How to Optimize SQL Queries? A Comparison Between Split, Holistic, and Hybrid Approaches

Luca Gretscher
Saarland University, Saarland Informatics Campus
Saarbrücken, Germany
luca.gretscher@bigdata.uni-saarland.de

Jens Dittrich
Saarland University, Saarland Informatics Campus
Saarbrücken, Germany
jens.dittrich@bigdata.uni-saarland.de

ABSTRACT

Relational database systems internally construct a physical query execution plan (QEP) that specifies exactly how to compute a desired result. However, choosing a QEP involves determining a specific join order, deciding how to access base relations, specifying concrete physical implementations to compute the algebraic operations defined by the given SQL query, and much more. In general, choosing the optimal QEP w.r.t. a predefined cost model is a hard optimization task, referred to as query optimization problem (QOP), that requires super-exponential time in the worst-case.

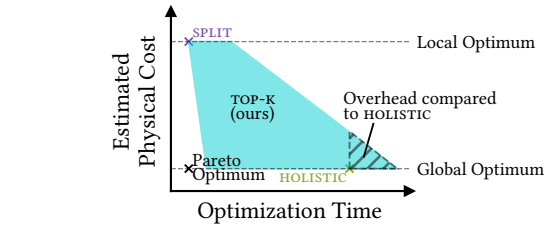
Even though query optimization is a fundamental problem that has been studied for decades now, related work often focuses only on a specific subtask like join ordering. Furthermore, by inspecting open-source database systems, fundamentally different query optimization strategies can be observed. These strategies exhibit vastly different optimization times while having a major impact on the resulting QEP qualities. In this work, we revisit two conceptually different approaches to solve query optimization, namely SPLIT and HOLISTIC. We discuss their advantages and disadvantages and present a detailed experimental evaluation in our research database system mutable. Additionally, we propose a hybrid strategy called TOP-K that is able to rediscover the holistically optimal QEPs while being significantly closer to the optimization time of SPLIT.

PVLDB Reference Format:
Luca Gretscher and Jens Dittrich. How to Optimize SQL Queries? A Comparison Between Split, Holistic, and Hybrid Approaches. PVLDB, 18(11): 3910 - 3922, 2025.
doi:10.14778/3749646.3749663

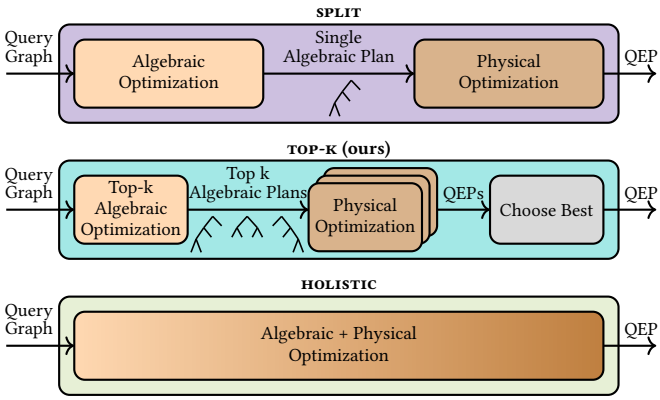
PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/BigDataAnalyticsGroup/mutable-QO-approaches/tree/submission.

1 INTRODUCTION

Conceptually, the QOP can be described as two-phase optimization problem [18]. Firstly, a given query graph[1] is transformed into a tree-structured algebraic plan inducing a partial order on the joins. Therefore, this step is referred to as join ordering or *algebraic optimization*. Secondly, a physical operator tree is constructed implementing the given algebraic plan but determining concrete physical implementations and specifying a total order on all operands[2]. This step is called *physical optimization*.



(a) Solution space w.r.t. QOP.



(b) Conceptual description.

Figure 1: Overview of the different approaches to solve QOP.

However, related work as well as concrete database systems may implement query optimization independent from this conceptual description. While DuckDB has an internal interface that clearly separates algebraic from physical optimization [6], other open-source database systems like PostgreSQL, MySQL, or SQLite unify both optimizations steps to a certain degree by applying the idea of interesting properties [24, 25, 32, 34] pioneered by System R [30]. Therefore, we classify implementations to solve the whole query optimization problem into two categories.

(1) **SPLIT:** Solve algebraic and physical optimization independently one after another.
(2) **HOLISTIC:** Incorporate both algebraic and physical optimization into a single optimization task.

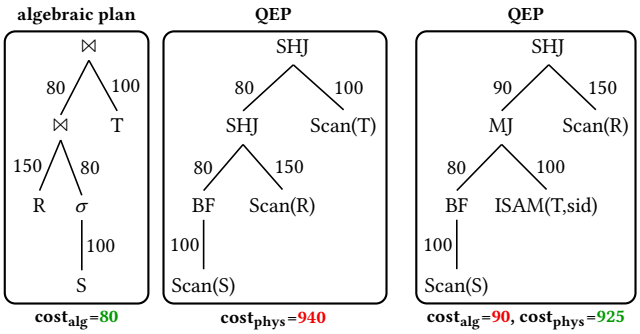
Proceedings of the VLDB Endowment, Vol. 18, No. 11 ISSN 2150-8097.
doi:10.14778/3749646.3749663

[1] We define a query graph as an undirected join graph with vertices for each relation and edges according to the join predicates, extended by non-join operators like filter, grouping, and projection. It purely represents the semantics of an SQL query.

[2] Total order refers to a determined sibling order. Determining the execution order of independent pipelines is out of the scope of this paper and subject of related work [14].

```
SELECT *
FROM R, S, T
WHERE S.val < 42 AND R.id = S.rid AND S.id = T.sid;
```

(a) SQL query inducing multiple join orders.



(b) Applying SPLIT. **(c) Applying HOLISTIC.**

Figure 2: Running example. Edges of the plans are annotated with the respective cardinalities.

Figure 1a gives an overview of the advantages and disadvantages of choosing the problem granularity to be either SPLIT or HOLISTIC. Clearly, there is a trade-off between spending more time for optimization to reduce the estimated physical cost and thus hopefully also the running time for the resulting globally optimal QEP and executing a faster optimized but potentially less efficient locally optimal QEP. Therefore, we also propose a hybrid strategy TOP-K trying to reach the Pareto optimum by not only considering a single algebraic plan for physical optimization but the top k algebraic plans. However, as TOP-K involves configuring the hyperparameter k, it spans an entire space between SPLIT and HOLISTIC depicted in turquoise. Furthermore, misconfiguration of k may lead to an overhead in the optimization time while not improving the QEP quality compared to HOLISTIC. All approaches are depicted conceptually in Figure 1b.

1.1 Running Example

Consider the example SQL query in Figure 2a inducing two join orders, i.e., (R ⋈ σ(S)) ⋈ T and R ⋈ (σ(S) ⋈ T). We assume that the base relation S is already sorted on its primary key S.id and that an index on the attribute T.sid is available.

SPLIT. The approach SPLIT first determines a specific join order under a given algebraic cost model. We apply the commonly used algebraic cost model C_{out} [8, 18] that computes the sum of all intermediate result cardinalities. In our example, the resulting algebraic plan is depicted on the left in Figure 2b. Afterward, the separate physical optimization step receives this single algebraic plan as input and determines physical operators for both the accesses to base relations as well as the algebraic operators, i.e., the filter and the two joins in our example. Furthermore, a total order for sibling nodes, which were not yet ordered in the algebraic plan, is introduced. We assume two physical operators implementing the access to base relations — the scan and the indexed sequential access method (ISAM) — and four physical operators implementing the algebraic filter and join operator — the branching filter (BF), the simple hash join (SHJ), the merge join (MJ), and the nested-loops join (NLJ) — with the following physical cost models[3].

$$C_{\text{Scan}}(x) := |x| \tag{1}$$
$$C_{\text{ISAM}}(x, attr) := 1.1 * |x| \tag{2}$$
$$C_{\text{BF}}(x) := |x| \tag{3}$$
$$C_{\text{SHJ}}(x, y) := 1.5 * |x| + |y| \tag{4}$$
$$C_{\text{MJ}}(x, y) := |x| + |y| \tag{5}$$
$$C_{\text{NLJ}}(x, y) := |x| * |y| \tag{6}$$

The ISAM provides the accessed data sorted on the specified attribute, while the MJ needs its input relations to be sorted on the join attributes. However, as the join attribute S.rid neither is assumed to be sorted nor an index exists, only the SHJ or the NLJ can be used for the join between R and the filtered S, whereby the SHJ always dominates the NLJ due to its additive cost function. As the SHJ only propagates existing sortedness of its probe input but does not introduce any particular sortedness for its output, the top-level join also has to be computed utilizing a SHJ. Due to Definition (4), the smaller input to a SHJ is ordered to be the left child, which results in the final QEP shown on the right in Figure 2b.

HOLISTIC. In contrast, HOLISTIC incorporates both the algebraic and the physical optimization steps into a single optimization task that directly determines a total operator order. Therefore, all possible join orders including concrete physical operators have to be enumerated. While doing so, it is essential to not only memorize a single QEP per subproblem of the query but rather the locally best QEP *per distinct output property* similar to interesting properties in prior work [30]. Thus, the globally optimal QEP can be found in the entire search space. The resulting final QEP is depicted in Figure 2c. By comparing this QEP to SPLIT, we observe that another — more costly w.r.t. C_{out} — join order is applied, however, the overall physical costs are estimated to be lower.

TOP-K. Our proposed approach TOP-K works similar to SPLIT but considers the top k algebraic plans for physical optimization. In our example, the holistically optimal join order is the second cheapest w.r.t. algebraic costs. Therefore, TOP-K with k=2 would exactly produce the QEP depicted in Figure 2c.

This example showcases that HOLISTIC is able to find a *globally* optimal QEP w.r.t. QOP whereas the greedy SPLIT may be restricted by the predetermined join order. The same restriction holds for TOP-K if the hyperparameter k is set s.t. it does not include the join order chosen by HOLISTIC. Therefore, optimizing according to SPLIT or TOP-K may only yield a *locally* optimal QEP even if both involved optimization steps are globally optimal by themselves.

1.2 Contributions

In this paper, we make the following contributions.

(1) We revisit SPLIT and HOLISTIC in detail and present pseudocode for both. Furthermore, we show how to extend our physical optimization implementation to support fused physical operators. (Section 2.1 to Section 2.5)
(2) We propose TOP-K, a hybrid approach between the two extrema SPLIT and HOLISTIC. (Section 2.4)

[3]The magic constants 1.1 and 1.5 represent a per-tuple overhead induced by accessing the index structure or rather hash table. The values are experimentally validated to be order preserving.

(3) We analyze the time complexity and the optimality w.r.t. finding a QEP under a given cost model of SPLIT and HOLISTIC. (Section 3 and Section 4)
(4) We evaluate the optimization time needed by the three approaches for different query shapes and sizes. (Section 5.2)
(5) We showcase three scenarios where HOLISTIC outperforms SPLIT w.r.t. end-to-end performance. (Section 5.3)
(6) We investigate whether these or similar scenarios occur in benchmarks like TPC-H, JOB, or CEB. Our results show that HOLISTIC is able to outperform SPLIT for a few relatively small queries, however, SPLIT is often superior due to its significantly lower optimization time and some limitations of our implementation. In some cases, TOP-K achieves the best end-to-end performance. (Section 5.4)
(7) We propose a simple guideline to decide which optimization approach to apply in certain scenarios. (Section 5.5)

2 ALGORITHMS

In this section, we revisit and contrast SPLIT and HOLISTIC based on detailed pseudocode. Algorithm 1 and Algorithm 2 depict this pseudocode as side-by-side comparison and highlight lines that perform physical optimization in brown. Additionally, a helper function to choose physical operators for a given algebraic plan is defined in Algorithm 3. To explain the pseudocode, we revisit the running example in Figure 2 and optimize it step by step.

Afterward, we explain how to extend our physical optimization implementation to support fused physical operators, i.e., physical operators that implement the logic of multiple algebraic operators, like the hash-based group-join. Additionally, we propose a hybrid approach called TOP-K.

2.1 SPLIT

The core idea of SPLIT is to first transform a given query graph into an algebraic plan before physically optimizing this single algebraic plan afterwards. In Algorithm 1, these individual steps are clearly separated in the main function SPLIT (lines 42–44). To accelerate the optimization time, we apply dynamic programming and store already solved intermediate results in a plan table.

Algebraic Optimization. In line 4, the algebraic optimization step allocates a fresh plan table. This plan table maps a subproblem, i.e., a subset of all base relations determining which base relations are already joined, to an algebraic plan for that subproblem together with its estimated algebraic cost. Note that the cost column represents a cumulative cost consisting of the recursive costs of the children and the cost of the subproblem itself.

Firstly, lines 6–16 initialize the plan table by inserting entries for all (possibly filtered) sources of the query graph. Note that a nested query graph, e.g., due to a nested SQL query, is recursively optimized and the received algebraic plan is inserted into the plan table for the singleton subproblem of the nested source (line 12). The costs for all these entries are set to 0 according to C_{out} (line 16). For our running example, three entries are inserted during this initialization and are depicted in the first three rows of Table 1a.

Secondly, the plan table is filled with entries for every possible join order. Any plan enumerator that is applicable for dynamic programming, i.e., for any enumerated join both children subproblems are already solved, can be utilized in line 18. For the scope of our example, we assume that all join orders including Cartesian products are enumerated by a plan enumerator called PE_{all}. Join enumeration returns the left and right subproblems to join together with the respective join condition. As defined in lines 19–25, each enumerated join is then constructed, its algebraic cost is estimated, and the plan table is updated if the cost is lower than beforehand. Note that if a subproblem was not yet inserted into the plan table, its former cost is implicitly assumed to be infinity. As shown in Table 1a, the subproblems {R, S} and {S, T} of our example can be computed using a join, whereas no join predicate for the subproblem {R, T} exist resulting in a comparatively large intermediate result and thus algebraic cost w.r.t. C_{out}. For the subproblem {R, S, T} multiple subproblem splits, i.e., join orders, are enumerated. However, since {R, S} was estimated to be the cheapest of the subproblems containing two base relations, the join order $(R \bowtie \sigma(S)) \bowtie T$ is chosen. For simplicity, we omit the cardinality of the final join result in the cost column as it is added for every possible join order.

Finally, lines 27–32 update the final algebraic plan by adding all operations that are present after all sources have been joined, i.e., grouping, aggregation, sorting, limit, and projection. However, the actual join order remains unchanged. As the running example does not include such post-join operations, the plan table depicted in Table 1a is already complete and the final algebraic plan in the last row is returned (line 33).

Physical Optimization. The subsequent physical optimization step utilizes a slightly adapted plan table structure. Besides storing QEPs and physical costs instead of algebraic plans and algebraic costs, the mapping key is no longer only a subproblem. Instead, a combination of a subproblem and an output property is used as compound key. We refer to this output property as *post-condition* from this point on. Line 37 first allocates such an empty plan table before lines 38–39 call the helper function OPTIMIZE_PHYSICALLY for each algebraic subplan in a bottom-up manner.

Algorithm 3 defines this helper function that iterates over all available physical operators (line 3), all children permutations to determine a total order (line 9), and the Cartesian product of the current children permutation (line 11). During iteration, lines 4–5 skip physical operators which do not *cover*, i.e., implement, the given algebraic operator. For the time being, covering is only defined on singleton physical operators, however, we extend this to fused physical operators, i.e., a single physical operator which unites the logic of multiple algebraic operators, in Section 2.5. Additionally, pre-conditions of physical operators, e.g., the condition that a MJ needs its input relations to be sorted on the join attributes, are tested against the post-conditions provided by the children permutation. If not fulfilled, the respective children permutation is skipped as well (lines 12–13). For each of the qualifying combinations, lines 14–21 construct the respective QEP, estimate its physical cost, and update the plan table. This part is very similar to the algebraic step (see lines 19–25 of Algorithm 1). To finally determine the optimal QEP, the entry of the final subproblem with the lowest cost while ignoring the post-conditions is chosen (line 41 of Algorithm 1).

Table 1b depicts the enumeration for our running example given the algebraic plan constructed beforehand. Note that we need to distinguish between subproblems with and without filter and denote the filtered base relation S by S'. We can observe that two QEPs

Algorithm 1: Pseudocode for SPLIT.

```
1:  ▷ Optimizes the query graph G by applying the algebraic step of SPLIT.
2:  function SPLIT_ALGEBRAIC_STEP(G)
3:      ▷ maps subproblems to the locally optimal algebraic plan/cost
4:      PT = new AlgebraicPlanTable()
5:      ▷ optimize sources and optional filters
6:      for src ∈ G.sources() do
7:          s = Subproblem(src)
8:          if src.is_base_relation() then
9:              alg_plan = new ScanOp(src)
10:             PT[s].plan = alg_plan
11:         else                          ▷ optimize nested query graph recursively
12:             PT[s].plan = SPLIT_ALGEBRAIC_STEP(src)
13:         if src.has_filter() then
14:             alg_plan = new FilterOp(s, src.filter())
15:             PT[s].plan = alg_plan
16:         PT[s].cost = 0                ▷ initialize sources without cost
17:     ▷ optimize join order by enumerating subproblem splits
18:     for l, r, cond ∈ PE.enumerate(G) do
19:         alg_plan = new JoinOp(l, r, cond)
20:         ▷ estimate cost according to algebraic cost model and add subcosts
21:         cost = AC.estimate(alg_plan) + PT[l].cost + PT[r].cost
22:         s = alg_plan.get_subproblem()  ▷ unifies left and right subproblems
23:         if not PT.has(s) or cost < PT[s].cost then
24:             PT[s].plan = alg_plan                          ▷ update plan
25:             PT[s].cost = cost                              ▷ update cost
26:     ▷ optimize post-join operations, e.g., grouping
27:     s = Subproblem(G.joins())          ▷ subproblem with all sources joined
28:     if G.has_grouping() then
29:         alg_plan = new GroupingOp(s, G.grouping())
30:         PT[s].plan = alg_plan                              ▷ update plan
31:
32:     ...                    ▷ similar if's for aggregation, sorting, limit, and projection
33:     return PT[s].plan                                      ▷ return final algebraic plan

34: ▷ Optimizes the algebraic plan by applying the physical step of SPLIT.
35: function SPLIT_PHYSICAL_STEP(alg_plan)
36:     ▷ maps subproblems to the locally optimal QEP per distinct post-condition
37:     PT = new PhysicalPlanTable()
38:     for all alg_op ∈ alg_plan do                           ▷ in bottom-up manner
39:         OPTIMIZE_PHYSICALLY(PT, alg_op)
40:     s = alg_plan.get_subproblem() ▷ subproblem representing entire algebraic plan
41:     return plan where cost is minimal in PT[s]             ▷ return final QEP

42: function SPLIT(G)
43:     alg_plan = SPLIT_ALGEBRAIC_STEP(G)
44:     return SPLIT_PHYSICAL_STEP(alg_plan)
```

Algorithm 2: Pseudocode for HOLISTIC.

```
1:  ▷ Optimizes the query graph G by applying HOLISTIC.
2:  function HOLISTIC_HELPER(G)
3:      ▷ maps subproblems to the locally optimal QEP/cost per distinct post-condition
4:      PT = new PhysicalPlanTable()
5:      ▷ optimize sources and optional filters
6:      for src ∈ G.sources() do
7:          s = Subproblem(src)
8:          if src.is_base_relation() then
9:              alg_plan = new ScanOp(src)
10:             OPTIMIZE_PHYSICALLY(PT, alg_plan)
11:         else                          ▷ optimize nested query graph recursively
12:             PT[s] = HOLISTIC_HELPER(src)
13:         if src.has_filter() then
14:             alg_plan = new FilterOp(s, src.filter())
15:             OPTIMIZE_PHYSICALLY(PT, alg_plan)
16:
17:     ▷ optimize join order by enumerating subproblem splits
18:     for l, r, cond ∈ PE.enumerate(G) do
19:         alg_plan = new JoinOp(l, r, cond)
20:         OPTIMIZE_PHYSICALLY(PT, alg_plan)
21:
22:
23:
24:
25:
26:     ▷ optimize post-join operations, e.g., grouping
27:     s = Subproblem(G.joins())          ▷ subproblem with all sources joined
28:     if G.has_grouping() then
29:         alg_plan = new GroupingOp(s, G.grouping())
30:         OPTIMIZE_PHYSICALLY(PT, alg_plan)
31:         s = alg_plan.get_subproblem()  ▷ update current subproblem
32:     ...                    ▷ similar if's for aggregation, sorting, limit, and projection
33:     return PT[s]                       ▷ return post-cond. to QEP/cost mapping

34:
35:
36:
37:
38:
39:
40:
41:

42: function HOLISTIC(G)
43:     map = HOLISTIC_HELPER(G)
44:     return plan where cost is minimal in map.values()      ▷ return final QEP
```

Algorithm 3: Pseudocode for the physical optimization.

```
1:  ▷ Physically optimizes the algebraic operator. Inserts found QEPs into the plan table.
2:  function OPTIMIZE_PHYSICALLY(PT, alg_op)
3:      for all registered physical operators phys_op do
4:          if not phys_op covers alg_op then
5:              continue
6:          pre_conds = phys_op.get_pre_conds()
7:          s_children = phys_op.get_children_subproblems(alg_op)
8:          ▷ iterate over permutations of already solved children subproblems
9:          for p ∈ permute(s_children) do
10:             ▷ iterate over Cartesian product of current permutation
11:             for post_conds, entries ∈ PT[p[0]] × … × PT[p[n]] do
12:                 if not pre_conds.implied_by(post_conds) then
13:                     continue
14:                 qep = new phys_op(entries[0].plan, …, entries[n].plan)
15:                 post_cond = phys_op.get_post_cond(post_conds)
16:                 ▷ estimate cost according to physical cost model and add subcosts
17:                 cost = PC.estimate(qep) + entries[0].cost + … + entries[n].cost
18:                 s = qep.get_subproblem() ▷ unifies and adapts children subproblems
19:                 if not PT.has(s, post_cond) or cost < PT[s][post_cond].cost then
20:                     PT[s][post_cond].plan = qep                ▷ update plan
21:                     PT[s][post_cond].cost = cost               ▷ update cost
```

for the subproblem {T} are memorized since they result in different post-conditions. Also sortedness on the join attribute S.id for the subproblem {R, S'} can be provided, however, is quite costly due to building the hash table on the relatively large input R. Therefore, applying a MJ as top-level join (last row) is more costly than the final QEP that makes use of a SHJ on SHJ(S, R) and T_1.

Pruning. There are multiple ways to exploit pruning. For example, we can utilize pruning in the physical optimization step, i.e., in OPTIMIZE_PHYSICALLY. In line 19, our pseudocode is simplified to only check the plan table entry for the given post-condition. However, pruning can be added by discarding entries which are dominated by others, i.e., in terms of cost and post-condition. For example, the SHJ-entry for the subproblem {S', T} in Table 2 can be pruned since the MJ-entry for this subproblem dominates it. Vice versa, instead of always inserting new entries, old entries are updated if they are subsumed by the new one. This avoids unnecessary – in the sense of implied – entries in the plan table and is applied for all approaches.

Table 1: Enumeration of SPLIT.

(a) Algebraic optimization step.

Subproblem	Algebraic Plan	Cost
{R}	R	0
{S}	σ(S)	0
{T}	T	0
{R, S}	R $\bowtie$ σ(S)	80
{S, T}	σ(S) $\bowtie$ T	90
{R, T}	R $\times$ T	15,000
{R, S, T}	(R $\bowtie$ σ(S)) $\bowtie$ T	80 + |R, S, T|

(b) Physical optimization step.

Subproblem	Post-Condition	QEP	Cost
{R}	—	Scan(R) =: R	150
{S}	sorted on S.id	Scan(S)	100
{S'}	sorted on S.id	BF(Scan(S)) =: S	200
{T}	—	Scan(T) =: T_1	100
	sorted on T.sid	ISAM(T,sid) =: T_2	110
{R, S'}	—	SHJ(S, R)	620
	sorted on S.id	SHJ(R, S)	655
{R, S', T}	—	SHJ(SHJ(S, R), T_1)	940
	sorted on S.id/T.sid	MJ(SHJ(R, S), T_2)	945

2.2 HOLISTIC

In general, SPLIT and HOLISTIC have much in common. Both optimize their data sources potentially recursively, enumerate join orderings, add post-join operations, and can be implemented using dynamic programming. The main difference between both approaches is the point in time at which the physical optimization step is performed, as highlighted in brown. Instead of first constructing the entire algebraic plan before applying physical optimization only once thereafter like in Algorithm 1, HOLISTIC determines the physical operators directly at construction of algebraic operators. Therefore, the helper function OPTIMIZE_PHYSICALLY is called multiple times — particularly for every enumerated join order — as described in Algorithm 2.

Table 2 depicts the enumeration of HOLISTIC for the running example. We observe that the plan table mostly represents a superset of the one in Table 1b. The first five rows for singleton subproblems remain unchanged, however, subproblems representing another join order than utilized in SPLIT, e.g., {S', T} and {R, T}, are included. This observation is the key for finding a globally optimal QEP. Due to the additionally included subproblems, the full subproblem {R, S', T} is able to also apply the join order R $\bowtie$ (σ(S) $\bowtie$ T) and utilize a MJ between S and T_2. To finally determine the globally optimal QEP, again the entry with the lowest cost while ignoring the post-conditions is chosen (line 44 of Algorithm 2). As already stated in Section 1.1, this join order is more costly w.r.t. C_{out}, however, the resulting QEP has lower estimated physical cost compared to the QEP computed by SPLIT due to the freedom of applying any possible join order while already considering the physical cost model.

Lastly, we want to mention two caveats of Algorithm 2. (1) In line 31, we keep track of the current subproblem and elevate subproblems to the operations occurring after the final join, e.g., distinguish between joining all sources and applying an additional grouping afterward. This change is necessary since physical optimization is not only performed during join ordering but also for all other operations. (2) Strictly speaking, the recursion for a nested query graphs in line 12 yields a greedy local optimization. However, under certain assumptions, which are fulfilled in practice as discussed in the next section, the resulting local optimum coincides with the global optimum.

Table 2: Enumeration of HOLISTIC.

Subproblem	Post-Condition	QEP	Cost
{R}	—	Scan(R) =: R	150
{S}	sorted on S.id	Scan(S)	100
{S'}	sorted on S.id	BF(Scan(S)) =: S	200
{T}	—	Scan(T) =: T_1	100
	sorted on T.sid	ISAM(T,sid) =: T_2	110
{R, S'}	—	SHJ(S, R)	620
	sorted on S.id	SHJ(R, S)	655
{S', T}	—	SHJ(S, T_1)	520
	sorted on S.id/T.sid	MJ(S, T_2)	490
{R, T}	—	NLJ(R, T_1)	15,250
	sorted on T.sid	NLJ(R, T_2)	15,260
{R, S', T}	—	SHJ(MJ(S, T_2), R)	925
	sorted on S.id/T.sid	SHJ(R, MJ(S, T_2))	955

2.3 Pruning Variant: HOLISTIC$_{OPT}$

Besides pruning in the physical optimization step, we can utilize known pruning techniques for top-down and bottom-up plan enumerators to restrict the search space during holistic optimization. For top-down like TD_{basic} [4, 5], DeHaan and Tompa [2] propose the application of *branch-and-bound pruning* using *accumulated-cost bounding* to prune, for example, the right subproblem of a join if the left subproblem was already more costly than the remaining budget to the cost of the currently best found plan. For bottom-up like DP_{ccp} [19] or PE_{all}, Haffner and Dittrich [8] propose the application of *initialized cost-based pruning* using GOO [3] to get an upper bound for the physical costs early, i.e., before the actual optimization, and to prune subplans that are more costly than this upper bound. We implemented both techniques for HOLISTIC and refer to the pruning variant as HOLISTIC$_{OPT}$ from now on.

2.4 Hybrid Approach: TOP-K

When comparing SPLIT and HOLISTIC one can make an interesting observation. HOLISTIC can be rephrased as two-step optimization problem as follows. Firstly, solve algebraic optimization by computing not only the best algebraic plan but all possible join orders. Afterward, perform the physical optimization step for all algebraic plans received. In practice, this strategy would surely increase the optimization time as both steps are no longer interleaved, however, it hints at the application of a hybrid approach, namely TOP-K.

TOP-K computes the k algebraic plans with minimal algebraic cost before resuming with the physical optimization of all those algebraic plans. Afterward, the QEP with the lowest physical cost is selected. Note that to compute the k optimal algebraic plans w.r.t. the algebraic cost model, the k cheapest partial algebraic plans have to be stored for each enumerated subproblem during the algebraic optimization step.

The hyperparameter k can be chosen freely, e.g., k=1 decays to be SPLIT, whereas k=∞ decays to be HOLISTIC. If k is chosen large enough, i.e., the join order of the globally optimal QEP is contained in the k cheapest algebraic plans, the QEP determined by TOP-K coincides with the QEP determined by HOLISTIC. Therefore, we will mostly focus on the two extrema SPLIT and HOLISTIC in the following but also include TOP-K in our evaluation. Future work could replace the static number k with a dynamic value, i.e., start with a small k and then incrementally increase k as long as a certain budget, e.g., the ratio of estimated physical cost versus invested optimization time, has not been exhausted.

2.5 Extension to Fused Physical Operators

So far, we restricted ourselves to singleton physical operators that only implement a single algebraic operator. However, there are also fused physical operators that directly implement the logic of multiple algebraic operators. The most prominent example of a fused physical operator is the hash-based group-join (HBGJ) [11, 18, 20] uniting the join and the grouping logic by reusing a single hash table. We say that the HBGJ covers the *pattern* of an algebraic join operator followed by an algebraic grouping operator. The core idea of physical optimization is to cover a given algebraic plan with patterns of available physical operators while fulfilling the pre-conditions mentioned in the former sections.

As stated in Section 2.2, we only locally optimize nested query graphs due to the recursion. However, as long as there are no fused physical operators breaking this "barrier", e.g., a fused operator covering first a grouping and afterward a join, the local optimum coincides with the global optimum as we still return the entire post-condition to QEP and cost mapping in the recursion step (line 33 in Algorithm 2). To the best of our knowledge, no current database system implements such a fused physical operator as a grouping will never appear before a join without nesting the query graph. Additionally, future work could try to resolve this issue by extending the post-conditions. For example, we could add an available hash table to the post-condition and reuse it in a later join which basically simulates the aforementioned fused operator.

3 TIME COMPLEXITY

In this section, we analyze the algorithms' time complexity.

SPLIT. Conceptually, SPLIT only performs both optimization steps exactly *once*. The worst-case complexity of algebraic optimization is covered by a large body of related work and determined to be exponential in the number of base relations n [13, 27], even using state-of-the-art enumeration algorithms like DP_{ccp}. This can be highlighted by the fact that one needs to enumerate all subproblems, i.e., elements of the powerset of all base relations. Furthermore, there are $\binom{n}{m}$ subproblems of size m and for each of them all possible splits have to be enumerated. This can be simplified by choosing all subsets of size 1 up to size $m-1$, i.e., $\sum_{i=1}^{m-1}\binom{m}{i} = 2^m - 2$. In total, we need to enumerate $\sum_{m=1}^{n}\binom{n}{m} * (2^m - 2) = -2^{n+1} + 3^n + 1$ possible join orders, which resides in $O(3^n)$.

The following physical optimization step involves testing a finite number of physical operators for each algebraic operator contained in the received algebraic plan. Moreover, a total order has to be determined using non-commutative cost models. In the worst case, different sibling orderings result in different post-conditions that propagate throughout the entire optimization process. For example, a physical join operator, receiving three or rather four QEPs with different post-conditions for its left and right subproblems, may yield up to 12 QEPs with different post-conditions. However, the number of enumerated QEPs is still exponentially bounded by the number of joins as they are the only non-unary operator. The number of joins is bounded by the number of base relations n minus 1. Note that such an exponential propagation highly depends on the actual physical implementation, e.g., a SHJ yields the post-condition of its probe child and thus stops the exponential propagation locally. As both the algebraic and the physical optimization step require exponential time, SPLIT's overall time complexity resides in $O(3^n)$.

HOLISTIC. On the other hand, HOLISTIC has to perform one conceptual physical optimization step *per* found algebraic plan, i.e., the exponential effort for physical optimization has to be spent multiple times instead of just once. Even if implemented efficiently using dynamic programming, still all algebraic plan structures determining the join order together with all possible base relation permutations to achieve the total order, have to be enumerated. The number of plan structures is asymptotically bounded by the Catalan number [18], i.e., exponentially, whereas the number of permutations is computed in $O(n!)$ for worst-case query graph shapes. This faculty renders HOLISTIC to reside in the super-exponential time class and to grow much faster than SPLIT w.r.t. optimization time. Our experimental evaluation in Section 5.2 confirms our findings.

4 OPTIMALITY

In this section, we analyze the algorithms' optimality.

SPLIT. The algorithm SPLIT might get stuck in a local optimum due to its greediness of determining an algebraic plan without considering physical operators. This greediness issue is already showcased by our running example in Section 1.1. The algebraic optimization step decides to perform the join between R and S first because it results in the smaller intermediate result (see algebraic plan of Figure 2b). The succeeding physical optimization step cannot change the join order anymore and is only able to choose concrete physical operators (see QEP of Figure 2b). However, including the assumption from our running example regarding sortedness of S.id and index on T.sid, we have already shown that utilizing a MJ and a different join order yields a cheaper QEP w.r.t. the physical cost model (see Figure 2c) that HOLISTIC is able to find.

HOLISTIC. In contrast, HOLISTIC enumerates all algebraic plan structures while testing all possible physical operators for each structure. For our running example, this enumeration is depicted in Table 2 and represents a superset of the enumeration of SPLIT in Table 1b. Therefore, HOLISTIC is guaranteed to find the globally optimal QEP. Even the recursion for nested query graphs does not prevent HOLISTIC from being globally optimal under the practical assumptions made at the end of Section 2.5.

5 EVALUATION

In this section, we want to experimentally evaluate and compare the four approaches SPLIT, HOLISTIC, $\text{HOLISTIC}_{\text{OPT}}$, and TOP-K. In general, we try to answer the following research questions (RQs).

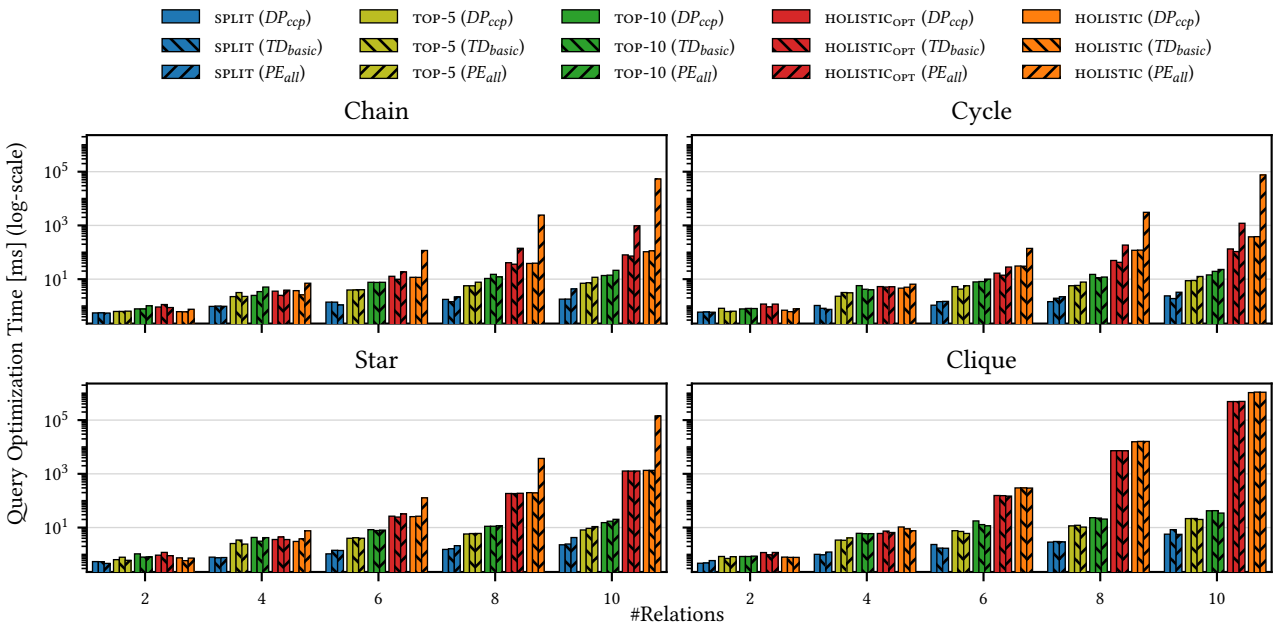


Figure 3: Optimization times for the different approaches when varying the query graph shape, the number of base relations, and the applied join enumeration algorithm.

(RQ 1) How much overhead does HOLISTIC, HOLISTIC$_{OPT}$, or TOP-K induce compared to SPLIT? How does the query shape and size influence the optimization time? (Section 5.2)

(RQ 2) Are there scenarios where HOLISTIC or rather TOP-K find better QEPs than SPLIT w.r.t. the given cost model? When does these approaches outperform SPLIT w.r.t. end-to-end performance? (Section 5.3)

(RQ 3) Do these scenarios occur in common benchmarks and datasets? How do the four approaches perform? (Section 5.4)

(RQ 4) How can the aforementioned scenarios be detected? (Section 5.5)

Therefore, we begin by investigating the optimization time of all approaches under varying query graph structures and join enumeration algorithms. Afterward, we provide three microbenchmarks showcasing that HOLISTIC is able to yield better QEPs than SPLIT and that TOP-K is able to recover these QEPs if the hyperparameter k is chosen large enough. Additionally, we also survey the TPC-H, JOB, and CEB benchmarks to verify our results. Lastly, we derive a simple guideline describing when to use which optimization approach for a given query graph to achieve best overall performance.

5.1 Experimental Setup

We implemented all four optimization approaches in mutable [9, 10], a main-memory database system currently developed in our group. All experiments are conducted using a columnar data layout and our just-in-time compiling push-based execution backend [23].

We run all our experiments on a Linux machine with an Intel® Xeon® CPU E5-2620 v4 (2.10 GHz, 20 MiB L3) and 4x8 GiB DDR4 RAM. Since mutable does not yet support multi-threading, all queries run on a single core. All data accessed in the experiments is memory resident. To remove the influence of cardinality estimation, we inject the true cardinalities for each executed query into mutable's cardinality estimator. We repeat each experiment five times and report the median. In the following, with *optimization time* we exclusively refer to the time required by the optimization process as described in Section 2, and with *query execution time* we refer to the sum of optimization and running time.

The algorithms SPLIT and TOP-K always assume C_{out} [8, 18] as algebraic cost model. The physical cost model is given by Definitions (1)–(6) and extended as follows.

$$C_{\text{HBG}}(x) := 1.5 * |x| \tag{7}$$

$$C_{\text{HBGJ}}(x, y) := 1.5 * |x| + |y| + |x, y| \tag{8}$$

Note that $|x, y|$ denotes the cardinality of the subproblem $\{x, y\}$. In both cases, we omit the costs for base relations and their respective scans as they occur in every possible QEP.

5.2 Optimization Time

In this section, we try to answer RQ 1 by investigating the optimization time required by SPLIT, HOLISTIC, HOLISTIC$_{OPT}$, and TOP-K. We conduct one experiment per query graph shape, i.e., chain, cycle, star, and clique, in which we vary the size of the query graph, i.e., the number of base relations, and measure the optimization time needed to find its optimal QEP using either SPLIT, TOP-K with k=5, TOP-K with k=10, HOLISTIC$_{OPT}$, or HOLISTIC. We also vary the join enumeration algorithm since DP_{ccp} is the state-of-the-art algorithm, TD_{basic} enables branch-and-bound pruning, and PE_{all} makes HOLISTIC truly holistic by also considering Cartesian products. Note that we denote for example TOP-K with k set to 5 as TOP-5.

Figure 3 shows the results of this experiment. The key observation is that the optimization time of all approaches grows at least exponentially with the number of base relations due to the linear

grow on a logarithmic scale, as stated in Section 3. We are also able to observe the significant overhead induced by computing the base relation permutations during HOLISTIC. For example, holistically optimizing a query graph containing 10 base relations requires up to 1000s whereas SPLIT always finds its local optimum in under 10ms. The TOP-K configurations form a middle ground between the two extrema SPLIT and HOLISTIC and its optimization time increases the larger k is chosen. For large queries, TOP-K performs significantly better than HOLISTIC, especially when applying PE_{all}. For small queries, TOP-K enumerates all possible join orders similar to HOLISTIC, however, without incorporating both optimization steps into a single one. Thus, we observe an overhead of TOP-K compared to HOLISTIC (see Figure 1a). The potential gain[4] of applying pruning is clearly visible when comparing $\text{HOLISTIC}_{\text{OPT}}$ with HOLISTIC, especially for PE_{all} where many Cartesian products can be directly pruned. Interestingly, there is no improvement for star queries when applying DP_{ccp} or TD_{basic} since joining dimension tables in a suboptimal order with the fact table is not costly enough to prune due to the output cardinality of each subproblem being bounded by the size of the fact table. Furthermore, our results confirm the efficiency of DP_{ccp} and TD_{basic} compared to PE_{all}, especially for HOLISTIC as significantly less physical plans have to be enumerated without Cartesian products. This effect is most visible for sparse query graph shapes like chain or cycle and decreases with more dense query graph shapes like star until there is no difference visible for a fully connected query graph, i.e., a clique. Actually, for cliques, DP_{ccp} and TD_{basic} perform slightly worse since they enumerate the same join orders as PE_{all} but with a more complex computation.

5.3 End-To-End Microbenchmarks

With three end-to-end microbenchmarks, we showcase that HOLISTIC is able to find a QEP that is superior w.r.t. estimated cost to the QEP found by applying SPLIT. However, as shown in the last section, HOLISTIC is computationally more expensive and thus takes more time to find the QEP. Therefore, there is a trade-off between optimization time and running time.

We also optimize our microbenchmarks utilizing TOP-2 and $\text{HOLISTIC}_{\text{OPT}}$, which find the same QEPs as HOLISTIC. As the optimization times are mostly negligible in these experiments, i.e., below 3ms, there is no real difference between TOP-2, $\text{HOLISTIC}_{\text{OPT}}$, and HOLISTIC. Moreover, as the choice of plan enumerator does not influence the resutling QEP, we only state the mean of utilizing DP_{ccp}, TD_{basic}, and PE_{all} in the remainder of this section.

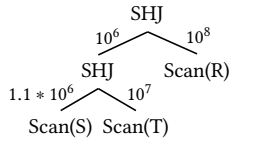
All microbenchmarks run on a synthetic dataset. Primary keys as well as foreign keys in the base relations are represented as 4-byte integer values. The data is drawn uniformly at random s.t. it fulfills the desired cardinalities specified in each microbenchmark.

Exploiting Sortedness. In the first microbenchmark, we aim at reusing existing sortings. We consider a simple select-project-join query shown in Figure 4a. The query consists of a total of two explicit joins forming a query graph commonly known as chain shape. In this microbenchmark, base relation R represents, e.g., students, S a subclass of R, e.g., PhD students, and T some relationship with R, e.g., attending courses. We may assume R and S to be presorted on their primary keys R.id and S.rid, respectively. Moreover, the join between S and T as well as its specified cardinality may occur in practice as not all PhD students have to attend courses anymore.

[4]We defined the cardinalities s.t. they favor pruning to the maximum, but do not influence the enumeration of join orders and QEPs for approaches other than $\text{HOLISTIC}_{\text{OPT}}$.

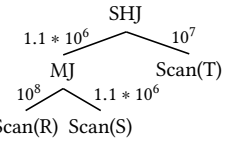
```
SELECT *
FROM R, S, T
WHERE R.id = S.rid AND S.rid = T.rid;
```

(a) SQL query inducing sortedness exploitation.



$\textbf{cost}_{\textbf{alg}}=10^6$, $\textbf{cost}_{\textbf{phys}}=1.1315 * 10^8$

(b) Applying SPLIT.



$\textbf{cost}_{\textbf{alg}}=1.1 * 10^6$, $\textbf{cost}_{\textbf{phys}}=1.1275 * 10^8$

(c) Applying HOLISTIC.

Figure 4: Microbenchmark exploiting sortedness. QEP edges are annotated with the respective cardinalities for sf=1.0.

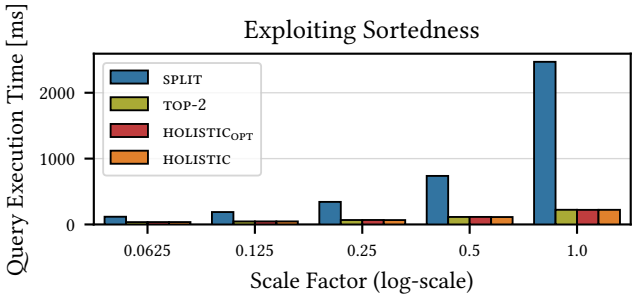


Figure 5: Query execution times for the final QEPs shown in Figure 4 when varying the scale factor and the approach.

On the one hand, HOLISTIC tries to exploit existing sortings by executing a MJ similar to our running example in Figure 2. On the other hand, SPLIT does not consider physical output properties during the algebraic optimization step and chooses another join order disabling the application of a MJ in the subsequent physical optimization step. Both found QEPs are depicted in Figure 4.

Figure 5 shows the query execution times for the final QEPs for varying scale factors (sf) and approaches utilized. We observe that the holistically optimized QEP dominates the one chosen by SPLIT for all scale factors with an increasing speedup from 3.2x to 11.1x.

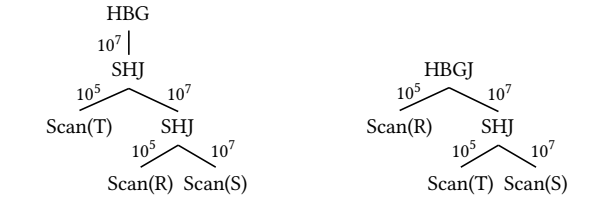
Exploiting Group-Join. The second microbenchmark investigates the exploitation of fused physical operators by utilizing the query defined in Figure 6a. The query induces two possible join orders and a subsequent grouping on one of the join keys.[5]

HOLISTIC enables the use of a HBGJ by performing the join that does not contain the grouping key at first. Again, SPLIT is more restricted in the sense that the algebraic optimization step determines the other possible join order ruling out the application of a fused group-join in the subsequent physical optimization step. Therefore, the grouping has to be executed using a separate physical operator, i.e., the hash-based grouping (HBG). Figure 6 shows both resulting QEPs. Note that even if no intermediate algebraic

[5]Note that the stated query could be rewritten to a single-relation-query on the base relation S and grouping directly on S.rid as the joins with R and T do not add any information here. However, filters on the other base relations would require the joins again. For simplicilty, we omit these filters in this microbenchmark.

```sql
SELECT R.id, COUNT(*)
FROM R, S, T
WHERE R.id = S.rid AND S.tid = T.id
GROUP BY R.id;
```

(a) SQL query inducing group-join exploitation.



$\textbf{cost}_{\textbf{alg}}=2*10^7$, $\textbf{cost}_{\textbf{phys}}=4.025*10^7$ | $\textbf{cost}_{\textbf{alg}}=2*10^7$, $\textbf{cost}_{\textbf{phys}}=3.03*10^7$

(b) Applying SPLIT. **(c) Applying HOLISTIC.**

Figure 6: Microbenchmark exploiting group-join. QEP edges are annotated with the respective cardinalities for sf=1.0.

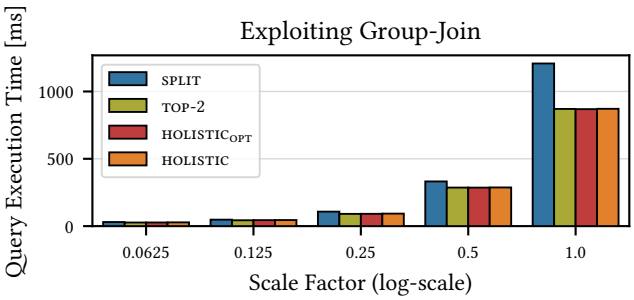


Figure 7: Query execution times for the final QEPs shown in Figure 6 when varying the scale factor and the approach.

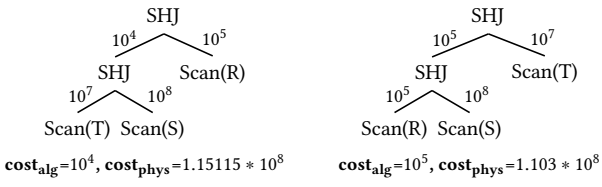
plan is constructed by HOLISTIC, the algebraical cost would still be $2*10^7$ for the determined join order and thus the same as for SPLIT. So in theory, SPLIT could have chosen this join order as well resulting in the same QEP as HOLISTIC (we circumvented that by slighlty adapting the cardinalities). However, HOLISTIC will always force the favorable join order enabling the HBGJ.

We perform an experiment with the same configurations, i.e., varying scale factor and approaches, as for the first microbenchmark. Figure 7 depicts the results. This time, SPLIT is on par with HOLISTIC for small scale factors as the optimization overhead of HOLISTIC is too high for comparatively low running times of 30ms. However, with larger datasets, the query running time increases while the optimization time remains constant. From scale factor 0.25 onwards, HOLISTIC becomes increasingly superior to SPLIT. This indicates that the holistic QEP is again more beneficial in practice than the QEP computed by SPLIT, however, with a smaller absolute difference than in the first microbenchmark. However, we would always be able to find a threshold at which HOLISTIC outperforms SPLIT w.r.t. query execution time by upscaling the dataset even if the holistic QEP is only slighlty more beneficial in practice.

Removing Algebraic Cost Abstraction. In the last microbenchmark, we examine the impact of purely applying a physical cost model in HOLISTIC in contrast to applying two different cost models, i.e., first algebraic then physical, in SPLIT. Therefore, we again consider a simple select-project-join query shown in Figure 8a, now consisting of three relations and a total of three joins including one

```sql
SELECT *
FROM R, S, T
WHERE R.sid = S.id AND S.id = T.sid;
```

(a) SQL query inducing multiple join orders.



$\textbf{cost}_{\textbf{alg}}=10^4$, $\textbf{cost}_{\textbf{phys}}=1.15115*10^8$ | $\textbf{cost}_{\textbf{alg}}=10^5$, $\textbf{cost}_{\textbf{phys}}=1.103*10^8$

(b) Applying SPLIT. **(c) Applying HOLISTIC.**

Figure 8: Microbenchmark removing algebraic cost abstraction. QEP edges are annotated with the respective cardinalities for sf=1.0.

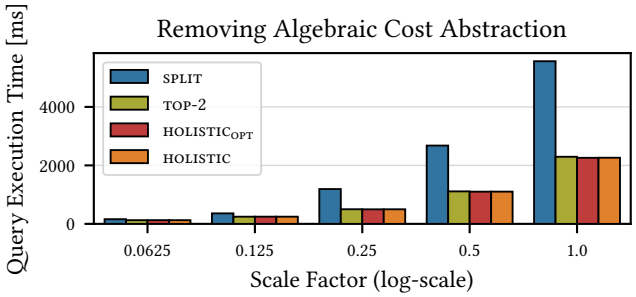


Figure 9: Query execution times for the final QEPs shown in Figure 8 when varying the scale factor and the approach.

transitively derived join between R.sid and T.sid. Thus, multiple join orders are possible. We do not assume any existing sorting but assume the cardinalities as given in Figure 8.[6]

As mentioned in Section 5.1, SPLIT applies C_{out} as algebraic cost model. C_{out} can be seen as a simplified version of the physical cost model given by Definitions (1)–(8). While this abstraction accumulates the cost computation during join ordering, it also causes inaccuracy, e.g., by removing the factor estimating the cost of creating a hash table. Therefore, there are cases where SPLIT "oversimplifies" which results in finding only a local optimum instead of the global optimum guaranteed to be found by HOLISTIC. This third microbenchmark triggers exactly such a scenario as the QEPs and costs depicted in Figure 8 confirm. Even if the algebraic cost are lower for the QEP found by SPLIT, it is forced to build the hash table for the lower SHJ on one huge input (either T or S). In contrast, HOLISTIC "divides" these two huge inputs across two different joins making it possible to build both required hash tables on smaller inputs each (on R and the join result of R and S).

Again, we measured the query execution time for different scale factors and approaches. The results shown in Figure 9 have similar characteristics as the first microbenchmark. HOLISTIC dominates SPLIT for all scale factors with an increasing speedup up to 2.5x.

In conclusion, RQ 2 can be confirmed since we found three independent scenarios where HOLISTIC, and thus also TOP-2 and HOLISTIC$_{OPT}$, are superior to SPLIT, especially for larger datasets.

[6]Note that there must be some filter on S to achieve the cardinality of the subproblem {S, T} (due to the foreign key property), however, we omit these filters for simplicity.

Table 3: Performance of SPLIT compared to TOP-K, HOLISTIC$_{OPT}$, and HOLISTIC for varying join enumeration algorithms on TPC-H and JOB. For TOP-K, HOLISTIC$_{OPT}$, and HOLISTIC, we also specify the speedup over SPLIT in parantheses, i.e., $t_{SPLIT}/t_{TOP\text{-}K}$, $t_{SPLIT}/t_{HOLISTIC_{OPT}}$, or $t_{SPLIT}/t_{HOLISTIC}$, respectively. All measurements are rounded to one decimal place.

Query	TPC-H q3											
Join Enumeration	DP_{ccp}				TD_{basic}				PE_{all}			
Approach	SPLIT	TOP-2	HOLISTIC$_{OPT}$	HOLISTIC	SPLIT	TOP-2	HOLISTIC$_{OPT}$	HOLISTIC	SPLIT	TOP-2	HOLISTIC$_{OPT}$	HOLISTIC
Optimization Time [ms]	1.3	1.9 (0.66)	3.0 (0.42)	3.5 (0.36)	1.3	1.9 (0.65)	3.0 (0.42)	3.4 (0.37)	1.3	1.9 (0.66)	3.1 (0.41)	4.2 (0.30)
Running Time [ms]	222.1	157.1 (1.41)			221.3	158.0 (1.40)			217.7	156.7 (1.39)		
∑ [ms]	223.4	159.0 (1.40)	160.1 (1.40)	160.6 (1.39)	222.6	159.9 (1.39)	161.0 (1.38)	161.4 (1.38)	219.0	158.6 (1.38)	159.8 (1.37)	160.9 (1.36)

Query	JOB q5c											
Join Enumeration	DP_{ccp}				TD_{basic}				PE_{all}			
Approach	SPLIT	TOP-3	HOLISTIC$_{OPT}$	HOLISTIC	SPLIT	TOP-3	HOLISTIC$_{OPT}$	HOLISTIC	SPLIT	TOP-3	HOLISTIC$_{OPT}$	HOLISTIC
Optimization Time [ms]	6.7	13.0 (0.52)	61.8 (0.11)	293.1 (0.02)	7.0	13.4 (0.52)	59.9 (0.12)	251.0 (0.03)	8.0	14.5 (0.55)	78.5 (0.10)	667.4 (0.01)
Running Time [ms]	1953.9	5055.6 (0.39)			1963.8	5039.6 (0.39)			2026.7	5099.8 (0.40)		
∑ [ms]	1960.7	5068.6 (0.39)	5117.3 (0.38)	5348.7 (0.37)	1970.8	5053.1 (0.39)	5099.6 (0.39)	5290.6 (0.37)	2034.7	5114.3 (0.40)	5178.3 (0.39)	5767.2 (0.35)

Query	JOB q8c											
Join Enumeration	DP_{ccp}				TD_{basic}				PE_{all}			
Approach	SPLIT	TOP-2	HOLISTIC$_{OPT}$	HOLISTIC	SPLIT	TOP-2	HOLISTIC$_{OPT}$	HOLISTIC	SPLIT	TOP-2	HOLISTIC$_{OPT}$	HOLISTIC
Optimization Time [ms]	3.2	4.2 (0.75)	35.1 (0.09)	315.1 (0.01)	3.2	4.3 (0.74)	32.3 (0.10)	320.5 (0.01)	2.3	3.5 (0.66)	48.3 (0.05)	1956.2 (0.00)
Running Time [ms]	2070.0	2079.7 (1.00)			2514.8	2503.9 (1.00)			3157.4	2516.9 (1.25)		
∑ [ms]	2073.2	2083.9 (0.99)	2114.8 (0.98)	2394.8 (0.87)	2517.9	2508.2 (1.00)	2536.2 (0.99)	2824.4 (0.89)	3159.7	2520.4 (1.25)	2565.2 (1.23)	4473.1 (0.71)

Query	JOB q11a											
Join Enumeration	DP_{ccp}				TD_{basic}				PE_{all}			
Approach	SPLIT	TOP-35	HOLISTIC$_{OPT}$	HOLISTIC	SPLIT	TOP-35	HOLISTIC$_{OPT}$	HOLISTIC	SPLIT	TOP-2	HOLISTIC$_{OPT}$	HOLISTIC
Optimization Time [ms]	24.7	367.4 (0.07)	2562.8 (0.01)	13410.8 (0.00)	24.9	372.2 (0.07)	2423.8 (0.01)	13721.4 (0.00)	12.3	15.2 (0.81)	8430.6 (0.00)	376719.0 (0.00)
Running Time [ms]	388.7	340.9 (1.14)			345.4	297.2 (1.16)			442.9	436.2 (1.02)		
∑ [ms]	413.4	708.3 (0.58)	2903.7 (0.14)	13751.7 (0.03)	370.3	669.4 (0.55)	2721.0 (0.14)	14018.6 (0.03)	455.2	451.5 (1.01)	8866.8 (0.05)	377155.2 (0.00)

5.4 TPC-H, JOB, and CEB Benchmarks

To investigate whether cases as described by the three microbenchmarks also occur in practice and to answer RQ 3, we survey the TPC-H [1], JOB [15, 29], and CEB [21, 22] benchmarks. Therefore, we compare the resulting QEPs of the different optimization approaches and measure again both the optimization time and the running time. JOB and CEB can be considered real-world as they query the IMDb dataset[7], whereas TPC-H queries synthetic data on scale factor 1.0 but is considered here due to containing post-join operations like grouping. For TPC-H, we tested every query with its substitution parameters set as specified for query validation. For CEB, we restrict ourselves to one query per query template.

Overall, out of the 19 TPC-H[8], 113 JOB, and 16 CEB queries we considered, HOLISTIC returns a different QEP than SPLIT for 51 queries, but only yields a significant speedup for very few queries. Therefore, we will examine those queries as well as additional scenarios explaining why no speedup is achieved by HOLISTIC. Additionally, we apply HOLISTIC$_{OPT}$, as well as TOP-K for the presented scenarios with the hyperparameter k set s.t. the same QEP is found as by HOLISTIC. Thus, we only show one running time for these approaches but state the optimization times individually.

Sample Queries. With our conducted evaluation, we detect four scenarios where SPLIT and HOLISTIC produce different QEPs. For each scenario, we investigate one representative sample query w.r.t. its performance. Our findings are summarized in Table 3.

In TPC-H q3, HOLISTIC is able to exploit a group-join independent of the utilized join enumeration whereas SPLIT is forced to execute singleton join and grouping operators due to unfavorable join order. Note that we adapted the date parameters for the filters slightly to achieve the desired cardinalities. Our experiment confirms that the QEP chosen by HOLISTIC indeed yields a superior running time in practice. Due to the comparatively low optimization effort for a query with only three base relations, the query execution time when applying HOLISTIC, as well as TOP-2 and HOLISTIC$_{OPT}$, reaches a speedup of approximately 1.4x compared to SPLIT.

In JOB q5c, HOLISTIC is able to exploit sortedness by utilizing multiple indexes to enable two MJs. In contrast, the join order chosen by SPLIT prevents the use of MJs. However, our measurements show that the QEP chosen by HOLISTIC actually yields a deteriorated running time in practice. This deterioration is caused by an additional materialization before the MJ necessary to execute the pull-based MJ logic[9] in an otherwise push-based execution engine [31]. Our cost model according to Defintion 5, however, does not incorporate this materialization. Also the optimization effort for this query with five base relations is relatively high when optimizing holistically s.t. SPLIT overall is superior by a factor of more than 2x. Although TOP-3 and HOLISTIC$_{OPT}$ significantly lower this optimization effort, they also fall short at end-to-end performance compared to SPLIT.

JOB q8c is an instance of our third microbenchmark that specifies a different join order due to the removal of the algebraic cost abstraction. Only SPLIT with PE_{all} results in a substantially different join order than the other 11 configurations due to a chosen Cartesian product which, however, only produces an intermediate of size 1. Interestingly, this join order induces a smaller search space for the subsequent physical optimization step which explains the faster optimization time of this configuration compared to SPLIT utilizing DP_{ccp} or TD_{basic}. As the QEPs match when applying DP_{ccp} or rather TD_{basic}, the running time also matches between all approaches. Note that the running time when applying DP_{ccp} differs from the others

[7] Note that we cut strings to 100 characters to meet mutable's memory limit of 16GiB.
[8] TPC-H q13, q15, and q20 are omitted due to lack of support of outer joins and views in mutable.

[9] A pull-based execution has control from which child the next tuple is requested. This is necessary in the MJ logic to switch between inputs multiple times in constrast to, for example, SHJ logic that only needs all tuples from its build input and then all tuples from its probe input.

due to a different children order of a NLJ which is not recognized by the symmetric cost function stated in Definition 6. Therefore, only the increased optimization time influences the query execution time. This optimization overhead renders HOLISTIC to be worse by a factor of 0.87x. TOP-2 as well as $\text{HOLISTIC}_{\text{OPT}}$ significantly reduce the optimization time and thus are on par with SPLIT. For PE_{all}, the aforementioned different join order when applying SPLIT performs worse in practice. Again, TOP-2 and $\text{HOLISTIC}_{\text{OPT}}$ find the holistic QEP in a time comparable to SPLIT. Therefore, compared to SPLIT, TOP-2 and $\text{HOLISTIC}_{\text{OPT}}$ are superior with a speedup of more than 1.2x. Furthermore, HOLISTIC might become superior for PE_{all} and larger datasets as the running time becomes the bottleneck.

Lastly, optimizing JOB q11a yields completely different QEPs with different join orders. Again, the physical optimization step in SPLIT is faster due to a smaller search space when resolving the join order determined by PE_{all}. To discover the globally optimal QEP found by HOLISTIC for DP_{ccp} or rather TD_{basic}, we need to apply TOP-35, i.e., the chosen join order is only the 35th cheapest w.r.t. the algebraic cost model. This explains the relatively large optimization time in these configuration. The running times are similar for all configurations and negligible when comparing them with the optimization effort needed by HOLISTIC, even if pruning significantly lowers the optimization time. In conclusion, this experiment highlights the overhead of holistically optimizing large queries, e.g., 8 base relations in q11a, and the potential of fast hybrid strategies to take advantage of even small speedups in the running times, e.g., in this experiment a speedup of approximately 1.15x for TOP-K and HOLISTIC compared to SPLIT for DP_{ccp} or TD_{basic}.

Limitations. In the considered benchmarks, HOLISTIC was only superior in one case, TOP-K and $\text{HOLISTIC}_{\text{OPT}}$ in two cases. On the one hand, this can be explained by the relatively low running times of the majority of queries that favors fast optimization like SPLIT, on the other hand, the benchmarks themselves impose certain limitations. JOB and CEB only contain select-project-join queries for which a group-join can not be exploited due to the missing grouping. Furthermore, sortedness can not be exploited using the TPC-H schema as join keys are only used for a specific join. Also in the CEB and JOB queries sortedness is difficult to exploit due to missing filters on join keys which disables the application of index scans fusing an algebraic scan and filter operator.

Additionally, our implementation has some limitations. The operators ISAM and index scan are limited to a sorted array index and their cost models are difficult to estimate. Additionally, they are currently parameterized with templates for the accessed attribute which artificially blows up the search space. Moreover, the MJ implementation is limited to 1:N joins and introduces additional materialization as explained for JOB q5c.

Discussion. Supporting additional physical operators, e.g., worst-case optimal joins [26] as fused operator, would increase the search space and the potential gain of holistically enumerating all QEPs. Additionally, we have seen for JOB q5c that an imprecise cost model may cause deteriorated running times in practice. Similarly, the same holds for imprecise cardinality estimates as they serve as input for the cost model. However, we expect all optimization approaches to suffer similarly from such inaccuracies as they rely on the same cost estimates. For the scope of this paper, we focus on the defined cost model and injecting the true cardinalities.

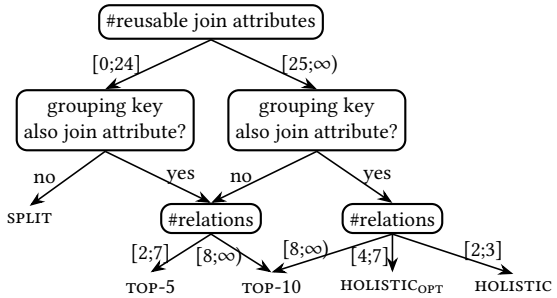


Figure 10: Decision graph as optimization guideline.

5.5 Optimization Guideline

In this section, we address RQ 4 by proposing a decision graph, depicted in Figure 10, for choosing a suitable optimization approach for a given query.

Since exploiting sortedness is only applicable iff join attributes are reused and exploiting a fused group-join is only applicable iff the grouping key is also a join attribute, we add those two conditions as nodes in the decision tree. We experimentally determined a value of 24 to be a good threshold for the number of reusable join attributes. If no condition is fulfilled, there is almost no potential gain for holistic optimization, thus we apply SPLIT. If only one condition is fulfilled, we apply TOP-K which also includes the algebraic cost abstraction scenario in our guideline by trying out multiple join orders that are more or less equally costly. Depending on the query size, we either use TOP-5 or TOP-10. If both conditions are fulfilled, we want to optimize holistically. We apply $\text{HOLISTIC}_{\text{OPT}}$, except for very small queries for which the initialized cost-based pruning would only induce an optimization overhead and thus applying HOLISTIC instead is beneficial. However, due to the computational effort to optimize large queries holistically, we define a threshold from which we fall back to TOP-10. Setting k to such a comparatively small number is often sufficient in practice as our microbenchmarks and the experiments in the previous section confirm.

We evaluate our proposed guideline along with SPLIT, TOP-5, TOP-10, and $\text{HOLISTIC}_{\text{OPT}}$ on all 51 aforementioned queries where HOLISTIC finds a different QEP than SPLIT. Note that we focus on DP_{ccp} for join enumeration and omit HOLISTIC as the majority of queries benefit from pruning. Figure 11 depicts the distribution of slowdown factors compared to the theoretical optimum across all queries. Each violin plot includes a box plot indicating the median, first and third quartiles, and whiskers extending 1.5 times the interquartile range. SPLIT and TOP-5 show similar performance, with almost all queries having a slowdown factor below 2x, indicating that both approaches typically achieve the optimal or a near-optimal query execution time. However, there are a few notable outliers, e.g., JOB q15a, where both miss the optimal QEP, leading to a slowdown of nearly 6x. TOP-10 shows greater variance and a higher median slowdown. While it finds the optimal QEP for JOB q15a, other queries, such as JOB q5c, q20c, and q33b, suffer from increased optimization overhead and become outliers. $\text{HOLISTIC}_{\text{OPT}}$ confirms earlier observations: despite being optimal for four queries, its significant optimization overhead makes it unsuitable as a default choice. Lastly, applying our guideline results in slowdown factors similar to TOP-5, with 75% of queries below

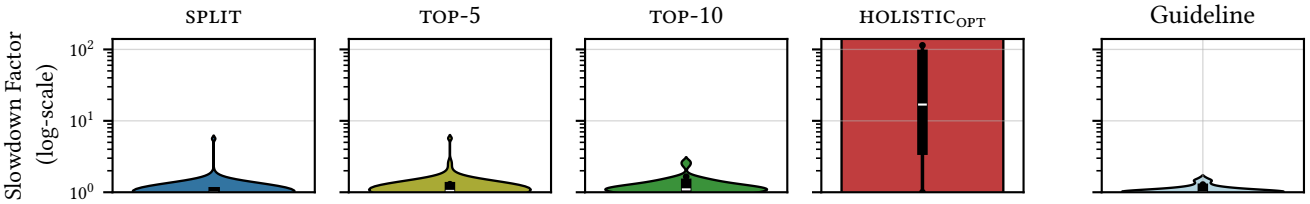


Figure 11: Robustness of different approaches compared to the theoretical optimum that chooses the optimization approach yielding the lowest execution time per query. The slowdown factor is computed by $t_{approach}/t_{optimum}$ per query, where $t_{approach}$ is the query execution time applying the respective approach and $t_{optimum}$ is the minimal execution time over all optimization approaches. Each plot shows the distribution of slowdown factors across all considered queries.

1.5x. Unlike fixed strategies, our guideline avoids outliers entirely, making it a more robust choice across all queries.

6 RELATED WORK

We investigate related publications in the fields of query optimization as well as concrete implementations of other database systems.

Related Publications. The conceptual separation of the QOP into algebraic and physical optimization steps is covered by a large body of work, especially in textbooks and teaching materials. Moerkotte [18] provides an extensive description of this separation. In that work, algebraic optimization is referred to as logical optimization that additionally includes rule-based optimizations like projection and selection push-down. The physical optimization step defined by Moerkotte is responsible for choosing physical operators while considering different output properties. Note that these properties are equivalent to our post-conditions, however, are only used in the physical optimization step in contrast to HOLISTIC. The seminal paper of Selinger et al. [30] lays the foundation for HOLISTIC as they incorporate the aforementioned properties into join ordering. They focus on different access paths, i.e., different physical implementations, but are limited to segment scans versus index scans and NLJs versus MJs. Moreover, the term *interesting orders* as first output property is coined. Other papers often restrict themselves to subproblems of the QOP like join ordering [8, 19, 33] by assuming SPLIT as optimization approach. To the best of our knowledge, HOLISTIC is never mentioned and evaluated explicitly even if, for example, Marcus et al. [17] apply a holistic yet machine-learned optimization in their query optimizer Neo and extend this idea by providing per-query optimization hints in Bao [16]. DeHaan and Tompa [2] introduce the idea of pruning in top-down join ordering. Besides accumulated-cost bounding, they also propose predicated-cost bounding as instantiation of branch-and-bound pruning, which may be implemented into HOLISTIC as well. Moerkotte and Neumann [20] make the case for fused operators by proposing group-joins. Simple implied equivalences are described that enable the application of a HBGJ in queries like TPC-H Q3. Our framework builds upon their ideas and is thus able to reproduce and extend this scenario in our evaluation. The concept of interesting orders, introduced by System R, is deepened and formally defined by Simmen et al. [32]. Later, Wang and Cherniack [34] extend these output properties to *interesting groupings*. They observe that sort-based physical implementations, e.g., a sort-based grouping, often only require their input to be grouped instead of fully sorted. Therefore, keeping track of existing groupings introduced by, for example, MJs enables the optimizer to utilize these physical operators.

Related Database Systems. DuckDB has a clear interface for its query optimizer that points out the application of SPLIT [6]. Moreover, DuckDB utilizes DP_{ccp} in combination with the cost model C_{out} and implements many rule-based optimization rules on the intermediate algebraic plan, however, to the best of our knowledge lacks the use of a physical group-join operator. PostgreSQL optimizes the join order before optimizing the post-join operators, however in contrast to DuckDB, it seems to apply HOLISTIC similar to System R since the join order optimizer returns multiple access paths from which the post-join optimizer can freely choose [7]. These access paths represent basically the mapping from post-conditions to local QEPs with its costs. Furthermore, PostgreSQL also does not support a group-join operator. Similarly, MySQL — and most likely its evolution MariaDB — implements HOLISTIC using interesting orders as post-condition [28]. In general, MySQL follows former research in this area, e.g., incorporating interesting groupings as well. In contrast, SQLite did not evolve from database research. Therefore, quite uncommon terms are used for some features, however, SQLite still utilizes HOLISTIC similar to System R by distinguishing between different access methods during join ordering [12], however, it ignores bushy plan structures.

7 CONCLUSION

In this work, we revisited and analyzed two existing query optimization strategies. With our experimental evaluation, we confirm the theoretical findings that HOLISTIC induces a significant optimization overhead, especially for larger queries, even when applying pruning. Nevertheless, we showcase scenarios where this overhead is justified for accelerating query execution and present an optimization guideline to detect such scenarios. In three microbenchmarks, HOLISTIC outperforms SPLIT by up to one order of magnitude. Furthermore, we propose a hybrid TOP-K optimization strategy and demonstrate that it is able to recover the holistic QEPs while improving the optimization time by multiple orders of magnitude compared to HOLISTIC. Additionally, we survey the TPC-H, JOB, and CEB benchmarks, where SPLIT is superior for the majority of the queries. However, we identify some queries for which HOLISTIC and especially TOP-K are able to outperform SPLIT by approximately 1.4x. Lastly, we propose a simple guideline that robustly determines a suitable optimization approach to apply for a given query.

REFERENCES

[1] Transaction Processing Performance Council. 2022. TPC-H Specification. Retrieved January 9, 2025 from https://www.tpc.org/TPC_Documents_Current_Versions/pdf/TPC-H_v3.0.1.pdf

[2] David DeHaan and Frank Wm. Tompa. 2007. Optimal top-down join enumeration. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Beijing, China, June 12-14, 2007*, Chee Yong Chan, Beng Chin Ooi, and Aoying Zhou (Eds.). ACM, 785–796. https://doi.org/10.1145/1247480.1247567

[3] Leonidas Fegaras. 1998. A New Heuristic for Optimizing Large Queries. In *Database and Expert Systems Applications, 9th International Conference, DEXA '98, Vienna, Austria, August 24-28, 1998, Proceedings (Lecture Notes in Computer Science)*, Gerald Quirchmayr, Erich Schweighofer, and Trevor J. M. Bench-Capon (Eds.), Vol. 1460. Springer, 726–735. https://doi.org/10.1007/BFB0054528

[4] Pit Fender. 2014. *Algorithms for Efficient Top-Down Join Enumeration.* Ph.D. Dissertation. University of Mannheim. https://ub-madoc.bib.uni-mannheim.de/36655

[5] Pit Fender and Guido Moerkotte. 2011. A new, highly efficient, and easy to implement top-down join enumeration algorithm. In *Proceedings of the 27th International Conference on Data Engineering, ICDE 2011, April 11-16, 2011, Hannover, Germany*, Serge Abiteboul, Klemens Böhm, Christoph Koch, and Kian-Lee Tan (Eds.). IEEE Computer Society, 864–875. https://doi.org/10.1109/ICDE.2011.5767901

[6] DuckDB Foundation. 2025. DuckDB GitHub. Retrieved December 5, 2024 from https://github.com/duckdb/duckdb/blob/main/src/include/duckdb/execution/physical_plan_generator.hpp#L41

[7] The PostgreSQL Global Development Group. 2025. PostgreSQL GitHub. Retrieved December 5, 2024 from https://github.com/postgres/postgres/blob/master/src/backend/optimizer/plan/planmain.c#L38

[8] Immanuel Haffner and Jens Dittrich. 2023. Efficiently Computing Join Orders with Heuristic Search. *Proc. ACM Manag. Data* 1, 1 (2023), 73:1–73:26. https://doi.org/10.1145/3588927

[9] Immanuel Haffner and Jens Dittrich. 2023. mu*t*able: A Modern DBMS for Research and Fast Prototyping. In *13th Conference on Innovative Data Systems Research, CIDR 2023, Amsterdam, The Netherlands, January 8-11, 2023*. www.cidrdb.org. https://www.cidrdb.org/cidr2023/papers/p41-haffner.pdf

[10] Immanuel Haffner, Marcel Maltry, Joris Nix, Jens Dittrich, and Luca Gretscher. 2023. mu*t*able. https://mutable.uni-saarland.de

[11] Waqar Hasan. 1996. *Optimization of SQL queries for parallel machines.* Ph.D. Dissertation. Stanford University, USA. https://searchworks.stanford.edu/view/3131424

[12] Richard Hipp. 2024. SQLite: How It Works. Retrieved October 24, 2024 from https://sqlite.org/talks/howitworks-20240624.pdf#page=100

[13] Toshihide Ibaraki and Tiko Kameda. 1984. On the Optimal Nesting Order for Computing N-Relational Joins. *ACM Trans. Database Syst.* 9, 3 (1984), 482–502. https://doi.org/10.1145/1270.1498

[14] 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

[15] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter A. Boncz, Alfons Kemper, and Thomas Neumann. 2015. How Good Are Query Optimizers, Really? *Proc. VLDB Endow.* 9, 3 (2015), 204–215. https://doi.org/10.14778/2850583.2850594

[16] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Nesime Tatbul, Mohammad Alizadeh, and Tim Kraska. 2020. Bao: Learning to Steer Query Optimizers. *CoRR* abs/2004.03814 (2020). arXiv:2004.03814 https://arxiv.org/abs/2004.03814

[17] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Chi Zhang, Mohammad Alizadeh, Tim Kraska, Olga Papaemmanouil, and Nesime Tatbul. 2019. Neo: A Learned Query Optimizer. *Proc. VLDB Endow.* 12, 11 (2019), 1705–1718. https://doi.org/10.14778/3342263.3342644

[18] Guido Moerkotte. 2023. *Building Query Compilers.* https://pi3.informatik.uni-mannheim.de/~moer/querycompiler.pdf

[19] Guido Moerkotte and Thomas Neumann. 2006. Analysis of Two Existing and One New Dynamic Programming Algorithm for the Generation of Optimal Bushy Join Trees without Cross Products. In *Proceedings of the 32nd International Conference on Very Large Data Bases, Seoul, Korea, September 12-15, 2006*, Umeshwar Dayal, Kyu-Young Whang, David B. Lomet, Gustavo Alonso, Guy M. Lohman, Martin L. Kersten, Sang Kyun Cha, and Young-Kuk Kim (Eds.). ACM, 930–941. http://dl.acm.org/citation.cfm?id=1164207

[20] Guido Moerkotte and Thomas Neumann. 2011. Accelerating queries with group-by and join by groupjoin. *Proc. VLDB Endow.* 4, 11 (Aug. 2011), 843–851. https://doi.org/10.14778/3402707.3402723

[21] Parimarjan Negi. 2023. CEB GitHub. Retrieved January 9, 2025 from https://github.com/learnedsystems/CEB

[22] Parimarjan Negi, Ryan Marcus, Andreas Kipf, Hongzi Mao, Nesime Tatbul, Tim Kraska, and Mohammad Alizadeh. 2021. Flow-Loss: Learning Cardinality Estimates That Matter. *Proc. VLDB Endow.* 14, 11 (2021), 2019–2032. https://doi.org/10.14778/3476249.3476259

[23] 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

[24] Thomas Neumann and Guido Moerkotte. 2004. A combined framework for grouping and order optimization. In *Proceedings of the Thirtieth International Conference on Very Large Data Bases - Volume 30* (Toronto, Canada) *(VLDB '04)*. VLDB Endowment, 960–971.

[25] Thomas Neumann and Guido Moerkotte. 2004. An Efficient Framework for Order Optimization. In *Proceedings of the 20th International Conference on Data Engineering (ICDE '04)*. IEEE Computer Society, USA, 461.

[26] Hung Q. Ngo, Ely Porat, Christopher Ré, and Atri Rudra. 2012. Worst-case Optimal Join Algorithms. *CoRR* abs/1203.1952 (2012). arXiv:1203.1952 http://arxiv.org/abs/1203.1952

[27] Kiyoshi Ono and Guy M. Lohman. 1990. Measuring the Complexity of Join Enumeration in Query Optimization. In *16th International Conference on Very Large Data Bases, August 13-16, 1990, Brisbane, Queensland, Australia, Proceedings*, Dennis McLeod, Ron Sacks-Davis, and Hans-Jörg Schek (Eds.). Morgan Kaufmann, 314–325. http://www.vldb.org/conf/1990/P314.PDF

[28] Oracle. 2025. MySQL GitHub. Retrieved December 5, 2024 from https://github.com/mysql/mysql-server/blob/trunk/sql/join_optimizer/interesting_orders.h#L33

[29] Greg Rahn and Max Halford. 2025. JOB GitHub. Retrieved January 9, 2025 from https://github.com/gregrahn/join-order-benchmark

[30] Patricia G. Selinger, Morton M. Astrahan, Donald D. Chamberlin, Raymond A. Lorie, and Thomas G. Price. 1979. Access Path Selection in a Relational Database Management System. In *Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data, Boston, Massachusetts, USA, May 30 - June 1*, Philip A. Bernstein (Ed.). ACM, 23–34. https://doi.org/10.1145/582095.582099

[31] Amir Shaikhha, Mohammad Dashti, and Christoph Koch. 2016. Push vs. Pull-Based Loop Fusion in Query Engines. *CoRR* abs/1610.09166 (2016). arXiv:1610.09166 http://arxiv.org/abs/1610.09166

[32] David Simmen, Eugene Shekita, and Timothy Malkemus. 1996. Fundamental techniques for order optimization. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data* (Montreal, Quebec, Canada) *(SIGMOD '96)*. Association for Computing Machinery, New York, NY, USA, 57–67. https://doi.org/10.1145/233269.233320

[33] Bennet Vance and David Maier. 1996. Rapid Bushy Join-order Optimization with Cartesian Products. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data, Montreal, Quebec, Canada, June 4-6, 1996*, H. V. Jagadish and Inderpal Singh Mumick (Eds.). ACM Press, 35–46. https://doi.org/10.1145/233269.233317

[34] Xiaoyu Wang and Mitch Cherniack. 2003. Avoiding sorting and grouping in processing queries. In *Proceedings of the 29th International Conference on Very Large Data Bases - Volume 29* (Berlin, Germany) *(VLDB '03)*. VLDB Endowment, 826–837.