RESEARCH-ARTICLE

Experimental Evaluation of Optimizing Memory Consumption in SAP HANA using PEOopt

LUKAS LANDGRAF, TUD Dresden University of Technology, Dresden, Sachsen, Germany

FLORIAN WOLF, SAP SE, Walldorf, Baden-Wurttemberg, Germany

WOLFGANG LEHNER, TUD Dresden University of Technology, Dresden, Sachsen, Germany

Open Access Support provided by:

TUD Dresden University of Technology

SAP SE

Published: 22 June 2025

Citation in BibTeX format

SIGMOD/PODS '25: International Conference on Management of Data
June 22 - 27, 2025
Berlin, Germany

Conference Sponsors:
SIGMOD

https://doi.org/10.1145/3722212.3724437
ISBN: 9798400715648

Experimental Evaluation of Optimizing Memory Consumption in SAP HANA Using PEOopt

Lukas Landgraf
TU Dresden, SAP SE
Walldorf, Germany
lukas.landgraf@tu-dresden.de

Florian Wolf
SAP SE
Walldorf, Germany
florian.wolf01@sap.com

Wolfgang Lehner
TU Dresden
Dresden, Germany
wolfgang.lehner@tu-dresden.de

Abstract

Materialized intermediate results dominate memory consumption in query processing, which is a challenge in the age of Big Data workloads. Modern query processors use pipelines to produce materialized intermediate results. The order by which pipelines are executed influence the lifetime of materialized intermediate results, directly impacting the overall consumed memory. By scheduling the execution of pipelines, we can therefore save memory.

Our previous work provided a proof-of-concept for pipeline execution ordering to reduce memory consumption during runtime using a simple join query workload. This paper presents the experimental evaluation of an end-to-end implementation prototype called *PEOopt* within SAP HANA, which generalizes our previous work. For long running queries our prototype reduces the memory footprint by up to 36% without compromising any execution time performance. Hence, we prove the effectiveness of saving memory in real world database systems like SAP HANA.

CCS Concepts

• **Information systems → Query optimization**; **Main memory engines**.

Keywords

query processing, query optimization, pipeline, memory optimization, main memory database system

ACM Reference Format:
Lukas Landgraf, Florian Wolf, and Wolfgang Lehner. 2025. Experimental Evaluation of Optimizing Memory Consumption in SAP HANA Using PEOopt. In *Companion of the 2025 International Conference on Management of Data (SIGMOD-Companion '25), June 22–27, 2025, Berlin, Germany*. ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3722212.3724437

1 Introduction

Modern query processors fuse operators to continuous pipelines. Those pipelines arise from Query Execution Plans (QEPs), which are trees of relational physical operators. Those operators have a specific materialization behavior for their inputs or outputs. For instance, the build side of a hash join triggers the materialization of the incoming build side tuples in a hash table. Such materializations mark the end of the pipeline, where so-called pipeline breakers contain the intermediates. Those pipeline breakers dominate the memory consumption of a query. In turn, the lifespan of a pipeline breaker and how all pipeline breakers overlap over time influences memory peak consumption and the average memory consumption. Those pipeline breakers may not be used by a currently executed pipeline. Several opportunities for reducing memory consumption by handling unused pipeline breakers exist, e.g., by spilling them to far memory using CXL or RDMA [5, 26]. As we found out, we can additionally influence the lifetime of pipeline breakers by changing the Pipeline Execution Order (PEO), which is the order in which pipelines are executed one after another. The impact of a PEO on memory consumption poses an optimization opportunity. The following conditions enable this optimization opportunity without impacting execution time:

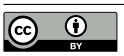
- A high level of intra-pipeline parallelization enables the execution of one pipeline at a time [23]. Executing one pipeline at a time then creates a choice of pipelines that can be executed next.
- As pipelines can depend on other pipelines, the order of execution is constrained by dependencies.
- The execution time of a pipeline does not change regardless of any previously executed pipelines. Caching effects are negligible for sufficiently large intermediate results. Therefore, reordering pipelines does not impact execution time.

With this work we contribute an experimental evaluation of the PEO enumeration approach while enabling the approach for all queries. PEO enumeration is the process of identifying good PEOs that yield the highest possible memory consumption savings. Memory optimization using PEO enumeration has been conceptualized in our previous work [21]. In contrast, this work provides a comprehensive evaluation for the potential of our approach which we implemented in SAP HANA [20, 27, 28, 32], an industry-grade Database Management System (DBMS).

Contributions

We summarize the contributions of this paper as follows:

- Using our prototype of SAP HANA with our new component called *PEOopt* (Pipeline Execution Order Optimization), we provide a first systematic exploration of memory savings enabled by *PEOopt*.
- We use real cardinalities to show that our extended memory cost model can estimate the actual memory consumption of the query processor.
- We demonstrate the usability of *PEOopt* in productive scenarios by showing that estimated cardinalities from SAP HANA's cardinality estimator together with our memory cost model can estimate the actual memory consumption of the query processing.



SIGMOD-Companion '25, Berlin, Germany

ACM ISBN 979-8-4007-1564-8/2025/06
https://doi.org/10.1145/3722212.3724437

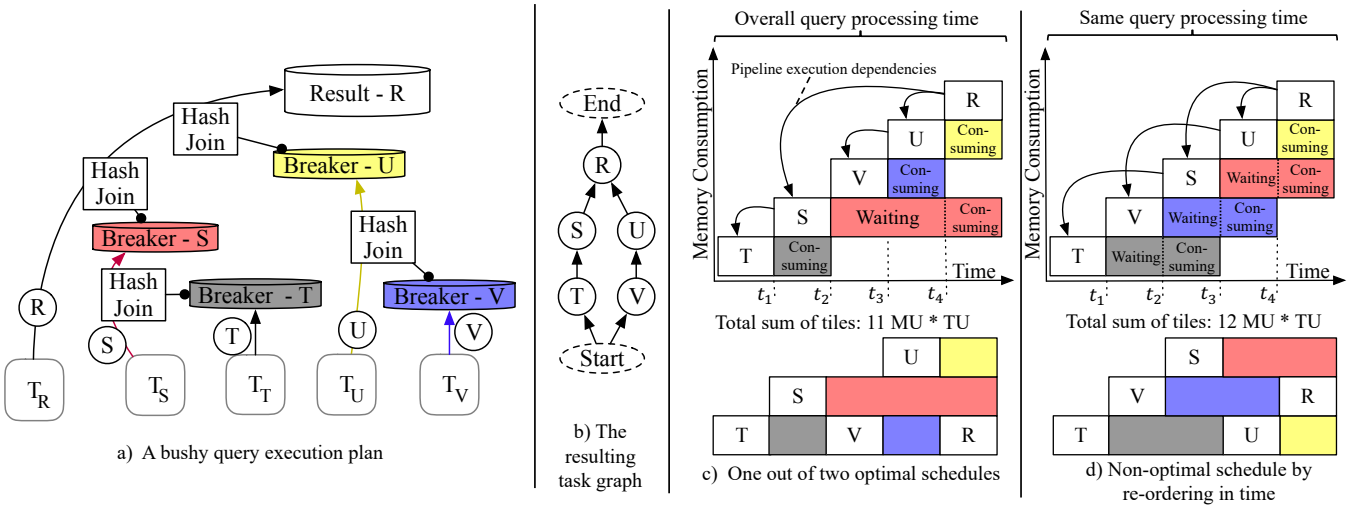


Figure 1: Query execution plan and two pipeline execution orders.

- We show that *PEOopt* does not create a significant overhead and does not alter the execution time of the given QEP. We further show that the savings do not correlate to the complexity of the given QEP.

The paper is structured as follows. In Section 2, we explore how to estimate execution time of individual pipelines and how to estimate the memory size of their pipeline breakers. Through this, we provide the foundation for the pipeline execution model and systematically estimate the memory consumption for individual PEOs. In Section 3, we describe how we model dependencies and the allocation behavior of pipeline breakers. We state algorithms that systematically explore the PEO space and estimate the PEO-induced memory consumption, extending our existing algorithms [21]. In Section 4, we describe the implementation of *PEOopt*, our system to reduce memory consumption using PEO enumeration, within a prototype of SAP HANA. Further, we explain how the approach in [21] can be integrated into DBMS in general and highlight some challenges of integration into SAP HANA. We evaluate our approach in Section 5 and we provide an overview over related work in Section 6. Finally, we conclude with Section 7.

2 Foundations

Optimizing for memory consumption requires the estimation of possible memory consumers during query execution. Hence, we first identify relevant memory consumers that are reflected in the pipeline execution model and describe the properties of the pipeline execution model. We then derive the task model from the execution model, where each task is assigned the execution cost of a pipeline and the memory cost of the pipeline's pipeline breaker. We then introduce the cost model to estimate execution costs and memory costs of pipelines in the task model. We then use this task model to enumerate Pipeline Execution Orders (PEOs) and estimate their memory consumption.

2.1 Execution Model

Our optimization starts with a QEP, which is the result of optimizing a query plan through the query optimizer. The query executor then normally uses the QEP for subsequent execution. A QEP consists of operators. Certain operators break pipelines to calculate an intermediate result or to form a data structure for further processing. Typical examples of pipeline breakers are sorting or aggregation operators or the build side of a hash join, as can be seen in Figure 1a, where all breakers contain hash tables for hash joins. Consequently, a QEP can be seen as a set of pipelines with dependencies between pipelines [18]. A QEP embeds physical operators and pipeline breakers in different formats, like hash tables. The execution of a single pipeline can be parallelized using an almost arbitrarily high number of threads. Each thread can independently pick up chunks of tuples ("morsel" [23], e.g., 100,000 tuples) and push them through the pipeline. The thread executes all subsequent operators that are contained in a pipeline for each tuple in the chunk. The pipeline materializes tuples in the pipeline breaker. Executing only one pipeline at a time can result in complete utilization of all threads with potentially high cache locality. This is called morsel-based parallelism [23]. Instead of executing different pipelines in parallel, the query engine can then execute one pipeline after another without sacrificing query execution time. A side effect of morsel-based parallelism is that it creates freedom of choice for the next executable pipeline. This freedom of choice then enables different Pipeline Execution Orders (PEOs) with potentially different memory footprints, thus providing optimization potential. Hence, we can enumerate the optimal PEO. We assume that the execution time of different PEOs is nearly identical as caching effects of pipeline breakers are negligible for large analytical datasets. Caching effects between pipelines can only occur if a previously materialized pipeline breaker is directly consumed by the next pipeline. Yet, the pipeline breaker itself might not fit into the last level cache itself. Hence, if the materialized data is sufficiently large and thus worth scheduling, it is safe to assume that caching effects are negligible.

2.2 Task Model

Executing a pipeline can be considered a task, since the execution of a pipeline has a start and end time, consumes resources, has dependencies, and is schedulable [10, 11]. In our case, start and end times are defined by the pipeline execution times and the consumed resource is memory. We then build a task graph from those pipelines, which is called an Activity-on-Node Graph (AONG) [11, 21]. A task graph for a pipelined QEP is characterized by its tasks and its dependencies, called precedence constraints [12]. Tasks are the set of all pipelines/tasks as nodes in an AONG [11, 21]. They have execution costs and memory costs. Precedence constraints arise through two conditions: the constraint of executing only one pipeline at a time and the possible dependency of pipelines on other pipeline breakers. The precedence constraints within an AONG constrain the execution order of pipelines, where a constraint $p \prec q$ denotes that task p must be executed before task q. As we see in Figure 1a, pipeline "R" executes two hash joins and requires that the hash tables in "Breaker - S" and "Breaker - U" are being produced by pipelines "S" and "U". The latter pipelines also depend on "T" and "V", respectively. From this, we can derive $S \prec R, U \prec R, V \prec U$, and $T \prec S$, as shown in Figure 1b. On top of those constraints, AONGs have two traversal rules. (1) During processing, a task can only be processed once all its predecessors have been processed. (2) Each task continues to consume memory and therefore stays *active* until all successors have been processed. This behavior is a direct result of the materialization behavior of pipeline breakers, which we will demonstrate in the following.

A PEO is a sequence of pipelines that fulfills all constraints and traversal rules of an AONG. Figures 1c/d illustrate the memory consumption difference between two PEOs. They show the materialization behavior of pipeline breakers for two out of six possible PEOs. Figure 1c visualizes the allocation behavior for the PEO (TSVUR) with a Gantt-chart. The rectangles in a Gantt-chart indicate two kinds of costs: the memory cost of each breaker (height of the blocks, y-axis) and the lifetime for each breaker (on the x-axis). As the first pipeline, T is executed until point-in-time t_1. The engine allocates the pipeline breaker of T fully from the start. The pipeline breaker of T exists from time 0 until t_2. From t_1 until t_2, "Breaker - T" is consumed by pipeline S, after which it is deallocated. S is executed from t_1 until t_2 and its pipeline breaker waits from t_2 until t_4, after which it is consumed by R. In a similar manner, the rest of the pipeline and pipeline breakers are processed. Each pipeline breaker transitions between three phases: *"constructing"*, e.g., a hash table in a pipeline breaker, which directly corresponds to the execution time of the respective pipeline, *"consuming"* the breaker by another pipeline, and *"waiting"* of the breaker for consumption. The bottom parts of the figures show the memory integral (MI) (memory consumed over time, illustrated by the consumed area) for each PEO, which is different for both PEOs (11 memory units[MU] * time units[TU], vs. 12 MU*TU). The difference in MI arises from the PEO in Figure 1c having one waiting phase less than the PEO in Figure 1d, despite "Breaker - S" having one more waiting phase when compared to the PEO in Figure 1d. We can also see that the allocated memory changes over time, which illustrates that the memory peak (MP) can differ between PEOs, as the MP is the measure of highest memory consumption during execution.

To calculate the memory consumption we need to estimate the memory size of breakers and the execution cost of pipelines to find this trade-off. To do so, we introduce the cost model to calculate the memory consumption estimate of each PEO.

3 PEOopt Implementation

PEOopt facilitates a cost-based optimization of memory consumption using Pipeline Execution Order (PEO) enumeration. This optimization needs a cost model [21], which we describe in the following. Afterwards, we will modify the algorithms of [21] to accommodate the changes in the cost model and to handle shared sub-plans, which is a necessity for the implementation of PEOopt in HANA as described in Section 4.

3.1 Cost Model

In this subsection, we first define the behavior of operators that cause pipeline breakers and then the cost model to calculate execution/memory costs. Typically, pipelines are started by leaf operators (e.g., table scans), which push tuples [23] and end in a pipeline breaker. Also, they can be started by pipeline breakers of other pipelines which act as leaf nodes. Each of the operators starting from the leaf node contribute to the execution cost or might introduce a pipeline breaker. Operators are categorized by their breaking behavior as follows:

- Source operators push tuples (like table scans), no breaker.
- Operators like unions or index joins do not cause breakers. Index joins are not breakers (except for left semi joins), since they rely on a static resource (the index) and can push tuples without needing to materialize.
- Build/probe-sided operators have one breaker, such as hash joins or nested loop joins.
- Left semi joins have breakers on both sides.
- Left semi index joins and aggregations cause a breaker.
- Multi-successor operators like shared sub-plans cause task nodes with multiple successors (multi-successor nodes). A shared sub-plans is a part of a QEP that is a child of multiple operators, and enable optimizations like semi join reduction [37] or common sub-queries [16].

Since operator implementations vary, the implementation always needs to model the implemented breaking behavior of the specific engine. Based on the operators inside a pipeline, we can consequently model the pipeline execution cost.

We calculate the execution cost based on the c_{out} cost function [21, 34]. c_{out} assigns costs to any (sub-)tree by adding up the output cardinalities of all operators in a QEP. By subtracting those parts of the tree that are not part of the pipeline p_i, we can obtain its execution cost e_i. We calculate c_{out} for the operator below a pipeline's breaker as c_i. Let p_j denote all pipelines that are executed before p_i. We subtract the execution cost c_j of those pipelines p_j executed before p_i, since they do not contribute to the execution costs of their successors. Further, we consider σ_i as writing costs to the pipeline breaker. This yields the pipeline's own execution cost e_i.

$$e_i = c_i - \sum_{p_i \succ p_j} c_j + \sigma_i \tag{1}$$

For instance, in Figure 1a we see different sub-trees, which perform joins of tables or relations produced by other joins. To address sub-trees, we introduce a terminology: in Figure 1a UV denotes the sub-tree, which joins U and V. RSTUV then denotes the entire tree. Using this terminology, we can describe the cost of the pipeline in our example, which runs to the root of the tree: $e_{RSTUV} = c_{RSTUV} - c_{UV} - c_{ST}$ if we assume that the writing cost $\sigma_1 = 0$. A matrix-based calculation method of all pipeline execution costs is given in [21]. The memory cost m_i of each task node is calculated by multiplying the row size w_i and the cardinality f_i at each breaker, such that $m_i = w_i \cdot f_i$ [21].

We then calculate the lifetime l_i for each pipeline breaker of pipeline p_i. This is the sum of constructing, consuming, and waiting times for each breaker. The lifetime depends on the PEO and the precedence relationships between pipelines. Aforementioned construction phases always reflect the execution cost of the respective pipeline. Consuming and waiting phases always reflect execution costs of other pipelines. Referring back to Figure 1c, the first waiting phase for the pipeline breaker of S is equal to the construction phase of pipeline V. The second waiting phase is equal to the construction phase of pipeline U, and the final consuming phase is the construction phase of pipeline R. In general, the lifetime is then a multiplication of the vector $\vec{e}$ of execution times and a binary vector h_i denoting the existence of the pipeline breaker of p_i for a specific PEO O during the execution of the respective pipelines. For "Breaker - T" in Figure 1c with the PEO (TSVUR) as the basis, the binary vector is $h_S = (0, 1, 1, 1, 1)$, as "Breaker - S" is active during execution of S, V, U, and R, and $\vec{e} = (1TU, 1TU, 1TU, 1TU, 1TU)$ such that $l_S = \vec{e} \cdot h_S^T = 4TU$, which is the lifetime of "Breaker - S". We define the MI [21] and MP as:

$$MI = \sum_{i=1}^{k} m_i l_i \quad (2) \qquad MP = max_{0 \le t \le l_{all}} \left(\sum_{p_i \text{is active}} m_i \right) \quad (3)$$

where l_{all} is the total query duration and t is any point in time during query execution. Equation 3 expresses the sum of all pipeline breakers p_i that consume memory during t, which is a point in time during query execution. For Figure 1b, the peak can then be found between t_3 and end of query execution, where either the breakers of U,V,S are active or R,U,S are active. We evaluate the MP additionally to the MI in Section 5, as other literature also discussed memory-bound scenarios [10].

This model assumes that all the memory of a breaker is allocated immediately after the start of the pipeline. This is adequate for hash tables and group-by structures as pipeline breakers, because these data structures write with random access into memory locations. For the sake of simplicity, we assume this behavior for all pipeline breakers, which is reflected in Equation 2.

The goal of the optimization approach is to save memory in terms of MI or MP without sacrificing execution time. While reordered pipelines do not cause overhead, enumerating PEOs may be complex. Next, we introduce the search algorithms for finding PEOs and assigning their memory consumption.

3.2 Search Algorithms

We proposed four algorithms to find optimal or near-optimal PEOs with regards to MI in [21]. Our first algorithm is Exhaustive Search (ES) stated in Algorithm 1, where we added the contributions of this paper in black to accommodate task nodes with multiple successors. The algorithm is a depth-first recursive algorithm, which passes the first explorable nodes in the first call. The complexity of this modified exhaustive algorithm has been described in another previous work [22]. We briefly describe this algorithm here. Looking at Figure 1, we pass V and T as the `firstPossibleNodes`. Each of the nodes `n` has `predecessorNodes` (nodes it depends on) and `successorNodes` (nodes that depend on `n`). The algorithm tracks what nodes can be explored next (line 4), which nodes are blocking resources (line 6), and if any pipeline breakers are ready to be deallocated (lines 11 to 14). We modified the deallocation step in ES to consider if all depending nodes have already been processed. Once all dependencies of a node have been calculated and have been stored, the node is added to the set of explorable nodes (line

Algorithm 1 Call with ES(firstPossibleNodes, [start], 0, 0). Returns all possible PEOs. Lines in gray from [21], lines in black new.

```
1:  function ES(activeNodes, visitedNodes, memPeak, MI)
2:      peos = []
3:      // insert "Theta Skip" step here if needed
4:      for n in activeNodes do
5:          newNodes = activeNodes.copy().remove(n)
6:          nextPeak = memPeak + n.memorySize
7:          nextMI = MI + nextPeak * n.lifetime
8:          for p in n.predecessorNodes do
9:              //n no longer depends on p
10:             p.nextCount = p.nextCount - 1
11:             if p.nextCount == 0 then
12:                 //p is no longer needed
13:                 nextPeak -= p.memorySize
14:             end if
15:         end for
16:         if n.isEnd() then
17:             return [PEO(visitedNodes + [n], nextMI, nextPeak)]
18:         end if
19:         // insert "Branch Pruning" step here if needed
20:         for next in n.successorNodes do
21:             next.prevCount =- 1
22:             if next.prevCount == 0 then
23:                 newNodes.append(next)
24:             end if
25:         end for
26:         newVisitedNodes = visitedNodes + [n]
27:         peos += ES(newNodes, newVisitedNodes, nextPeak,
28:             nextMI)
29:         for next in n.successorNodes do
30:             node.next.prevCount =+ 1
31:         end for
32:         for prev in n.predecessorNodes do
33:             prev.nextCount = prev.nextCount + 1 //restore state
34:         end for
35:     end for
36:     return peos
37: end function
```

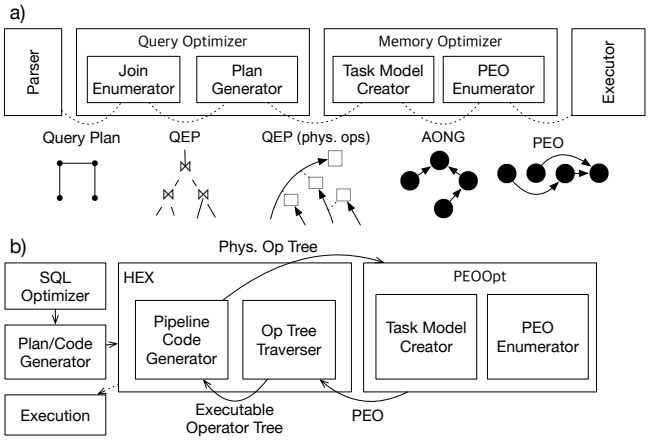


Figure 2: a) Flow of a query through a database engine with a memory optimizer, b) flow of a query in our commercial database system.

22). After this, the algorithm further explores the tree through recursion, carrying the current memory consumption, active nodes, and available nodes for processing (line 27). After that, we need to restore the state of still needed visits to continue visiting other available nodes. To call the algorithm, we pass the first possible nodes. To speedup enumeration, we introduced three heuristics in [21]. We also extended those heuristics to track the lifetime of nodes with multiple successors. ES serves as the basis for Branch Pruning Search (BPS) and Theta Skip Search (TSS). BPS uses an upper bound MI estimate to prune PEO alternatives during recursion if they exceed an upper bound MI, given through a heuristic. TSS introduces a step of pushing tasks to PEOs, where the given task produces pipeline breakers of minimal size. As such, this may lose potentially optimal enumeration variants, but it reduces the complexity of the algorithm while producing a near-optimal PEO. Longest Path Heuristic (LPH) is a simple heuristic that subsequently picks the longest paths from the start node to the end node of an AONG and inserts those paths into the PEO. In Figure 1b, this would result in inserting either VU or TS into the PEO first and then inserting the next sub-PEO that is remaining. LPH as stated in [21] cannot process arbitrary multi-successor nodes. We chose to only support multi-successor nodes with a fan-out of two that result from semi join reduction, since we do not execute common sub-queries. As a result, we do not have to modify the basic principle of the LPH stated in [21]. Aforementioned concepts serve as the basis for the implementation, which we will discuss next.

4 SAP HANA Prototype with PEOopt

In this section, we present the integration of the approach into SAP HANA. First, we give an overview over query processing and integration of the memory optimization approach. Next, we describe the implementation of *PEOopt* into our existing system and how memory consumption is monitored.

Integration. Figure 2 illustrates the flow of an OLAP query through a DBMS, resulting in a physical plan. Physical operators contain information on pipelines and pipeline breakers. Usually, a pipeline

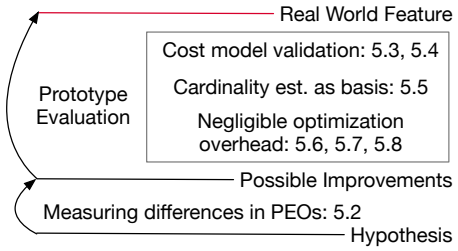


Figure 3: Flow of features towards practical deployment.

execution order is deterministic for a given QEP. For example, we could always execute the right-most not-yet-executed pipeline for "QEP (phys. ops)" in Figure 2a. Instead, the optimizer ("Memory Optimizer") use the QEP with physical operators for the memory optimization. With the operator cardinalities and pipeline information at hand, the optimizer constructs pipelines. The pipelines are then handed over to the "PEO Enumerator". The optimizer orders the pipelines to optimize regarding memory consumption. The "Executor" then executes the ordered pipelines and produces the result tuples. The implementation reflects this separation of responsibilities, as we will see in the next subsection.

Implementation in SAP HANA. We integrate *PEOopt* as the memory optimizer into the HEX engine of SAP HANA, which processes HTAP workloads in thousands of mission critical customer scenarios [32]. Figure 2b visualizes the resulting architecture. The SQL Optimizer component creates a physical plan that is converted to a pipelining operator representation by the "Plan/Code Generator". The query execution engine has to traverse this tree-based representation to create executable pipelines. Prior to *PEOopt* the "Op Tree Traverser" of HEX traversed the QEP in a right-deep fashion, creating pipelines in the process and implicitly setting the execution order. Before this traversal, we invoke the "Memory Optimizer" that implements *PEOopt*. We feed the traverser our final PEO, which produces executable pipelines. We inserted our approach in this specific step, changing the execution order while complying to implicit assumptions of the pre-existing implementation. Additionally, we had to conceptually account for shared sub-plans to facilitate semi join reduction [37].

To investigate the impact on memory consumption, it is relevant to know that there are other memory consumers which are related to scheduling and bookkeeping within our query engine, introducing further noise. Further, we exclude the final result buffer when estimating the MI/MP in equations 2 and 3, since HANA streams results to the client. With the implementation of our approach at hand, we now proceed to evaluate our cost model and implementation for several benchmarks.

5 Evaluation

Next, we present our experimental evaluation of *PEOopt* in SAP HANA. Section 5.1 describes our experimental setup. To show possible improvements of *PEOopt*, Section 5.2 presents an analysis of brute-force enumerated and executed Pipeline Execution Orders (PEOs), while measuring the actual memory consumption. From Section 5.3 on, we then evaluate the different building blocks of

PEOopt. Section 5.3 and 5.4 evaluate our MI and MP cost models. Section 5.5 analyzes the impact of the SAP HANA cardinality estimator on *PEOopt* and evaluates the end-to-end performance of *PEOopt* in SAP HANA. Sections 5.6 to 5.8 focus on the enumeration overhead of *PEOopt* and discusses the impact of changing the execution order of pipelines on query execution times. Figure 3 summarizes our evaluation steps in an illustration.

5.1 Setup

We run our experiments on a four-socket system with four Intel Xeon Gold 6448Y CPUs with a total of 128 physical cores and 1TB RAM, running a SUSE Linux 15.5 with a 5.14.21 kernel. We use one socket to run the experiments. We constrain queries to run in the HEX [33] query execution engine of SAP HANA, since we implemented PEOopt in HEX. The query optimizer picked the QEP using estimated cardinalities as is common practice, in contrast to real cardinalities used in the experiments in [21].

Regarding the memory cost, we investigate five measures: **estimated** MI based on **real** cardinalities (est. MI (real cards.)), **estimated** MI based on **estimated** cardinalities (est. MI (est. cards.)), **measured** MIs (msrd. MIs), **estimated** memory peak based on **real** cardinalities (est. MP (real cards.)), and **measured** MP (msrd. MP). We omit the estimated MP based on estimated cardinalities because the execution time does not play a role when compared to the MI. The est. MI (real cards.) is using our MI metric with the cardinalities of pipeline breakers observed after executing the queries. We will spend the largest part of our evaluation for investigating the est. MI (real cards.) due to its novelty when compared to the MP. To measure the actual memory consumption, i.e., msrd. MIs/MPs, we use a special memory allocator component that tracks all memory allocations within the execution of a query. Note that this is different to using our cost model with real cardinalities, i.e., est. MI (real cards.), where only pipeline breakers are modeled as memory consumers. Msrd. MIs are given in *megabyte seconds* [MB ∗ s], msrd. MPs in *megabyte* [MB].

For the benchmarks we chose to evaluate the Join Order Benchmark (JOB) [24], the JCC-H benchmark scale factor 100 (SF100) [9], TPC-H benchmark SF100 [31], as well as the Star Schema Benchmark SF100 (SSB) [29]. Additionally, we evaluate a set of queries from an undisclosed customer. The number of pipelines in the queries ranges between 70 and 96. Since the data and queries are confidential, we cannot disclose them. We provide more details on the executed QEPs in the discussion of the experiments.

For each PEO we execute 9 runs for obtaining measurement data. We execute one warm-up run for each PEO to ensure the availability of tables in resource pools and caches. Further, we execute the best and the worst PEO according to our enumeration based on est. MI (est. cards.). We additionally include all remaining PEOs for queries with less than/equal 12 PEOs. Up to 120 existing PEOs we execute 12 random PEOs and for more than 120 existing PEOs per QEP at least 10% randomly drawn PEOs. Msrd. MI and msrd. MPs for each PEO are calculated by taking the median of the 9 measurements.

5.2 Best Case Performance

Before evaluating the optimization capabilities of *PEOopt* using our cost model, we are going to identify the memory saving that *PEOopt* could achieve in the best case. To identify the best possible savings, we brute-force enumerate and execute PEOs for a given query, and compare the measured memory consumption of each PEO. We present the results in Figures 4 and 5, showing the improvement of the best PEO compared to the worst PEO, w.r.t. measured memory consumption. We show that significant differences up to 36% in terms of MI can be achieved for MIs above 2.0MB ∗ s.

Join Order Benchmark. The JOB contains 113 queries that are based on 33 query categories with different parameters. All of the queries contain at least five joins. Out of these queries, we are able to execute 111 queries. Queries 7c and 13d cannot be executed in the PEOopt prototype. Figure 4a illustrates the relative differences in MI for 10 queries with the largest relative differences. Each bar represents a query and is annotated with the highest absolute measured MI or MP as well as the number of possible PEOs. For nine queries we obtain a difference of at least 20% between best and worst PEO. Queries from 20a-c onward have at least 12 joins and tend to produce more possible PEOs. However, the number of PEOs is still below the expected exponential explosion [21]. This is due to the query optimizer selecting index joins wherever possible, which causes fewer pipeline breakers. However, the highest number of PEOs could be seen for Query 28b with 63,360 PEOs. The latter exhibits a maximum MI 0.2MB ∗ s, s.t. the absolute savings are minimal and the number of PEOs does not correlate with the savings potential.

In an ideal scenario where the cost model is able to find the best and the worst PEO, we see that up to 55% savings for the best over the worst case are possible in Query 25c. For those queries we see a relatively small measured MI in Figure 4a. Queries with high absolute MIs are 8c, 8d, and 9d, which see MIs as high as 49.6MB ∗ s. For those, we saw possible savings of 8%, 7%, and 5%, showing the potential of the approach.

For best case peak memory savings, we can see the difference in measured MP for 10 queries with the highest relative difference between best and worst in Figure 4b. Query 25c has the highest relative difference in msrd. MP with 27%. In this particular case, a query with high potential for MI savings also has high potential for MP savings. Other results show that there is now strong correlation between possible MI and MP savings.

TPC-H SF100. We evaluate 15 of 19 TPC-H queries, of which 10 are shown in Figures 4c and 4d. Apart from Query 1 all queries contain joins and again the optimizer made use of index-based physical operators whenever possible, which - in combination with possible chained dependencies between pipelines - reduces the number of possible PEOs. Query 11 shows high savings in both MI and MP with 46% and 48% respectively, but has a lower absolute MI when compared to Query 15. Query 15 exhibits a high MI and MP, with considerable savings of 37% for the MI and 15% for the MP.

Star Schema Benchmark SF100. For the SSB on SF100 we can report results for all queries. Figures 4e and 4f show the results of 10 queries. Queries 1.2 and 1.3 have only one PEO and are consequently not included since there is no room for improvements. Similar to the previous benchmarks, using indices and chained dependencies between pipelines leads to a decreased number of possible PEOs per QEP. Due to the size of the dataset at SF100 we

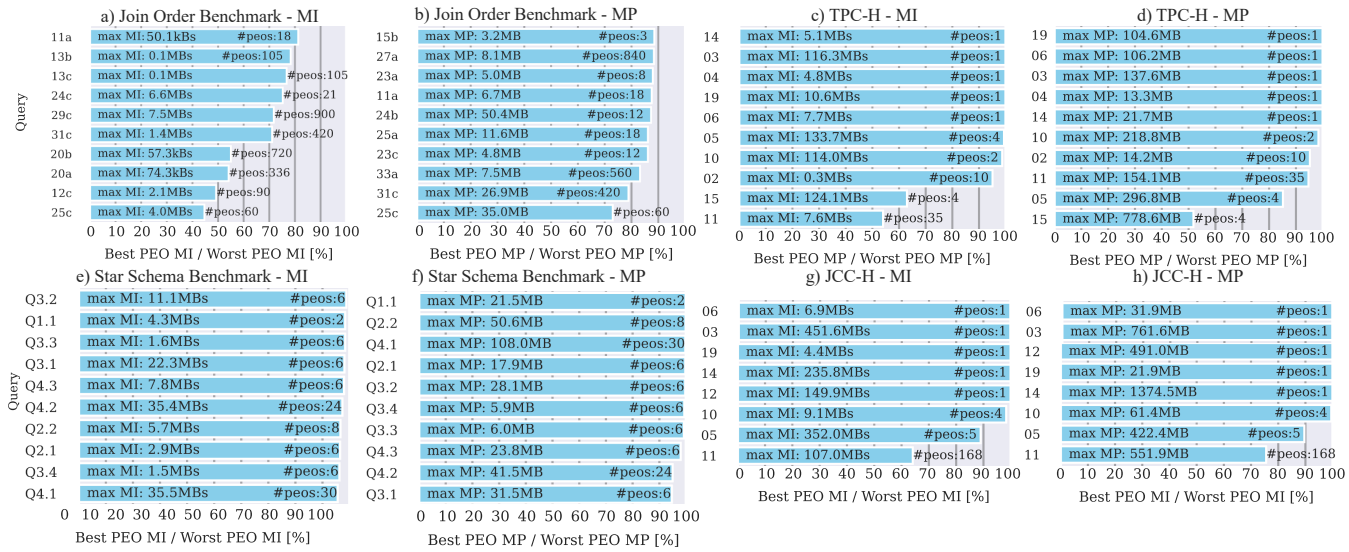


Figure 4: Achievable savings of MI and MP, comparing the best and the worst PEO.

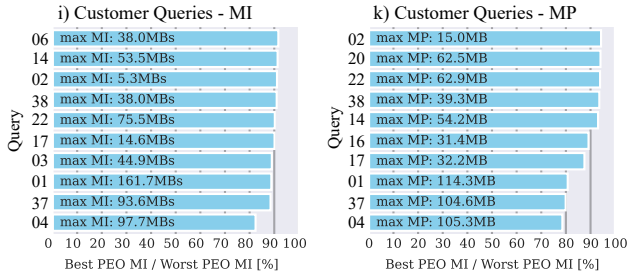


Figure 5: Achievable savings in MI and MP for customer queries.

are able to observe large MIs. Query 4.1 shows the highest absolute measured maximum MI of 35.5MB ∗ s, while having the highest relative difference of 5% between best and worst, while the highest PEO count is 30 possible PEOs. For the MP, this is the case for Query 3.1 at 6%. When compared to the previous two benchmarks, the possible savings are low, but considerable.

JCC-H SF100. The JCC-H benchmark contains 22 queries, of which we can evaluate 8 queries after execution. Out of those, queries 5, 10, 11 exhibit more than one PEO. Queries 5 and 11 induce high MIs at 352MB ∗ s and 107MB ∗ s. The same is the case for the MP at 422MB and 552MB. The resulting differences between best and worst PEO for each measure are shown in Figures 4g and 4k. For Query 11 we saw optimization potential of 36% for MI and 25% for MP, providing considerable optimization potential.

Customer Queries. In total, the customer query set contains 38 queries. These queries are generated, exhibit a high selectivity, but join up to 99 tables. Figures 5i and k show MI and MP differences for 12 executable and measurable queries. Due to high enumeration complexity, we are not able to execute the ES algorithm and chose to rely on randomly picked PEOs and the LPH. Query 1, for instance, has 48 hash joins, 22 semi joins through semi join reduction, and 5 index joins. The other queries provide similar complexity. We ran the ES algorithm on Query 1 and aborted at ~40,000,000 enumerations. The best/worst PEO is the best/worst encountered PEO among the random PEOs. The highest savings potential we encounter for Query 4, which provides potential for 18% savings in MI and 21% in MP. Four further queries show potential of more than 10% improvements.

Summary. In all benchmarks there is a clear difference between best and worst PEO for both measures for certain queries. Queries with high memory integrals show considerable relative savings. Those savings are only obtainable if a model is able to detect the best possible PEO. We look into this in the following subsection.

5.3 Quality of Cost Model for Memory-Constrained Scenarios

Next, we verify if our model is fit to estimate the MP. The MP is a measurement motivated by [10], which can be used to avoid memory consumption spikes and detect optimization potential for the MI. In contrast to [21] we provide the best PEO in terms of MP. The validation of the cost model for finding good PEOs in terms of MP is provided here. To better compare the impact of cost model effects and cardinality estimation errors, as well the difference between MI and MP, we show the results in the same figure. There is one figure for each benchmark. In the following graphs, the coloring of dots indicate the algorithms used, which are Exhaustive Search (ES), Longest Path Heuristic (LPH), Branch Pruning Search (BPS), and Theta Skip Search (TSS). These do not denote the PEOs found through enumeration for the best MP, but rather the best MI. Random PEOs are represented as grey dots. Ideally, we want so see a linear correlation between the x-value, which is est. MP (real cards.), and the y-value, which is the measured

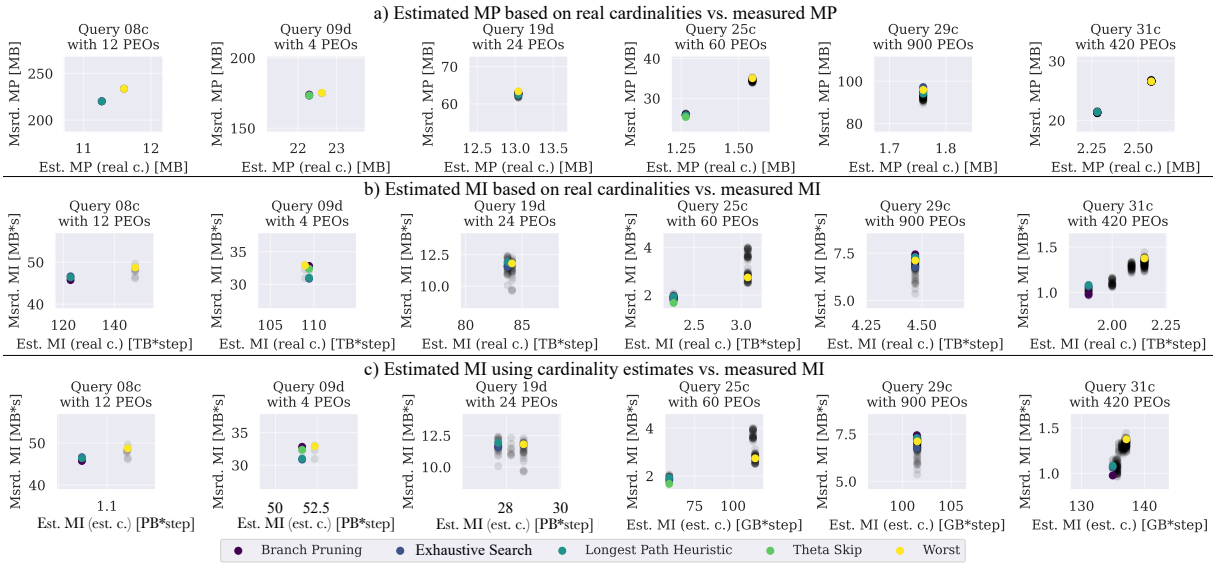


Figure 6: Correlation between a) est. MP (real cards.) and msrd. MP, b) est. MI (real cards.) and msrd. MI, and c) est. MI (est. cards.) and msrd. MI for each PEO of the respective JOB query.

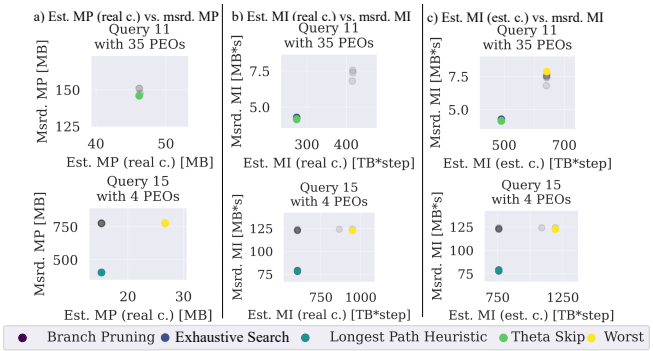


Figure 7: Correlation between a) est. MP (real cards.) and msrd. MP, b) est. MI (real cards.) and msrd. MI, and c) est. MI (est. cards.) and msrd. MI for each PEO of the respective TPC-H query.

MP. The data discussed here is visualized in category (a) of each figure. We show that the best PEOs can be correctly estimated for QEPs with sufficiently large differences in MP.

Join Order Benchmark. For the JOB we visualize the estimated MP (real cards.) against the measured MP for queries with the highest median measured MP for indicated queries in Figure 6a. For this, we visualize six queries that exemplify the queries contained in the benchmark. Each dot represents one PEO with a median msrd. MP. Noise is introduced through oversizing in hash tables or buffers and non-pipeline-breaker memory consumers, which is not captured by our the model. We visualize queries with exemplary behavior in Figure 6a, for which we sample 12, 4, 24, 60, 100, and 400 random PEOs, respectively. For Query 8d, 9d, 25c, and 31c we observe a correlation between est. MP (real cards.) and msrd. MP. This suggests that the cost model works for estimating MP. As seen in Figure 6b, an obvious correlation between MI and MP is introduced through the MI being naturally lower when the MP is lower, i.e., avoiding materialization of many tuples at the same time. While the correlation between MI and MP holds here, it is not a necessary condition. When further compared to the MI, we can see that the MP is less exposed to noise, which is natural as the MP is not depending on estimated execution cost/execution time.

Other benchmarks and customer data. For the TPC-H benchmark we encounter five queries with more than one PEO. Out of these, Queries 11 and 15 show significant estimated differences, which are shown in Figure 7a. For both queries the model is able to correctly predict the order of the PEOs, as can be seen in Figure 7a. For query 15, the algorithms ES and LPH found the best PEO for MP, which also is the best PEO in terms of MI.

For SSB, the model could not find significant differences for the evaluated queries, as shown in Figure 8a. For this query set, we always have a big pipeline breaker that dominates the memory consumption in MP and, as we will see later, in MI. Differences for Query 3.1 and 4.2 are again introduced by noise.

Regarding JCC-H, Query 15, the model was able to estimate differences between the PEOs w.r.t. the est. MP (real cards.). This can be seen in Figure 9a, showing good correlation between estimate and measurement. For Query 11, we did not see large differences in MP in the plot, which is due to missing data points.

With regards to customer queries, we were able to evaluate the MP for Query 3. We could not identify any significant differences for the sampled PEOs.

Summary. The target of this subsection is to show if the given cost model can serve as the means to estimate memory peaks. When considering Queries 8d, 9d, 25c, and 31c of the JOB, as well as JCC-H Query 5, 11, and TPC-H Query 15, we see that the cost model

correctly predicts the best possible PEO for MP. It further shows a good correlation between estimates and measurements. We also see that there is a possible correlation between MI and MP.

Next, we will analyze if the cost model is suitable for estimating the MI in *PEOopt*.

5.4 Cost Model Validation for the Memory Integral

To showcase the viability of our approach realized in *PEOopt*, next we experimentally validate our cost model for the MI in order to see if the approach can be used to lower memory pressure in systems with execution of potentially multiple queries in parallel. For the experiments, we eliminate the impact of cardinality estimation errors from our memory cost estimations by using real cardinalities. Real cardinalities are observed post-mortem from query executions and are used as parameters for our cost models. We evaluate the correlation of the calculated memory cost with the measured memory integral of the PEO. The results discussed in this subsection are presented in part (b) of each respective figure.

Join Order Benchmark. In general, the cost model produces good results for queries with PEOs that have large differences in est. MI. There, the cost model correctly estimates the best PEO. If the estimated difference in MI is not large enough, the correct best PEO cannot be estimated. In Figure 6b, we visualize the est. MI (real cards.) against the msrd. MI. When compared to the MP, the clustering of PEOs is not as intense. This is due to further noise introduced during execution when measuring for the MI, since the MI additionally depends on the execution time of pipelines. For Query 8c, we see a correlation between est. MI (est. cards.) and msrd. MI. For Queries 9d, 19d, and 25c, the divergence from correlation lies within the allocation size for hash tables, which induces overhead and is not modeled. Another reason is the implementation of physical operators as a single pipeline with several helper procedures and data structures, which are not explicitly reflected in the model. In Queries 9d and 19d, this is further amplified, resulting in non-correlation. Queries with good correlation are 25c and 31c. Query 12a shows similar behavior. Here, the relative differences between PEOs are sufficiently big enough to overcome the influence of noise. 68 queries in the benchmark show a vertical or dotted pattern of PEO-dots like in Query 29c, indicating that the model could not find a difference in MI for any PEO. However, all those queries (with the exception of 24c and 29c) have a msrd. MI less than 2.0MB $*$ s. Still, those queries show differences in measurements, which is also due to additional data structures and noise from varying pipeline execution times. However, we argue that the model finds good correlation in most cases and for others the difference is small, while the search overhead itself is cheap, as we see in Subsection 5.6.

Other benchmarks and customer queries. In the TPC-H Benchmark, Queries 11 and 15 show significant estimate differences, which are shown in Figure 7b. For both queries the cost model estimates differences in est. MI (real cards.) and finds the correct order of PEOs w.r.t. msrd. MI.

In SSB we found Query 2.1 to expose significant differences in relative est. MI, as can be seen in Figure 8b. For this query, the absolute msrd. MI and estimated differences are low, such that the model cannot correctly order the PEOs. For the other queries in SSB, the results are similar. The cause are many small pipeline breakers combined with one dominating pipeline breaker. Further, usage of index joins avoids materialization. The SSB is therefore a benchmark where savings are small and unpredictable, as one big breaker dominates the memory consumption and index joins from dimension tables reduce the number of possible PEOs.

For the JCC-H benchmark we visualize Queries 5, 10, and 11 in Figure 9b. The query shows a tendency to a correlation, but with low relative differences in estimate MI. Query 10 does not expose differences in estimates. The prototype finds the correct best PEO, but the worst PEO according to the cost model is not the experimentally verified worst.

For our customer queries we are able to extract real cardinalities and to reconstruct the QEP for Query 3 of our customer query set, as can be seen in Figure 10b. Despite having a high MI of around 42MB $*$ s for all PEOs, we could not find a correlation between est. MI (real cards.) and msrd. MI. The QEP contains 44 hash joins, 27 hash semi joins through semi join reduction and 10 index joins. Furthermore, most pipeline breakers contain at most 2,000 tuples. Aside from sampling 24 PEOs, we are able to execute the LPH (based on est. cardinalities). The est. MI (real c.) is the worst of the shown PEOs, however, the msrd. MI is the best. This suggests that the correlation for real cardinalities for this query is not given, incidentally the correlation is given for estimated cardinalities, as we see in the next section.

Summary. Our measurements show good correlation for relevant queries of the JOB and the JCC-H benchmark. For queries with no significant differences in MI, no correlation could be predicted. Therefore, a correlation between estimate and worst is observed if a difference is estimated. In cases without correlation queries exhibit low absolute MIs, which amplifies the impact of unmodelled behavior like oversized hash tables, as was the case for some quick running queries of the JOB. When the model fails to establish a correlation between est. and msrd. MI, this is due to small differences in both estimated and measured MI. Next, we will compare estimated MIs based on cardinality estimates to evaluate the feasibility of deployment without real cardinalities.

5.5 Impact of Cardinality Estimation

We now investigate if our model can be deployed in real world scenarios to achieve savings. There, only cardinality estimates provided by a cardinality estimator are available. We investigate if we can still observe a correlation for the est. MI (est. cards.) given by the cardinality estimator of SAP HANA. As such, the estimated MI is calculated using the estimated c_{out} costs. We show that estimated cardinalities can serve as a basis for estimating the msrd. MI and are feasible to be used in an industry DBMS. *PEOopt* used cardinality estimates for enumerating PEOs, such that the coloring of the PEOs in the diagrams of this section model the actual choice of the particular algorithms. Colored dots indicate a used algorithm, while grey dots indicate randomly chosen PEOs. Algorithms and randomly sampled PEOs may be identical but may be shown as different dots, when they are overlapping. The data discussed here is visualized in category (c) of each figure.

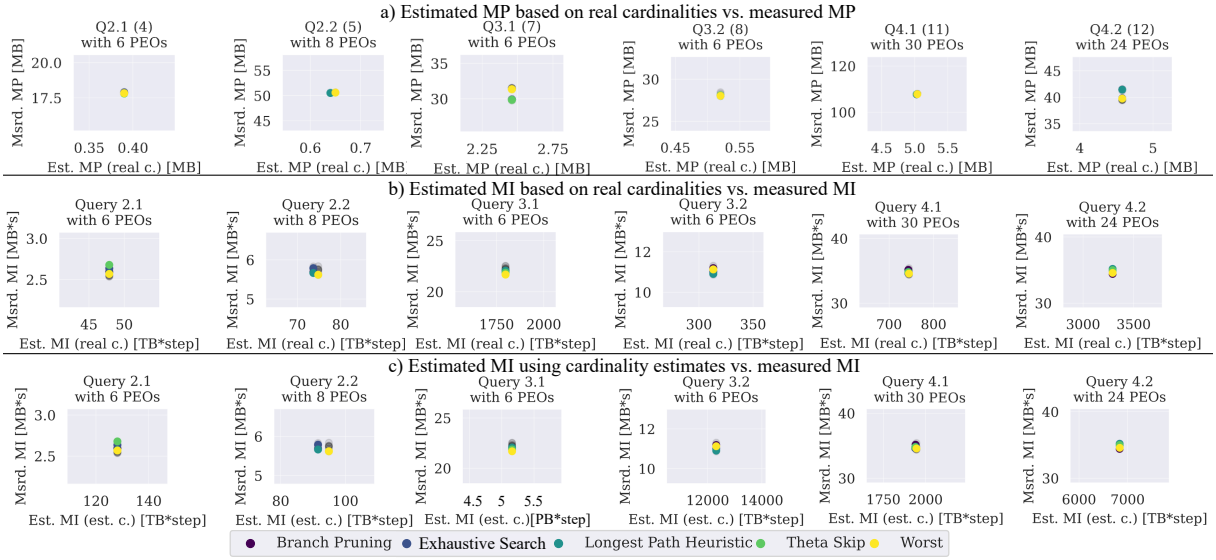


Figure 8: Correlation between a) est. MP (real cards.) and msrd. MP, b) est. MI (real cards.) and msrd. MI, and c) est. MI (est. cards.) and msrd. MI for each PEO of the respective SSB query.

Join Order Benchmark. In general, since the cardinality estimates correlate with real cardinalities, they can be also used as the basis for estimating the MI. For Queries 8d, 25c, and 31c, the previously seen relationship between estimate and measured values is maintained. For the remaining queries the behavior is similar. This fact can be seen in Figure 6c. When taking the algorithms into account, we can see that for all queries with good correlation all algorithms deliver near-optimal or optimal results. Upon closer inspection, we can observe that for Query 31c the PEO found by ES is the same as for the heuristic LPH, which indicates the quality of the heuristic. For Queries 9d and 19d, the relative differences in MI increase even further when taking cardinality estimates as the basis. For Query 29c, there were no changes in the distribution of PEOs. The results indicate that the model can be used with a cardinality estimator of sufficient quality.

Other benchmarks and customer queries. Regarding TPC-H, our prototype finds the best possible PEO for Query 11 and Query 15 when using cardinality estimates as the basis for MI. As the cardinality estimates are quite close to the actual cardinalities, we only see a divergence between MIs based on real cardinalities and estimated cardinalities of at most 2x. This is visible when comparing Sub-figures 7b and c, suggesting a moderate divergence between estimates and real cardinalities.

For the SSB, estimated cardinalities again differ from the real cardinalities. As can be seen in Figure 8b, PEOs diverge along the x-axis while not exhibiting differences in msrd. MI. For all algorithms, the optimizer could not identify good PEOs, as differences in msrd. MI are either small or no significant differences in est. MI (est. cards.) are identified.

W.r.t. JCC-H, we observe a correlation between the observed measures for Query 11, which corresponds to the behavior for the est. MI (real cards.). For Query 15, the results for estimated

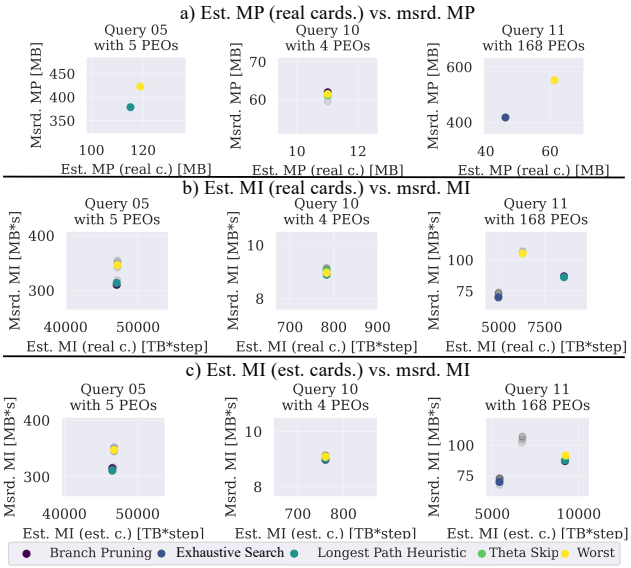


Figure 9: Correlation between a) est. MP (real cards.) and msrd. MP, b) est. MI (real cards.) and msrd. MI, and c) est. MI (est. cards.) and msrd. MI for each PEO of the JCC-H benchmark.

cardinalities are similar to those of the real cardinalities. For both aforementioned queries, we observe a divergence of about 1.5-2x with respect to the estimated MIs of Sub-figures 9b and c. Both cases strengthen the case for the usability of the cardinality estimator for our approach.

For our customer queries we are able to obtain data between est. MI (est. cards.) and msrd. MI for Queries 1 to 4. When looking into the executed QEPs, we see that cardinality estimates are much

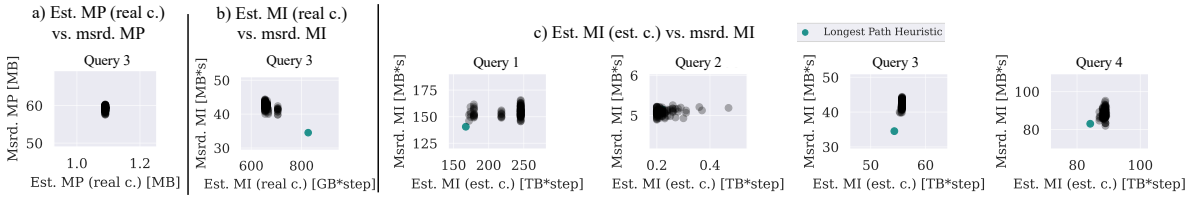


Figure 10: Correlation between a) est. MP (real cards.) and msrd. MP, b) est. MI (real cards.) and msrd. MI, and c) est. MI (est. cards.) and msrd. MI for each PEO of the customer query set.

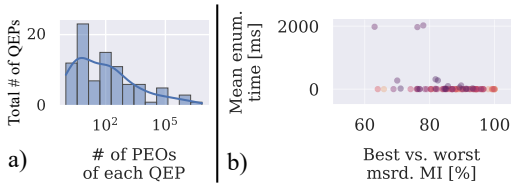


Figure 11: Enumeration complexity for the JOB.

smaller than the real cardinalities (e.g., estimate is 8 tuples, while ~1,650 are pushed). This suggests that frequent hash table resizing occurred, which can have an impact on execution time. In other hash tables, only a small amount of tuples is materialized. When looking at the only possible algorithm, LPH, we can see that the chosen PEO is robust w.r.t. the eventually measured MI. As we can see in Figure 10c, for Query 1, 3, and 4 the PEO is among the best. This suggests that the heuristic works for resolving very complex QEPs and achieves good results.

Summary. We maintained the correlation between est. MI (est. cards.) and msrd. MI for certain queries when taking cardinality estimates as the basis. The quality of the estimated MI is heavily depending on the quality of the cardinality estimator. We see that no mistakes with big differences in msrd. MI are made in choosing the PEO, indicating that deployment of the model yields memory savings. For complex queries LPH yields good results, such is the case for our customer queries.

5.6 Complexity of PEO Enumeration

The following three subsections discuss the lack of impact of our enumeration approach on end-to-end execution time, which is the last contribution to prove real world deployability as shown in Figure 3. The first part is to observe the manageable complexity of our approach. The precise mathematical framework to calculate the complexity of enumerating PEOs for memory consumption has been described in [22], where we showed that the problem may cause combinatorial explosion. In practice, only a small amount of QEPs produce complex AONGs. The number of PEOs per QEP is tied to the number of pipeline breakers in the QEP, the number of joins, and whether the optimizer chooses to ship indexes. In Figure 11a, we visualize the number of PEOs per QEP in a histogram for the JOB. The number of QEPs with high PEO count naturally decreases in a Zipf-distribution reminiscent way, which is tied to the benchmark. We found similar results for the remaining three benchmarks as well, where the data is also included in Figures 4c-g. The results show that for a moderate number of joins, a combinatorial explosion in possible PEOs is unlikely even for exhaustive enumeration. Regarding our customer queries with a high number of joins and high number of PEOs, we saw that either random sampling or finding good PEOs through heuristics is necessary to avoid long enumeration times.

5.7 Relationship between Complexity and Savings

In the previous subsection we saw that the number of PEOs per QEP can be expected to be manageable, which is reflected in the enumeration time needed to find the best PEO. We now observe the execution time that is consumed for enumeration. A visualization of the measured enumeration overhead for the JOB is shown in Figure 11b. We see that the enumeration overhead is not tied to the possible savings. We argue that for the observed savings even exhaustive enumeration is viable and justifiable, since the complexity of the identified QEPs is low.

5.8 Constant Execution Time

Lastly, it is important to observe that changing PEOs does not lead to a variation in the execution time. For the already discussed six JOB queries we plotted the execution times for different sampled PEOs. For those, we execute each and plot the execution time in box plots in Figure 12. The orange lines denote the median, the box's upper and lower bounds the 75th and 25th percentile, and the lines the 100th and 0th percentile. Circles are outliers. As can be seen, there is no drastic difference between individual PEOs. We argue that the observed differences are due to noise and minimal caching effects for previously used hash tables/buffers. With queries that use larger datasets, these effects decline significantly.

6 Related Work

Morsel-based Pipeline Execution Model

The foundations for the pipeline execution model were laid with the push-based Volcano model [18], which already touched upon the memory consumption issue in hash joins. However, Morsel-based parallelism [23] is the basis for our execution model, which is push-based and divides data from a source-operator (e.g., table scan) into morsels.

Memory Optimization for Query Processing

Scheduling. Before the advent of cheap memory, it was necessary to estimate whether query processing can be facilitated using the

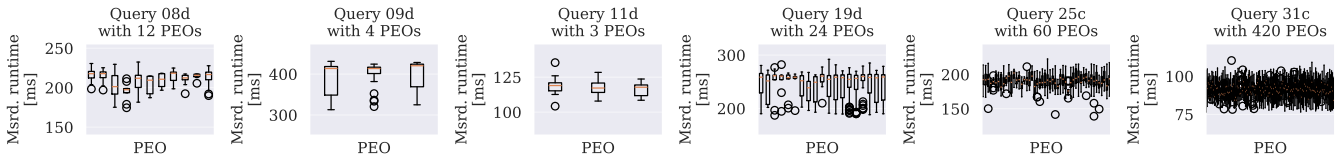


Figure 12: Median runtime for six JOB queries across different PEOs.

available memory to minimize the response time of a query. Early work on disk-based DBMS [10] thus took an approach similar to ours and classified pipelines as tasks and tried to minimize the maximum memory peak through choosing which pipelines to execute, but without regard to overall memory consumption. Instead, they chose to optimize for query response times and assumed that the overall available memory is not sufficient. Focusing on a similar topic is [6], where the authors considered continuous streaming queries and proposed an algorithm to minimize the current memory peak during execution. Another fundamental paper [15] tried to optimize query response time considering communication overhead for distributing work among nodes by describing CPU time as a *time-shared* resource and memory as a *space-shared* resource. Much like [21], both works built upon fundamental scheduling theory literature, but did not aim to minimize overall memory consumption, instead they focused on minimizing query execution times in cases where main memory is scarce.

Since hash tables can overflow in memory during processing, Graefe [18] suggested using a) the query optimizer to anticipate overflow and b) an overflow resolution scheme to make the hash tables for the individual probe step fit into memory. When [18] was written, main memory was considered scarce and spilling partitioned hash tables to disk after the build phase was necessary.

Compression. Our technique serves as a contribution to the overall field of memory consumption optimization in query processing. We can consider compression of intermediate results as one possible approach to facilitate reduced memory consumption, using SIMD-instruction based algorithms to speed up processing and to save memory [13, 25]. The authors of [8] also suggest that the footprint can further be reduced by estimating cardinalities accurately, which avoids pre-allocating too much memory for auxiliary data structures like hash tables [8]. Further savings can be achieved through using compressed or concise auxiliary data structures like concise hash tables [7] or hash tables using perfect hash functions [14].

Usage of Memory Hierarchies. While in-memory DBMS usually try to hold all query processing data in DRAM, a recent proposal [19] compared partitioned and non-partitioned hash joins in NVRAM. While NVRAM seemed to solve issues like the limited memory size of DRAM, manufacturers seem to have abandoned the technology [4]. More fundamentally, [17] discussed using a common buffer cache for both base data and intermediate data and concluded that, while disk based systems used different buffer managers for base data and intermediate data, this leads to further complications in scheduling and communication between buffer managers, which is addressed by recent work [30]. Today, Presto is a DBMS that implements spilling intermediates to disk [3, 35] to temporarily reduce memory consumption without introducing pipeline breakers through partitioning, whereas most industry DBMS like Oracle, PostgreSQL, and SAP HANA partition spill hash tables to disk while processing to reduce memory consumption or avoid out-of-memory situations. Here, they use traditional partitioning-based algorithms that cause additional breakers [1, 2, 36].

7 Conclusion

In this work we present our implementation of PEOopt in a prototype of SAP HANA to evaluate the PEO enumeration approach experimentally. We analyze the potential of the approach for four standard DBMS benchmarks, evaluate the potential memory savings for most of the queries and their QEPs and demonstrate the enumeration complexity. In our evaluation, we first found that queries with more than 1,000 possible PEOs are rare, but the savings potential can be high. We found that our approach can facilitate high memory savings of up to 55%. We also found that our cost model generally produced a correlation between estimated memory consumption and measured memory consumption for queries with high memory integrals and with sufficiently big estimated differences. Further, our cost model produced better correlation with measured memory integrals the higher the absolute measured memory integrals of the individual query was. There, the model is able to achieve improvements of up to 36%. We could see a correlation between estimates and measurements for the memory peak as the observed measure. We also took cardinality estimates given by our DBMS as an input parameter for the pipeline execution cost estimation proposed in [21]. Despite cardinality estimates diverging from the real cardinalities, they still serve as a good basis for full enumeration in simple queries or for a simple heuristic in very complex queries. We also see that for standard benchmarks, all PEOs were enumerable using the Exhaustive Search algorithm. For complex queries from our customer benchmark, however, heuristics are needed. We found that the query runtime is not impacted by the PEO choice.

Acknowledgements

We thank SAP for the provided funding, work environment, and infrastructure. We thank Tobias Goetz, Dirk Habich, Hanna Kruppe, Robert Lasch, Adrian Lutsch, and Ulrike Schöbel for providing feedback on the writing style and plausibility of the paper. This work was partly funded by (1) the German Research Foundation (DFG) via a Reinhart Koselleck-Project (LE-1416/28-1), (2) the German Research Foundation (DFG) priority program SPP 2377 under grant no. LE 1416/30-1, and (3) the Horizon Europe research and innovation program under grant agreement no. 101189551 (CHORYS).

References

[1] [n. d.]. Oracle SQL and PL/SQL Optimization for Developers: Hash Join. https://oracle.readthedocs.io/en/stable/sql/joins/hash-join.html. Accessed: 2024-07-08.

[2] [n. d.]. PostgreSQL Spill to Disk Recommendations in Redgate Monitor. https://www.red-gate.com/hub/product-learning/redgate-monitor/postgresql-spill-to-disk-recommendations-in-redgate-monitor. Accessed: 2024-07-08.

[3] [n. d.]. Presto Documentation: Spill to Disk. https://prestodb.io/docs/current/admin/spill.html. Accessed: 2023-12-07.

[4] [n. d.]. Why Intel killed its Optane memory business. https://www.theregister.com/2022/07/29/intel_optane_memory_dead/. Accessed: 2023-10-13.

[5] Minseon Ahn, Andrew Chang, Donghun Lee, Jongmin Kim, Jaemin Jung, Oliver Rebholz, Vincent Pham, Krishna Malladi, and Yang Seok Ki. 2022. Enabling CXL Memory Expansion for In-Memory Database Management Systems. In *Proceedings of the 18th International Workshop on Data Management on New Hardware (DaMoN '22)*.

[6] Brian Babcock, Shivnath Babu, Rajeev Motwani, and Mayur Datar. 2003. Chain: Operator scheduling for memory minimization in data stream systems. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*. 253–264.

[7] Ronald Barber, G. Lohman, Ippokratis Pandis, Vijayshankar Raman, R. Sidle, G. Attaluri, N. Chainani, Sam Lightstone, and D. Sharpe. 2014. Memory-efficient hash joins. *Proceedings of the VLDB Endowment* 8 (12 2014), 353–364. https://doi.org/10.14778/2735496.2735499

[8] Martin Boissier. 2018. Reducing the Footprint of Main Memory HTAP Systems: Removing, Compressing, Tiering, and Ignoring Data. In *PhD@VLDB*. https://api.semanticscholar.org/CorpusID:52128331

[9] Peter A. Boncz, Angelos-Christos G. Anadiotis, and Steffen Kläbe. 2017. JCC-H: Adding Join Crossing Correlations with Skew to TPC-H. In *TPC Technology Conference*.

[10] Luc Bouganim, Olga Kapitskaia, and Patrick Valduriez. 1998. Memory-adaptive scheduling for large query execution. In *Proceedings of the seventh international conference on Information and knowledge management*. 105–115.

[11] Peter Brucker and Sigrid Knust. 2011. *Complex Scheduling* (2nd ed.). Springer Publishing Company, Incorporated.

[12] Houssine Chetto, Maryline Chetto, and T. Bouchentouf. 1990. Dynamic Scheduling of Real-Time Tasks under Precedence Constraints. *Real-Time Systems* 2 (1990), 181–194. https://api.semanticscholar.org/CorpusID:1291941

[13] Patrick Damme, Annett Ungethüm, Johannes Pietrzyk, Alexander Krause, Dirk Habich, and Wolfgang Lehner. 2020. MorphStore: Analytical Query Engine with a Holistic Compression-Enabled Processing Model. *Proc. VLDB Endow.* 13, 12 (jul 2020), 2396–2410. https://doi.org/10.14778/3407790.3407833

[14] Kevin P. Gaffney and Jignesh Patel. 2024. Is Perfect Hashing Practical for OLAP Systems? *CIDR* (January 2024). https://www.cidrdb.org/cidr2024/papers/p65-gaffney.pdf

[15] Minos Garofalakis and Yannis Ioannidis. 1997. Parallel Query Scheduling and Optimization with Time- and Space-Shared Resources. (07 1997).

[16] Georgios K. Giannikis, Gustavo Alonso, and Donald Kossmann. 2012. SharedDB: Killing One Thousand Queries With One Stone. *Proc. VLDB Endow.* 5 (2012), 526–537. https://api.semanticscholar.org/CorpusID:6276571

[17] Goetz Graefe. 1993. Query Evaluation Techniques for Large Databases. *ACM Comput. Surv.* 25, 2 (June 1993), 73–169. https://doi.org/10.1145/152610.152611

[18] G. Graefe. 1994. Volcano— An Extensible and Parallel Query Evaluation System. *IEEE Trans. on Knowl. and Data Eng.* 6, 1 (feb 1994), 120–135. https://doi.org/10.1109/69.273032

[19] Wentao Huang, Yunhong Ji, Xuan Zhou, Bingsheng He, and Kian-Lee Tan. 2023. A Design Space Exploration and Evaluation for Main-Memory Hash Joins in Storage Class Memory. *Proc. VLDB Endow.* 16, 6 (apr 2023), 1249–1263. https://doi.org/10.14778/3583140.3583144

[20] Kihong Kim, Hyunwook Kim, Jinsu Lee, Taehyung Lee, Alexander Böhm, Norman May, Guido Moerkotte, Daniel Ritter, Ralf Dentzer, Heiko Gerwens, Irena Kofman, and Mihnea Andrei. 2025. Enterprise Application-Database Co-Innovation for Hybrid Transactional/Analytical Processing: A Virtual Data Model and Its Query Optimization Needs. In *Companion of the 2025 International Conference on Management of Data, SIGMOD/PODS 2025, Berlin, Germany, June 22-27*. ACM. to appear.

[21] Lukas Landgraf, Wolfgang Lehner, Florian Wolf, and Alexander Boehm. 2022. Memory Efficient Scheduling of Query Pipeline Execution. In *12th Conference on Innovative Data Systems Research, CIDR 2022, Chaminade, CA, USA, January 9-12, 2022*. www.cidrdb.org. https://www.cidrdb.org/cidr2022/papers/p82-landgraf.pdf

[22] Lukas Landgraf, Florian Wolf, and Wolfgang Lehner. 2025. Complexity Analysis of Pipeline Execution Order Enumeration for Query Execution Plans. In *Datenbanksysteme für Business, Technologie und Web (BTW 2025)*. Gesellschaft für Informatik, Bonn, 95–112. https://doi.org/10.18420/BTW2025-04

[23] Viktor Leis, Peter Boncz, Alfons Kemper, and Thomas Neumann. 2014. Morsel-Driven Parallelism: A NUMA-Aware Query Evaluation Framework for the Many-Core Age. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data* (Snowbird, Utah, USA) *(SIGMOD '14)*. Association for Computing Machinery, New York, NY, USA, 743–754. https://doi.org/10.1145/2588555.2610507

[24] Viktor Leis, Bernhard Radke, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. 2018. Query Optimization through the Looking Glass, and What We Found Running the Join Order Benchmark. 27, 5 (oct 2018), 643–668. https://doi.org/10.1007/s00778-017-0480-7

[25] Christian Lemke, Kai-Uwe Sattler, Franz Faerber, and Alexander Zeier. 2010. Speeding Up Queries in Column Stores. In *Data Warehousing and Knowledge Discovery*, Torben Bach Pedersen, Mukesh K. Mohania, and A. Min Tjoa (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 117–129.

[26] Feng Li, Manoj Sy, and Vivek Narasayya. 2016. Accelerating Relational Databases by Leveraging Remote Memory and RDMA. 355–370. https://doi.org/10.1145/2882903.2882949

[27] Norman May, Alexander Böhm, and Wolfgang Lehner. 2017. SAP HANA – The Evolution of an In-Memory DBMS from Pure OLAP Processing Towards Mixed Workloads. In *Datenbanksysteme für Business, Technologie und Web (BTW 2017)*. Gesellschaft für Informatik, Bonn, 545–546.

[28] Norman May, Alexander Böhm, Daniel Ritter, Frank Renkes, Mihnea Andrei, and Germany Wolfgang Lehner Walldorf. 2025. SAP HANA Cloud: Data Management for Modern Enterprise Applications. In *Companion of the 2025 International Conference on Management of Data, SIGMOD/PODS 2025, Berlin, Germany, June 22-27*. ACM. to appear.

[29] Patrick O'Neil, Elizabeth O'Neil, Xuedong Chen, and Stephen Revilak. 2009. *The Star Schema Benchmark and Augmented Fact Table Indexing*. Springer-Verlag, Berlin, Heidelberg, 237–252. https://doi.org/10.1007/978-3-642-10424-4_17

[30] Riki Otaki, Jun Hyuk Chang, Charles Benello, Aaron J. Elmore, and Goetz Graefe. 2025. Resource-Adaptive Query Execution with Paged Memory Management. In *15th Conference on Innovative Data Systems Research, CIDR, Amsterdam, Netherlands, January 19-22, 2025*. www.cidrdb.org.

[31] Meikel Poess and Raghu Nambiar. 2010. TPC Benchmark H Standard Specification. https://doi.org/10.13140/RG.2.1.1883.9288

[32] Iraklis Psaroudakis, Florian Wolf, Norman May, Thomas Neumann, Alexander Böhm, Anastasia Ailamaki, and Kai-Uwe Sattler. 2014. Scaling Up Mixed Workloads: A Battle of Data Freshness, Flexibility, and Scheduling. In *TPC Technology Conference*. https://api.semanticscholar.org/CorpusID:1953646

[33] Daniel Ritter. 2023. HEX: SAP's new HANA Execution Engine. https://www.cidrdb.org/cidr2023/slides/sponsor-talk-sap-slides.pdf. In *13th Conference on Innovative Data Systems Research*. www.cidrdb.org. Accessed: 2024-08-22.

[34] Wolfgang Scheufele and Guido Moerkotte. 1997. On the complexity of generating optimal plans with cross products (extended abstract). In *Proceedings of the Sixteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS '97)*. Association for Computing Machinery, New York, NY, USA, 238–248. https://doi.org/10.1145/263661.263687

[35] Raghav Sethi, Martin Traverso, Dain Sundstrom, David Phillips, Wenlei Xie, Yutian Sun, Nezih Yegitbasi, Haozhun Jin, Eric Hwang, Nileema Shingte, and Christopher Berner. 2019. Presto: SQL on Everything. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. 1802–1813. https://doi.org/10.1109/ICDE.2019.00196

[36] Reza Sherkat, Colin Florendo, Mihnea Andrei, Rolando Blanco, Adrian Dragusanu, Amit Pathak, Pushkar Khadilkar, Neeraj Kulkarni, Christian Lemke, Sebastian Seifert, Sarika Iyer, Sasikanth Gottapu, Robert Schulze, Chaitanya Gottipati, Nirvik Basak, Yanhong Wang, Vivek Kandiyanallur, Santosh Pendap, Dheren Gala, Rajesh Almeida, and Prasanta Ghosh. 2019. Native store extension for SAP HANA. 12, 12 (Aug. 2019), 2047–2058. https://doi.org/10.14778/3352063.3352123

[37] K. Stocker, D. Kossmann, R. Braumandi, and A. Kemper. 2001. Integrating semi-join-reducers into state-of-the-art query processors. In *Proceedings 17th International Conference on Data Engineering*. 575–584. https://doi.org/10.1109/ICDE.2001.914872