SAHARA: Memory Footprint Reduction of Cloud Databases with Automated Table Partitioning

Michael Brendle
michael.brendle@uni.kn
University of Konstanz

Nick Weber*
n.weber@celonis.com
Celonis SE

Mahammad Valiyev*
mahammad.valiyev@tum.de
Technical University of Munich

Norman May
norman.may@sap.com
SAP SE

Robert Schulze
robert.schulze@mailbox.org
SAP SE

Alexander Böhm
alexander.boehm@sap.com
SAP SE

Guido Moerkotte
moerkotte@uni-mannheim.de
University of Mannheim

Michael Grossniklaus
michael.grossniklaus@uni.kn
University of Konstanz

ABSTRACT

Enterprises increasingly move their databases into the cloud. As a result, database-as-a-service providers are challenged to meet the performance guarantees assured in service-level agreements (SLAs) while keeping hardware costs as low as possible. Being cost-effective is particularly crucial for cloud databases where the provisioned amount of DRAM dominates the hardware costs. A way to decrease the memory footprint is to leverage access skew in the workload by moving rarely accessed cold data to cheaper storage layers and retaining only frequently accessed hot data in main memory. In this paper, we present SAHARA, an advisor that proposes a table partitioning for column stores with minimal memory footprint while still adhering to all performance SLAs. SAHARA collects lightweight workload statistics, classifies data as hot and cold, and calculates optimal or near-optimal range partitioning layouts with low optimization time using a novel cost model. We integrated SAHARA into a commercial cloud database and show in our experiments for real-world and synthetic benchmarks a memory footprint reduction of 2.5× while still fulfilling all performance SLAs provided by the customer or advertised by the DBaaS provider.

1 INTRODUCTION

As enterprises are increasingly moving their databases into the cloud, database-as-a-service (DBaaS) providers (e.g., Amazon Redshift [14], Snowflake [18], or SAP HANA Cloud [61]) need to reduce hardware costs instead of *only* focusing on the classical database objective of maximizing performance to remain competitive in the marketplace. DBaaS providers can tailor their hardware setup to customer needs based on different compute, memory, and storage nodes. Such flexible provisioning models enable DBaaS providers to adapt the (virtual) hardware to the expected workload by configuring database instances appropriately. Additionally, DBaaS providers typically host database instances of thousands of customers. Consequently, multiple database instances, e.g., different tenants, can be placed on the same (virtual) node to increase the *tenant density* and to utilize available hardware resources more efficiently.

*Work done while at SAP SE.

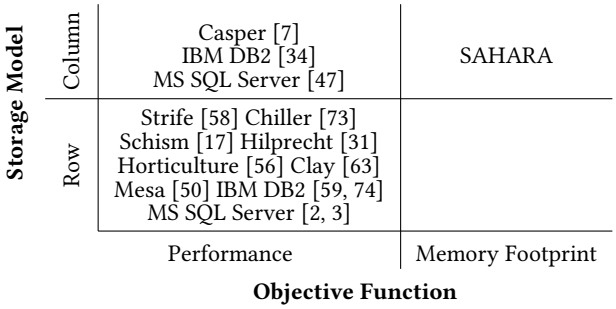
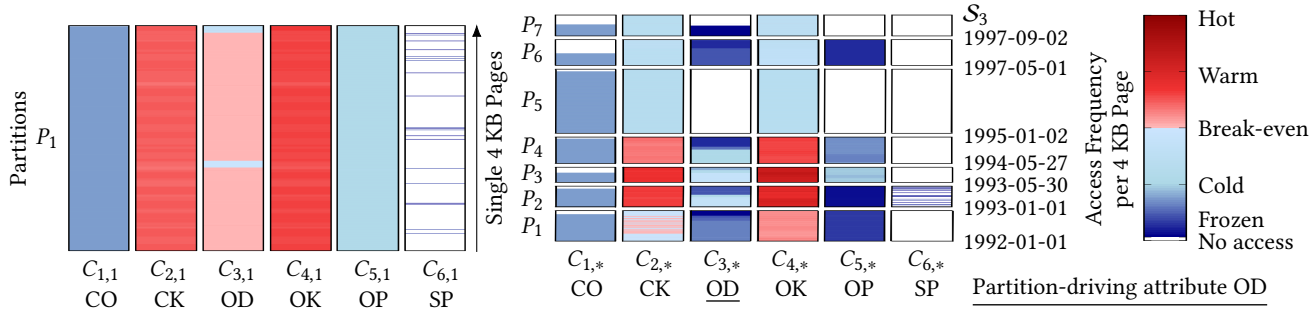


Figure 1: Comparison between SAHARA and state-of-the-art table partitioning advisors on their objective function and storage model.

Previous work [44] identified the provisioned amount of DRAM as the primary driver of hardware costs. For example, in 2021, a memory-optimized Google Cloud [26] instance costs monthly only $18 per vCPU and $80 per TB of provisioned disk space, while main memory is prized at $2606 per TB of DRAM. Since DBaaS providers are able to scale memory nodes flexibly, reductions in memory footprint (e.g., smaller buffer pool sizes for database instances) quickly translate to substantial hardware cost savings. However, service-level agreements (SLAs) guarantee customers a certain level of performance. As a result, DBaaS providers are challenged to meet the performance SLAs provided by the customer or advertised by the DBaaS provider while keeping the memory footprint of their offerings as low as possible.

In this paper, we present SAHARA[1], which proposes a table partitioning for each relation such that the buffer pool size is minimized while all performance SLAs are fulfilled. To illustrate the main idea of SAHARA, let us consider the query "`SELECT DISCOUNT FROM LINEITEM WHERE SHIPDATE >= 1994-12-24 and SHIPDATE < 1995-01-01`" that selects the discount of all shipped line items between Christmas and New Year's Eve 1994. Assume that `LINEITEM` is stored on pages in a disk-based column store with a buffer pool, that it is not clustered by `SHIPDATE`, and that it does not have any index on `SHIPDATE`. Under this assumption, the whole `SHIPDATE` column must be scanned to evaluate the selection predicate. Further, to project on `DISCOUNT`, almost all pages of the `DISCOUNT` column are accessed because the qualifying tuples are likely distributed over all `DISCOUNT` pages. In

[1] A storage advisor based on heavy and rare accesses.



(a) Non-partitioned ORDERS relation

(b) Range-partitioned ORDERS relation on attribute O_ORDERDATE as chosen by SAHARA

Figure 2: Example of page accesses to six attributes of ORDERS for a non-partitioned and a range-partitioned layout. After executing 200 queries of the JCC-H benchmark [10], the pages are classified as hot or cold according to the five-minute rule [27]. The range-partitioned layout proposed by SAHARA consists of fewer hot pages than the non-partitioned layout. The buffer pool size can then be reduced with SAHARA's layout by keeping only hot-classified pages in DRAM.

contrast, creating a range partitioning on SHIPDATE with partition boundary values 1994-12-24 and 1995-01-01 reduces the number of accessed pages and, therefore, the required buffer pool size. There are two reasons for this: First, due to partition pruning, only the SHIPDATE column of the single range partition [1994-12-24, 1995-01-01) must be scanned to evaluate the selection predicate. The SHIPDATE column of all other range partitions is not accessed. Second, because of the correlated storage of the DISCOUNT column with the SHIPDATE column that follows from the range partitioning, only DISCOUNT pages that refer to the range partition [1994-12-24, 1995-01-01) are accessed to project on DISCOUNT. All other range partitions are not accessed.

To show that SAHARA's idea also works for more complex workloads, we consider Fig. 2, illustrating two partitioning layouts for six attributes of ORDERS after executing 200 queries of the JCC-H benchmark [10]. The non-partitioned layout is shown on the left, while the right shows the range-partitioned layout proposed by SAHARA. Each column partition consists of multiple 4 KB pages (i.e., each horizontal line represents a single 4 KB page). To quantify the impact of SAHARA's layout on the buffer pool size, we count the number of physical page accesses of all operators by the workload. To draw a reasonable border between frequently (hot) and rarely (cold) accessed pages, we consider the five-minute-rule[2] to classify pages as hot or cold [27]. 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 SAHARA consists of fewer hot pages than the non-partitioned layout. In particular, only a subset of O_CUSTKEY (CK), O_ORDERDATE (OD), and O_ORDERKEY (OK) is frequently accessed in SAHARA's layout compared to the non-partitioned layout. Consequently, the buffer pool size can be reduced with SAHARA's layout by keeping only hot-classified pages in DRAM.

Fig. 1 shows that existing table partitioning advisors focuses on the classical database objective of maximizing performance. In particular, existing table partitioning advisors [17, 31, 50, 56, 58, 63] do the exact opposite of what SAHARA intends to achieve: To balance the load on partitions, they distribute accesses evenly across all partitions and, therefore, generate pages with mixed temperatures (hot and cold data). This pollutes the buffer pool with cold data because all pages with hot data (including cold data) are required to be held in DRAM to maximize performance.

Besides a different objective function, Fig. 1 shows that state-of-the-art table partitioning advisors are mainly designed for row stores [2, 3, 17, 31, 50, 56, 58, 59, 63, 73, 74]. SAHARA instead is a table partitioning advisor designed for column stores and considers, in contrast to related work [47, 74], the impact of dictionary compression on the memory footprint of range partitioning layouts. This aspect is crucial since many column stores allow dictionary compression [1].

Furthermore, SAHARA collects physical data accesses and, therefore, is not sensitive to skew in the distribution of data accesses, unlike IBM DB2 [34, 59, 74] and Microsoft SQL Server [2, 3, 47] that rely on the optimizer's what-if API. Finally, SAHARA is a table partitioning advisor that handles all operators, contrary to Casper [7], which considers only selections.

Our contributions are summarized below:

- We formalize the problem of minimizing the memory footprint while fulfilling performance SLAs for range partitionings with optional dictionary compression (Sec. 3).
- We introduce SAHARA, a table partitioning advisor that collects lightweight workload statistics (Sec. 4) and calculates (near)-optimal range partitioning layouts with low optimization time (Sec. 5) for estimates of accesses as well as storage sizes (Sec. 6) using a novel cost model (Sec. 7).
- We integrated SAHARA prototypically into SAP HANA Cloud and analyze the memory footprint, hardware costs, precision of estimates, optimality, overhead, and optimization time for real-world and synthetic benchmarks (Sec. 8).

2 OVERVIEW

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 [43]. 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.

[2]The five-minute-rule is a simple rule of thumb based on economic considerations comparing the cost-performance ratio of DRAM and secondary storage: "[d]ata referenced every five minutes should be memory resident" [27].

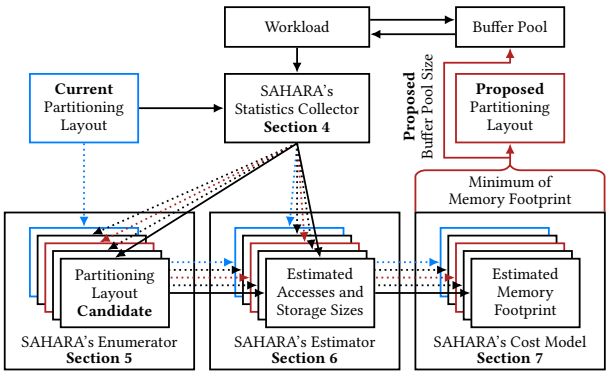


Figure 3: The system model of SAHARA consists of four building blocks: a statistics collector, an enumerator for partitioning layout candidates, an estimator for workload's data accesses and storage sizes, and a cost model for the memory footprint of partitioning layouts.

2.1 Problem Statement

For a given relation with n attributes, let attribute A_k ($1 \le k \le n$) refer to the partition-driving attribute, e.g., O_ORDERDATE (OD) in the range-partitioned ORDERS relation in Fig. 2. For each attribute A_k, there exists a set of potential range partitioning specifications $\mathbb{S}_k$. For example, Fig. 2 shows the range specification $\mathcal{S}_3 \in \mathbb{S}_3$ of the O_ORDERDATE (OD) attribute. The problem we consider is to find a partition-driving attribute A_k, a range partitioning specification $\mathcal{S}_k \in \mathbb{S}_k$, and a buffer pool size $B \in \mathbb{N}$ such that the memory footprint $\mathcal{M}$ of a workload W is minimized (e.g., lower monetary memory costs), while the workload execution time $\mathcal{E}$ for range partitioning specification $\mathcal{S}_k$ and buffer pool size B does not violate a performance *SLA* (e.g., a maximum workload execution time provided by the customer or advertised by the DBaaS provider):

$$\begin{array}{ll} \underset{1\le k\le n, \mathcal{S}_k\in\mathbb{S}_k, B\in\mathbb{N}}{\arg\min} & \mathcal{M}(\mathcal{S}_k, W, B) \\ \text{subject to} & \mathcal{E}(\mathcal{S}_k, W, B) \le SLA. \end{array}$$

2.2 System Model

Fig. 3 shows the system model of SAHARA. We first collect data access statistics during workload execution for the *current* partitioning layout (Sec. 4). This could also be a non-partitioned layout if SAHARA was not applied before. We then enumerate partitioning layout *candidates*, where each partitioning layout *candidate* is identified by a partition-driving attribute A_k and a range partitioning specification $\mathcal{S}_k \in \mathbb{S}_k$. Sec. 5 presents exact and heuristic enumeration algorithms to determine a partitioning layout. Afterwards, in Sec. 6, the statistics collected on the *current* partitioning layout must be transformed into estimates of statistics for each partitioning layout *candidate*. The reason is that workload's data accesses differ for each partitioning layout *candidate* due to partition pruning. In addition, compression ratios can also differ due to the number of values replicated into the dictionaries of multiple partitions. Finally, our cost model calculates for each partitioning layout *candidate* the memory footprint $\mathcal{M}$ based on estimated accesses, storage sizes, a given *SLA*, and the hardware configuration (Sec. 7). A partitioning layout *candidate* with minimal memory footprint $\mathcal{M}$ is proposed. Besides, a buffer pool size B is calculated to fulfill the *SLA*. As shown in Fig. 3, we may also end up in the current partitioning layout.

3 PROBLEM FORMALIZATION

We formalize the problem of Sec. 2.1 by defining range partitioning layouts for a relation in a column store.

Definition 3.1. Let R be a **relation** with n **attributes** $A_1, ..., A_n$. A **range partitioning specification** $\mathcal{S}_k = \{v_{1_k}, ..., v_{p_k}\} \subseteq \Pi^D_{A_k}(R)$ with $v_{1_k} < \ldots < v_{p_k}$ and $v_{1_k} = \min(\Pi^D_{A_k}(R))$ is a subset of the domain of the **partition-driving attribute** A_k ($1 \le k \le n$).

The main idea is that we record accesses on the domain of each attribute and classify value ranges as hot or cold. We then choose a partition-driving attribute A_k and propose a range partitioning specification $\mathcal{S}_k$ from a set of **range partitioning specifications** $\mathbb{S}_k$. For example, the right side of Fig. 2 shows the proposed range partitioning specification $\mathcal{S}_3 = \{1992\text{-}01\text{-}01, \ldots, 1997\text{-}09\text{-}02\}$ for partition-driving attribute O_ORDERDATE (OD). Further, we call any other attribute $A_i \neq A_k$ a **passive attribute**.

Definition 3.2. A **partitioning** $\mathcal{P}(\mathcal{S}_k) = \{P_1, \ldots, P_{p_k}\}$ of a relation R into p_k partitions is generated by

$$P_j := \begin{cases} \sigma_{v_{j_k} \le A_k < v_{(j+1)_k}}(R) & (j < p_k) \\ \sigma_{v_{p_k} \le A_k}(R) & (j = p_k) \end{cases}, \text{ for all } 1 \le j \le p_k.$$

The partitioning $\mathcal{P}(\mathcal{S}_k)$ is generated by selecting only tuples of relation R for partition P_j, where the value of the partition-driving attribute A_k is between two partition boundaries v_{j_k} and $v_{(j+1)_k}$ of $\mathcal{S}_k$. For example, Fig. 2 shows the partitioning $\mathcal{P}(\mathcal{S}_3) = \{P_1, ..., P_7\}$ generated from the range partitioning specification $\mathcal{S}_3$.

Definition 3.3. We associate with every tuple in relation R a unique **global tuple identifier** $\texttt{gid} \in [1, |R|]$, and with every tuple in a partition P_j a unique **local tuple identifier** $\texttt{lid} \in [1, |P_j|]$. Given a partition P_j and a local tuple identifier lid, the global tuple identifier is retrieved by $P_j[\texttt{lid}].\texttt{GID}$.

We associate local and global tuple identifiers to identify the same tuple of different partitioning layouts.

Definition 3.4. An **uncompressed column partition** $C^u_{i,j}$ of A_i ($1 \le i \le n$) in P_j ($1 \le j \le p_k$) is a vector of length $|P_j|$ with

$$C^u_{i,j}[\texttt{lid}] = P_j[\texttt{lid}].A_i, \qquad \text{for all } 1 \le \texttt{lid} \le |P_j|,$$

where $P_j[\texttt{lid}].A_i$ retrieves the value of attribute A_i in partition P_j for the tuple with the local tuple identifier lid.

An uncompressed column partition is a vector of all values of an attribute for a partition. The local tuple identifiers determine the placement of the values inside the vector. Almost all column stores allow for optional dictionary compression [1]. We thus introduce definitions for dictionaries and compressed columns.

Definition 3.5. Let $\Pi^D_{A_i}(P_j) = \{v_{1_{i,j}} < \ldots < v_{d_{i,j}}\}$ denote the domain of an attribute A_i of partition P_j. The **dictionary** of attribute A_i of partition P_j is a bijection $D_{i,j} = (\texttt{vid}_{i,j} : \Pi^D_{A_i}(P_j) \rightarrow [1, d_{i,j}])$ with $\texttt{vid}_{i,j}(v_{y_{i,j}}) = y_{i,j}$.

The dictionary $D_{i,j}$ of attribute A_i of partition P_j is a bijection $\texttt{vid}_{i,j}$, where $\Pi^D_{A_i}(P_j)$ is the domain and $[1, d_{i,j}]$ is the range of the function, such that the y-th value of the domain returns number y.

Definition 3.6. A **dictionary-compressed column partition** $C^c_{i,j}$ of attribute A_i in partition P_j is a vector of numbers in $[1, d_{i,j}]$, such that

$$C^c_{i,j}[\texttt{lid}] = \texttt{vid}_{i,j}(C^u_{i,j}[\texttt{lid}]), \qquad \text{for all } 1 \le \texttt{lid} \le |P_j|.$$

The dictionary-compressed column partition $C^c_{i,j}$ stores the numbers returned by the bijection $\texttt{vid}_{i,j}$ of the dictionary $D_{i,j}$ for all values of attribute A_i for partition P_j.

Definition 3.7. We define a **column partition** $C_{i,j}$ depending on the effectiveness of dictionary compression:

$$C_{i,j} \coloneqq \begin{cases} (C^c_{i,j}, D_{i,j}) & \text{if } ||C^c_{i,j}|| + ||D_{i,j}|| \leq ||C^u_{i,j}|| \\ C^u_{i,j} & \text{otherwise,} \end{cases}$$

where $||\ldots||$ is the number of bytes to store a(n) (un-)compressed column partition or a dictionary. The storage size in bytes of $C_{i,j}$ is then defined as $||C_{i,j}|| = \min(||C^c_{i,j}|| + ||D_{i,j}||, ||C^u_{i,j}||)$.

Fig. 2 shows all column partitions $C_{1,1}, \ldots, C_{6,7}$ for ORDERS generated from the range partitioning specification $\mathcal{S}_3$. Finally, we define the range partitioning layout as a set of all column partitions $C_{i,j}$.

Definition 3.8. A **partitioning layout** $\mathcal{L}(R, A_k, \mathcal{S}_k)$ for a relation R and a range partitioning specification $\mathcal{S}_k$ with partition-driving attribute A_k consists of the set of all column partitions $C_{i,j}$:

$$\mathcal{L}(R, A_k, \mathcal{S}_k) \coloneqq \{C_{i,j} \mid 1 \leq i \leq n, 1 \leq j \leq p_k\}.$$

4 STATISTICS COLLECTION

We begin by describing our approach by explaining how workload statistics are collected for the *current* partitioning layout (cf. Fig. 3). On the one hand, we record *domain* (dictionary) accesses to enumerate range partitioning layout candidates in Sec. 5. On the other hand, we capture *row* (logical tuple identifiers) accesses to calculate the memory footprint of a partitioning layout in Sec. 7 based on estimated accesses and storage sizes (Sec. 6).

As we focus on reducing the memory footprint for a workload, the distribution of data accesses over time is crucial because it impacts the buffer pool eviction policy [23, 55]. For example, if a data item is accessed twice within a short period, it is likely cached in the buffer pool at the second access. Thus, we track accesses only within specified time windows so that the statistics are not dominated by many accesses occurring only during a short period. However, by increasing the length of a time window, it becomes more difficult to separate the access pattern of individual queries. Therefore, Sec. 7 shows how the length of a time window should be chosen.

Definition 4.1. Let Q be a set of queries (**workloads**), Ω a set of time windows, and R a relation. A **workload trace** W is then defined as

$$W \subseteq \{(\texttt{gid}, A_i, q, \omega) \mid 1 \leq \texttt{gid} \leq |R|, A_1 \leq A_i \leq A_n, q \in Q, \omega \in \Omega\},$$

where each element in W denotes a single access to attribute A_i of the tuple with global tuple identifier gid by query q within time window ω.

We record accesses block-wise to reduce the memory overhead of the statistics collection. This can lead to imprecise access frequencies if a block contains values with heavily skewed access patterns. As a result, smaller block sizes lead to more precise access frequencies. We previously analyzed the impact of the block size on the precision of the access frequency of values and showed how workload statistics are collected space and time-efficiently [12]. In our experiments in Sec. 8, we set the block size such that 1% additional memory is spent on statistics compared to the data set size.

Definition 4.2 (Row block counter). We define a **row block access** for an attribute A_i, a partition P_j, a local block number z, and a time window ω as

$$x_{\text{block}}(A_i, P_j, z, \omega) \coloneqq \begin{cases} 1 & \exists\, \texttt{gid}, q, \texttt{lid} : (\texttt{gid}, i, q, \omega) \in W \\ & \wedge\ \texttt{lid} \in [1, |P_j|] \wedge\ P_j[l].\texttt{GID} = \texttt{gid} \\ & \wedge\ \lfloor \texttt{lid}/\text{RBS}_{i,j} \rfloor = z \\ 0 & \text{otherwise,} \end{cases}$$

where the **row block size** $\text{RBS}_{i,j}$ is the number of local tuple identifiers that are grouped for counting accesses.

A row block access is recorded if W contains at least one element that accesses the attribute A_i of the tuple by query q within time window ω, such that the lid of partition P_j corresponds to the gid of the tuple and falls into the local block number z.

To define domain block accesses, we assume a Boolean function $\text{eval}(i, v, q)$ that evaluates a value v for a conjunction of predicates in query q's WHERE clause on A_i.

Definition 4.3 (Domain block counter). Let $\Pi^D_{A_i}(R) = \{v_{1_i} < .. < v_{u_i} < .. < v_{d_{i,j}}\}$ denote the domain of an attribute A_i. We define a **domain block access** for an attribute A_i, a domain block number y, and a time window ω as

$$v_{\text{block}}(A_i, y, \omega) \coloneqq \begin{cases} 1 & \exists\, \texttt{gid}, q, v_{u_i} : (\texttt{gid}, i, q, \omega) \in W \\ & \wedge\ \text{eval}(i, v_{u_i}, q) \wedge\ R[\texttt{gid}].A_i = v_{u_i} \\ & \wedge\ \lfloor u_i/\text{DBS}_i \rfloor = y \\ 0 & \text{otherwise,} \end{cases}$$

where the **domain block size** DBS_i is the number of consecutive values in the domain constituting a block.

A domain block is accessed if there is at least one query in the workload trace that satisfies the predicate during the given time window and is part of the specified domain block.

Example. Fig. 4 presents the statistics collected for the execution of JCC-H Query 3 [10] during one time window. Row blocks of each accessed attribute are shown on the top left, whereas domain blocks are at the bottom left. The query execution plan is illustrated on the right. We show the first access to each attribute because we only record whether or not a block was accessed during a time window. Blocks accessed by an operator are highlighted using a unique color and number ⓧ to identify the query execution plan operator that caused that access. The selection operators ❶ and ❷ touch all row blocks of C_MKTSEGMENT and O_ORDERDATE, but the respective domain blocks only record if domain values satisfied the WHERE clause (Def. 4.3). Therefore, range-partitioning ORDERS on O_ORDERDATE (OD) with [1993-05-29, 1998-08-03) would create a column partition that is never accessed. In particular, ORDERS has 15 million tuples which are all fetched without partitioning, given the range partitioning specification $\mathcal{S} = \{1992\text{-}01\text{-}01, 1993\text{-}05\text{-}29\}$, we would fetch 3,204,724 tuples. The subsequent hash join ❸ touches all row and domain blocks on the build (CUSTOMER) and the probe side (ORDERS). However, only a subset of the rows is accessed, e.g., the customer with C_CUSTKEY (CK) '5004' was filtered out by ❶. Hence, the buffer pool is polluted with cold data since all pages but not all rows are read. Next, an index nested loop join ❹ touches all row blocks in ORDERS, but only ≈75% of the row blocks in LINEITEM. For example, the order with L_ORDERKEY (OK) '43' comprises 3 million items, which spans multiple blocks, but was already filtered out by ❷. The following selection ❺ filters all L_SHIPDATE (SD) values smaller than 1993-05-30. Values larger than 1993-09-26 are not read since the L_SHIPDATE (SD) of an item is not 121 days after its O_ORDERDATE (OD), and orders with an O_ORDERDATE (OD) larger

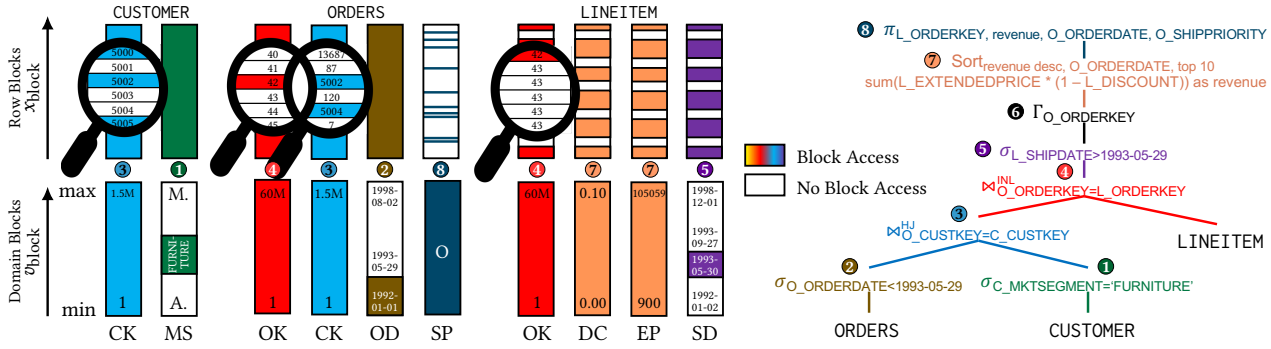


Figure 4: Collected statistics on row and domain blocks for the execution of JCC-H Query 3 during one time window.

than 1993-05-28 were filtered out by ❷. While such constraints are only known to domain experts [11] and cannot be extracted from query 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 (SD), i.e., 75% of LINEITEM pages are fetched without partitioning, while a partitioning layout based on the range partitioning specification $\mathcal{S} = \{1992\text{-}01\text{-}01, 1993\text{-}05\text{-}30, 1993\text{-}09\text{-}27\}$ would access only 5% of the pages. While the following group-by operator ❻ does not create new accesses, the sorting operator ❼ additionally accesses L_DISCOUNT (DC) and L_EXTENDEDPRICE (EP). Finally, the projection ❽ accesses only ten blocks of L_SHIPPRIORITY (SP) since it is a top-k query.

5 DETERMINING PARTITIONING LAYOUTS

We now explain how we determine a partitioning layout based on the collected statistics (Sec. 4). Since any attribute A_k may be the partition-driving attribute, we compute a partitioning layout for each possible A_k. Afterwards, we propose the layout that minimizes the memory footprint most while not violating the customer's *SLA*. We identify an optimal range partitioning for A_k in Sec. 5.1 and present a heuristic in Sec. 5.2 to lower the optimization time.

5.1 Optimal Range Partitioning Layout

Alg. 1 finds an optimal range partitioning specification for a partition-driving attribute A_k using dynamic programming (DP). The idea is to calculate the optimal range partitioning for d ($1 \le d \le d_k$) distinct values of the domain of A_k (d_k is the number of distinct values of A_k) by using a previously calculated optimal range partitioning with $d-1$ or less distinct values. We then find the optimal range partitioning for A_k iteratively. Alg. 1 uses two two-dimensional arrays `cost` and `split`. Array `cost` (resp. `split`) stores at position `[d][s]` the optimal memory footprint $\mathcal{M}$ (resp. partition border) for a range partitioning with d distinct values and the s-smallest value $v_{s_k} \in \Pi^D_{A_k}(R)$ as the lower bound of the range partition.

The first `for` loop (Lines 2 to 10) iterates over the number of distinct values d, while the second `for` loop (Lines 3 to 10) iterates over all possible start positions s. For each combination of d and s, we initialize the `cost` array at position `[d][s]` with the memory footprint $\mathcal{M}$ for a single range partition for the value range $[v_{s_k}, v_{(s+d)_k})$ (or $[v_{s_k}, \infty)$ for the last range) with A_k as a partition-driving attribute (Lines 4 and 5). Sec. 6 and 7 explain how the memory footprint for this single range partition is estimated and calculated. The `split` array is initialized with ∞ to

Algorithm 1: Optimal Range Partitioning Layout

Memory: `cost[d][s]`: optimal memory footprint $\mathcal{M}$ for a range partition of d distinct values starting at value v_{s_k}
`split[d][s]`: optimal partition border for a range partition of d distinct values starting at value v_{s_k}

```
1  Function DP(R, A_k, x_block, v_block):
2      for 1 ≤ d ≤ d_k do //iterate over distinct values
3          for 1 ≤ s ≤ d_k − d + 1 do //iterate over start values
4              v_ub_k ← ∞; if s+d < d_k then v_ub_k ← v_(s+d)_k
5              cost[d][s] ← M(R, A_k, x_block, v_block, v_s_k, v_ub_k)
6              split[d][s] ← ∞ // no partition border
7              for 1 ≤ b < d do // iterate over partition borders
8                  if cost[b][s] + cost[d-b][s+b] < cost[d][s]
9                      cost[d][s] ← cost[b][s] + cost[d-b][s+b]
10                     split[d][s] ← b // partition border v_(s+b)_k
11     S_k ← {} //create range partitioning specification
12     build(d_k, 1, S_k) //build range partitioning specification
13     return (cost[d_k][1], S_k)
14 Procedure build(d, s, &S_k): // pass S_k as reference
15     if split[d][s] = ∞ then S_k = S_k ∪ {v_s_k}
16     else
17         build(split[d][s], s, S_k)
18         build(d-split[d][s], s+split[d][s], S_k)
```

indicate that there is no partition border (Line 6). Afterwards, we check if it is more beneficial to have a partition border at $v_{(s+b)_k}$ ($1 \le b < d$) and update `cost` and `split` accordingly (Lines 7 to 10). For this, we combine the previously calculated optimal range partitioning for b distinct values starting at v_{s_k} with the previously calculated optimal range partitioning for $d-b$ distinct values starting at $v_{(s+b)_k}$. Finally, we build and return the optimal range partitioning specification with its memory footprint $\mathcal{M}$ (Lines 11 to 13). Lines 14 to 18 show the recursive build of the specification based on the `split` array. The complexity of Alg. 1 is $O(d_k^3)$ due to the three `for` loops over the distinct values d_k of attribute A_k.

Correctness. We now prove that Alg. 1 finds the range partitioning specification for a partition-driving attribute A_k with the minimal memory footprint $\mathcal{M}$.

Theorem 5.1. *Alg. 1 finds an optimal range partitioning specification for a partition-driving attribute A_k according to $\mathcal{M}$.*

Proof. We prove the correctness of Alg. 1 by induction over the number of distinct values d for value ranges $[v_{s_k}, v_{(s+d)_k})$. Base case (d=1): The only possible range partitioning specification for the value range $[v_{s_k}, v_{(s+1)_k})$ of any starting value $v_{s_k} \in \Pi^D_{A_k}(R)$ is a single range partition. Alg. 1 is correct since

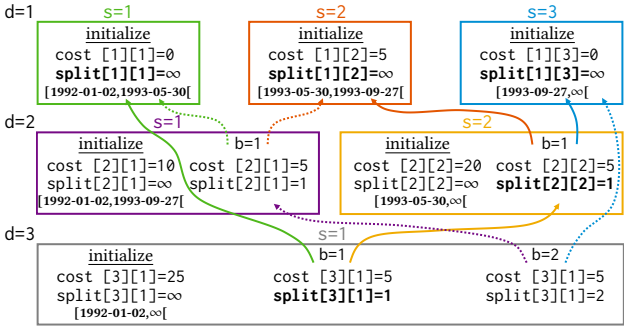


Figure 5: Example of the optimized version of Alg. 1 with `L_SHIPDATE` as a partition-driving attribute.

`cost[1][s]` is initialized with the memory footprint of the single range partition on the range $[v_{s_k}, v_{(s+1)_k})$ (Line 5), `split[1][s]` with the partition border ∞ (Line 6), and both are not updated in Lines 7 to 10 since the condition of the `for` loop is not satisfied for any b if d=1.

Induction step ($d-1 \rightarrow d$): We now prove that Alg. 1 finds the optimal range partitioning specification for the value range $[v_{s_k}, v_{(s+d)_k})$ of any starting value $v_{s_k} \in \Pi^D_{A_k}(R)$ with d distinct values. We assume the induction hypothesis that Alg. 1 finds the optimal range partitioning specification for a value range with less than d distinct values. First, we have to show that Alg. 1 considers that the optimal range partitioning specification can be a single range partition on the value range $[v_{s_k}, v_{(s+d)_k})$. This is considered by the initialization of `cost[d][s]` and `split[d][s]` (Lines 5 and 6). Second, we have to show that Alg. 1 considers that the optimal range partitioning specification can be a combination of optimal range partitioning specifications for the value ranges $[v_{s_k}, v_{(s+b)_k})$ and $[v_{(s+b)_k}, v_{(s+d)_k})$ with a partition border at $v_{(s+b)_k}$ $(1 \leq b < d)$. This is considered since Alg. 1 iterates over all partition borders $v_{(s+b)_k}$ and updates `cost[d][s]` and `split[d][s]` if the sum of the memory footprint for the optimal range partitioning specifications on the value ranges $[v_{s_k}, v_{(s+b)_k})$ and $[v_{(s+b)_k}, v_{(s+d)_k})$ is smaller than `cost[d][s]` (Lines 7 to 10). By the induction hypothesis, the memory footprint of the optimal range partitioning specification for both value ranges can be fetched from `cost[b][s]` and `cost[d-b][s+b]` since b and $d-b$ are smaller than d. Hence, Alg. 1 is correct. □

Optimization. We further optimize the runtime of Algorithm 1 by iterating only over domain blocks (instead of all distinct values) and considering only partition borders between two domain blocks if at least one time window is accessed differently. We still find an optimal range partitioning for uncompressed column partitions by applying both pruning strategies. In contrast, with dictionary compression, we may not find the optimal range partitioning if pruning is applied. If some values occur only in a single column partition, the storage size decreases because a dictionary-compressed column partition may require fewer bits to store the `vid` (since only a subset of the active domain of the attribute is present in this column partition), if additional compression techniques such as bit-packing [60, 71] are applied. We argue that the performance benefit is superior to the pruning of the search space. However, Alg. 1 considers all values and finds the optimal partitioning.

Example. Fig. 5 presents how the optimized version of Alg. 1 finds the optimal range partitioning for `LINEITEM` with `L_SHIPDATE` as a partition-driving attribute for JCC-H Query 3. The domain block counters for `L_SHIPDATE` (SD) in Fig. 4 show only three potential lower bound values of a range partition: 1992-01-02, 1993-05-30, and 1993-09-27. Thus, Fig. 5 shows the iteration over d $(1 \leq d \leq 3)$ horizontally (Lines 2 to 10) and the iteration over the potential lower bound values v_{s_k} (Lines 3 to 10) vertically. For each combination of d and s, we show the initialize step (Lines 4 to 6) of the `cost` and `split` array at position `[d][s]` for a single range partition for the value range $[v_{s_k}, v_{(s+d)_k})$ (or $[v_{s_k}, \infty)$ for the last range). We also denote each step of the iteration over the partition borders at $v_{(s+b)_k}$ $(1 \leq b < d)$ (Lines 7 to 10). The recursive build of the optimal range partitioning specification from the `split` array is highlighted in bold.

5.2 Heuristic Approach MaxMinDiff

Since Alg. 1 finds an optimal partitioning but has cubic complexity, we now present a heuristic to lower optimization time. The idea is to leverage the partition-driving attribute domain block counters and cluster values 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 *MaxMinDiff*, i.e., the number of time windows with accesses to a non-empty and strict subset of the domain blocks, is smaller or equal than a tuning parameter $\Delta \in \mathbb{N}$.

Fig. 6 illustrates the calculation of *MaxMinDiff* for two boundaries `l` and `r`, based on domain block counters (y-axis) of column `ORDERS.O_ORDERDATE` for 200 JCC-H queries during 89 time windows (x-axis). We highlight domain block accesses in red if, for a given time window $\omega \in \Omega$, all domain blocks between `l` and `r` are accessed (22 time windows). Such domain blocks will be grouped into a single partition. In contrast, we highlight domain block accesses in blue if, for a given time window, only a non-empty and strict subset of the domain blocks between `l` and `r` is accessed (16 time windows). Therefore, in this example, *MaxMinDiff* is 16.

Alg. 2 describes the heuristic for finding a near-optimal range partitioning for a partition-driving attribute A_k. 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 $\widehat{\texttt{r}}$ (Lines 2 to 6).

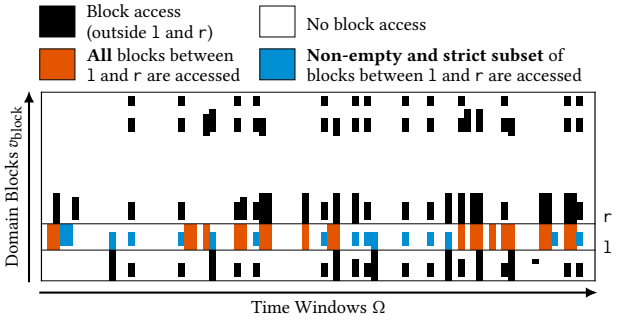


Figure 6: The calculation of MaxMinDiff based on domain block counters of `O_ORDERDATE` after 200 JCC-H queries.

Algorithm 2: Heuristic Approach MaxMinDiff

```
1  Function Heuristic(R, A_k, v_block, l, r, Δ):
2      hot ← l; f ← 0
3      for l ≤ y < r do // search for hottest domain block
4          f̂ ← 0; for ω ∈ Ω do f̂ ← f̂ + v_block(A_k, y, ω)
5          if f̂ > f then hot ← y
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 ← MaxMinDiff(R, A_k, v_block, l̂ − 1, r̂)
10         if r > r̂ then Δ_r ← MaxMinDiff(R, A_k, v_block, l̂, r̂ + 1)
11         if Δ_l > Δ ∧ Δ_r > Δ then break
12         if Δ_l ≤ Δ_r then l̂ ← l̂ − 1 else r̂ ← r̂ + 1
13     S_k ← {} //create range partitioning specification
14     if l < l̂ then S_k ← S_k ∪ Heuristic(R, A_k, v_block, l, l̂, Δ)
15     S_k ← S_k ∪ {v_(l̂ · DBS_k)_k} //partition border at pos l̂ · DBS_k
16     if r > r̂ then S_k ← S_k ∪ Heuristic(R, A_k, v_block, r̂, r, Δ)
17     return S_k
18 Function MaxMinDiff(R, k, v_block, l, r):
19     Diff ← 0
20     for ω ∈ Ω do
21         max ← 0; min ← 1
22         for l ≤ y < r do
23             max ← maximum(max, v_block(k, y, ω))
24             min ← minimum(min, v_block(k, y, ω))
25         Diff ← max − min + Diff
26     return Diff
```

Afterwards, we iteratively extend the current range partition to the left or right as long as *MaxMinDiff* of the current range partition is smaller or equal than Δ (Lines 7 to 12). Lines 18 to 26 show the calculation of *MaxMinDiff*, as illustrated in Fig. 6. For each time window $\omega \in \Omega$ (Lines 20 to 25), we loop over all domain block indexes y between $\mathtt{l}$ and $\mathtt{r}$ (Lines 22 to 24) and add a time window ω to *MaxMinDiff* if at least one (Line 23) but not all domain blocks (Line 24) were accessed during ω, i.e., only a non-empty and strict subset of the domain blocks was accessed within ω. Next, the heuristic is called recursively on all domain block indexes smaller and on all domain block indexes larger than $\widehat{\mathtt{l}}$ and $\widehat{\mathtt{r}}$ (Lines 14 and 16). We also add the current lower bound value $v_{(\widehat{\mathtt{l}} \cdot \mathrm{DBS}_k)_k}$ as partition border to the range partitioning specification $\mathcal{S}_k$ (Line 15). To obtain the index of the value in the domain of A_k, we multiply the domain block index $\widehat{\mathtt{l}}$ by the domain block size DBS_k of A_k (Definition 4.3). Finally, the range partitioning specification $\mathcal{S}_k$ is returned (Line 17).

The heuristic leverages only domain block counters instead of using the cost function. Therefore, the proposed partitioning is only near-optimal, while the complexity is reduced to $O({d_k}^2)$.

6 ACCESS AND STORAGE SIZE ESTIMATOR

We now describe how SAHARA estimates accesses and storage sizes of partitioning layout *candidates* based on collected statistics of the *current* partitioning layout (Sec. 4). We first describe the estimation of column partition accesses and then of storage sizes. These estimates are required in Alg. 1 (Line 5) to initialize single range partitions generated for a value range $[v_{lb_k}, v_{ub_k})$ (or $[v_{lb_k}, \infty)$ for the last range) with a memory footprint $\mathcal{M}$ and to compute an optimal range partitioning specification from it recursively. The calculation of the memory footprint $\mathcal{M}$ of a single range partition is shown in Sec. 7.

6.1 Estimating Column Partition Accesses

We start by estimating an access $\widehat{x}^{\mathrm{col}}$ during a time window ω for the column partition of the partition-driving attribute A_k by leveraging its domain block counters v_{block} (Sec. 4). The estimate depends on whether the domain block counters record at least one access during ω, which falls into the value range $[v_{lb_k}, v_{ub_k})$ of the partition boundaries. If no access exists, we assume that the column partition will not be accessed, e.g., partition pruning is applied.

Definition 6.1. The **estimate of a column partition access** $\widehat{x}^{\mathrm{col}}$ for a time window $\omega \in \Omega$ for a partition-driving attribute A_k with range partition specification boundaries $v_{lb_k}, v_{ub_k} \in \mathcal{S}_k \cup \{\infty\}$ is

$$\widehat{x}^{\mathrm{col}}(A_k, v_{lb_k}, v_{ub_k}, \omega) := \begin{cases} 1 & \exists\, y : v_{\mathrm{block}}(A_k, y, \omega) = 1 \wedge \\ & \lfloor lb_k/\mathrm{DBS}_k \rfloor \le y < \lceil ub_k/\mathrm{DBS}_k \rceil \\ 0 & \text{otherwise.} \end{cases}$$

To estimate accesses to a column partition of a passive attribute A_i, i.e., an attribute different than the partition-driving attribute A_k, we need to consider how the range partition of the partition-driving attribute impacts accesses to the passive attribute, e.g., partition pruning [43] also influences accesses to passive attributes. We argue that three cases exist for estimating the column partition accesses $\widehat{x}^{\mathrm{col}}$ during a time window ω to a passive attribute A_i.

Case 1: The passive attribute was not accessed during ω, i.e., all row block counters of A_i during ω are zero. Thus, the column partition of A_i will not be accessed during ω.

Case 2: The range partition of the partition-driving attribute influences accesses to the passive attribute. This is the case if the set of rows accessed in A_i during ω is a subset of the rows accessed in A_k, i.e., for each local tuple identifier, A_i's row block counter is smaller or equal than A_k's row block counter during ω. We then use the already estimated access $\widehat{x}^{\mathrm{col}}$ during ω from A_k (Definition 6.1).

Case 3: Otherwise, the range partition of the partition-driving attribute does not influence accesses to the passive attribute during ω. We assume that the column partition of A_i will be accessed during ω.

Definition 6.2. We define an **estimate of a column partition access** $\widehat{x}^{\mathrm{col}}$ for a time window $\omega \in \Omega$ for a passive attribute A_i based on a partition-driving attribute $A_k \neq A_i$ with range partition specification boundaries $v_{lb_k}, v_{ub_k} \in \mathcal{S}_k \cup \{\infty\}$ as

$$\widehat{x}^{\mathrm{col}}(A_i, A_k, v_{lb_k}, v_{ub_k}, \omega) := \begin{cases} 0 & \forall\, j, z : x_{\mathrm{block}}(A_i, P_j, z, \omega) = 0 \\ \widehat{x}^{\mathrm{col}}(A_k, v_{lb_k}, v_{ub_k}, \omega) & \\ \quad \forall\, j, \mathtt{lid} : & \\ \quad x_{\mathrm{block}}(A_i, P_j, \lfloor \mathtt{lid}/\mathrm{RBS}_{i,j} \rfloor, \omega) & \\ \quad \le x_{\mathrm{block}}(A_k, P_j, \lfloor \mathtt{lid}/\mathrm{RBS}_{k,j} \rfloor, \omega) & \\ 1 & \text{otherwise.} \end{cases}$$

For example, based on the row block counters of O_CUSTKEY (CK) and O_ORDERDATE (OD) in Fig. 4, we observe that the rows accessed in CK are a subset of the rows accessed in OD. Further, all accesses to OD read domain values [1992-01-01, 1993-05-29). Consequently, the column partition of the passive attribute CK defined by the value range [1993-05-29, 1998-08-03) of OD will not be accessed.

6.2 Estimating Column Partition Sizes

We now estimate a column partition's storage size, both for partition-driving and passive attributes. The estimate of the uncompressed column partition size depends on the estimated cardinality of the range partition and the attribute data type size in bytes.

Definition 6.3. We define an **estimate of an uncompressed column partition size** $\widehat{||C^u||}$ in bytes for an attribute A_i based on a partition-driving attribute A_k with range partition boundaries $v_{lb_k}, v_{ub_k} \in \mathcal{S}_k \cup \{\infty\}$ as

$$\widehat{||C^u||}(A_i, A_k, v_{lb_k}, v_{ub_k}) \coloneqq \texttt{CardEst}(A_k, v_{lb_k}, v_{ub_k}) \cdot ||v_i||,$$

where $\texttt{CardEst}(A_k, v_{lb_k}, v_{ub_k}) \approx \left|\sigma_{v_{lb_k} \leq A_k < v_{ub_k}}(R)\right|$ is a cardinality estimate provided by the database [16] and $||v_i||$ is the average storage size of the data type of attribute A_i.

To estimate a compressed column partition's size, we first estimate the dictionary size, which is influenced by the number of values replicated within the dictionaries of different partitions. Hence, we multiply the estimated distinct count and the attribute data type size in bytes.

Definition 6.4. We define an **estimated dictionary size** $\widehat{||D||}$ in bytes for an attribute A_i based on a partition-driving attribute A_k with range partition specification boundaries $v_{lb_k}, v_{ub_k} \in \mathcal{S}_k \cup \{\infty\}$ as

$$\widehat{||D||}(A_i, A_k, v_{lb_k}, v_{ub_k}) \coloneqq \texttt{DvEst}(A_i, A_k, v_{lb_k}, v_{ub_k}) \cdot ||v_i||.$$

where $\texttt{DvEst}(A_i, A_k, v_{lb_k}, v_{ub_k}) \approx \left|\Pi^D_{A_i}\left(\sigma_{v_{lb_k} \leq A_k < v_{ub_k}}(R)\right)\right|$ is the estimated distinct count provided by the database [16].

The estimated dictionary-compressed column partition size depends on the number of bits needed to represent all `vids` of the attribute's domain within a column partition (assuming bit packing [60, 71]). We multiply this value by the estimated cardinality.

Definition 6.5. The **estimate of a dictionary-compressed column partition size** $\widehat{||C^c||}$ in bytes for an attribute A_i based on a partition-driving attribute A_k with range partition specification boundaries $v_{lb_k}, v_{ub_k} \in \mathcal{S}_k \cup \{\infty\}$ is

$$\widehat{||C^c||}(A_i, A_k, v_{lb_k}, v_{ub_k}) \coloneqq \left\lceil \frac{\lceil log_2(\texttt{DvEst}(A_i, A_k, v_{lb_k}, v_{ub_k}))\rceil}{8 \cdot \texttt{CardEst}(A_k, v_{lb_k}, v_{ub_k})^{-1}} \right\rceil.$$

7 COST MODEL

Based on the estimated accesses and storage sizes (Sec. 6), we can calculate the memory footprint $\mathcal{M}$ of a single range partition for the value range $[v_{lb_k}, v_{ub_k})$. Alg. 1 (Line 5) employs this memory footprint to propose a table partitioning recursively. Moreover, we calculate a buffer pool size $B \in \mathbb{N}$ to fulfill a given performance *SLA*, i.e., the maximum workload execution time.

The idea is that column partitions that are frequently accessed (Def. 7.1) are classified as hot and configured to hold all data in DRAM. Column partitions that are rarely accessed are classified as cold, and data is loaded on-demand from disk upon each read. The buffer pool size is calculated by summing up the sizes of all column partitions classified as hot. To fulfill a given performance *SLA*, the classification depends on the hardware configuration, e.g., disk speed, as well as the number of accesses and the *SLA* itself. To classify column partitions as hot or cold, we consider the five-minute-rule as the cost break-even point of storing data in DRAM versus performing disk I/O for every access [27]. As prices, capacities, and performance of these two storage tiers evolve at a different pace, we refer to the rule as a timeless π-second-rule:

$$\pi \coloneqq \frac{\text{Disk Costs [\$]}}{\text{Disk IOP [Page/s]}} \;/\; \text{DRAM Costs [\$/Page]}. \quad (1)$$

Accordingly, we classify a column partition as hot if it is accessed more often than every π-seconds. A misclassification of a hot column partition as cold induces many expensive disk IOPs, degrades performance, and potentially violates the *SLA*. In contrast, misclassification of a cold column partition as hot increases the DRAM consumption and thus the memory footprint.

Definition 7.1. Given an estimated column partition size $\widehat{||C_{i,j}||}$, an estimated access frequency $\widehat{X}^{\text{col}}_{i,j}$, a maximum workload execution time *SLA*, and π, the **memory footprint of a column partition** $C_{i,j}$ in \$ that fulfills the *SLA* is

$$\mathcal{M}(\widehat{||C_{i,j}||}, \widehat{X}^{\text{col}}_{i,j}, SLA, \pi) \coloneqq \begin{cases} \mathcal{M}_{\text{hot}}(\widehat{||C_{i,j}||}) & \text{if } SLA/\widehat{X}^{\text{col}}_{i,j} \leq \pi \\ \mathcal{M}_{\text{cold}}(\widehat{||C_{i,j}||}, \widehat{X}^{\text{col}}_{i,j}, SLA) & \text{else,} \end{cases}$$

where the access frequency $\widehat{X}^{\text{col}}_{i,j}$ is the sum over all estimated accesses $\widehat{x}^{\text{col}}$ (Sec. 6) of all time windows $\omega \in \Omega$.

According to the π-second-rule, a data item accessed twice within π seconds should be cached in the buffer pool at the second access. Hence, the time window length should not be set substantially smaller than π. Otherwise, statistics could be dominated by many accesses occurring only during a short period, cached in the buffer pool. In addition, the Nyquist–Shannon sampling theorem proves that a sample rate of $\pi/2$ is sufficient to achieve precise statistics [64]. Therefore, we set the time window length to $\pi/2$.

We now specify the cost functions $\mathcal{M}_{\text{hot}}$ and $\mathcal{M}_{\text{cold}}$. The memory footprint of a hot classified column partition is only affected by the estimated column partition size in bytes and the DRAM costs (in \$ per byte) because all data is held in DRAM.

Definition 7.2. Given an estimated column partition size $\widehat{||C_{i,j}||}$, the **memory footprint of a hot column partition** in \$ is

$$\mathcal{M}_{\text{hot}}(\widehat{||C_{i,j}||}) \coloneqq \text{DRAM Costs [\$/B]} \cdot \widehat{||C_{i,j}||}.$$

The memory footprint of a column partition classified as cold considers the estimated column partition size, the estimated number of accesses, the *SLA*, and the hardware configuration because data is fetched for every access.

Definition 7.3. Given an estimated column partition size $\widehat{||C_{i,j}||}$ in bytes, an estimated access frequency $\widehat{X}^{\text{col}}_{i,j}$, and a maximum workload execution time *SLA* in seconds, the **memory footprint of a cold column partition** in \$ is

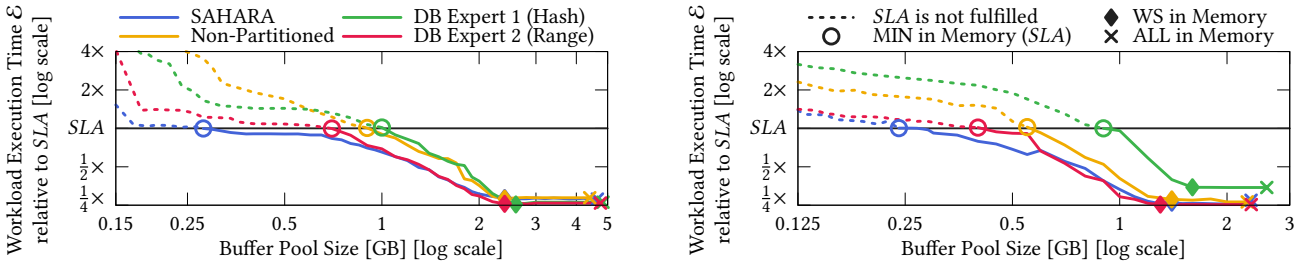
$$\mathcal{M}_{\text{cold}}(\widehat{||C_{i,j}||}, \widehat{X}^{\text{col}}_{i,j}, SLA) \coloneqq \frac{\widehat{X}^{\text{col}}_{i,j}}{SLA[\text{s}]} \cdot \left\lceil \frac{\widehat{||C_{i,j}||}[\text{B}]}{s_p[\text{B/Page}]} \right\rceil \cdot \frac{\text{Disk Costs [\$]}}{\text{Disk IOP}[\frac{\text{Page}}{\text{s}}]}$$

where s_p is the size of a page in bytes.

We propose a buffer pool size based on the hot classified column partitions, such that the performance *SLA* is fulfilled.

Definition 7.4. Given a partitioning layout $\mathcal{L}(R, A_k, \mathcal{S}_k)$, a maximum workload execution time *SLA*, and π, we define the proposed **buffer pool size** B as

$$B(\mathcal{S}_k, SLA, \pi) \coloneqq \sum_{C_{i,j} \in \mathcal{L}(R, A_k, \mathcal{S}_k)} \mathbf{1}\{SLA/\widehat{X}^{\text{col}}_{i,j} \leq \pi\} \cdot \widehat{||C_{i,j}||}.$$



(a) JCC-H (SF 10)

(b) Join Order Benchmark

Figure 7: Comparison of end-to-end workload execution times (y-axis) for varying buffer pool sizes (x-axis) between SAHARA and partitioning layouts proposed by database experts, running the workloads JCC-H and JOB.

Further, two system-specific restrictions exist: A minimum partition cardinality and a page size. First, if the partition cardinality is below a certain threshold, the overhead of scheduling jobs and opening and closing partitions becomes too large, and we assign an infinite memory footprint to the range partition such that Alg. 1 proposes a partitioning, where the cardinality of each range partition is above the threshold. Second, the column partition size is at least the system's disk page size.

8 EXPERIMENTAL EVALUATION

We evaluate the memory footprint reduction achieved by SAHARA (Sec. 8.1), hardware cost savings (Sec. 8.2), the precision of access and storage size estimations (Sec. 8.3), optimality of layouts (Sec. 8.4), and the overhead and optimization time (Sec. 8.5). We implemented SAHARA as a prototype in SAP HANA Cloud [46, 61], a fully automatic advisor that only depends on static hardware or software related properties. First, we discuss the experimental setup.

Hardware: 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 of 8 disks (HGST HUC101812CSS204 HDD) with 10k rpm and a SAS 12 Gbit/s interface.

Workloads: The JCC-H benchmark [10] (scale factor 10) is our first workload. It extends the TPC-H benchmark [69] with data and query skew. For example, special shopping events such as the Black Friday are reflected by corresponding spikes in the `O_ORDERDATE` column of the `ORDERS` table. Our second workload is the Join Order Benchmark (JOB) [40]. JOB consists of 113 queries and uses real-world data from IMDb with data skew and correlations that aggravate estimation errors. We randomly sampled 200 queries for both JCC-H and JOB. Query and data skew, as well as data correlation, pose a challenging environment for SAHARA.

Parameters: We calculate $\pi = 70$ by inserting the prices, capacities, and performance of our hardware into Equation 1. As a result, we set the time window length to $\pi/2 = 35$, such that we fulfill the Nyquist–Shannon sampling theorem (Sec. 7). Further, we set the minimum partition cardinality to 100,000 based on the multi-threading and partitioning capabilities of SAP HANA Cloud. The page size varies between 4 KB and 16 MB, depending on the column partition data type [65]. Finally, logical tuple identifiers are grouped into blocks of 4 KB, and domain blocks are limited to at most 5000 per attribute, such that 1% additional memory is spent on data access counters compared to the data set size. Overall, the parameters are neither workload-specific nor need tuning by a database administrator.

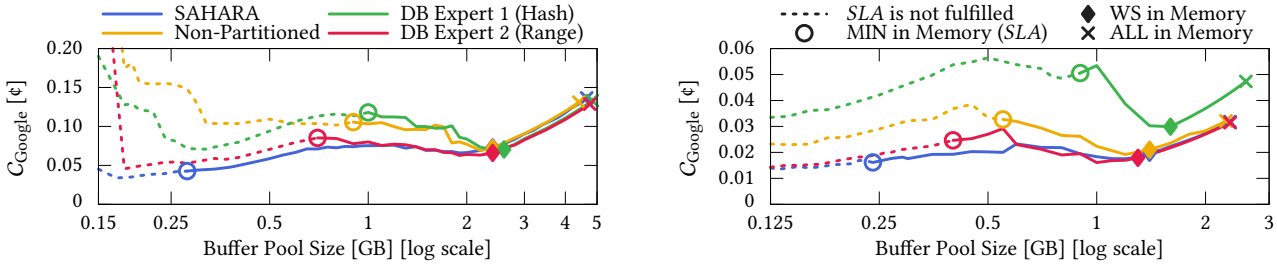
Baseline and Database Experts: To demonstrate SAHARA's effectiveness, we compare SAHARA against combinations of partitioning layouts and buffer pool sizes. As a baseline, we include the non-partitioned layout. Since related approaches (Sec. 9) optimize performance and, therefore, differ in their objective function to SAHARA, we compare ourselves to carefully hand-optimized partitioning layouts for memory footprint reduction and hardware cost savings proposed by experts. For JCC-H, the layout referred to as *DB Expert 1* represents the recommendation [22] of hash-partitioning the primary key columns of `ORDERS` and `LINEITEM`. The layout referred to as *DB Expert 2* represents the recommendation [15] of range-partitioning the columns `O_ORDERDATE` and `L_SHIPDATE`. 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 layout referred to as *DB Expert 2* creates range partitions on columns with selective filter predicates, e.g., on `TITLE.PRODUCTION_YEAR`. The layouts from SAHARA and all database experts are published on https://github.com/SAHARAEngineer/SAHARA.

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 footprint. 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.

8.1 Exp. 1: Memory Footprint Reduction

The first experiment analyzes the effect of the partitioning layouts on the minimal required buffer pool size, i.e., the smallest memory footprint to fulfill a performance *SLA* provided by a customer. As *SLA*, we choose a maximum workload execution time 4× slower than the in-memory workload execution time $\mathcal{E}$ on a non-partitioned layout. For other *SLA*s, we observed similar behavior.

Fig. 7(a) shows on the y-axis the relative end-to-end workload execution time for the previously explained partitioning layouts of JCC-H. The x-axis represents the buffer pool size. The storage sizes differ for all layouts since the partitioning specification impacts dictionary compression and additional compression techniques such as bit-packing. For instance, hash partitioning



(a) JCC-H (SF 10)

(b) Join Order Benchmark

Figure 8: Comparison of hardware memory costs in ¢ on Google Cloud (y-axis) for varying buffer pool sizes (x-axis) between SAHARA and partitioning layouts proposed by database experts, running the workloads JCC-H and JOB.

produces many duplicate dictionary entries. The execution times of all 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 layout 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. Thus, SAHARA increases the tenant density by 2.5× compared to layouts proposed by experts. 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 *SLA*s.

The measurements for JOB in Fig. 7(b) 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 partitioning layouts due to many duplicate dictionary entries caused by hash partitioning.

8.2 Exp. 2: Hardware Cost Savings

The second experiment analyzes the hardware cost that a DBaaS provider needs to pay for executing the workload. As SAHARA optimizes the memory footprint, we calculate the DRAM and disk costs with a fixed number of CPUs. The task of proposing an appropriate number of CPUs [19, 20] is beyond the scope of the paper. We run the experiment on the introduced on-premise hardware but map the provisioned resource costs to a so called *memory-optimized* Google Cloud instance, priced at $2606.10 per TB/month of DRAM and $80.00 per TB/month for regional standard provisioned disk space (HDD) [26]. While DRAM and disk space are billed per GB on Google Cloud, DBaaS providers can reduce hardware costs internally on a more fine-granular level by placing multiple database instances on the same node. Hence, we consider memory costs C_{Google} of a Google Cloud instance per MB/s in ¢.

Fig. 8(a) shows on the y-axis the memory cost C_{Google} in ¢ for different partitioning layouts of JCC-H and on the x-axis the buffer pool size. We use the same definition of the *SLA* as in Experiment 1. The costs of all 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 is 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.

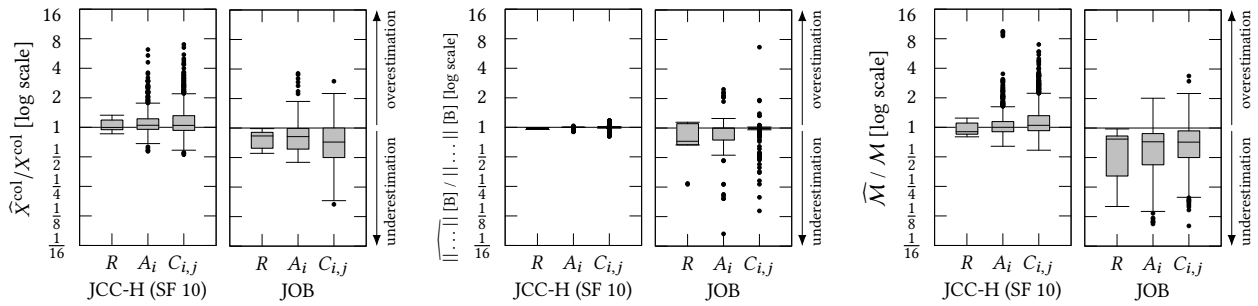
The measurements for JOB in Fig. 8(b) show similar behavior. SAHARA achieves a cost-optimal buffer pool size, still fulfilling the *SLA*, at only 240 MB (0.15¢), while other layouts require a buffer pool size of at least 1000 MB for minimal costs (0.16¢).

8.3 Exp. 3: Precision of Estimates

The third experiment evaluates how precisely SAHARA estimates data accesses, storage sizes, and the memory footprint. We generated for JCC-H 67 and for JOB 37 random partitioning layouts with a random partition-driving attribute. We then compared the estimated and actual values at relation, attribute, and column partition level. For JCC-H (JOB), we analyzed 67 (37) estimates at relation, 1030 (310) at attribute, and 5699 (2237) at column partition level.

Data Accesses. Fig. 9(a) shows the ratio of estimated and actual data accesses at relation, attribute, and column partition level for both JCC-H (left side) and JOB (right side). Overestimation is shown on the top, underestimation at the bottom. Since partition pruning impacts the number of data accesses in a range-partitioned layout and SAHARA proposes a new layout based on the collected statistics, the current layout can impact the precision of the estimates. However, 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. In general, estimates for JCC-H are more accurate than for JOB because JOB is based on the real-world IMDb dataset, whereas the dataset of JCC-H remains synthetic.

Storage Size. Fig. 9(b) shows the ratio of estimated and actual storage size at relation, attribute, and column partition level. We observe that all storage size estimates for JCC-H are bound by a factor of 1.5. For JOB most estimates are bound by a factor of 2. SAHARA tends to underestimate storage sizes because cardinalities in commercial databases tend to be underestimated [40].



(a) Estimated vs. actual data accesses (b) Estimated vs. actual storage sizes (c) Estimated vs. actual memory footprint

Figure 9: Precision of data access $\widehat{X}^{\text{col}}/X^{\text{col}}$, storage size $\widehat{||\ldots||}/||\ldots||$, and memory footprint $\widehat{\mathcal{M}}/\mathcal{M}$ at relation, attribute, and column partition level for random partitioning layouts with random partition-driving attribute.

Memory Footprint. Figure 9(c) shows the ratio of estimated and actual memory footprint at relation, attribute, and column partition level. We observe again that most estimates for JCC-H are bound by a factor of 2, while estimates for JOB are underestimated.

8.4 Exp. 4: Optimality

The fourth experiment evaluates the impact of the estimated memory footprint $\widehat{\mathcal{M}}$ on the output of SAHARA. We created partitioning layouts with the lowest estimated memory footprint $\widehat{\mathcal{M}}$ for all possible partition-driving attributes and number of partitions. We then ran the workload and compared the actual memory footprint $\mathcal{M}$ for each layout against SAHARA, the non-partitioned layout, and the layouts proposed by database experts.

Fig. 10 shows on the y-axis the actual memory footprint $\mathcal{M}$ for 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 (Sec. 8.3), the proposed 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 a higher memory footprint than SAHARA due to a different partitioning specification. DB Expert 1 picks the wrong partition-driving attribute (`L_ORDERKEY`) and has a higher memory footprint than most other layouts. `L_RECEIPTDATE` and `L_COMMITDATE` as partition-driving attributes also have a low memory footprint due to their correlation with `L_SHIPDATE`. We observed similar behavior (not shown) for other tables of JCC-H and JOB.

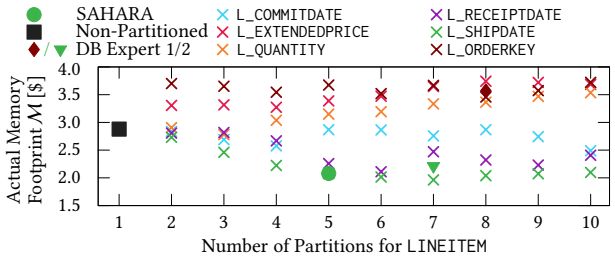


Figure 10: Comparison between SAHARA and other layouts of `LINEITEM` on the actual memory footprint $\mathcal{M}$.

As SAHARA's choice is close to the optimum, it particularly reduces data accesses and increases the compression ratio. The reason is that an increasing number of partitions would separate hot and cold data better into disjoint partitions by reducing the number of accesses and, therefore, reducing the memory footprint. However, an increasing number of partitions would also increase the storage size in most cases due to dictionary duplicates and, therefore, increases the memory footprint. SAHARA instead balances both.

Using the MaxMinDiff heuristic (Alg. 2) instead of Alg. 1 (DP) increases the memory footprint $\mathcal{M}$ (not shown) not at all or by a tiny margin: For JCC-H, `ORDERS` (0.6%) and `LINEITEM` (0.8%); For JOB, `AKA_NAME` (0.1%), `CAST_INFO` (2.9%), `CHAR_NAME` (4.3%), and `MOVIE_INFO` (6.5%). MaxMinDiff provides near-optimal partitioning layouts because the memory footprint increases by at most 6.5%.

In sum, 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.

8.5 Exp. 5: Overhead and Optimization Time

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

The results (Tab. 1) show that SAHARA has a low optimization time and a low memory overhead. The runtime overhead is notable but enables substantial memory footprint and hardware cost savings. To reduce the overhead, statistics may be collected only periodically or sampling is applied. For detailed space and time efficient implementation techniques the reader is referred to [12]. In sum, SAHARA is practical and can be applied in production.

Table 1: Overhead for statistics collection and optimization time for determining the optimal partitioning layout.

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

9 RELATED WORK

Physical Design Advisors. Popular approaches for physical design advice include index advisors [2, 13, 35, 38, 49, 53, 74], storage model advisors [4, 6, 28], table placement advisors [39, 45, 54, 68], and resource advisors for elasticity [19, 20]. We limit the discussion to table partitioning advisors, which are orthogonal to the approaches mentioned above. Data skipping techniques [67] work on a more fine-granular level and can be applied within table partitions. Similarly, Qd-tree [72] analyzes filter predicates and groups rows into pages to minimize I/O cost by routing the queries to the blocks that need to be accessed.

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

Besides SAHARA, Casper [7] is the only table partitioning advisor specifically built for column stores. All other partitioning advisors are mainly designed for row stores. In Casper, the partition-driving attribute has to be provided by the DBA, and only selections are considered. SAHARA instead recommends a partition-driving attribute, estimates data access correlations between passive and partition-driving attributes, and handles all operators.

Schism [17], Clay [63], Horticulture [56], Mesa [50], Hilprecht et al. [31], Strife [58], and Chiller [73] are table partitioning advisors for distributed DBMS and designed for row stores. In particular, they aim to minimize cross-partition transactions by distributing hot accesses evenly across all server nodes. In contrast, the hot and cold partitions proposed by SAHARA intentionally lead to unbalanced access patterns.

Table partitioning advisors in IBM DB2 [34, 59, 74] and Microsoft SQL Server [2, 3, 47] support column stores only partially. For example, IBM DB2 does not support range partitioning for column store tables [33]. 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, SAHARA generates partitioning proposals based on actual data accesses and lower optimization time.

Classification of Hot and Cold Data. Disk-based DBMS employ buffer pools with fixed page sizes to manage data larger than main memory [30]. Replacement policies [23, 55] have been proposed to minimize the number of I/O operations, while the buffer pool size has to be provided by the DBA. Related work [29, 42] showed that a buffer pool induces significant computation and memory overhead when all data fits in memory. To avoid this overhead, in-memory DBMS were initially designed without a buffer pool [9, 36, 66]. However, recent work [41, 51, 65] showed how modern buffer pool designs still achieve in-memory speed. Nevertheless, DRAM remains an expensive resource. Therefore, related work [5, 8, 21, 24, 25, 32, 42, 70] focuses on identifying hot and cold data, intending to move cold data to secondary storage or compressing cold data with a higher compression ratio.

The main difference between related work and SAHARA is their hot and cold classification. While related work requires the DBA to specify a memory budget, SAHARA uses the five-minute-rule to classify data as hot and cold without additional tuning knobs, based only on the hardware and the workload.

Furthermore, the systems differ in the way hot and cold data is identified. Project Siberia [5, 24, 42] and X-Engine [32] leverage access frequencies at row, respectively, at extent granularity, to determine temperatures. Project Siberia collects log samples to estimate the access frequency, while SAHARA counts block-wise and collects actual accesses of the workload. Anti-Caching [21] and LeanStore [41, 51] utilize replacement policies instead of access counters to identify cold data. Unlike both approaches, SAHARA's goal is to propose a range partitioning that separates hot and cold data that necessitates fine-granular access statistics, e.g., on the domain. Also, access frequencies need to be calculated for the cost model. HyPer [25] uses flags of the CPU's MMU for each virtual memory page to identify cold pages for compression. In contrast to our work, they lack a formal definition of the temperature. Hyrise [8] and Mosaic [70] 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* [10, 23, 32], SAHARA classifies hot and cold data at a more fine-granular level and proposes a range partitioning.

MAT [52] collects memory accesses on processors for only analyzing table partitionings and buffer pool sizes. SAHARA instead collects data accesses inside the database and proposes instead analyzing a table partitioning and a buffer pool size.

Cost Models. Query optimizers utilize cost models [40, 48, 62] to minimize query response time. Lomet [44] proposes a cost model for a cost/performance analysis by assigning every operation, either main memory or secondary storage operation costs. SAHARA's cost model instead assigns the memory footprint to a column partition and still fulfills performance *SLA*s. This allows SAHARA to build an optimal table partitioning recursively.

Histograms. The heuristic MaxMinDiff was inspired by histogram construction. While traditional histograms [37, 57] group values based on a one-dimensional domain, e.g., access frequency, MaxMinDiff considers the distribution of accesses over time.

10 CONCLUSION AND FUTURE WORK

We presented the first table partitioning advisor that optimizes the table partitioning on the memory footprint while still fulfilling performance *SLA*s. The proposed range partitioning is based on hot- and cold-classified value ranges, such that pages of hot-classified partitions contain mainly hot data while pages of cold-classified partitions group cold data. This allows to reduce the buffer pool size substantially to keep only pages with a high density of hot data in DRAM but still adhering to all performance *SLA*s. Furthermore, SAHARA's partitioning proposal is based on actual data access statistics. Therefore, SAHARA does not rely on the optimizer's what-if API and is not sensitive to any skew in the distribution of data accesses. In addition, SAHARA collects data accesses from all operators during statistics collection and, therefore, can be used for any workload. Further, SAHARA considers dictionary compression and partition pruning. Both are excellent further opportunities to reduce the memory footprint of databases and are employed in many column stores. Finally, we integrated SAHARA into a prototype of a commercial cloud database and showed that SAHARA reduces the memory footprint (e.g., the buffer pool size) and memory costs by 2.5× compared to database experts while still adhering to all *SLA*s. Therefore, SAHARA is practical for popular free and mature commercial systems.

In future, we plan to predict the future workload based on an observed workload to decide if proactive re-partitioning is beneficial. This is the case, for example, if the re-partitioning costs are amortized by a better fit of the table layout to the future workload.

REFERENCES

[1] Daniel Abadi, Peter A. Boncz, Stavros Harizopoulos, Stratos Idreos, and Samuel Madden. 2013. The Design and Implementation of Modern Column-Oriented Database Systems. *Foundations and Trends in Databases* 5, 3 (2013), 197–280.

[2] Sanjay Agrawal, Surajit Chaudhuri, Lubor Kollar, Arun Marathe, Vivek Narasayya, and Manoj Syamala. 2004. Database Tuning Advisor for Microsoft SQL Server 2005. *PVLDB* (2004), 1110–1121.

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

[4] Ioannis Alagiannis, Stratos Idreos, and Anastasia Ailamaki. 2014. H2O: A Hands-free Adaptive Store. In *SIGMOD '14*. ACM, 1103–1114.

[5] Karolina Alexiou, Donald Kossmann, and Per-Åke Larson. 2013. Adaptive Range Filters for Cold Data: Avoiding Trips to Siberia. *PVLDB* 6, 14 (2013), 1714–1725.

[6] Joy Arulraj, Andrew Pavlo, and Prashanth Menon. 2016. Bridging the Archipelago between Row-Stores and Column-Stores for Hybrid Workloads. In *SIGMOD '16*. ACM, 583–598.

[7] Manos Athanassoulis, Kenneth S. Bøgh, and Stratos Idreos. 2019. Optimal Column Layout for Hybrid Workloads. *PVLDB* 12, 13 (2019), 2393–2407.

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

[9] Peter A. Boncz. 2002. *Monet: A Next-Generation DBMS Kernel For Query-Intensive Applications*. Ph.D. Dissertation. Universiteit van Amsterdam, Amsterdam, The Netherlands.

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

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

[12] Michael Brendle, Nick Weber, Mahammad Valiyev, Norman May, Robert Schulze, Alexander Böhm, Guido Moerkotte, and Michael Grossniklaus. 2021. Precise, Compact, and Fast Data Access Counters for Automated Physical Database Design. In *BTW 2021*. GI.

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

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

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

[16] Graham Cormode, Minos Garofalakis, Peter J. Haas, and Chris Jermaine. 2011. Synopses for Massive Data. *Foundations and Trends in Databases* 4, 1–3 (2011), 1–294.

[17] Carlo Curino, Evan Jones, Yang Zhang, and Sam Madden. 2010. Schism: A Workload-driven Approach to Database Replication and Partitioning. *PVLDB* 3, 1-2 (2010), 48–57.

[18] 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. 2016. The Snowflake Elastic Data Warehouse. In *SIGMOD '16*. ACM, 215–226.

[19] Sudipto Das, Divyakant Agrawal, and Amr El Abbadi. 2013. ElasTraS: An Elastic, Scalable, and Self-Managing Transactional Database for the Cloud. *ACM Trans. Database Syst.* 38, 1, Article 5 (2013), 45 pages.

[20] Sudipto Das, Shoji Nishimura, Divyakant Agrawal, and Amr El Abbadi. 2011. Albatross: Lightweight Elasticity in Shared Storage Databases for the Cloud Using Live Data Migration. *PVLDB* 4, 8 (2011), 494–505.

[21] Justin DeBrabant, Andrew Pavlo, Stephen Tu, Michael Stonebraker, and Stan Zdonik. 2013. Anti-caching: A New Approach to Database Management System Architecture. *PVLDB* 6, 14 (2013), 1942–1953.

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

[23] Peter J. Denning. 1968. The Working Set Model for Program Behavior. *Commun. ACM* 11, 5 (1968), 323–333.

[24] Ahmed Eldawy, Justin Levandoski, and Per-Åke Larson. 2014. Trekking Through Siberia: Managing Cold Data in a Memory-optimized Database. *PVLDB* 7, 11 (2014), 931–942.

[25] Florian Funke, Alfons Kemper, and Thomas Neumann. 2012. Compacting Transactional Data in Hybrid OLTP & OLAP Databases. *PVLDB* 5, 11 (2012), 1424–1435.

[26] Google. 2021. Google Cloud Pricing. https://cloud.google.com/compute/all-pricing. Last accessed on 2021-08-31.

[27] Jim Gray and Franco Putzolu. 1987. 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, 395–398.

[28] Richard A. Hankins and Jignesh M. Patel. 2003. Data Morphing: An Adaptive, Cache-Conscious Storage Technique. *PVLDB* (2003), 417–428.

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

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

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

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

[33] IBM. 2021. IBM DB2: Restrictions, Limitations, and Unsupported Database Configurations for Column-Organized Tables. https://www.ibm.com/docs/en/db2/11.5?topic=to-restrictions-limitations-unsupported-database-configurations-column-organized-tables. Last accessed on 2021-08-31.

[34] IBM. 2021. IBM DB2: The Design Advisor. https://www.ibm.com/docs/en/db2/11.5?topic=strategy-design-advisor. Last accessed on 2021-08-31.

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

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

[37] Robert Philip Kooi. 1980. *The Optimization of Queries in Relational Databases*. Ph.D. Dissertation. Case Western Reserve University, Cleveland, OH, USA.

[38] Jan Kossmann, Stefan Halfpap, Marcel Jankrift, and Rainer Schlosser. 2020. Magic mirror in my hand, which is the best in the land? An Experimental Evaluation of Index Selection Algorithms. *PVLDB* 13, 11 (2020), 2382–2395.

[39] K. Ashwin Kumar, Abdul Quamar, Amol Deshpande, and Samir Khuller. 2014. SWORD: Workload-Aware Data Placement and Replica Selection for Cloud Data Management Systems. *The VLDB Journal* 23, 6 (2014), 845–870.

[40] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter A. Boncz, Alfons Kemper, and Thomas Neumann. 2015. How Good Are Query Optimizers, Really? *PVLDB* 9, 3 (2015), 204–215.

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

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

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

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

[45] Ryan Marcus, Olga Papaemmanouil, Sofiya Semenova, and Solomon Garber. 2018. NashDB: An End-to-End Economic Method for Elastic Database Fragmentation, Replication, and Provisioning. In *SIGMOD '18*. ACM, 1253–1267.

[46] Norman May, Alexander Boehm, and Wolfgang Lehner. 2017. SAP HANA – The Evolution of an In-Memory DBMS from Pure OLAP Processing Towards Mixed Workloads. In *BTW 2017*. GI, 545–546.

[47] Microsoft. 2021. MS SQL Server: Database Engine Tuning Advisor. https://docs.microsoft.com/en-gb/sql/relational-databases/performance/database-engine-tuning-advisor?view=sql-server-ver15. Last accessed on 2021-08-31.

[48] Guido Moerkotte. 2020. *Building Query Compilers*. http://pi3.informatik.uni-mannheim.de/~moer/querycompiler.pdf, Mannheim, Germany. Last accessed on 2021-08-31.

[49] Vikram Nathan, Jialin Ding, Mohammad Alizadeh, and Tim Kraska. 2020. Learning Multi-Dimensional Indexes. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 985–1000.

[50] Rimma Nehme and Nicolas Bruno. 2011. Automated Partitioning Design in Parallel Database Systems. In *SIGMOD '11*. ACM, 1137–1148.

[51] Thomas Neumann and Michael Freitag. 2020. Umbra: A Disk-Based System with In-Memory Performance. *CIDR '20*, Article 29 (2020), 7 pages.

[52] Stefan Noll, Jens Teubner, Norman May, and Alexander Böhm. 2020. Analyzing Memory Accesses with Modern Processors. In *DaMoN '20*. ACM, Article 1, 9 pages.

[53] Matthaios Olma, Manos Karpathiotakis, Ioannis Alagiannis, Manos Athanassoulis, and Anastasia Ailamaki. 2020. Adaptive partitioning and indexing for in situ query processing. *The VLDB Journal* 29, 1 (2020), 569–591.

[54] Oguzhan Ozmen, Kenneth Salem, Jiri Schindler, and Steve Daniel. 2010. Workload-aware Storage Layout for Database Systems. In *SIGMOD '10*. ACM, 939–950.

[55] Elizabeth J. O'Neil, Patrick E. O'Neil, and Gerhard Weikum. 1993. The LRU-K Page Replacement Algorithm for Database Disk Buffering. In *SIGMOD '93*. ACM, 297–306.

[56] Andrew Pavlo, Carlo Curino, and Stanley Zdonik. 2012. Skew-Aware Automatic Database Partitioning in Shared-Nothing, Parallel OLTP Systems. In *SIGMOD '12*. ACM, 61–72.

[57] Viswanath Poosala, Peter J. Haas, Yannis E. Ioannidis, and Eugene J. Shekita. 1996. Improved Histograms for Selectivity Estimation of Range Predicates. In *SIGMOD '96*. ACM, 294–305.

[58] Guna Prasaad, Alvin Cheung, and Dan Suciu. 2020. Handling Highly Contended OLTP Workloads Using Fast Dynamic Partitioning. In *SIGMOD '20*. Association for Computing Machinery, New York, NY, USA, 527–542.

[59] Jun Rao, Chun Zhang, Nimrod Megiddo, and Guy Lohman. 2002. Automating Physical Database Design in a Parallel Database. In *SIGMOD '02*. ACM, 558–569.
[60] Mark A. Roth and Scott J. Van Horn. 1993. Database Compression. In *SIGMOD '93*. ACM, 31–39.
[61] SAP. 2021. SAP HANA Cloud. https://saphanajourney.com/hana-cloud/. Last accessed on 2021-08-31.
[62] P. Griffiths Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. 1979. Access Path Selection in a Relational Database Management System. In *SIGMOD '79*. ACM, 23–34.
[63] Marco Serafini, Rebecca Taft, Aaron J. Elmore, Andrew Pavlo, Ashraf Aboulnaga, and Michael Stonebraker. 2016. Clay: Fine-Grained Adaptive Partitioning for General Database Schemas. *PVLDB* 10, 4 (2016), 445–456.
[64] C. E. Shannon. 1949. Communication in the Presence of Noise. *Proceedings of the IRE* 37, 1 (1949), 10–21.
[65] Reza Sherkat, Colin Florendo, Mihnea Andrei, Rolando Blanco, Adrian Dragusanu, Amit Pathak, Pushkar Khadilkar, Neeraj Kulkarni, Christian Lemke, Sebastian Seifert, and et al. 2019. Native Store Extension for SAP HANA. *PVLDB* 12, 12 (2019), 2047–2058.
[66] 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. 2005. C-Store: A Column-Oriented DBMS. *PVLDB* (2005), 553–564.
[67] Liwen Sun, Michael J. Franklin, Jiannan Wang, and Eugene Wu. 2016. Skipping-Oriented Partitioning for Columnar Layouts. *PVLDB* 10, 4 (2016), 421–432.
[68] Rebecca Taft, Essam Mansour, Marco Serafini, Jennie Duggan, Aaron J. Elmore, Ashraf Aboulnaga, Andrew Pavlo, and Michael Stonebraker. 2014. E-Store: Fine-Grained Elastic Partitioning for Distributed Transaction Processing Systems. *PVLDB* 8, 3 (2014), 245–256.
[69] TPC. 2021. TPC-H Standard Specification. http://tpc.org/tpc_documents_current_versions/pdf/tpc-h_v3.0.0.pdf.
[70] Lukas Vogel, Viktor Leis, Alexander van Renen, Thomas Neumann, Satoshi Imamura, and Alfons Kemper. 2020. Mosaic: A Budget-Conscious Storage Engine for Relational Database Systems. *PVLDB* 13, 12 (2020), 2662–2675.
[71] Till Westmann, Donald Kossmann, Sven Helmer, and Guido Moerkotte. 2000. The Implementation and Performance of Compressed Databases. *SIGMOD Rec.* 29, 3 (2000), 55–67.
[72] Zongheng Yang, Badrish Chandramouli, Chi Wang, Johannes Gehrke, Yinan Li, Umar Farooq Minhas, Per-Åke Larson, Donald Kossmann, and Rajeev Acharya. 2020. Qd-Tree: Learning Data Layouts for Big Data Analytics. In *SIGMOD '20*. ACM, 193–208.
[73] Erfan Zamanian, Julian Shun, Carsten Binnig, and Tim Kraska. 2020. Chiller: Contention-Centric Transaction Execution and Data Partitioning for Modern Networks. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 511–526.
[74] Daniel C. Zilio, Jun Rao, Sam Lightstone, Guy Lohman, Adam Storm, Christian Garcia-Arellano, and Scott Fadden. 2004. DB2 Design Advisor: Integrated Automatic Physical Database Design. *PVLDB* (2004), 1087–1097.