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

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

Nick Weber∗
n.weder@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 2.5x 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.

© 2022 Copyright held by the owner/author(s). Published in Proceedings of the 25th International Conference on Extending Database Technology (EDBT), 29th March-1st April, 2022, ISBN 978-3-89318-086-8 on OpenProceedings.org. Distribution of this paper is permitted under the terms of the Creative Commons license CC-by-nc-nd 4.0.

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 priced 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 SAHARA1, 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.
To balance the load on partitions, they distribute accesses evenly over all partitions and are thus unsuitable for memory footprint reduction as a second level. For large (fact) tables, a multi-level partitioning approach focusing on partition boundary values 1994-12-24 and 1995-01-01 reduces the footprint as a second level.

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].

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 it 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).

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 focuses on range partitioning 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 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. “Data referenced every five minutes should be memory resident” [27].
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 \( A_1, \ldots, A_n \). A range partitioning specification \( S_k = \{a_{k1}, \ldots, a_{kn}\} \subseteq \Pi_{A_i}(R) \) with \( a_{k1} < \ldots < a_{kn} \) and \( a_{kn} = \min(\Pi_{A_i}(R)) \) is a subset of the domain of the partition-driving attribute \( A_k \) (\( 1 \leq k \leq n \)).

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 \( S_k \) from a set of range partitioning specifications \( S_k \). For example, the right side of Fig. 2 shows the proposed range partitioning specification \( S_k = \{1992-01-01, \ldots, 1997-09-02\} \) for partition-driving attribute \( O_{ORDERDATE} \). Further, we call any other attribute \( A_1 \neq A_k \) a passive attribute.

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

\[
P_j := \{ a_{0j} \leq a_{kj} < a_{(j+1)l} \} \quad (j < p_k), \quad a_{0j} \leq a_{kj} \leq a_{(j+1)l}, \quad \text{for all } 1 \leq j \leq p_k.
\]

The partitioning \( \mathcal{P}(S_k) \) 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 \( a_{kj} \) and \( a_{(j+1)l} \) of \( S_k \). For example, Fig. 2 shows the partitioning \( \mathcal{P}(S_k) = \{P_1, \ldots, P_7\} \) generated from the range partitioning specification \( S_k \).

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

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

Definition 3.4. An uncompressed column partition \( C_{ij}^p \) of \( A_i \) (\( 1 \leq i \leq n \)) in \( P_j (1 \leq j \leq p_k) \) of a vector of length \( |P_j| \) with \( C_{ij}^p[1id] = P_j[1id].A_i \) for all \( 1 \leq 1id \leq |P_j| \), where \( P_j[1id].A_i \) retrieves the value of attribute \( A_i \) in partition \( P_j \) for the tuple with the local tuple identifier \( 1id \).

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 \( \Pi_{A_i}^D(P_j) = \{a_{ij1} < \ldots < a_{ij1d_{ij}}\} \) 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 \( D_{ij} = (\text{vid}_{ij}: \Pi_{A_i}^D(P_j) \rightarrow [1, d_{ij}]) \) with \( \text{vid}_{ij}(a_{ijl}) = y_{ijl} \).

The dictionary \( D_{ij} \) of attribute \( A_i \) of partition \( P_j \) is a bijection \( \text{vid}_{ij} \), where \( \Pi_{A_i}^D(P_j) \) is the domain and \( [1, d_{ij}] \) 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 \( C_{ij}^D \) of attribute \( A_i \) in partition \( P_j \) is a vector of numbers in \([1, d_{ij}]\), such that

\[
C_{ij}^D[1id] = \text{vid}_{ij}(C_{ij}^p[1id]), \quad \text{for all } 1 \leq 1id \leq |P_j|.
\]

2.1 Problem Statement

For a given relation with \( n \) attributes, let attribute \( A_k \) (\( 1 \leq k \leq n \)) refer to the partition-driving attribute, e.g., \( O_{ORDERDATE} \) (OD) in the range-partitioned \( \text{ORDERS} \) relation in Fig. 2. For each attribute \( A_k \), there exists a set of potential range partitioning specifications \( S_k \). For example, Fig. 2 shows the range specification \( S_1 = S_3 \) of the \( O_{ORDERDATE} \) (OD) attribute. The problem we consider is to find a partition-driving attribute \( A_k \), a range partitioning specification \( S_k \) of \( S_k \), and a buffer pool size \( B \in \mathbb{N} \) such that the memory footprint \( M \) of a workload \( W \) is minimized (e.g., lower monetary memory costs), while the workload execution time \( E \) for range partitioning specification \( S_k \) 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):

\[
\begin{align*}
\arg\min_{1 \leq k \leq n, S_k \in S_k, B \in \mathbb{N}} & \quad M(S_k, W, B) \\
\text{subject to} & \quad E(S_k, W, B) \leq \text{SLA}.
\end{align*}
\]

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 \( S_k \). 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 \( M \) based on estimated accesses, storage sizes, a given SLA, and the hardware configuration (Sec. 7). A partitioning layout candidate with minimal memory footprint \( M \) 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.
The dictionary-compressed column partition \( C_{i,j}^- \) stores the numbers returned by the bijection \( v_i d_{i,j} \) of the dictionary \( D_{i,j} \) for all values of \( A_i \) for partition \( P_j \).

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

\[
C_{i,j} := \begin{cases} 
(C_{i,j}^a, D_{i,j}) & \text{if } ||C_{i,j}^a|| + ||D_{i,j}|| \leq ||C_{i,j}^-|| \\
C_{i,j}^w & \text{otherwise,}
\end{cases}
\]

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

Fig. 2 shows all column partitions \( C_{i,1}, \ldots, C_{i,7} \) for \( \text{ORDERS} \) generated from the range partitioning specification \( S_2 \). Finally, we define the range partitioning layout as a set of all column partitions \( C_{i,j} \).

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

\[
L(R, A_k, S_k) = \{ C_{i,j} | 1 \leq i \leq n, 1 \leq j \leq p(k) \}.
\]

## 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), \( \Omega \) a set of time windows, and \( R \) a relation. A workload trace \( W \) is then defined as

\[
W \subseteq \{ (\text{gid}, A_i, q, \omega) | 1 \leq \text{gid} \leq |R|, A_1 \leq A_i \leq A_n, q \in Q, \omega \in \Omega \},
\]

where each element in \( W \) denotes a single access to attribute \( A_i \) of the tuple with global tuple identifier \( \text{gid} \) by query \( q \) within time window \( \omega \).

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 \( \omega \) as

\[
\delta_{\text{block}}(A_i, P_j, z, \omega) := \begin{cases} 
1 & \exists \text{gid}, q, 1 \ldots : (\text{gid}, i, q, \omega) \in W \\
& \wedge 1 \ld \in [1, |P_j|] \wedge |P_j|, \text{GID} = \text{gid} \\
& \wedge [1 \ld/\text{RBS}_{i,j}] = z \\
0 & \text{otherwise},
\end{cases}
\]

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 \( \omega \), such that the \( 1 \ld \) 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 \( \text{eval}(i, o, q) \) that evaluates a value \( o \) for a conjunction of predicates in query \( q \)'s WHERE clause on \( A_i \).

**Definition 4.3.** (Domain block counter). Let \( \mathcal{P}^D_{A_i}(R) = \{ v_{\text{hi}} < \ld < v_{\text{lo}}, < \ld_a, \ld_b \ld \} \) 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 \( \omega \) as

\[
\delta_{\text{block}}(A_i, y, \omega) := \begin{cases} 
1 & \exists \text{gid}, v_{\text{hi}}, v_{\text{lo}} : (\text{gid}, i, q, \omega) \in W \\
& \wedge \text{eval}(i, v_{\text{hi}}, q) \wedge R[\text{gid}] = y \\
& \wedge [u_{\text{hi}/\text{RBS}_{i,j}}] = y \\
0 & \text{otherwise},
\end{cases}
\]

where the domain block size \( \text{DBS}_{A_i} \) 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 on 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 \( \mathcal{O} \) to identify the query execution plan operator that caused that access. The selection operators \( \mathcal{O} \) and \( \mathcal{O} \) touch all row blocks of \( \text{CUSTMD} \) and \( \text{ORDER} \), but the respective domain blocks only record if domain values satisfied the \( \text{WHERE} \) clause (Def. 4.3). Therefore, range-partitioning \( \text{ORDERS} \) on \( \text{ORDER} \) with \( 1995-03-09, 1998-08-03 \) would create a column partition that is never accessed. In particular, \( \text{ORDERS} \) has 15 million tuples which are all fetched without partitioning, given the range partitioning specification \( S = (1992-01-01, 1995-05-29) \), we would fetch 3,204,724 tuples. The subsequent hash join \( \mathcal{O} \) 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 \( \text{CUSTMD}(\text{CK}) \) '5004' was filtered out by \( \mathcal{O} \). Hence, the buffer pool is polluted with cold data since all pages but not all rows are read. Next, an index nested loop join \( \mathcal{O} \) touches all row blocks in \( \text{ORDERS} \), but only \( \approx 75\% \) of the row blocks in \( \text{LINEITEM} \). For example, the order with \( \text{LORDR} \) 'OK' '43' comprises 3 million items, which spans multiple blocks, but was already filtered out by \( \mathcal{O} \). The following selection \( \mathcal{O} \) filters all \( \text{L\_SHIPDATE} \) (SD) values smaller than 1993-05-30. Values larger than 1993-09-26 are not read since the \( \text{L\_SHIPDATE} \) (SD) of an item is not 121 days after its \( \text{L\_ORDERDATE} \) (OD) and orders with an \( \text{L\_ORDERDATE} \) (OD) larger
than 1993-05-28 were filtered out by \( \sigma \).

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 \( L_{\text{SHIPDATE}} (SD) \), i.e., 75% of \( L_{\text{LINEITEM}} \) pages are fetched without partitioning, while a partitioning layout based on the range partitioning specification \( S = [1992-01-01, 1993-05-30, 1993-09-27) \) would access only 5% of the pages. While the following group-by operator \( \Pi \) does not create new accesses, the sorting operator \( \Join \) additionally accesses \( L_{\text{DISCOUNT}} (DC) \) and \( L_{\text{EXTENDEDPRICE}} (EP) \). Finally, the projection \( \pi \) accesses only ten blocks of \( L_{\text{SHIPPRICITY}} (SP) \) 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 \( d (1 \leq d \leq d_k) \) 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 \( \text{cost} \) and \( \text{split} \). Array \( \text{cost} \) (resp. \( \text{split} \)) stores at position \([d][s]\) the optimal memory footprint \( \text{cost}[d][s] \) (resp. partition border) for a range partitioning with \( d \) distinct values and the \( s \)-smallest value \( v_s \in \Pi^D_{A_k} (R) \) 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 \( M \) for a single range partition for the value range \([v_s, v_{s+d-1}) \) (or \( v_s, \infty \) 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 \( \infty \) to indicate that there is no partition border (Line 6). Afterwards, we check if it is more beneficial to have a partition border at \( v_{s+b} \) for \( b \leq d \) and update cost and \( \text{split} \) accordingly (Lines 7 to 10). For this, we combine the previously calculated optimal range partitioning for \( b \) distinct values starting at \( v_s \), with the previously calculated optimal range partitioning for \( d-b \) distinct values starting at \( v_{s+b} \), to get the memory footprint for \( d \) distinct values at \( v_s \) (Line 13).

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

**Theorem 5.1.** Alg. 1 finds an optimal range partitioning specification for a partition-driving attribute \( A_k \) according to \( M \).

**Proof.** We prove the correctness of Alg. 1 by induction over the number of distinct values \( d \) for value ranges \([v_s, v_{s+d}) \). Base case \((d=1)\): The only possible range partitioning specification for the value range \([v_s, v_{s+1}) \) of any starting value \( v_s \in \Pi^D_{A_k} (R) \) is a single range partition. Alg. 1 is correct since
We now prove that Alg. 1 finds the optimal range partitioning for \( L_{\text{SHIPDATE}} \) as a partition-driving attribute.

\[
\begin{align*}
\text{cost}_{d}[s][s] & = 0 \\
\text{split}_{d}[s][s] & = \infty \\
\end{align*}
\]

**Example.** Fig. 6 presents how the optimized version of Alg. 1 finds the optimal range partitioning for \( L_{\text{LINEITEM}} \) with \( L_{\text{SHIPDATE}} \) as a partition-driving attribute for JCC-H Query 3. The domain block counters for \( L_{\text{SHIPDATE}} \) (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 \( v_{d,k} \) (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 \([v_{d,k}, v_{(s+d)k}]\) for \( v_{d,k} \) (or \( v_{d,k}, \infty \) 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 \( \Lambda \in \mathbb{N} \).

Fig. 6 illustrates the calculation of MaxMinDiff for two boundaries \( 1 \) and \( r \), based on domain block counters (\( y \)-axis) of column \( O_{\text{ORDERDATE}} \). \( O_{\text{ORDERDATE}} \) 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 1 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 1 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 \( 1 \) 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 \( \hat{1} \) and \( \hat{r} \) (Lines 2 to 6).

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 \( v_{id} \) (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.
Algorithm 2: Heuristic Approach MaxMinDiff

Function Heuristic(R, A_k, vblock, l, r, ∆):
    hot ← l; f ← 0
    for l ≤ y < r do
        // evaluate potential range partitions
        if l < y then
            f ← f + block(A_k, y, ∆)
        if r > y then
            f ← f + block(A_k, y, ∆)
        // initialize range partition
        if ∆ > 0 then
            f ← f + ∆
        // extend range partition
        if ∆ > 0 then
            l ← f
        if ∆ > 0 then
            r ← f + ∆ + 1
        // create range partitioning specification
        lblock ← {l}
        rblock ← {r}
        \( S_{k} \leftarrow \{ \} \)
        \( S_{k} \leftarrow S_{k} \cup \{ [l, DBS, k] \} \)
        \( S_{k} \leftarrow S_{k} \cup \{ [r, DBS, k] \} \)
    \( S_{k} \leftarrow S_{k} \cup \{ [l, DBS, k], [r, DBS, k] \} \)
    return Diff

6.1 Estimating Column Partition Accesses

We start by estimating an access \( \overline{x} \) during a time window \( \omega \) 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 \( \omega \), which falls into the value range \( [l_{ub}, r_{ub}] \) of the partition boundaries. If no access exists, we assume that the column partition will not be accessed, e.g., partition pruning is applied.

\( \text{Definition 6.1.} \ The \ \text{estimate of a column partition access} \ \overline{x} \text{for a time window } \omega \in \Omega \text{ for a partition-driving attribute } A_k \ \text{with range partition specification boundaries} \ \{ l_{ub}, r_{ub} \} \in S_k \) \cup \{ (\infty) \} is

\[
\overline{x}(A_k, v_{lb}, v_{ub}, \omega) := \begin{cases} 
1 & \exists y : v_{block}(A_k, y, \omega) = 1 \land \left[ l_{ub}[DBS, k] \leq y < [ub_{k}/DBS, k] \right] \\
0 & \text{otherwise.}
\end{cases}
\]

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 \( \overline{x} \) during a time window \( \omega \) for a passive attribute A_i.

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

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 \( \omega \) 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 \( \omega \). Then we use the already estimated access \( \overline{x} \) during \( \omega \) 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 \( \omega \). We assume that the column partition of A_i will be accessed during \( \omega \).

\( \text{Definition 6.2.} \ We \ \text{define an estimate of a column partition access} \ \overline{x} \text{for a time window } \omega \in \Omega \text{ for a passive attribute } A_i \based on a partition-driving attribute } A_k \ \text{with range partition specification boundaries} \ \{ l_{ub}, r_{ub} \} \in S_k \cup \{ (\infty) \} as

\[
\overline{x}(A_i, A_k, v_{lb}, v_{ub}, \omega) := \begin{cases} 
0 & \forall j, z : x_{block}(A_i, P_j, z, \omega) = 0 \\
\overline{x}(A_k, v_{lb}, v_{ub}, \omega) & \forall j, \text{id} : \\
\text{x_{block}(A_i, P_j, \text{id}/DBS, k), } \omega \leq x_{block}(A_k, P_j, \text{id}/DBS, k), \omega \text{ otherwise.}
\end{cases}
\]

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 $|C|'$ in bytes for an attribute $A_i$ based on a partition-driving attribute $A_k$ with range partition boundaries $v_{lb_k}, v_{ub_k} \in S_k \cup \{\infty\}$ as

$$|C|'((A_i, A_k, v_{lb_k}, v_{ub_k})) := \text{CardEst}(A_k, v_{lb_k}, v_{ub_k}) \cdot |\sigma_i||,$$

where $\text{CardEst}(A_k, v_{lb_k}, v_{ub_k}) \approx |\sigma_{v_{lb_k} \leq A_k < v_{ub_k}}(R)|$ is a cardinality estimate provided by the database [16] and $|\sigma_i||$ 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 $|\hat{D}|$ in bytes for an attribute $A_i$ based on a partition-driving attribute $A_k$ with range partition specification boundaries $v_{lb_k}, v_{ub_k} \in S_k \cup \{\infty\}$ as

$$|\hat{D}|((A_i, A_k, v_{lb_k}, v_{ub_k})) := \text{DvEst}(A_k, v_{lb_k}, v_{ub_k}) \cdot |\sigma_i||,$$

where $\text{DvEst}(A_k, v_{lb_k}, v_{ub_k}) \approx |\prod_{A_k} |\sigma_{v_{lb_k} \leq A_k < v_{ub_k}}(R)||$ 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 $|\hat{C}|'$ in bytes for an attribute $A_i$ based on a partition-driving attribute $A_k$ with range partition specification boundaries $v_{lb_k}, v_{ub_k} \in S_k \cup \{\infty\}$ is

$$|\hat{C}|'((A_i, A_k, v_{lb_k}, v_{ub_k})) := \frac{|\text{Log}_2(|\text{DvEst}(A_k, v_{lb_k}, v_{ub_k})|)|}{8} \cdot |\text{CardEst}(A_k, v_{lb_k}, v_{ub_k})|^{-1}.$$

7 COST MODEL

Based on the estimated accesses and storage sizes (Sec. 6), we can calculate the memory footprint $M$ 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 criterion. 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 database 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 $\pi$-second-rule:

$$\pi := \frac{\text{Disk Costs \ [$]\}}{\text{Disk IOP [Page/s]}} / \text{DRAM Costs \ [$/Page]}.$$  \hspace{1cm} (1)

Accordingly, we classify a column partition as hot if it is accessed more often than every $\pi$-seconds. A misclassification of a hot column partition as cold induces many expensive disk IOPs, degrades performance, and potentially violates the SLA. In contrast, a 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 $|\hat{C}|'$, an estimated access frequency $\hat{x}_{i,j}$, a maximum workload execution time $\text{SLA}$, and $\pi$, the memory footprint of a column partition $C_{i,j}$ in $\$$ that fulfills the SLA is

$$M(|C_{i,j}|, \hat{x}_{i,j}, \text{SLA}, \pi) := \begin{cases} \frac{M_{\text{hot}}(|C_{i,j}|)}{\pi} & \text{if } SLA/\hat{x}_{i,j} \leq \pi \\ M_{\text{cold}}(|C_{i,j}|, \hat{x}_{i,j}, \text{SLA}) & \text{else}, \end{cases}$$

where the access frequency $\hat{x}_{i,j}$ is the sum over all estimated accesses $\hat{x}_{i,j}$ (Sec. 6) of all time windows $\omega$.

According to the $\pi$-second-rule, a data item accessed twice within $\pi$ seconds should be cached in the buffer pool at the second access. Hence, the time window length should not be set substantially smaller than $\pi$. 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 $M_{\text{hot}}$ and $M_{\text{cold}}$. 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 $|\hat{C}|'$, the memory footprint of a hot column partition in $\$$ is

$$M_{\text{hot}}(|\hat{C}|') := \text{DRAM Costs \ [$/B]} \cdot |\hat{C}|'.$$

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 $|\hat{C}|'$ in bytes, an estimated access frequency $\hat{x}_{i,j}$, and a maximum workload execution time $\text{SLA}$ in seconds, the memory footprint of a cold column partition in $\$$ is

$$M_{\text{cold}}(|\hat{C}|', \hat{x}_{i,j}, \text{SLA}) := \frac{\hat{x}_{i,j}}{\text{SLA}[s]} \left[ \frac{|\hat{C}|'}{|\hat{C}|'}[\text{B}] \right] \frac{\text{Disk Costs \ [$]}}{\text{Disk IOP [Page/s]}} \frac{\text{Page}}{s}$$

where $sp$ 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 $\mathcal{L}(R, A_k, S_k)$, a maximum workload execution time $\text{SLA}$, and $\pi$, we define the proposed buffer pool size $B$ as

$$B(S_k, \text{SLA}, \pi) := \sum_{C_{i,j} \in \mathcal{L}(R, A_k, S_k)} 1(\text{SLA}/\hat{x}_{i,j} \leq \pi) \cdot |\hat{C}|'.$$
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 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 \( \text{ORDERS} \) column of the \( \text{ORDERS} \) 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 \( \text{O_ORDERTIME} \) and \( \text{L_SHIPDATE} \). 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 \( \text{movie_id} \) and the primary key column \( \text{id} \) of table \( \text{TITLE} \). 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 \( \text{TITLE} \), \text{PRODUCT\_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 \text{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 \text{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 \text{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 4x slower than the in-memory workload execution time \( E \) on a non-partitioned layout. For other SLAs, 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
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 SLAs.

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 our 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 $C_{\text{Google}}$ of a Google Cloud instance per MB/s in ϵ.

Fig. 8(a) shows on the y-axis the memory cost $C_{\text{Google}}$ in ϵ 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 0.045 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 (0.066) 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 (0.15x), while other layouts require a buffer pool size of at least 1000 MB for minimal costs (0.16x).

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].
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 $\hat{M}$ on the output of SAHARA. We created partitioning layouts with the lowest estimated memory footprint $\hat{M}$ for all possible partition-driving attributes and number of partitions. We then ran the workload and compared the actual memory footprint $M$ 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 $M$ 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 proposed by database experts. As SAHARA estimates are accurate (Sec. 8.3), the proposed layout with five partitions and $L_{\text{SHIPDATE}}$ 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 ($L_{\text{ORDERKEY}}$) and has a higher memory footprint than most other layouts. $L_{\text{RECEIPTDATE}}$ and $L_{\text{COMMITDATE}}$ as partition-driving attributes also have a low memory footprint due to their correlation with $L_{\text{SHIPDATE}}$. We observed similar behavior (not shown) for other tables of JCC-H and JOB.

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 $M$ (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.

<table>
<thead>
<tr>
<th>Workload</th>
<th>JCC-H</th>
<th>JOB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Statistics Collection: Memory Overhead</td>
<td>0.39%</td>
<td>0.28%</td>
</tr>
<tr>
<td>Statistics Collection: Runtime Overhead</td>
<td>14.84%</td>
<td>18.74%</td>
</tr>
<tr>
<td>Optimization Time: Alg. 1 (DP)</td>
<td>3.06sec</td>
<td>1.45sec</td>
</tr>
<tr>
<td>Optimization Time: Alg. 2 (MaxMinDiff)</td>
<td>0.02sec</td>
<td>0.01sec</td>
</tr>
</tbody>
</table>
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 [29, 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 SLAs. 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 SLAs. 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 SLAs. 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 SLAs. 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.


