Precise, Compact, and Fast Data Access Counters for Automated Physical Database Design

Michael Brendle[1] Nick Weber[2,*] Mahammad Valiyev[3,*] Norman May[4] Robert Schulze[4]
Alexander Böhm[4] Guido Moerkotte[5] Michael Grossniklaus[6]

Abstract: Today's database management systems offer numerous tuning knobs that allow an adaptation of database system behavior to specific customer needs, e. g., maximal throughput or minimal memory consumption. Because manual tuning by database experts is complicated and expensive, academia and industry devised tools that automate physical database tuning. The effectiveness of such advisor tools strongly depends on the availability of accurate statistics about the executed database workload. For advisor tools to run online, workload execution statistics must also be collected with low runtime and memory overhead. However, to the best of our knowledge, no approach collects precise, compact, and fast workload execution statistics for a physical database design tool. In this paper, we present data structures that solve the problem of providing workload execution statistics with high precision, low memory consumption, and low runtime overhead. In particular, we show how existing approaches can be combined and for which advisor tools, new data structures need to be designed. We evaluate our data structures in a prototype of a commercial database and show that they outperform previous approaches using real-world and synthetic benchmarks.

1 Introduction

Modern database management systems (DBMS) offer a plethora of tuning knobs to adapt the system behavior to specific customer needs [Ag04; Ra02]. As a result, finding an optimal configuration that meets all requirements (e. g., with respect to throughput or memory consumption) is usually a difficult task performed by experts. Since manual database tuning by experts is expensive or even infeasible in managed database-as-a-service (DBaaS) environments, academia and industry devised tools for automated physical database design [Lu19]: **(1) Index advisors** improve query performance by creating (clustered) indexes on columns frequently referenced in selective query predicates [Ag04; Ko20; Na20]. **(2) Data compression advisors** reduce the table memory consumption, and thus, the amount of data read and processed by physically compacting columns [Da19; Le10]. **(3) Buffer pool size advisors** lower the Total Cost of Ownership (TCO) by setting the buffer

[1] University of Konstanz, P.O. Box 188, 78457 Konstanz, Germany michael.brendle@uni-konstanz.de
[2] Celonis SE, Theresienstr. 6, 80333 Munich, Germany n.weber@celonis.com
[3] Technical University of Munich, Boltzmannstraße 3, 85748 Garching mahammad.valiyev@tum.de
[4] SAP SE, Dietmar-Hopp-Allee 16, 69190 Walldorf, Germany firstname.lastname@sap.com
[5] University of Mannheim, 68131 Mannheim, Germany moerkotte@uni-mannheim.de
[6] University of Konstanz, P.O. Box 188, 78457 Konstanz, Germany michael.grossniklaus@uni-konstanz.de
* Work done while at SAP SE

doi:10.18420/btw2021-04

pool size to the working set size such that memory costs are minimized without impairing performance [Da16; St06]. Finally, **(4) table partitioning advisors** enable partition pruning, an effective method of reducing the amount of data to be read [ABI19; Ag04; Cu10; Ra02; Se16]. Furthermore, separating frequently accessed (hot) and rarely accessed (cold) data into disjoint partitions can increase the buffer pool hit ratio.

All aforementioned physical database design tools require an objective function, e. g., the workload performance or memory footprint, while respecting given constraints, e. g., a memory budget or maximum workload execution time. To do this, advisor tools consider a set of potential new physical layout alternatives (e. g., by enumeration). For each alternative, the advisor calculates a change in the objective function based on the data, the workload, and the current physical layout. Accurate statistics about the executed workload are of particular importance for the effectiveness of many advisors. For example, index advisors rely on precise knowledge of query predicate selectivities, data compression advisors depend on understanding how much data is sequentially read (e. g., scans) or randomly accessed (e. g., index join), buffer pool size advisors are based on page access statistics, whereas table partitioning advisors build upon row- or value-level access statistics.

Obviously, there is a trade-off between the accuracy of workload execution statistics and their runtime and memory overhead. Ideally, workload execution statistics are collected with low overhead, such that advisor tools can be executed online to adapt to dynamically changing workloads. However, in practice, workload execution statistics are either gathered offline, e. g., by executing a representative sample of the workload on a separate node [Ag04; Cu10; Ra02], or collected with low precision, e. g., by tracking access frequencies at page granularity instead of per row and attribute, combined with sampling [FKN12; Hu19; No20]. As a result, to the best of our knowledge, no approach collects precise, compact, and fast workload execution statistics for an advisor tool.

In this work, we formalize, analyze, and solve the problem of providing workload execution statistics with high precision, low memory consumption, and low runtime overhead as input to automated physical database design tools. Our contributions are as follows:

- we demonstrate and discuss four practical use cases of automated physical database design advice that require workload execution statistics as input (Section 2);
- we define the workload execution statistics that need to be collected, and we subsequently formalize the problem (Section 3);
- we discuss and classify related work with respect to their precision, space efficiency, and runtime overhead (Section 4);
- we present data structures for collecting precise, compact, and fast workload execution statistics (Section 5); and
- we implement our data structures prototypically in SAP HANA and show for each use case that workload execution statistics are provided with high precision and low memory and runtime overhead using real-world and synthetic benchmarks (Section 6).

2 Use Cases of Physical Database Design Advice

This section introduces four use cases of automated physical database design advice in column stores that require workload execution statistics $FStat$ about a workload W. For now, it suffices to think of W as a set of SQL statements and $FStat$ as statistics about W collected during the execution of W.

We argue that automated physical database design tools can be categorized according to their objective function, aiming either for maximum performance or minimum memory footprint. Besides that, advisor tools need to fulfill given constraints, e. g., a memory budget or a maximum workload execution time. In Section 2.1, we introduce an index advisor and a data compression advisor that focus on in-memory performance, i. e., speeding up query response times of given workloads. Section 2.2 presents a buffer pool size advisor and a table partitioning advisor that optimize for memory footprint.

In the following, $\mathcal{R}$ denotes a set of n relations, and $\mathcal{A}(R_i)$ is the set of m_i attributes of relation $R_i \in \mathcal{R}$. Further, $D(A_{i,j}) = \{v_{i,j,1}, .., v_{i,j,k}, .., v_{i,j,d_{i,j}}\}$ refers to the active domain of attribute $A_{i,j} \in \mathcal{A}(R_i)$ with $v_{i,j,1} < .. < v_{i,j,k} < .. < v_{i,j,d_{i,j}}$, where $d_{i,j}$ is the number of distinct values in $A_{i,j}$. Finally, $R_i[\texttt{rid}_i].A_{i,j} \in D(A_{i,j})$ is the value of the row with row id $\texttt{rid}_i \in [1, |R_i|]$ of attribute $A_{i,j} \in \mathcal{A}(R_i)$, where $|R_i|$ is the cardinality of $R_i \in \mathcal{R}$.

2.1 Automated Physical Database Design for Maximizing Performance

Creating a (clustered) index on a column improves the performance if the workload includes selective filter predicates. Traversing the index is then faster than performing a full column scan. Besides that, we assume that a memory budget is given to create indexes only on those attributes where they yield the largest benefit [Ag04; Ko20; Ra02].

Use Case 1 (Index Advisor) *Let $\mathbb{A}_{i,s} \in \wp(\mathcal{A}(R_i))$ be a set of attributes from the power set of all attributes that is uniquely identified by $s \in [1, |\wp(\mathcal{A}(R_i))|]$, $I_{i,s}$ a single-/multi-column index defined over $\mathbb{A}_{i,s}$, and $\mathbb{I}$ the set of all possible indexes over all relations. An index advisor proposes an index configuration $IC \subseteq \mathbb{I}$ such that the estimated execution time $\widehat{\mathcal{E}}$ of a workload W based on workload execution statistics $FStat$ is minimized while the estimated additional memory consumption $\widehat{\mathcal{M}}$ of the indexes adheres to a given memory budget MB:*

$$\underset{IC \subseteq \mathbb{I}}{\arg\min}\ \widehat{\mathcal{E}}(IC, W, FStat) \qquad \textit{subject to } \widehat{\mathcal{M}}(IC) \leq MB.$$

Applying compression to a column may reduce its size, and thus, the amount of data processed by sequential scans. In contrast, compression may deteriorate the time to dereference individual row ids (e. g., during projections) since the decompression of individual rows or blocks may incur multiple random memory accesses, depending on the

compression technique. In practice, robust performance is often preferred, and a column would only be compressed if the speed of critical SQL statements does not decline compared to an uncompressed column [Da19; Le10]).

Use Case 2 (Data Compression Advisor) *Let $\mathbb{C}_{i,j}$ be a set of compressed and uncompressed storage layouts for an attribute $A_{i,j} \in R_i$, $C^u_{i,j} \in \mathbb{C}_{i,j}$ be the uncompressed storage layout, and $W_{crit} \subseteq W$ be the subset of (business) critical SQL statements in the workload, defined by the user. A data compression advisor proposes for each attribute $A_{i,j} \in \mathcal{A}(R_i)$ of each relation $R_i \in \mathcal{R}$ a physical storage layout $C_{i,j} \in \mathbb{C}_{i,j}$ such that the estimated execution time $\widehat{\mathcal{E}}$ of a workload W based on workload execution statistics $FStat$ is minimized, while for each critical SQL statement $q \in W_{crit}$, the estimated execution time $\widehat{\mathcal{E}}$ does not exceed the estimated execution time $\widehat{\mathcal{E}}$ without compression:*

$$\begin{aligned} \underset{\forall R_i \in \mathcal{R} \forall A_{i,j} \in \mathcal{A}(R_i) : C_{i,j} \in \mathbb{C}_{i,j}}{\arg\min} \quad & \widehat{\mathcal{E}}(\{C_{i,j} \mid 1 \le i \le n, 1 \le j \le m_i\}, W, FStat) \\ \textit{subject to} \quad & \forall q \in W_{crit} : \widehat{\mathcal{E}}(\{C_{i,j} \mid 1 \le i \le n, 1 \le j \le m_i\}, q, FStat) \\ & \le \widehat{\mathcal{E}}(\{C^u_{i,j} \mid 1 \le i \le n, 1 \le j \le m_i\}, q, FStat). \end{aligned}$$

2.2 Automated Physical Database Design for Memory Footprint Reduction

A buffer pool size advisor aims for a minimal buffer pool size such that a performance constraint, e. g., a maximum workload execution time, is still fulfilled. To do this, a buffer pool size advisor needs to identify the workload's working set and configure the buffer pool size so that all hot pages can still be held in DRAM.

Use Case 3 (Buffer Pool Size Advisor) *A buffer pool size advisor proposes a minimal buffer pool size $B \in \mathbb{N}$ such that the estimated execution time $\widehat{\mathcal{E}}$ of a workload W based on workload execution statistics $FStat$ does not violate a given threshold SLA:*

$$\underset{B \in \mathbb{N}}{\arg\min} \quad B \qquad \textit{subject to} \quad \widehat{\mathcal{E}}(B, W, FStat) \le SLA.$$

A buffer pool is a simple and practical approach to retain data's hot working set in DRAM. Its most significant drawback is that mixing hot and cold data within the same page pollutes the buffer cache and works against its effectiveness. Table range partitioning separates hot and cold data into disjoint range partitions, and hence, improves the buffer pool hit ratio.

Use Case 4 (Table Partitioning Advisor) *Let $\mathbb{S}_i$ be a set of range partitioning specifications for a relation $R_i \in \mathcal{R}$. A table partitioning advisor proposes a buffer pool size $B \in \mathbb{N}$, and for each relation $R_i \in \mathcal{R}$ a range-partitioning $S_i \in \mathbb{S}_i$ such that the buffer pool size B is minimized, while the estimated execution time $\widehat{\mathcal{E}}$ of workload W with workload execution statistics $FStat$ does not violate a maximum workload execution time SLA.*

$$\underset{B \in \mathbb{N}, R_i \in \mathcal{R} : S_i \in \mathbb{S}_i}{\arg\min} \quad B \qquad \textit{subject to} \quad \widehat{\mathcal{E}}(\{S_i \mid 1 \le i \le n\}, B, W, FStat) \le SLA.$$

3 Problem Statement

We now formalize the problem of providing workload execution statistics *FStat* with high precision, low memory consumption, and low runtime overhead as input to automated physical database design tools. We start this section by defining *FStat* for a workload W and show exemplary *FStat* after executing JCC-H Query 3 [BAK18].

Definition 1 (Workload Execution Statistics) *We define a workload W as a multiset of SQL statements[7] and $T(q)$ as the physical execution plan of a SQL statement $q \in W$. For a workload W, we define workload execution statistics FStat:*

F1 (Index Advisor): For each executed SQL statement $q \in W$, FStat stores for each selection $\sigma_p(R_i)$ on a base relation $R_i \in \mathcal{R}$ in the physical execution plan $T(q)$ that consists of an index-SARGable predicate p, a tuple $(|\sigma_p(R_i)|, \mathcal{F}(p))$, where $|\sigma_p(R_i)|$ is the output cardinality of $\sigma_p(R_i)$ and $\mathcal{F}(p)$ are the free attributes contained in p.

F2 (Data Compression Advisor): For each executed SQL statement $q \in W$, FStat stores for each attribute $A_{i,j} \in R_i$ a pair $(s_{i,j}, r_{i,j})$, where $s_{i,j}$ is the number of rows in $A_{i,j}$ that were sequentially accessed by q (e. g., by a selection $\sigma_p(R_i) \in T(q)$, where p contains $A_{i,j}$), and $r_{i,j}$ is the number of rows that were randomly accessed in $A_{i,j}$ by q (e. g., by a projection $\Pi_{A_{i,j}} \in T(q)$).

F3 (Buffer Pool Size Advisor): For each executed SQL statement $q \in W$, FStat stores the access frequency $f_{P_{i,j,u}}$ to each page $P_{i,j,u} \in \mathbb{P}_{i,j}, u \in [1, |\mathbb{P}_{i,j}|]$ (i. e., $P_{i,j,u}$ stores for a set of rows the values $R_i[\texttt{rid}_i].A_{i,j}$), where $\mathbb{P}_{i,j}$ is the set of all pages of $A_{i,j} \in R_i$.

F4 (Table Partitioning Advisor): For each executed SQL statement $q \in W$, FStat stores the access frequency $f_{v_{i,j,k}}$ for each value $v_{i,j,k} \in D(A_{i,j})$, where $f_{v_{i,j,k}}$ is the sum of

- *the number of sequential reads of $A_{i,j}$ by q such that $\exists R_i[\texttt{rid}_i].A_{i,j} = v_{i,j,k}, \texttt{rid}_i \in [1, |R_i|]$ that is part of the matching rows (e. g., by a selection $\sigma_p(e) \in T(q)$ where p references $A_{i,j}$ and $v_{i,j,k}$ satisfy p)[8], and*
- *the number of random reads of rows in $A_{i,j}$ by q such that $R_i[\texttt{rid}_i].A_{i,j} = v_{i,j,k}, \forall \texttt{rid}_i \in [1, |R_i|]$ (e. g., by a projection $\Pi_{A_{i,j}} \in T(q)$).*

We execute JCC-H Q3 [BAK18] to demonstrate *FStat*. Figure 1 shows the optimal query execution plan, identified by SAP HANAs query optimizer [MBL17].

$\lvert\sigma_p(R_i)\rvert$	$\mathcal{F}(p)$
3,774,696	{ O_ORDERDATE }
299,496	{ C_MKTSEGMENT }

Tab. 1: Collected statistics *FStat F1* for selections $\sigma_p(R_i)$ of JCC-H Q3.

Table 1 shows *FStat F1* for an index advisor. Since the most selective predicate is applied to C_MKTSEGMENT, an index advisor might propose an index on this attribute. Depending on the memory budget, the advisor might also recommend an index on O_ORDERDATE. The selection on L_SHIPDATE is not recorded since it is not performed on a base relation in the plan.

[7] We consider multisets of SQL statements to account for realistic workloads with repeated queries.

[8] We record only accesses to rows that match the predicate since we assume that a range partition generated for a value $v_{i,j,k}$ is pruned if the value does not satisfy the predicate.

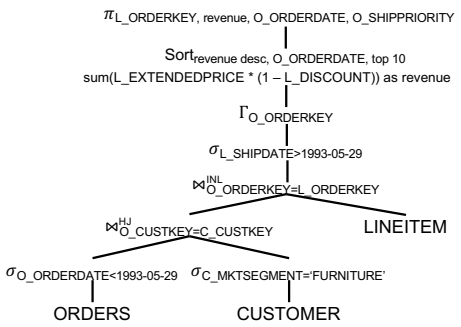


Fig. 1: Optimal query execution plan for JCC-H Q3, identified by SAP HANAs query optimizer.

$A_{i,j}$	$s_{i,j}$	$r_{i,j}$
C_CUSTKEY	0	299,496
C_MKTSEGMENT	1,500,000	0
O_ORDERKEY	0	1,015,311
O_CUSTKEY	0	3,774,696
O_ORDERDATE	15,000,000	377,432
O_SHIPPRIORITY	0	10
L_ORDERKEY	0	3,045,935
L_DISCOUNT	0	1,074,616
L_EXTENDEDPRICE	0	1,074,616
L_SHIPDATE	0	3,045,935

Tab. 2: Collected statistics *FStat F2* about the number of rows that were sequentially ($s_{i,j}$) and randomly ($r_{i,j}$) read for each $A_{i,j}$.

Table 2 shows *FStat F2* for a data compression advisor. Since C_MKTSEGMENT exposes only sequential but no random reads, a data compression advisor might suggest compression. A data compression advisor might also propose compression of O_ORDERDATE since the amount of data processed by sequential scans is reduced. However, random accesses would slow down the time of dereferencing individual row ids due to compression. Therefore, the data compression advisor needs to consider the trade-off between the gain of speeding up sequential reads and the loss of slowing down random accesses.

Figure 2 shows for each 256KB page $P_{i,j,u}$ (x-axis) of L_EXTENDEDPRICE the access frequency $f_{P_{i,j,u}}$ (y-axis), i. e., *FStat F3*. Due to dictionary compression in SAP HANA [MBL17], pages contain either value-id array chunks (600 pages) or dictionary data (40 pages). Since only ≈75% of the value-id array pages are accessed, a buffer pool size advisor might propose reducing the buffer pool size such that all hot pages can still be held in DRAM.

Figure 3 shows for each value $v_{i,j,k}$ of the active domain of O_ORDERDATE (x-axis) the access frequency $f_{v_{i,j,k}}$ (y-axis), i. e., *FStat F4*. A table partitioning advisor might propose a (hot) range-partition for data items with O_ORDERDATE between 1993-01-29 and 1993-05-28 since only those values are accessed frequently. In contrast, data items with O_ORDERDATE larger than 1993-05-28 have an access frequency of 0 and a corresponding (cold) table partition will be pruned by the predicate on O_ORDERDATE.

Problem 1 *The problem we consider is to provide workload execution statistics FStat, which are precise (i. e., as accurate as possible), compact (i. e., the memory footprint compared to the data set size should be as small as possible), and fast (i. e., the runtime overhead during workload execution should be as low as possible).*

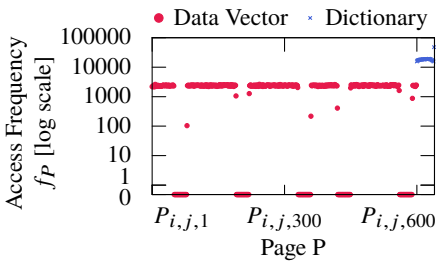


Fig. 2: Collected statistics *FStat F3* about the access frequency $f_{P_{i,j,u}}$ of each page $P_{i,j,u}$ of L_EXTENDEDPRICE for JCC-H Q3.

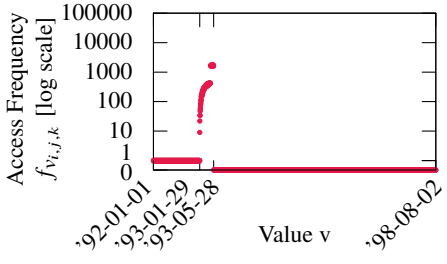


Fig. 3: Collected statistics *FStat F4* about the access frequency $f_{v_{i,j,k}}$ of each value $v_{i,j,k} \in D$(O_ORDERDATE) for JCC-H Q3.

4 Related Work

This section discusses and classifies related approaches of collecting workload execution statistics *FStat* with respect to the precision for use cases F1 to F4, space efficiency, and runtime overhead. The considered approaches are summarized in Table 3.

The first type of workload execution statistics are **row-level data access counters**. Project Siberia [LLS13] analyzes log samples to estimate the access frequency of rows, and SAP ASE [Gu18] caches runtime access patterns of rows. In row stores, this approach yields precise access frequencies of pages (F3). To also track access frequencies of active domain values precisely (F4), separate counters per domain value and attribute are needed, which results in high memory consumption and runtime overhead. Furthermore, with row-level counting, it is unable to deduce the output cardinality of selections (F1). Finally, the total number of rows that were accessed sequentially or randomly can only be tracked if separate counters of each access type exist (F2).

Another class of workload execution statistics are **graphs**. In Schism [Cu10] and Clay [Se16], each row is represented as a node, and edges connect rows if accessed within the same transaction. The weight of an edge denotes the number of transactions that accessed both rows. Graphs are as precise as row-level data access counters. However, the memory and runtime overhead depends on the workload. If transactions touch only a few rows, an adjacency list results in low memory and runtime overhead. In contrast, if transactions touch many rows, both an adjacency list or a matrix result in high memory and runtime overhead.

To further improve the memory and runtime overhead, **block-level data access counters** were proposed. For example, X-Engine [Hu19] leverages access frequencies at extent level collected during workload execution, and HyPer [FKN12] uses for each virtual memory page flags of the CPU's MMU to identify cold pages. Block-level data access counters provide precise access frequencies of pages (F3). The tracking accuracy for accesses to the active domain (F4) depends heavily on the workload and falls short in the presence of heavy

Approach for collecting workload execution statistics	Precise *FStat*				Compact	Fast
	F1	*F2*	*F3*	*F4*		
Row-level data access counters [Gu18; LLS13]	✗	✓	✓	✓	✗	✗
Graph representation [Cu10; Se16]	✗	✓	✓	✓	●	●
Block-level data access counters [FKN12; Hu19]	✗	✓	✓	●	✓	●
SQL statements + What-if API [Ag04; Ra02]	●	●	●	●	✓	✗
Memory access tracing [No20]	✗	✗	✓	✓	✗	✗
Our approach	✓	✓	✓	✓	✓	✓

Tab. 3: Comparison between different approaches for collecting workload execution statistics *FStat* as input to advisor tools with respect to their precision, space efficiency, and runtime overhead.

hitters. The total number of rows sequentially or randomly accessed is available if separate counters for each access type are maintained (F2). The access granularity cannot be tracked as row-level access counters (F1). While block-level access counters are compact, their runtime overhead depends on the workload. In the worst-case, all counters of all blocks accessed need to be incremented (e. g., during a full column scan).

A traditional approach of collecting workload execution statistics is to feed the workload's **SQL statements** into offline physical design advisors, which rely on the query optimizer's **what-if API** [Ag04; Ra02]. While the collected SQL statements are compact, the most significant drawback is that physical accesses to the data are not tracked. Thus, the approach fails to provide accurate statistics as it relies on estimates.

Instead of collecting workload execution statistics inside the database, **memory access tracing** [No20] uses the PEBS mechanism of Intel processors to trace memory accesses, which are mapped to the data to determine precise access frequencies of pages (F3) and values of the active attribute domain (F4). While only single memory accesses are traced, the access granularity (F1) and access type (F2) cannot be identified. Since memory traces are logged and analyzed offline, the memory and runtime overhead is high.

In sum, no approach collects precise, compact, and fast workload execution statistics *FStat* for a physical database design tool. In the next section, we show how existing approaches can be combined and for which advisor tools new data structures need to be designed.

5 Data Access Counters

We begin describing our approach by explaining how precise, compact, and fast workload execution statistics for an index advisor can be collected (Section 5.1). Afterwards, we present data structures for a data compression advisor (Section 5.2), a buffer pool size advisor (Section 5.3), and a table partitioning advisor (Section 5.4).

5.1 Use Case 1: Index Advisor

The most popular approaches of providing workload execution statistics for index advisors (*FStat F1*) consider SQL statements as input to the optimizer's what-if API. As a result, those approaches are limited in their performance due to what-if analysis and rely on the availability of precise cardinality estimates. To address these limitations, we track the actual output cardinalities of selections $\sigma_p(R_i)$ at query execution time. Since tracking the exact output cardinalities $|\sigma_p(R_i)|$ of all selections would consume too much memory, we introduce a threshold parameter $\phi \in (0, 1]$ to capture only selections with an output cardinality less than $\phi \cdot |R_i|$ since only selective predicates benefit from indexes [KAI17]. To reduce the memory overhead further, we group the actual output cardinalities into intervals $[b^r, b^{r+1}), b \in \mathbb{R}_{>0}, 0 \leq r \leq \lceil log_b(\phi \cdot |R_i|) \rceil$ and instead only count the number of selections per interval. The estimated output cardinality for selections that are recorded to the interval $[b^r, b^{r+1})$ is $\sqrt{b^r \cdot b^{r+1}}$. Hence, we determine an error (i. e., the ratio between the actual and recorded output cardinality) of $\sqrt{b}$ for arbitrary complex predicates. In our experiments in Section 6, we set the interval base parameter b to 2, such that the actual and recorded output cardinalities differ at most by a factor of $\sqrt{2}$.

Since an index advisor may recommend multi-column indexes, we would need one set of intervals (i. e., $[b^r, b^{r+1}), b \in \mathbb{R}_{>0}, 0 \leq r \leq \lceil log_b(\phi \cdot |R_i|) \rceil$) per combination of free attributes per relation, i. e., in total, $2^{m_i} - 1$ (= $|\wp(\mathcal{A}(R_i)) \setminus \{\}|$) set of intervals. As a result, the memory consumption of our approach using 32-bit counters for a relation R_i with m_i attributes would be $(\lceil log_b(\phi \cdot |R_i|) \rceil + 1) \cdot (2^{m_i} - 1) \cdot 4$ bytes. To meet the memory requirements, we propose lazy counters, only created if (1) the corresponding combination of free attributes actually occurred in selection predicates and (2) the selectivity of this attribute combination is below ϕ. We argue that this number of attribute combinations is significantly smaller than the number of all attribute combinations. For example, for LINEITEM with scale factor 10 (i. e., 16 attributes and 60,000,000 rows) and $b = 2$, counters for all combinations of free attributes constitute 0.32% of the data set size of LINEITEM in SAP HANA (1.90 GB), while our lazy counters constitute only 0.02% of the data set size.

Section 6 demonstrates that our approach has a high precision as well as a low memory and runtime overhead. We summarize the presented data structure in the following:

Access Counter 1 (Index Advisor)
Physical Accesses: *We consider each selection $\sigma_p(R_i)$ consisting of an index-SARGable predicate, and its actual output cardinality $|\sigma_p(R_i)|$, collected during query execution.*
Lazy Counters: *For a base $b \in \mathbb{R}$ and a set of attributes $\mathbb{A}_{i,s} \in \wp(\mathcal{A}(R_i))$, we create and maintain integer counters $X^{idx}_{i,s,0}, \ldots, X^{idx}_{i,s,r} \ldots, X^{idx}_{i,s,\lceil log_b(\phi \cdot |R_i|) \rceil}$ if there exists a selection $\sigma_p(R_i) \in T(q), q \in W$ such that $\mathbb{A}_{i,s} \subseteq \mathcal{F}(p)$ and $|\sigma_p(R_i)| < \phi \cdot |R_i|$.*
Interval Counting: *A counter $X^{idx}_{i,s,r}$ is incremented by 1 for a selection $\sigma_p(R_i) \in T(q), q \in W$ if $|\sigma_p(R_i)| > 0$ and $r = \lceil log_b(|\sigma_p(R_i)|) \rceil$ and $|\sigma_p(R_i)| < \phi \cdot |R_i|$. For $|\sigma_p(R_i)| = 0$, $X^{idx}_{i,s,0}$ is incremented by* 1.

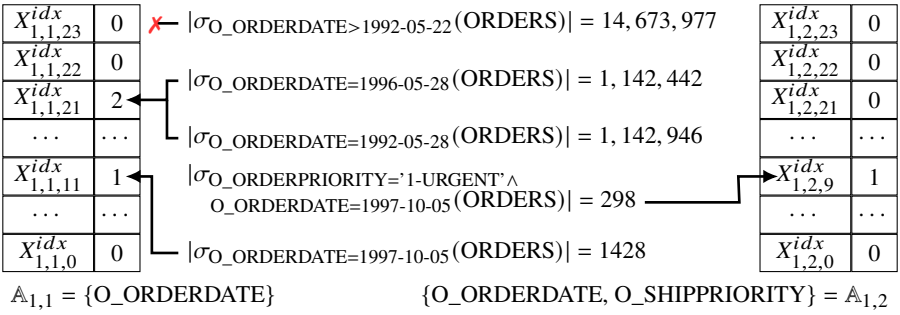


Fig. 4: Illustration of our approach for collecting workload execution statistics for an index advisor.

Figure 4 shows for five selections on ORDERS with scale factor 10 (15,000,000 rows) how the access counters with base $b = 2$ are updated. We show the access counters for selection predicates containing attribute O_ORDERDATE (left), and selection predicates containing O_ORDERDATE and O_ORDERPRIORITY (right). The first selection on O_ORDERDATE matches 14,673,977 rows, and thus no counter is updated for $\phi = 0.1$. The counter $X^{idx}_{1,1,21}$ is updated twice, by the second ($\lceil log_2(1,142,442) \rceil = 21$) and the third selection ($\lceil log_2(1,142,946) \rceil = 21$). The fourth selection updates the counter $X^{idx}_{1,2,9}$ for the attribute set of O_ORDERDATE and O_SHIPPRIORITY as 298 rows match, and two attributes are referenced in the predicate.

As future work, we plan to collect for a join $e \bowtie_{A_{\hat{i},\hat{j}}=A_{i,j}} R_i$, where e is an expression (e. g., $\sigma_p(R_{\hat{i}})$), the cardinality of expression e (i. e., $|e|$) for attribute $A_{i,j}$ of relation R_i. The reason is that an index on an attribute $A_{i,j}$ may improve the performance if $|e|$ is small. Traversing the index on $A_{i,j}$ is then faster than building a hash table on $A_{i,j}$.

5.2 Use Case 2: Data Compression Advisor

In Section 4, we have shown that existing approaches of collecting workload execution statistics for data compression advisors (*FStat F2*) do not consider the type of access (i. e., sequential vs. random access). We propose to count both the number of rows accessed sequentially and randomly by the workload. Maintaining just two counters per attribute fulfills the space efficiency requirement. Section 6 shows that our approach also achieves a low runtime overhead. Note that besides workload execution statistics, characteristics of the data (e. g., number of distinct values, value distribution, or whether the data is sorted) are also needed to propose an optimal compression layout (Use Case 2) [Da19]. Moreover, these statistics are typically available in databases today with sufficient quality. However, workload execution statistics are essential in estimating the performance benefit, particularly for (business) critical queries. We summarize the presented access counter in the following:

Access Counter 2 (Data Compression Advisor)
Physical Accesses: *We consider the physical data accesses during execution of workload W.*
Access Type: *For each attribute $A_{i,j} \in R_i$, we create and maintain an integer counter $X^s_{i,j}$, which tracks the number of rows sequentially read, and an integer counter $X^r_{i,j}$, which tracks the number of rows randomly accessed.*

$A_{i,j}$	$X^s_{i,j}$	$X^r_{i,j}$
C_CUSTKEY	0	299,496
C_MKTSEGMENT	1,500,000	0
O_ORDERKEY	0	1,015,311
O_CUSTKEY	0	3,774,696
O_ORDERDATE	15,000,000	377,432
O_SHIPPRIORITY	0	10
L_ORDERKEY	0	3,045,935
L_DISCOUNT	0	1,074,616
L_EXTENDEDPRICE	0	1,074,616
L_SHIPDATE	0	3,045,935

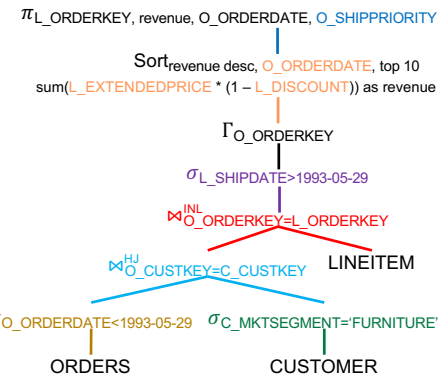


Fig. 5: Illustration of the data structure for collecting *FStat F2* for a data compression advisor.

Figure 5 shows for JCC-H Q3 how $X^s_{i,j}$ and $X^r_{i,j}$ are updated. Note that these statistics are actual values from the execution with SAP HANA. Data accesses by an operator in the plan and updating the corresponding counter are highlighted using a unique color. The selection on O_ORDERDATE causes 15,000,000 sequential row accesses, while the join between ORDERS and CUSTOMER causes 299,496 random row accesses to C_CUSTKEY and 3,774,696 random accesses to O_CUSTKEY (a customer has on average 10 orders). The projection on O_SHIPPRIORITY generates 10 random row accesses due to the top-10 query.

5.3 Use Case 3: Buffer Pool Size Advisor

Block-level data access counters provide precise access frequencies of pages if the block size equals the page size. However, keeping track of accesses that span multiple pages requires updating $|\mathbb{P}_{i,j}|$-many block counters. Instead of updating for each query the frequencies of all touched pages individually, we propose to update only the respective start and end page counters: If a query accesses the pages $[P_{i,j,v}, P_{i,j,w}), P_{i,j,v}, P_{i,j,w} \in \mathbb{P}_{i,j}$, the corresponding counter to page $P_{i,j,v}$ is incremented, while the counter of page $P_{i,j,w+1}$ is decremented since $P_{i,j,w}$ is the last accessed page. This enables counter updates in constant time. Since we decrement the counter of the following page, in total $|\mathbb{P}_{i,j} + 1|$ counters are needed to be able to decrement a counter for accesses to the last page $P_{i,j,|\mathbb{P}_{i,j}+|}$. After

statistics collection, the final page access frequencies are derived by calculating the prefix sum of the counters up to the target page. We argue that the statistics are considerably more often updated than read (e. g., after a sampling phase) and that we thus meet the runtime overhead requirements. Furthermore, the memory overhead is low because only a single 64-bit signed integer counter per page is stored. For example, in SAP HANA [Sh19] the memory footprint varies between 0.2% (64 bit/4 KB) and 0.00005% (64 bit/16 MB), depending on the page size. We present the data structure below:

Access Counter 3 (Buffer Pool Size Advisor)
Physical Accesses: *We consider the physical data accesses by the workload W.*
Start/End Block Counting: *For each attribute $A_{i,j} \in R_i$, we create and maintain integer counters $X^P_{i,j,1}, \ldots, X^P_{i,j,v}, \ldots, X^P_{i,j,(|\mathbb{P}_{i,j}+1|)}$. For physical accesses to pages in the range $[P_{i,j,v}, P_{i,j,w}[, P_{i,j,v}, P_{i,j,w} \in \mathbb{P}_{i,j}$, counter $X^P_{i,j,v}$ is incremented by 1, and counter $X^P_{i,j,(w+1)}$ is decremented by 1. The access frequency $f_{P_{i,j,u}}$ for page $P_{i,j,u}$ is defined as $f_{P_{i,j,u}} = \sum_{v=1}^{u} X^P_{i,j,v}$.*

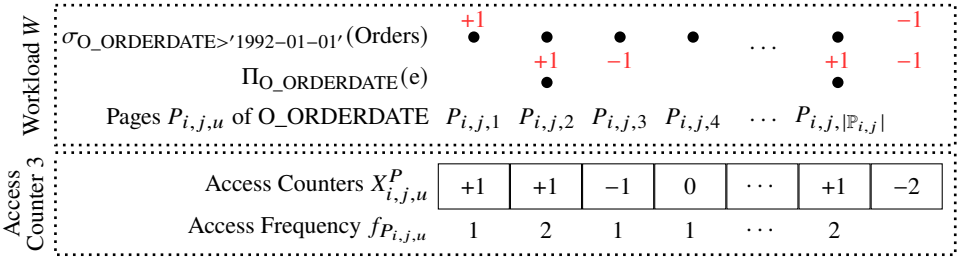


Fig. 6: Illustration of the data structure for collection *FStat F3* for a buffer pool size advisor.

Figure 6 shows for a selection and a projection on O_ORDERDATE how the page accesses are counted. The selection changes counter $X^P_{i,j,1}$ by +1 and counter $X^P_{i,j,(|\mathbb{P}_{i,j}+1|)}$ by −1, while the projection increments only the counter of the accessed page by +1 and decrements the counter of the following page by −1. Note that for accesses to the last page, the counter $X^P_{i,j,(|\mathbb{P}_{i,j}+1|)}$ is decremented. We compute the prefix sum of the counters up to the target page to obtain the access frequencies of individual pages, e. g., page $P_{i,j,2}$ has an access frequency of 2 ($= X^P_{i,j,1} + X^P_{i,j,2}$).

5.4 Use Case 4: Table Partitioning Advisor

A naïve approach of tracking the access frequencies of values in the active attribute domain (*FStat F4*) is to group values into value ranges and to increment a value range counter by one whenever a value or sub-range of the value range is read. With the counter representing

the access frequency of each value in the range, frequencies are overestimated substantially. Instead, we propose to count the number of actually accessed values. A single random read would increase the counter by one, whereas a full column scan would increment the counter by the number of values in the range (i. e., the block size). The access frequency of a value is then obtained by dividing the value range counter by the block size. The calculated access frequencies are nevertheless prone to skewed access patterns. More specifically, access frequencies of heavy hitters are underestimated, whereas frequencies of rarely accessed values (i. e., the long tail) are overestimated.

To improve precision in such situations, we propose to employ the space-saving algorithm and its stream-summary data structure [MAE05] in order to monitor the top-h most frequently accessed values of a value range. However, depending on h, not all values stored in the stream-summary are true heavy hitters. To identify actual heavy hitters from the values stored in the stream-summary, we additionally consider each values' value range counter. Since the stream-summary substantially overestimates access frequencies of rarely accessed values, we argue that the estimated frequency of a heavy hitter must not be significantly larger than its corresponding value range counter. Since the stream-summary also tends to overestimate heavy hitters, we tolerate a slightly larger estimated access frequency. Therefore, we introduce a tolerance parameter λ, such that the estimated access frequency of the stream-summary is only considered if its estimate is at most λ-times larger than its corresponding value range counter.

To calculate the access frequency of a value, we first check if the corresponding value range contains heavy hitters. If this is the case, we subtract their accumulated access count from the value range counter. The estimated access frequency of values from the long tail is given by the remaining block count divided by the number of values from the long tail in the value range. The estimated access frequency of heavy hitters is simply taken from the stream-summary.

Our approach can be tuned to fulfill the space requirement by configuring the block size and the number of heavy hitter candidates tracked by the stream-summary data structure. We show in Section 6 that our approach also achieves high precision while having a low runtime overhead. The presented data structure is summarized in the following:

Access Counter 4 (Table Partitioning Advisor)
Block Counting: *For each attribute $A_{i,j} \in R_i$, we create counters $X^{val}_{i,j,0}, \ldots, X^{val}_{i,j,b}, \ldots, X^{val}_{i,j,\lfloor d_{i,j}/b_{i,j} \rfloor}$, where the block size $b_{i,j}$ is the number of values grouped into a block.*
Stream-summary*: For each attribute $A_{i,j} \in R_i$, we create a stream-summary data structure $SS^h_{i,j}$ such that $D(SS^h_{i,j})$ is the domain of the monitored top-h most frequently accessed values. For a value $v_{i,j,k}$, the estimated access frequency is given by $SS^h_{i,j}(v_{i,j,k})$ if $v_{i,j,k} \in D(SS^h_{i,j})$, otherwise* 0.
Physical Accesses: *We consider the physical data accesses during execution of workload W. For a sequential read on $A_{i,j}$, $X^{val}_{i,j,b}$ is incremented by the number of values that fall into*

the given block and have at least one matching row. The values are also inserted into $SS^h_{i,j}$. *For a random read* $R_i[\texttt{rid}_i].A_{i,j} = v_{i,j,k}, \texttt{rid}_i \in [1, |R_i|]$, $X^{val}_{i,j,\lceil k/b_{i,j}\rceil}$ *is incremented by* 1, *and the value is inserted into* $SS^h_{i,j}$.

Access Frequency*: The estimated access frequency* $\widehat{f}_{v_{i,j,k}}$ *is calculated as follows:*

$$\widehat{f}_{v_{i,j,k}} = \begin{cases} SS^h_{i,j}(v_{i,j,k}) & \text{if } isHH(v_{i,j,k}) \\ \left\lceil \left(X^{val}_{i,j,\lfloor k/b_{i,j}\rfloor} - numHHAccesses\right) / \left(b_{i,j} - numHH\right) \right\rceil & \text{otherwise,} \end{cases}$$

$$\text{where} \quad isHH(v_{i,j,k}) = \begin{cases} 1 & \text{if } v_{i,j,k} \in D(SS^h_{i,j}) \wedge SS^h_{i,j}(v_{i,j,k}) \leq \lambda \cdot X^{val}_{i,j,\lfloor k/b_{i,j}\rfloor} \\ 0 & \text{otherwise.} \end{cases}$$

$$numHH = \sum_{k'=\lfloor k/b_{i,j}\rfloor \cdot b_{i,j}}^{\lceil k/b_{i,j}\rceil \cdot b_{i,j}-1} isHH(v_{i,j,k'})$$

$$numHHAccesses = \sum_{k'=\lfloor k/b_{i,j}\rfloor \cdot b_{i,j}}^{\lceil k/b_{i,j}\rceil \cdot b_{i,j}-1} isHH(v_{i,j,k'}) \cdot SS^h_{i,j}(v_{i,j,k'}).$$

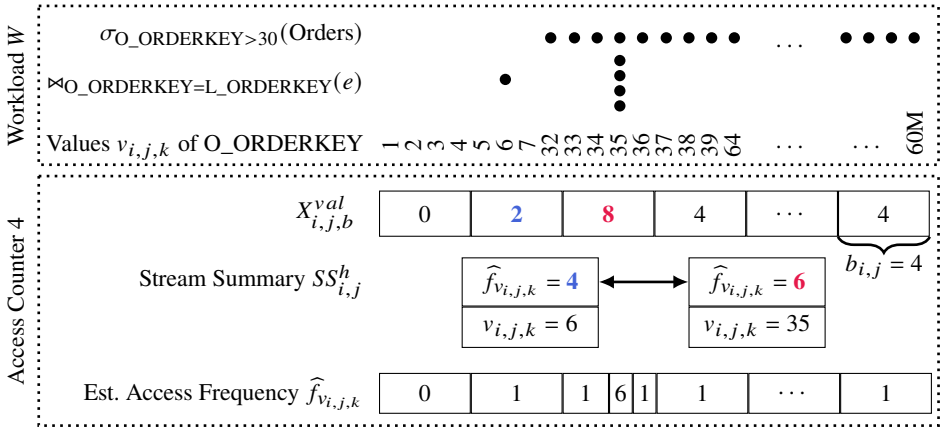


Fig. 7: Illustration of the data structure for collection *FStat F4* for a partitioning advisor.

Figure 7 shows for a selection and a join of attribute O_ORDERKEY how the access frequencies of values are estimated based on the block counter and the stream-summary. For example, the value 35 stored in the stream-summary is a heavy hitter as 6 is not larger than $\lambda \cdot X^{val}_{i,j,2}$ with $\lambda = 1.2$. Therefore, the counter $X^{val}_{i,j,2}$ is decremented by 6, which results in an estimated access frequency of 1 for the values from the long tail, i. e., 33, 34, and 36. In contrast, value 6 is not classified as a heavy hitter as the estimated access frequency 4 is more than λ-times larger than $X^{v,l}_{i,j,1}$ with $\lambda = 1.2$.

6 Experimental Evaluation

We evaluate the presented access counters with respect to their precision, space efficiency, and runtime overhead using real-world and synthetic benchmarks for an index advisor (Section 6.1), a data compression advisor (Section 6.2), a buffer pool size advisor (Section 6.3), and a table partitioning advisor (Section 6.4). We implemented our access counters prototypically in SAP HANA [MBL17]. First, we discuss the experimental setup.

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

The first workload is the synthetic TPC-H benchmark [TP18] with scale factor 10, consisting of 22 templated queries. To create a challenging environment for our access counters, we consider as second workload the JCC-H benchmark [BAK18] (scale factor 10), which extends TPC-H with data and query skew. For example, special shopping events such as Black Friday are reflected by corresponding spikes in `O_ORDERDATE`. To cover the experiments in an acceptable time, we excluded queries Q9, Q16, Q20, and Q21 for JCC-H since parameter combinations led to query execution times larger than five minutes due to the data and query skew. Our third workload is the Join Order Benchmark (JOB) [Le15]. JOB consists of 33 different query templates (113 different queries in total) and uses real-world data from IMDb with data skew and correlations that aggravate estimation errors.

For the evaluation, we randomly generated for each benchmark a workload of 200 queries. The following table shows how often each templated query occurs in each workload:

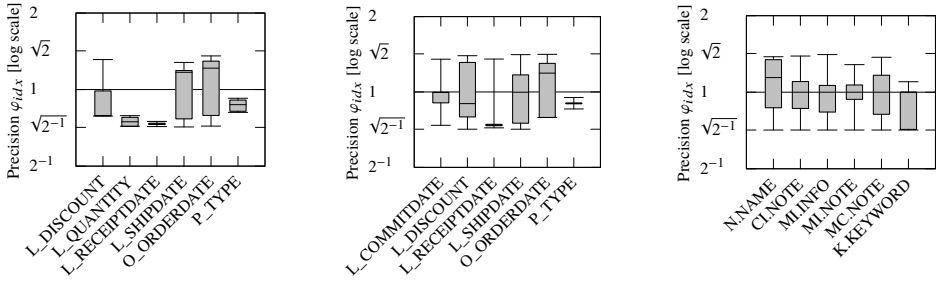
Query ID	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33
TPC-H	8	11	11	5	14	16	9	5	8	11	5	11	10	7	12	4	8	8	10	9	9	9											
JCC-H	11	15	7	14	19	9	9	12	–	7	14	10	12	8	9	–	9	14	10	–	–	11											
JOB	5	6	4	4	7	11	4	5	5	3	6	2	9	4	6	11	12	16	17	7	4	9	5	5	5	7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

6.1 Use Case 1: Index Advisor

We start by evaluating Access Counter 1 for collecting workload execution statistics for an index advisor. Since we group actual output cardinalities into intervals $[b^r, b^{r+1})$ and count only the number of selections per interval, we calculate the precision of our approach by dividing the estimated output cardinality (i. e., $\sqrt{b^r \cdot b^{r+1}}$) by its actual output cardinality: $\varphi_{idx} = |\widetilde{\sigma_p(R_i)}|/|\sigma_p(R_i)|$. In our experiments, we set the interval base parameter b to 2. Hence, the actual and recorded output cardinalities differ at most by a factor of $\sqrt{2}$.

Figure 8 shows for six attributes $\mathbb{A}_{i,s} \subseteq \mathcal{F}(p), \forall \sigma_p(R_i) \in T(q), \forall q \in W$ of each benchmark the precision φ_{idx}, i. e., the ratio of estimated and actual output cardinalities. Overestimation is shown on the top, underestimation at the bottom. Each boxplot shows the 0.00, 0.25, 0.5,

0.75, and 1.00 percentiles. We observe for all attributes and all benchmarks that φ_{idx} of all selections is at most $\sqrt{2}$ in accordance with our choice of b.



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

Fig. 8: Precision of our approach for collecting workload execution statistics for an index advisor.

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

Tab. 4: Precision, space efficiency, and runtime overhead compared to precise counters.

Table 4 shows the results with respect to precision, space efficiency, and runtime overhead of precise counting (i. e., one counter per output cardinality) and our approach (i. e., lazy counters and interval counting). While precise counting achieves perfect precision, its memory overhead varies between 8.4% and 10.6% and is thus substantial. Our approach instead still attains reasonably accurate estimates, differing at most by a factor of $\sqrt{2}$. The memory overhead is also negligible due to lazy counting in combination with intervals. Both approaches yield a low runtime overhead since only the actual output cardinalities of selections are tracked. We conclude that our access counters are precise, compact, and fast.

6.2 Use Case 2: Data Compression Advisor

We now evaluate Access Counter 2 for collecting workload execution statistics for a data compression advisor. Table 5 shows the results with respect to precision, space efficiency, and runtime overhead. Our approach is 100% precise since, for each attribute, the exact number of rows accessed sequentially

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

Tab. 5: Precision, space efficiency, and runtime overhead for our access counters of a data compression advisor.

and randomly by the workload is counted. Maintaining just two 64-bit integer counters per attribute is also space-efficient. For example, for the Join Order Benchmark with 108 attributes in 21 relations, the total memory consumption is only 1.73 KB (= 108 · 16 bytes). Compared to the data set size in SAP HANA (2.28 GB), this represents only 0.00008%. As the runtime overhead is also low (between 4.7% and 9.1%), we conclude that our access counters for a data compression advisor are precise, compact, and fast.

6.3 Use Case 3: Buffer Pool Size Advisor

In the third experiment, we evaluate Access Counter 3 for collecting workload execution statistics for a buffer pool size advisor. Table 6 shows the results with respect to the precision, space efficiency, and runtime overhead of naïve block-level counting (i. e., updating the frequencies of all touched pages) and our approach (i. e., updating only the frequencies of start and end pages). Both approaches are 100% precise since, for each memory page, all physical accesses are tracked. Compared to the data set size, the memory overhead is at most 0.2% compared to the tables data size, given the smallest page size of 4 KB in SAP HANA (64 bit/4 KB) [Sh19]. We use one signed 64-bit integer counter per page as counters may become negative. The runtime overhead of naïve block-level counting varies between 8.3% and 21.8%. Our approach results only in a runtime overhead between 5.2% and 13.5%, as updates to the counter are done in constant time for queries that span multiple pages. We conclude that our access counters are precise, compact, and fast.

	Naïve Block-Level Counting			Our Approach		
Workload	TPC-H	JCC-H	JOB	TPC-H	JCC-H	JOB
Precision	100% precise			100% precise		
Memory Overhead	≤ 0.2%	≤ 0.2%	≤ 0.2%	≤ 0.2%	≤ 0.2%	≤ 0.2%
Runtime Overhead	8.3%	13.1%	21.8%	5.2%	9.2%	13.5%

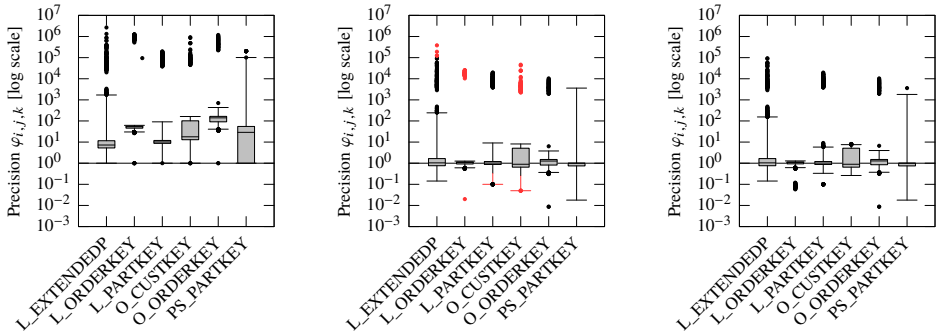
Tab. 6: Precision, space efficiency, and runtime overhead compared to naïve block access counters.

6.4 Use Case 4: Table Partitioning Advisor

Finally, we evaluate Access Counter 4 for collecting workload execution statistics for a table partitioning advisor. To fulfill the space efficiency requirement, we limit the access counters' memory footprint to 1% compared to the column size (encoded column and dictionary). For example, for O_ORDERDATE (23 MB, 2406 distinct values), we create one counter per domain value, while for O_ORDERKEY (105MB, 15,000,000 distinct values), domain values are grouped into ranges of 115 values each. We also maintain a stream-summary for attributes with a block size larger than one to track the top-100 most frequently accessed values. Finally, we set $\lambda = 1.2$, i. e., a value is classified as heavy hitter if its access frequency estimated by the stream-summary is at most 1.2× larger than its value range counter. We experimentally evaluated $\lambda = 1.2$ as a good choice. To calculate the

precision of a value $\varphi_{i,j,k}$, we divide the estimated access frequency by the actual access frequency, i. e., $\varphi_{i,j,k} = \widehat{f_{v_{i,j,k}}} / f_{v_{i,j,k}}$.

In the JCC-H benchmark, 29 of 61 attributes yield a block size larger than one, i. e., cannot grant 100% precision within a memory budget of 1% of the column size. Figure 9 shows the precision $\varphi_{i,j,k}$ of three approaches and six representative attributes with a block size larger than one. Overestimation is shown on the top, underestimation at the bottom. The boxplot displays the 0.0001, 0.25, 0.5, 0.75, and 0.9999 percentiles. Outliers are highlighted as dots above or below the boxplot.



(a) Naïve block-level counters (b) Our approach (− stream-sum.) (c) Our approach (+ stream-sum.)

Fig. 9: Precision of our approach (with and without the stream-summary data structure) compared to naïve block-level access counters for a table partitioning advisor, executing the JCC-H benchmark.

Figure 9(a) shows the precision of naïve block-level counters, i. e., the block counter is incremented by one whenever a value or sub-range of the block is read. The results confirm the statement in Section 5.4 that access frequencies are overestimated substantially.

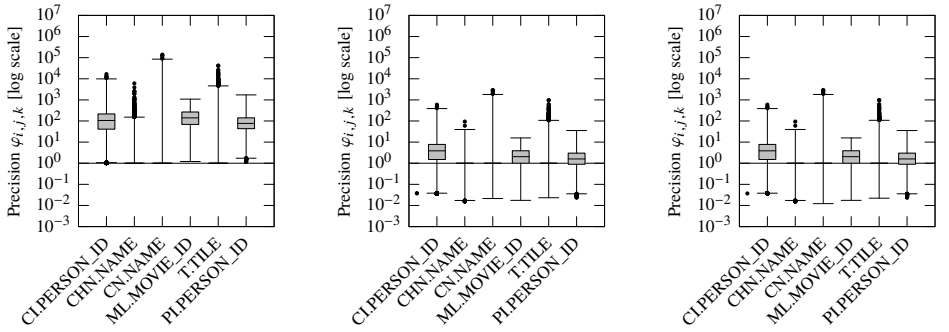
Figure 9(b) shows the precision of our approach that counts the number of actually accessed block values, while the access frequency of a value is obtained by dividing the total number of accessed values by the block size. We observe that our approach dramatically improves precision by several orders of magnitude, most of the estimates are within a bound of factor 2. However, for all six attributes, heavy hitters are underestimated, and rarely accessed values are overestimated (shown on the top and bottom of Figure 9(b)).

Figure 9(c) shows the precision obtained by adding a stream-summary to identify heavy hitters. To emphasize the difference with and without the stream-summary, we mark these values in Figure 9(b) in red, which are estimated correctly in Figure 9(c). For example, the heavy hitters of L_ORDERKEY (shown in red at the bottom in Figure 9(b)) are estimated precisely in Figure 9(c). Accordingly, rarely accessed values of the corresponding block are overestimated without the stream-summary (shown at the top of Figure 9(b)) but

estimated precisely with the stream-summary. We observe similar results for O_CUSTKEY, L_PARTKEY, and L_EXTENDEDPRICE.

We omit measurements of the precision for the TPC-H benchmark since the results are very similar compared to the JCC-H benchmark by ignoring the heavy hitters.

In the Join Order Benchmark, 47 of 108 attributes yield a block size larger than one. Figure 10 shows the precision $\varphi_{i,j,k}$ for six representative attributes. We again observe that naïve block-level counters overestimate access frequencies substantially (Figure 10(a)), while our approach improves the precision by 1-2 orders of magnitude (Figure 10(b)). However, we do not observe substantial improvement by adding a stream-summary like for the JCC-H benchmark (Figure 10(c)). The reason is that the JCC-H benchmark exhibits heavy hitters by design, while the Join Order Benchmark exposes only limited data and query skew.



(a) Naïve block-level counters (b) Our approach (− stream-sum.) (c) Our approach (+ stream-sum.)

Fig. 10: Precision of our approach (with and without the stream-summary data structure) compared to naïve block-level access counters for a table partitioning advisor, executing the Join Order Benchmark.

Table 7 shows the space efficiency and runtime overhead of row-level access counters, naïve block-level access counters, and our approach, with and without the stream-summary data structure. While row-level data access counters are 100% precise, their memory overhead is high, and the runtime overhead is also notable. In contrast, naïve block-level access counters and our approach (without stream-summary) use a fixed memory budget of 1% and achieve

	Row-Level Counters			Block-Level Counters & Our approach (− s.s.)			Our approach (+ stream summary)		
Workload	TPC-H	JCC-H	JOB	TPC-H	JCC-H	JOB	TPC-H	JCC-H	JOB
Memory Overhead	10.80%	10.82%	20.53%	≤ 1%	≤ 1%	≤ 1%	≤ 1%	≤ 1%	≤ 1%
Runtime Overhead	3.9%	14.7%	15.6%	2.1%	9.7%	9.6%	13.8%	22.7%	23.6%

Tab. 7: Memory and runtime overhead for our approach compared to row and block-level counters.

low runtime overhead. However, naïve block-level access counters are imprecise, while our approach achieves precise estimates (Figures 9 and 10). Adding the stream-summary data structure further improves the precision (Figure 9) at the cost of increasing the runtime overhead. Therefore, we argue that our approach (without the stream-summary) is preferred if the runtime overhead is critical. Otherwise, the stream-summary data structure may be added to improve the precision with low memory overhead.

7 Conclusion

We presented data structures that solve the problem of providing workload execution statistics with high precision, low memory consumption, and low runtime overhead to automated physical database design tools. Since no approach in the literature collects precise, compact, and fast workload execution statistics for an advisor tool, we presented how existing approaches can be combined and for which advisors new data structures have to be designed. Our evaluation showed that our data access counters outperform related work to provide precise, compact, and fast workload execution statistics for an index advisor, a data compression advisor, a buffer pool size advisor, and a table partitioning advisor using real-world and synthetic benchmarks.

References

[ABI19] Athanassoulis, M.; Bøgh, K. S.; Idreos, S.: Optimal Column Layout for Hybrid Workloads. Proc. VLDB Endow. 12/13, pp. 2393–2407, Sept. 2019.

[Ag04] Agrawal, S.; Chaudhuri, S.; Kollar, L.; Marathe, A.; Narasayya, V.; Syamala, M.: Database Tuning Advisor for Microsoft SQL Server 2005. In: Proceedings of the Thirtieth International Conference on Very Large Data Bases. VLDB '04, VLDB Endow., pp. 1110–1121, 2004.

[BAK18] Boncz, P.; Anatiotis, A.-C.; Kläbe, S.: JCC-H: Adding Join Crossing Correlations with Skew to TPC-H. In (Nambiar, R.; Poess, M., eds.): Performance Evaluation and Benchmarking for the Analytics Era. Springer International Publishing, Cham, Switzerland, pp. 103–119, 2018.

[Cu10] Curino, C.; Jones, E.; Zhang, Y.; Madden, S.: Schism: A Workload-Driven Approach to Database Replication and Partitioning. Proc. VLDB Endow. 3/1–2, pp. 48–57, Sept. 2010.

[Da16] Das, S.; Li, F.; Narasayya, V. R.; König, A. C.: Automated Demand-Driven Resource Scaling in Relational Database-as-a-Service. In: Proceedings of the 2016 International Conference on Management of Data. SIGMOD '16, Association for Computing Machinery, New York, NY, USA, pp. 1923–1934, 2016.

[Da19] Damme, P.; Ungethüm, A.; Hildebrandt, J.; Habich, D.; Lehner, W.: From a Comprehensive Experimental Survey to a Cost-Based Selection Strategy for Lightweight Integer Compression Algorithms. ACM Trans. Database Syst. 44/3, June 2019.

[FKN12] Funke, F.; Kemper, A.; Neumann, T.: Compacting Transactional Data in Hybrid OLTP & OLAP Databases. Proc. VLDB Endow. 5/11, pp. 1424–1435, July 2012.

[Gu18] Gurajada, A.; Gala, D.; Zhou, F.; Pathak, A.; Ma, Z.-F.: BTrim: Hybrid in-Memory Database Architecture for Extreme Transaction Processing in VLDBs. Proc. VLDB Endow. 11/12, pp. 1889–1901, Aug. 2018.

[Hu19] Huang, G.; Cheng, X.; Wang, J.; Wang, Y.; He, D.; Zhang, T.; Li, F.; Wang, S.; Cao, W.; Li, Q.: X-Engine: An Optimized Storage Engine for Large-Scale E-Commerce Transaction Processing. In: Proceedings of the 2019 International Conference on Management of Data. SIGMOD '19, Association for Computing Machinery, New York, NY, USA, pp. 651–665, 2019.

[KAI17] Kester, M. S.; Athanassoulis, M.; Idreos, S.: Access Path Selection in Main-Memory Optimized Data Systems: Should I Scan or Should I Probe? In: Proceedings of the 2017 ACM International Conference on Management of Data. SIGMOD '17, Association for Computing Machinery, New York, NY, USA, pp. 715–730, 2017.

[Ko20] Kossmann, J.; Halfpap, S.; Jankrift, M.; Schlosser, R.: Magic Mirror in My Hand, Which is the Best in the Land? An Experimental Evaluation of Index Selection Algorithms. Proc. VLDB Endow. 13/12, pp. 2382–2395, July 2020.

[Le10] Lemke, C.; Sattler, K.-U.; Faerber, F.; Zeier, A.: Speeding Up Queries in Column Stores. In (Bach Pedersen, T.; Mohania, M. K.; Tjoa, A. M., eds.): Data Warehousing and Knowledge Discovery. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 117–129, 2010.

[Le15] Leis, V.; Gubichev, A.; Mirchev, A.; Boncz, P.; Kemper, A.; Neumann, T.: How Good Are Query Optimizers, Really? Proc. VLDB Endow. 9/3, pp. 204–215, Nov. 2015.

[LLS13] Levandoski, J. J.; Larson, P.-Å.; Stoica, R.: Identifying hot and cold data in main-memory databases. In: 2013 IEEE 29th International Conference on Data Engineering (ICDE). ICDE '13, IEEE, pp. 26–37, 2013.

[Lu19] Lu, J.; Chen, Y.; Herodotou, H.; Babu, S.: Speedup Your Analytics: Automatic Parameter Tuning for Databases and Big Data Systems. Proc. VLDB Endow. 12/12, pp. 1970–1973, Aug. 2019.

[MAE05] Metwally, A.; Agrawal, D.; El Abbadi, A.: Efficient Computation of Frequent and Top-k Elements in Data Streams. In (Eiter, T.; Libkin, L., eds.): Database Theory - ICDT 2005. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 398–412, 2005.

[MBL17] May, N.; Böhm, A.; Lehner, W.: SAP HANA – The Evolution of an In-Memory DBMS from Pure OLAP Processing Towards Mixed Workloads. In (Mitschang, B.; Nicklas, D.; Leymann, F.; Schöning, H.; Herschel, M.; Teubner, J.; Härder, T.; Kopp, O.; Wieland, M., eds.): Datenbanksysteme für Business, Technologie und Web (BTW 2017). Gesellschaft für Informatik, Bonn, pp. 545–546, 2017.

[Na20] Nathan, V.; Ding, J.; Alizadeh, M.; Kraska, T.: 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, pp. 985–1000, 2020.

[No20] Noll, S.; Teubner, J.; May, N.; Böhm, A.: Analyzing Memory Accesses with Modern Processors. In: Proceedings of the 16th International Workshop on Data Management on New Hardware. DaMoN '20, Association for Computing Machinery, New York, NY, USA, 2020.

[Ra02] Rao, J.; Zhang, C.; Megiddo, N.; Lohman, G.: Automating Physical Database Design in a Parallel Database. In: Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data. SIGMOD '02, Association for Computing Machinery, New York, NY, USA, pp. 558–569, 2002.

[Se16] Serafini, M.; Taft, R.; Elmore, A. J.; Pavlo, A.; Aboulnaga, A.; Stonebraker, M.: Clay: Fine-Grained Adaptive Partitioning for General Database Schemas. Proc. VLDB Endow. 10/4, pp. 445–456, Nov. 2016.

[Sh19] Sherkat, R.; Florendo, C.; Andrei, M.; Blanco, R.; Dragusanu, A.; Pathak, A.; Khadilkar, P.; Kulkarni, N.; Lemke, C.; Seifert, S.; Iyer, S.; Gottapu, S.; Schulze, R.; Gottipati, C.; Basak, N.; Wang, Y.; Kandiyanallur, V.; Pendap, S.; Gala, D.; Almeida, R.; Ghosh, P.: Native Store Extension for SAP HANA. Proc. VLDB Endow. 12/12, pp. 2047–2058, Aug. 2019.

[St06] Storm, A. J.; Garcia-Arellano, C.; Lightstone, S. S.; Diao, Y.; Surendra, M.: Adaptive Self-Tuning Memory in DB2. In: Proceedings of the 32nd International Conference on Very Large Data Bases. VLDB '06, VLDB Endowment, pp. 1081–1092, 2006.

[TP18] TPC: TPC Benchmark H Standard Specification, `http://www.tpc.org/tpc_documents_current_versions/pdf/tpc-h_v2.18.0.pdf`, 2018.