Quantifying TPC-H Choke Points and Their Optimizations

Markus Dreseler, Martin Boissier, Tilmann Rabl, Matthias Uflacker
Hasso Plattner Institute
University of Potsdam, Germany
firstname.lastname@hpi.de

ABSTRACT

TPC-H continues to be the most widely used benchmark for relational OLAP systems. It poses a number of challenges, also known as "choke points", which database systems have to solve in order to achieve good benchmark results. Examples include joins across multiple tables, correlated subqueries, and correlations within the TPC-H data set. Knowing the impact of such optimizations helps in developing optimizers as well as in interpreting TPC-H results across database systems.

This paper provides a systematic analysis of choke points and their optimizations. It complements previous work on TPC-H choke points by providing a quantitative discussion of their relevance. It focuses on eleven choke points where the optimizations are beneficial independently of the database system. Of these, the flattening of subqueries and the placement of predicates have the biggest impact. Three queries (Q2, Q17, and Q21) are strongly influenced by the choice of an efficient query plan; three others (Q1, Q13, and Q18) are less influenced by plan optimizations and more dependent on an efficient execution engine.

PVLDB Reference Format:
Markus Dreseler, Martin Boissier, Tilmann Rabl, Matthias Uflacker. Quantifying TPC-H Choke Points and Their Optimizations. *PVLDB*, 13(8): 1206-1220, 2020.
DOI: https://doi.org/10.14778/3389133.3389138

1. MOTIVATION

TPC-H continues to be relevant for the evaluation of relational database systems. Hundreds of papers reference the benchmark each year. It is more widely discussed than its successor, TPC-DS, as shown in Figure 1. This is despite of its shortcomings, such as the linear scaling of non-fact tables, its homogeneous data distribution, the use of 3NF instead of a star schema layout (which is more typical for this type of analytical workloads [35]), or the non-comparability of its handcrafted queries with the auto-generated queries of actual workloads [56].

Proceedings of the VLDB Endowment, Vol. 13, No. 8
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3389133.3389138

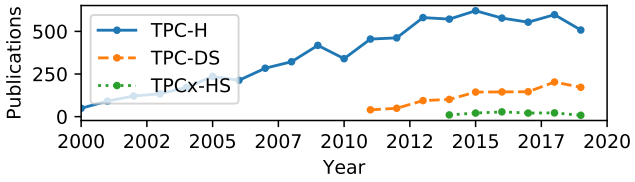


Figure 1: Number of publications indexed on Google Scholar referencing "TPC-H", "TPC-DS", or "TPCx-HS", starting with the year of the benchmark's publication. In the nine years after being published (1999-2007), TPC-H was referenced in 1 633 publications, while TPC-DS was only referenced 1 094 times in the respective nine-year frame.

While TPC-H was originally designed to compare database systems end-to-end, researchers also use it to benchmark implementation details and algorithms. Higher throughput numbers are taken as "proof" that the chosen approach is better. This may be true when all other components are identical, but more often than not, comparisons use different database systems and better throughput numbers are (at least implicitly) taken as support for the presented approach. In this case, the influence of other optimizations that differ between the systems is often underestimated [46].

For example, when looking at the performance of our research DBMS Hyrise on TPC-H Q6, Hyrise is almost seven times faster than MonetDB. As Q6 is scan-heavy, one might argue that Hyrise is better at scanning. However, that performance benefit is achieved by an optimization that avoids access to 82% of the input data (cf. Section 4.5). Without this optimization, Hyrise is only 1.8× faster. While the former comparison might be more interesting from a marketing perspective, we consider the latter to be more accurate when discussing the performance of a scan-heavy query.

To allow for a better comparison of TPC-H results between systems, we believe that a discussion of these optimizations should be part of their evaluation. In 2013, Boncz, Neumann, and Erling presented an analysis of the "choke points" within TPC-H, i.e., of the challenges a DBMS needs to address to efficiently execute the benchmark [11]. They define choke points to be the "technological challenges underlying a benchmark, whose resolution will significantly improve the performance of a product". For TPC-H, 28 choke points were identified and grouped into challenges for the aggregate and join operators, the locality of data accesses,

the evaluation of expressions, the handling of correlated subqueries, and the parallel execution of the benchmarks.

Their work serves as an excellent reference when describing the different optimizations made in a given DBMS. While the choke points were described in great detail, no benchmark numbers were given, making it difficult to estimate their impact. This paper provides such a quantitative evaluation. For new database systems, this knowledge helps in deciding which choke points to focus on first. While a full-fledged commercial DBMS cannot forgo any of these optimizations, developers of purpose-built research systems often have to focus their resources. Knowing the relevance of different optimizations enables a more educated prioritization. Furthermore, when comparing different systems, knowing how much an optimization or the lack thereof affects the final results helps in comparing the systems. Finally, a quantification of the choke points enables benchmark designers to compare their relevance to real-world workloads and adapt the benchmarks accordingly.

With this paper, we make the following contributions: (1) For eleven choke points, we provide analyses of their impact. We focus on choke points that optimize the query plan on a logical level and, as such, can be studied mostly independently of the execution engine and the scheduling model. The analyses show that three choke points have an impact of one order of magnitude or more for affected queries while for two others even the biggest impact is 25% or less. (2) We share our experiences when implementing these optimizations and provide pointers into open-source code both as a reference for their implementation and to enable the reproduction of our results. (3) We describe optimizations that have not yet been listed as TPC-H choke points, namely semi join reductions and between compositions.

This paper is organized as follows: In Section 2, we classify the choke points depending on which step of the query execution is affected. Our experimental setup is described in Section 3, where we describe Hyrise as the DBMS being used in this evaluation, compare its performance to other database systems, and describe the methodology with which we isolate and evaluate the choke points. The evaluation is split into Section 4, which covers the choke points that affect the structure of the query plan, and Section 5, where logical choke points within the operators are covered. In Section 6, we compare the relative impact of all choke points and share our learnings from implementing the related optimizations. Related work covering the areas of TPC-H optimizations and benchmarks outside of TPC-H is discussed in Section 7.

2. CHOKE POINT CATEGORIZATION

We group the choke points into three categories: First, *plan-level choke points* affect the cardinalities of the intermediary tables early on. This includes join ordering, predicate pushdown and ordering, as well as the flattening of subqueries. The cardinality reductions achieved by these optimizations are on the logical/relational level.

Second, *logical operator choke points* are those that affect the plan on the level of a single operator. While the input and output cardinalities of the operator remain unchanged, its logical efficiency is improved. An optimization in this group is to remove group-by columns that are functionally dependent on other group-by columns. Doing so reduces the logical complexity of the aggregate.

Finally, for *implementation-specific choke points*, the query plan is unchanged, but the physical efficiency of operators is improved. An example of this is to add bloom filters to joins or to replace the regular expressions used for `LIKE` expressions with a more efficient Boyer-Moore matcher.

We focus on the logical optimizations, i.e., the first two groups, as these are more comparable across database systems. Hyrise also addresses choke points on the operator level and uses optimizations such as SIMD, compressed execution, and bloom filters. These are enabled throughout all experiments but not discussed in this paper.

All systems profit from accessing fewer tuples, independently of whether the system is row- or column-oriented, in-memory or disk-based. For columnar in-memory systems, the absolute numbers are expected to be similar to ours. In the case of disk- and row-based systems, optimizations at the early stages of a plan, such as physical access avoidance, that reduce the number of tuples read from disk, are expected to have a bigger impact.

Our grouping differs from that used in Boncz et al. in that we use the scope of the choke point (plan-level or operator-level) and the difference between logical or physical optimizations as the primary criteria, while they group their choke points by the affected operator(s). A mapping between the choke point categorizations is given in Table 3.

3. EXPERIMENTAL SETUP

Our evaluation is based on the research DBMS Hyrise [17]. We chose this system because we are deeply familiar with it and know where to turn off specific optimizations as well as how doing so affects other components. As such, it allows us to perform more in-depth analyses than a high-level comparison of different database systems would. In this section, we give an introduction to the architecture of Hyrise, limited to the key points that become relevant in the remainder of the paper. For a more detailed explanation of Hyrise, we refer to our previous work [17]. We then benchmark Hyrise against other in-memory database systems. We show that the baseline performance is comparable, which gives us the confidence that the choke point analyses are not weakened by fundamental performance bottlenecks. Finally, we describe the methodology with which the choke points are evaluated.

3.1 Characterization of Hyrise

Hyrise[1] is a columnar in-memory DBMS that serves as a platform for research projects [6, 22, 50] in the area of enterprise data management. One of the common goals of these is to evaluate new concepts in an actual DBMS. As such, we are spending significant time building a system that can execute both the most common benchmarks as well as enterprise workloads [7]. We do not rely on hand-optimized query plans tailored to TPC-H, but make sure that the optimizations presented here also apply outside of TPC-H.

Tables are horizontally partitioned into *chunks* that hold 100 000 rows. We have experimentally determined this number to be efficient for TPC-H [17]. Chunks serve as a unit for parallelization, compression, and have their own statistics. In that, the best comparison would be to HyPer's morsels [28], which are also partitions of the table data based on the number of rows and are used for dynamic placement and parallelization. Interestingly, the optimal size for

[1]C++ code at `https://github.com/hyrise/hyrise`

morsels was also determined to be 100 000. In Hyrise, a chunk holds one *segment* per column of the table. As a result, all tables are stored in a column-oriented fashion. These segments can be compressed using the dictionary, run-length, LZ4, and frame of reference compression schemes. In this paper, dictionary encoding is used.

The translation from SQL to operator plans follows a common pattern: SQL strings are parsed using our flex/bison-based SQL parser[2]. The parsing result is translated into a logical query plan (LQP), which is then transformed by the optimizer. Next, the LQP is translated into a physical query plan (PQP), which contains the actual operators.

Optimizations convert the LQP from one consistent state into a more efficient one. Some optimizations are exclusively rule-based, such as the one-time calculation of constant arithmetic expressions. Others are cardinality-based, such as the join ordering algorithm. For this, histograms with bins of equal distinct counts (on a table level) and MinMax filters (on a chunk level, cf. [31]) are used.

Operators are executed one table at a time. With the exception of aggregates and projections, they produce position lists, which are indirections into the original table [1]. We will shortly discuss the operators that are most relevant for TPC-H: Table scans are vectorized using AVX-512 for a faster generation of the resulting position lists. Projections calculate results by interpreting the expression tree on materialized inputs. For joins, Hyrise uses a single-pass radix-partitioned hash join (cf. [10]). While a sort-merge based join operator exists, the implementation of the hash join is more efficient. The sort operator uses a stable sort algorithm and does multiple passes for sorting a table on multiple columns. Similar to the join, the evaluations in this paper use only a hash-based aggregate operator. Finally, Hyrise follows the insert-only principle where rows are never updated, only invalidated and re-inserted with new values. The MVCC validate operator is responsible for removing rows that are invisible to the current transaction [51, 57].

The relative cost of these operators is visualized in Figure 2. For each query, the time spent in the respective operator is tracked and plotted as a percentage of the entire execution time. Operators that take less than 1% of the queries' execution time are excluded. This hides five operators that do not contribute significantly to the run time. For this reason, not all bars add up to 100%. The right-most bar represents the arithmetic mean of their relative costs across all executions, meaning that all queries are weighted the same, independently of their absolute execution time.

[2] `https://github.com/hyrise/sql-parser`

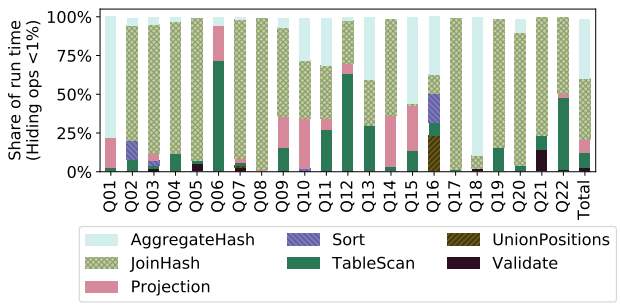


Figure 2: Relative time spent in the different operators. UnionPositions is an operator that unifies the results from disjunctive predicates. The table scan is over-represented in Q11 and Q22, as it includes the cost of evaluating a subquery that was intentionally not flattened.

3.2 Preliminary Evaluation

In this paper, we aim at neutrally evaluating the benefits and drawbacks of certain TPC-H optimizations. At the same time, we believe that this evaluation is only meaningful if the test system has a reasonable baseline performance. For Figure 3, we compared Hyrise to four other systems. DuckDB [47], MonetDB [9] and Umbra [38] were evaluated on our own hardware. The numbers for HyPer [25], which is not open-source, were taken from the Flare paper [19], but were measured on comparable hardware. Umbra, the SSD-enabled "evolution of HyPer", was benchmarked using a pre-release prototype and the included TPC-H demo script. The benchmarks on MonetDB were executed using their provided tpch-scripts[3]. For DuckDB, we used the included benchmark, which was slightly modified to allow for a scale factor of 10. These systems were chosen not as direct competitors but because of their active development, the variety of chosen approaches, and the ease of obtaining TPC-H numbers. Single-threaded execution is not the default mode for most systems and these results cannot be used to establish a ranking.

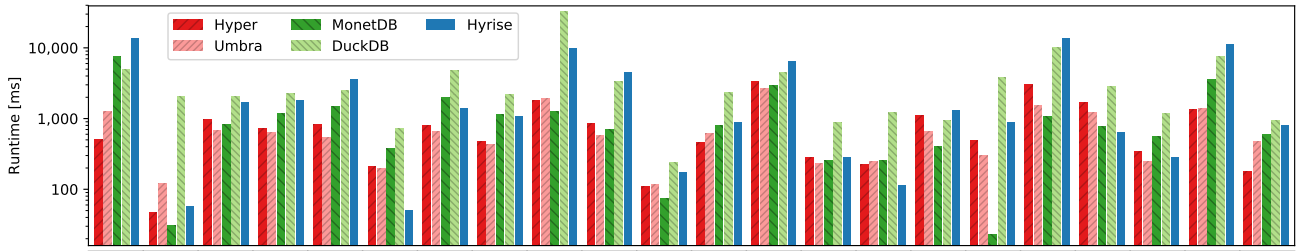
The partition pruning (cf. Section 4.5) in Hyrise allows it to skip more than 80% of the data in Q6 and Q20, making it the fastest DBMS for those queries. On the other hand, the hash-based aggregate operator in Hyrise is relatively slow, leading to the worst result across all systems in Q1 and Q18.

3.3 Methodology

For all choke points, we modified Hyrise and removed the optimizations in question. Depending on the choke point, this was achieved by removing a plan optimization or removing optimizing code within an operator. We have then re-executed the benchmark. The throughput difference between the binaries with and without optimizations is reported as the optimization's benefit. Optimizations that were unrelated to the choke point were left in place. In some cases, optimizations are dependent on each other and the lack of one optimization might make a different optimization cause performance regressions. In these cases, we temporarily disabled the following optimizations, too.

For the aforementioned groups of choke points, the impact of cardinality-reducing optimizations is independent of the degree of parallelism. For example, the cardinality-reducing effects of a join ordering algorithm are independent of the number of threads and the effort required of the join will be the same. However, parallelism introduces a number of additional influence factors: The hardware (more specifically, the number of CPU cores and the NUMA layout) becomes more important, the number of parallel query streams either favors inter- or intra-query parallelism, and orthogonal optimizations such as scan and join sharing [24] become relevant. In the interest of an in-depth discussion of the selected choke points, these dimensions are excluded from the scope of this paper. We thus limit the evaluation to choke points that apply to single- and multi-threaded executions alike.

[3] `https://github.com/MonetDBSolutions/tpch-scripts`



	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10	Q11	Q12	Q13	Q14	Q15	Q16	Q17	Q18	Q19	Q20	Q21	Q22
Hyper	**498**	47	969	725	821	207	804	467	1,782	846	111	**460**	3,307	283	227	1,112	485	3,061	1,666	345	**1,347**	**177**
Umbra	1,258	119	**676**	**629**	**536**	196	**665**	**435**	1,938	**574**	116	617	**2,645**	**229**	245	653	301	1,538	1,216	**247**	1,386	477
MonetDB	7,488	**31**	816	1,185	1,505	372	1,965	1,136	**1,252**	708	**74**	786	2,970	258	259	**403**	**23**	**1,054**	765	560	3,535	605
DuckDB	4,874	2,083	2,080	2,283	2,533	723	4,832	2,217	32,681	3,377	241	2,363	4,466	887	1,232	925	3,837	10,039	2,807	1,192	7,452	939
Hyrise	13,559	57	1,719	1,796	3,579	**50**	1,404	1,078	9,740	4,496	173	882	6,388	282	**113**	1,318	894	13,628	**642**	282	11,104	794

Figure 3: TPC-H query execution time in milliseconds compared across different database systems. Single-threaded, SF 10. For HyPer, which is not available publicly, numbers are taken from Essertel et al. [19]. Umbra, MonetDB, DuckDB, and Hyrise were measured using TPC-H benchmark facilities as provided by their respective authors on hardware that is comparable to that used for the HyPer benchmarks.

The system used is a 2017 Fujitsu Primergy RX4770 M4 with four Xeon 8180 CPUs (2.5 GHz base, 3.8 GHz turbo, 28 physical cores). Benchmarks were bound to a single node using `numactl -N 0 -m 0`. Each NUMA node has 512 GB of DDR4-2666 memory, distributed over 8 DIMMs. This results in a combined read/write throughput of 20 553.32 MB/s for a single core and a 2:1 read/write ratio. Hyrise was compiled with GCC 9.2[4] and `-O3 -march=native`.

TPC-H was executed using the `hyriseBenchmarkTPCH` binary, which automatically generates the data and the queries, executes the benchmark, and stores the results. To measure the impact of optimizations on individual queries, the queries were executed for 60 seconds each, after which the last run was allowed to finish. We report how the number of queries executed in that time frame has changed as a result of the optimization. The used scale factor was 10. For larger scale factors, disabling optimizations such as subquery flattening or predicate pushdown results in execution times of more than an hour for a single execution of a single query.

In Hyrise, the discussed optimizations have to be disabled in the source code. For each experiment, patch files were applied to the Hyrise source code. These are provided as a starting point for reproduction and exploring the optimizations' implementation[5]. Instructions for compilation, execution, and visualization are included.

4. PLAN-LEVEL CHOKE POINTS

The first group of choke points contains those that affect the entire query plan by reducing the cardinalities early on. These are beneficial for all systems, as an operator that has to work on fewer rows will be faster than one working on a high number of rows, even if the latter is better optimized. We start with the order of joins in the query plan, as changing this order is one of the most significant structural changes to the entire plan. Next, we look at the position and order of single-table predicates and show how pushing these further down the query plan improves performance. After that, we discuss how clustering the data does not only help with executing these predicates faster, but also allows for entire partitions to be pruned, which reduces the cardinality of a table even before the first operator is executed. Next, we look at how the twelve queries that use subqueries can be reformulated to use a more efficient query plan. Finally, we look how reusing results across query plans and across query executions affects the TPC-H performance.

4.1 Join Ordering

All but two queries (Qs 1 and 6) operate on multiple tables and most queries operate on three tables or more. In most cases, join predicates are not explicitly stated as `t1 JOIN t2 ON ...`, but implicitly as `FROM t1, t2 WHERE t1.a = t2.a`. The first responsibility of a join ordering algorithm is to match unpredicated cross joins and their predicates. Without this step, only seven of the 20 join queries, which would otherwise run in the scale of milliseconds, finish within a minute (Qs 4, 13, 15, 18, 20, 21, and 22). The other 14 queries are virtually unexecutable. As such, we do not consider join predicate matching an optimization, but an indispensable feature. It is not part of our evaluation.

The second responsibility of the join ordering algorithm is to determine the order in which tables are joined. Both steps can be handled by a join ordering algorithm that transforms the LQP to a join graph [42], performs the join ordering on that graph, and rebuilds an optimized LQP.

For the experiment in Figure 4, three join orders are compared. The baseline is the join order as defined by the SQL query. The two bars show the improvement of the dynamic

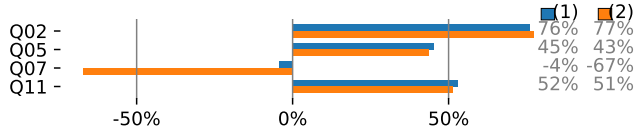


Figure 4: Benefit of (1) DPccp and (2) greedy join ordering compared to the order as defined in the original SQL queries. Queries with a change <10% are omitted in this and following graphs.

[4]For Hyrise, GCC performs consistently better than clang; GCC 9.2 binaries have a TPC-H throughput that is 11% higher than that of clang 9.

[5]`https://github.com/hyrise/tpch_paper`

programming-based DPccp [32] and a bottom-up greedy join ordering algorithm [20]. Three queries (Qs 2, 5, and 11) are improved significantly. Q7 is the only query where DPccp and the greedy algorithm produce a different join order. The greedy algorithm joins the unfiltered customers and orders table, destroying the join order that was initially optimal. DPccp first reduces the size of the customers table by joining it with the nation table and reducing the list of customers to those from the two queried nations. This is equivalent to the initial order; the performance hit of 4% is thus caused by the cost of join ordering itself.

Neumann et al. found that the difference between the worst and best execution time of Q5 is more than a factor of 100 [37]. When executing Q5 with an inverse-greedy join ordering (i.e., picking the two biggest tables first), Hyrise ran out of memory, as it did for Q7. As such, we can validate this finding. All other queries were not affected as significantly.

While some workloads, such as TPC-DS, require sophisticated join ordering algorithms and complex statistics, we find that a greedy join ordering algorithm and very simple (min/max/count/distinct count) statistics are sufficient for all but Q7. For implementors in the early phase of development, we thus suggest to focus the development effort on other optimizations before starting to implement more complex join ordering algorithms such as DPccp.

4.2 Predicate Pushdown and Ordering

A primary goal of query optimization is to reduce cardinalities as early as possible. The easiest way to achieve this is to execute the most selective predicates first. This consists of two steps that operate on a plan where conjunctive predicates have already been separated: First, *Predicate Pushdown* moves predicates past joins, aggregates, and projections. Hyrise uses a recursive implementation, which starts at the top of the LQP, removes predicates from the query plan as they are encountered and re-inserts them once they cannot be pushed any further. Second, for *Predicate Ordering*, multiple predicates that share a common final position are reordered with regards to their selectivity so that the most restrictive predicate is executed first. To estimate these costs, Hyrise uses the columns' histograms. More sophisticated algorithms might also take the execution time of the predicate itself into account, which might be affected by the data type of the scanned column, the length of string columns, or the column's compression method.

While database systems that compile multiple successive predicates on different columns into a single operator do not need to materialize their intermediary results [18], they still benefit from evaluating the most restrictive predicate first. This is because they can use short-circuit evaluation and skip the load of the values from the other columns.

For Figure 5, the baseline is a query plan where all predicates are executed at their original position in the SQL query. Joins operate on unfiltered tables and aggregates calculate groups that are later excluded from the result. Subqueries are flattened (Section 4.7) but not reordered. The first bar shows the effect of predicate pushdown, the second bar that of both predicate pushdown and reordering.

Most queries profit from a better predicate placement. In extreme cases, predicate pushdown reduces cardinalities by orders of magnitude. In Q14, the predicate on `l_shipdate` removes more than 98% of the rows in the lineitem table, which is then joined with the unfiltered part table.

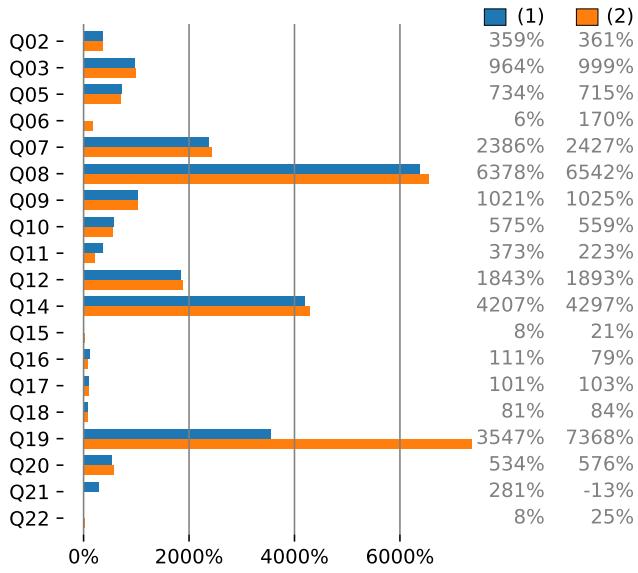


Figure 5: Benefits of (1) Predicate Pushdown and (2) Predicate Ordering.

For TPC-H, predicate pushdown can be done entirely rule-based, without any statistics being required. As such, it is among the first optimizations that should be implemented.

4.3 Between Composition

In Q19, a range predicate on `l_quantity` is expressed using `>=` and `<=` predicates. Qs 4, 5, 6, 10, 12, 14, and 15 filter by a date range using `>=` and `<` predicates. This is unavoidable as SQL only supports inclusive between predicates. Identifying these cases and rewriting them into left- or right-exclusive between predicates helps with optimization. For example, `o_orderdate >= '[DATE]' AND o_orderdate < '[DATE]' + '1' year` in Q5 can be rewritten into `[date , date + 1 year)` (left inclusive, right exclusive).

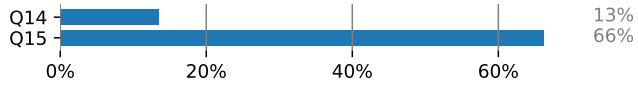


Figure 6: Qs 14 and 15 profit from rewriting lower and upper bounds into a single range predicate.

The DBMS can benefit from this optimization in two ways. First, for database systems that produce intermediary results after each operator, this step can be avoided if one operator is executed instead of two. Second, the combined predicate will likely be more restrictive than the two individual predicates. As such, it will be placed lower in the query plan, reducing the cardinality earlier.

Figure 6 shows that only Qs 14 and 15 profit noticeably. In the other queries, the between predicate comes too late in the query plan to still make a noticeable difference.

4.4 Join-Dependent Predicate Duplication

Two queries, Q7 and Q19, include predicates that operate on multiple tables without being a join predicate. In Q17, `(n1.name = 'NATION1' AND n2.name = 'NATION2')OR (n1.name = 'NATION2' AND n2.name = 'NATION1')` uses

n1 and n2 and thus cannot be pushed below the join. This means that the join is first performed on all input rows even though only rows that belong to two of 25 nations can qualify. By extracting the condition `n_name = 'NATION1' OR n_name = 'NATION2'` and adding it to both inputs of the join, the input cardinality, and thus, the cost of the join can be reduced. This is not a predicate pushdown as the original predicate is still needed after the join to reject tuples where both nation names are identical. Figure 7 shows the results of this optimization for both affected queries.

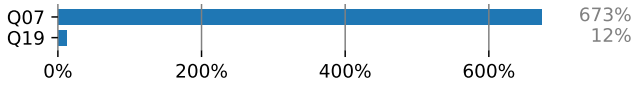


Figure 7: In Qs 7 and 19, additional predicates should be created to filter input tables before they reach the join.

It makes sense to make this optimization part of the predicate pushdown, even if it does not actually perform a pushdown. This is so that (a) predicates that cannot be pushed down below a join can be identified together with the corresponding join and (b) the newly created, additional predicates can be part of the pushdown on the join's input sides.

4.5 Physical Locality

The lineitem table is filtered by its `l_shipdate` attribute in seven out of 22 queries, more than on any other attribute. Similarly, the orders table is filtered on the `o_orderdate` attribute in five queries. Clustering the tables on these attributes enables the scan to use a binary search instead of a linear search, reducing the logical cost of the scan. At the same time, the clustering can be used to reduce the cardinality even before the scan: Ranges of the tables where no tuple qualifies can be identified and skipped entirely. This can be based on partition criteria, range-based statistics (e.g., small materialized aggregates [31], zone maps [58]), or, in the case of Hyrise, MinMax statistics on the chunk level. `l_shipdate` contains values within an 83-month frame. For Q6, which selects line items that were shipped within a twelve-month frame, clustering allows to exclude up to 85% of the data without touching a single row.

For Figure 8, the data as generated by tpch-dbgen serves as the baseline. Three comparative benchmarks were executed. First, the lineitem and orders tables were shuffled, making sure that not even the default order can be exploited. Second, the tables were clustered by `l_shipdate` and `o_orderdate`, but the DBMS could not exploit this information, meaning that partition pruning and early exits in operators were disabled. Third, the same clustering was used and the DBMS was allowed to use partition pruning[6].

Compared to the original order, shuffling the data (first bar) has a negative impact of 10% or more on twelve queries. This is because the data as generated by tpch-dbgen is not in a random order, but sorted by the primary key. This is beneficial for joins, as it either reduces the cost of the sort phase (for sort-based joins) or improves the physical locality (i.e., the number of DRAM cache hits) of the hash table lookups (for hash-based joins).

[6]In all cases, we exclude the detection of correlations, which will be described as the following choke point.

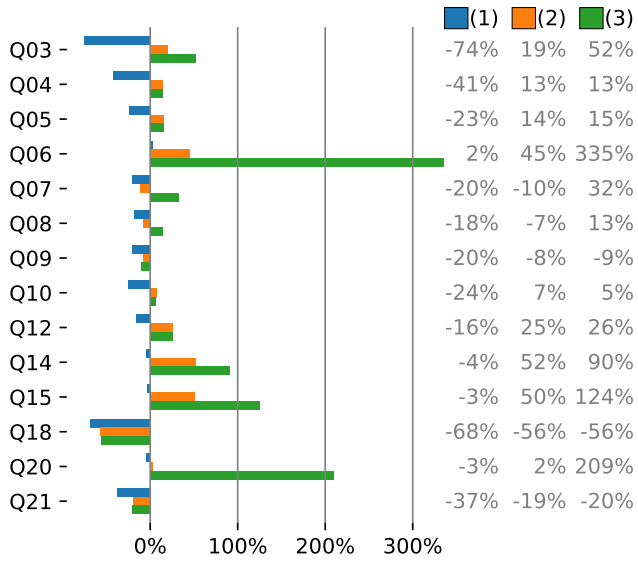


Figure 8: Performance impact of different clustering attributes with and without access avoidance. The baseline uses the tables as generated by tpch-dbgen. The tables lineitem and orders are (1) shuffled, (2) clustered by `l_shipdate` and `o_orderdate` without exposing that information to the DBMS, and (3) clustered with exposing the information, enabling access avoidance and sorted scans.

Clustering the data (second bar) without allowing the DBMS to make use of this fact has an ambivalent effect. Most notably, in Q18, the unfiltered group-by on `l_orderkey` suffers from reduced physical access locality. On the other hand, queries that filter on `l_shipdate` profit from the clustering even if it is not exposed to the DBMS. This is because of better CPU cache locality and a lower ratio of branch mispredictions [49]. Across all queries, the effects even out.

Once the DBMS is allowed to take the clustering into account (third bar), the benefits of physical locality and access avoidance outweigh the reduced join and aggregate performance. Excluding 85% of data in Q6 more than triples its throughput. This is especially notable as join and aggregate operations are the biggest cost factor for Hyrise when executing TPC-H (cf. Figure 2).

When compared to similar evaluations [26, 30], Hyrise spends less time on the scans (where partition pruning saves it from performing the actual scan) and projections, but disproportionately much on joins and aggregates. This is caused by the implementation of these operators materializing more data than necessary.

4.6 Correlated Columns

Clustering the lineitem table on `l_shipdate` does not only improve accesses on that column, but also helps in exploiting correlations between columns. For example, the values in `l_shipdate` and `l_receiptdate` are always apart by a maximum of 30 days. When the table is clustered by `l_shipdate`, the first segment holds values from `1992-01-02` to `1992-04-08`. Because of the relationship between the two columns, `l_receiptdate` is now between `1992-01-04` and `1992-05-08` (which is the highest ship date plus 30 days).

Figure 9: Allowing the DBMS to exploit the correlation between a clustered and an unclustered column improves the performance of predicates on `l_receiptdate` and `l_returnflag`.

Table 1: Pruning ratios based on a chunk size of 100 000 rows. Numbers vary slightly depending on the random query parameters.

Q	lineitem	orders	Q	lineitem	orders	Q	lineitem	orders
1	3 / 600	n/a	9	0 / 600	0 / 150	17	0 / 600	n/a
2	n/a	n/a	10	300 / 600	143 / 150	18	0 / 600	0 / 150
3	277 / 600	76 / 150	11	0 / 600	0 / 150	19	0 / 600	n/a
4	0 / 600	143 / 150	12	501 / 600	0 / 150	20	508 / 600	n/a
5	0 / 600	126 / 150	13	n/a	n/a	21	0 / 600	71 / 150
6	508 / 600	n/a	14	591 / 600	n/a	22	n/a	0 / 150
7	417 / 600	0 / 150	15	576 / 600	n/a			
8	0 / 600	104 / 150	16	n/a	n/a	avg	205 / 600	55 / 150

Scans that probe only for values outside of that range may skip the first chunk. Note that this is possible even though the table is not explicitly clustered by `l_receiptdate`.

While the TPC-H specification prohibits giving that knowledge explicitly to the database, the DBMS is allowed to identify these correlations and exploit them. This enables the system to propagate pruning information from one column to another. In Hyrise, this information is retrieved from the respective segment's MinMax statistics.

An additional correlation exists between `l_shipdate` and `l_returnflag`. The return flag is defined to be randomly 'R' or 'A' if `l_receiptdate <= '1995-06-17'` and 'N' otherwise. As the range of potential receipt dates is correlated with the range of ship dates within a chunk, this means that for chunks where `l_shipdate > '1995-06-18'`, a search for `l_returnflag = 'R'` will return no results. The effect of this can be seen in Q10, where accesses to half of the lineitem table can be avoided. The columns `o_orderstatus` and `l_linestatus` are correlated with the clustered date columns, but not used by the queries in a way where the DBMS can profit from this correlation.

We have shown the benefits of clustering and access avoidance in Section 4.5. For the previous choke point, we have explicitly prohibited the DBMS from exploiting pruning information on columns without an explicit clustering. Figure 9 shows that removing this restriction has a notable positive impact on Q10 (which filters on `l_returnflag`) and Q12 (which filters on `l_receiptdate`).

Summarizing the last two choke points, TPC-H is extremely amenable for range-based access avoidance. On average, when a query accesses the lineitem or orders table, a third of data in that table does not have to be accessed, see Table 1. These results match those found by Nica et al. [41]. This is a pattern that also occurs in real-world workloads. Naturally, the dates of orders and their shipping date will be increasing during the lifetime of the database and both transactional and analytical workloads tend to access recent data more frequently. We have witnessed this in the ERP system of a Global 2000 system [7]. SAP has even reported an improvement of 2 to 3 orders of magnitude reached by using partition pruning on a central finance system [41].

Recently, it has been suggested that these correlations can also be exploited across joins [43]. After local partition pruning, the statistics of the pruned partitions can be used to create a new predicate over the join columns, which is then pushed across the join. The authors claim that for half of the queries, a third of all data accesses can be avoided.

From a benchmarking perspective, we question the missing correlation between the primary keys and the dates. For us, it seems plausible that orders are entered into the database roughly in the order of their order date. As such, auto-generated primary keys can reasonably be expected to be in increasing order, too. This would save benchmark implementors from having to choose between exploiting physical locality in the joins or in the predicates.

4.7 Flattening Subqueries

Six TPC-H queries use correlated subqueries (Qs 2, 4, 17, 20, 21, 22). The easiest way to implement these is to follow the semantics of this language construct: "Each <subquery> in the <search condition> is effectively executed for each row of T and the results used in the application of the <search condition> to the given row of T. If any executed <subquery> contains an outer reference to a column of T, then the reference is to the value of that column in the given row of T." [2, page 7.6]. In fact, we found it helpful to implement this approach first and gradually work towards more effective query plans. One of the most common approaches to improve these subqueries is to remove the comparisons with outer values from the subquery, which causes the subquery to return a table containing the values for *all* instances of these outer values. This table is then joined onto the outer table using the previously removed predicates as the join conditions. Neumann et al. call this "Simple Unnesting" [39]. The approach applies to `(NOT) IN` and `(NOT) EXISTS` as well as scalar subqueries and differs only in the selection of the join predicate. Additionally, it can be used to flatten `(NOT) IN` expressions that operate on uncorrelated subqueries (cf. Section 5.2).

An optimization that we recently became aware of and that is not yet implemented in Hyrise is the coalescence of different subquery types [4], which would apply to Q21. Other subquery flattening approaches can cover additional quantifiers such as `ALL` queries [23] or cases where the correlated attribute is not immediately available in the outer query [39] but were not found to be necessary for TPC-H.

This is the choke point with the biggest impact. For six queries, it reduces the runtime by two orders of magnitude or more as shown in Figure 10. Qs 2, 17, and 20 profit from

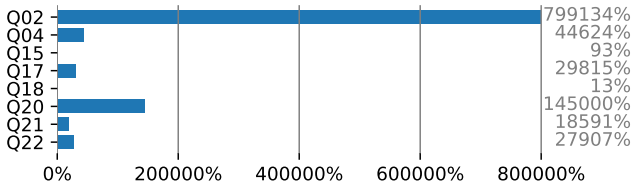


Figure 10: Flattening all correlated subqueries is one of the most important optimizations, as it keeps the DBMS from having to execute the subquery once per input row. Its impact is dependent on the cardinality of the outer relation.

correlated scalar subqueries being flattened while Qs 4, 21, and 22 use correlated `(NOT) EXISTS` subqueries. In Q18, the benefit comes from replacing an uncorrelated IN expression with a join. While Q15 does not have a correlated subquery, flattening the scalar subquery enables the optimizer to reuse subplans (cf. Section 4.9).

4.8 Semi Join Reduction

Part of flattening the subqueries was to remove the correlated predicate from the subquery. In Q17, this means that `0.2 * AVG(l_quantity)`, which was previously calculated once for each of the part × lineitem combinations that qualified on `p_brand` and `p_container` predicates, is now calculated on the entire lineitem table. Of the resulting rows, less than a percent is used.

The goal of the optimizer should be that the average quantity is calculated only for those partkeys that are relevant for the following join. For this, the Hyrise optimizer adds a semi join [5] in front of the aggregate and removes all lineitem rows that will not find a join partner in the following join. This is visualized in Figure 11.

Stocket et al. describe these semi join reductions as follows: "[A]pply join predicates early in a plan in order to reduce the size of intermediate query results and, thus, reduce the cost of other operations. In other words, [...] apply the same join predicates twice or more often in a query plan" [54]. This plan-level optimization is orthogonal to join optimizations in the execution engine, such as bloom filters. We ran experiments that showed that the combination of semi join reductions and join bloom filters result in a better performance than the two approaches on their own. Bloom filters bring their own optimization challenges, such as choosing an appropriate accuracy, which are outside the scope of this paper.

In TPC-H, another use case can be found in Q4, which filters the lineitem table for rows where `l_commitdate < l_receiptdate`. This reduces its cardinality by less than 40%. It is then joined with the orders table, which was filtered on `l_orderdate`. Less than 5% of the orders qualify. Adding a semi join reduction reduces the input to the predicate on the lineitem table accordingly.

Figure 12 shows performance improvements for three queries when the tables used in the subquery are filtered based on external predicates (Qs 2, 17, and 20) as well as two queries where the cardinality is reduced by removing rows

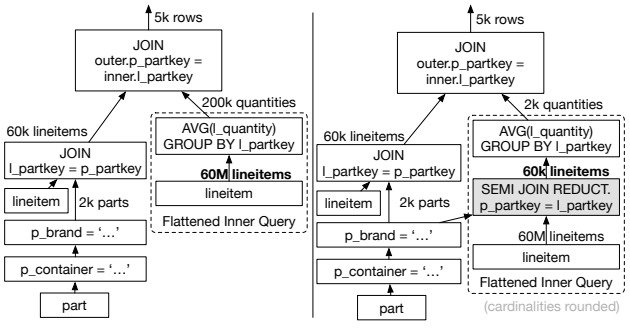


Figure 11: Comparison of the flattened query plan for Q17 without (left) and with (right) semi join reductions. The input cardinality for the aggregate operator is reduced by a factor of 1 000.

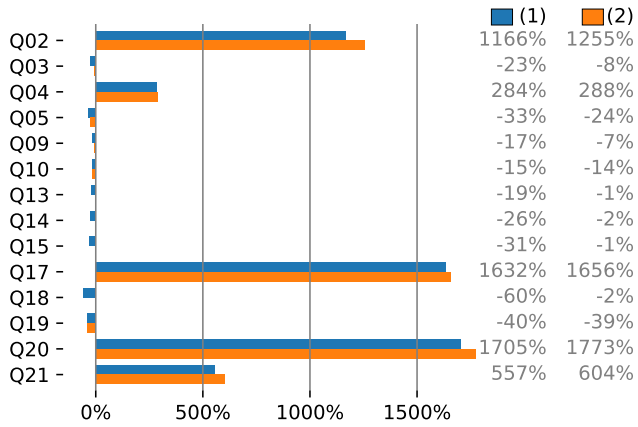


Figure 12: Semi joins remove rows that will not find a join partner in a later join. Adding them improves five queries. If they are added to all joins (1), their selectivity is often insufficient to justify their cost. Selectively adding them based on cardinality estimations (2) improves the overall results.

that will not qualify for the following join (Qs 4 and 21). When the semi join reduction is applied indiscriminately (first bar), nine queries have performance regressions of 10% or more. When the decision on whether to add a semi join reduction is based on the input tables' cardinalities, this regression is fixed for six queries. The regression for the remaining queries is not a conceptional issue, but the result of suboptimal cardinality estimations.

Boncz et al. propose a different approach to reducing the cardinalities in the inner query: They propagate the predicates from the outer query into the subquery[7]. For Q17, this requires adding the filtered part table into the subquery and joining it with lineitem before the aggregate is calculated. Similarly, in Q2, the filters on `p_size` and `p_type` can be propagated into the subquery. For the queries affected by their optimization, Qs 2, 17, and 20, we have manually rewritten the queries to reflect this second approach and confirmed that the performance is indistinguishable from that achieved when using semi join reductions.

We chose semi join reductions over predicate propagations for three reasons: First, semi join reductions only require the addition of a single join on a predicate that has already been identified when it was extracted from the subquery before. As such, we found this approach easier to implement than having to find predicate candidates and identifying joins that need to be duplicated. Second, semi join reductions can be estimated and pushed down similarly to a predicate and do not introduce new join ordering issues. Third, as seen for Q4, they can be applied outside of flattened subqueries, too. However, as seen in the benchmark, identifying cases where this is beneficial is not trivial.

4.9 Subplan Reuse

In four queries (Qs 2, 11, 15, 17, and 21), the same intermediary result is used in the inner and outer query. For example, Q15 uses the revenue view (which contains the revenue per supplier for a given time frame) both to calculate the highest revenue achieved and to filter for the supplier(s)

[7]CP5.2: Moving Predicates into a Subquery

that achieved this revenue. As the view is dropped after each execution, it has to be recalculated independently of how the DBMS implements views. This calculation makes for more than 80% of Q15s execution cost. Q21 has two almost identical subqueries on lineitem, which, when flattened, can share their self join with the outer instance of lineitem. Duplicate subplans also occur outside of an outer/inner join relationship. In Q7, two instances of the nation table are filtered by the same predicates (cf. Section 4.4).

TPC-H makes it relatively easy to identify such reuse opportunities. In the mentioned cases, the subqueries are identical and comparing their LQP subtrees recursively is sufficient. This could already be made more difficult if the syntactical order of columns in predicates was different in the outer and inner queries (e.g., `p_partkey = ps_partkey` vs. `ps_partkey = p_partkey`). Furthermore, if one of the subtrees had an additional predicate, the plans would not be logically equal anymore. In that case, the optimizer would have to decide whether pulling that predicate out of the common subplan (and thereby undoing the predicate push-down) is justified by the opportunity to reuse the subplan. If the additional predicate had caused a different join order, the optimizer would also have had to prove the equivalence of the two join graphs.

The subplan reuse detection in Hyrise does not detect all reuse opportunities. The reason for that is a Hyrise-specific rule, the column pruning rule. It detects columns that are unused in the query plan and removes them early on. For the projection, this means that fewer columns have to be materialized. However, in Q21, one subquery requires `l_receiptdate` and `l_commitdate`, while the other does not. Enabling subplan reuse for these cases is ongoing work.

For this reason, we present two experiments in Figure 13: First, we show the improvement of subplan reuse as it is currently implemented in Hyrise. Second, we disable the column pruning rule, thus enabling more common subplans to be detected. The performance regressions seen here are caused by the unnecessary projections being introduced as the result of the disabled column pruning and are not indicative of the performance of this optimization.

Boncz et al. also report optimization opportunities for Q20[8]. We believe that this is due to the predicates duplicated into the inner join. This does not occur in Hyrise, as it uses a semi join reduction instead (cf. Section 4.8).

For HyPer, it is reported that "decorrelation [cf. Section 4.7], selective join push-down [cf. Section 4.4], and re-use together result in a speedup of a factor 500" [11]. For Hyrise, we measured a comparable factor of 520.

[8]CP5.3: Overlap between Outer- and subquery

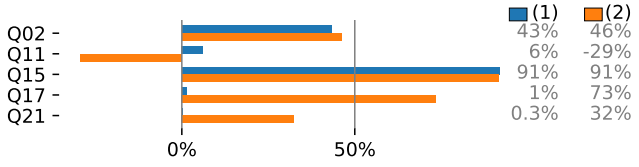


Figure 13: When joins or predicates occur multiple times in the logical query plan, they should only be translated to a single physical operator and should share its results. In Hyrise, the column pruning rule hinders the detection in Qs 17 and 21. We show the performance of subplan reuse with (1) and without (2) enabled column pruning.

4.10 Result Reuse

Instead of reusing the result of subplans within a TPC-H query's execution plan, a DBMS could also cache entire results. Some queries have a very limited parameterization space. Q18, for example, chooses its l_quantity parameter from the integers from the range between 312 and 315. As such, it would be sufficient to run the query four times and store the small results in a 7 KB cache. On the other side of the spectrum, Q16 has 1.6×10^16 variations and a result size of 28 232 B, requiring almost 400 EB for a fully populated cache. In Table 2, we show the required size and the estimated benefit of such a result cache.

While result caching is a technique employed by various DBMSs in production, we have not seen it playing any role in the dissemination of TPC-H results in the research world. As such, we present the theoretical benefits of such a result cache only for the sake of completeness.

Table 2: Analysis of TPC-H queries (SF=1) with regards to their number of parameters, the number of parameter variations, the size of the validation result, the size of a fully filled cache, the execution duration, and the relative benefit (calculated as execution duration in μs divided by the size of a fully filled cache).

Q	Params	Variations	Result	Full Cache	Execution	Benefit
1	1	61	2871 B	171 KB	1,654 s	9,44
2	3	1250	2207 B	2,6 MB	0,113 s	0,04
3	2	155	1836 B	277,9 KB	0,165 s	0,58
4	1	58	1142 B	64,7 KB	0,357 s	5,39
5	2	25	1130 B	27,6 KB	0,234 s	8,28
6	3	80	716 B	55,9 KB	0,010 s	0,17
7	2	600	1693 B	992 KB	0,642 s	0,63
8	2	3750	977 B	3,5 MB	0,193 s	0,05
9	1	92	15622 B	1,4 MB	0,641 s	0,45
10	1	24	7697 B	180,4 KB	0,328 s	1,78
11	1	25	9532 B	232,7 KB	0,040 s	0,17
12	3	210	1185 B	243 KB	0,116 s	0,47
13	1	16	1795 B	28 KB	0,444 s	15,45
14	1	60	721 B	42,2 KB	0,022 s	0,50
15	1	58	1561 B	88,4 KB	0,014 s	0,15
16	1	1.6E16	4.5E20 B	397,6 EB	0,140 s	0,00
17	2	1000	718 B	701,2 KB	0,715 s	1,00
18	1	4	1784 B	7 KB	1,279 s	179,22
19	6	15625000	716 B	10,4 GB	0,117 s	0,00
20	3	11500	972 B	10,7 MB	0,368 s	0,03
21	1	25	921 B	22,5 KB	0,889 s	38,62
22	7	2.4E9	1472 B	3321,3 GB	0,076 s	0,00

5. LOGICAL OPERATOR CHOKE POINTS

The previous optimizations modified the structure of the query plan to be semantically identical but more efficient. By reducing cardinalities early, the amount of work for following operators was reduced. The following choke points differ from the previous group of optimizations in that the structure of the plan, and thus, the input and output cardinalities of all operators remain unchanged. Instead, the logical properties of a single plan node are changed. This means that the potential impact of the related choke point is limited to the cost of those operators. We discuss the removal of functionally dependent group-by columns as well as different execution strategies for `IN` predicates.

5.1 Dependent Group-By Keys

TPC-H uses the SQL-92 standard. It requires output columns in an aggregated query to either be aggregate functions or part of the group-by definition [2, 7.9 (7)]. As such, Q10, which "identifies customers who might be having problems" and returns them by descending revenue, does not only group on the customer key, which would be sufficient to identify a customer. Six additional columns, including the customer's name and address, are also part of the group-by clause. Identifying these functional dependencies based on primary and foreign keys allows reducing the cost of grouping in both hash- and sort-based aggregate operators.

This applies to queries 3, 10, and 18. However, only Q10 shows a significant improvement of 24%. Q3 is dominated by a join between the almost unfiltered lineitem and orders table. In Q18, two aggregates are calculated: First, `SUM (l_quantity)`, grouped by `l_partkey` is calculated on the unfiltered lineitem table. Only later, a second aggregate contains functionally dependent columns from `lineitem` and `customer`. For SF 10, this aggregate operates on less than 1 000 rows, depending on the random parameters.

SQL-99 changed the syntax rules for a query specification to allow not only group-by columns, but also additional columns that are functionally dependent on these [3, 7.11 (13)]. Given that database systems like Oracle 11g do not support this syntax and still require all non-aggregate columns to be included in the group-by list, this pattern is still in use and this optimization continues to be relevant.

This optimization can be extended to include cases where the primary key is not present. Queries 5, 7, 9, and 10 group on `n_name`, Q21 groups on `s_name`. Both columns are non-null and unique[9]. Replacing `GROUP BY s_name` with `s_suppkey` would change the column type from 18 characters to a simple integer. However, as Figure 2 shows, the relative importance of the aggregate is low in these queries. In Q10, the only query where it has a share of 20%, this optimization does not bring a noticeable performance benefit as the nation table holds only 25 records. While we see benefits of this in other workloads, this extension is presented as a negative result in the context of TPC-H.

5.2 Large IN Clauses

TPC-H uses IN clauses with constant elements in Qs 12 (2 elements), 16 (8 elements), 19 (2 and 4 elements), and 22 (7 elements). In a DBMS that does not use just-in-time compilation, there are three ways to evaluate this type of IN expression. The first is to use an interpretative expression evaluator, which handles arbitrarily complex and nested expressions (e.g., `x IN (1, 2.1, 3 + a)`). While this is the most flexible approach, it requires two nested loops, checking for each input value whether it is contained in the list of IN values. It also comes with the interpretative cost incurred by the flexibility of general-purpose expressions (e.g., virtual method calls). The second way is to split IN values into disjunctive predicates, which are evaluated independently. This enables optimizations which can only handle a single predicate, e.g., scans on dictionary-compressed columns. The results of the disjunction are later merged. Finally, in the third approach, a semi join hashes the list of IN values and probes the input values. While this allows for an O(1) probing step, the overhead of the join may outweigh its benefits. Also, in case of the join, all IN expressions have to be of the same type.

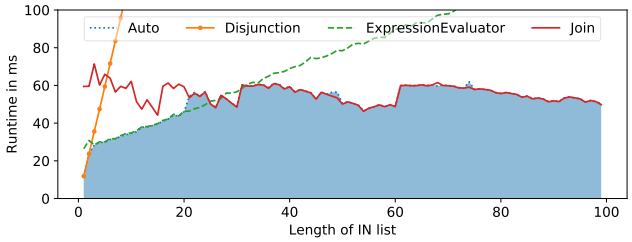


Figure 14: Cost of `SELECT * FROM lineitem WHERE l_suppkey IN (...)` on SF 1 (6M rows in lineitem) with a varying length of the IN list using different evaluation strategies. The shaded "Auto" line denotes the approach chosen by the Hyrise optimizer.

We have conducted an experiment to evaluate the three approaches using a predicate on the `l_suppkey` column with up to 99 values. For up to three values, the disjunctive strategy outperforms the evaluative approach. At about 30 IN values, the join strategy becomes most advantageous as the initial costs have been amortized and the hash lookup becomes faster than the linear search (Figure 14).

Looking at the TPC-H queries, this means that only Qs 12 and 19 benefit from being rewritten into disjunctions. For the latter, the join and aggregate operators outweigh the `p_type` filter and the benefit is only 3.5%. The improvement in Q12 of 60% is due to the replacement of the interpretative expression evaluator with a more efficient scan operator for the scan of the lineitem table.

An interesting finding is that `l_shipmode in ('AIR', 'AIR REG')` in Q19 can be rewritten to `l_shipmode = 'AIR'`. This is because the TPC-H specification defines seven different shipping modes [55, 4.2.2.13]. "AIR REG" is not among them, but "REG AIR" is. An optimizer with accurate statistics containing the values of the table may thus rewrite the predicate to skip "AIR REG". This improves the overall throughput of Q19 by another 12%. We have found no information on whether this optimization is made possible by intention or is an error in the specification.

6. LEARNINGS

This section summarizes the experiments from the previous sections, compares the relevance of the different choke points and provides insights gained during the implementation of the optimizations.

6.1 Relevance of Choke Points

In Table 3, we give an overview of all choke points evaluated in this paper. For each query, the relative change in percent is reported. Also, we provide the geometric mean, which describes the relevance of the choke point for the entire TPC-H benchmark.

Addressing two choke points, namely Predicate Pushdown and Ordering (Section 4.2) and the flattening of subqueries (Section 4.7) improves the TPC-H performance by almost 30×. The combined benefit of the remaining optimizations is less than a factor of 3.

[9]While explicit unique indexes on these columns are prohibited as per paragraph 1.5.6 of the specification, implicit knowledge, for example from the cardinality of the columns' dictionaries, can be used.

Table 3: Influence of choke points in percent. Cells for queries that are not affected by a choke point are left empty, queries where the relative change was less than 10% are marked with * for better readability. The last column references the respective choke point as described by Boncz et al.

Section, Choke Point	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10	Q11	Q12	Q13	Q14	Q15	Q16	Q17	Q18	Q19	Q20	Q21	Q22	Mean	Boncz
4.1: Join Ordering		+76	*	*	+45		*	*	*	*	+52	*	*	*	*	*	*	*	*	*	*	*	+7	2.3
4.2: Pred. Positioning	*	+361	+999	*	+715	+170	+2427	+6542	+1025	+559	+223	+1893	*	+4297	+21	+79	+103	+84	+7368	+576	-13	+25	+396	4.2d
4.3: Between Comp.				*	*					*		*		+13	+66				*				+3	-
4.4: Pred. Duplication							+673												+12				+10	4.2b
4.5: Phys. Locality	*		+52	+13	+15	+335	+32	+13	*	*	*	+26		+90	+124		*	-56	*	+209	-20	*	+21	3.2
4.6: Correl. Columns										+13		+10										*	+1	3.3
4.7: Flattening Subq.		+799134		+446											+93		+29815	+13		+145000	+18591	+27907	+510	5.1
4.8: Semi Join Red.		+1255	*	+288	-24		*	*	*	-14	*	*				*	+1656		-39	+1773	+604		+62	-
4.9: Subplan Reuse		+46									*				+92		+73				+32		+9	5.3
5.1: Dependent group-by			*							+24								*					+1	1.4
5.2: Large IN Clauses												+60				*			*			*	+2	4.2c

For us, the most surprising finding is the low relevance of the join ordering algorithm, which only makes for a noticeable difference in three queries. The other surprising discovery is how uneven the optimization potential is distributed across the TPC-H queries. In six queries (Qs 2, 4, 17, 20, 21, and 22), flattening subqueries brought an improvement of two orders of magnitude or more. This plan transformation can thus be considered to be the single most important optimization for TPC-H.

On the other hand, there are four queries in which even the most significant improvement did not double the performance, namely Qs 1, 13, 16, and 18. For these queries, performance differences between multiple systems are more likely to be caused by their runtime performance.

6.2 Implementation

When developing optimizations for TPC-H, developers have to be aware that SQL features introducing edge cases are largely missing from TPC-H. With the exception of Q13, no NULL values and no outer joins occur in TPC-H. Also, all joins are equi-joins, tempting developers to implement plan transformations that stop being correct when non-inner or non-equi joins are added later on. Projects such as sqllogictest from SQLite[10] and SQLsmith[11] provide comprehensive test suites. The gold standard would be to integrate automatic validation of query plan transformations [13, 14].

Even when optimizations are correct and beneficial for TPC-H, this does not necessarily mean that they are beneficial for other benchmarks or real-world applications. Especially because of the homogeneous data distribution in SQL, developers should include additional benchmarks, such as TPC-DS, JCC-H [8], the Join Order Benchmark [29], or a skewed version of the star schema benchmark [48] and ensure that the optimizer can deal with such skewed data. On the other hand, sophisticated optimizations come with their own fixed costs that have to be amortized. Due to the analytical nature of TPC-H, the internal cost of the optimizer rarely plays a significant role. Developers that optimize not exclusively for analytical workloads should also run transactional workloads (e.g., TPC-C or -E) with short-running queries to ensure that the cost of plan optimization does not dominate that of the execution.

To enable an iterative development process, we found it vital to minimize the time and number of steps needed for a single benchmark iteration, thus increasing the developers' efficiency. This includes reducing the compile time and being able to easily execute TPC-H in different configurations (varying scale factors, number of threads, number of streams, isolated or mixed query execution, ...). Besides our `hyriseBenchmarkTPCH` binary, we found DuckDB's benchmark suite to be a good example of this. Ideally, this would not be the responsibility of the DBMS developers but would be provided by the benchmark itself [34].

Finally, developers should be aware of external influence factors to their benchmarks that may influence their benchmarks. Some factors, such as CPU frequency scaling or background processes on the system, are well known. Often these are alleviated through repeated executions of the same benchmark. However, there are additional influences that are persistent across multiple executions of the same benchmark. A code modification in one place may change the size of the binary, moving parts of a hot loop over page boundaries and thus affecting the performance in seemingly unrelated places. Even worse, iterative rebuilds of the same code may show different performance characteristics than a fresh build. This can be caused by varying linking orders, which again affect the locality and cache-friendliness of hot code [33]. In extreme cases, where the hot loop of our SIMD scan was affected, this reduced the performance by up to 10%. Existing mitigation approaches [12, 16] are outdated or difficult to integrate into a DBMS benchmark setup. We implore compiler researchers to further look into mitigations for these variations. In the absence of this, continuous benchmarking helps in distinguishing outliers from actual performance regressions. Again, DuckDB serves as a good example[12].

7. RELATED WORK

To our knowledge, this is the first paper to systematically compare the impact of the different plan-level choke points. However, individual choke points have been studied before:

Join Ordering (Section 4.1) was discussed by Neumann and Radke [40], who compare 14 different join ordering algorithms for their optimization cost. They found that "TPC-H is no challenge for join ordering algorithms". This confirms our finding that the join ordering choke point is of relatively low significance for TPC-H.

Hellerstein and Stonebraker first identified that in addition to predicate pushdown, the order of the predicates (Section 4.2) is becoming increasingly important [53]. Their ar-

[10] `https://www.sqlite.org/sqllogictest/doc/trunk/about.wiki`

[11] `https://github.com/anse1/sqlsmith`

[12] `https://www.duckdb.org/benchmarks/individual_results/Q02`

gument was that some predicates containing user-defined functions are more expensive to execute than others, for example, because they perform more complicated calculations. In the world of columnar in-memory databases, these differences in the performance of predicate evaluation are also caused by the CPU's caches, the cost of writing positional result lists, and even the efficiency of branch prediction [49].

Physical locality, partition pruning, and column correlations (Section 4.5f) are also discussed by Nica et al. [41]. They used the same clustering for the lineitem and orders tables in order to simulate enterprise databases where data is aged into cold storage. While they find similar pruning ratios for TPC-H data, they found that the latency of accessing the cold storage was the bottleneck in the parallel execution, making it difficult for their implementation to benefit from partition pruning.

On the side of the execution engine, which is not covered in this paper, it is more difficult to comprehensively analyze all choke points and their potential optimizations. This is not only because of the variety of different implementations for a single choke point, but also because they are more tightly coupled. For example, the arithmetic operator performance (CP4.1a [11]) and the interpreter overhead (CP4.1d) interact. The first can be addressed by using vectorized execution, the latter by using JIT. These approaches are hard to unite. Kersten et al. built a test system to allow for an isolated comparison of both [27]. Across five TPC-H queries (Qs 1, 3, 6, 9, and 18), they did not identify one approach as clearly superior.

Looking at the area of benchmarking, database benchmarks nowadays serve two purposes that are hard to reconcile. On the one hand, they are supposed to allow for a comprehensive comparison of database systems from different vendors. For this, they need to cover as many use cases as possible and make it difficult for implementors to implement unfair shortcuts. On the other hand, researchers use database benchmarks to compare new concepts either to their own implementation or to third-party systems. Here, benchmarks should be easy to implement and reproduce.

An example of the first approach is TPC-DS, which was designed to be more complicated than TPC-H in almost all dimensions [35, 44]. An analysis that focuses on the physical system resources stressed by TPC-DS was done by Poess et al. [45]. They compare the utilization of the CPU, the memory, and the network across four unnamed database systems, finding that TPC-DS covers a wide spectrum of resource requirements. While this complexity makes the benchmark more reflective of the variety of queries seen in the real world, it also makes it significantly harder to implement. Only a few publications use the entire TPC-DS benchmark. Others cite either missing support for SQL features or subpar results caused by incomplete optimizations as reasons for lacking queries. Floratou et al. called these selective executions of the benchmark "unscientific comparisons in the name of marketing" [21].

The TPC found that "TPC benchmarks have become extremely complicated to develop and run" [34]. As a reaction, the TPC Express series of benchmarks has been introduced. TPCx-HS [36] is designed to be easier to execute, but, as of now, has not reached the popularity in the research community that TPC-H and TPC-DS have (cf. Figure 1).

The CH-benCHmark [15] combines the transactional and analytical challenges of TPC-C and TPC-H. Others have worked on reducing the complexity of TPC-H. Shao et al. developed DBmbench, a benchmark suite that contains the scaled-down benchmarks μTPC-C and μTPC-H [52]. Besides the complexity of setting up a correct TPC benchmark, they have found the "complex sequences of database operations that may be reordered by the database system depending [sic] the nature of the sequence, the database system configuration and the dataset size" to be an issue that causes substantial variances to the benchmark behavior. Their benchmarks contain only three simplified queries that are claimed to accurately reflect the microarchitectural challenges of TPC-C and TPC-H. This effectively removes most of the plan-based choke points in TPC-H, allowing for a more direct comparison of two execution engines.

A first comparison of the TPC-H choke points to real-world data sets has been done by Vogelsgesang et al. [56]. Some choke points, such as the correlations between columns (cf. Section 4.6) were found to be specific to TPC-H. They highlight the relevance of larger IN clauses (cf. Section 5.2) and two other execution-side choke points.

With this paper, we are the first to isolate multiple logical choke points and quantify their impact within a uniform experimental setup.

8. CONCLUSION

We have evaluated eleven TPC-H choke points using the columnar in-memory DBMS Hyrise. Of these, predicate placement, which, on average, increases the query performance by 4×, and subquery flattening (5.1×) were found to be the most relevant. Not all queries benefit equally from the presented optimizations. Qs 2, 17, and 21 are virtually not executable without optimizations; Qs 1, 13, and 18 hardly benefit from plan-level optimizations at all.

Knowing the choke points' relevance helps in understanding the characteristics of the benchmark and in comparing it to real-world workloads. For new systems without a full-featured optimizer, it supports the prioritization of the development efforts. When comparing database systems with differing levels of optimization, it helps in understanding which queries might be affected by the differences.

The evaluation focused on logical optimizations that transform the query plan to reduce cardinalities early. For these optimizations, we expect other database systems to produce comparable results. For reproducibility, we have provided the source code modifications for the individual benchmarks.

Future work should include an analysis of choke points in variations of TPC-H as well as in more complex benchmarks, such as TPC-DS. Similarly, industry could contribute by providing choke point analyses of their applications. While choke points in the execution engine are not applicable across database systems to the same degree as logical optimizations, future analyses of their impact in different systems, including row- or disk-based systems, and their interplay with different scale factors and multi-threading approaches would give developers a more solid data basis when deciding between different approaches.

Acknowledgements

We thank the TU Munich database group for providing us with early access to Umbra.

References

[1] D. J. Abadi, D. S. Myers, D. J. DeWitt, and S. Madden. Materialization strategies in a column-oriented DBMS. In *Proceedings of the 23rd International Conference on Data Engineering*, ICDE, pages 466–475, 2007.

[2] American National Standards Institute. *American National Standard for Information Systems, Database Language — SQL: ANSI X3.135-1992*. 1992.

[3] American National Standards Institute. *American National Standard for Information Systems, Database Language — SQL: ANSI X3.135-1999*. 1999.

[4] S. Bellamkonda, R. Ahmed, A. Witkowski, A. Amor, M. Zaït, and C. C. Lin. Enhanced subquery optimizations in Oracle. *PVLDB*, 2(2):1366–1377, 2009.

[5] P. A. Bernstein and D. W. Chiu. Using semi-joins to solve relational queries. *Journal of the ACM*, 28(1):25–40, 1981.

[6] M. Boissier and M. Jendruk. Workload-driven and robust selection of compression schemes for column stores. In *Proceedings of the 22nd International Conference on Extending Database Technology*, EDBT, pages 674–677, 2019.

[7] M. Boissier, C. A. Meyer, T. Djürken, J. Lindemann, K. Mao, P. Reinhardt, T. Specht, T. Zimmermann, and M. Uflacker. Analyzing data relevance and access patterns of live production database systems. In *Proceedings of the 25th ACM International Conference on Information and Knowledge Management*, CIKM, pages 2473–2475, 2016.

[8] P. A. Boncz, A. Anatiotis, and S. Kläbe. JCC-H: adding join crossing correlations with skew to TPC-H. In *Performance Evaluation and Benchmarking for the Analytics Era - 9th TPC Technology Conference. Revised Selected Papers*, TPCTC, pages 103–119, 2017.

[9] P. A. Boncz, M. L. Kersten, and S. Manegold. Breaking the memory wall in MonetDB. *Communications of the ACM*, 51(12):77–85, 2008.

[10] P. A. Boncz, S. Manegold, and M. L. Kersten. Database architecture optimized for the new bottleneck: memory access. In *Proceedings of the 25th International Conference on Very Large Data Bases*, VLDB, pages 54–65, 1999.

[11] P. Boncz, T. Neumann, and O. Erling. TPC-H analyzed: hidden messages and lessons learned from an influential benchmark. In *Performance Characterization and Benchmarking - 5th TPC Technology Conference. Revised Selected Papers*, TPCTC, pages 61–76, 2014.

[12] J. Chen and J. Revels. Robust benchmarking in noisy environments. *CoRR*, abs/1608.04295, 2016. arXiv: `1608.04295`.

[13] S. Chu, C. Wang, K. Weitz, and A. Cheung. Cosette: an automated prover for SQL. In *8th Biennial Conference on Innovative Data Systems Research, Online Proceedings*, CIDR, 2017.

[14] S. Chu, K. Weitz, A. Cheung, and D. Suciu. HoTTSQL: Proving Query Rewrites with Univalent SQL Semantics. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI*, pages 510–524, 2017.

[15] R. L. Cole, F. Funke, L. Giakoumakis, W. Guy, A. Kemper, S. Krompass, H. A. Kuno, R. O. Nambiar, T. Neumann, M. Poess, K. Sattler, M. Seibold, E. Simon, and F. Waas. The mixed workload CH-benCHmark. In *Proceedings of the Fourth International Workshop on Testing Database Systems*, DBTest, 2011.

[16] C. Curtsinger and E. D. Berger. STABILIZER: Statistically Sound Performance Evaluation. In *Architectural Support for Programming Languages and Operating Systems*, ASPLOS, pages 219–228, 2013.

[17] M. Dreseler, J. Kossmann, M. Boissier, S. Klauck, M. Uflacker, and H. Plattner. Hyrise re-engineered: an extensible database system for research in relational in-memory data management. In *Proceedings of the 22nd International Conference on Extending Database Technology*, EDBT, pages 313–324, 2019.

[18] M. Dreseler, J. Kossmann, J. Frohnhofen, M. Uflacker, and H. Plattner. Fused table scans: combining AVX-512 and JIT to double the performance of multi-predicate scans. In *34th IEEE International Conference on Data Engineering Workshops*, ICDE Workshops, pages 102–109, 2018.

[19] G. M. Essertel, R. Y. Tahboub, J. M. Decker, K. J. Brown, K. Olukotun, and T. Rompf. Flare: optimizing Apache Spark with native compilation for scale-up architectures and medium-size data. In *13th USENIX Symposium on Operating Systems Design and Implementation*, OSDI, pages 799–815, 2018.

[20] L. Fegaras. A new heuristic for optimizing large queries. In *Proceedings of the 9th International Conference on Database and Expert Systems Applications*, DEXA, pages 726–735, 1998.

[21] A. Floratou, F. Özcan, and B. Schiefer. Benchmarking SQL-on-Hadoop systems: TPC or not TPC? In *Big Data Benchmarking - 5th International Workshop. Revised Selected Papers*, WBDB, pages 63–72, 2014.

[22] S. Halfpap and R. Schlosser. Workload-driven fragment allocation for partially replicated databases using linear programming. In *Proceedings of the 35th International Conference on Data Engineering*, ICDE, pages 1746–1749, 2019.

[23] D. Inkster, M. Zukowski, and P. A. Boncz. Integration of VectorWise with Ingres. *SIGMOD Record*, 40(3):45–53, 2011.

[24] R. Johnson, N. Hardavellas, I. Pandis, N. Mancheril, S. Harizopoulos, K. Sabirli, A. Ailamaki, and B. Falsafi. To share or not to share? In *Proceedings of the 33rd International Conference on Very Large Data Bases*, VLDB, pages 351–362, 2007.

[25] A. Kemper and T. Neumann. HyPer: A hybrid OLTP & OLAP main memory database system based on virtual memory snapshots. In *Proceedings of the 27th International Conference on Data Engineering*, ICDE, pages 195–206, 2011.

[26] T. R. Kepe, E. C. de Almeida, and M. A. Z. Alves. Database processing-in-memory: an experimental study. *PVLDB*, 13(3):334–347, 2019.

[27] T. Kersten, V. Leis, A. Kemper, T. Neumann, A. Pavlo, and P. A. Boncz. Everything you always wanted to know about compiled and vectorized queries but were afraid to ask. *PVLDB*, 11(13):2209–2222, 2018.

[28] V. Leis, P. A. Boncz, A. Kemper, and T. Neumann. Morsel-Driven Parallelism: A NUMA-Aware Query Evaluation Framework for the Many-Core Age. In *International Conference on Management of Data*, SIGMOD, pages 743–754, 2014.

[29] V. Leis, A. Gubichev, A. Mirchev, P. A. Boncz, A. Kemper, and T. Neumann. How good are query optimizers, really? *PVLDB*, 9(3):204–215, 2015.

[30] Y. Li and J. M. Patel. WideTable: an accelerator for analytical data processing. *PVLDB*, 7(10):907–918, 2014.

[31] G. Moerkotte. Small materialized aggregates: A light weight index structure for data warehousing. In *Proceedings of the 24rd International Conference on Very Large Data Bases*, VLDB, pages 476–487, 1998.

[32] G. Moerkotte and T. Neumann. 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*, VLDB, pages 930–941, 2006.

[33] T. Mytkowicz, A. Diwan, M. Hauswirth, and P. F. Sweeney. Producing Wrong Data Without Doing Anything Obviously Wrong! In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS, pages 265–276, 2009.

[34] R. O. Nambiar and M. Poess. Keeping the TPC relevant! *PVLDB*, 6(11):1186–1187, 2013.

[35] R. O. Nambiar and M. Poess. The making of TPC-DS. In *Proceedings of the 32nd International Conference on Very Large Data Bases*, VLDB, pages 1049–1058, 2006.

[36] R. O. Nambiar, M. Poess, A. Dey, P. Cao, T. Magdon-Ismail, D. Q. Ren, and A. Bond. Introducing TPCx-HS: the first industry standard for benchmarking big data systems. In *Performance Characterization and Benchmarking. Traditional to Big Data - 6th TPC Technology Conference. Revised Selected Papers*, TPCTC, pages 1–12, 2014.

[37] T. Neumann. Engineering high-performance database engines. *PVLDB*, 7(13):1734–1741, 2014.

[38] T. Neumann and M. J. Freitag. Umbra: A disk-based system with in-memory performance. In *10th Conference on Innovative Data Systems Research, Online Proceedings*, CIDR, 2020.

[39] T. Neumann and A. Kemper. Unnesting arbitrary queries. In *Datenbanksysteme für Business, Technologie und Web"*, BTW, pages 383–402, 2015.

[40] T. Neumann and B. Radke. Adaptive optimization of very large join queries. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD, pages 677–692, 2018.

[41] A. Nica, R. Sherkat, M. Andrei, X. Chen, M. Heidel, C. Bensberg, and H. Gerwens. Statisticum: data statistics management in SAP HANA. *PVLDB*, 10(12):1658–1669, 2017.

[42] K. Ono and G. M. Lohman. Measuring the complexity of join enumeration in query optimization. In *Proceedings of the 16th International Conference on Very Large Data Bases*, VLDB, pages 314–325, 1990.

[43] L. Orr, S. Kandula, and S. Chaudhuri. Pushing data-induced predicates through joins in big-data clusters. *PVLDB*, 13(3):252–265, 2019.

[44] M. Poess, R. O. Nambiar, and D. Walrath. Why you should run TPC-DS: A workload analysis. In *Proceedings of the 33rd International Conference on Very Large Data Bases*, VLDB, pages 1138–1149, 2007.

[45] M. Poess, T. Rabl, and H. Jacobsen. Analysis of TPC-DS: the first standard benchmark for SQL-based big data systems. In *Proceedings of the 2017 Symposium on Cloud Computing*, SOCC, pages 573–585, 2017.

[46] M. Raasveldt, P. Holanda, T. Gubner, and H. Mühleisen. Fair Benchmarking Considered Difficult: Common Pitfalls In Database Performance Testing. In *7th International Workshop on Testing Database Systems*, DBTest, 2:1–2:6, 2018.

[47] M. Raasveldt and H. Mühleisen. DuckDB: an embeddable analytical database. In *Proceedings of the 2019 International Conference on Management of Data, SIGMOD*, pages 1981–1984, 2019.

[48] T. Rabl, M. Poess, H. Jacobsen, P. E. O'Neil, and E. J. O'Neil. Variations of the Star Schema Benchmark to Test the Effects of Data Skew on Query Performance. In *ACM/SPEC International Conference on Performance Engineering*, ACPE, pages 361–372, 2013.

[49] K. A. Ross. Conjunctive selection conditions in main memory. In *Proceedings of the 21st ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 109–120, 2002.

[50] R. Schlosser, J. Kossmann, and M. Boissier. Efficient scalable multi-attribute index selection using recursive strategies. In *Proceedings of the 35th International Conference on Data Engineering*, ICDE, pages 1238–1249, 2019.

[51] D. Schwalb, M. Faust, J. Wust, M. Grund, and H. Plattner. Efficient transaction processing for Hyrise in mixed workload environments. In *Proceedings of the 2nd International Workshop on In Memory Data Management and Analytics*, IMDM, pages 16–29, 2014.

[52] M. Shao, A. Ailamaki, and B. Falsafi. DBmbench: fast and accurate database workload representation on modern microarchitecture. In *Proceedings of the 2005 Conference of the Centre for Advanced Studies on Collaborative Research*, pages 254–267, 2005.

[53] E. Simon. Predicate migration: optimizing queries with expensive predicates. *ACM SIGMOD Digital Review*, 2, 2000.

[54] K. Stocker, D. Kossmann, R. Braumandi, and A. Kemper. Integrating Semi-Join-Reducers into State-of-the-Art Query Processors. In *Proceedings of the 17th International Conference on Data Engineering*, ICDE, pages 575–584, 2001.

[55] Transaction Processing Performance Council. *TPC Benchmark H (Decision Support) - Standard Specification*. 1993.

[56] A. Vogelsgesang, M. Haubenschild, J. Finis, A. Kemper, V. Leis, T. Muehlbauer, T. Neumann, and M. Then. Get real: how benchmarks fail to represent the real world. In *Proceedings of the Workshop on Testing Database Systems*, DBTest'18, 1:1–1:6, 2018.

[57] Y. Wu, J. Arulraj, J. Lin, R. Xian, and A. Pavlo. An empirical evaluation of in-memory multi-version concurrency control. *PVLDB*, 10(7):781–792, 2017.

[58] M. Ziauddin, A. Witkowski, Y. J. Kim, J. Lahorani, D. Potapov, and M. Krishna. Dimensions based data clustering and zone maps. *PVLDB*, 10(12):1622–1633, 2017.