Memory Efficient Scheduling of Query Pipeline Execution

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

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

Alexander Boehm
SAP SE, Walldorf, Germany
alexander.boehm@sap.com

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

ABSTRACT

State-of-the-art query engines pursue a pipeline-based query execution model. Using such a model, a pipeline computes a query plan fragment up to a pipeline breaker resulting in an intermediate result, which will be consumed by subsequent pipelines. Interestingly, the ordering of execution of such pipelines poses an opportunity for memory savings. Within this paper, we tackle the challenge to compute an optimal schedule of the individual pipelines with respect to minimizing the memory consumption needed for a particular query execution plan. We therefore will precisely state the problem and show the potential of an optimal pipeline execution ordering. We will then provide a formal model to describe the search space and propose four different algorithms to identify optimal/near-optimal schedules. The paper also presents insights into our implementation within a prototypical query execution engine and reports results relying on the Join Order Benchmark scenarios. Specifically, the experimental evaluation focuses on the identified memory savings and the stability of the query runtime behavior. Furthermore, the evaluation reports on the small overhead of the proposed search algorithms during the planning time, thus emphasizing the practical applicability of our approach.

ACM Reference Format:
Lukas Landgraf, Florian Wolf, Alexander Boehm, and Wolfgang Lehner. 2022. Memory Efficient Scheduling of Query Pipeline Execution. In *Proceedings of CIDR '22: Conference on Innovative Data Systems Research (CIDR '22).* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

The optimization of memory consumption during query processing plays a significant role due to ever growing demands of data size [2], query workload and query complexity in cloud scenarios. This observation holds especially for main-memory database systems aiming to optimally make use of existing hardware resources. Furthermore, this especially holds for main memory, as it is a driving factor of hardware costs [1], to increase memory utilization. In such modern cloud systems, we are confronted with large numbers of complex OLAP queries, referencing numerous tables [14]. Those queries are typically executed using pipelines as the state-of-the-art execution model to achieve scalability and throughput [9, 15].

CIDR '22, January 9-12, 2022, Santa Cruz Chaminade, CA

ACM ISBN 978-1-4503-XXXX-X/18/06.
https://doi.org/10.1145/1122445.1122456

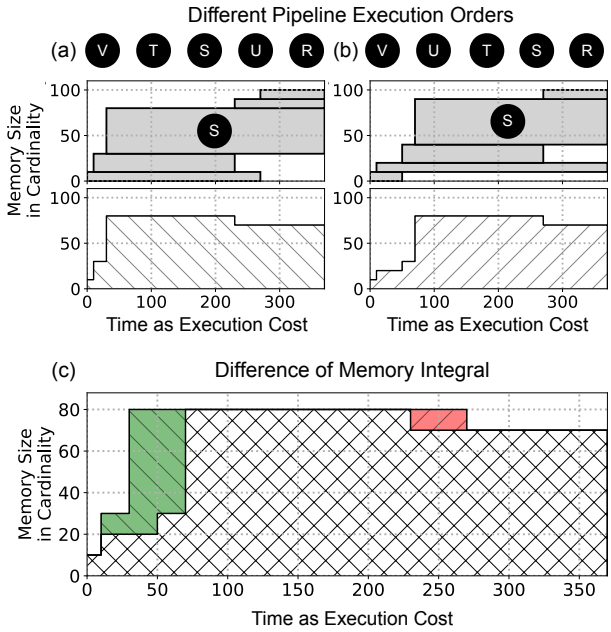

Figure 1: Comparison of memory consumption of pipeline breakers over time between two different pipeline execution orders illustrating potential memory savings.

Within such an execution model, pipelines [4, 8] are logical units of executions and thus independent of each other during execution. For scheduling, a pipeline may require several other pipelines to be processed first. A pipeline performs a mix of unary operators, might probe into the output structures of other pipelines and eventually builds another intermediate acting as a pipeline breaker. Since intermediates have different sizes and the number of existing intermediates is varying during the execution of the individual pipelines, the memory utilization will change over time. Moreover, the order of execution of individual pipelines, i.e. the time of producing, storing, and consuming the intermediate by subsequently executed pipelines has a significant impact on the *memory profile* (the timely utilization of main memory resources as a graph) of a query.

Figure 1 illustrates the potential memory savings of the example query execution plan in Figure 2 joining five tables R, S, T, U, and V. The upper diagrams in Figures 1(a) and (b) contain the lifetimes of the breakers of the pipelines, which themselves contain the

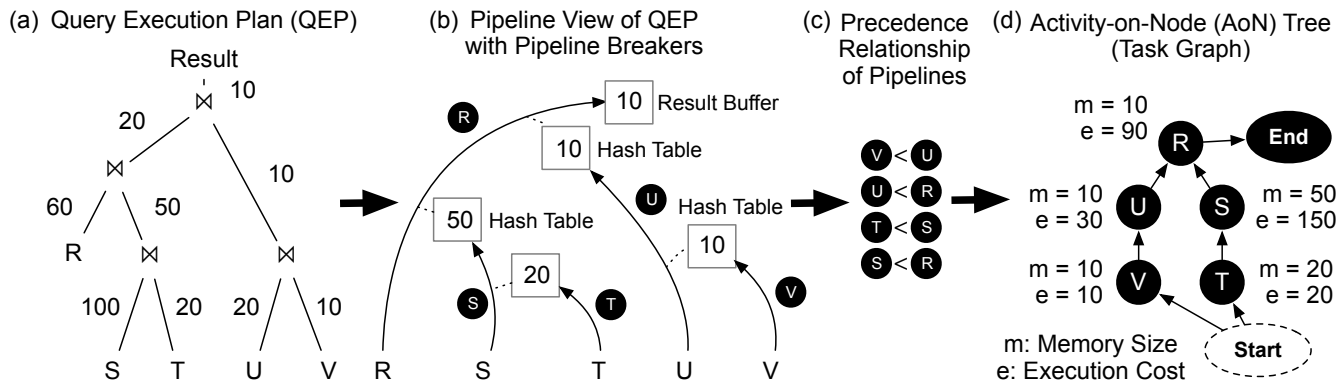

Figure 2: Example QEP of joining five tables with annotated cardinalities, extended by according pipeline view with pipeline breakers and their sizes, from which a precedence relationship of pipelines and further an AoN-Tree can be derived.

intermediates at the end of each pipelines. Those breaker lifetimes add up and result in the memory profile in the lower part of said two figures. In Figure 2, the intermediate result cardinalities of the individual pipeline fragments are annotated at the query execution plan edges. To evaluate the memory usage of the query execution plan over time, we need the memory sizes and the lifetimes of the pipeline breakers. For the example in Figure 2(b) and (d), we assume that the memory sizes of the pipeline breakers are equal to the cardinality of the intermediate result in the pipeline breaker. The lifetime of a pipeline breaker depends on the execution time, i.e., the execution cost, of other pipelines. Because the example is a pure join query, we abstract the execution cost of a pipeline to be the number of hash join probes executed by the pipeline.

From the precedence relationship of pipelines in Figure 2(c), we can derive at least two different pipeline execution schedules resulting in creation and consumption of intermediates at different points during query execution. The two execution orders in Figure 1, (V,U,T,S,R) and (V,T,S,U,R) thus result in different memory profiles. In the plot at the bottom of Figure 1, the green area shows the savings, and the red area the losses of the execution order (V,U,T,S,R) compared to (V,T,S,U,R): executing pipeline (S) with the large intermediate as late as possible seems favorable to keep this intermediate in memory as short as possible. Comparing the two different memory profiles of these execution orders shows a potential memory utilization reduction of 6.3%, illustrating the potential impact of pipeline execution orderings.

Contributions

We introduce a lightweight optimization step **after** query optimization but before plan execution, which can be easily integrated into existing DBMS. Our contributions build on query execution plans as the output of a query optimizer, annotated with estimated cost and cardinalities. Based on these foundations, the paper encompasses the following contributions:

- This paper introduces *Pipelines Execution Orders* (PEOs) as a conceptional model for our scheduling optimizations. (Section 2)
- We show the potential of optimizing the schedule of query execution pipelines with respect to memory consumption during the lifetime of a query. (Subsection 5.2)
- The paper discusses different optimization goals (memory integral; peak consumption; memory robustness) and make the case for the *memory integral* as a measure for the memory consumption of a given join tree. (Subsection 3.1)
- We introduce a formal framework to efficiently derive an **optimal** schedule. (Subsection 3.2)
- To identify query execution pipeline schedules, we describe and evaluate different algorithms (Exhaustive Search, Branch Pruning Search, Theta Skip Search, and Longest Path Heuristic). (Subsection 3 and Section 4)
- We provide insights of an integration into an existing prototypical database engine applying the pipeline-based execution model and investigate the additional planning overhead. (Subsection 5.5)
- We experimentally show that the schedule may only have an impact on the memory utilization over time. (Subsection 5.4)

As of now, we rely on the observation of [9] that one pipeline-at-a-time achieves the best throughput, and thus keep a system busy by processing only a single pipeline at any time; the memory optimized scheduling of multiple concurrent pipelines is subject to future research and goes beyond the mission of this CIDR-publication.

Structure

This paper continues with the introduction of PEOs as conceptional model for our scheduling optimizations in Section 2 and presents different scheduling algorithms in the Sections 3 and 4. Section 5 will then provide insights into the implementation and findings of the experimental evaluation. The paper closes with a brief review of related work (Section 6) and a summary (Section 7).

2 PIPELINE EXECUTION ORDER

Our underlying conceptual model interprets a query execution plan as a **set of pipelines** and a **set of pipeline dependencies**, as illustrated in Figure 2. A pipeline is a sequence of physical operators,

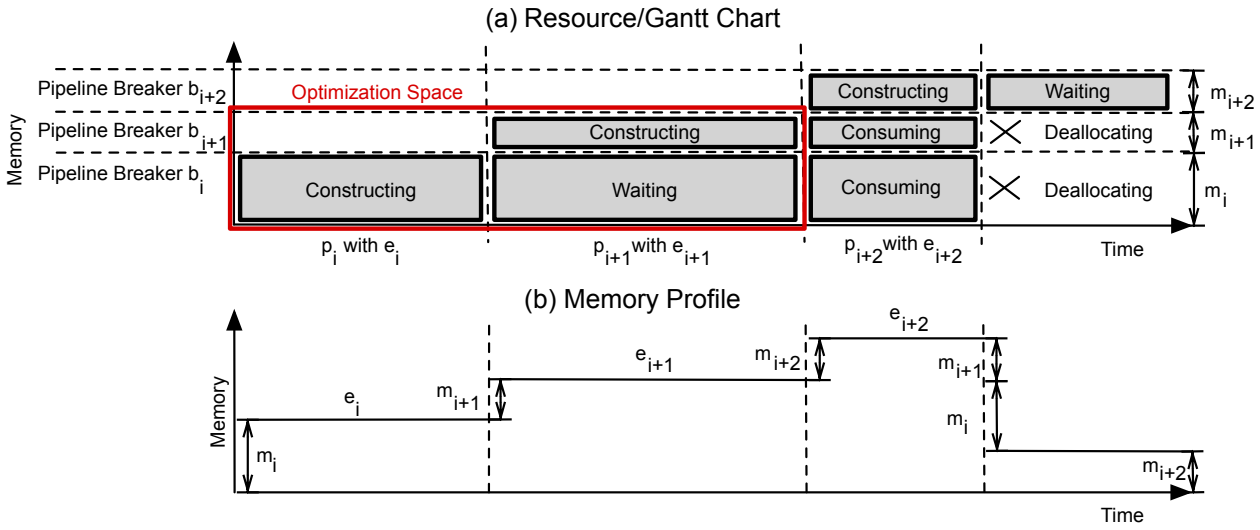

Figure 3: Example of memory consumption charts: Resource (Gantt) chart showing the consumed memory of each pipeline over time, and corresponding memory profile, showing the sum of used memory resources over time.

in which fragments of intermediate results are passed from one operator to the next. We refer to hash joins as our standard example of a physical operator. The build-side of a hash join then corresponds to the end of a pipeline, called a pipeline breaker. A pipeline with potentially multiple operators may then also depend on the intermediate results in the pipeline breakers of other pipelines. In the example of a join query using hash joins, one pipeline may contain the probe-sides of different hash joins, probing into the corresponding build-sides, i.e., pipeline breakers of other pipelines. Once a pipeline has finished and produced a new intermediate or the final result, all referenced intermediates, e.g., hash tables against which the current pipeline has probed, can be released. In other words: once a pipeline breaker is built, it has to stay in memory until it has been consumed, e.g., probed.

To illustrate the optimization space we utilize, Figure 3(a) shows an example Gantt-chart [5] containing the different phases in the lifetime of three pipeline breakers b_i, $b_{i,1}$, and $b_{i,2}$ of the corresponding pipelines p_i, $p_{i,1}$ and $p_{i,2}$. From the Gantt-chart on the upper part of Figure 3, we can derive the memory profile or memory integral in 3(b). The y-axis in Figure 3 describes the consumed memory size of the pipeline breakers, and the x-axis the lifespan of the pipeline breakers, which can be separated into four phases: (1) The *constructing* phase, which is dominated by the execution of the preceding pipeline, and fills the breaker with tuples. The entire hash table is assumed to be allocated at an instant and before execution. (2) The *waiting* phase, where the breaker is not touched and waits to be consumed. (3) The *consumption* phase, where another pipeline reads the intermediate result from the pipeline breaker, e.g., through probing. (4) The *deallocating* phase, which frees the memory of the pipeline breaker.

The example in Figure 3 further shows the interdependencies between the phases of the three different pipelines. The pipeline breakers b_i and $b_{i,1}$ are supposed to be consumed by the pipeline $p_{i,2}$. For a correct execution, it does not matter whether b_i or $b_{i,1}$ is executed first. This degree of freedom creates the optimization space in this example. Figure 3 illustrates the worse decision, i.e., executing p_i first: The intermediate result in b_i is larger compared to $b_{i,1}$, and the larger result in b_i has to stay in memory for the construction time of $b_{i,1}$, which is considerably longer compared to the construction time of b_i.

Definition 1 (Pipeline Execution Order). *A Pipeline Execution Order (PEO) O is a sequence of n pipelines $^1p_1, \ldots, p_i, \ldots, p_j, \ldots, p_n{}^\circ$ given in a query execution plan where pipeline p_i Ÿ p_j, if the computation of pipeline p_j requires the output of pipeline p_i. For all dependencies p_i Ÿ p_j in O there must be no sub-sequence $^1\ldots, p_j, \ldots, p_i, \ldots{}^\circ$, such that the PEO fulfills all dependencies.*

We refer to the term *precedence relationships* [5, 6] for the dependencies p_i Ÿ p_j (see Figure 2). The notation p_i Ÿ p_j (read p_j depends on p_i) is used for interdependent tasks in scheduling theory literature [5, 6] and we use it similarly to describe legal orderings of pipelines. The precedence relationships of individual pipelines can be derived from the QEP according to the selected physical operators. Alternatively, the dependencies can be explicitly provided by the query optimizer.

In our experiments in Subsection 5.4, we found that different PEOs result in the same query execution time, because all pipelines of a QEP have to be executed. Solely the memory consumption can be different. Typically, query execution engines choose one QEP that is correct according to the definition, but not necessarily a QEP that has optimal memory consumption.

3 OPTIMAL SCHEDULING ALGORITHMS

In order to find an optimal PEO with respect to memory consumption, we enumerate PEOs, assign cost to each PEO, and consider the cheapest PEO as optimal. Next, we discuss optimization goals,

from which we derive our memory cost model for PEOs. Further, we describe the PEO search space and present two enumeration algorithms, which traverse the Activity-on-Node Tree (AoN-Tree) and guarantee to find the optimal PEO according to estimated cost.

3.1 Optimization Goals

Optimizing the PEO with respect to memory consumption can have different objectives. We identified the following optimization goals:

Minimizing Memory Integral: The memory integral quantifies the space below the memory profile, which describes the memory utilization over the query execution time. For a given PEO, the memory integral is computed by accumulating the product of pipeline breaker lifetime and pipeline breaker size over all pipeline breakers in the query execution plan. The lifetime of a pipeline breaker starts at the beginning of the corresponding pipeline and ends once the intermediate in the pipeline breaker was consumed, i.e., at the end of its consuming pipeline. Thus, the lifetime of the pipeline breaker also depends on the execution time of sibling pipelines that have to be executed before the consuming pipeline can start.

Minimizing Peak Consumption: Next to minimizing the memory integral, minimizing the peak consumption aims at generating a PEO with the lowest peak memory consumption during the lifetime of a query. While this strategy might be extremely beneficial for multi-query scenarios with extreme memory pressure, we position this strategy as a tie breaker for PEOs with the same memory integral. During our experimental evaluations, we observed an implicit reduction of the average memory consumption in PEOs with a lower overall memory integral.

Robust Memory Consumption: Minimizing the variance of the memory consumption throughout the lifetime of a query is the third optimization goal and aims for a more constant, i.e. more stable memory consumption behavior. This strategy is particularly interesting from a system perspective, ideally looking at all actively running queries in a system. Because this is beyond the scope of the optimization of a single query and thus beyond the scope of this paper, we do not further elaborate on this aspect.

Throughout our experiments in Section 5, we identified the strongest benefits when minimizing the memory integral. Consequently, we continue with a memory integral cost model for PEOs.

3.2 Cost Model

Next, we describe the building blocks of a memory-integral-based cost model for PEOs. In addition to the PEO, another input of our cost model is the corresponding QEP with estimated intermediate result cardinalities and estimated costs of all sub-trees in the query execution plan.

To formalize our cost function, we rely on the notation shown in Figure 5: A QEP has sub-tress s_i with estimated execution cost c_i. The result of a sub-tree s_i has the cardinality f_i, the row size w_i, and is written into the pipeline breaker b_i. Pipeline breaker b_i marks the end of pipeline p_i. In contrast to the execution cost c_i of the entire sub-tree s_i, the cost of executing just pipeline p_i is execution cost e_i. Because the estimated execution cost c_i of sub-tree s_i does not contain the cost of writing the result of sub-tree s_i into pipeline breaker b_i, we have to consider this effort as execution cost σ_i. Finally, l_i is the lifetime of a pipeline breaker b_i. The sub-trees s_i, their estimated cardinalities f_i, row size w_i, and execution costs c_i and σ_i, are input parameters of our cost model and have to be calculated in the initial query optimization.

Var.	Definition	[Unit]/(Type)
s_i	Sub-Tree	no unit
p_i	Pipeline of s_i	no unit
c_i	Generic Cost Model Cost for sub-tree s_i	[step]
e_i	Execution Cost of p_i	[step]
b_i	Pipeline Breaker of p_i	no unit
m_i	Memory Cost of b_i	[B]
f_i	Cardinality of b_i	no unit
w_i	Tuple Size of b_i	[B]
σ_i	Writing Costs of b_i	[step]
l_i	Lifetime of b_i	[step]
MI	Memory Integral	[step*B]
O	PEO	(tuple of pipelines)
P	Set of all pipelines	(set)
D	Dependency Matrix	(matrix)

Figure 4: This table lists all variables used in Section 3.2

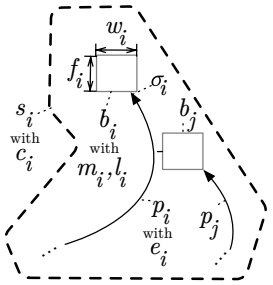

Figure 5: Illustration of pipelines with annotated notation, showing a QEP subtree s_i having execution cost c_i, a pipeline p_i having execution cost e_i, and the corresponding pipeline breaker b_i, storing a result with cardinality f_i, row size w_i, an overall memory size m_i, and writing cost σ_i.

3.2.1 Memory Size of a Pipeline Breaker. We approximate the memory size m_i (unit [B]) of a pipeline breaker b_i by the product of its corresponding cardinality f_i and row size w_i so that: $m_i = w_i \ \ f_i$. In the example of hash joins, this approximation is based on the assumption that pipeline breakers consist of densely-packed hash tables to be oblivious to the hash table implementation. We further define the vector of pipeline breaker memory sizes $\hat{m}$ for a given PEO $O = {}^1p_0, \ldots, p_i, \ldots, p_n{}^0$ as $\hat{m} = {}^1m_1, \ldots, m_i, \ldots, m_n{}^0$.

3.2.2 Pipeline Execution Time. To approximate the lifetime of a single pipeline breaker, we first have to approximate the execution times of single pipelines. We argue that the execution time

of a pipeline can be approximated by the execution cost e_i of the pipeline, which we propose to derive as follows: Take the estimated execution cost c_i of the corresponding sub-tree s_i and subtract the estimated execution cost c_j of all sub-trees s_j of s_i, where s_j are only sub-trees whose pipeline breaker b_j is consumed by pipeline p_i. To further approximate the pipeline execution time σ_i is added to e_i in order to account for the cost of writing the result of sub-tree s_i and pipeline p_i into pipeline breaker b_i.

$$e_i = c_i \quad \sum_{p_j < p_i} c_j \ , \ \sigma_i \tag{1}$$

For our running example, we use the C_{out} cost function [7], which just adds up the output cardinalities of all operators, including base table scans. Because the basic C_{out} cost function does not split the cost of a hash join in build and probe cost, we set the cost σ_i of writing into the pipeline breaker, e.g., the hash table, to 0, since C_{out} already includes the cardinality of the result.

Equation 1, however, does not specify which costs obtained from the cost function c_j to subtract from c_i. Next we will define a dependency matrix to correctly calculate the pipeline execution costs for all pipelines.

Recall that a PEO O is defined as a sequence of pipelines $O = {}^1p_k, \ldots, p_i, \ldots, p_j{}^o$ based on the set of pipelines $P = \mathtt{f}p_1, ..., p_n\mathtt{g}$. In order to be able to reflect the precedence relationship we use the matrix $D : P \quad P \ ! \ \mathtt{f}0, 1\mathtt{g}$, called the dependency matrix. The matrix is filled with 1s where the corresponding pipeline of each row is depending on the pipeline breaker of the pipeline indicated by the column:

$$D = \begin{array}{c} \\ p_1 \\ \ldots \\ p_i \\ \ldots \\ p_n \end{array} \begin{array}{c} \begin{array}{ccccc} p_1 & \ldots & p_j & \ldots & p_n \end{array} \\ \begin{pmatrix} 0 & \ldots & 1 & \ldots & 0 \\ \ldots & \ldots & \ldots & \ldots & \ldots \\ 0 & \ldots & 0 & \ldots & 0 \\ \ldots & \ldots & \ldots & \ldots & \ldots \\ 0 & .. & 0 & \ldots & 0 \end{pmatrix} \end{array}$$

or more formally:

$$D_{i,j} = \begin{cases} 1 \text{ if } p_i \text{ depends on } p_j \\ 0 \text{ else} \end{cases}$$

We denote $\mathbf{c} = {}^1c_k, \ldots, c_i, \ldots, c_j{}^o$ as the vector containing the execution costs c_i of a subtree s_i according to pipeline p_i, in the order of pipelines in the PEO O. Also, we define a vector $\mathbf{f} = {}^1f_k, \ldots, f_i, \ldots, f_j{}^o$ of corresponding result cardinalities in each pipeline breaker, and vector $\boldsymbol{\sigma} = {}^1\sigma_k, \ldots, \sigma_i, \ldots, \sigma_j{}^o$ to account the cost of writing into the pipeline breakers. Eventually, we can derive the equation for the vector $\mathbf{e}$ of execution costs e_i for each pipeline p_i:

$$\mathbf{e} = {}^{11}E \quad D^o \quad \mathbf{c}^{>o>} \ , \ \boldsymbol{\sigma}, \tag{2}$$

where E is the unity matrix in the n-th dimension. Operation " " is the standard matrix multiplication and ">" denotes the transposition of a matrix. Subtracting D from E, gives us a matrix where each line specifies which cost should be added and subtracted to calculate the execution cost of a single pipeline. Multiplying the matrix $E \quad D$ with the transposed execution cost vector $\mathbf{c}$, gives us a vector of execution cost per pipeline to which we add $\boldsymbol{\sigma}$ to eventually obtain the final pipeline execution cost vector $\mathbf{e}$.

Example. We consider the example of Figure 2. Let us derive the dependency matrix D for the PEO (V,T,S,U,R) with the precedence relationship V < U, U < R, T < S, and S < R. The annotated pipeline execution cost e_i in the AoN-Tree in Figure 2 can be written as: $\mathbf{e} = {}^110, 20, 150, 30, 90^o$. To keep the example simple, we set $\boldsymbol{\sigma} = 0$, without loss of generality. Consequently, the dependency matrix D is:

$$D = \begin{array}{c} \\ V \\ T \\ S \\ U \\ R \end{array} \begin{array}{c} \begin{array}{ccccc} V & T & S & U & R \end{array} \\ \begin{pmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 \end{pmatrix} \end{array}$$

Recall that the order of pipelines in both dimensions of the D-matrix corresponds to the order on pipelines given by the PEO. The second cell in the third row of D has a "1", since S depends on T, the first cell in the fourth row of D has the entry "1", since U depends on V, and the third and fourth cell in the fifth row of D contain "1"s, since R depends on S and U. We derive the vector $\mathbf{c}$ of sub-tree execution cost c_i from the C_{out} costs calculated from the intermediate result cardinalities, annotated in the QEP in Figure 2:

$$\mathbf{c}_{out} = {}^110, 20, 170, 40, 300^o$$

Inserting $\mathbf{c}_{out}$ into Equation 2 eventually yields to the transposed result vector. Recall that we defined $\boldsymbol{\sigma} = 0$ in this example:

$$\mathbf{e} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 & 1 \end{pmatrix} \begin{pmatrix} 10 \\ 20 \\ 170 \\ 40 \\ 300 \end{pmatrix} = \begin{pmatrix} 10 \\ 20 \\ 150 \\ 30 \\ 90 \end{pmatrix}$$

3.2.3 Lifetime of Pipeline Breakers. The lifetime of a pipeline breaker b_i consists of the execution time e_i of its respective pipeline p_i, and the execution times of other pipelines until the pipeline breaker b_i is consumed. For the example PEO (V, T, S, U, R) in Figure 1, the lifetime of pipeline breaker of pipeline V is the sum of the execution times of the pipelines V, T, S, and U. Sequences like V, T, S, U, which define the lifetime of a pipeline breaker, are always sub-sequences of the original PEO. Further, the precedence relationship V < U, which describes that V is consumed by U, specifies the span of the subsequence of the original PEO. Given a PEO $O = {}^1p_1, \ldots, p_i, \ldots, p_j, \ldots, p_k, \ldots, p_n{}^o$, the corresponding pipeline execution cost vector $\mathbf{e} = {}^1e_1, \ldots, e_j, \ldots, e_n{}^o$, and the precedence relationship $p_i \ \ddot{Y} \ p_k$, we can calculate the lifetime l_i of the pipeline breaker b_i of pipeline p_i by:

$$l_i = \sum_{j=i}^{k} e_j \tag{3}$$

Based on Equation 3, we define the pipeline lifetime vector $\mathbf{l}$ for a PEO $O = {}^1p_1, \ldots, p_i, \ldots, p_n{}^o$ as $\mathbf{l} = {}^1l_1, \ldots, l_i, \ldots, l_n{}^o$.

3.2.4 Memory Integral Metric. Based on the presented building blocks, we can define our memory integral metric. We multiply the vector of pipeline breaker memory costs $\mathbf{m}$ with the transposed

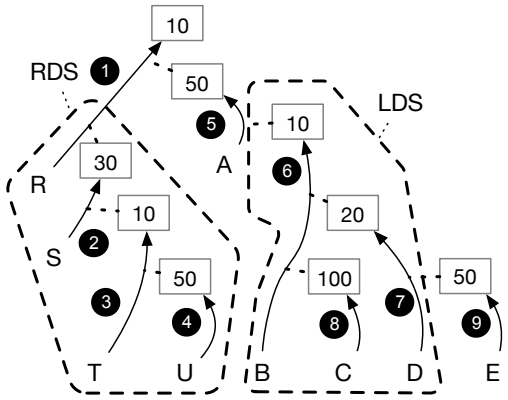

Figure 6: A pipeline execution plan with two deep trees.

vector of pipeline breaker lifetimes $\vec{l}$, such that the memory integral is computed by:

$$MI = \vec{m} \cdot \vec{l} \tag{4}$$

Consequently, the memory integral is the sum over all pipeline breakers b_i multiplying their memory size m_i with their lifetime l_i. The memory consumption of the pipeline breakers is 0 before they are created, m_i as soon as the respective pipeline starts operating until b_i is destructed, and 0 afterwards again.

3.3 PEO Search Space

After presenting the cost model, we shift the focus towards the PEO search space. We use Activity-on-Node Trees, as illustrated on the right-hand-side of Figure 2, to systematically derive one or more PEOs. AoN-Trees are similar to Activity-on-Node Graphs [6].

Definition 2 (Activity-on-Node Tree). *An Activity-on-Node Tree (AoN-Tree) is a directed acyclic graph of nodes reflecting individual pipelines. A node P_i does have the children $P_k, \ldots, P_n$, if the pipeline p_i consumes the pipelines $p_k, \ldots, p_n$. Each node P_i holds the estimated lifetime cost l_i of the pipeline p_i, and the estimated memory size m_i of the result in the pipeline breaker b_i.*

The size of the search space of PEOs depends on the structure of the AoN-Tree. We can summarize the following patterns:

Right-deep structures: If individual pipelines are dependent on exactly a single intermediate, there is exactly a single PEO possible; any search for alternatives can thus be abandoned. For example, only considering the plan fragment denoted as "RDS" in Figure 6, (1) depends on (2), (2) depends on (3), and (3) depends on (4); thus, the logical dependencies prescribe the physical execution of the pipelines.

Left-deep structures: If there is a plan consisting of n pipelines that exhibits left-deep structures for probe sides, i.e., there is a single continuous pipeline from one table to the result, with multiple build sides to be probed against, the total number of PEOs is $1^n \quad 1^0!$. If all build-side pipelines have a length of 1, the best execution order is determined by successively choosing the pipelines by their smallest pipeline breakers. As soon as the constraint on the pipeline length is relaxed, the tree needs to be exhaustively searched for the best order. For example, if pipeline (9) would not be existent, the plan fragment denoted as "LDS" in Figure 6 would provide a corresponding scenario: pipeline (6) would probe against (8) and (7) resulting in two alternative PEOs: (8, 7, 6) and (7, 8, 6).

Bushy structures: In general, we consider bushy execution plans, which fall between the two extremes of left and right deep structures. It is worth mentioning that the problem of identifying a memory optimal PEO is non-linear in nature, i.e. the size of an intermediate at the root of an execution fragment is not enough to decide on the optimal ordering of the consuming pipeline, thus requiring proper search algorithms.

3.4 Exhaustive Search Algorithm

"Exhaustive Search" (ES) is the reference algorithm which we created for finding the optimal PEO according to estimated cardinalities and cost. The algorithm implements an optimization to calculate the memory cost, i.e., the memory integral, on the fly while traversing the AoN-Tree in a deepest-first-search fashion through recursion. We further explain the calculation of the memory peak value. Each node in the AoN-Tree represents a pipeline, having a lifetime and a memory size of the corresponding pipeline breaker. Initially, we invoke our recursive algorithm with the nodes that are directly reachable from the start node in the AoN-Tree as heap of *active nodes*. From the heap of active nodes, the algorithm continuously takes a node, removes the node from the copied *active nodes* heap, adds the node's memory size to the current memory peak, and calculates the new memory integral using the lifetime of the node. Whenever another node becomes directly reachable in the AoN-Tree, we append the node to our active nodes. The processed node is appended to the heap of *visited nodes* and we enter a recursion step using the new active nodes. If all nodes of the tree have been processed, the recursion returns the PEO, its memory integral, and memory peak value in an array of PEOs. For the evaluation we skipped collecting all PEOs and instead logged the currently best and worst PEO discovered during the search.

Algorithm 1 defines the recursive procedure. Each node has a successor node `next`, a count of unvisited previous nodes `prevCount`, an array of previous nodes `prevs`, memory size `memorySize` and execution costs `lifetime`. The start node has an array of successor nodes `nextNodes`. The recursion starts with the start node's next nodes as the active nodes (`activeNodes`) and logs the start node in the array of visited nodes (`visitedNodes`). In Line 4 of Algorithm 1, we start the iteration of all nodes among our active nodes. We remove the chosen node from the latter and calculate the memory peak as well as the current memory integral. If the chosen node has previous nodes, we remove the memory sizes of those from the memory peak in Line 9 and 10, since the memory of the pipeline breakers belonging to those nodes is freed. Lines 16 to 24 contain the mechanism to determine whether all previous nodes of a specific successor node have been visited with Line 17 appending the successor to the active nodes. Line 24 then again increments the `prevCount` for the currently visited node. The function as a whole returns an array of PEOs. In Line 21 and 23 each iteration adds the returned PEOs from each recursion. Those collected PEOs are then

Algorithm 1 Finds all possible PEOs for a given AoN-Tree. The algorithm is recursive and needs to be initiated using ES(Start.nextNodes, [Start], 0, 0), where Start is the start node of the AoN-Graph. It returns an array of all possible PEOs.

```
1:  function ES(activeNodes, visitedNodes, memPeak, MI)
2:      peos = []
3:      // This line can accommodate the Theta Skip Step
4:      for n in activeNodes do
5:          newNodes = activeNodes.copy()
6:          newNodes.remove(n)
7:          nextPeak = memPeak + n.memorySize
8:          nextMI = MI + nextPeak * n.lifetime
9:          for i in n.prevs do
10:             nextPeak -= i.memorySize
11:         end for
12:         if n.isEnd() then
13:             return [PEO(visitedNodes + [n], nextMI, nextPeak)]
14:         end if
15:         // This line can accommodate the Branch Pruning Step
16:         n.next.prevCount =- 1
17:         if n.next.prevCount == 0 then
18:             newNodes.append(n.next)
19:         end if
20:         newVisitedNodes = visitedNodes + [n]
21:         rec = ES(newNodes, newVisitedNodes, nextPeak,
22:             nextMI)
23:         peos += rec
24:         node.next.prevCount =+ 1
25:     end for
26:     return traversals
27: end function
```

returned to the caller in Line 26. In the end, the initial call receives all possible PEO, which can then be filtered for the PEO with the smallest memory integral.

3.5 Branch Pruning Search Algorithm

Algorithm 2 These lines can be inserted in Algorithm 1 to obtain the Branch Pruning Algorithm. The heuristic memory integral is obtained from the Longest Path Heuristic, for instance.

```
heuristicMI = heuristic.getMemoryIntegral()
if nextMemIntegral > heuristicMI then
    return []
end if
```

Based on the exhaustive search algorithm, our **"Branch Pruning Search" (BPS)** returns an empty array of PEOs as soon as the current memory integral steps over a threshold, i.e., a memory integral calculated by a heuristic from Section 4.2, thus indicating that inside the recursion no better PEO had been found. If the array of PEOs is empty, it will return the PEO produced by the heuristic. In the evaluation, we use branch pruning with the memory integral produced by the PEO of the longest path heuristic as a baseline. The branch pruning step can be accommodated in Line 15 of Algorithm 1.

4 HEURISTIC SCHEDULING ALGORITHMS

The algorithms presented up until this point guarantee to find the optimal PEO with respect to estimated cost. In this section, we explore two further algortihms, which do not guarantee to find the optimal PEO with respect to estimated cost, but near-optimal PEOs. First, we introduce the Theta Skip Search algorithm, which is a tradeoff between a smaller number of enumerated PEOs, and potentially higher estimated memory cost of the selected PEO. The second algorithm is the Longest Path Heuristic, which found PEOs with near-optimal memory cost in our experimental evaluation.

4.1 Theta Skip Search Algorithm

Algorithm 3 These lines can be inserted in Algorithm 1 to obtain the Theta Skip Search algorithm. Note that the recursion also needs to accommodate these lines.

```
1:  currentSmallest = 0
2:  threshold = 1 kB // or other size
3:  smallestNode = Null
4:  for n in activeNodes do
5:      if n.memorySize < threshold then
6:          currentSmallest = n.memorySize
7:          smallestNode = n
8:      end if
9:  end for
10: if smallestNode != Null then
11:     memPeak += smallestNode.memorySize
12:     MI += memPeak * smallestNode.lifetime
13:     for n in smallestNode.prevNodes do
14:         memPeak -= n.memorySize
15:     end for
16:     smallestNode.next.prevCount -= 1
17:     if smallestNode.next.prevCount == 0 then
18:         activeNodes.append(node.next)
19:     end if
20:     rec = ES(activeNodes\{smallestNode},
21:         visitedNode + smallestNode, memPeak, MI)
22:     smallestNode.next.prevCount += 1
23:     return rec
24: end if
```

In some QEPs, we may encounter many pipeline breakers with a small memory size. Even if executing them eagerly might cause the PEO to become suboptimal, the difference to a similar better PEO might be negligible, while we save a possible recursion step for each skipped node. Thus, the **"Theta Skip Search" (TSS)** algorithm always chooses nodes with small memory sizes below a threshold (for example 1KByte) and if many of such are found, chooses the smallest one directly. We also equipped this algorithm with the branch-pruning step. Algorithm 3 shows the necessary pseudo code, which needs to be integrated in Line 3 of Algorithm 1. Lines 3 to 9 search for the node with the smallest memory footprint which also lies below the memory size threshold. Then it proceeds to

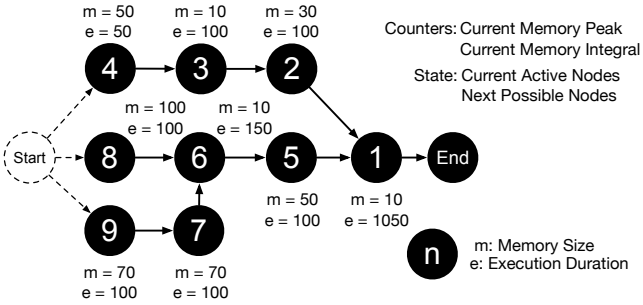

Figure 7: The AoN-Tree corresponding to Figure 6. For each node, m denotes the memory size of each breaker node, and e denotes the execution cost.

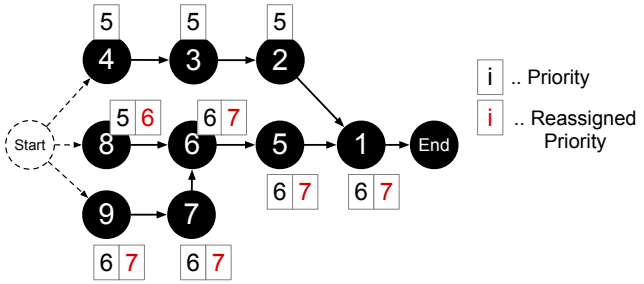

Figure 8: We assign distinct priorities to the nodes of Figure 7 according to the hop-length from the nodes outgoing from the start node to the end node for each node along the path.

calculate the memory peak and the memory integral accordingly. It updates the `prevCount` as usual and then enters the recursion. This algorithm is the only considered algorithm which is neither producing only one PEO nor optimal but practically feasible, since it reduces the branching complexity.

4.2 Longest Path Heuristic

Figure 6 provides a more complex scenario with 9 base tables and 8 hash tables. The right side (with an arrow) represents the build side and thus the result of the input pipeline. The left side represents the probe side and thus the dependency on a specific intermediate. The corresponding AoN-Tree is shown in Figure 7. Pipeline (1) originating out of table R and probing into the hash tables built by (2) and (3) is the root node of the AoN-Tree with (2) and (3) as its child nodes. For the ease of understanding the algorithms presented in Subsection 3 and Section 4, the tree also contains "Start" and "End" nodes, which serve as starting/end points of the algorithms, and do not incur any memory or execution costs.

The heuristic **"Longest Path Heuristic" (LPH)** is based on the observation that deepest first execution of interdependent pipelines will yield near-optimal PEOs. For example, Figure 6 shows that while interleaving pipeline executions of different subtrees will yield the optimal result, the execution of one subtree after another in order of their length from the starting node to the result node is "close" to the optimal PEO. The corresponding algorithm works as follows: for each node within the subtree, we assign a priority to the node which corresponds to the longest node-path to which a node belongs. In Figure 7, take the path from (9) to (End): the path length is 6, since we have 6 nodes along the way. Each node on this path is assigned with the priority 6. On the path from (8) to (End), each node should be assigned with 5. In this case, only (8) will be assigned with priority 5, since the other nodes already have a longer path they belong to. The PEO-constructing algorithm will then keep a heap of available nodes for traversal, and always choose the one with the highest priority.

The priority assignment is shown in Algorithm 4. The number of hops from one of the starter's nodes to the end node (that is, how many nodes have to be passed along a path) shall be equivalent to the priority by which the path and all its nodes should be executed. Since a single node may belong to multiple paths, its priority must

Algorithm 4 Assign priorities to all nodes according to the number of hops from the start node's next nodes to the end node. Higher priorities on the path override lower priorities.

```
1:  function AssignPrioritiesForLPH(startNode, endNode)
2:      firstNodes = startNode.nextNodes
3:      List<Node,int> nodeToPriority
4:      for node in firstNodes do
5:          length = getPathLength(node, endNode)
6:          nodeToPriority.insert(node, length)
7:      end for
8:      sortByPriority(nodeToPriority)
9:      offsetCounter = 0
10:     runningPriority = 0
11:     for nodeAndPriority in nodeToPriority do
12:         if runningPriority == nodeAndPriority.priority then
13:             offsetCounter++
14:         else
15:             runningPriority = nodeAndPriority.priority
16:         end if
17:         nodeAndPriority.priority += offsetCounter
18:     end for
19:     for np in nodeToPriority do
20:         currentNode = np.node
21:         while currentNode != endNode do
22:             np.node.priority = np.priority
23:             currentNode = currentNode.next
24:         end while
25:     end for
26: end function
```

be dominated by the priority of the most "important" path it is associated with. In the algorithm we facilitate this by first examining the distance from the starter node's next nodes to the end nodes in Lines 3 - 7, and label the starter's next nodes (and only those) accordingly. We then sort the nodes from lowest to highest priority in Line 8 to be able to override node priorities from lowest to highest without checks during node labeling. To avoid having same priorities on paths with the same hop-distance from start to end, we need to introduce a running offset, which will be used to make same-distance paths have a distinct priority. This avoids "upgrading" lower priority paths to match priorities of longer distance paths.

For all the labelled nodes (Line 11), we first check if the running priority matches the next node's priority, and if so, we increment the offset counter; if not, we set the current running priority to the priority of the next node. Line 17 then re-assigns the priority label for the current node by the running offset. We do not increment by "1", since this would result in duplicate priorities. We finally set the priorities of all nodes in Lines 19 till 25 from lowest to highest. This algorithm still has a non-deterministic element, which is the sorting in Line 8. Nodes with same length then might or might not be upgraded effectively.

Algorithm 5 Returns a PEO according to assigned priorities in descending order.

```
1:  function getPEOForPriorities(startNode, endNode)
2:      possibleNodes = start.nextNodes // array of nodes
3:      nodeOrder = [startNode]
4:      startNode.startTime = 0
5:      currentTime = 0
6:      while possibleNodes != [endNode] do
7:          highestPriority = 0
8:          nextNode = null
9:          for n in possibleNodes do
10:             if n.priority > highestPriority then
11:                 highestPriority = n.priority
12:                 nextNode = n
13:             end if
14:         end for
15:         nextNode.startTime = currentTime
16:         currentTime += nextNode.lifetime
17:         for prevNode in nextNodes.prevs do
18:             prevNode.endTime = currentTime
19:         end for
20:         nodeOrder.append(nextNode)
21:         nextNode.next.prevCount -= 1
22:         if nextNode.next.prevCount == 0 then
23:             possibleNodes.append(nextNode.next)
24:         end if
25:         possibleNodes.remove(nextNode)
26:     end while
27:     endNode.endTime = currentTime
28:     nodeOrder.append(endNode)
29:     MI = 0
30:     for n in nodeOrder do
31:         MI += (n.endTime - n.startTime) * n.memorySize
32:     end for
33:     return PEO(nodeOrder, MI)
34: end function
```

Algorithm 5 lists the mechanism of the priority execution. Given the start node `startNode` and the end node `endNode` of the AoN-Tree, where each node in the tree has an assigned `priority`, and the start and end times are initialized to 0. The first possible nodes are obtained from the start node in Line 2. Iteratively, the algorithm chooses the node with the highest available priority in Lines 9 to 14. It then calculates the start and end time stamps of the node, and annotates the nodes using those in Lines 15 and 18. In Lines 21 to 24 it determines which nodes can be appended to the current `possibleNodes`, and adds them in that case. In Lines 29 to 32, the memory integral `MI` is calculated using the start and end times, and returns the obtained order of nodes as the PEO.

5 EXPERIMENTAL EVALUATION

As mentioned within the motivation, the experimental investigation aims at three aspects of the problem of memory-efficient ordering of query execution pipelines. First, we confirm the motivation of the work by showing the predicted memory integral improvements with respect to different PEOs of individual queries. Second, we show that there is a correlation between the predicted memory integral and the measured memory integral in execution. Third, we aim at confirming the hypothesis that different PEOs (with potentially different memory profiles) show the same runtime performance, i.e., there is no performance degradation when executing more memory-efficient PEOs. Fourth, we demonstrate the efficiency of the algorithms and provide numbers to quantify the minimal overhead during planning time according to the different types of algorithms.

5.1 Experimental Setting

We implemented all algorithms within the database engine prototype used by previous work [17, 18], which implements pipelined query execution [1]. The PEO search algorithm implementations are single-threaded. The entire system is compiled with GCC 7.5.0 using option -O3. We measured the memory utilized by a pipeline breaker by multiplying the number of rows by the size of the tuples contained within the pipeline breaker in bytes. Execution time is measured by recording the time before and after the execution of a pipeline. The cardinalities for optimization of the PEO were taken after execution of the respective query. The query was re-executed for ten iterations.

For the evaluation we chose to evaluate all queries of the Join Order Benchmark [10] and removed all non-join query operators, because we (a) focus on pipeline breakers induced by hash joins and (b) the prototype only supports joins but not all other necessary operators. Due to the removal of such operators, we were only able to execute queries 1-14, 20, 26, 27, 32, 33. As the cost model for enumeration and prediction we chose C_{out} as the basic cost function that we use to compute Equation 2, with $\omega = 0$ since we add no writing costs. We also disregard final breakers as consumers of memory, since they only add a constant to the memory integral that does not depend on the PEO.

The experiments were executed on a 2-socket NUMA system using two 20-core Intel Xeon Platinum 8260 CPUs clocked at 2.40GHz. The system was equipped with 160GByte of RAM. The operating system was equipped with a Linux 5.11.11 kernel. We executed each query at least once before starting with the actual measurements. To minimize external effects (like caching), we sequentially executed the set of queries clustered by the individual search algorithms.

5.2 Minimizing Memory Integrals

Let us first examine the predicted improvements comparing the PEOs for the different algorithms we introduced. For queries 6,

[1] Finalist in the 2018 ACM SIGMOD Programming Contest

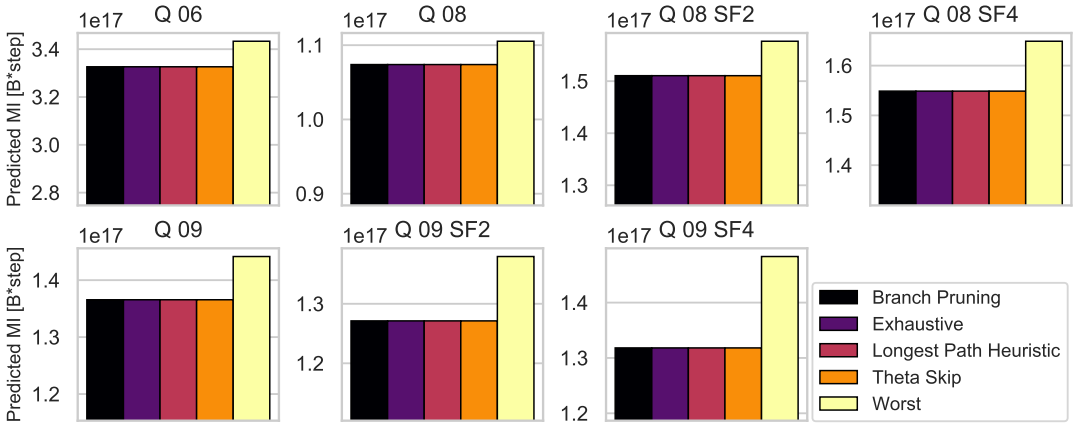

Figure 9: The predicted memory integral for each of the algorithms is shown for the individual queries. SF* indicates the scale factor with which the "aka_name" table is scaled. Note the baseline MI for each diagram, lower is better.

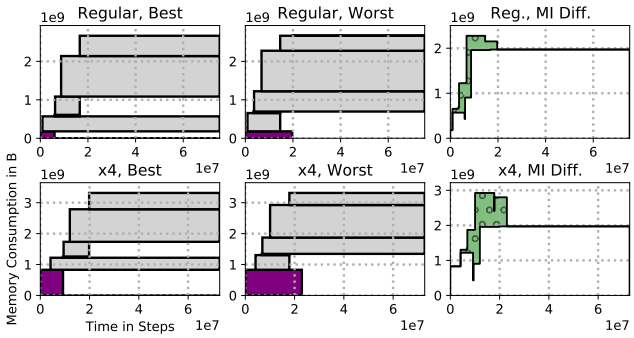

Figure 10: Gantt diagrams showing the memory integral areas over time for query 9 of the JOB. In purple, the pipeline originating from the base table "aka_name" is indicated.

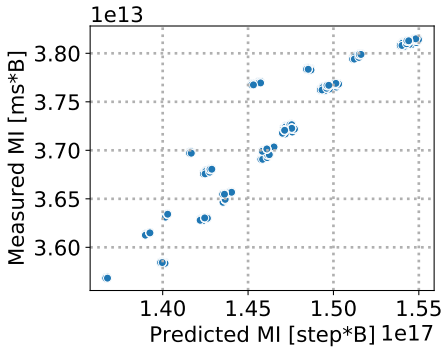

Figure 11: Each dot in this graph represents a PEO and their respective predicted as well as their measured memory integral.

8, and 9 of the Join Order Benchmark we were able to observe a difference between the best and worst possible memory integral. Figure 9 shows the resulting memory integral (MI) improvements, where the worst PEO serves as the relative baseline. We also introduce two skewed data sets: for the first one, "SF2", we scaled the "aka_name" table to 2.075 million tuples, where we do not introduce any join partners to any join, and for "SF4" to 4.15 million tuples the same accordingly. We did this to demonstrate that for highly skewed data sets, the improvements might be more substantial. As Figure 9 shows, we were able to observe improvements of 3.1% for query 6 with the worst PEO as the baseline, where the non-optimal algorithms also found the best possible PEO. For the other considered queries 8 and 9, this was not the case for neither data set. For the unscaled data set, query 8 yielded an MI improvement of 2.8%, where the optimal PEO was less than 0.001% better than the PEO found by the heuristic. The results are similar for the other query-data-set combinations. For query 9 and the SF4 data set, we were able to observe improvements in MI of 11.1%.

5.3 Correlation between Predicted and Measured Memory Integral

To examine the correlation between the predicted memory integral and the measured memory integral, we measured the pipeline execution times of the individual pipelines for 20 iterations. After doing so, we took the average of all pipeline execution times and calculated the memory integrals for each possible PEO for a given query using the averages. This is plausible, since pipeline execution times do not depend on the PEO and thus should stay constant. In Figure 11 the predicted and measured memory integrals are plotted of all 840 PEOs for query 9 of the Join Order Benchmark with the "aka_name" column scaled to 4.15 million tuples and the "company_name" scaled to 240 thousand tuples. Each dot represents a PEO with the predicted and the measured memory integral. The graph indicates that there is a linear correlation between the predicted and the measured memory integral.

Algo	Query	06	08	09	20	26	33
ES		63µs	357µs	1881µs	342µs	11ms	302ms
LPH		49µs	71µs	81µs	152µs	194µs	348µs
TSS		63µs	366µs	1560µs	217µs	449µs	148ms
BPS		59µs	196µs	679µs	245µs	5065µs	67ms

Figure 12: Optimization times for the indicated queries

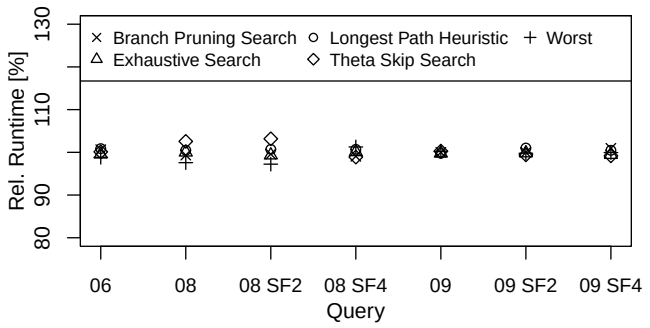

Figure 13: The execution times of different queries including optimization time is shown with different pipeline execution orders provided by the indicated algorithms. The average execution time of any run for one query is taken as a baseline.

5.4 Performance Stability

For observing the performance stability, we evaluated the optimization times of query 6, 8, 9, 20, 26, and 33 for all search algorithms. The table in Figure 12 shows the respective optimization times. Except for query 6 and 8, the exhaustive algorithm incurs the largest optimization times, and for query 33 exhibiting 302ms of optimization time due to 60480 total enumerations of PEOs, while the actual execution of the query is only 11.057s. This is an outlier in terms of optimization time relative to overall execution costs, while also not providing significant memory integral improvements. TSS as our only non-optimal non-heuristic only outperforms BPS for queries 20 and 26, which might be caused by the early skipping of nodes with small memory costs. In the other cases, BPS was also faster, and for query 33 it outperformed TSS by 81ms. Except for query 8 and 9 the LPH found close-to or optimal PEOs, while always outperforming the other algorithms in terms of runtime, and for query 33 only incurring optimization times of just 348µs.

5.5 Overhead for identifying PEOs

We now demonstrate the absence of an impact of the optimization on query execution in terms of a) negligible optimization costs and b) absence of impact of the executed PEO on overall query execution times. The execution times for the strategies provided by each algorithm are plotted in Figure 13. With query 33 being an extreme outlier, the optimization had an impact of less than 0.006% of the overall execution time in the worst case. Furthermore, the data empirically shows that the choice of ordering of pipelines has no impact on the execution speed, being masked by other effects.

6 RELATED WORK

While pipeline-based query execution reflects the state-of-the-art in modern database engines, the impact of orderings with respect to memory consumption for intermediates holding the materialized pipeline breakers is not yet investigated in depth. Nevertheless, this research work is tapping into the rich bouquet of work in the context of query optimization. For example, [15] is mainly concerned with the question of "how can we organize query processing such that the data can be kept in CPU registers as long as possible?" The order of pipelines is not of interest. Scheduling of individual work units (e.g., Morsels [9]) in the context of multi-core systems is also discussed, e.g., [11, 16]; compared to execution pipelines, morsels are significantly more fine granular and thus the optimization aims to exploit existing hardware capabilities but not to reduce the overall memory consumption. Finally, the work also touches upon query optimization beyond performance. Related in the wider sense is the research field for robust query optimization [13, 17, 20], which again may subsume our goal of optimizing pipeline execution orders without compromising performance as one of multiple facets. Performance simulations and query optimization or query runtime estimation [12, 19] is also related but may be considered orthogonal, since our reordering approach does not influence the overall query execution time. From an algorithmic perspective, the problem of ordering pipelines with dependencies is also addressed in the context of operations research; [5, 6] provides an in-depth overview of a magnitude of optimization algorithms, which also served as a blueprint to devise search algorithms outlines in Subsection 3.3 and Section 4. The authors of Chain [3] take a similar approach to our work by examining memory consumption for incoming streams of tuples and keeping it below the available main memory bounds of the system. In their model, they also handle operators via pipelines, but those pipelines do not handle tuples in a tuple-at-a-time fashion, but instead write processed tuples into intermediates after each operator. Thus, they identify different algorithmic strategies to process incoming streams of tuples in a memory efficient way across several operator stages, and propose a new algorithm, called Chain, which is optimal according to the goal of always keeping the memory consumption as low as possible at a given time.

7 CONCLUSION

In this paper, we are motivating the case of scheduling the execution of query pipelines in order to optimize for memory consumption. Using an illustrative example, we showcased the potential of a "good" ordering of execution pipelines in query execution graphs. We discussed different optimization goals, provided a framework to formally describe the search space, and devised four different algorithms to identify PEOs for a given query. Our proposed heuristics incorporated into the exhaustive algorithm provide early non-candidate pruning, thus presenting themselves as promising algorithmic approaches. Within the experimental section, we provided insights into our findings based on an implementation of the algorithms in the context of a pipelined execution engine using the Join-Order-Benchmark. We reported detailed facts on a sample of queries with respect to potential savings of the memory integral, showed the stability of the performance even for memory-efficient

PEOs, and quantified the overhead of the search algorithms (which proved to be negligible even for large and complex queries).

Optimizing for memory consumption is not only beneficial for increasing the overall memory utilization in cloud settings, it is also helpful to avoid/delay out-of-memory situations and increase the concurrency of existing systems. Since we consider it extremely relevant to give more weight to non-performance aspects in query optimization and query execution, we hope to have contributed to this line of research and opened up a research playground for many other related aspects on this topic.

ACKNOWLEDGEMENT

This work was partly funded by the German Research Foundation (DFG) via a Reinhart Koselleck-Project (LE-1416/28-1).

REFERENCES

[1] [n.d.]. Cost/Performance in Modern Data Stores: How Data Caching Systems Succeed. https://www.microsoft.com/en-us/research/uploads/prod/2018/07/CostPerformance-in-Modern-Data-Stores-How-Data-Cashing-Systems-Succeed-slides.pdf. Accessed: 2021-12-14.

[2] [n.d.]. Public cloud services annual growth rate worldwide from 2020 to 2022, by segment. https://www.statista.com/statistics/258718/market-growth-forecast-of-public-it-cloud-services-worldwide/. Accessed: 2021-12-14.

[3] Brian Babcock, Shivnath Babu, Rajeev Motwani, and Mayur Datar. [n.d.]. Chain: Operator Scheduling for Memory Minimization in Data Stream Systems. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data* (San Diego, California). ACM, New York, NY, USA, 253–264. https://doi.org/10.1145/872757.872789

[4] Peter A. Boncz, Marcin Zukowski, and Niels Nes. 2005. MonetDB/X100: Hyper-Pipelining Query Execution. In *CIDR*.

[5] Peter Brucker. 2007. *Scheduling algorithms* (5th ed.). Springer Publishing Company, Incorporated.

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

[7] Sophie Cluet and Guido Moerkotte. 1995. On the Complexity of Generating Optimal Left-Deep Processing Trees with Cross Products. 54–67.

[8] 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

[9] Viktor Leis, Peter A. Boncz, Alfons Kemper, and Thomas Neumann. 2014. Morsel-driven parallelism: a NUMA-aware query evaluation framework for the many-core age. In *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*, Curtis E. Dyreson, Feifei Li, and M. Tamer Özsu (Eds.). ACM, 743–754. https://doi.org/10.1145/2588555.2610507

[10] Viktor Leis, Bernhard Radke, Andrey Gubichev, Atanas Mirchev, Peter A. Boncz, Alfons Kemper, and Thomas Neumann. 2018. Query optimization through the looking glass, and what we found running the Join Order Benchmark. *VLDB J.* 27, 5 (2018), 643–668. https://doi.org/10.1007/s00778-017-0480-7

[11] David Lo, Liqun Cheng, Rama Govindaraju, Parthasarathy Ranganathan, and Christos Kozyrakis. 2015. Heracles: improving resource efficiency at scale. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture, Portland, OR, USA, June 13-17, 2015*, Deborah T. Marr and David H. Albonesi (Eds.). ACM, 450–462. https://doi.org/10.1145/2749469.2749475

[12] Ryan C. Marcus and Olga Papaemmanouil. 2019. Plan-Structured Deep Neural Network Models for Query Performance Prediction. *Proc. VLDB Endow.* 12, 11 (2019), 1733–1746. https://doi.org/10.14778/3342263.3342646

[13] Volker Markl, Vijayshankar Raman, David E. Simmen, Guy M. Lohman, and Hamid Pirahesh. 2004. Robust Query Processing through Progressive Optimization. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Paris, France, June 13-18, 2004*. ACM, 659–670. https://doi.org/10.1145/1007568.1007642

[14] 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)*, Bernhard Mitschang, Daniela Nicklas, Frank Leymann, Harald Schöning, Melanie Herschel, Jens Teubner, Theo Härder, Oliver Kopp, and Matthias Wieland (Eds.). Gesellschaft für Informatik, Bonn, 545–546.

[15] Thomas Neumann. 2011. Efficiently Compiling Efficient Query Plans for Modern Hardware. *Proc. VLDB Endow.* 4, 9 (2011), 539–550. https://doi.org/10.14778/2002938.2002940

[16] Aunn Raza, Periklis Chrysogelos, Angelos-Christos G. Anadiotis, and Anastasia Ailamaki. [n.d.]. Adaptive HTAP through Elastic Resource Scheduling. In *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA]*. ACM, 2043–2054. https://doi.org/10.1145/3318464.3389783

[17] Florian Wolf, Michael Brendle, Norman May, Paul R. Willems, Kai-Uwe Sattler, and Michael Grossniklaus. 2018. Robustness Metrics for Relational Query Execution Plans. *Proc. VLDB Endow.* 11, 11 (July 2018), 1360–1372. https://doi.org/10.14778/3236187.3236191

[18] Florian Wolf, Norman May, Paul R. Willems, and Kai-Uwe Sattler. 2018. On the Calculation of Optimality Ranges for Relational Query Execution Plans. In *Proceedings of the 2018 International Conference on Management of Data* (Houston, TX, USA) *(SIGMOD '18)*. Association for Computing Machinery, New York, NY, USA, 663–675. https://doi.org/10.1145/3183713.3183742

[19] Wentao Wu, Yun Chi, Hakan Hacigümüs, and Jeffrey F. Naughton. 2013. Towards Predicting Query Execution Time for Concurrent and Dynamic Database Workloads. *Proc. VLDB Endow.* 6, 10 (2013), 925–936. https://doi.org/10.14778/2536206.2536219

[20] Shaoyi Yin, Abdelkader Hameurlain, and Franck Morvan. 2015. Robust Query Optimization Methods With Respect to Estimation Errors: A Survey. *SIGMOD Rec.* 44, 3 (2015), 25–36. https://doi.org/10.1145/2854006.2854012