega$ `// set end timestamp of prediction based on` η `and` ω
10 predict workload $\mathcal{P}$ based on $\mathcal{O}$, f_s and f_e `// apply Algorithm 5`
11 $\mathcal{L}_{fut} :=$ feed SAHARA with $\mathcal{P}$ and the current physical schema $\mathcal{L}_{cur}$
12 $\mathcal{L}_{cur} := \mathcal{L}_{fut}$ `// switch to the proposed physical schema` $\mathcal{L}_{fut}$
13 $o_s := o_s + \eta \cdot \omega$ `// update start timestamp of observation`

the OUTATIME framework (cf. Line 3) and describe one iteration of the framework (cf. Lines 4 to 13). We begin by calculating the end timestamp of the observed workload o_e as $o_s + \omega$ and wait until that time to generate the observed workload $\mathcal{O}$ (cf. Lines 5 to 7). We then calculate the future start and end timestamps f_s and f_e, as described in Equations 5.12 and 5.13, based on the prediction confidence factor η and the observation period ω (cf. Lines 8 and 9). This allows using Algorithm 5 to generate the predicted workload $\mathcal{P}$ (cf. Line 10). Next, we feed the predicted workload $\mathcal{P}$ and the current physical schema $\mathcal{L}_{cur}$ into SAHARA and switch to the proposed future physical schema $\mathcal{L}_{fut}$ (cf. Lines 11 and 12). Note that the current physical schema $\mathcal{L}_{cur}$ could also be considered as the best alternative. Finally, we increment o_s by $\eta \cdot \omega$ and start the next iteration of OUTATIME. Therefore, the duration of one framework iteration is $\eta \cdot \omega$.

5.9 EXPERIMENTAL EVALUATION

The experimental evaluation of the presented OUTATIME framework is the final contribution of this chapter. We integrated OUTATIME into a prototype of SAP HANA (cf. Section 2.3). As the physical database design advisor, we use the table partitioning advisor SAHARA presented in Chapter 4. After discussing the experimental setup in Section 5.9.1, we examine in Section 5.9.2 how accurately OUTATIME predicts the future workload. Next, we evaluate in Section 5.9.3 the hardware cost and performance improvements achieved by OUTATIME compared to existing approaches. We then investigate the impact of the prediction confidence parameter η and observation period ω on the precision (cf. Section 5.9.4). Finally, we evaluate OUTATIME's prediction time in Section 5.9.5.

5.9.1 *Experimental Setup*

We use the same test system as introduced in Section 3.4.1. The workload predictor is implemented in `Python` and executed single-threaded on the same hardware. For our statistical methods we utilize the `scipy` [147] module which extends the `numpy` package [123]. We detect linear and exponential drifts by employing `scipy.stats.pearsonr` for the Pearson correlation. Further, we apply `scipy.stats.linregress` to predict the future fresh assignments and statement arrival rates for linear and exponential drifts. We also exploit the *Fast Fourier Transform* (FFT) [124] by `scipy.fft` to detect and predict reoccurring drifts quickly. To model subsequent and regular assignments, we utilize `scipy.stats.rv_discrete` to create discrete random variables.

WORKLOAD As a real-world scenario for workload drift, we consider the QADB of the SAP HANA development project [12, 13]. The QADB records statistics about more than 30 billion test runs, in total 2.7 TB of compressed data. The most relevant tables are: `makes` (m), `servers` (s), `install_sessions` (is), `test_profiles` (tp), `test_cases` (tc), `test_case_info` (tci), and `test_log_files` (tlf). We recorded the workload during a regular workday between 08:00 and 20:00.

PARAMETERS We set the observation period ω to 360 minutes and the prediction confidence factor η to 1.0. In Section 5.9.4, we analyze the impact of both parameters on the accuracy of the predicted workload. Furthermore, we set the discretization interval δ to 5 minutes. As elaborated in Section 5.5.1, the discretization interval should not be set larger than $\pi/2$ to still guarantee a table partitioning advice with high quality. Therefore, our choice is accurate enough for a table partitioning advisor that evicts cold data to a *standard persistent disk* offered by Google Cloud [60–62], resulting in $\pi = 4983.82$ seconds (cf. Section 2.4). Following Buda and Jarynowski's recommendation [27], we set the parameters for detecting linear and exponential drifts φ_{lin} and φ_{exp} to 0.8 each. In order to detect reoccurring drifts and static workloads, we set φ_{cyc} and φ_{static} to 0.1 each. Both parameters represent a maximal allowed fluctuation (cf. Section 5.5.2) and we argue that fluctuations smaller than 10% are imperceptible.

BASELINES As a baseline, we consider the *backward-looking approach* used by most existing work (cf. Section 5.10), where the observed workload $\mathcal{O}$ instead of the predicted workload $\mathcal{P}$ is fed into SAHARA. To demonstrate the potential of the OUTATIME framework, we also perform a best-case analysis, called *delphi approach*, where the actual future workload $\mathcal{F}$ is fed into SAHARA. For all approaches, the physical schema is only changed if the expected benefits of workload cost savings outweigh the costs of table repartitioning.

TOP-*k* Statements	Relative Execution Count	Relative Execution Time
TOP-10	33.18%	31.85%
TOP-100	84.81%	70.08%
TOP-1000	99.66%	97.69%

Table 5.3: Showing the relative total execution count for the 10, 100, and 1000 SQL statements with the highest execution count, respectively, their relative total execution time for the 10, 100, and 1000 SQL statements with the longest execution time.

5.9.2 *Experiment 1: Precision of Workload Prediction*

The first experiment evaluates how accurately the proposed framework predicts the future workload. We present a detailed investigation of the accuracy for a representative subset of the workload. In particular, we pick the 9 SQL statements with the highest execution count from the SQL plan cache of the QADB. As these are only `SELECT` and `UPDATE` statements, we add the `INSERT` statement with the highest execution count as our tenth statement to investigate a more diverse set of SQL statements. We argue that picking the SQL statements with the highest execution count lead to a considerable amount of totally executed SQL statements in the entire workload. This is because SQL statements in QADB are heavily skewed. To illustrate this, let us consider Table 5.3, where we show for the 10, 100, and 1000 SQL statements with the highest execution count their relative total execution count. In addition, we list the relative total execution time for the most long-running SQL statements. We observe that considering the 10 SQL statements with the highest execution count represents around one-third of the total workload execution count. Further, adding the `INSERT` statement with the highest execution count is important as the 100 SQL statements with the highest execution count in QADB contain 72 `SELECT` and 12 `UPDATE` statements but also 16 `INSERT` statements. Therefore, adding the `INSERT` statement with the highest execution count results in a more representative workload sample.

5.9.2.1 *Hot and Cold Classification*

We start our accuracy analysis on the hot and cold classification of parameter values assigned to host variables as we use the predicted workload as input to our table partitioning advisor. This is crucial as hot-classified parameter values in the observed workload tend to cool down over time due to data aging. As a result, rows belonging to hot-classified parameter value ranges in the observed workload, however, cold-classified in the future, can be proactively grouped into cold range partitions and evicted to cheaper storage layers.

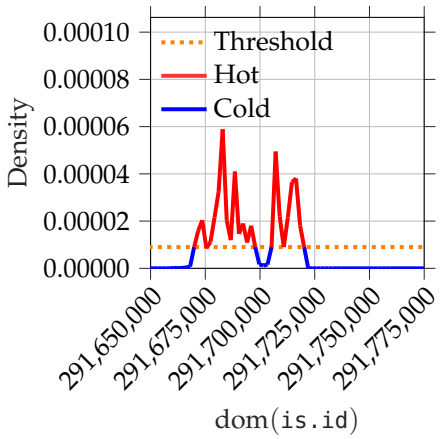
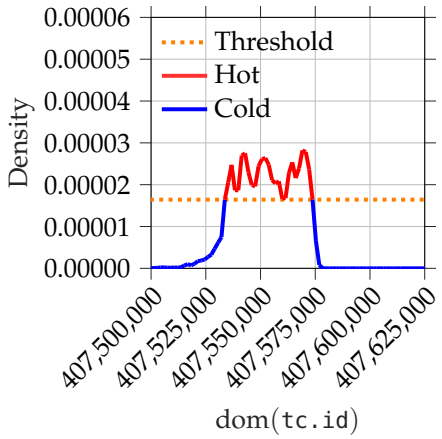
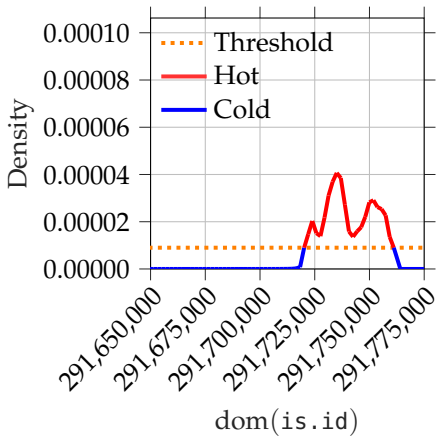
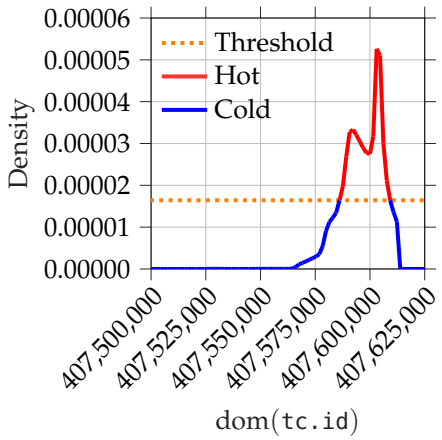
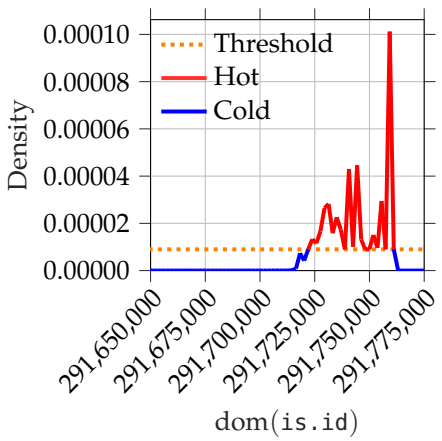
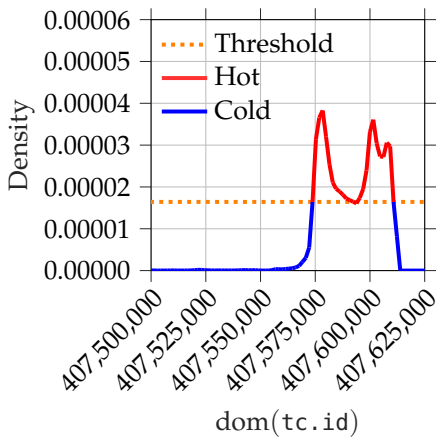
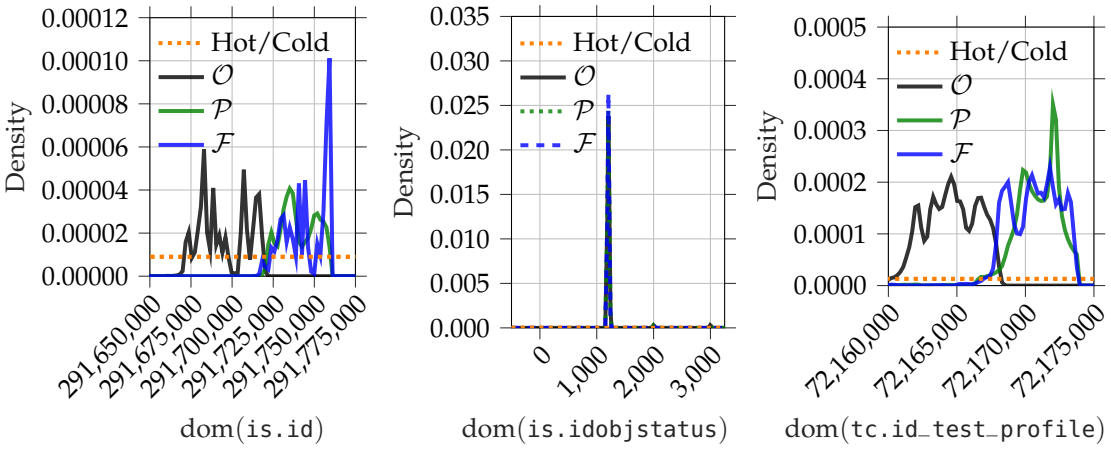
	Statement S_1, Host Variable h_{11}	Statement S_3, Host Variable h_{32}
Hot/Cold Classification	$f_e - f_s = o_e - o_s = 21600$ sec $\pi = 4983.82$ sec $\lvert\alpha(\mathcal{F}, h_{11})\rvert = 481248$ Hot/Cold Threshold $= 0.000009$ $= (21600 \text{ sec}/4983.82 \text{ sec})/481248$	$f_e - f_s = o_e - o_s = 21600$ sec $\pi = 4983.82$ sec $\lvert\alpha(\mathcal{F}, h_{32})\rvert = 263909$ Hot/Cold Threshold $= 0.000016$ $= (21600 \text{ sec}/4983.82 \text{ sec})/263909$
Observed Workload		
Predicted Workload		
Future Workload		

Table 5.4: Classifying parameter values to host variables h_{11} and h_{32} of SQL statements S_1 and S_3 as hot or cold based on the π-second rule for the observed, predicted, and future workload.

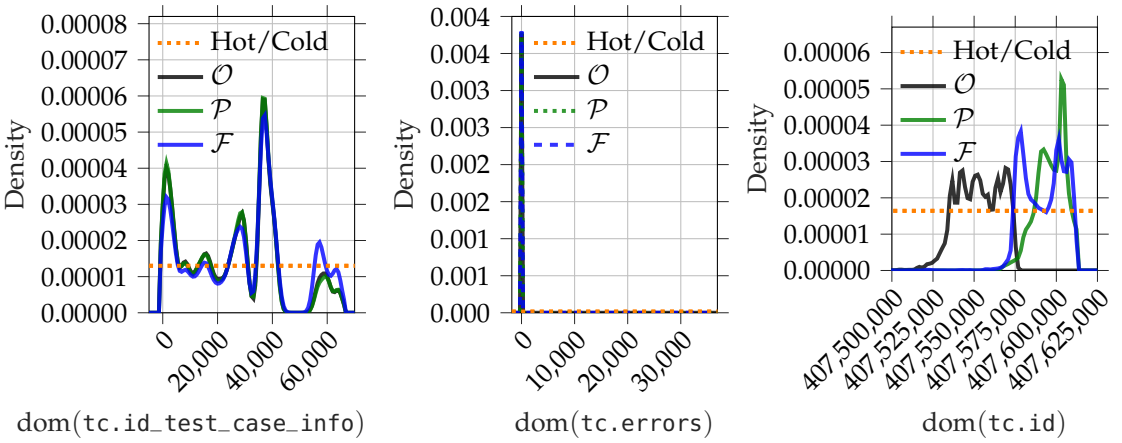
In Table 5.4, we show the hot/cold classification for parameter values assigned to host variables h_{11} and h_{32} of SQL statements S_1 and S_3 for the observed, predicted, and future workload. For each considered host variable, we pick the minimum and maximum parameter value assigned during the observed, predicted, and future workload. We then illustrate on the x-axis the domain between the minimum and maximum assigned parameter value as a continuous range. On the y-axis, we visualize the relative frequency for assigned parameter values as a probability density function. In addition, we draw a reasonable border between frequently and rarely assigned parameter values by considering the π-second rule calculated for a *standard persistent disk* offered by Google Cloud [60–62], resulting in $\pi = 4983.82$ seconds (cf. Section 2.4). Since observed, predicted, and future workloads are spanned over 6 hours (= 21600 seconds), a parameter value is classified as hot if accessed at least five times (21600 seconds / 4983.82 seconds = 4.33). Hot parameter values are then shown in red, whereas cold parameter values are blue. As we visualize the frequency as a probability density function, the total number of assigned parameter values equals 1. Hence, the relative hot/cold threshold is calculated by dividing the absolute hot/cold threshold (= 4.33) by the total number of assigned parameter values in the future (= $|\alpha(\mathcal{F}, h_{qr})|$). For example, there are 481248 assignments to host variable h_{11} during the future workload, such that the relative hot/cold threshold is 0.000009 (= 4.33/481248).

We observe from Table 5.4 that the hot/cold classification of the predicted workload $\mathcal{P}$ is very similar to the hot/cold classification of the future workload $\mathcal{F}$. In contrast, the hot/cold classification of the observed workload $\mathcal{O}$ is far off since data cools down over time. To give an example, the observed workload classifies parameter values to host variable h_{11} in the value range $[291670000, 291720000)$ as hot while all other parameter values are cold. By contrast, the predicted and the future workloads classify the value range $[291720000, 2917600000)$ as hot and all other parameter values as cold. This is because the parameter values classified as hot in the observed workload cool down over time due to data aging and new parameter values become hot. Our workload predictor detects this workload drift in the observed workload and forecasts the future workload, which can be used to accurately classify parameter values as hot or cold. Similar behavior is recognizable to host variable h_{32}, where the observed workload classifies only the value range $[407530000, 407570000)$ as hot, whereas the predicted workload classifies the value range $[407578000, 407610000)$ as hot like the future workload.

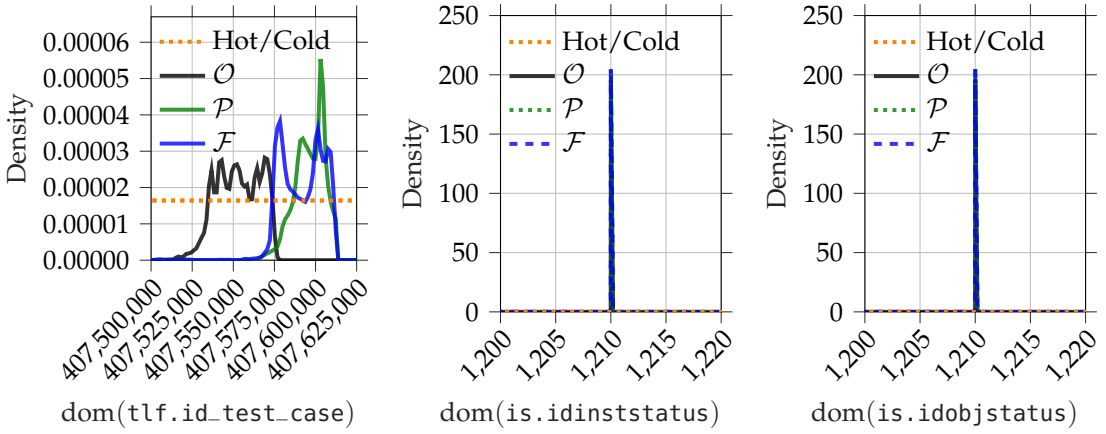
We now extend our hot/cold classification analysis to all selected 10 SQL statements. In Figure 5.12, we show for each of the 18 host variables of our 10 SQL statements the hot/cold classification of the observed $\mathcal{O}$, predicted $\mathcal{P}$, and future workload $\mathcal{F}$. We again pick the



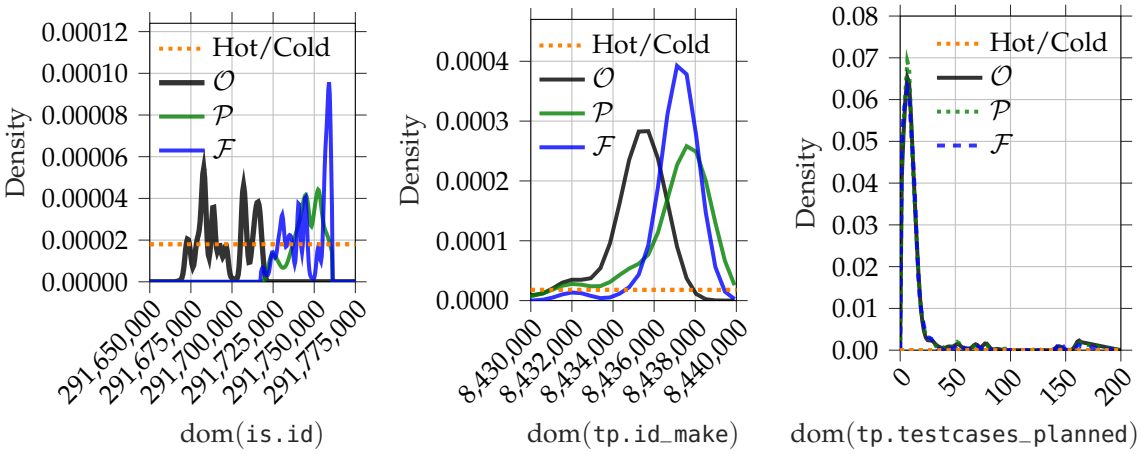
(a) Statement S_1, Host Variable h_{11} (b) Statement S_1, Host Variable h_{12} (c) Statement S_2, Host Variable h_{21}



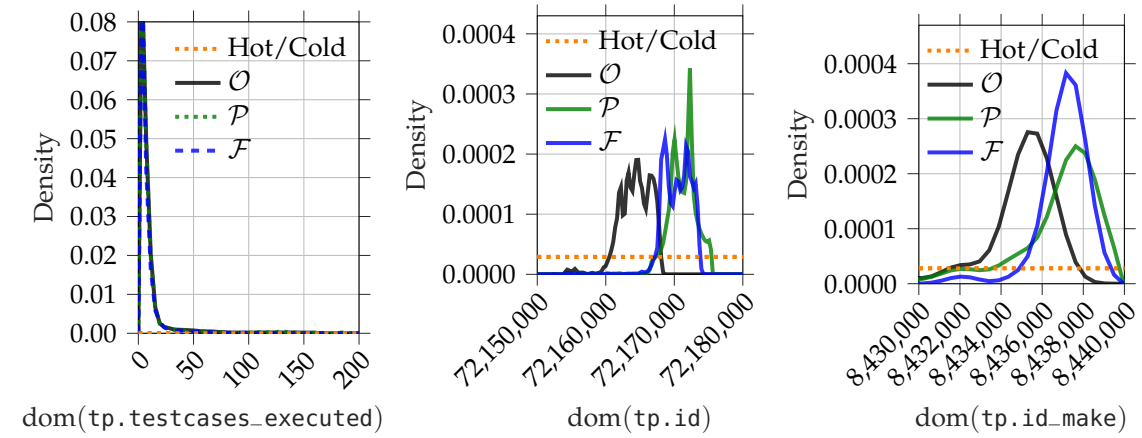
(d) Statement S_2, Host Variable h_{22} (e) Statement S_3, Host Variable h_{31} (f) Statement S_3, Host Variable h_{32}



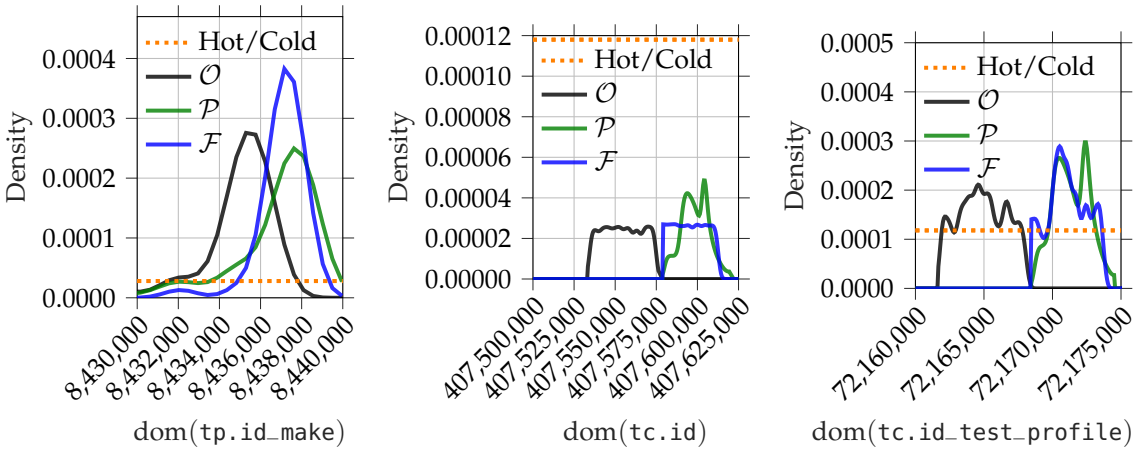
(g) Statement S_4, Host Variable h_{41} (h) Statement S_5, Host Variable h_{51} (i) Statement S_5, Host Variable h_{52}



(j) Statement S_5, Host Variable h_{53} (k) Statement S_6, Host Variable h_{61} (l) Statement S_7, Host Variable h_{71}



(m) Statement S_7, Host Variable h_{72} (n) Statement S_7, Host Variable h_{73} (o) Statement S_8, Host Variable h_{81}



(p) Statement S_9, Host Variable h_{91} (q) Statement S_{10}, Host Variable h_{101} (r) Statement S_{10}, Host Variable h_{102}

Figure 5.12: Density functions on assignments to host variables of the observed $\mathcal{O}$, predicted $\mathcal{P}$, and future workload $\mathcal{F}$ for our-10 SQL statements in QADB's plan cache.

minimum and maximum parameter value assigned to the host variable during the observed, predicted, and future workload to illustrate a continuous value range on the x-axis. The relative frequency for assigned parameter values is visualized as a probability density function on the y-axis. Additionally, we draw the same threshold for classifying data as either hot or cold as in Table 5.4 by considering the π-second rule of $\pi = 4983.82$ seconds, calculated for a *standard persistent disk* offered by Google Cloud (cf. Section 2.4).

We observe that classifying data as hot or cold by utilizing the predicted workload $\mathcal{P}$ is accurate in all 18 considered cases by comparing it to the hot/cold classification for the future workload $\mathcal{F}$. By contrast, the observed workload $\mathcal{O}$ fails in 10 of the 18 cases to classify hot and cold data precisely for the future. This is expected as hot-classified parameter values in the observed workload are often cold in the future workload as the workload changes over time in QADB.

In the remaining 8 cases, the observed $\mathcal{O}$ and predicted workload $\mathcal{P}$ lead to a similar hot and cold classification as the future workload $\mathcal{F}$. This is not surprising as the QADB workload also contains host variables with static assignments. For example, in Figure 5.13d, observed, predicted, and future workloads assign the same parameter values to the host variable h_{22} of statement S_2, which belongs to attribute `tc.id_test_case_info`. Since the dimension table `test_case_info` represents existing tests, e.g., their SQL string (cf. Section 5.3.3), this behavior is expected as the same tests are repeatedly executed over days and weeks to identify bugs or authorize patches.

While static assignments are the root cause in 7 out of the 8 cases, the assigned parameter values to host variable h_{101} are far off in the observed workload compared to the predicted and future workload (cf. Figure 5.12r). The reason is that S_{10} is an `INSERT` statement in which each parameter value of `id` is only assigned once to host variable h_{101}. Since we consider the π-second rule of $\pi = 4983.82$ during a workload observation of 6 hours, a parameter value must be accessed at least five times to be classified as hot (21600 seconds / 4983.82 seconds $= 4.33$). As a result, all assigned parameter values to host variable h_{101} are classified as cold, and thus the hot/cold classification is the same for the observed, predicted, and future workload.

5.9.2.2 *Precision Metric R^2*

In addition to the analysis of hot and cold classification between the observed $\mathcal{O}$, predicted $\mathcal{P}$, and future workload $\mathcal{F}$ in Figure 5.12, we now examine how well the observed $\mathcal{O}$ and predicted workload $\mathcal{P}$ approximates the future workload $\mathcal{F}$ in total. This is crucial as the predicted workload of our approach can be used for any physical database design advisor that relies on workload statistics and thus may not focus on classifying data as hot or cold. To do this, we use the coefficient of determination $R^2 \in (-\infty, 1]$ [77] between the observed $\mathcal{O}$

Statement	Predicted Number of Statament Instantiations	Actual Number of Statament Instantiations	Attribute of the Host Variable	Precision R^2	
				Observed Workload $\mathcal{O}$	Predicted Workload $\mathcal{P}$
S_1	534,026	481,248	`is.id`	−0.87	0.29
			`is.idobjstatus`	0.99	0.99
S_2	412,080	328,784	`tc.id_test_profile`	−1.45	0.67
			`tc.id_test_case_info`	0.91	0.91
S_3	327,471	263,909	`tc.errors`	1.00	1.00
			`tc.id`	−1.29	0.47
S_4	327,259	263,715	`tlf.id_test_case`	−1.29	0.47
S_5	260,667	235,740	`is.idinststatus`	1.00	1.00
			`is.idobjstatus`	1.00	1.00
			`is.id`	−1.16	−0.27
S_6	177,048	232,719	`tp.id_make`	−0.01	0.83
S_7	195,175	149,117	`tp.testcases_planned`	0.99	0.99
			`tp.testcases_executed`	0.96	0.96
			`tp.id`	−1.44	0.61
S_8	118,872	156,281	`tp.id_make`	0.03	0.83
S_9	118,728	155,765	`tp.id_make`	0.03	0.84
S_{10}	38,336	36,824	`tc.id`	−20.47	−6.16
			`tc.id_test_profile`	−6.20	0.01

Table 5.5: For our 10 SQL statements of QADB as a representative workload sample, we illustrate the predicted number of statement instantiations, the actual number of statement instantiations, and the R^2 score for observed and predicted assignments to each host variable.

and future workload $\mathcal{F}$ as well as between the predicted $\mathcal{P}$ and future workload $\mathcal{F}$. An R^2 score of 1 indicates a perfect prediction, while especially negative values, which are unbounded, represent inaccurate predictions. In Table 5.5, we show for our 10 SQL statements (column 1) the R^2 score for the observed workload $\mathcal{O}$ (column 5) and the predicted workload $\mathcal{P}$ (column 6) for each host variable (column 4). In addition, we depict the predicted (column 2) and the actual number of statement instantiations (column 3).

We observe that the predicted workload $\mathcal{P}$ has a higher R^2 score in all cases than the observed workload $\mathcal{O}$. In addition, the R^2 score of the predicted workload is frequently close to 1. Contrary to that, the observed workload has a negative R^2 score for host variables whose assignments drift significantly. The observed workload is expected to be far off from the future workload in those cases because parameter

values often drift over time. For example, the observed workload $\mathcal{O}$ assigns parameter values between 72,160,000 and 72,167,000 to the host variable h_{21} of SQL statement S_2, whereas the predicted $\mathcal{P}$ and future workload $\mathcal{F}$ assigns parameter values between 72,167,000 and 72,174,000, as observable in Figure 5.13d. Nonetheless, we also observe that a host variable with static assignments shows a strong R^2 score for both the predicted and observed workload. As an example, the host variable that belongs to `tp.testcases_planned` in statement S_7 has mostly static assignments (cf. Figure 5.12l).

While the predicted workload has a high R^2 score in most cases, we recognize that the R^2 score of the host variable that belongs to attribute `id` in statement S_{10} is negative for the predicted and observed workload. As statement S_{10} is an `INSERT` statement, all assignments to the host variable belonging to attribute `id` are classified as fresh (cf. Section 5.6.1). We then extrapolate the series of fresh assignments from the morning and noon (08:00 to 14:00) into the afternoon and evening (14:00 to 20:00). Since many developers start tests at 18:00, however, a steeper slope for the series of fresh assignments can be observed in the future workload compared to the predicted workload. Nevertheless, we argue that this workload behavior does not indicate a conceptual problem. The reason is that a more extended observation period (e.g., three days) would detect and anticipate this workload behavior as a daily reoccurring drift using discrete Fourier transform (cf. Section 5.5.2). In addition, we note that this workload behavior does not affect the hot/cold classification for assignments to the host variables of statement S_{10}. As discussed in Section 5.9.2.1, every parameter value is assigned only once to host variable h_{101}, resulting in a cold classification for all assigned parameter values (cf. Figure 5.12r).

We also observe that the predicted number of statement instantiations differs at most by a factor of 1.31 (e.g., for statement S_6). Additionally, we argue that the presented R^2 scores are representative of the entire workload as we considered a diverse set of `SELECT`, `UPDATE`, and `INSERT` statements, as well as our 10 SQL statements with their 18 host variables, contain around one-third of the total workload execution count. We also recognize that the remaining SQL statements with the highest 100 execution count tend to follow a similar workload drift pattern. Some SQL statements, for instance, have only an additional host variable in the `WHERE` clause compared to a SQL statement that we considered.

5.9.3 *Experiment 2: Memory Costs and Performance*

The second experiment analyzes how the table partitioning layouts proposed by OUTATIME, a backward-looking approach, and a Delphi approach affect the relation between memory costs and workload

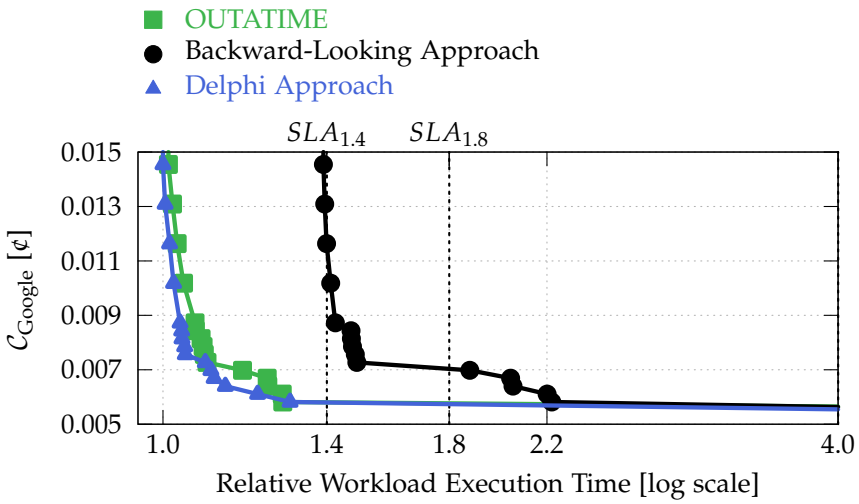


Figure 5.13: Relation between workload execution times (x-axis) and memory costs on Google Cloud (y-axis) between OUTATIME, a backward-looking, and a delphi approach.

performance. We consider the same memory costs $\mathcal{C}_{\text{Google}}$ of a Google Cloud instance per MB/s in ¢ as introduced in Section 4.8.3.

In Figure 5.13, we show on the y-axis the memory cost $\mathcal{C}_{\text{Google}}$ in ¢ when running the actual future workload using the three proposed partitioning layouts. Since DRAM costs comprise the majority of the hardware costs, we achieve different memory costs by adjusting the buffer pool size of SAP HANA (cf. Section 2.3.3). The resulting workload execution times are then shown on the x-axis. We observe that OUTATIME consistently outperforms a backward-looking approach in terms of performance. For example, with enough available memory (i.e., high memory costs), OUTATIME improves the execution time by 1.4× compared to a backward-looking approach.

In order to demonstrate the memory cost savings of OUTATIME, let us consider the two possible SLAs in Figure 5.13, denoted as $\text{SLA}_{1.4}$ and $\text{SLA}_{1.8}$. OUTATIME is able to satisfy $\text{SLA}_{1.4}$ with 2× fewer memory costs compared to a backward-looking approach, respectively, 1.3× for $\text{SLA}_{1.8}$. Furthermore, we observe that the performance and memory cost improvements of OUTATIME are close to the best case of the delphi approach.

The observed memory cost savings mainly result from a sharper separation of frequently and rarely accessed data into hot and cold partitions in OUTATIME compared to the backward-looking approach. More specifically, the physical schema proposed by OUTATIME groups frequently accessed data in the future into hot partitions and, therefore, separates data that was accessed frequently in the past but will only be accessed rarely in the future into cold partitions. This avoids polluting the buffer pool with cold data, allows for more aggressive partition pruning, and speeds up delta merges in SAP HANA.

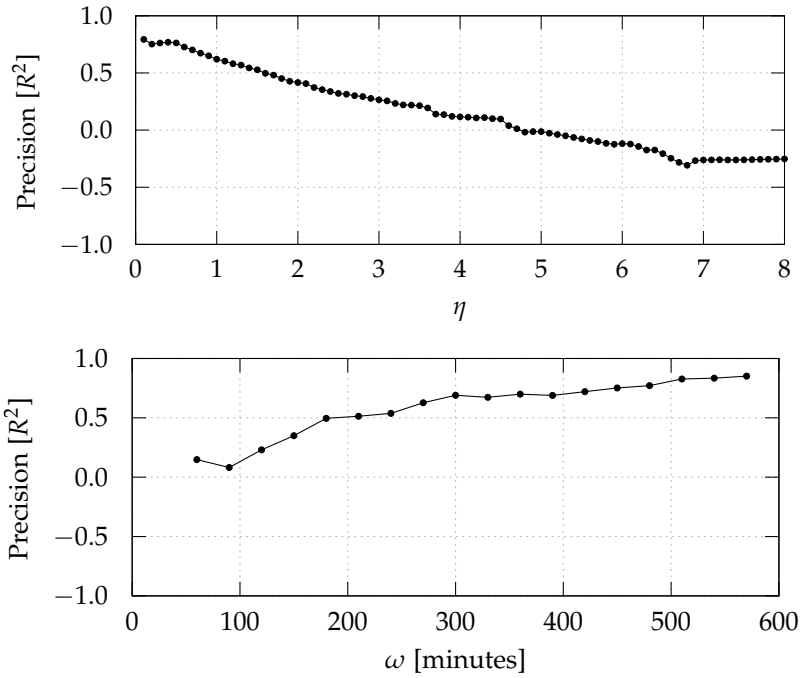


Figure 5.14: Impact of the prediction confidence factor η and observation period ω on the precision score R^2.

5.9.4 *Experiment 3: Sensitivity to η and ω*

The third experiment analyzes the impact of the prediction confidence factor η and the observation period ω on the prediction accuracy. To demonstrate the impact of both parameters, we predict the future workload for SQL statement S_6 for all possible combinations of η between 0.1 and 8.0 in steps of 0.1 and ω between 60 and 570 minutes in steps of 30 minutes. We start the workload observation at $o_s = 08{:}00$ for all parameter combinations. As we recorded the QADB workload between 08:00 and 20:00, we only executed parameter combinations of η and ω so that the future workload ended at $f_e \leq 20{:}00$. To illustrate, the parameter combination $\eta = 2.0$ and $\omega = 400$ minutes with $o_s = 08{:}00$ is not executed as the future workload ends at 21:20. We then calculate for each η between 0.1 and 8.0 and for each ω between 60 and 570 minutes the average R^2 score for all executed parameter combinations in which the parameter is used.

In Figure 5.14, we show on the top the average R^2 score (y-axis) for a different η between 0.1 and 8.0 (x-axis), and on the bottom the average R^2 score (y-axis) for a different ω between 60 and 570 minutes (x-axis). In general, we observe that the longer the predicted period, i.e., the larger η is, the less accurate predictions become. The poor accuracy resulting from η larger than 1 is expected as it is no longer possible to detect reoccurring drifts with a period longer than the observation period. Furthermore, we observe that more extended observation periods lead to higher R^2 scores.

We conclude that our choice of ω set to 360 minutes and η set to 1.0 leads to accurate predictions. In addition, a high prediction confidence factor η such as $\eta = 1.0$ is beneficial because table repartitioning costs are more likely amortized (cf. Section 5.8).

5.9.5 *Experiment 4: Prediction Time*

The final experiment analyzes the time required by OUTATIME to predict the future workload $\mathcal{P}$ for our 10 SQL statements, which demonstrate a representative workload sample. We measured the prediction time on the introduced hardware (cf. Section 5.9.1) while running the workload predictor, implemented in `Python`, single-threaded for observation periods ω between 60 and 360 minutes, and a prediction confidence factor η set to 1.0.

Observation period ω [min]	60	120	180	240	300	360
Prediction time [sec]	44.9	208.2	338.6	425.1	567.6	624.1

Table 5.6: Prediction time of OUTATIME with a prediction confidence factor $\eta = 1$ for observation periods ω between 60 and 360 minutes.

Table 5.6 shows that the required prediction time ranges between only 1.2% and 3.2% of the observation period. The prediction time can be reduced by parallelizing the execution of our workload predictor. Since we separately predict the future workload for each SQL statement, our approach can be easily parallelized. Predicting the future workload for the 100 SQL statements with the highest execution count is also recommended when lowering the prediction time due to a multi-threaded execution. Note that the prediction of the 100 SQL statements with the highest execution count might only increase the prediction time by a factor of 3 compared to the prediction of the 10 SQL statements with the highest execution count. The reason is that the 10 SQL statements with the highest execution count already predict around one-third of all statement instantiations in QADB.

5.10 RELATED WORK

In the context of automated physical database design, workload prediction has become a popular research area [131, 132]. Early work predicted the future workload type, e.g., the ratio of OLAP to OLTP statements [52, 53, 108]. Other approaches determine the hardware resources needed in the future, e.g., to adhere to performance guarantees assured in SLAs [117, 157]. In today's cloud databases age, cost-effective resource scaling received renewed attention when predicting the future [14, 29, 42, 59, 72, 83, 143, 154]. Others focus on forecasting the query response time whenever the workload changes [6, 58, 66, 170, 171]. In summary, all of these approaches only predict

high-level workload statistics, e.g., query response time or the hit ratio of the buffer pool. Since physical database design advisors rely on fine-granular statistics about the workload (cf. Chapter 3), the presented workload predictors are unsuitable for most physical database design advice. Our approach instead predicts fine-granular statistics about the workload, such as the future statement arrival rate and the future assignments to host variables, which can serve as input to arbitrary physical design advisors.

Holze et al. propose a detector for reoccurring and irregular workload drifts at the granularity of statement arrival rates [75]. While this approach considers only a limited number of workload drift types, we propose a framework that considers the five workload drift types in real-world applications: linear, exponential, reoccurring, static, and irregular (cf. Section 5.3). In addition, we handle overlapping workload drift types such as linear and reoccurring. Our framework also allows us to exchange all detectors, classifiers, and predictors in order to support workload drifts in novel environments. Furthermore, Holze et al. only detect workload drift types while our approach predicts the future workload based on detected workload drift types.

Other works use Markov models to predict future OLTP statements based on currently executed statements [49, 74, 134]. QueryBot5000 and Proteus additionally predicts the future statement arrival rate, which serves, for instance, as input to an index advisor [3, 107]. However, these approaches are limited to predict only the future statement arrival rate. Consequently, their predictions can only serve as input to index advisors but cannot be used for table partitioning advisors. The reason is that a table partitioning advisor needs knowledge about future assignments of parameter values in the `WHERE` clause to propose a range partition that can be pruned in the future workload (cf. Chapters 3 and 4). Therefore, we present a workload predictor that predicts future parameter values assigned to the host variables of the statements and the future statement arrival rate. As a result, the predicted workload by our approach can be used as input to arbitrary physical database design advisors that rely on workload statistics.

5.11 DISCUSSION

In this chapter, we presented a workload predictor that forecasts the future workload based on detected workload drifts in the statement arrival rate (cf. Section 5.5) and in the parameter values assigned to the host variables (cf. Section 5.6). The proposed workload predictor addresses the third challenge that is considered throughout this dissertation (cf. Section 1.3.3). Predicting the future workload is crucial as real-world applications are characterized by drifting workloads, such that the current physical schema may no longer be optimal. As a consequence, the database-as-a-service provider may increase the

buffer pool size of hosted database instances to still fulfill performance commitments. This may lead to increased internal costs for database-as-a-service providers and can lower their profitability. The main contribution of this chapter is that the predicted workload by our approach can be used as input to arbitrary physical database design advisors that rely on workload statistics, e.g., the table partitioning advisor from Chapter 4. This is because we predict future parameter values that are assigned to the host variables of the statements, whereas related work is limited to predicting only the future statement arrival rate (cf. Section 5.10). In addition, being a framework, our workload predictor allows exchanging its detectors, classifiers, and predictors to support workload drifts in novel applications. Finally, our results in Section 5.9 show accurate predictions of the future workload and hardware cost savings of up to 2× compared to a backward-looking approach, which uses the observed workload as input to a physical database design advisor.

6 CONCLUSION

Constrained by performance commitments assured in SLAs, database-as-a-service providers are challenged to minimize their internal costs in order to enhance profitability. Since database workloads are often skewed [7, 20, 46, 51, 56, 76, 102, 162] and DRAM constitutes the primary driver of hardware costs [60, 106], database-as-a-service providers can decrease their internal costs by evicting cold data to cheaper storage layers such as HDD or SSD. Therefore, only hot data should remain in DRAM. As database management systems load and evict data at page level from secondary storage to DRAM and vice versa [70], keeping only disk pages with hot data in the buffer pool can result in substantial cost savings. For a current physical schema, however, data are often not organized according to their access pattern. Instead, data may be sorted, for instance, based on the insertion order. Consequently, cold data can be stored on the same disk page as hot data. This leads to the problem of polluting the buffer pool with cold data. As a result, keeping both hot and cold data in the buffer pool wastes expensive DRAM capacities, which offers substantial cost reduction opportunities for database-as-a-service providers.

In this dissertation, we propose a solution to this problem by recommending a range partitioning layout for each relation in which hot disk pages contain mainly hot data (cf. Chapter 4). Our solution is based on a typical workload characteristic, where tuples of a relation are either frequently or rarely accessed according to a value range of a specific attribute of that relation. As a consequence, we group tuples that belong to hot-classified value ranges into hot range partitions, whereas tuples for cold-classified value ranges are gathered into cold range partitions. We then keep only hot-classified column partitions with a high density of hot data in the buffer pool. This results in a reduced buffer pool size while still adhering to SLAs and prevents the pollution of DRAM with cold data. In order to propose a range partitioning layout with high quality, we utilize accurate workload statistics collected during workload execution with low memory consumption and low runtime overhead (cf. Chapter 3). Furthermore, we periodically optimize the physical schema by employing a forward-looking approach. In particular, we forecast the approximate future workload, which is unknown during optimization, and use the predicted workload as input to our table partitioning advisor (cf. Chapter 5).

We now discuss the key findings based on the contributions made in this dissertation. Finally, we outline future research directions.

6.1 DISCUSSION OF KEY FINDINGS

As a first key finding, we identify that range partitioning offers excellent potential to reduce the buffer pool size while still adhering to performance commitments. We underpin this insight with our results in Section 4.8.2, where the proposed physical schema by our table partitioning advisor (cf. Chapter 4) reduces the buffer pool size by up to 3.2× compared to a physical schema in which all tables are unpartitioned. In addition, performance commitments are fulfilled with both physical schemata. The substantial main memory savings can be achieved as data accesses in JCC-H [22], JOB [98], and QADB [12, 13] follow a common access pattern: rows of a table are either frequently or rarely accessed according to a value range of a specific column of that table. To give an example, in JCC-H [22], rows of `orders` with an `o_orderdate` in the value range [1993-05-30, 1994-05-27) are frequently accessed, whereas rows for the value range [1995-01-01, 1997-05-01) are rarely accessed (cf. Section 4.1). We observe similar behavior for workloads with temporal skews like QADB [12, 13]. Rows in `test_cases`, for instance, are frequently accessed with a `test_profile_id` $\geq 72,164,500$, while rows with a `test_profile_id` $< 72,164,500$ are rarely accessed (cf. Section 5.1). We take advantage of this workload characteristic and group rows into partitions according to either hot- or cold-classified value ranges of a specific column of that table. As a consequence, a column partition contains either a high density of hot data or purely cold data. This avoids the pollution of the buffer pool with cold data when only the hot column partitions remain in the buffer pool. Whereas our recommendation is based on the workload's access pattern, data in unpartitioned tables are often not organized according to their access pattern. To illustrate, rows in an unpartitioned `orders` table are sorted by their `o_orderkey`, which does not correlate with their `o_orderdate` in JCC-H [22]. Therefore, cold data can be on the same disk page as hot data. This leads to an increased buffer pool size that still fulfills SLAs compared to our approach because the buffer pool is polluted with cold data.

Our second key finding is that range partitioning layouts optimized for memory footprint achieve similar performance as layouts solely optimized for performance but require a substantially smaller buffer pool size. This observation is derived from our results in Section 4.8.2, where the proposed layout by our approach always requires a smaller buffer pool size for different SLAs compared to layouts mainly optimized for performance. Besides, we achieve similar performance even when enough main memory is available. When optimizing a physical schema for performance, common approaches are hash, round-robin, and range partitioning [34, 45, 103]. For example, joins can be accelerated by employing hash or round-robin partitioning to tables on their primary key–foreign key relationship. However, as hash and

round-robin partitioning distribute data accesses evenly across partitions, hot and cold data are mixed on the same disk page, similar to unpartitioned tables. This leads to the pollution of the buffer pool with cold data. By contrast, range partitioning based on frequently accessed value ranges in the `WHERE` clause of SQL statements improves performance and reduces memory footprint due to partition pruning. Nonetheless, range partitioned layouts proposed by our approach achieve more considerable memory savings as several other aspects are considered in addition to partition pruning. First, we gather statistics for data accesses of all operators because every single data access may impact the hot/cold classification (cf. Section 4.4). Second, we consider dictionary compression and bit-packing because both impact the storage size of range partitioning layouts (cf. Section 4.6.2). Finally, we utilize a cost model (cf. Section 4.7) on memory footprint and performance when determining a range partitioning layout.

The third key finding is that near-optimal range partitioning layouts can be proposed with low optimization time, and low performance and memory impact on currently executed workloads. Ideally, an automated table partitioning advice should be proposed quickly and without impairing system performance as well as memory consumption, such that database-as-a-service providers can more easily optimize their hosted database instances. However, collecting accurate workload statistics, which are of particular importance to the quality of the advice, can result in high memory consumption and a noticeable performance impact. Further, proposing optimal range partitioning layouts may require considerable computation and memory resources. To address this trade-off, we track data accesses block-wise to reduce memory consumption and employ a stream-summary data structure to cope with heavily skewed access patterns to achieve high precision (cf. Section 3.3.4). Our results in Section 3.4.5 demonstrate that our statistics collection is precise, compact, and fast. Additionally, we propose a heuristic approach (cf. Section 4.5.2) to determine range partitioning layouts by only utilizing the domain block counters without employing the cost model. This lowers the optimization time (cf. Section 4.8.6) compared to the optimal approach using dynamic programming but does not guarantee optimality. Nevertheless, we experimentally demonstrate that our heuristic approach proposes near-optimal range partitioning layouts (cf. Section 4.8.5).

Our fourth key finding is that, in applications where the workload changes over time, range partitioning layouts should be optimized for the future workload instead of the observed workload. This insight is backed up by our experiments (cf. Section 5.9.3), where a forward-looking approach reduces the buffer pool size by up to 2× compared to a backward-looking approach, while performance commitments are still fulfilled for both approaches. The main reason for the reduction of the buffer pool size is that hot-

classified value ranges in the observed workload, which cool down over time, are proactively grouped into cold range partitions that can be evicted to cheaper storage layers. To give an example: In QADB [12, 13], the backward-looking approach groups frequently accessed rows of `test_cases` with `id_test_profile` $\geq 72,153,000$ into a hot range partition and all other records into a cold range partition. As the future workload only frequently access rows with `id_test_profile` $\geq 72,164,500$, the forward-looking approach gathers rows with $72,153,000 \leq$ `id_test_profile` $< 72,164,500$ into another cold range partition that is evicted to disk. Thus, only rows with `id_test_profile` $\geq 72,164,500$ remain in the buffer pool, and the pollution of the buffer pool with cold data is avoided in the future.

The fifth and final key finding is that a future workload can be predicted accurately based on an observed workload. We underpin this takeaway with our results in Section 5.9.2, which show accurate predictions of the future QADB workload [12, 13], employing our workload predictor (cf. Chapter 5). Those results are of particular importance as existing workload predictors share an exclusive focus on forecasting the statement arrival rate and do not consider the prediction of future parameterization of SQL statements. Consequently, their predictions cannot be used for table partitioning advisors that optimize for memory footprint. The reason is that knowledge about future assignments of parameter values in the `WHERE` clause is required to group rarely accessed value ranges into cold range partitions, which the future workload will prune. Hence, we predict the future parameter values assigned to the host variables of SQL statements (cf. Section 5.6), in addition to their arrival rate (cf. Section 5.5).

In summary, the five key findings demonstrate that the contributions made in this dissertation enable substantial reductions in the buffer pool size while still adhering to performance commitments. A reduced buffer pool size by our approach then results in substantial hardware cost savings, as our experiments in Sections 4.8.3 and 5.9.3 demonstrate. This is particularly crucial for database-as-a-service providers, which can adjust the virtualized hardware for each database instance in a versatile manner [71]. As a result, database-as-a-service providers can lower their internal costs to enhance profitability. Moreover, lower internal costs can give database-as-a-service providers the ability to compete for a higher market share by lowering their price offerings.

6.2 FUTURE RESEARCH DIRECTIONS

We start the discussion about future work on alternative partitioning criteria and models for workload prediction. Next, we broaden our perspective on complementary approaches for memory footprint reduction. Finally, we examine the implications of our approach to commercial database management systems in the future.

6.2.1 *Table Partitioning*

We demonstrate that our table partitioning advisor enables considerable buffer pool size reductions while adhering to SLAs. The substantial memory savings are achievable as JCC-H [22], JOB [98], and QADB [12, 13] share a common workload characteristic: rows in a table are either frequently or rarely accessed according to a value range of a certain column of that table. For example, a range of `o_orderdate` in `orders` of JCC-H [22], a range of `episode_nr` in `title` of JOB [98], or a range of `id_test_profile` in `test_cases` of QADB [12, 13]. We benefit from this workload characteristic by grouping rows into range partitions that belong to a hot- or cold-classified value range. Nevertheless, the variety of workloads in real-world applications can result in an access pattern, where range partitioning may not be the best choice. Let us assume a product table in which neither value ranges on price, category, rating, or product identifier can accurately separate hot and cold data. Therefore, future research should investigate alternative partitioning criteria that better fit those circumstances. One promising research direction is *list partitioning*, e.g., supported by Oracle Database [126], where tuples are grouped by a list of discrete, non-consecutive values into the same partition. As an example, products belonging to the categories iPhone and Playstation 5 could be merged into a hot partition. Further, *multi-level partitioning*, e.g., supported by SAP HANA [145], seems promising as tuples are grouped based on values of two or more attribute domains. To illustrate, a partition may consist of cheap products with an excellent rating.

Range, list, or multi-level partitioning are effective when hot and cold data can be identified by values from attribute domains of that table. However, they are ineffective when hot and cold data classification depends on another table where a foreign key–primary key relationship exists. Hence, another promising future research direction is *derived partitioning* [31, 175], where tuples are grouped based on the partitioning of a related table. For example, in JCC-H [22], the partitioning of `lineitem` can be derived by grouping all tuples into a partition that belongs to a range partition of `orders`. Note that range partition `lineitem` on `l_shipdate` and `orders` on `o_orderdate` may lead to a similar physical schema as `o_orderdate` and `l_shipdate` can strongly correlate. Such a correlation may not hold for other workloads, though. The benefits of derived partitioning are memory footprint reduction and performance improvement of joins. To give an example, whenever a filter on `o_orderdate` prunes range partitions of `orders` before a semi-join between `orders` and `lineitem` is executed, all derived partitions of `lineitem` must not be accessed, reducing memory consumption [175]. Additionally, performance can be improved within SAP HANA's partition-wise join strategy since only the derived partitions must be pair-wise joined [121].

As not all database management systems support list, multi-level, or derived partitioning, another promising future direction is to add an access frequency column to each table in which the access frequency of each row is stored. This allows using range partitioning, supported by almost every database [103], on access frequency ranges. SAP HANA, for instance, supports a so-called `DATAAGING` column but delegates the logic for determining the hotness of each row to the application running on top of the database [144]. Therefore, future work may revisit the classification of data as hot or cold at row granularity, similar to Project Siberia [7, 51, 102] or SAP ASE [130]. Moreover, to identify hot and cold data at row level with high precision, low memory consumption, and low runtime overhead, future research may consider the low-level data access counters of Section 3.3.4, too.

6.2.2 *Workload Prediction*

Since workloads are not only skewed over the domain but also over time, we propose a framework for workload prediction in Chapter 5. Although our approach achieves accurate predictions of the future workload for QADB [12, 13] (cf. Section 5.9), our prediction may be less accurate when forecasting workloads induced by other data-driven applications. One root cause can be the occurrence of functional dependencies between the parameters of multiple host variables. This is because we predict assignments to each host variable independently of each other. To illustrate, *Ferrari* and *red* strongly correlate when assigned to host variables that correspond to attributes *manufacturer* and *color* in a car dealership table. A promising future research direction is to employ techniques that have already improved cardinality estimation when two or more attributes of the same table are involved in the `WHERE` clause of an SQL statement, e.g., multi-dimensional histograms [82, 137], sampling [37, 161], or a regression model [50].

In our framework for workload prediction, the detection and classification steps are based on statistical characteristics, e.g., the Pearson correlation coefficient of the statement arrival rate, and a decision DAG with threshold parameters. As we demonstrate the applicability of our approach solely for QADB [12, 13], we may have over-fitted our approach. One reason can be our selection of threshold parameters in the decision DAG. Nevertheless, we propose a framework that allows exchanging all detectors, classifiers, and predictors to make adoptions for a more diverse set of workloads accomplishable. Thus, a promising future research direction is to investigate more robust detectors and classifiers. This can be achieved by leveraging complex statistical methods [163], utilizing a shape-based classifier [105, 129], or employing a feed-forward neural network as a classifier [84]. In addition, future work may study the benefits of more advanced predictors, e.g., kernel regression [17] or recurrent neural networks [168].

Instead of detecting workload drift types for classification and utilizing a separate predictor for each drift type, future work may investigate the benefits of a *model-based* approach, where the entire workload is represented in a single model. The main benefit of a model-based approach is that it is not limited to predefined workload drift types. Model-based approaches have already been applied to predict future statement arrival rates and future hardware resources, e.g., by Markov chains [49, 74, 134], ARMA models [14, 143], or ensemble methods [107]. Hence, one of the promising future research directions is to extend the use case of model-based approaches and predict future assignments to host variables, in addition to statement arrival rates.

6.2.3 *Memory Footprint Reduction of Database Management Systems*

We focus in this dissertation on evicting cold data to the storage tiers HDD or SSD to lower hardware costs while still adhering to performance commitments. Nowadays, new memory technologies such as persistent memory (PMEM) aim to fill the gap between SSD and DRAM. The benefit of PMEM over DRAM is lower prices per GB at the cost of lower memory bandwidth and higher latency [85]. Compared to SSDs, PMEM offers lower latency but is more expensive. Previous work, for instance, focused on placing columns [20, 162] and data structures like dictionaries [96] on different memory and storage tiers to reduce hardware costs. Additionally, multi-tier buffer managers were designed to load and evict disk pages on a more heterogeneous memory hierarchy [10, 16, 142]. Thus, one promising future research question is how table partitioning can be used to minimize hardware costs while fulfilling performance commitments on a multi-tier buffer manager. Such an approach may aim to evict never accessed (frozen) data to HDD, move rarely accessed (cold) data to SSD, push moderately accessed (warm) data on PMEM, and keep only frequently accessed (hot) data in DRAM. The π-second rule (cf. Section 2.4) could be utilized for all considered memory and storage devices to identify a more heterogeneous access frequency scale of frozen, cold, warm, and hot data.

Table partitioning is one aspect of the physical schema that offers excellent potential to reduce the memory footprint. Nonetheless, other aspects also show great prospects for DRAM savings while adhering to performance commitments. For example, *data compression* can reduce the memory footprint by compressing cold data heavily, whereas hot data remain uncompressed to avoid performance degradation [2, 19, 30, 40, 56]. In addition, *clustered indexes*, including database cracking, reorganize data such that accessing the index by selective filter predicates reduces the number of accessed disk pages compared to a full-column scan [80, 81, 89, 118, 125]. Furthermore, *small materialized aggregates* (SMA) [113] cache metadata of data blocks in main memory

to skip data without qualifying tuples, e.g., during filter predicate evaluation. This leads to approaches like MTO [48] or Qd-tree [172] that reshuffle data to minimize the number of accessed disk pages when SMA are maintained per disk page. As most physical design advisors focus only on a limited number of physical design aspects at a time, future research should focus on advisors that consider a large number of physical design aspects at once, similar to Proteus [3]. This is promising as each aspect of the physical schema can impact the effectiveness of another physical design aspect. To give an example, the effectiveness of dictionary compression is influenced by table partitioning (cf. Section 4.6.2). Moreover, a clustered index or SMA is less effective when applied to the partition-driving attribute of a range partitioning layout and vice versa.

The allocated main memory for the buffer manager is one expensive hardware resource in a database management system. However, other building blocks of a database management system can also require substantial DRAM costs, e.g., SQL plan cache, result cache, statistics cache, query optimizer, or intermediate results during query processing. Previous work focused, for instance, on reducing the allocated main memory for intermediate results by utilizing a concise hash table [15] or scheduling pipelines of the physical query execution plan to minimize the lifetime of pipeline breakers [92]. As the physical schema impacts the choice of a physical query execution plan (e.g., index nested loop join vs. hash join), one promising future research direction is to investigate the impact of the physical schema on memory savings of intermediate results. For example, a physical database design advisor may invoke the query optimizer's what-if API [79, 111] to additionally consider the potential memory savings achieved during query processing when recommending a physical schema.

6.2.4 *Implications to Commercial Database Management Systems*

We implement all approaches of this dissertation into a prototype of the commercial database management system SAP HANA (cf. Section 2.3) and demonstrate substantial main memory savings while still adhering to performance commitments assured in SLAs. This shows the applicability of the contributions made in this dissertation. In addition, early research in this work impacted the architecture of the NSE advisor [151], which recommends whether a column should be kept in DRAM (i.e., column loadable) or is loaded in small pieces (i.e., page loadable) through the buffer manager (cf. Section 2.3.3). Consequently, the next step would be to integrate our entire approach into production, e.g., by extending the NSE advisor, to ensure that the contributions made in this dissertation become a daily companion in many data-driven applications around the world.

BIBLIOGRAPHY

[1] Daniel Abadi, Peter A. Boncz, Stavros Harizopoulos, Stratos Idreos, and Samuel Madden. "The Design and Implementation of Modern Column-Oriented Database Systems." In: *Foundations and Trends in Databases*. Vol. 5. 3. Now Publishers Inc., 2013, pp. 197–280.

[2] Daniel Abadi, Samuel Madden, and Miguel Ferreira. "Integrating Compression and Execution in Column-Oriented Database Systems." In: *SIGMOD '06*. ACM, 2006, pp. 671–682.

[3] Michael Abebe, Horatiu Lazu, and Khuzaima Daudjee. "Proteus: Autonomous Adaptive Storage for Mixed Workloads." In: *SIGMOD '22*. ACM, 2022.

[4] Sanjay Agrawal, Surajit Chaudhuri, Lubor Kollar, Arun Marathe, Vivek Narasayya, and Manoj Syamala. "Database Tuning Advisor for Microsoft SQL Server 2005." In: *VLDB '04*. VLDB Endowment, 2004, pp. 1110–1121.

[5] Sanjay Agrawal, Vivek Narasayya, and Beverly Yang. "Integrating Vertical and Horizontal Partitioning into Automated Physical Database Design." In: *SIGMOD '04*. ACM, 2004, pp. 359–370.

[6] Mert Akdere, Ugur Çetintemel, Matteo Riondato, Eli Upfal, and Stanley B. Zdonik. "Learning-based Query Performance Modeling and Prediction." In: *ICDE '12*. IEEE, 2012, pp. 390–401.

[7] Karolina Alexiou, Donald Kossmann, and Per-Åke Larson. "Adaptive Range Filters for Cold Data: Avoiding Trips to Siberia." In: *Proceedings VLDB Endowment*. Vol. 6. 14. VLDB Endowment, 2013, pp. 1714–1725.

[8] Raja Appuswamy, Goetz Graefe, Renata Borovica-Gajic, and Anastasia Ailamaki. "The Five-Minute Rule 30 Years Later and Its Impact on the Storage Hierarchy." In: *Commun. ACM*. Vol. 62. 11. ACM, 2019, pp. 114–120.

[9] J.Scott Armstrong and Fred Collopy. "Error measures for generalizing about forecasting methods: Empirical comparisons." In: *International Journal of Forecasting*. Vol. 8. 1. 1992, pp. 69–80.

[10] Joy Arulraj, Andy Pavlo, and Krishna Teja Malladi. "Multi-Tier Buffer Management and Storage System Design for Non-Volatile Memory." In: *CoRR*. 2019.

[11] Manos Athanassoulis, Kenneth S. Bøgh, and Stratos Idreos. "Optimal Column Layout for Hybrid Workloads." In: *Proceedings VLDB Endowment*. Vol. 12. 13. VLDB Endowment, 2019, pp. 2393–2407.

[12] Thomas Bach, Artur Andrzejak, and Ralf Pannemans. "Coverage-based reduction of test execution time: Lessons from a very large industrial project." In: *ICSTW '17*. IEEE, 2017, pp. 3–12.

[13] Thomas Bach, Ralf Pannemans, and Sascha Schwedes. "Effects of an economic approach for test case selection and reduction for a large industrial project." In: *ICSTW '18*. IEEE, 2018, pp. 374–379.

[14] Francisco J. Baldan, Sergio Ramirez-Gallego, Christoph Bergmeir, Francisco Herrera, and Jose M. Benitez. "A Forecasting Methodology for Workload Forecasting in Cloud Systems." In: *TCC*. Vol. 6. 4. IEEE, 2018, pp. 929–941.

[15] R. Barber, G. Lohman, I. Pandis, V. Raman, R. Sidle, G. Attaluri, N. Chainani, S. Lightstone, and D. Sharpe. "Memory-Efficient Hash Joins." In: *Proceedings VLDB Endowment*. Vol. 8. 4. VLDB Endowment, 2014, pp. 353–364.

[16] Bishwaranjan Bhattacharjee, Mustafa Canim, Christian A Lang, George A Mihaila, and Kenneth A Ross. "Storage Class Memory Aware Data Management." In: *IEEE Data Eng. Bull.* Vol. 33. 4. Citeseer, 2010, pp. 35–40.

[17] Herman J. Bierens. "The Nadaraya–Watson kernel regression function estimator." In: *Topics in Advanced Econometrics*. Cambridge University Press, 1994, pp. 212–247.

[18] Carsten Binnig, Stefan Hildenbrand, and Franz Färber. "Dictionary-Based Order-Preserving String Compression for Main Memory Column Stores." In: *SIGMOD '09*. ACM, 2009, pp. 283–296.

[19] Martin Boissier. "Robust and Budget-Constrained Encoding Configurations for in-Memory Database Systems." In: *Proceedings VLDB Endowment*. Vol. 15. 4. VLDB Endowment, 2021, pp. 780–793.

[20] Martin Boissier, Rainer Schlosser, and Matthias Uflacker. "Hybrid data layouts for tiered HTAP databases with pareto-optimal data placements." In: *ICDE '18*. IEEE, 2018, pp. 209–220.

[21] Peter A. Boncz. "Monet: A Next-Generation DBMS Kernel For Query-Intensive Applications." PhD thesis. Universiteit van Amsterdam, Amsterdam, The Netherlands, 2002.

[22] Peter A. Boncz, Angelos-Christos Anatiotis, and Steffen Kläbe. "JCC-H: Adding Join Crossing Correlations with Skew to TPC-H." In: *TPCTC '17*. Springer International Publishing, 2017, pp. 103–119.

[23] Peter A. Boncz, Thomas Neumann, and Orri Erling. "TPC-H Analyzed: Hidden Messages and Lessons Learned from an Influential Benchmark." In: *Performance Characterization and Benchmarking*. Springer International Publishing, 2014, pp. 61–76.

[24] Michael Brendle, Nick Weber, Mahammad Valiyev, Norman May, Robert Schulze, Alexander Böhm, Guido Moerkotte, and Michael Grossniklaus. "Precise, Compact, and Fast Data Access Counters for Automated Physical Database Design." In: *BTW '21*. GI, 2021, pp. 79–100.

[25] Michael Brendle, Nick Weber, Mahammad Valiyev, Norman May, Robert Schulze, Alexander Böhm, Guido Moerkotte, and Michael Grossniklaus. "SAHARA: Memory Footprint Reduction of Cloud Databases with Automated Table Partitioning." In: *EDBT '22*. OpenProceedings.org, 2022, pp. 13–26.

[26] Nicolas Bruno and Surajit Chaudhuri. "An online approach to physical design tuning." In: *ICDE '07*. IEEE, 2007, pp. 826–835.

[27] Andrzej Buda and Andrzej Jarynowski. *Life-time of correlations and its applications. Vol. 1*. 2010.

[28] Mengchu Cai, Martin Grund, Anurag Gupta, Fabian Nagel, Ippokratis Pandis, Yannis Papakonstantinou, and Michalis Petropoulos. "Integrated Querying of SQL database data and S3 data in Amazon Redshift." In: *IEEE Data Eng. Bull.* Vol. 41. 2. Citeseer, 2018, pp. 82–90.

[29] Rodrigo N. Calheiros, Enayat Masoumi, Rajiv Ranjan, and Rajkumar Buyya. "Workload Prediction Using ARIMA Model and Its Impact on Cloud Applications' QoS." In: *TCC*. Vol. 3. 4. IEEE, 2015, pp. 449–458.

[30] Lujing Cen, Andreas Kipf, Ryan Marcus, and Tim Kraska. "LEA: A Learned Encoding Advisor for Column Stores." In: *aiDM '21*. ACM, 2021, pp. 32–35.

[31] S. Ceri, M. Negri, and G. Pelagatti. "Horizontal Data Partitioning in Database Design." In: *SIGMOD '82*. ACM, 1982, pp. 128–136.

[32] Donald D. Chamberlin and Raymond F. Boyce. "SEQUEL: A Structured English Query Language." In: *SIGFIDET '74*. ACM, 1974, pp. 249–264.

[33] J. M. Cheng, C. R. Loosley, A. Shibamiya, and P. S. Worthington. "IBM Database 2 performance: Design, implementation, and tuning." In: *IBM Systems Journal*. Vol. 23. 2. 1984, pp. 189–210.

[34] Cisco Systems, Inc. *TPC-H Full Disclosure Report: Microsoft SQL Server 2019*. 2019. URL: http://tpc.org/results/fdr/tpch/cisco~tpch~30000~cisco_ucs_c480_m5_server~fdr~2019-11-01~v04.pdf.

[35] Edgar F. Codd. "A Relational Model of Data for Large Shared Data Banks." In: *Commun. ACM*. Vol. 13. 6. ACM, 1970, pp. 377–387.

[36] Edgar F. Codd. "Further Normalization of the Data Base Relational Model." In: *Data Base Systems (Courant Computer Science Symposia 6)*. Prentice-Hall, Englewood Cliffs, 1972, pp. 33–64.

[37] Graham Cormode, Minos Garofalakis, Peter J. Haas, and Chris Jermaine. "Synopses for Massive Data." In: *Foundations and Trends in Databases*. Vol. 4. Now Publishers Inc., 2011, pp. 1–294.

[38] Carlo Curino, Evan Jones, Yang Zhang, and Sam Madden. "Schism: A Workload-driven Approach to Database Replication and Partitioning." In: *Proceedings VLDB Endowment*. Vol. 3. 1-2. VLDB Endowment, 2010, pp. 48–57.

[39] Benoît Dageville, Thierry Cruanes, Marcin Zukowski, Vadim Antonov, Artin Avanes, Jon Bock, Jonathan Claybaugh, Daniel Engovatov, Martin Hentschel, Jiansheng Huang, Allison W. Lee, Ashish Motivala, Abdul Q. Munir, Steven Pelley, Peter Povinec, Greg Rahn, Spyridon Triantafyllis, and Philipp Unterbrunner. "The Snowflake Elastic Data Warehouse." In: *SIGMOD '16*. ACM, 2016, pp. 215–226.

[40] Patrick Damme, Annett Ungethüm, Juliana Hildebrandt, Dirk Habich, and Wolfgang Lehner. "From a Comprehensive Experimental Survey to a Cost-Based Selection Strategy for Lightweight Integer Compression Algorithms." In: *ACM Trans. Database Syst.* Vol. 44. 3. ACM, 2019.

[41] Sudipto Das, Divyakant Agrawal, and Amr El Abbadi. "ElasTraS: An Elastic, Scalable, and Self-Managing Transactional Database for the Cloud." In: *ACM Trans. Database Syst.* Vol. 38. 1. ACM, 2013, 5::1–5::45.

[42] Sudipto Das, Feng Li, Vivek R. Narasayya, and Arnd Christian König. "Automated Demand-Driven Resource Scaling in Relational Database-as-a-Service." In: *SIGMOD '16*. ACM, 2016, pp. 1923–1934.

[43] Sudipto Das, Shoji Nishimura, Divyakant Agrawal, and Amr El Abbadi. "Albatross: Lightweight Elasticity in Shared Storage Databases for the Cloud Using Live Data Migration." In: *Proceedings VLDB Endowment*. Vol. 4. 8. VLDB Endowment, 2011, pp. 494–505.

[44] Justin DeBrabant, Andrew Pavlo, Stephen Tu, Michael Stonebraker, and Stan Zdonik. "Anti-caching: A New Approach to Database Management System Architecture." In: *Proceedings VLDB Endowment*. Vol. 6. 14. VLDB Endowment, 2013, pp. 1942–1953.

[45] Dell Inc. *TPC-H Full Disclosure Report: Exasol 7.1*. 2021. URL: http://tpc.org/results/fdr/tpch/dell~tpch~30000~dell_poweredge_r6525~fdr~2021-05-26~v02.pdf.

[46] Peter J. Denning. "The Working Set Model for Program Behavior." In: *Commun. ACM*. Vol. 11. 5. ACM, 1968, pp. 323–333.

[47] Roman Diachuk. "Table Partitioning Advisor for Dynamic Workloads." MA thesis. TU Dresden, Dresden, Germany, 2020.

[48] Jialin Ding, Umar Farooq Minhas, Badrish Chandramouli, Chi Wang, Yinan Li, Ying Li, Donald Kossmann, Johannes Gehrke, and Tim Kraska. "Instance-Optimized Data Layouts for Cloud Analytics Workloads." In: *SIGMOD '21*. ACM, 2021, pp. 418–431.

[49] Naiqiao Du, Xiaojun Ye, and Jianmin Wang. "Towards Workflow-Driven Database System Workload Modeling." In: *DBTest '09*. ACM, 2009.

[50] Anshuman Dutt, Chi Wang, Azade Nazi, Srikanth Kandula, Vivek Narasayya, and Surajit Chaudhuri. "Selectivity Estimation for Range Predicates Using Lightweight Models." In: *Proceedings VLDB Endowment*. Vol. 12. 9. VLDB Endowment, 2019, pp. 1044–1057.

[51] Ahmed Eldawy, Justin Levandoski, and Per-Åke Larson. "Trekking Through Siberia: Managing Cold Data in a Memory-optimized Database." In: *Proceedings VLDB Endowment*. Vol. 7. 11. VLDB Endowment, 2014, pp. 931–942.

[52] Said S. Elnaffar. "A Methodology for Auto-Recognizing DBMS Workloads." In: *CASCON '02*. IBM Press, 2002.

[53] Said S. Elnaffar and Patrick Martin. "An intelligent framework for predicting shifts in the workloads of autonomic database management systems." In: *AISTA '04*. 15-18. IEEE, 2004, pp. 1–8.

[54] Franz Färber, Norman May, Wolfgang Lehner, Philipp Große, Ingo Müller, Hannes Rauhe, and Jonathan Dees. "The SAP HANA Database–An Architecture Overview." In: *IEEE Data Eng. Bull.* Vol. 35. 1. Citeseer, 2012, pp. 28–33.

[55] Jane Fedorowicz. "Database Evaluation Using Multiple Regression Techniques." In: *SIGMOD '84*. ACM, 1984, pp. 70–76.

[56] Florian Funke, Alfons Kemper, and Thomas Neumann. "Compacting Transactional Data in Hybrid OLTP & OLAP Databases." In: *Proceedings VLDB Endowment*. Vol. 5. 11. VLDB Endowment, 2012, pp. 1424–1435.

[57] João Gama, Indrė Žliobaitė, Albert Bifet, Mykola Pechenizkiy, and Abdelhamid Bouchachia. "A Survey on Concept Drift Adaptation." In: *ACM Comput. Surv.* Vol. 46. 4. ACM, 2014.

[58] Archana Ganapathi, Harumi Kuno, Umeshwar Dayal, Janet L. Wiener, Armando Fox, Michael Jordan, and David Patterson. "Predicting Multiple Metrics for Queries: Better Decisions Enabled by Machine Learning." In: *ICDE '07*. IEEE, 2009, pp. 592–603.

[59] Zhenhuan Gong, Xiaohui Gu, and John Wilkes. "PRESS: PRedictive Elastic ReSource Scaling for cloud systems." In: *CNSM '10*. IEEE, 2010, pp. 9–16.

[60] Google. *Google Cloud Pricing*. 2022. URL: https://cloud.google.com/compute/all-pricing.

[61] Google. *Memory-optimized Machines*. 2022. URL: https://cloud.google.com/compute/docs/memory-optimized-machines.

[62] Google. *Persistent Disk Performance*. 2022. URL: https://cloud.google.com/compute/docs/disks/performance.

[63] Goetz Graefe. "The Five-minute Rule Twenty Years Later, and How Flash Memory Changes the Rules." In: *DaMoN '07*. ACM, 2007, 6:1–6:9.

[64] Jim Gray and Goetz Graefe. "The Five-minute Rule Ten Years Later, and Other Computer Storage Rules of Thumb." In: *SIGMOD Rec.* Vol. 26. 4. ACM, 1997, pp. 63–68.

[65] Jim Gray and Franco Putzolu. "The 5 Minute Rule for Trading Memory for Disc Accesses and the 10 Byte Rule for Trading Memory for CPU Time." In: *SIGMOD '87*. ACM, 1987, pp. 395–398.

[66] Chetan Gupta, Abhay Mehta, and Umeshwar Dayal. "PQR: Predicting Query Execution Times for Autonomous Workload Management." In: *ICAC '08*. IEEE, 2008, pp. 13–22.

[67] Aditya Gurajada, Dheren Gala, Fei Zhou, Amit Pathak, and Zhan-Feng Ma. "BTrim: Hybrid in-Memory Database Architecture for Extreme Transaction Processing in VLDBs." In: *Proceedings VLDB Endowment*. Vol. 11. 12. VLDB Endowment, 2018, pp. 1889–1901.

[68] Theo Haerder and Andreas Reuter. "Principles of Transaction-Oriented Database Recovery." In: *ACM Comput. Surv.* Vol. 15. 4. ACM, 1983, pp. 287–317.

[69] Stavros Harizopoulos, Daniel J. Abadi, Samuel Madden, and Michael Stonebraker. "OLTP through the Looking Glass, and What We Found There." In: *SIGMOD '08*. ACM, 2008, pp. 981–992.

[70] J. M. Hellerstein, M. Stonebraker, and J. Hamilton. *Architecture of a Database System*. Now Publishers Inc., 2007.

[71] Antony S. Higginson, Clive Bostock, Norman W. Paton, and Suzanne M. Embury. "Placement of Workloads from Advanced RDBMS Architectures into Complex Cloud Infrastructure." In: *EDBT '22*. OpenProceedings.org, 2022, pp. 487–497.

[72] Antony S. Higginson, Mihaela Dediu, Octavian Arsene, Norman W. Paton, and Suzanne M. Embury. "Database Workload Capacity Planning Using Time Series Analysis and Machine Learning." In: *SIGMOD '20*. ACM, 2020, pp. 769–783.

[73] Benjamin Hilprecht, Carsten Binnig, and Uwe Röhm. "Learning a Partitioning Advisor for Cloud Databases." In: *SIGMOD '20*. ACM, 2020, pp. 143–157.

[74] Marc Holze, Ali Haschimi, and Norbert Ritter. "Towards workload-aware self-management: Predicting significant workload shifts." In: *ICDEW '10*. IEEE, 2010, pp. 111–116.

[75] Marc Holze and Norbert Ritter. "Autonomic Databases: Detection of Workload Shifts with n-Gram-Models." In: *Advances in Databases and Information Systems*. Springer Berlin Heidelberg, 2008, pp. 127–142.

[76] Gui Huang, Xuntao Cheng, Jianying Wang, Yujie Wang, Dengcheng He, Tieying Zhang, Feifei Li, Sheng Wang, Wei Cao, and Qiang Li. "X-Engine: An Optimized Storage Engine for Large-scale E-commerce Transaction Processing." In: *SIGMOD '19*. ACM, 2019, pp. 651–665.

[77] Ann J. Hughes and Dennis E. Grawoig. *Statistics: A Foundation for Analysis*. Addison Wesley, 1971.

[78] IBM. *IBM DB2: Restrictions, Limitations, and Unsupported Database Configurations for Column-Organized Tables*. 2022. URL: https://www.ibm.com/docs/en/db2/11.5?topic=to-restrictions-limitations-unsupported-database-configurations-column-organized-tables.

[79] IBM. *IBM DB2: The Design Advisor*. 2022. URL: https://www.ibm.com/docs/en/db2/11.5?topic=strategy-design-advisor.

[80] Stratos Idreos, Martin L. Kersten, and Stefan Manegold. "Database Cracking." In: *CIDR '07*. 2007, pp. 68–78.

[81] Stratos Idreos, Martin L. Kersten, and Stefan Manegold. "Updating a Cracked Database." In: *SIGMOD '07*. ACM, 2007, pp. 413–424.

[82] Yannis Ioannidis. "The History of Histograms (abridged)." In: *VLDB '03*. Morgan Kaufmann, 2003, pp. 19–30.

[83] Sadeka Islam, Jacky Keung, Kevin Lee, and Anna Liu. "Empirical prediction models for adaptive resource provisioning in the cloud." In: *Future Generation Computer Systems*. Vol. 28. 1. Elsevier, 2012, pp. 155–162.

[84] Brian Kenji Iwana, Volkmar Frinken, and Seiichi Uchida. "DTW-NN: A novel neural network for time series recognition using dynamic alignment between inputs and weights." In: *Knowledge-Based Systems*. Vol. 188. 2020.

[85] Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amir Saman Memaripour, Yun Joon Soh, Zixuan Wang, Yi Xu, Subramanya R. Dulloor, Jishen Zhao, and Steven Swanson. "Basic Performance Measurements of the Intel Optane DC Persistent Memory Module." In: *CoRR*. 2019.

[86] Lars Kegel. "Feature-based Time Series Analytics." PhD thesis. TU Dresden, Dresden, Germany, 2020.

[87] Alfons Kemper and Thomas Neumann. "HyPer: A Hybrid OLTP&OLAP Main Memory Database System Based on Virtual Memory Snapshots." In: *ICDE '11*. IEEE, 2011, pp. 195–206.

[88] Michael S. Kester, Manos Athanassoulis, and Stratos Idreos. "Access Path Selection in Main-Memory Optimized Data Systems: Should I Scan or Should I Probe?" In: *SIGMOD '17*. ACM, 2017, pp. 715–730.

[89] Jan Kossmann, Stefan Halfpap, Marcel Jankrift, and Rainer Schlosser. "Magic mirror in my hand, which is the best in the land? An Experimental Evaluation of Index Selection Algorithms." In: *Proceedings VLDB Endowment*. Vol. 13. 11. VLDB Endowment, 2020, pp. 2382–2395.

[90] Jens Krueger, Changkyu Kim, Martin Grund, Nadathur Satish, David Schwalb, Jatin Chhugani, Hasso Plattner, Pradeep Dubey, and Alexander Zeier. "Fast Updates on Read-Optimized Databases Using Multi-Core CPUs." In: *Proceedings VLDB Endowment*. Vol. 5. 1. VLDB Endowment, 2011, pp. 61–72.

[91] Tirthankar Lahiri, Shasank Chavan, Maria Colgan, Dinesh Das, Amit Ganesh, Mike Gleeson, Sanket Hase, Allison Holloway, Jesse Kamp, Teck-Hua Lee, Juan Loaiza, Neil Macnaughton, Vineet Marwah, Niloy Mukherjee, Atrayee Mullick, Sujatha Muthulingam, Vivekanandhan Raja, Marty Roth, Ekrem Soylemez, and Mohamed Zait. "Oracle Database In-Memory: A dual format in-memory database." In: *ICDE '15*. 2015, pp. 1253–1258.

[92] Lukas Landgraf, Florian Wolf, Alexander Boehm, and Wolfgang Lehner. "Memory Efficient Scheduling of Query Pipeline Execution." In: *CIDR '22*. 2022.

[93] 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: *SIGMOD '16*. ACM, 2016, pp. 311–326.

[94] Per-Åke Larson, Cipri Clinciu, Campbell Fraser, Eric N. Hanson, Mostafa Mokhtar, Michal Nowakiewicz, Vassilis Papadimos, Susan L. Price, Srikumar Rangarajan, Remus Rusanu, and Mayukh Saubhasik. "Enhancements to SQL Server Column Stores." In: *SIGMOD '13*. ACM, 2013, pp. 1159–1168.

[95] Robert Lasch, Ismail Oukid, Roman Dementiev, Norman May, Suleyman S. Demirsoy, and Kai-Uwe Sattler. "Fast & Strong: The Case of Compressed String Dictionaries on Modern CPUs." In: *DaMoN'19*. ACM, 2019.

[96] Robert Lasch, Robert Schulze, Thomas Legler, and Kai-Uwe Sattler. "Workload-Driven Placement of Column-Store Data Structures on DRAM and NVM." In: *DaMoN '20*. ACM, 2021, 5:1–5:8.

[97] Juchang Lee, Yong Sik Kwon, Franz Färber, Michael Muehle, Chulwon Lee, Christian Bensberg, Joo Yeon Lee, Arthur H. Lee, and Wolfgang Lehner. "SAP HANA distributed in-memory database system: Transaction, session, and metadata management." In: *ICDE '13*. 2013, pp. 1165–1173.

[98] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter A. Boncz, Alfons Kemper, and Thomas Neumann. "How Good Are Query Optimizers, Really?" In: *Proceedings VLDB Endowment*. Vol. 9. 3. VLDB Endowment, 2015, pp. 204–215.

[99] Viktor Leis, Michael Haubenschild, Alfons Kemper, and Thomas Neumann. "LeanStore: In-Memory Data Management beyond Main Memory." In: *ICDE '18*. IEEE, 2018, pp. 185–196.

[100] Christian Lemke, Kai-Uwe Sattler, Franz Faerber, and Alexander Zeier. "Speeding Up Queries in Column Stores." In: *Data Warehousing and Knowledge Discovery*. Springer Berlin Heidelberg, 2010, pp. 117–129.

[101] Christian Lemke, Kai-Uwe Sattler, and Franz Färber. "Kompressionstechniken für spaltenorientierte Bi-Accelerator-Lösungen." In: *BTW '09*. GI, pp. 486–497.

[102] Justin J Levandoski, Per-Åke Larson, and Radu Stoica. "Identifying hot and cold data in main-memory databases." In: *ICDE '13*. IEEE, 2013, pp. 26–37.

[103] Sam S Lightstone, Toby J Teorey, and Tom Nadeau. *Physical Database Design*. Morgan Kaufmann Publishers Inc., 2010.

[104] Jessica Lin, Eamonn Keogh, Stefano Lonardi, and Bill Chiu. "A Symbolic Representation of Time Series, with Implications for Streaming Algorithms." In: *DMKD '03*. ACM, 2003, pp. 2–11.

[105] Jason Lines, Luke M. Davis, Jon Hills, and Anthony Bagnall. "A Shapelet Transform for Time Series Classification." In: *SIGKDD '12*. ACM, 2012, pp. 289–297.

[106] David Lomet. "Cost/Performance in Modern Data Stores: How Data Caching Systems Succeed." In: *DaMoN '18*. ACM, 2018, 9:1–9:10.

[107] Lin Ma, Dana Van Aken, Ahmed Hefny, Gustavo Mezerhane, Andrew Pavlo, and Geoffrey J. Gordon. "Query-Based Workload Forecasting for Self-Driving Database Management Systems." In: *SIGMOD '18*. ACM, 2018, pp. 631–645.

[108] P. Martin, S. Elnaffar, and T. Wasserman. "Workload Models for Autonomic Database Management Systems." In: *ICAS '06*. IEEE, 2006.

[109] Norman May, Alexander Boehm, and Wolfgang Lehner. "SAP HANA – The Evolution of an In-Memory DBMS from Pure OLAP Processing Towards Mixed Workloads." In: *BTW '17*. GI, 2017, pp. 545–563.

[110] Ahmed Metwally, Divyakant Agrawal, and Amr El Abbadi. "Efficient Computation of Frequent and Top-k Elements in Data Streams." In: *ICDT '05*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 398–412.

[111] Microsoft. *MS SQL Server Version 15: Database Engine Tuning Advisor*. 2021. URL: https://docs.microsoft.com/en-gb/sql/relational-databases/performance/database-engine-tuning-advisor?view=sql-server-ver15.

[112] Microsoft. *MS SQL Server*. 2022. URL: https://www.microsoft.com/en-us/sql-server/.

[113] Guido Moerkotte. "Small Materialized Aggregates: A Light Weight Index Structure for Data Warehousing." In: *VLDB '98*. Morgan Kaufmann Publishers Inc., 1998, pp. 476–487.

[114] Guido Moerkotte, David DeHaan, Norman May, Anisoara Nica, and Alexander Boehm. "Exploiting Ordered Dictionaries to Efficiently Construct Histograms with Q-Error Guarantees in SAP HANA." In: *SIGMOD '14*. ACM, 2014, pp. 361–372.

[115] Guido Moerkotte, Thomas Neumann, and Gabriele Steidl. "Preventing Bad Plans by Bounding the Impact of Cardinality Estimation Errors." In: *Proceedings VLDB Endowment*. Vol. 2. 1. VLDB Endowment, 2009, pp. 982–993.

[116] Ingo Müller, Cornelius Ratsch, and Franz Faerber. "Adaptive String Dictionary Compression in In-Memory Column-Store Database Systems." In: *EDBT '14*. OpenProceedings.org, 2014, pp. 283–294.

[117] D. Narayanan, E. Thereska, and A. Ailamaki. "Continuous resource monitoring for self-predicting DBMS." In: *MASCOTS '05*. IEEE, 2005, pp. 239–248.

[118] Vikram Nathan, Jialin Ding, Mohammad Alizadeh, and Tim Kraska. "Learning Multi-Dimensional Indexes." In: *SIGMOD '20*. ACM, 2020, pp. 985–1000.

[119] Rimma Nehme and Nicolas Bruno. "Automated Partitioning Design in Parallel Database Systems." In: *SIGMOD '11*. ACM, 2011, pp. 1137–1148.

[120] Thomas Neumann and Michael Freitag. "Umbra: A Disk-Based System with In-Memory Performance." In: *CIDR '20*. 2020, 29::1–29::7.

[121] Anisoara Nica, Reza Sherkat, Mihnea Andrei, Xun Cheng, Martin Heidel, Christian Bensberg, and Heiko Gerwens. "Statisticum: Data Statistics Management in SAP HANA." In: *Proceedings VLDB Endowment*. Vol. 10. 12. VLDB Endowment, 2017, pp. 1658–1669.

[122] Stefan Noll, Jens Teubner, Norman May, and Alexander Böhm. "Analyzing Memory Accesses with Modern Processors." In: *DaMoN '20*. ACM, 2020, 1:1–1:9.

[123] NumPy. *NumPy: The fundamental package for scientific computing with Python*. 2022. URL: https://numpy.org.

[124] Henri J. Nussbaumer. "The Fast Fourier Transform." In: *Fast Fourier Transform and Convolution Algorithms*. Springer Berlin Heidelberg, 1981, pp. 80–111.

[125] Matthaios Olma, Manos Karpathiotakis, Ioannis Alagiannis, Manos Athanassoulis, and Anastasia Ailamaki. "Adaptive partitioning and indexing for in situ query processing." In: *The VLDB Journal*. Vol. 29. 1. 2020, pp. 569–591.

[126] Oracle. *Orcale Database: List Partitioning*. 2022. URL: https://docs.oracle.com/en/database/oracle/oracle-database/21/vldbg/partition-concepts.html.

[127] Orcale. *Oracle Database*. 2022. URL: https://www.oracle.com/database/.

[128] Elizabeth J. O'Neil, Patrick E. O'Neil, and Gerhard Weikum. "The LRU-K Page Replacement Algorithm for Database Disk Buffering." In: *SIGMOD '93*. ACM, 1993, pp. 297–306.

[129] John Paparrizos and Luis Gravano. "K-Shape: Efficient and Accurate Clustering of Time Series." In: *SIGMOD '15*. ACM, 2015, pp. 1855–1870.

[130] Amit Pathak, Aditya Gurajada, and Pushkar Khadilkar. "Life Cycle of Transactional Data in In-memory Databases." In: *ICDEW '18*. IEEE, 2018, pp. 122–133.

[131] Andrew Pavlo, Gustavo Angulo, Joy Arulraj, Haibin Lin, Jiexi Lin, Lin Ma, Prashanth Menon, Todd C Mowry, Matthew Perron, and Ian Quah. "Self-Driving Database Management Systems." In: *CIDR '17*. 2017.

[132] Andrew Pavlo, Matthew Butrovich, Lin Ma, Prashanth Menon, Wan Shen Lim, Dana Van Aken, and William Zhang. "Make Your Database System Dream of Electric Sheep: Towards Self-Driving Operation." In: *Proceedings VLDB Endowment*. Vol. 14. 12. VLDB Endowment, 2021, pp. 3211–3221.

[133] Andrew Pavlo, Carlo Curino, and Stanley Zdonik. "Skew-Aware Automatic Database Partitioning in Shared-Nothing, Parallel OLTP Systems." In: *SIGMOD '12*. ACM, 2012, pp. 61–72.

[134] Andrew Pavlo, Evan P. C. Jones, and Stanley Zdonik. "On Predictive Modeling for Optimizing Transaction Execution in Parallel OLTP Systems." In: *Proceedings VLDB Endowment*. Vol. 5. 2. VLDB Endowment, 2011, pp. 85–96.

[135] Karl Pearson. "Notes on Regression and Inheritance in the Case of Two Parents." In: *Proceedings of the Royal Society of London*. Vol. 58. The Royal Society, 1895, pp. 240–242.

[136] Meikel Poess and Dmitry Potapov. "Data Compression in Oracle." In: *VLDB '03*. Morgan Kaufmann Publishers Inc., 2003, pp. 937–947.

[137] Viswanath Poosala and Yannis E. Ioannidis. "Selectivity Estimation Without the Attribute Value Independence Assumption." In: *VLDB '97*. Morgan Kaufmann Publishers Inc., 1997, pp. 486–495.

[138] Guna Prasaad, Alvin Cheung, and Dan Suciu. "Handling Highly Contended OLTP Workloads Using Fast Dynamic Partitioning." In: *SIGMOD '20*. ACM, 2020, pp. 527–542.

[139] Vijayshankar Raman, Gopi Attaluri, Ronald Barber, Naresh Chainani, David Kalmuk, Vincent KulandaiSamy, Jens Leenstra, Sam Lightstone, Shaorong Liu, Guy M. Lohman, Tim Malkemus, Rene Mueller, Ippokratis Pandis, Berni Schiefer, David Sharpe, Richard Sidle, Adam Storm, and Liping Zhang. "DB2 with BLU Acceleration: So Much More Than Just a Column Store." In: *Proceedings VLDB Endowment*. Vol. 6. 11. VLDB Endowment, 2013, pp. 1080–1091.

[140] Jun Rao and Kenneth A. Ross. "Making B+- Trees Cache Conscious in Main Memory." In: *SIGMOD '00*. ACM, 2000, pp. 475–486.

[141] Jun Rao, Chun Zhang, Nimrod Megiddo, and Guy Lohman. "Automating Physical Database Design in a Parallel Database." In: *SIGMOD '02*. ACM, 2002, pp. 558–569.

[142] Alexander van Renen, Viktor Leis, Alfons Kemper, Thomas Neumann, Takushi Hashida, Kazuichi Oe, Yoshiyasu Doi, Lilian Harada, and Mitsuru Sato. "Managing Non-Volatile Memory in Database Systems." In: *SIGMOD '18*. ACM, 2018, pp. 1541–1555.

[143] Nilabja Roy, Abhishek Dubey, and Aniruddha Gokhale. "Efficient Autoscaling in the Cloud Using Predictive Models for Workload Forecasting." In: *CLOUD '11*. IEEE, 2011, pp. 500–507.

[144] SAP. *SAP HANA Database: Data Aging*. 2019. URL: https://help.sap.com/doc/abapdocu_753_index_htm/7.53/en-US/abenhana_data_aging.htm.

[145] SAP. *SAP HANA Cloud Database: Single-level and Multi-level Table Partitioning*. 2022. URL: https://help.sap.com/docs/HANA_CLOUD_DATABASE/f9c5015e72e04fffa14d7d4f7267d897/c2ea130bbb571014b024ffeda5090764.html.

[146] SAP. *SAP HANA Cloud Database*. 2022. URL: https://www.sap.com/products/hana/cloud.html.

[147] SciPy. *SciPy: Fundamental algorithms for scientific computing in Python*. 2022. URL: https://scipy.org.

[148] Seagate. *CheetAh 15K.5*. 2006. URL: https://www.seagate.com/docs/pdf/marketing/po_cheetah_15k_5.pdf.

[149] Marco Serafini, Rebecca Taft, Aaron J. Elmore, Andrew Pavlo, Ashraf Aboulnaga, and Michael Stonebraker. "Clay: Fine-Grained Adaptive Partitioning for General Database Schemas." In: *Proceedings VLDB Endowment*. Vol. 10. 4. VLDB Endowment, 2016, pp. 445–456.

[150] C. E. Shannon. "Communication in the Presence of Noise." In: *Proceedings of the IRE*. Vol. 37. 1. 1949, pp. 10–21.

[151] Reza Sherkat, Colin Florendo, Mihnea Andrei, Rolando Blanco, Adrian Dragusanu, Amit Pathak, Pushkar Khadilkar, Neeraj Kulkarni, Christian Lemke, and Sebastian Seifert. "Native Store Extension for SAP HANA." In: *Proceedings VLDB Endowment*. Vol. 12. 12. VLDB Endowment, 2019, pp. 2047–2058.

[152] Reza Sherkat, Colin Florendo, Mihnea Andrei, Anil K. Goel, Anisoara Nica, Peter Bumbulis, Ivan Schreter, Günter Radestock, Christian Bensberg, Daniel Booss, and Heiko Gerwens. "Page As You Go: Piecewise Columnar Access In SAP HANA." In: *SIGMOD '16*. ACM, 2016, pp. 1295–1306.

[153] 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: *SIGMOD '12*. ACM, 2012, pp. 731–742.

[154] Binbin Song, Yao Yu, Yu Zhou, Ziqiang Wang, and Sidan Du. "Host load prediction with long short-term memory in cloud computing." In: *The Journal of Supercomputing*. Vol. 74. 12. Springer Berlin Heidelberg, 2018, pp. 6554–6568.

[155] Michael Stonebraker and Lawrence A. Rowe. "The Design of POSTGRES." In: *SIGMOD '86*. SIGMOD '86. ACM, 1986, pp. 340–355.

[156] Mike Stonebraker, Daniel J. Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Sam Madden, Elizabeth O'Neil, Pat O'Neil, Alex Rasin, Nga Tran, and Stan Zdonik. "C-Store: A Column-Oriented DBMS." In: *VLDB '05*. VLDB Endowment, 2005, pp. 553–564.

[157] Adam J. Storm, Christian Garcia-Arellano, Sam S. Lightstone, Yixin Diao, and M. Surendra. "Adaptive Self-Tuning Memory in DB2." In: *VLDB '06*. VLDB Endowment, 2006, pp. 1081–1092.

[158] Liwen Sun, Michael J. Franklin, Jiannan Wang, and Eugene Wu. "Skipping-Oriented Partitioning for Columnar Layouts." In: *Proceedings VLDB Endowment*. Vol. 10. 4. VLDB Endowment, 2016, pp. 421–432.

[159] TPC. *TPC-H Standard Specification*. 2021. URL: http://tpc.org/tpc_documents_current_versions/pdf/tpc-h_v3.0.0.pdf.

[160] Mahammad Valiyev. "Workload-Based Partition Advisor using Physical Query Execution Plans for In-Memory Database Management Systems." MA thesis. TU Munich, Munich, Germany, 2019.

[161] Jeffrey S. Vitter. "Random Sampling with a Reservoir." In: *ACM Trans. Math. Softw.* Vol. 11. 1. ACM, 1985, pp. 37–57.

[162] Lukas Vogel, Viktor Leis, Alexander van Renen, Thomas Neumann, Satoshi Imamura, and Alfons Kemper. "Mosaic: A Budget-Conscious Storage Engine for Relational Database Systems." In: *Proceedings VLDB Endowment*. Vol. 13. 12. VLDB Endowment, 2020, pp. 2662–2675.

[163] Xiaozhe Wang, Kate Smith, and Rob Hyndman. "Characteristic-Based Clustering for Time Series Data." In: *Data Mining and Knowledge Discovery*. Vol. 13. 3. 2006, pp. 335–364.

[164] Nick Weber. "Design and Implementation of Precise, Compact and Fast Data Access Counters for Self-Driven In-Memory Databases." MA thesis. University of Mannheim, Mannheim, Germany, 2019.

[165] Till Westmann, Donald Kossmann, Sven Helmer, and Guido Moerkotte. "The Implementation and Performance of Compressed Databases." In: *SIGMOD Rec.* Vol. 29. 3. ACM, 2000, pp. 55–67.

[166] Thomas Willhalm, Ismail Oukid, Ingo Müller, and Franz Faerber. "Vectorizing Database Column Scans with Complex Predicates." In: *ADMS '13*. 2013, pp. 1–12.

[167] 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: *Proceedings VLDB Endowment*. Vol. 2. 1. VLDB Endowment, 2009, pp. 385–394.

[168] Ronald J. Williams and David Zipser. "A Learning Algorithm for Continually Running Fully Recurrent Neural Networks." In: *Neural Computation*. Vol. 1. 2. 1989, pp. 270–280.

[169] Shmuel Winograd. "On computing the discrete Fourier transform." In: *Mathematics of computation*. Vol. 32. American Mathematical Society, 1978, pp. 175–199.

[170] Wentao Wu, Yun Chi, Hakan Hacígümüş, and Jeffrey F. Naughton. "Towards Predicting Query Execution Time for Concurrent and Dynamic Database Workloads." In: *Proceedings VLDB Endowment*. Vol. 6. 10. VLDB Endowment, 2013, pp. 925–936.

[171] Wentao Wu, Yun Chi, Shenghuo Zhu, Junichi Tatemura, Hakan Hacigümüs, and Jeffrey F. Naughton. "Predicting query execution time: Are optimizer cost models really unusable?" In: *ICDE '13*. IEEE, 2013, pp. 1081–1092.

[172] Zongheng Yang, Badrish Chandramouli, Chi Wang, Johannes Gehrke, Yinan Li, Umar Farooq Minhas, Per-Åke Larson, Donald Kossmann, and Rajeev Acharya. "Qd-Tree: Learning Data Layouts for Big Data Analytics." In: *SIGMOD '20*. ACM, 2020, pp. 193–208.

[173] Erfan Zamanian, Julian Shun, Carsten Binnig, and Tim Kraska. "Chiller: Contention-Centric Transaction Execution and Data Partitioning for Modern Networks." In: *SIGMOD '20*. ACM, 2020, pp. 511–526.

[174] Daniel C. Zilio, Jun Rao, Sam Lightstone, Guy Lohman, Adam Storm, Christian Garcia-Arellano, and Scott Fadden. "DB2 Design Advisor: Integrated Automatic Physical Database Design." In: *VLDB '04*. VLDB Endowment, 2004, pp. 1087–1097.

[175] M. Tamer Özsu and Patrick Valduriez. *Principles of Distributed Database Systems*. Springer, 2020.

LIST OF FIGURES

LIST OF TABLES