Vrije Universiteit Amsterdam

Master Thesis

Join Order Optimization with (Almost) No Statistics

Author: Tom Ebergen (2701116)

daily supervisor: Laurens Kuiper (DuckDB Labs)
1st supervisor: Hannes Mühleisen (DuckDB Labs)
2nd reader: Peter Boncz (CWI)

A thesis submitted in fulfillment of the requirements for the degree of 'Parallel and Distributed Computing Systems' Master of Science in Computer Science

September 16, 2022

Abstract

Selecting a good join order for a query can have a significant impact on query execution performance. A good cardinality estimator is a necessary component of the join order optimization process. Cardinality estimators frequently rely on base table statistics to provide estimates. DuckDB is an in-process database system that can run directly on parquet and CSV files. When querying on these files, almost no statistics are provided, making accurate cardinality estimation and therefore join ordering, difficult. In this work we develop a cardinality estimator that requires little to no statistics, implement it in the DuckDB join order optimizer, and evaluate its performance.

Foreign-key Primary-key joins are very popular in practice, and exhibit properties that are used by our new cardinality estimator, even in the absence of base table statistics. When tested on the Join Order Benchmark using parquet files, the new join order optimizer selects plans that produce 66% less intermediate tuples on average. When run on base tables, 95% less intermediate tuples were produced, and results were comparable to state-of-the-art systems. The new join order optimizer improved end-to-end execution times by 18% on average when run on parquet files, and 25% on average when run on base tables.

By assuming all joins are foreign-key primary-key joins, and by leveraging their inherent properties, our simple cardinality estimator significantly improves the quality of join orders selected by the join order optimizer. The selected join orders improved by an order of magnitude and are comparable to state-of-the-art systems.

Contents

List of Figures

List of Tables

LIST OF TABLES

1

Introduction

Before a database system executes a query, many checks, transformations, and optimizations are performed. Query checks involve scanning, parsing, and semantically validating the query text to ensure correctness. Then, an internal representation of the query is created, usually as a tree-like data structure called a logical plan. The logical query plan is then optimized and passed to the execution engine which produces a query result.

In database research, the process of finding an optimal plan is called query optimization. There are many methods to optimize a query plan, including join order optimization, which determines the order in which the requested relations are joined. Join order optimization requires a cardinality estimator, a cost model, and a join enumeration algorithm (1, 2, 3). The performance difference between a query plan with a good join order and a bad join order can be extremely large, therefore, join order optimization has become a well-studied research topic within database systems.

Many query optimizers begin the optimization process by applying a set of rules that preserve the semantic meaning of a query while improving the execution strategy. One example of a rule-based optimization is `FILTER` push down, or removing cartesian joins and replacing them with `INNER` joins when a join condition exists. Both of these rules have the goal of minimizing the number of intermediate tuples generated, which results in faster execution times (4).

When rule-based optimization is finished, a query optimizer will attempt to optimize the join order. The join order of a query plan describes the order in which base tables are joined. Join order optimization is considered a cost-based optimization as there are many join orders to choose from, and a predicted cost for a join order is used to choose the optimal order. Figure 1.1 shows an example of two different join plans given three relations. A query that joins n relations will have $n-1$ joins, and $n!$ different join orders.

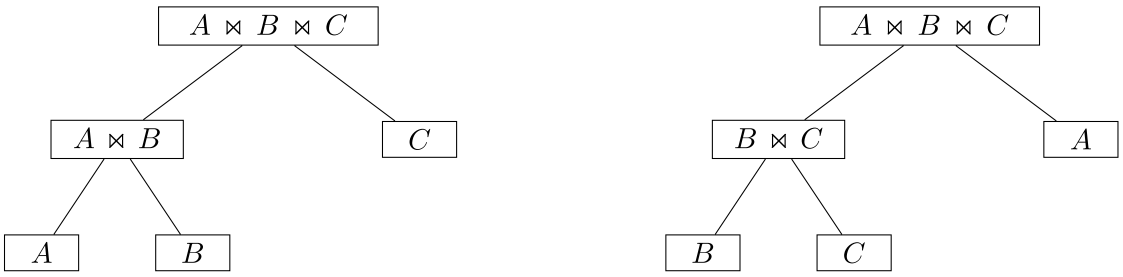


Figure 1.1: Two different join orders for three relations. The left plan joins relations A and B first, while the right plan joins relations B and C first. The order of the joins can affect the cost estimates produced by a cost model.

Enumerating and estimating the cost of $n!$ join plans is prohibitively expensive given time constraints and a potentially high number of relations. Therefore, the set of join orders considered, also known as the search space, must be pruned. Some join order optimizers do this by only considering left-deep trees (5, 6). The process of producing candidate join orders is done by a plan space enumeration algorithm, as illustrated in Figure 1.2.

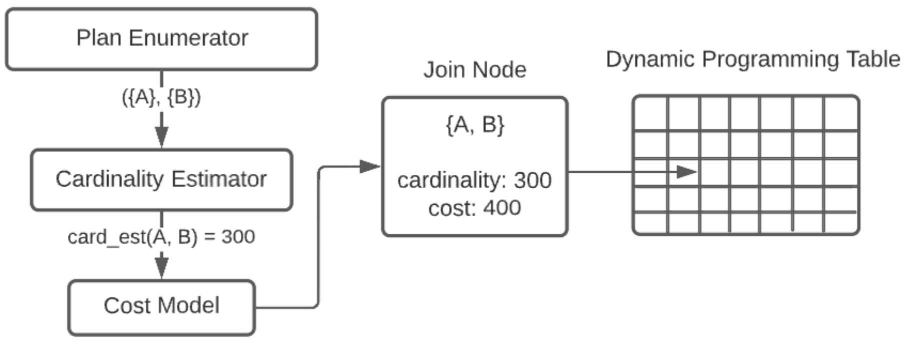


Figure 1.2: The join order optimization process. The plan enumerator emits pairs of sets of relations. The cardinality of joining the pair is calculated by the cardinality estimator which may use statistics like histograms and samples. The cardinality estimate is passed to the cost model, which assigns a cost to the join. A dynamic programming table stores the cost of each join and references to the child intermediate joins. The optimal join order is then found by examining the entry in the DP table with all relations and following the references to the child nodes.

A cost model is responsible for estimating the cost of a query plan (4). Many factors can affect the estimated cost of a query plan such as join order, estimated cardinality, and I/O time. A cost model normally has two categories of input. One describes the estimated amount of data that will be processed while executing the query plan, and the other is an estimate of the physical costs incurred when executing the query plan (i.e memory/network bandwidth). The first category is usually labeled as cardinality estimation. Many state-of-the-art systems (PostgreSQL, DB2, SQLServer) maintain table statistics to help with estimating the costs of both inputs (4, 7, 8, 9). With perfect cardinality estimates, the join order optimizer can select near-optimal plans. However, if these inputs are off, especially cardinality estimation, it can lead to sub-optimal and potentially disastrous plans. One reason why cardinality estimates are consistently inaccurate is because they are built on assumptions of independence and uniformity. These assumptions make it hard for state-of-the-art cardinality estimators to detect join-crossing correlations and skew, which are present in real-world data sets (1, 10).

DuckDB is a fast in-process database system developed by Mark Raasveldt and Hannes Mühleisen (11). It was designed to be an in-process analytical DBMS, or "The SQLite of analytics". It can run queries directly on CSV and parquet files. DuckDB has rule-based optimization, but the cardinality estimator used by the join order optimizer is naive. Every join is assumed to be a foreign-key primary-key join and the cardinality of a join is estimated to be the cardinality of the larger table. In addition, the effects of table filters are not accounted for when initializing the base table. The cost of a query plan then becomes the sum of all estimated cardinalities of the intermediate joins. While the assumption that every join is a foreign-key primary-key join is not necessarily a weakness, the formula used to calculate the cardinality of the join fails to consider the effects of independent filters, skew, correlation, or how column values can multiply or be filtered out throughout many joins.

The cardinality estimator in the DuckDB join order optimizer can be upgraded. Many state-of-the-art systems use statistics to aid with cardinality estimation and still select disastrous join orders. The intended workflow for DuckDB does not allow it to collect similar statistics as many sessions are short-lived, or data is queried directly from parquet or CSV files, which do not provide statistics. With these constraints, the goals for this thesis can be defined as such,

G1 Develop an accurate cardinality estimator using minimal to no base table statistics

G2 Enable the DuckDB join order optimizer to select query plans with faster end-to-end execution time

G3 Implement and evaluate these changes

This thesis presents and describes a minimal cardinality estimator and cost model for the DuckDB join order optimizer designed to achieve the mentioned goals. The new join order optimizer will not explicitly require table statistics, but will improve the quality of selected query plans when provided with an estimated HyperLogLog count for each column. Comparing query profile results from the *Join Order Benchmark* (JOB) and the TPC-H benchmark, the new DuckDB join order optimizer selects query plans that process 65% fewer tuples during query execution compared to plans selected by the old join order optimizer. When statistics are provided DuckDB's new join order optimizer selects query plans that process 95% fewer tuples during query execution compared to plans selected by the old join order optimizer.

2

Background

In 1979 *Selinger et al.* introduced the System R query optimizer. The System R optimizer had three distinct components; a query plan enumerator, a cardinality estimator, and a cost model to estimate how computationally expensive each join order would be. The System R plan enumerator restricted the search space to just left deep trees. For each subset of relations, the cheapest option was stored in a dynamic programming table, and the enumerator would add on new relation to each subset until all relations were joined (5).

In 2006, *Moerkotte et al.* introduced the query plan enumeration algorithm `DPccp`. `DPccp` improved on the SystemR algorithm by also considering bushy trees. A weakness of `DPccp`, however, was that it would only work on simple binary join predicates and inner joins (12). Two years later, *Moerkotte et al.* presented `DPhyp` as an improvement to `DPccp`. `DPhyp` models a given query as a hypergraph and does not suffer from the same limitations (13). `DPhyp` is the plan enumeration algorithm used in DuckDB today. `DPhyp` is built on dynamic programming principles, so a cost model must exhibit **optimal substructure** in order to properly find the optimal query plan. Optimal substructure for join ordering means that an optimal solution for joining n relations incorporates optimal solutions for joining $n-i$ and i relations (where $1 \leq i < n$). The optimal solutions for the sub-problems can be determined independently (14).

The plans produced by a join order enumerator are then evaluated by a cost model. In System R, the cost of a join order is the sum of the costs of each intermediate join. System R calculates the cost of a single join by estimating the two popular inputs described in the introduction; physical properties of the data and machine, and the estimated cardinality of the join (5). While the use of both inputs seems practical, *Leis et al.* analyzed the accuracy of state-of-the-art systems with similar cost model inputs. To determine the importance of inputs like I/O and CPU utilization (physical costs), the authors developed a simple cost model that only considered the sum of the estimated cardinalities of all intermediate join nodes. The state-of-the-art results were then compared to the results of the simple cost model. The analysis revealed that inaccurate cardinality estimates are the main reason why a query optimizer selects sub-optimal or "disastrous" query plans. Therefore, in the presence of inaccurate cardinality estimates, costs incurred by the physical properties of a machine can be considered negligible. Since cardinality estimation errors increase exponentially with the number of joins (15), the paper concludes that simply the sum of the estimated cardinalities of intermediate nodes has comparable cost accuracy to complex cost models that consider details regarding a systems hardware (1).

In order to accurately estimate the cardinality of a join, many of the same traditional database systems mentioned in (1) will collect column statistics like maximum frequency values, histograms, distinct value count, or column correlation (16, 17, 18). As mentioned in the introduction, DuckDB provides an estimate of the distinct count of columns ingested into DuckDB. Knowing the distinct count of two columns that are joined by an equality condition can help determine the distinct count of the join and the cardinality of the join. HyperLogLog statistics are not always available on parquet or CSV files.

When estimating the cardinality of join, it is also helpful to know what types of joins there are, and which ones are popular in practice. A very popular join type is the foreign-key primary-Key (FK-PK) join. An FK-PK join is popular because all values in a foreign key column are derived from a primary key column located in another table. When PK-FK joins occur between unfiltered tables, every value in the foreign key column must match a value in the primary key column. A foreign-key primary-key join satisfies the *Containment of Value Sets* assumption.

- *Containment of Value Sets.* If Y is an attribute appearing in several relations, then each relation chooses its values from the front of a fixed list of values $y_1, y_2, y_3, \ldots$, and has all the values in that prefix. As a consequence, if R and S are two relations with an attribute Y, and $V(R, Y) < V(S, Y)$, then every Y-value of R will be a Y-value of S (19).

While this thesis will focus on reducing the number of intermediate tuples processed during the execution of a query plan, understanding the low-level details of how a join gets executed is also important. Suppose two tables L and M are joining on $L.m = R.s$, and $|L| > |R|$. During a hash join, the relation with the least number of records is called the build side, so relation R in the current example. All records of $R.s$ are then hashed and placed into buckets, called the partitioning phase. In the ideal case, all of these records can fit into main memory, or even better, CPU cache. Once the partitioning phase is over, the probing phase begins. During the probing phase, each record of $L.m$ is hashed with the same hash function used for $R.s$, and the hash buckets of $R.s$ are probed for a match. All matches produce a record in the resulting join (4). With this knowledge, it is evident that smaller build sides for joins are preferred. If the build side of a join is large, the probing phase can take a long time, since every partitioned block needs to be loaded from main memory, or worse, storage.

Plan enumeration methods, cardinality estimation methods, base table statistics, and popular join types are all important concepts to understand when building a join order optimizer. A dynamic programming table that stores optimal join plans forces the cost model to exhibit optimal substructure. A cost model with optimal substructure requires a cardinality estimator that is symmetric, producing the same cardinality estimate for a set of relations regardless of join order. An accurate cardinality estimator must leverage properties of popular join types, and use table statistics whenever possible. With a basic understanding of these concepts, the rest of this thesis will describe how a join order optimizer provided with little to no statistics can produce query plans comparable to state-of-the-art systems.

3 Design

3.1 Cardinality Estimator

Foreign-Key Primary-Key (FK-PK) joins are common joins in query benchmarks, since they are common in real-life use cases. FK-PK joins satisfy the *Containment of Value Sets* property mentioned in Chapter 2. When this property is satisfied, knowing the number of distinct values in a column will help predict the cardinality of a join. Given the widespread use of FK-PK joins, the design of our cardinality estimator will assume that the *Containment of Value Sets* property is consistently guaranteed.

A table resulting from a FK-PK join can duplicate column values, or selectively filter them out given some join condition. The design of the new DuckDB cardinality estimator is influenced by these properties. To represent these properties when estimating cardinalities, we will introduce two new variables for each column, *multiplicity* and *selectivity.* We can define these variables more formally as

- *Multiplicity* - The average frequency of each distinct value in a column.
- *Selectivity* - The probability a column domain value is found in a column.

Selectivity is defined by the probability a column domain value is found in a column. This will require a variable to represent the current number of distinct values in a column, as well as a variable to represent the number of distinct values in its domain. Two variable names we can use to represent these numbers are *current domain* and *total domain.* The *current domain* will represent the number of distinct values in a column, while the *total domain* will represent the number of distinct values in the column's total domain. Following from the *containment of value sets* property, all distinct values of a column's *current domain* are selected from the column's *total domain.* A column's *current domain* can never be larger than its *total domain*, so the selectivity of a column should never be greater than one. *Total domain* and *current domain* will be abbreviated as *tdom* and *cdom* respectively for the rest of the paper. The selectivity for a column x in table A can be determined by the equation

$$\begin{aligned} cdom(A.x) &= \text{the number of distinct values in the current domain of } A.x \\ tdom(A.x) &= \text{the number of distinct values in the total domain of } A.x \\ sel(A.x) &= \frac{cdom(A.x)}{tdom(A.x)} \end{aligned} \tag{3.1}$$

Equation 3.1 shows that the selectivity of an unfiltered table can be < 1. It is important to note that every column's current domain may not be equal to its total domain in the absence of filtering. Certain data sets have "dead data" when joining two tables, meaning some primary keys never have a match in a foreign key table. For example, in the TPC-H data sets, only two-thirds of the customers have orders (20). By assuming a column's total domain could be more than the current domain, the new cardinality estimator can better handle cases of "dead data".

A column's *multiplicity* is defined as the average frequency of each distinct value, and cannot be defined with the same two variables. To define multiplicity, the cardinality of a table must also be known. Equation 3.2 is the formula to define the *multiplicity* of column x in table A

$$card(A) = \text{the cardinality of table } A$$

$$mul(A.x) = \frac{card(A)}{cdom(A.x)} \tag{3.2}$$

A column's multiplicity is defined as the table cardinality divided by the current number of distinct values in the table. This definition assumes uniformity in the column, making our multiplicity equation susceptible to skew and correlation. In primary key columns, every value is guaranteed to be distinct, so the *current domain* of the column is equal to the *total domain* of the column and the cardinality of the column. This makes the *multiplicity* and *selectivity* equal to 1, meaning there are no duplicated values, or missing values. A foreign key column, however, is allowed to have duplicated column values, and the average frequency by which those values are duplicated is determined by the cardinality of the table and the *current domain* of the column.

The next step is combining the *sel* and *mul* equations into a cardinality estimation equation. Suppose there are two relations, A and B joining on $A.w = B.x$. A and B can be base tables or intermediate tables. Knowing $card(A)$ and $card(B)$, the goal is to determine $card(A \bowtie B)$. If we select a random tuple z from table A, we can predict the number of resulting tuples z will produce in $(A \bowtie B)$, with the following equation

$$P(z \in (A \bowtie B).x) = P(z \in B.x) \cdot mul(B.x) \tag{3.3}$$

Our probability equation can be reformulated using our selectivity equation since selectivity naturally represents the probability that a value is in the current domain of a table. So we can write the following

$$sel(B.x) = P(z \in B.x) \tag{3.4}$$

To get an estimated count of tuples in $(A \bowtie B)$ resulting from A, we can modify equation 3.3 and multiply it by *card(A)*. We can write this as $(A \bowtie B)_A$,

$$\begin{align} (A \bowtie B)_A &= card(A) \cdot P(z \in B.x) \cdot mul(B.x) \tag{3.5} \\ &= card(A) \cdot sel(B.x) \cdot mul(B.x) \tag{3.6} \end{align}$$

We can compute how many tuples end up in $(A \bowtie B)$ from table B using a similar equation

$$\begin{align} (A \bowtie B)_B &= card(B) \cdot P(z \in A.w) \cdot mul(A.w) \tag{3.7} \\ &= card(B) \cdot sel(A.w) \cdot mul(A.w) \tag{3.8} \end{align}$$

Equations 3.6, and 3.8 can be expanded by substituting the equations for *mul* (3.2) and *sel* (3.1),

$$
\begin{aligned}
card(A \bowtie B)_A &= card(A) \cdot sel(B.x) \cdot mul(B.x) && (3.9)\\
&= card(A) \cdot \frac{cdom(B.x)}{tdom(B.x)} \cdot \frac{card(B)}{cdom(B.x)} && (3.10)\\
&= \frac{card(A) \cdot card(B)}{tdom(B.x)} && (3.11)
\end{aligned}
$$

$$
\begin{aligned}
card(A \bowtie B)_B &= card(B) \cdot sel(A.w) \cdot mul(A.w) && (3.12)\\
&= card(B) \cdot \frac{cdom(A.w)}{tdom(A.w)} \cdot \frac{card(A)}{cdom(A.w)} && (3.13)\\
&= \frac{card(B) \cdot card(A)}{tdom(A.w)} && (3.14)
\end{aligned}
$$

Equations 3.11 and 3.14 are nearly identical, except for the *tdom* term in the denominator. The join condition is $A.w = B.x$, which are the *tdom* terms present in 3.11 and 3.14. Since every join is assumed to be a FK-PK join, the *Containment of Value Sets* property is satisfied, and the two columns have the same total domain and $tdom(A.w) = tdom(B.x)$. Consequently, equations 3.11 and 3.14 are equivalent.

Equation 3.11 enables us to represent the selectivity of a filter in the following way,

$$
jsel(A.*,\ B.*) = \frac{1}{tdom(A.*)} \qquad (3.15)
$$

Where a pair $jsel(A.*, B.*)$ represents the *join selectivity* of a join condition between relations A and B. The *s in the equation represent the columns used in the join. Our cardinality estimation equation then becomes,

$$
\begin{aligned}
card(A \bowtie B) &= \frac{card(B) \cdot card(A)}{tdom(A.w)}\\
&= card(A) \cdot card(B) \cdot jsel(A.*,\ B.*) && (3.16)
\end{aligned}
$$

This result is a simple, yet effective cardinality estimation equation. A benefit of this cardinality estimation equation is that it does not need to keep track of a *cdom*, only table cardinalities and *tdoms*. Barely any statistics are required. This cardinality estimation equation also assumes independence and uniformity within the data set, so inaccurate estimates are likely in the presence of skew and correlation.

While this equation was developed independently, it is very similar to the formula described in (2, 4, 5), which is

$$card(A \bowtie B) = \frac{card(A) \cdot card(B)}{\text{Max}(V(A.w), V(B.x))} \tag{3.17}$$

For a join condition $A.w = B.x$. $V(A.*)$ represents the number of distinct values in column $A.*$. While the two equations are extremely similar, the equation 3.11 was developed using multiplicity and selectivity ideas. Equation 3.17 was developed using the probability that a tuple from the cartesian product would be in the resulting equality join.

Equation 3.16 assumes that the two joining relations have only one join condition between them. It is possible, however, to join n tables with an arbitrary number of conditions. When this happens, our cardinality estimator must determine which join conditions to apply. As an example, suppose three tables A, B and C are joined with three join conditions, $A.* = B.*$, $B.* = C.*$, and $C.* = A.*$.

Calculating the cardinalities of $(A \bowtie B), (B \bowtie C)$, and $(C \bowtie A)$, we get

$$\begin{aligned} card(A \bowtie B) &= card(A) \cdot card(B) \cdot \mathit{jsel(A.*,\ B.*)} \qquad (3.18) \\ card(B \bowtie C) &= card(B) \cdot card(C) \cdot \mathit{jsel(B.*,\ C.*)} \\ card(C \bowtie A) &= card(C) \cdot card(A) \cdot \mathit{jsel(C.*,\ A.*)} \end{aligned}$$

If we join each of these tables with the third table, two more filters can be applied. Equation 3.18 will be used as an example.

$$\begin{aligned} card(C \bowtie (A \bowtie B)) = card(A) \cdot card(B) \cdot card(C) \cdot \\ \mathit{jsel(C.*,\ B.*)} \cdot \mathit{jsel(C.*,\ A.*)} \cdot \mathit{jsel(A.*,\ B.*)} \end{aligned} \tag{3.19}$$

While in practice it seems like all filters should be applied, studies on cardinality estimators have shown that they notoriously underestimate cardinalities and that these errors are propagated as more joins are considered (1, 15, 21). In order to avoid underestimating cardinalities while still applying at least one join condition to each table, we need to re-characterize this problem as a graph theory problem.

Given

- a set of relations $\{R_1, R_2, ..., R_{n+1}\} = \mathfrak{R}$
- a set of join selectivities, $\mathfrak{J}$, where $\forall\ jsel(A.*, B.*) \in \mathfrak{J},\ A \in \mathfrak{R} \wedge B \in \mathfrak{R}$

We can create a graph G using relations in R as vertices and join conditions as edges. Each edge then has a weight equivalent to the join selectivity of the join condition. Once this graph is constructed, a minimum spanning tree of the graph will find a set of join selectivities such that each relation will have at most one join selectivity applied, and the product of the join selectivities will be the most selective. Our cardinality equation then looks like

$$card(\mathcal{R}) = \prod_{R \in \mathcal{R}} card(R) \cdot \Gamma(\mathcal{R}, \mathcal{J}) \tag{3.20}$$

Where $\Gamma(J)$ is the product of all join selectivities in the minimum spanning tree of the graph created using $(\mathcal{R}, \mathcal{J})$. This cardinality estimation equation above means that we can estimate the cardinality of **any** join as a calculation of just base tables and filters. In addition, it requires only the knowledge of the number of distinct values in a column. The equation fits very nicely into the dynamic programming approach for join order optimization as well. Every entry in the dynamic programming table is represented as a set of relations, and now the cardinality is also represented as a set of relations.

3.2 Cardinality Estimation with No Statistics (Equivalence Sets)

Currently, Equation 3.20 only works under the assumption that DuckDB has access to table statistics, specifically the number of distinct values in a column (or at least an estimate). These statistics are available on base tables and are estimated as a function of the HyperLogLog count on 10% of values in a column, and the total cardinality of the table.

In the absence of statistics and distinct column counts, Equation 3.20 breaks down. Parquet and CSV files seldomly provide statistics on distinct column counts, only min, max, and table cardinality are provided. To make sure Equation 3.20 can still work in the absence of a statistics-provided distinct count, we introduce the idea of **equivalence sets**. An equivalence set is a set of columns that are transitively joined. By assuming all joins between columns in an equivalence set are FK-PK joins, all columns have the same *tdom*.

```
SELECT count(*) from A, B, C
WHERE A.x = B.y
AND B.y = C.z;
```

In the query above, if every join is assumed to be a FK-PK join, then {`A.x, B.y, C.z`} is an equivalence set. This equivalence set has one *tdom* value, which can be determined via two strategies depending on if statistics are available or not. If the number of distinct values is known for a subset of columns in the equivalence set, the *tdom* for the equivalence set will be the highest reported distinct value count among the columns in the equivalence set. If none of the columns in the equivalence set have a reported distinct value count, then the column with the lowest cardinality in the equivalence set is assumed to be a primary key and will initialize the *tdom* for the equivalence set. The lowest cardinality column in an equivalence set is used as the primary key because primary key columns typically have lower cardinalities than their foreign key column counterparts. Given the *tdom* of a column, we can calculate the join selectivity (or *jsel*) of any join condition that uses that column.

3.3 Commutativity & Associativity

As mentioned in Chapter 2, the current DuckDB join order optimizer uses the dynamic programming algorithm `Dhyp` developed by *Moerkotte et al.* To ensure an **optimal substructure** when calculating the optimal join order, every optimal join must be the result of two other optimal joins. Since the cost of a join is just the sum of intermediate cardinalities, we can guarantee the optimal substructure. However, to ensure that cardinality estimates do not change, Equation 3.20 must produce the same result regardless of join order. Therefore Equation 3.20 must be associative and commutative.

To prove commutativity and associativity we need to prove the following

$$card(A \bowtie B) = card(B \bowtie A) \tag{3.21}$$

$$card(A \bowtie (B \bowtie C)) = card((A \bowtie B) \bowtie C) \tag{3.22}$$

The proofs for these are trivial given how we have defined the equation. The cardinality equation takes as input a set of base tables and a set of filters. The result is then a product of the cardinalities of the relations and $\Gamma(\mathfrak{J})$. Multiplication is by definition commutative and associative. Therefore we can conclude that our equation is associative and commutative.

3.4 Selectivity Estimation

When Equation 3.20 is applied on base tables, it uses the estimated cardinality of a table after a filter has been applied. With no statistics on a table, it is hard to know the selectivity of a filter. In these cases, the new join order optimizer will naively estimate a selectivity of 20%. In the presence of HyperLogLog statistics, there are a few cases where selectivity estimates can be more accurate. For instance, if there is an equality filter on a column $A.f$, like $(A.f = 100)$, the resulting cardinality of the base table can be estimated to be $\frac{card(A)}{tdom(A.f)}$. This idea can be trivially extended to equality filters connected with an "OR". If a column is filtered in the following way

```
SELECT *
FROM A
WHERE A.f = 100 or A.f = 200 or A.f = 300;
```

we can estimate the cardinality to be $\frac{o \cdot card(A)}{tdom(A.f)}$, where o represents the number of equality filters joined with an `or`. This idea was already present in the SystemR paper (5).

3.5 Cost Model

Inspired by the conclusions made by *Leis et al.* in (1), the cost of a join plan will be the sum of all estimated cardinalities of all intermediate joins in our cost model. In this section, we will analyze how our cost function is affected by our cardinality estimator. We will start by analyzing the cost of a join between three tables A, B, and C with no table filters. The join conditions are $A.* = B.*$ and $B.* = C.*$,

$$card(A \bowtie B) = card(A) \cdot card(B) \cdot jsel(A.*, B.*)$$
$$card((A \bowtie B) \bowtie C) = card(A) \cdot card(B) \cdot card(C) \cdot jsel(A.*, B.*) \cdot jsel(B.*, C.*)$$

$$cost((A \bowtie B) \bowtie C) = card(A \bowtie B) + card((A \bowtie B) \bowtie C)$$
$$card((A \bowtie B) \bowtie C) = card(A) \cdot card(B) \cdot jsel(A.*, B.*) + card(A) \cdot card(B) \cdot card(C) \cdot jsel(A.*, B.*) \cdot jsel(B.*, C.*)$$
$$cost((A \bowtie B) \bowtie C) = card(A) \cdot card(B) \cdot jsel(A.*, B.*)\Big(1 + card(C) \cdot jsel(B.*, C.*)\Big)$$
$$cost((A \bowtie B) \bowtie C) = card(A \bowtie B)\Big(1 + card(C) \cdot jsel(B.*, C.*)\Big)$$

We can compare the above to $cost(A \bowtie (B \bowtie C))$

$$cost(A \bowtie (B \bowtie C)) = card(B) \cdot card(C) \cdot jsel(B.*, C.*)\Big(1 + card(A) \cdot jsel(A.*, B.*)\Big)$$
$$cost(A \bowtie (B \bowtie C)) = card(B \bowtie C)\Big(1 + card(A) \cdot jsel(A.*, B.*)\Big)$$

These equations show that the cost of an early join is distributive, and is applied again for every subsequent join in the join order. A benefit of this is that expensive first joins will be avoided, keeping the estimated cardinality of initial joins low, and reducing the cost of the whole join plan. This section serves only as a simple analysis of a left-deep join plan under the assumption there are always $n-1$ join conditions for n relations. It will be left as future work to analyze how this cost model behaves given bushy trees or more than n join conditions.

3.5.1 Weaknesses

The cost formula breaks down in the presence of only naive table filter estimates of 20% or 0.2. Continuing from the previous example in section 3.5, suppose table B has two foreign keys and is joined with filtered tables A and C. With no statistics for selectivity estimation, $cost(A \bowtie (B \bowtie C)) = cost((A \bowtie B) \bowtie C)$. The join conditions are $A.* = B.*$ and $B.* = C.*$. The cost of the two join orders can be calculated as follows,

$$\begin{aligned}
card(A \bowtie B) &= 0.2_A \cdot card(A) \cdot card(B) \cdot jsel(A.*, B.*)\\
card(A \bowtie B) &= 0.2_A \cdot card(A \bowtie B)^*\\
card((A \bowtie B) \bowtie C) &= 0.2_A \cdot card(A \bowtie B)^* \cdot 0.2_C \cdot card(C) \cdot jsel(B.*, C.*)\\
card((A \bowtie B) \bowtie C) &= 0.2_A \cdot 0.2_C \cdot card((A \bowtie B) \bowtie C)^*
\end{aligned}$$

$$card(A \bowtie B)^* = card(A \bowtie B) \text{ when no base table filters are applied.}$$

$$\begin{aligned}
cost((A \bowtie B) \bowtie C) &= card(A \bowtie B) + card((A \bowtie B) \bowtie C)\\
cost((A \bowtie B) \bowtie C) &= 0.2_A \cdot card(A \bowtie B)^* + 0.2_A \cdot 0.2_C \cdot card((A \bowtie B) \bowtie C)^*
\end{aligned}$$

If a perfect PK-FK relationship is assumed between $B.* = A.*$ and $B.* = C.*$, then $tdom(A.*) = card(A)$ and $tdom(C.*) = card(C)$. In this instance,

$$\begin{aligned}
jsel(A.*, B.*) &= \frac{1}{card(A)}\\
jsel(B.*, C.*) &= \frac{1}{card(C)}
\end{aligned} \tag{3.23}$$

The formula then reduces to

$$cost((A \bowtie B) \bowtie C) = 0.2_A \cdot card(B) + 0.2_A \cdot 0.2_C \cdot card(B) \tag{3.24}$$

Now we will compare this to the $cost(A \bowtie (B \bowtie C))$,

$$\begin{aligned} card(B \bowtie C) &= card(B) \cdot 0.2_C \cdot card(C) \cdot jsel(B.*, C.*) \\ card(A \bowtie (B \bowtie C)) &= 0.2_A \cdot card(A) \cdot card(B) \cdot 0.2_C \cdot card(C) \cdot jsel(A.*, B.*) \cdot jsel(B.*, C.*) \end{aligned}$$

$$\begin{aligned} cost(A \bowtie (B \bowtie C)) &= card(B \bowtie C) + card(A \bowtie (B \bowtie C)) \\ &= card(B) \cdot 0.2_C \cdot card(C) \cdot jsel(B.*, C.*) + \qquad (3.25) \\ &\quad 0.2_A \cdot card(A) \cdot card(B) \cdot 0.2_C \cdot card(C) \cdot jsel(A.*, B.*) \cdot jsel(B.*, C.*) \\ &= 0.2_C \cdot card(B) + 0.2_A \cdot card(B) \cdot 0.2_C \qquad (3.26) \end{aligned}$$

By only applying naive 0.2 selectivity estimates, there is no difference between $cost((A \bowtie B) \bowtie C)$ 3.24 and $cost(A \bowtie (B \bowtie C))$ 3.26. The term $0.2_A \cdot 0.2_C \cdot card(B)$ is present in both equations, the only difference being the selectivity applied on the cardinality of B in the first term (i.e $0.2_C \cdot card(B)$ vs. $0.2_A \cdot card(B)$). If the selectivity of the filters on table A and B were known, or the estimates differed in any way, the cost of each join order becomes different, and a cheaper join order can be selected (provided the estimates are accurate). This cost example shows that accurate selectivity estimation is very important. When no statistics for a table are available, it is not possible to have accurate filter estimation, so this weakness will be present when querying directly from parquet or CSV files. Improvements to selectivity estimation in the absence of statistics will be left as future work.

4

Implementation

Implementation of the cardinality estimation equation in Chapter 3 did not come without challenges. One issue that continually was a problem was ensuring that the cardinality estimate for a set of relations was the same regardless of the join order. The continuation of this problem led to the design of Equation 3.20

Algorithm 1 describes the code used to implement Equation 3.20 including the logic used to calculate $\Gamma(\mathfrak{J})$. Given a set of relations, it will return the estimated cardinality as a result of joining all the relations. Lines 4-6 calculate the cross product of all of the relations, while lines 7-41 calculate what join selectivities to apply. The join selectivies applied are from the minimum spanning tree on the graph created by the relations and the join conditions. The minimum spanning tree is calculated using a greedy strategy that selects at most $n - 1$ *tdoms* that join as many relations as possible given the input. The strategy for finding these algorithms is based on Kruskal's minimum spanning tree algorithm (14). The for loop on line 8 iterates through the filters in decreasing value of *tdoms*. Lines 9 - 14 check to see if the filter joins relations in the requested set. Lines 15-40 then determine if the filter connects relations that are already a part of a current spanning tree, or if the filter starts a separate spanning tree. A separate spanning tree may or may not be joined by a future edge. If not, the spanning trees are multiplied without a connecting join selectivity or *tdom*. This represents that a cartesian product is taking place.

When our cardinality estimator is initialized, the equivalence sets need to be initialized as well. The equivalence sets help determine the *tdoms* defining the join selectivity of an edge. As mentioned in section 3.2, relations in an equivalence set are transitively joined by equality filters. Assuming all the equality joins are FK-PK joins, the relations are assumed to have the same total domain. Calculating the equivalence classes happens

Algorithm 1 Algorithm for estimating the cardinality of a set of relations

```
Require: $Filters \leftarrow$ all filters
Require: $Relations \leftarrow$ relations being joined
 1: $numerator \leftarrow 1$
 2: $selectivity \leftarrow 1$
 3: $connected\_relations \leftarrow []$
 4: for $relation \in Relations$ do
 5:     $numerator \leftarrow numerator \cdot relation.card$
 6: while $connected\_relations[0].size()$ != $relations.size()$ do
 7:     for $filter$ in sort($Filters, filter.tdom$, decreasing) do
 8:         $left\_connected \leftarrow filter.left\_binding.column$ in $relations$
 9:         $right\_connected \leftarrow filter.right\_binding.column$ in $relations$
10:         # if the filter does not connect any relations, continue
11:         if not $left\_connected$ and not $right\_connected$ then
12:             continue
13:         for $subgraph, i$ in $connected\_relations$ do
14:             $left\_connected \leftarrow filter.left\_binding.column$ in $subgraph$
15:             $right\_connected \leftarrow filter.right\_binding.column$ in $subgraph$
16:             if $left\_connected$ XOR $right\_connected$ then
17:                 $subgraph.demonimator *= filter.tdom$
18:                 if $left\_connected$ and not $right\_connected$ then
19:                     $other\_table$ = filter.right_binding.column
20:                 if $right\_connected$ and not $left\_connected$ then
21:                     $other\_table$ = filter.left_binding.column
22:                 for $subgraph2$ in $connected\_relations[i + 1 :]$ do
23:                     if $other\_table$ in $subgraph2$ then
24:                         $subgraph.denominator *= subgraph2.denominator$
25:                         break
26:             if not $left\_connected$ and not $right\_connected$ then
27:                 $new\_subgraph \leftarrow Subgraph(denominator = filter.tdom)$
28:                 $connected\_relations += [new\_subgraph]$
29:                 continue
30: $selectivity \leftarrow$ prod($connected\_relations, x : x.denominator$)
31: $selectivity \leftarrow \frac{1}{selectivity}$
32: return $numerator * denominator$
```

in one pass through the filters before the join order optimization. Every filter connects two equivalence sets and each relation starts in its own equivalence set. With this as a starting point, one can iteratively go through each join condition and determine which existing equivalence sets are joined by the equality condition, and combine them into one. Algorithm 2 outlines the logic used to determine equivalence sets. To determine the total domain of each equivalence set, two strategies can be used. In the presence of distinct column value statistics, the total domain is initialized as the highest reported distinct count among all relations in the equivalence set. When no statistics are available, the total domain is initialized using the lowest cardinality column among the relations in the equivalence set.

Algorithm 2 Algorithm for determining equivalence sets

```
Require: $Filters \leftarrow$ all filters
 1: $equivalent\_sets \leftarrow []$
 2: for $filter$ in $Filters$ do
 3:     $matching\_set\_indexes \leftarrow []$
 4:     # Find the indexes of the equivalence sets the filter combines.
 5:     for $set$, $i$ in $equivalent\_sets$ do
 6:         if $filter.left\_column$ in $set$ then
 7:             $matching\_set\_indexes$ += $[i]$
 8:         else
 9:             if $filter.right\_column$ in $set$ then
10:                 $matching\_set\_indexes$ += $[i]$
11:     # Combine the equivalence sets based on the indexes gathered
12:     if length($matching\_set\_indexes$) == 2 then
13:         $es\_1 = matching\_set\_indexes[0]$
14:         for relation $\in$ $matching\_set\_indexes[1]$ do
15:             $equivalent\_sets[es\_1]$ += [relation]
16:     if length($matching\_set\_indexes$) == 1 then
17:         $equivalent\_sets[i]$ += $[filter.left\_column, filter.right\_column]$
18:     if length($matching\_set\_indexes$) == 0 then
19:         $equivalent\_sets$ += $[filter.left\_column, filter.right\_column]$
```

The setup operations required to initialize the new DuckDB join order optimizer are described in 3. First, the selectivity estimation is performed. If a column that is being joined has a filter, then the *tdom* of that column must also change. DuckDB implements push-down filters, so all columns in an equivalence set are filtered before they are joined.

When the filter affects all base tables, the excluded values no longer show up as intermediate tuples and therefore should not be considered in the *tdom* of the cardinality estimation equation.

Once selectivity estimation is completed, equivalence sets and *tdoms* are calculated. Finally, the join order optimizer is called.

Algorithm 3 Join Order Optimizer Setup and Execution

```
1: Selectivity_Estimation();
2: Init_equivalence_sets();
3: Init_Tdoms();
4: Join_Order_Optimization();
```

5 Experimental Setup

The new DuckDB join order optimizer described in Chapter 3 will be compared against PostgreSQL, SQLServer, and an older version of DuckDB with the original join order optimizer. The evaluation will compare the number of intermediate tuples generated, referred to as the intermediate cardinality count for the current and future sections, and end-to-end execution times. For end-to-end execution times, only the DuckDB systems will be compared, as the rest of the databases have different execution engines, making accurate comparisons of execution times nearly impossible. By comparing the intermediate cardinality counts between all systems, the evaluation can focus on the quality of the plans selected by each join order optimizer.

Intermediate cardinality counts for each database system will also be compared against explicit plans, which represent a baseline or control. Comparisons of intermediate cardinality counts will be normalized, so if a DBMS has an intermediate cardinality count twice that of the explicit plan, the reported value is 2. The baseline plans for the join order benchmark are taken from plans produced by *Hertzschuch et al.* in the paper *Simplicity Done Right for Join Ordering.* These plans use precise selectivity estimates and the maximum frequency statistic to determine a strict upper bound on the true cardinality of a join. For the TPC-h benchmark, plans produced by PostgreSQL will be used as the baseline. (1) shows that the plans produced by PostgreSQL and the estimated cardinalities are very accurate.

Each benchmark evaluation is split into two sections; the evaluation of cardinality sums between each tested database system, and the evaluation of execution times between DuckDB with the old join order optimizer and the new join order optimizer, referred to as “duckdb-old”, and “duckdb-tdoms” for the rest of the thesis. When comparing end-to-end execution times, the results will be normalized to the system that had a higher end-to-end execution time. Absolute execution times will be explained and presented when necessary.

5.1 Benchmarks

TPC-H

The TPCH benchmark is a decision support benchmark meant to simulate ad-hoc queries on data one would find in the real world. The benchmark is made up of 22 queries that are more complex than the average OLTP transactions. The underlying data of the TPC-H benchmark is meant to resemble the data of a business that operates on an international level. The TPC-H data set can be scaled up to 100TB by specifying a scale factor of 100000, but our evaluation is only performed on a data set with a scale factor of 10, meaning 10GB of data. The optimal join order for each query remains the same regardless of scale factor, and there is minimal skew and correlation. The TPC-h benchmark is not a very join-heavy benchmark, with only 6 joins in the most join-heavy query (20).

Join Order Benchmark

The join order benchmark is a suite of 113 queries that can be run on the IMDB dataset. Since the IMDB dataset is a real-world data set, it is full of correlations and non-uniform data distributions. These qualities make accurate cardinality estimation much more difficult. The data set is a snapshot from May 2013 and contains 3.1GB of data in CSV format. The queries focus specifically on join ordering, with queries having anywhere between 3 and 16 joins, with an average of 8 per query (1).

Join Order Benchmark on Parquet Files

To measure the quality of the new join order optimizer in the absence of statistics, the join order benchmark will be run on parquet files. To produce the parquet files, the JOB dataset will be converted to parquet. The same comparison metrics will be used, intermediate cardinality counts, and end-to-end execution time.

5.2 Hardware

The execution times presented in the following section were measured on a MacBook Pro (14-inch, 2021) with an M1 chip, 32 GB of memory, and 500 GB of storage. To identify queries with significant results, a difference of 100ms (or 0.1 second) will be required for a difference in end-to-end execution time. A difference of 100ms was chosen because it lies below the human perception threshold (10).

5.3 Result Classifications

Execution times will only be compared between DuckDB with the old join order optimizer, and DuckDB with the new join order optimizer. This allows us to isolate the impact of the new cardinality estimator on the end-to-end execution times. Therefore, for the current and following sections, DuckDB-tdoms refers to DuckDB with the new join order optimizer and DuckDB-old refers to DuckDB without the join order optimizer improvements.

Our research question focuses on cardinality estimation as the exclusive input to a cost model of a join order optimizer. Therefore we want to evaluate the quality of the plans produced by the join order optimizer by using intermediate cardinality counts and end-to-end execution times. With this in mind, we will focus on query results that meet characteristics described in the following categories,

C1 Queries where the new join order optimizer produces a lower intermediate cardinality count, but a higher end-to-end execution time.

C2 Queries where the new join order optimizer produces a higher intermediate cardinality count, but a lower end-to-end execution time.

C3 Queries where the new join order optimizer results in a lower end-to-end execution time. Intermediate cardinality counts for all queries in this category are guaranteed to be lower for plans generated by the new join order optimizer, otherwise the queries would end up in **C2**.

In order to reduce noise and identify noticeably faster queries, we constrain the above groups to require a difference in end-to-end execution time of at least 100ms or 0.1 second. All queries are run 10 times, and the median time is reported in the results in an effort to reduce noise.

6

Evaluation

6.1 TPCH Benchmark

Intermediate Cardinality Counts

Our first evaluation will be measuring intermediate cardinality counts of queries in the TPC-h benchmark. As mentioned earlier, work done in (1) shows that PostgreSQL's cardinality estimator is especially good for the TPC-H benchmark, so the PostgreSQL plans will be considered the baseline plans. Queries 15 and 18 are excluded from our results since PostgreSQL and SQLServer optimize the query using a "Group By" pushdown optimization, while DuckDB does not. Query 19 is excluded as well because DuckDB fails to push down a table filter in the same way as PostgreSQL and SQLServer. Since the Group By operator and table filter are not pushed down, the intermediate results for queries 15, 18, and 19 are skewed heavily in favor of PostgreSQL and SQLServer, hence their absence from our results.

Table 6.1: Comparison of intermediate cardinality counts of query plans chosen and executed by different database systems on the TPC-H benchmark. The intermediate cardinality counts are normalized to the explicit plan for each query. If a system produces twice as many intermediate tuples as the explicit plan, the normalized value is 2, while the explicit is always 1.

sys	max	avg	median
DuckDB-tdoms	18.6	4.674	1.000
DuckDB-old	18.60	6.817	3.105
explicit (PostgreSQL)	1.0	1.0	1.0
sqlserver	3.98	0.91	1.0

Table 6.1 presents a summary of the TPC-H intermediate cardinality counts of four database systems. The results show that the new join order optimizer improves intermediate cardinality counts across the min, max, avg, and median. If we analyze our results and look for queries that satisfy properties in the three categories mentioned in section 5.3, we only get q02, q03, q08, q09, and q21 showing up in C3. The intermediate cardinality counts for these queries are plotted in Figure 6.1. The largest improvement is query 9, where there is an improvement of almost 95%. The results of all intermediate cardinality counts (including for the excluded queries q15, q18, and q19) can be found in Chapter 9.

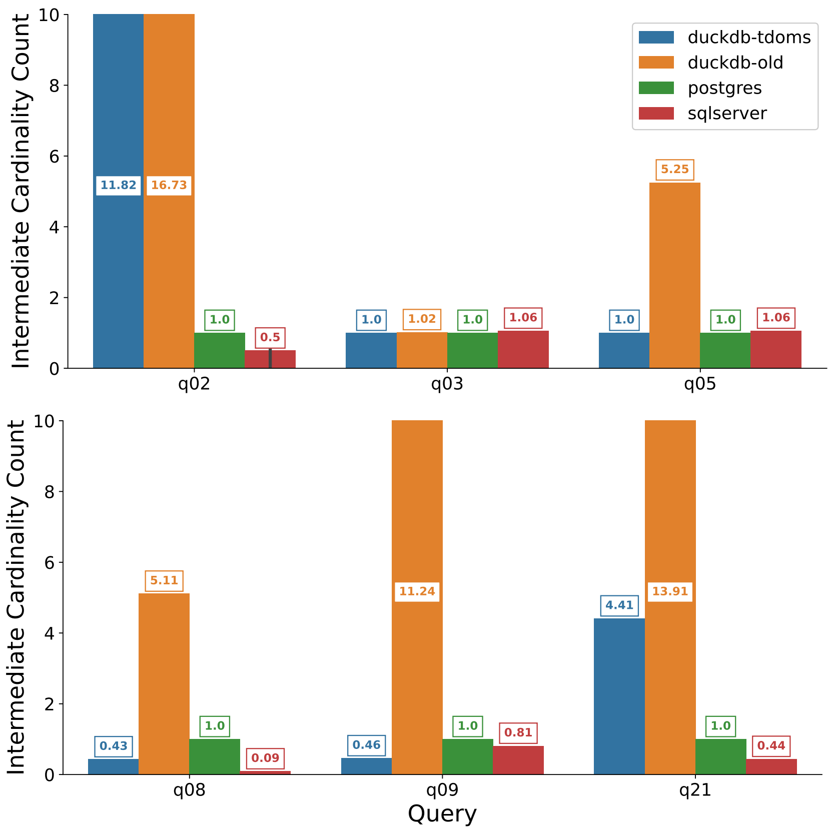


Figure 6.1: TPC-H queries where DuckDB-tdoms has a lower intermediate cardinality count than DuckDB-old. PostgreSQL and SQLServer are also provided as reference. All intermediate cardinality counts are normalized to the PostgreSQL results.

End-to-End Execution Times

Table 6.2 presents the end-to-end execution times of running the TPC-H benchmark on DuckDB with the new join order optimizer vs the old join order optimizer on scale factor 10. The new join order optimizer significantly improves the maximum end-to-end execu-

Table 6.2: Summary of end-to-end execution times of TPC-H queries on scale factor 10.

sys	max	avg	median
DuckDB-tdoms	1.163s	0.397s	0.300s
DuckDB-old	4.440s	0.593s	0.293s

tion time, with a reduction of 74%. Figure 6.2 shows the normalized end-to-end execution results for all queries that had faster execution times. The values indicate that a lower intermediate cardinality count does indeed translate to a faster execution time. The results for all end-to-end execution times (including for the excluded queries q15, q18, and q19) can be found in Chapter 9.

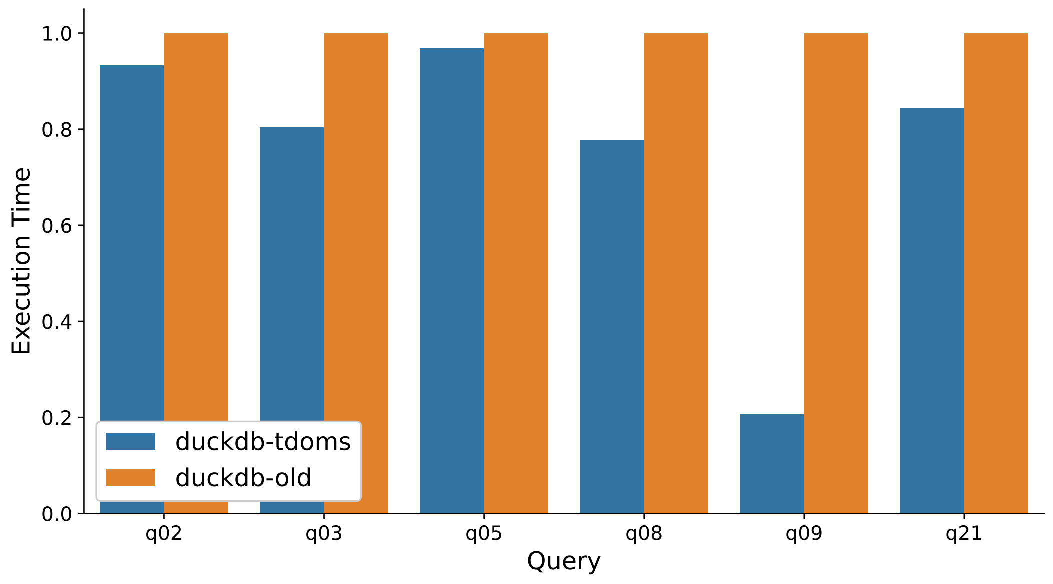


Figure 6.2: TPC-H queries where DuckDB-tdoms has faster end-to-end execution results. Each result is normalized to the end-to-end execution time of DuckDB-old. For each query in this figure, DuckDB-tdoms produced a lower intermediate cardinality count during execution.

6.2 Join Order Benchmark

Intermediate Cardinality Counts

Table 6.3 and Figure 6.3 summarize the performance of PostgreSQL, SQLServer, DuckDB-old, DuckDB-tdoms, and the explicit plans gathered from (21). The results show that JOB queries run on DuckDB-tdoms result in a lower intermediate cardinality count compared to DuckDB-old.

Table 6.3: Comparison of intermediate cardinality counts produced by query plans chosen and executed by different database systems. Intermediate cardinality counts are normalized to the explicit plan per query.

sys	max	avg	median
PostgreSQL	346.04	11.61	1.00
explicit	1.00	1.00	1.00
SQLserver	368.01	8.71	0.73
DuckDB-old	18874.70	391.86	14.085
DuckDB-tdoms	327.89	16.91	1.005

On average, DuckDB-old produced an intermediate cardinality count almost 400x greater than the explicit plans. DuckDB-tdoms plans resulted in an intermediate cardinality count 16.9x greater than the explicit plans, a reduction of more than 95%. A normalized median of 1.005x is a good indicator that even though some outliers skew the average, DuckDB-tdoms produces plans with comparable costs to the explicit and PostgreSQL plans at least half the time. The intermediate cardinality counts of every query can be found in Chapter 9.

Some of the intermediate cardinality counts reported by PostgreSQL, SQLServer, and DuckDB are less than the explicit plans. This can be for a few reasons. The explicit plans enumerate a much smaller subset of the plan space compared to DuckDB and potentially PostgreSQL and SQLServer. For this reason, the explicit plans do not find the optimal join orders. The explicit plans do, however, avoid disastrous plans by utilizing the maximum frequency statistics and precise selectivity estimation to produce a guaranteed upper bound for the true cardinality of a join. DuckDB-tdoms, DuckDB-old, PostgreSQL, and SQLServer do not make use of guaranteed upper bounds, and therefore produce plans that can still blow up.

18 queries have a higher intermediate cardinality count under DuckDB-tdoms compared to DuckDB-old. There is a lot of variability in these values and after some investigation, it was possible to identify specific reasons for why each query has a worse intermediate cardinality count under DuckDB-tdoms compared to DuckDB-old. Below is a more detailed explanation of each reason, and Table 6.4 labels each query with a reason as to why it performed worse with the new join order optimizer.

F1 The filter `company_name.country_code = '[us]'` is predicted to match 1488 tu-

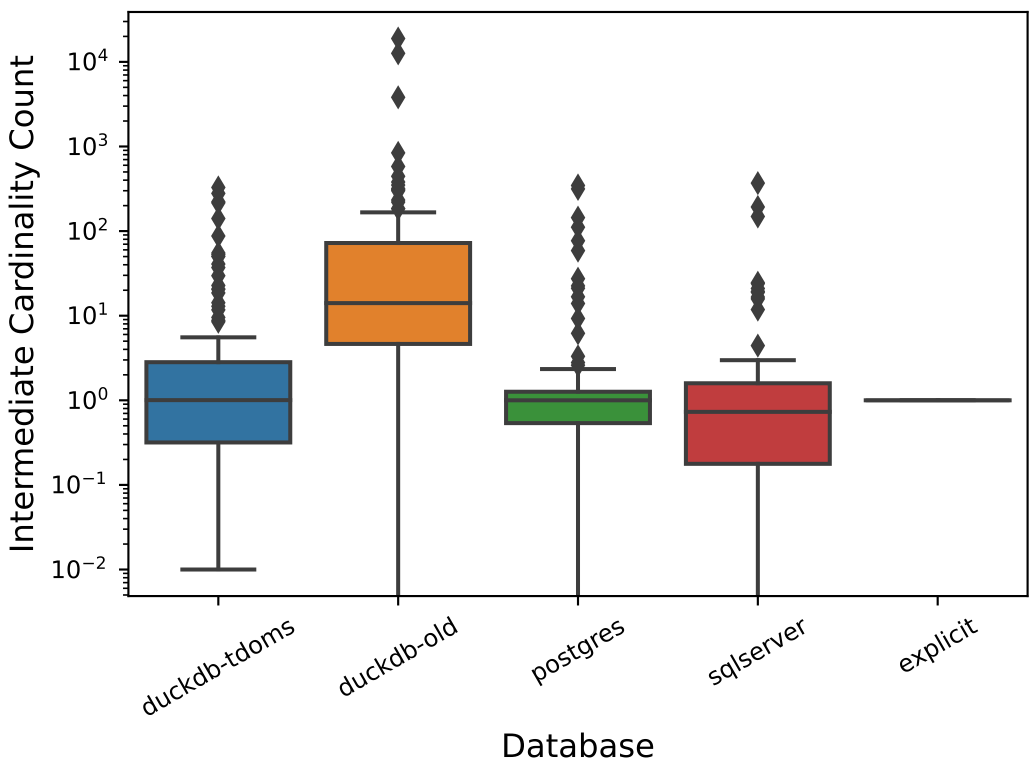


Figure 6.3: Box plot of intermediate cardinality counts for all JOB queries. The intermediate cardinality count for each query is normalized to the explicit plan. In these tests, the queries are run on base tables. Note that this is a **log** scale and not linear.

ples. In reality, the filter matches 84,842 tuples, which is 57x the predicted amount. With a lower predicted amount of tuples resulting from the filter DuckDB-tdoms will join the `company_name` base table earlier in the join process, causing more intermediate tuples to be produced than originally predicted.

F2 Filter estimation logic for `keyword IN (...)` filters has not yet been implemented. This logic is the same as `keyword = '...' OR keyword = '...'` as explained in section 3.4. Currently, a selectivity of 20% is applied which is consistently an overestimation for these types of filters.

F3 The filter `cast_info.note="(voice: English version)"` is predicted to match 362 tuples of the `cast_info` table by DuckDB-tdom's naive filter estimation. In reality 93,095 tuples are matched which is 257x the predicted amount. Similar to **F1**, filters that are predicted to be extremely selective result in the `cast_info` base table being one of the first joined in the join tree.

F4 Many of the filters are of the form `contains(title, "money")`. These types of filters employ a naive estimation of 20%. Queries in Table 6.4 with reason **F4** have many `contains` filters and each one is extremely selective. The old optimizer gets lucky and puts these selections at the bottom of the join tree. The new optimizer puts the selections closer to the top of the join tree.

S1 Only U.S companies are queried as part of a larger query. The U.S has many more movie companies than other countries, so naturally, when a query searches for U.S-based movie companies, it is incorrect to assume 20% of all companies are from the U.S. Queries with **S1** violate the independence assumption, and there is skew.

S2 Movie Ids with some rating are requested, which results in a join between `movie_info_idx` and `info_type`. This join has skew. 33% (459925) of the tuples in the `movie_info_idx` table have `info_type_id='rating'`. When a filter like `movie_info_idx.info > 6.0` is applied on the `movie_info_idx` table, the selectivity estimator assumes 20% of the tuples match, or 276007 tuples. This naive estimate is not the reason why our cardinality estimation is incorrect, but when joining on `movie_info_idx.info_type_id = info_type.id where info_type.info_type_id='rating'` our independence assumption negatively affects our estimations. We assume that all 113 `info_type.id` values are equally distributed across the 276007 tuples from the `movie_info_idx` table. In reality, since the filter `movie_info_idx.info > 6.0` has already been applied, most of the resulting 276007 tuples are already ratings. A query like

```
SELECT * FROM
movie_info_idx, info_type
WHERE movie_info_idx.info_type_id = info_type.id
AND info_type.info_type_id='rating';
```

predicts a result of 5018 tuples, when in fact the result is closer to 259848 tuples (these are the exact numbers for JOB query 14b, which requests all Movie IDs with a rating > 6.0 with the word “murder” in the title).

Table 6.4: JOB Queries producing higher intermediate cardinality counts when run on DuckDB-tdoms vs. DuckDB-old.

query	cost DuckDB-tdoms	cost DuckDB-old	reason(s)
06d	2,770,110	1,985,119	F2
07c	2,060,924	1,265,777	F2
08a	49,980	49,594	F3
08b	37,036	9,501	F3
09a	821,906	543,847	F1
09b	285,908	177,782	F1
09c	2,045,156	1,187,818	F1
09d	4,013,646	2,655,294	F1
11b	10,915	463	S1
12a	52,733	27,402	F1&S1
12b	1,163,408	534,533	F1&S1
12c	468,082	288,545	F1&S1
13b	1,159,042	119,728	F1&S1
13c	1,154,450	27,776	F1 & S1
14b	259,910	1,171	S2
19c	2,984,681	2,978,260	F1 & S1
22c	3,560,704	2,089,391	F1 & S2
22d	7,242,514	4,487,769	S2

End-to-End Execution Times

If we compare end-to-end execution times we see some favorable results. In this section, we are only comparing DuckDB-tdoms and DuckDB-old so we can isolate the effects of the new cardinality estimator in our cost model and join order optimizer. The results from Table 6.5 and Figure 6.4 show that overall end-to-end query times improved for the join order benchmark by an average of 25%. Comparing these improvements to the average improvements in intermediate cardinality counts, the difference is not as large as one would hope. Taking a look at Figure 6.4, however, we can see that DuckDB-tdoms is more robust as there are only three join order benchmark queries that take longer than half a second to complete, while DuckDB-old has 11 queries that take longer than half a second to complete.

Table 6.5: End-to-end execution time statistics on Join Order Benchmark queries run on base tables.

sys	min	max	avg	median
DuckDB-tdoms	0.005s	1.400s	0.167s	0.122s
DuckDB-old	0.026s	2.358s	0.224s	0.135s

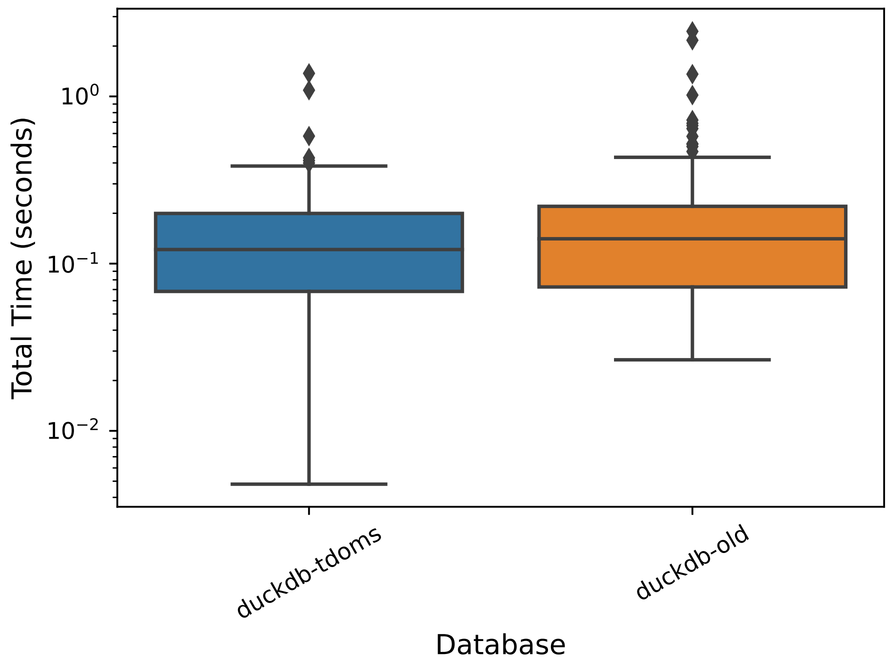


Figure 6.4: A box plot of end-to-end execution times for JOB queries. Note that this is a **log** scale and not linear.

We can now take a closer look at individual queries and see which ones fall into the three categories mentioned in section 5.3.

Figure 6.5 shows the timing results of all queries in category **C1**, which are queries that have a smaller intermediate cardinality count when run on DuckDB-tdoms, but take longer to execute when compared to DuckDB-old. Queries 08c and 08d are the only two queries in the category **C1**. For these queries, DuckDB-tdoms produces a plans that have a foreign-key foreign-key (FK-FK) join, which is not easy for the execution engine to handle. In a FK-FK join, the build side can have duplicate values, so when each value is hashed and placed into the bucket, hash collisions will be chained using linked lists. This will make the probing phase take longer as well, as the whole linked list must be traversed, requiring a random read from memory for each entry.

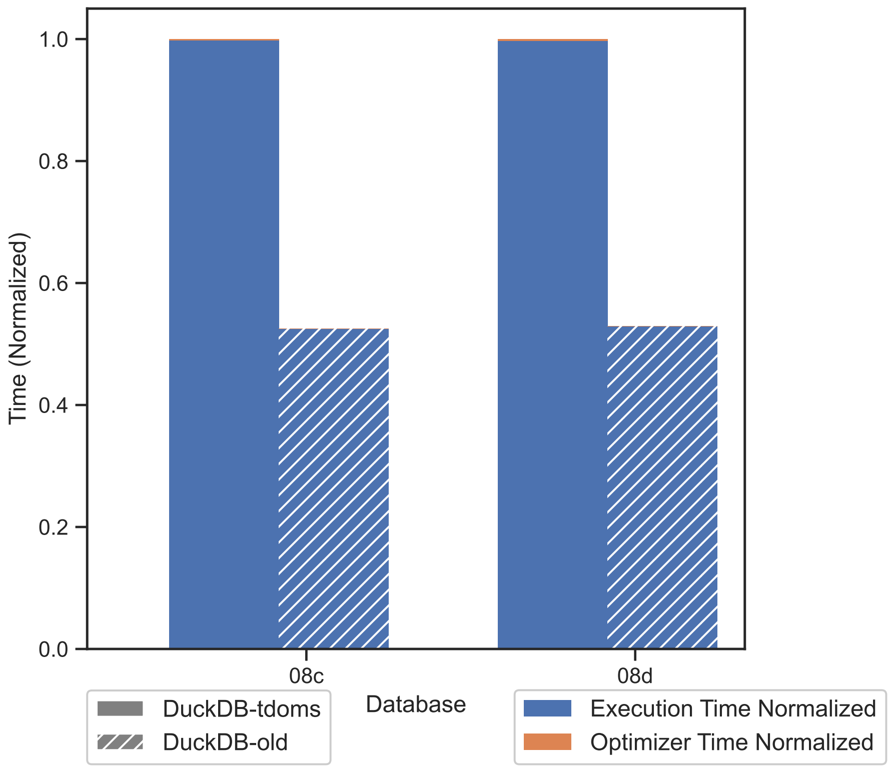


Figure 6.5: All JOB queries where DuckDB-tdoms had a lower intermediate cardinality count than DuckDB-old, but higher end-to-end execution time.

No JOB queries are in category C2, meaning no plans were found with higher intermediate cardinality counts and lower end-to-end execution times.

Figure 6.6 shows the normalized end-to-end execution times of all queries in category **C3** mentioned in Chapter 5. Queries in **C3** have lower intermediate cardinality count and faster end-to-end execution time when run on DuckDB-tdoms when compared to DuckDB-old. For each query, the intermediate cardinality count of DuckDB-tdoms is consistently more than 10x less than the intermediate cardinality counts of DuckDB-old. The absolute end-to-end execution times and cardinality counts for all queries can be found in Chapter 9.

6.3 Join Order Benchmark on Parquet Tables

Intermediate Cardinality Counts

We now take a look at the performance of our new join order optimizer when running join order benchmark queries on parquet files. DuckDB will not have access to table statistics in these tests, and therefore will use the lowest cardinality of a table as the *tdom* of an

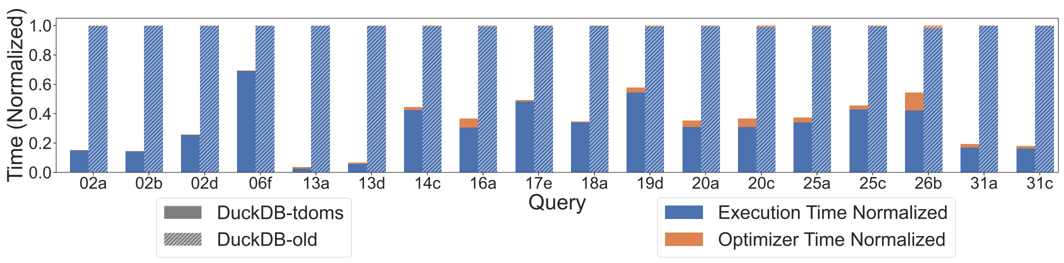


Figure 6.6: Execution and optimizer times of select JOB queries on DuckDB-tdoms and DuckDB-old. For every query in the plot, DuckDB-tdoms is at least 100ms faster than DuckDB-old. Times are normalized per query to the end-to-end execution time of DuckDB-old.

equivalence set. For these experiments, all tables in the IMDB dataset were exported to parquet files. The same experimental setup was used, with the only difference being that the JOB queries were amended to query directly from the parquet files.

Table 6.6: Comparison of intermediate cardinality counts produced by query plans selected by DuckDB-tdoms and DuckDB-old.

sys	max	avg	median
DuckDB-tdoms (on parquet files)	4007.07	133.814	1.595
DuckDB-old	18874.70	391.861	14.085
explicit	1.00	1.000	1.000
DuckDB-tdoms (on base tables)	327.89	16.91	1.005

Table 6.6 and Figure 6.7 show the results of running the join order benchmark on parquet files instead of base tables. Results for all queries can be found in Chapter 9. While these results are not as favorable as the results in section 6.2, there is still an improvement. The difference between DuckDB-tdoms on parquet files and DuckDB-tdoms on base tables comes from the fact that fewer statistics are available in parquet files, providing the query optimizer with less information to estimate selectivities on base tables.

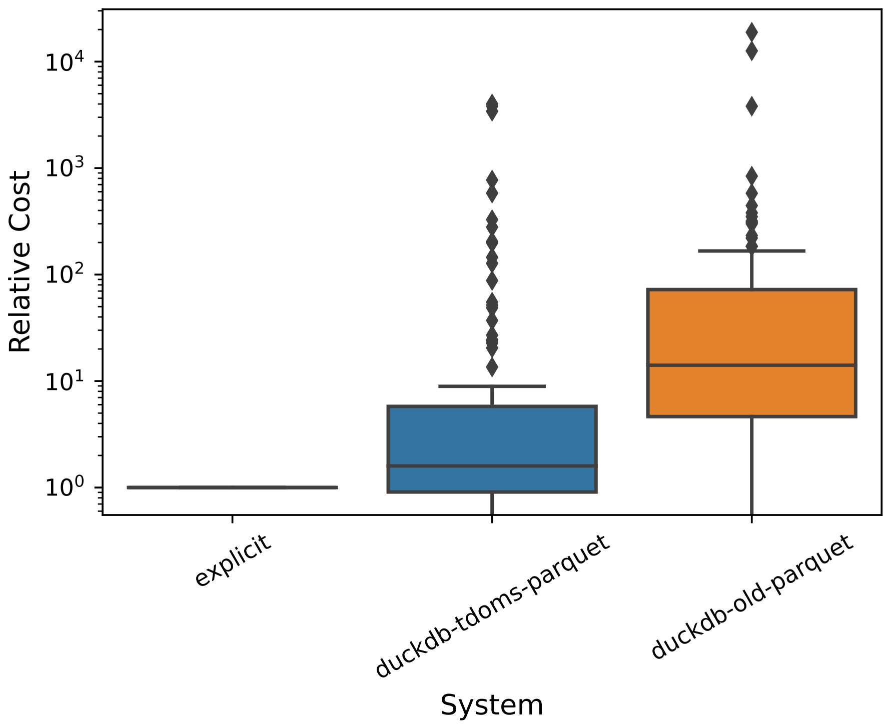


Figure 6.7: Box plot of intermediate cardinality counts of JOB queries executed on parquet files. Note that the figure has a **log** scale and not a linear scale.

End-to-End Execution Times

Table 6.7 clearly shows that the new join order optimizer selects plans with lower end-to-end execution times for JOB queries run on parquet files. Figure 6.8 further improves these results and shows that there are no significant outliers. No job queries on parquet data take longer than 1 second to execute when run on DuckDB-tdoms, while at least five job queries take longer than 1 second to execute when running on DuckDB-old.

Table 6.7: Summary of end-to-end execution time when running join order benchmark queries on parquet data.

sys	min	max	avg	median
DuckDB-tdoms-parquet	0.025s	0.817s	0.343s	0.343s
DuckDB-old-parquet	0.058s	2.538s	0.417s	0.346s

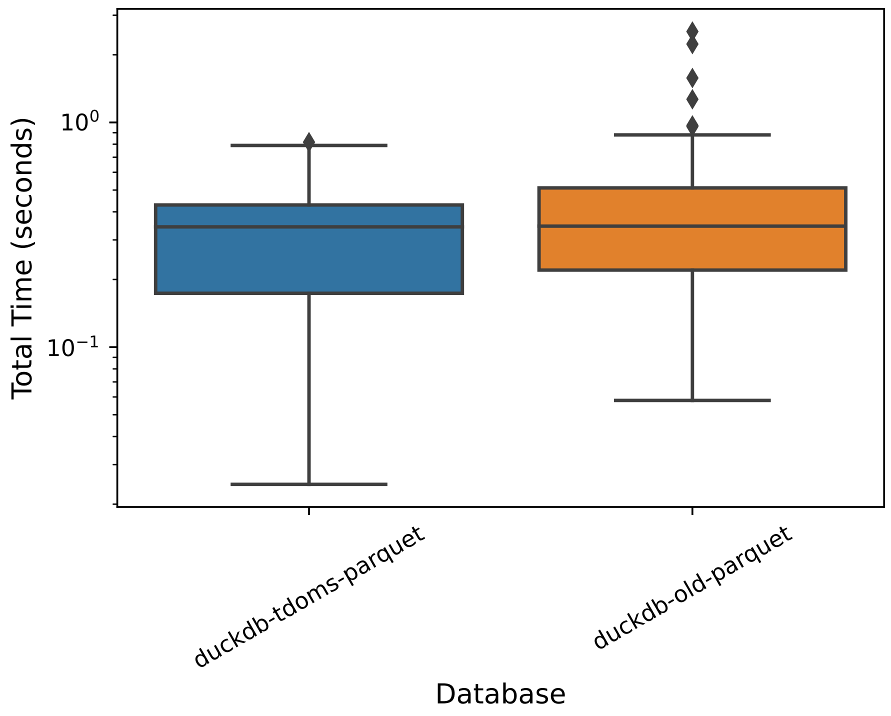


Figure 6.8: Boxplot of total time statistics for running JOB queries on parquet files. Note that this is a **log** scale and not linear scale.

When running the JOB queries on just parquet files, queries fall into the following categories described in section 5.3.

C1 Queries where the new join order optimizer produces a lower intermediate cardinality count, but a higher end-to-end execution time.

Queries 09d, 20b, 29a, 29b, 29c fall into this category. Figure 6.9 shows that for queries 29[a-c], the optimizer is the cause for the increased end-to-end execution times. An analysis of increased optimizer times can be found in section 6.4. The intermediate cardinality count difference between DuckDB-tdoms and DuckDB-old for query `09d` is 201, so the end-to-end execution difference is considered to be noise from reading parquet files. Query `20b` has a FK-FK join. As mentioned earlier, FK-FK hash joins take longer to complete due to duplicate values in the build side.

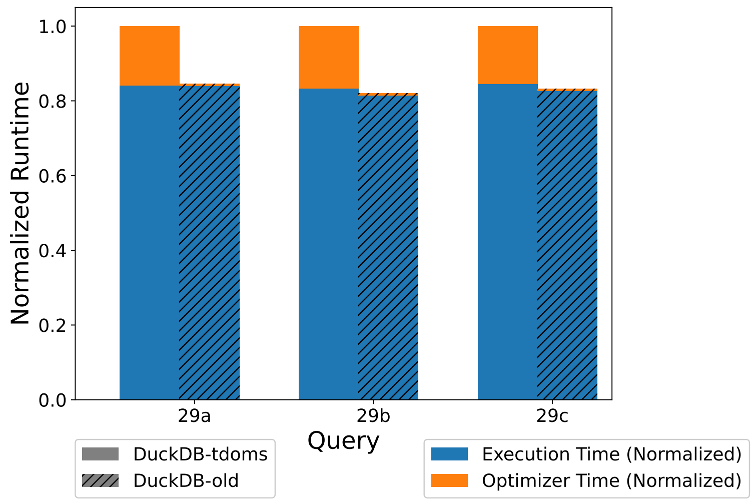


Figure 6.9: Optimizer and Execution run-times for JOB queries run on parquet files. The queries in this graph have lower intermediate cardinality counts, but higher end-to-end execution times.

C2 Queries where the new join order optimizer produces a higher intermediate cardinality count, but a lower end-to-end execution time.

Query 07c is in this class. Analyzing the cardinalities of the build-side of each join in query 07c, we see that DuckDB-old has 4x the amount of build-side tuples when compared to DuckDB-tdoms. Table 6.8 shows a summary of the statistics for query 07c. The intermediate cardinality count of DuckDB-tdoms is 1.5x that of DuckDB-old, and the average hash time was measured using one thread to avoid confusing execution results due to parallelism. Smaller build side cardinalities lead to faster hash join executions because the CPU will not have to swap as many hash buckets into CPU cache.

Table 6.8: Build side, probe side, and hash join execution statistics for q07. We see that although DuckDB-tdoms has 1.5x the intermediate cardinality count of DuckDB-old, the build side cardinality plays a major role in how fast the query can execute.

sys	cost	total build side cardinality	Average time to execute a hash join (seconds)
DuckDB-tdoms-parquet	1891945	205461	0.047s
DuckDB-old-parquet	1265777	827537	0.167s

C3 Queries where the new join order optimizer results in a lower end-to-end execution time.
22 queries are in this class, namely 02a, 02b, 02c, 02d, 05a, 05b, 06f, 07c, 08c, 08d, 13a, 13d, 17e, 17f, 18a, 19d, 20a, 20c, 25a, 25c, 31a and 31c. Each query has a lower cardinality count as well, otherwise, it would be in class **C2**.

6.4 Number of Relations vs. Optimizer Time

Figure 6.6, and Figure 6.9 show that the optimizer is taking more time for some of the queries. One factor that determines how long the optimizer takes is the number of relations present in the SQL query. Figure 6.10 compares the optimizer time of DuckDB-old with DuckDB-tdoms, plotting average optimizer time as a function of the number of relations for all the JOB queries.

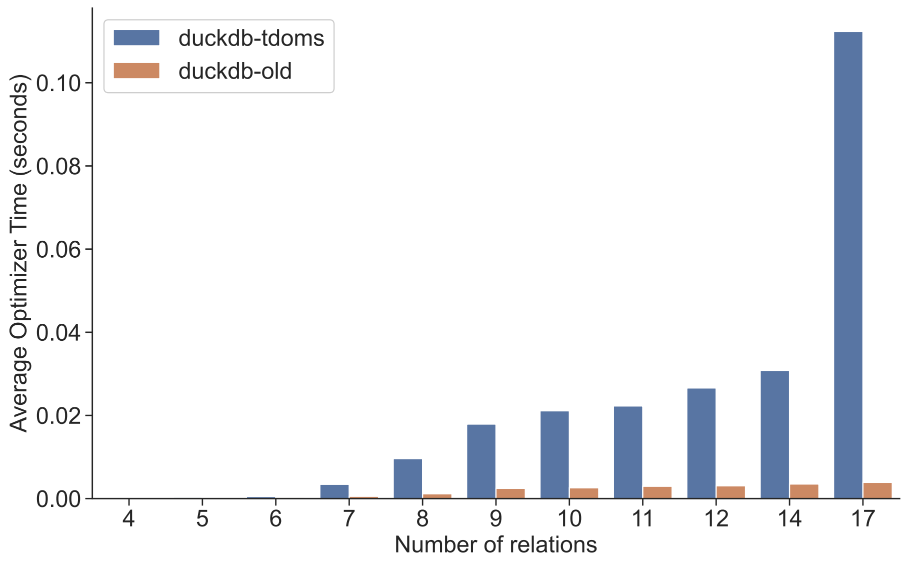


Figure 6.10: Optimizer time plotted as a function of the number of relations in a query. The optimizer timings are from running JOB queries on base tables.

Figure 6.10 plots optimizer time as a function of the number of relations in the query. The results show that as the number of relations grows, the longer it takes for the optimizer to finish. This is to be expected as there are $n!$ possible join orders, and the plan enumerator in DuckDB-tdoms can emit an exponential number of pairs of relation sets as join candidates. To avoid enumerating all possible pairs, the join order optimizer only emits 10,000 pairs, originally 2,000 pairs in DuckDB-old. This is one cause of the slowdown in the optimizer. Another reason is that every call to the cardinality estimator with a new distinct set of relations must call the minimum spanning tree algorithm, which is $O(n \cdot log(n))$ in the number of join conditions. This call was not present in DuckDB-old. When the plan enumerator finishes emitting 10,000 pairs, a greedy plan enumerator algorithm takes over, which is $O(n^3)$ in regards to the number of relations, so cubic growth occurs. The greedy algorithm was already present in DuckDB-old, so it does not contribute to the slowdown.

While the optimizer time takes longer, normalizing it to the execution time does not help with analysis. The optimizer time is solely dependent on the number of relations. The size of the data, and therefore the time spent executing the query plan, does not affect the optimizer time. As long as the number of relations remains the same, the optimizer will have the same run-time.

It is important to note that when the optimizer switches from dynamic programming plan enumeration to approximate plan enumeration, no information from the dynamic programming table is used by the approximate plan enumerator. Even if the dynamic programming table has a low-cost plan for (n-1) relations, the approximate join order optimizer will not consider that plan, and will instead build its own.

7

Discussion

In Chapter 1, three research goals were introduced in an effort to improve the quality of the DuckDB join order optimizer, and subsequently the query optimizer. In Chapter 3, a minimal cardinality estimator was introduced, requiring only the number of distinct values in a column. A simple cost model was also described, requiring only cardinality estimates as input.

The intermediate cardinality counts and end-to-end execution times from running the TPC-h benchmark clearly show that DuckDB-tdoms selects better plans than DuckDB-old on average. 6 of 19 queries had smaller intermediate cardinality counts and execution times. Query 9 had a intermediate cardinality count improvement of 95%, and a run-time improvement of 80% (Figure 6.1 and Figure 6.2). No queries in the TPC-H benchmark regressed, indicating that for queries that are not join heavy, the new join order optimizer did not regress the end-to-end performance of DuckDB.

The results produced from the Join Order Benchmark in Chapter 6 clearly show that DuckDB-tdoms selects plans that produce smaller intermediate cardinality counts than DuckDB-old on average. This is true when no statistics are available, and when statistics are available. The intermediate cardinality counts of DuckDB plans run on base tables are comparable to state-of-the-art systems as well, as min, max, average, and median intermediate cardinality counts are all within an order of magnitude of the same results produced by SQLServer and PostgreSQL (Table 6.3).

The difference in maximum end-to-end execution time for the Join Order Benchmark results shows that the new join order optimizer is more robust, and avoids disastrous plans selected by the previous join order optimizer. Figure 6.8 provides evidence of this when running benchmarks on parquet files, as the maximum execution time decreased from 2.5 seconds to less than 1 second. Figure 6.4 provides stronger evidence of this when running the Join Order Benchmark on base tables by reducing the end-to-end execution time from 2.5 seconds to less than 1.5 seconds. Removing two outliers (<2% of queries) that have expensive FK-FK joins, the highest execution time for a Join Order Benchmark query on base tables becomes 0.57 seconds.

The results in Table 6.4 showed 18 queries that had higher intermediate cardinality counts. While these values were a result of the naive selectivity estimation logic and skew within the database, only 11 of the queries resulted in higher end-to-end execution times. From these findings, we can conclude that intermediate cardinality counts do not **directly** correlate to end-to-end execution times for any one query. It is clear that physical factors such as build-side cardinalities, CPU memory, and cache misses affect the end-to-end execution time of certain queries, as well as optimizer time. Table 6.3 and Table 6.5, however, show that on average, a lower intermediate cardinality count will lead to a lower end-to-end execution time.

Another result worth mentioning is the effect of the changes on the optimizer run-time. Figure 6.10 clearly shows that the new join order optimizer takes more time compared to the old. Although this is the case, Table 6.5 justifies the increased optimizer time, since the average end-to-end execution time is less.

Chapter 6 presents more than enough data to prove that the research goals were achieved. For 15% of the join order benchmark queries, the new join order optimizer produces plans that are significantly better in cardinality counts and end-to-end execution times. For the 78% of the queries, there is no difference in end-to-end execution times. For the last 7% of the join order benchmark queries, the new join order optimizer produces plans that are worse, but their poor performance is outweighed by the improvements in other queries. The poorly performing queries are a result of skew and correlation in the database, which is something our cardinality estimator has not yet been made to handle. Table 6.4 describes why each query had a higher intermediate cardinality count with the new join order optimizer. Each one of these reasons can be labeled as an item for future work, which we describe below.

Future Work

- **Faster Cardinality Estimation**
  When predicting cardinalities we calculate the spanning tree of the subset of relations and join conditions. This can be made more efficient by caching spanning tree results from subsets and analyzing how many edges were added per relation.

- **Faster Plan Enumeration when Given Many Relations**
  If there are more than 10 relations, and a number of them have smaller cardinalities, the optimizer can immediately run the approximate heuristic instead of the dynamic programming plan enumeration algorithm. This will improve optimizer overhead. Another improvement in this area would be implementing the work of *Neumann et al.* in *Adaptive Optimization of Very Large Join Queries* where join order optimization can be scaled to thousands of queries (22).

- **Support for the `IN` Clause**
  As mentioned in Chapter 6, there is no selectivity estimation logic for filters like `IN ("...", ...)` beyond the naive 20% filter. Extending the selectivity estimator to estimate these clauses should improve the quality of the optimizer as well.

- **Sampling, Histograms, and Statistics** Selectivity estimation can be improved by adding samples, histograms, or more statistics. Work in (23) can estimate the maximum frequency of values in a column in a dynamic database. If accurate maximum frequency values are added, we can produce an accurate maximum upper bound on the cardinality of a join.

- **Selectivity Estimation in the Absence of Statistics**
  As mentioned in subsection 3.5.1, when no statistics are present for a column, the naive 20% filter is used, which will make the cost model less accurate. With more accurate selectivity estimation by gathering statistics on the fly, the cost model will also become more accurate and the quality of the query plans will improve.

The statements made in this section are drawn from evidence produced by running the join order optimizer on benchmarks that either have small scale and real-world skew, or large scale with synthetic skew. With more time, it would be interesting to test the new join order optimizer on benchmarks with larger data sets and real-world skew.

8

Related Work

The paper *Access Path Selection in a Relational Database Management System* was the first paper to introduce the concept of query optimizers with plan space enumeration, cardinality estimation, and a cost model. The plan enumerator in the 1970 paper produces all left-deep join plans. For every join, a cost is calculated based on the estimated cardinality of the join, the types of table scans (when joining base tables), the number of page fetches, and CPU usage. The paper includes selectivity estimation techniques for equality joins and comparison joins. The selectivity estimation techniques in the *System R* paper depend on index counts when available. The work in this thesis has a similar cost model and cardinality estimator, but goes further to include cases when no statistics or indexes are available. In addition, equivalence sets are introduced, which help to identify primary key columns and distinct column counts when no statistics are available. This technique is not mentioned in the System R paper. This thesis describes the distributive effect of the cost model used and proves the importance of selectivity estimation in the absence of statistics.

The plan enumeration algorithm used in DuckDB is an implementation of the `DPhyp` algorithm described in the paper *Dynamic Programming Strikes Back* by *Moerkotte et al.* The paper introduces an efficient plan enumeration algorithm that enumerate possible join orders for a given query plan (13). The plan space enumeration algorithm is only one component of join order optimization, and therefore cannot be directly compared to work done in this thesis.

The paper *Simplicity Done Right for Join Ordering* by *Hertzschuch et al.* also presents a fast and effective way to determine an optimal query plan. Assuming access to the common statistic of most frequent values (top-k statistics) and precise selectivity estimates of base table filters, the authors produce a query optimizer that outperforms PostgreSQL and MonetDB with similar (or better) execution times for all 113 Join Order Benchmark queries. Using the most frequent value statistic, an upper bound for the cardinality of a join can be determined. The paper also introduces an enumeration scheme different from the one proposed by *Moerkotte et al.* in *Dynamic Programming Strikes Back*. First, all FK-PK joins, described as non-expanding joins in the paper, are applied. Then, $n : m$ joins with the smallest upper bounds are applied. Then the next next $n : m$ joins are applied in order of smallest upper bound. The results are quite favorable, achieving a faster query response time for 62% of the join order benchmark queries. The differences between work in this thesis and that of *Hertzschuch et al.* is in relation to the plan enumeration scheme, and base table statistics. (21) assumes knowledge of most common value statistics and precise selectivity estimation, while the tdom method requires either only an estimated distinct count, or no statistics at all. Of all papers discussed in this section, this paper comes closest to producing similar results and knowledge.

Every Row Counts: Combining Sketches and Sampling for Accurate Group-By Result Estimates by *Freitag et al.* describes techniques developed to estimate the number of distinct values in a `group by` clause. The paper introduces an efficient HLL sketch implementation that supports updates, deletes, and inserts. In addition, an efficient algorithm for finding high value-frequencies is introduced. The results show that the new counting HLL implementation exhibits almost identical estimation accuracy in comparison to the baseline in the presence of deletions and updates, with an increased overhead 5%. The baseline was HLL run on an identical data set with no deletions (23). Frequency estimation also had favorable results, improving in speed by a median of 9.8x compared to a baseline hash table implementation. This paper proposes unique ways to gather statistics on dynamic data but did not provide any results evaluating these statistics when used with a cardinality estimator or in the larger scope of a query optimizer.

How I Learned to Stop Worrying and Love Re-optimization by *Perron et al.* introduces an orthogonal concept to query optimization called re-optimization. The research goal of the paper was to determine if re-optimizing poorly performing queries would provide any benefit to end-to-end execution time. To detect when re-optimization should occur, the system compares the cardinality of an intermediate join to the predicted cardinality. If the difference is large enough, the query gets re-optimized. The study finds that only 20 poorly performing queries from the join order benchmark make up more than 95% of the execution time difference when compared to plans chosen using perfect join estimates. The results show that a simulated re-optimization obtains the same performance benefits provided by perfect cardinality estimation when tested on the Join Order Benchmark.

The re-optimization isn't implemented in practice, however. Intermediate results are materialized, then the `EXPLAIN ANALYZE` output is analyzed to determine if a re-optimization would occur. If re-optimization does occur, the execution time of the re-optimization is compared to the execution time without. Similar to this thesis, the paper does not test on large scale data sets. A production implementation may provide different results, and the performance benefits are not guaranteed if tested in production. Re-optimization also requires asking the question "when do you re-optimize?". The answer can be very dependent on the data set, and on how accurate the cardinality estimator is (3).

Looking Ahead Makes Query Plans Robust introduces another query optimization technique based on Lookahead information passing or LIP. Using pre-computed filter structures called bloom filters, selectivities of filters on dimension tables are estimated based on results of probing the bloom filters in batch. The resulting join order is then determined by the predicted selectivities of the filters from most selective to least selective. The authors introduce a cost model for building the bloom filters, and execution times were used to measure the performance improvements. Overall, there was a 5 - 10x speedup when using LIP. Tests were done on the *STAR SCHEMA* Benchmark, an unnamed synthetic benchmark, and TPC-H at scale factor 100. The paper only considers queries where a known "fact" table can be used as the outer relation for every join, and plan enumeration search spaces are restricted to left deep trees for this reason (6). This thesis includes the use of a plan enumeration algorithm that considers bushy trees with no knowledge of dimension or fact tables, and includes an evaluation on query optimizers where no data pre-processing is performed (i.e building of bloom filters).

Are We Ready For Learned Cardinality Estimation? analyzes whether or not cardinality estimation and query optimizers can use learning techniques in production. The authors analyze five learned models and determine that in a static environment (with no inserts, updates, or deletes), learned models are more accurate but suffer from high training costs. In more dynamic environments, learned models can not keep up with data updates which may change correlation and skew. In addition, learned models are much harder to maintain (24). While the cardinality estimation models can be used as input for the cost model when determining an optimal plan, the time required to train each model makes their implementation in DuckDB implausible, as DuckDB is meant to be used for ad-hoc analytical queries. The high overhead of start-up and learning times do not fit within the scope of DuckDB use-cases.

LEO DB2s LEarning Optimizer introduces a simple learned optimizer that uses results from previous query executions to inform future calls to the optimizer. Statistics from previous query executions are stored as "correcting" statistics and do not replace base table statistics. LEO stores cardinality statistics, and second-order statistics (i.e number of pages in a table). Experiments on an artificial data set and TPC-H showed that adjustments provided by LEO enabled the optimizer to choose query plans that performed up to 14 times better than execution plans without adjustments (2). The paper suffers from the same problem as *How I Learned to Stop Worrying and Love Re-optimization*; when do you re-optimize? There is a trade-off between re-optimizing and improved run times.

HyperLogLog is a near-optimal probabilistic algorithm for estimating the number of distinct elements of large data sets. With only a single pass over the data and m bytes, HyperLogLog can produce an estimate of the distinct count with an accuracy of about $1.04/\sqrt{m}$. The input stream of a dataset is divided into m substreams, and every value is hashed. Then bit-order observables (i.e leading 0's) are maintained for each substream. It then produces an estimate of the distinct count based on the normalized version of the harmonic mean of the indicator function. The indicator function is based on the bit order observables from each stream (25). While the algorithm presented in the paper does not provide a direct comparison to the work in this thesis, it is important to understand how the statistics used in this thesis are gathered.

The paper *Cardinality Estimation Done Right: Index-Based Join Sampling* by *Leis et al.* also presents a query optimizer with an improved cardinality estimation. In order to overcome poor cardinality estimates, the paper introduces *index-based join sampling*. For a join $A.x = B.y$, rather than joining two independent random samples from A and B to predict the cardinality of $(A \bowtie B)$, a random sample from $A.x$ probes the index $B.y$. The algorithm assumes that indexes allow a user to cheaply count the number of matches, making the join sampling efficient and accurate. The plan enumeration is similar to that of System R's and only considers left-deep trees, however, the algorithm does not determine the join order, calculating only the estimated cardinality of a join using index-based join sampling. The results show that with a sampling budget of 100,000 index lookups, only 3% of JOB queries are off by a factor of 10 or more when compared to a baseline. For larger data sets, the samples must be larger as well to maintain similar results. The overhead for index-based sampling in RAM takes about 30ms and is therefore cheap and fast enough for applications that require fast response times (10). This thesis deviates from this work by not requiring indexes, which can be expensive to build in DuckDB, and are not present in parquet or CSV files.

9

Conclusion

This thesis presents a minimal cardinality estimator requiring little to no base table statistics. The motivations behind the design of the cardinality estimator were simple and are reflected in the equation. The cardinality estimator works in conjunction with the cost model and the plan enumerator to keep the logic of the join order optimizer simple and maintainable.

In DuckDB, the dynamic programming table used to maintain the current best join order requires a cost model to exhibit optimal substructure. Developing and implementing a cardinality estimator with this requirement was challenging, but ultimately achievable. The cardinality estimator does not account for skew or correlations and therefore is not very accurate. Inaccurate cardinality estimates are reflected in the selection of some JOB query plans, yet the JOB results as a whole show that the cardinality estimator is an improvement from what it previously was. Remembering our goals from Chapter 1, it means **G1** was achieved.

The results show an overall improvement to query plans selected by the DuckDB query optimizer. When executing the Join Order Benchmark on base tables, maximum, average, and median intermediate cardinality counts all decreased by a factor of 98%, 95%, and 92% respectively. When run on parquet files, the maximum, average, and median intermediate cardinality counts decreased by 78%, 66%, and 87%. The improvements in decreased cardinality counts were reflected in end-to-end execution time as well, with experiments on base tables showing an improvement of 40%, 25% and 10% for maximum, average, and median and 68%, 18%, and 1% for the same statistics but with the Join Order Benchmark run on parquet files. The conclusion can therefore be made that the new DuckDB query optimizer is more robust and selects more efficient query plans, indicating that goal **G2** was achieved.

The cardinality estimator required more optimization overhead, but the improvement in end-to-end execution times justifies the extra overhead. In addition, the code was merged to DuckDB master. Goal **G3** was therefore also achieved (26).

While the work presented in this thesis presents results comparable to state-of-the-art systems, there is still plenty of room for improvement. Better filter estimates on base tables with and without statistics will improve the accuracy of the cardinality estimator. In addition, the cardinality estimator can be made more efficient so that it does not need to repeat the complete spanning tree algorithm for every new set of relations. The plan enumerator can also be improved to switch to a greedy algorithm or adaptive optimization if more than 11 joins are detected, as the current plan enumerator will take too long.

In the future, it would be interesting to see the results of implementing the mentioned improvements in the DuckDB optimizer. If such a simple cardinality estimation technique can provide us with the favorable results presented in this thesis, it is worthwhile to investigate how these refinements could further improve the quality of the join order optimizer.

References

[1] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. **How Good Are Query Optimizers, Really?** *Proc. VLDB Endow.*, **9**(3):204215, nov 2015. 1, 3, 6, 13, 16, 23, 24, 27

[2] Michael Stillger, Guy M. Lohman, Volker Markl, and Mokhtar Kandil. **LEO - DB2's LEarning Optimizer**. In *Proceedings of the 27th International Conference on Very Large Data Bases*, VLDB '01, page 1928, San Francisco, CA, USA, 2001. Morgan Kaufmann Publishers Inc. 1, 13, 50

[3] Matthew Perron, Zeyuan Shang, Tim Kraska, and Michael Stonebraker. **How I Learned to Stop Worrying and Love Re-optimization**, 2019. 1, 49

[4] Ramez Elmasri and Shamkant B. Navathe. *Fundamentals of Database Systems*. Pearson, 7th edition, 2015. 1, 2, 3, 7, 13

[5] P. Griffiths Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. **Access Path Selection in a Relational Database Management System**. In *Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data*, SIGMOD '79, page 2334, New York, NY, USA, 1979. Association for Computing Machinery. 2, 5, 13, 16

[6] Jianqiao Zhu, Navneet Potti, Saket Saurabh, and Jignesh M. Patel. **Looking Ahead Makes Query Plans Robust: Making the Initial Case with in-Memory Star Schema Data Warehouse Workloads**. *Proc. VLDB Endow.*, **10**(8):889900, apr 2017. 2, 49

[7] **PostgreSQL: Documentation: 14: 52.89. pg_stats**, 2022. 3

[8] **Catalog statistics - IBM Documentation**, 2022. 3

[9] **Cardinality Estimation (SQL Server) - SQL Server | Miscrosoft Docs**, 2022. 3

[10] Viktor Leis, Bernhard Radke, Andrey Gubichev, Alfons Kemper, and Thomas Neumann. **Cardinality Estimation Done Right: Index-Based Join Sampling**. In *8th Biennial Conference on Innovative Data Systems Research, CIDR 2017, Chaminade, CA, USA, January 8-11, 2017, Online Proceedings*. www.cidrdb.org, 2017. 3, 25, 50

[11] Mark Raasveldt and Hannes Mühleisen. **DuckDB: An Embeddable Analytical Database**. In *Proceedings of the 2019 International Conference on Management of Data*, SIGMOD '19, page 19811984, New York, NY, USA, 2019. Association for Computing Machinery. 3

[12] Guido Moerkotte and Thomas 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 '06, page 930941. VLDB Endowment, 2006. 5

[13] Guido Moerkotte and Thomas Neumann. **Dynamic Programming Strikes Back**. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, page 539552, New York, NY, USA, 2008. Association for Computing Machinery. 5, 47

[14] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009. 5, 19

[15] Yannis E. Ioannidis and Stavros Christodoulakis. **On the Propagation of Errors in the Size of Join Results**. In *Proceedings of the 1991 ACM SIGMOD International Conference on Management of Data*, SIGMOD '91, page 268277, New York, NY, USA, 1991. Association for Computing Machinery. 6, 13

[16] Yannis E. Ioannidis and Stavros Christodoulakis. **Optimal Histograms for Limiting Worst-Case Error Propagation in the Size of Join Results**. *ACM Trans. Database Syst.*, **18**(4):709748, dec 1993. 6

[17] Yannis E. Ioannidis and Viswanath Poosala. **Balancing Histogram Optimality and Practicality for Query Result Size Estimation**. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*, SIGMOD '95, page 233244, New York, NY, USA, 1995. Association for Computing Machinery. 6

[18] Gregory Piatetsky-Shapiro and Charles Connell. **Accurate Estimation of the Number of Tuples Satisfying a Condition**. *SIGMOD Rec.*, **14**(2):256276, jun 1984. 6

[19] R Elmasri, Shamkant B Navathe, R Elmasri, and SB Navathe. *Fundamentals of Database Systems*. Springer, 2000. 6

[20] Meikel Poess and Raghu Nambiar. **TPC Benchmark H Standard Specification**, 01 2010. 10, 24

[21] Axel Hertzschuch, Claudio Hartmann, Dirk Habich, and Wolfgang Lehner. **Simplicity Done Right for Join Ordering**. In *CIDR*, 2021. 13, 29, 48

[22] Thomas Neumann and Bernhard Radke. **Adaptive Optimization of Very Large Join Queries**. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, page 677692, New York, NY, USA, 2018. Association for Computing Machinery. 45

[23] Michael J. Freitag and Thomas Neumann. **Every Row Counts: Combining Sketches and Sampling for Accurate Group-By Result Estimates**. In *9th Biennial Conference on Innovative Data Systems Research, CIDR 2019, Asilomar, CA, USA, January 13-16, 2019, Online Proceedings*. www.cidrdb.org, 2019. 45, 48

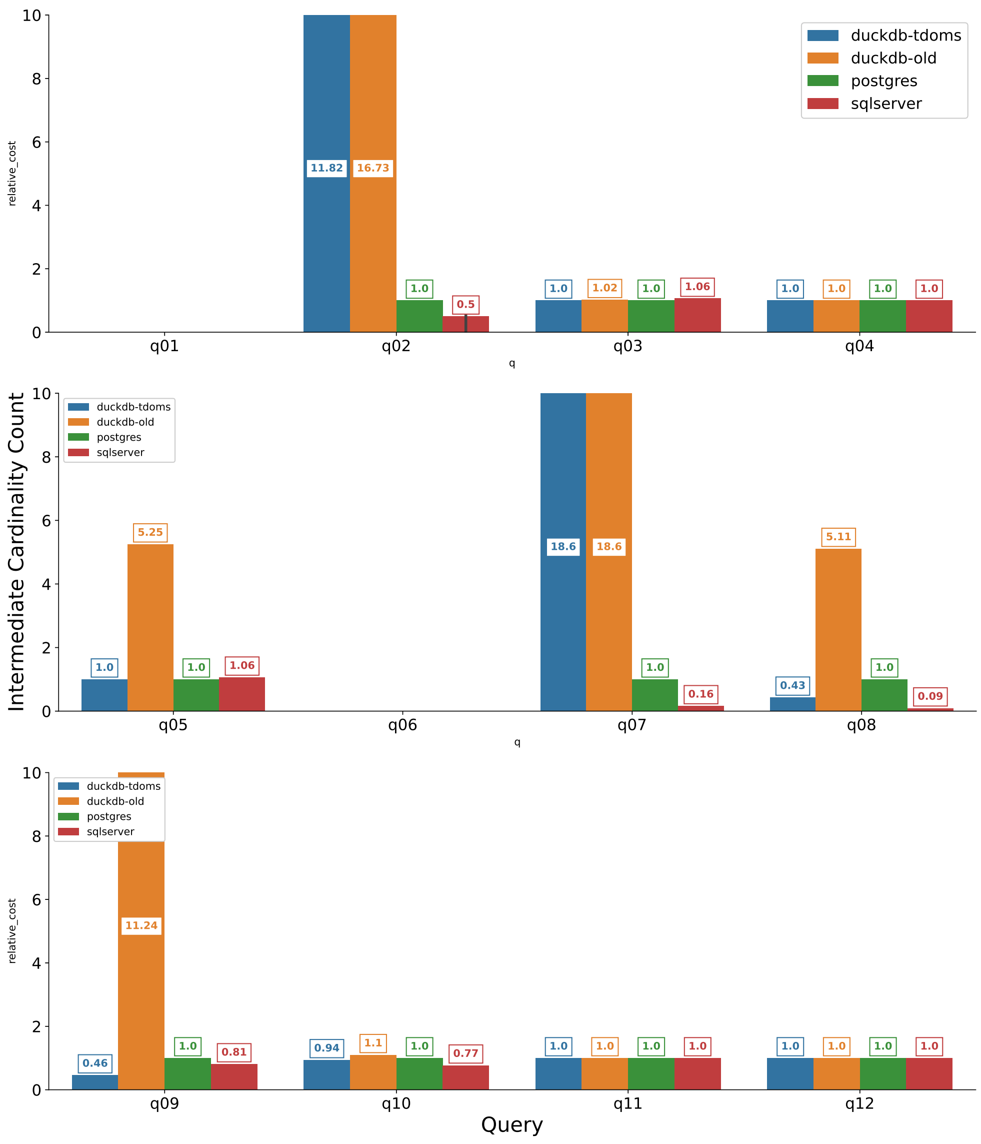
[24] Xiaoying Wang, Changbo Qu, Weiyuan Wu, Jiannan Wang, and Qingqing Zhou. **Are We Ready for Learned Cardinality Estimation?** **14**(9), 2021. 49

[25] Philippe Flajolet, Éric Fusy, Olivier Gandouet, and Frédéric Meunier. **HyperLogLog: the analysis of a near-optimal cardinality estimation algorithm**. In Philippe Jacquet, editor, *AofA: Analysis of Algorithms*, **DMTCS Proceedings vol. AH, 2007 Conference on Analysis of Algorithms (AofA 07)** of *DMTCS Proceedings*, pages 137–156, Juan les Pins, France, June 2007. Discrete Mathematics and Theoretical Computer Science. 50

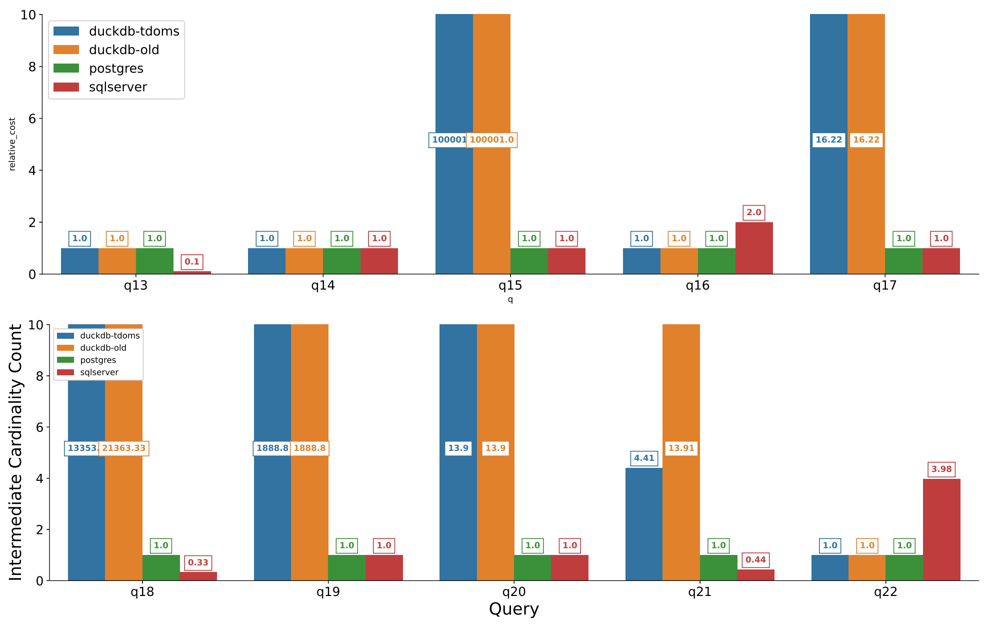
[26] Mark Raasveldt and Hannes Muehleisen. **DuckDB 0.5.0 Release**. 52

Appendix TPC-H Data

Intermediate cardinality counts for the TPC-H benchmark. If a query has no results, then there was no intermediate join present in the query tree.



Intermediate Cardinality Counts of TPC-H queries 1 to 12 run on base tables.



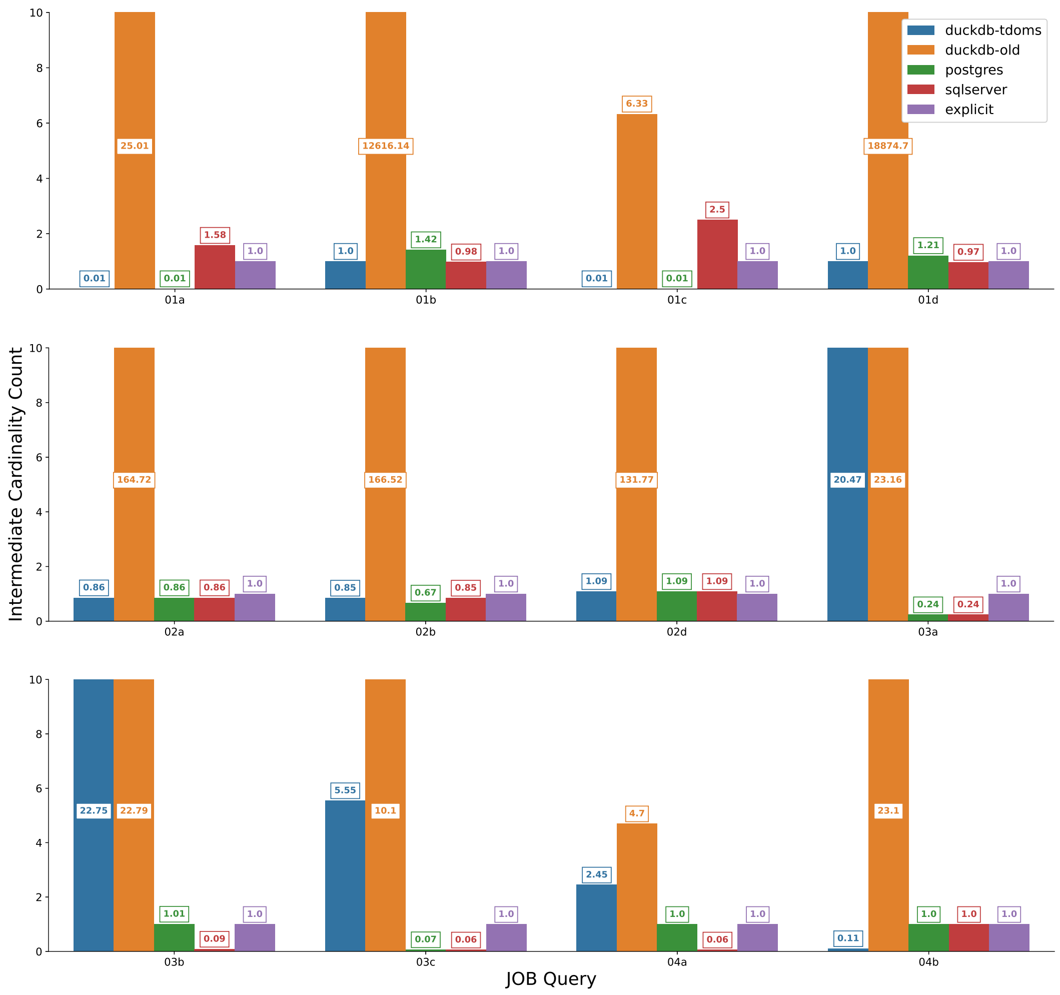
Intermediate Cardinality Counts of TPC-H queries 13 to 22 run on base tables.

Summary of end-to-end execution times of TPC-H queries on scale factor 10.

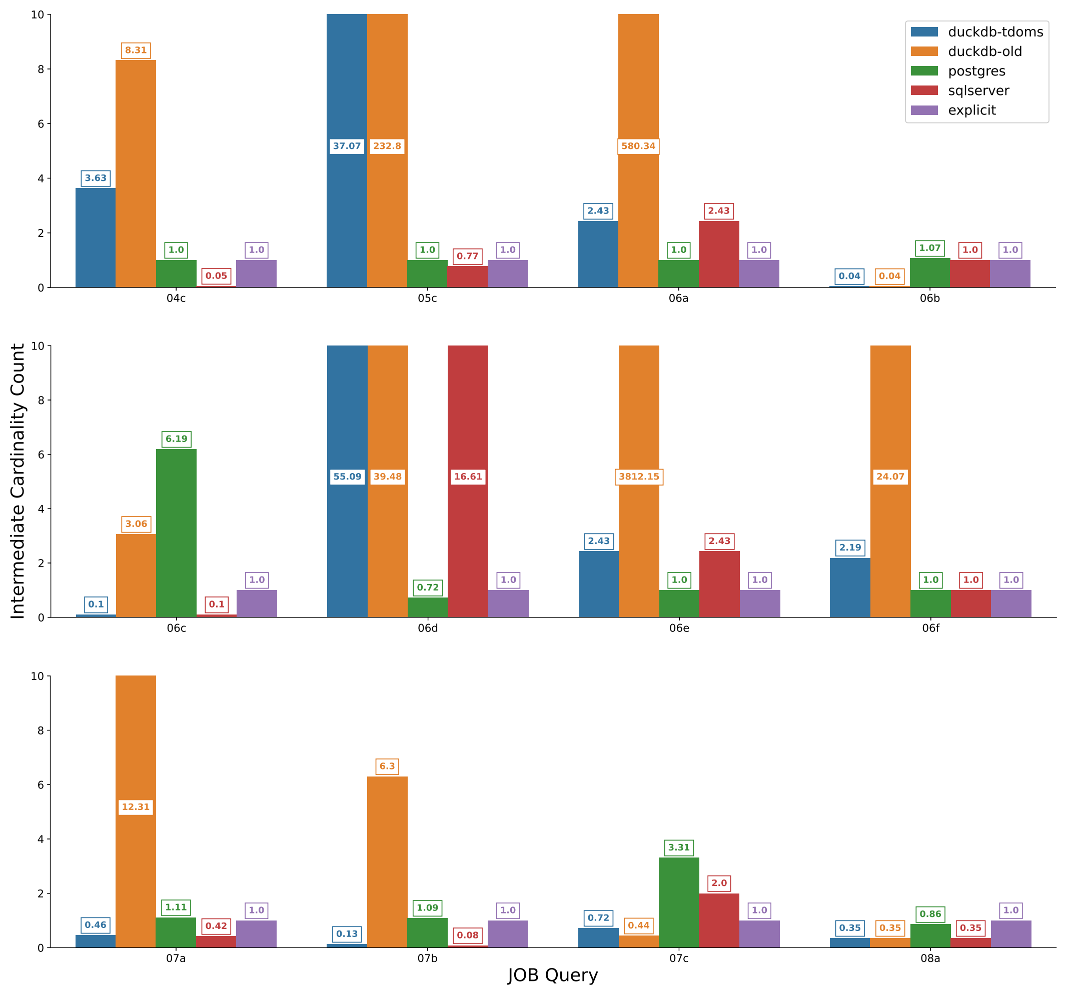
query	duckdb-tdoms total time	duckdb-old total time	duckdb-tdoms optimizer time	duckdb-old optimizer time
q01	0.375	0.379	0.000	0.000
q02	0.090	0.097	0.000	0.000
q03	0.196	0.244	0.000	0.000
q04	0.326	0.327	0.000	0.000
q05	0.196	0.203	0.001	0.000
q06	0.095	0.095	0.000	0.000
q07	0.541	0.536	0.001	0.000
q08	0.195	0.251	0.001	0.001
q09	0.913	4.440	0.001	0.000
q10	0.300	0.281	0.000	0.000
q11	0.049	0.050	0.000	0.000
q12	0.168	0.282	0.000	0.000
q13	0.457	0.443	0.000	0.000
q14	0.133	0.139	0.000	0.000
q15	0.421	0.426	0.000	0.000
q16	0.305	0.322	0.000	0.000
q17	0.818	0.833	0.000	0.000
q18	1.163	1.303	0.000	0.000
q19	0.288	0.293	0.000	0.000
q20	0.570	0.561	0.000	0.000
q21	1.020	1.209	0.000	0.000
q22	0.146	0.160	0.000	0.000

Appendix JOB data

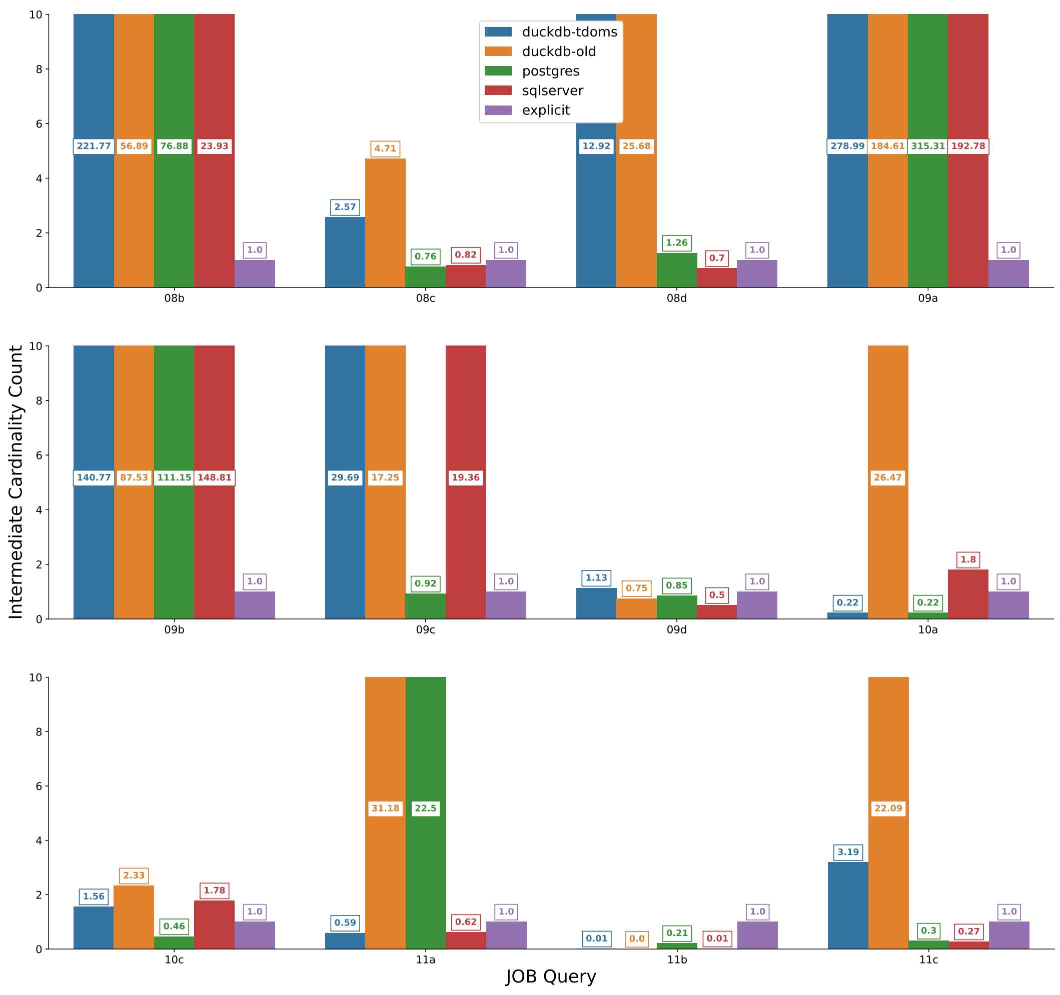
Intermediate cardinality counts for the Join Order Benchmark on base tables.



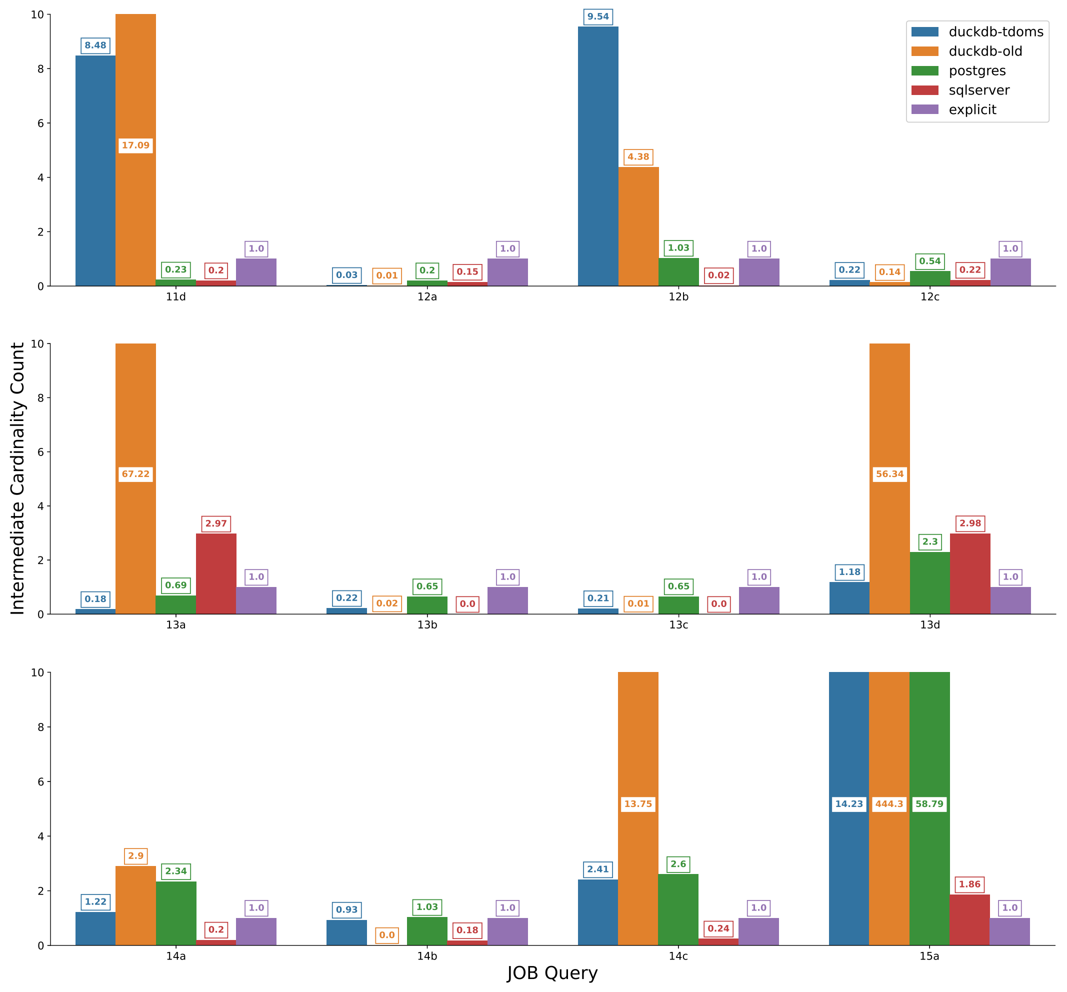
Intermediate Cardinality Counts of JOB queries 01a to 04b run on base tables.



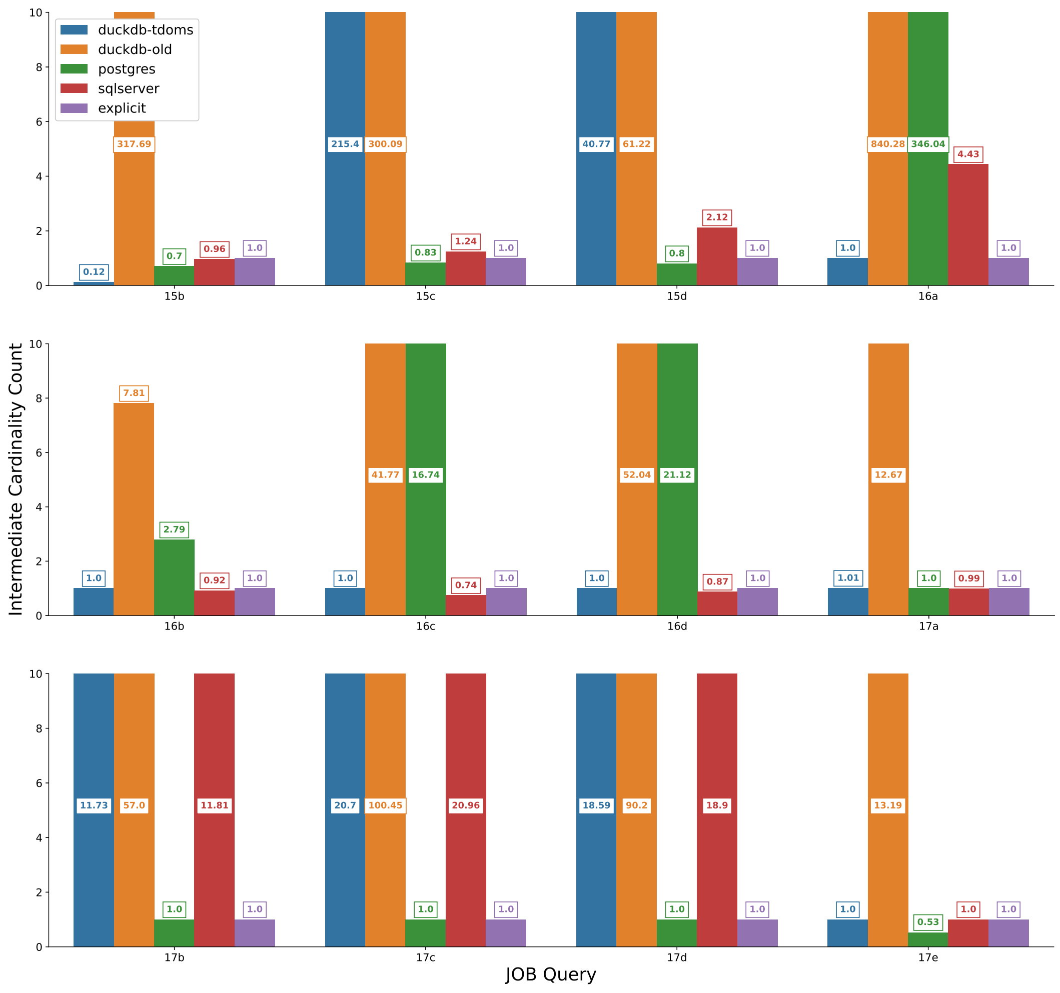
Intermediate Cardinality Counts of JOB queries 04c to 08a run on base tables.



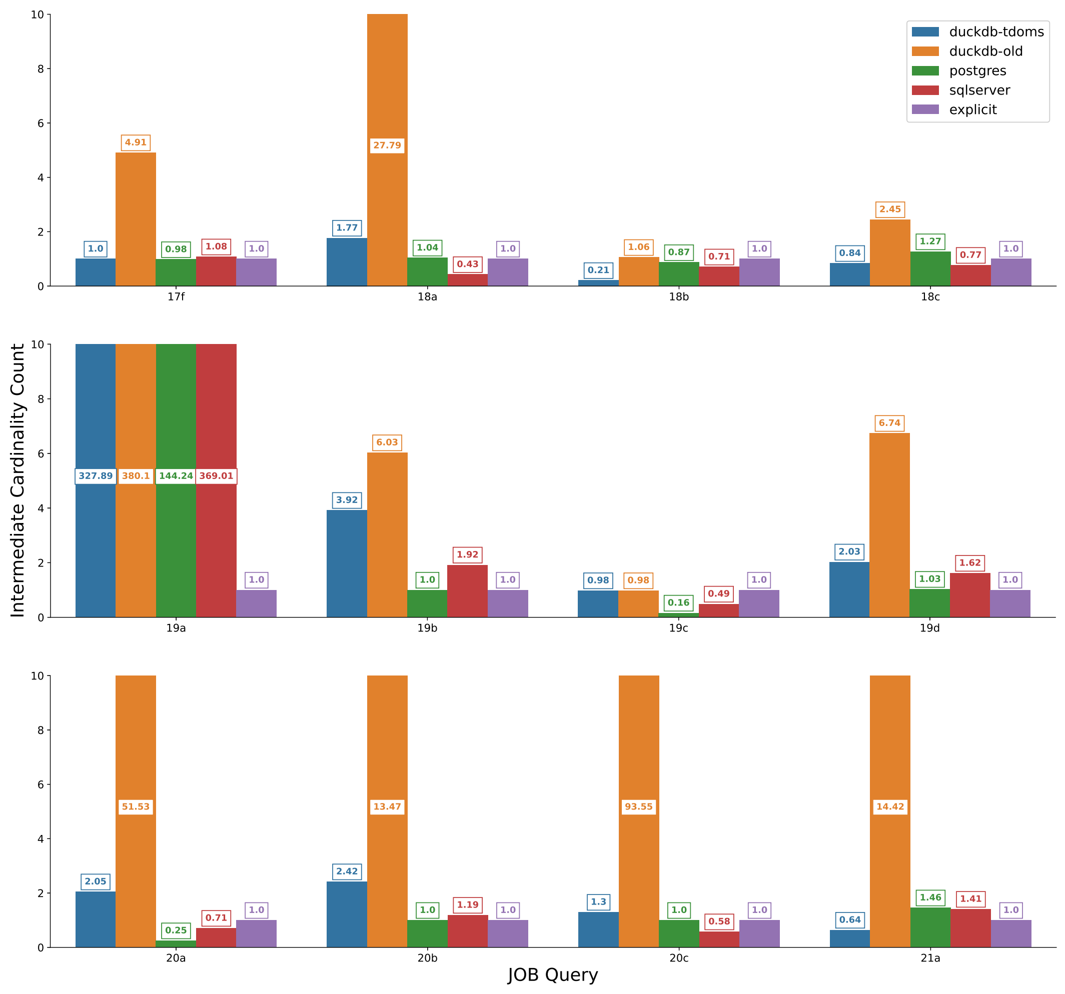
Intermediate Cardinality Counts of JOB queries 08b to 11c run on base tables.



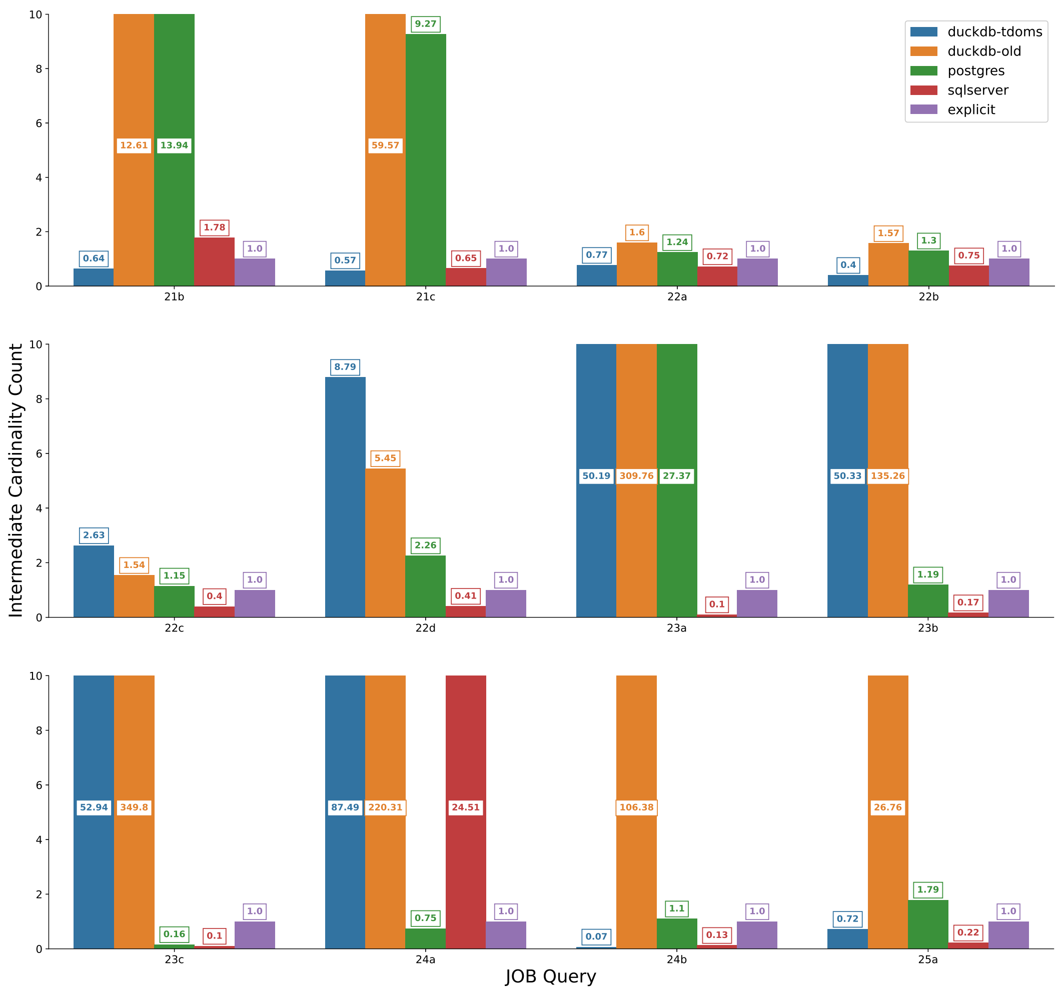
Intermediate Cardinality Counts of JOB queries 11d to 15a run on base tables.



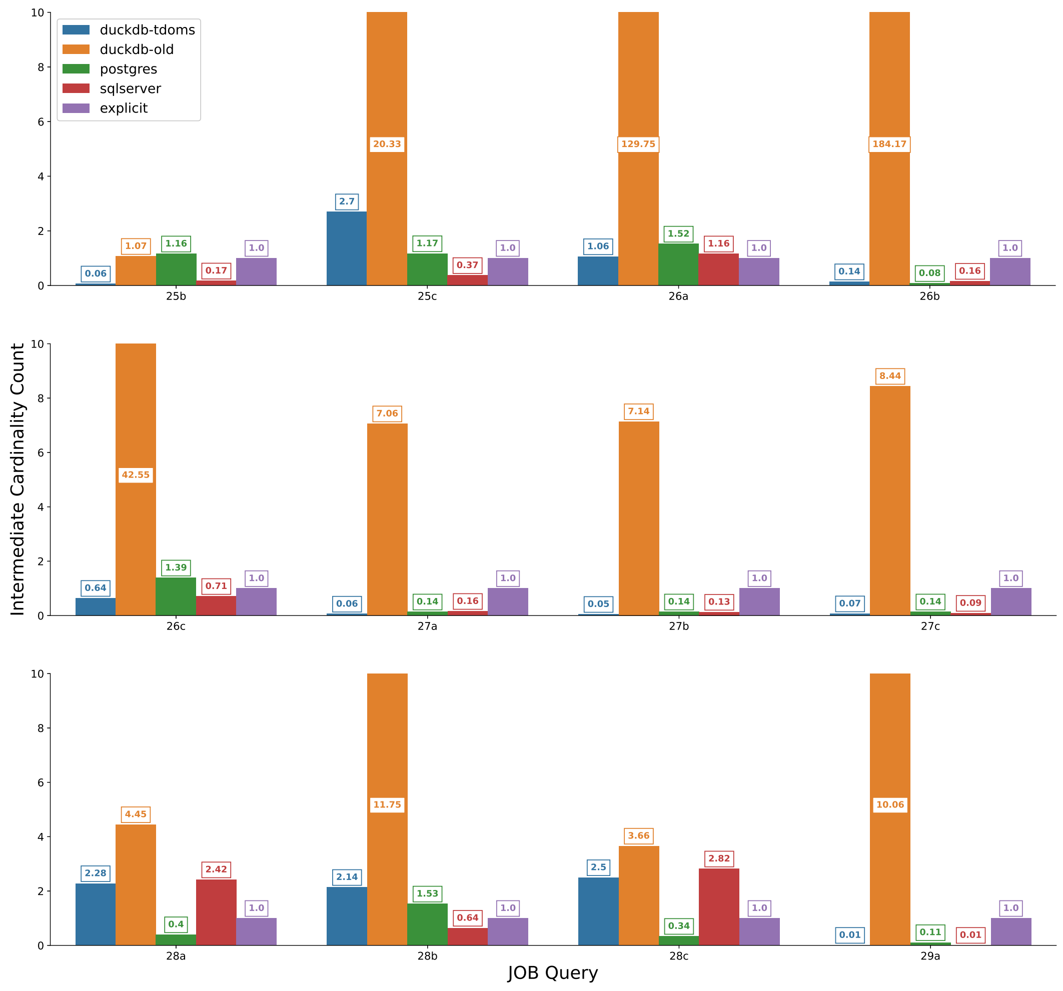
Intermediate Cardinality Counts of JOB queries 15b to 17e run on base tables.



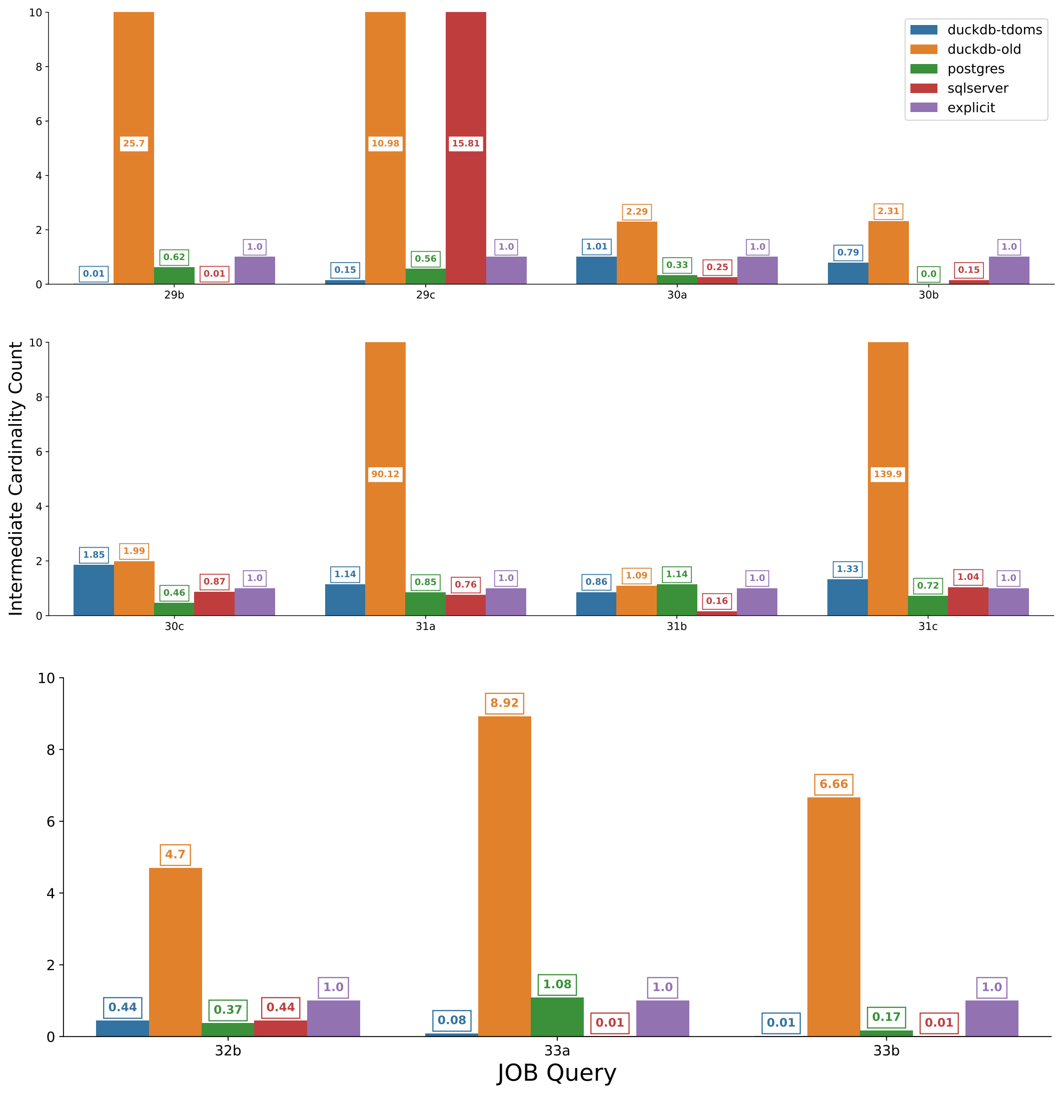
Intermediate Cardinality Counts of JOB queries 17f to 21a run on base tables.



Intermediate Cardinality Counts of JOB queries 21b to 25a run on base tables.



Intermediate Cardinality Counts of JOB queries 25b to 29a run on base tables.



Intermediate Cardinality Counts of JOB queries 29b to 33b run on base tables.

Summary of end-to-end execution times for queries 01a to 09d of the Join Order Benchmark when run on base tables. Results are in seconds.

query	relations	duckdb-tdoms-parquet total time	duckdb-old-parquet total time	duckdb-tdoms-parquet optimizer time	duckdb-old-parquet optimizer time
01a	5	0.024	0.077	0.000	0.000
01b	5	0.016	0.065	0.000	0.000
01c	5	0.017	0.033	0.000	0.000
01d	5	0.016	0.080	0.000	0.000
02a	5	0.029	0.194	0.000	0.000
02b	5	0.028	0.191	0.000	0.000
02c	5	0.013	0.194	0.000	0.000
02d	5	0.050	0.193	0.000	0.000
03a	4	0.086	0.107	0.000	0.000
03b	4	0.042	0.041	0.000	0.000
03c	4	0.100	0.179	0.000	0.000
04a	5	0.062	0.054	0.000	0.000
04b	5	0.018	0.027	0.000	0.000
04c	5	0.109	0.087	0.000	0.000
05a	5	0.029	0.074	0.000	0.000
05b	5	0.024	0.047	0.000	0.000
05c	5	0.105	0.094	0.000	0.000
06a	5	0.054	0.056	0.000	0.000
06b	5	0.052	0.043	0.000	0.000
06c	5	0.050	0.042	0.000	0.000
06d	5	0.264	0.073	0.000	0.000
06e	5	0.055	0.073	0.000	0.000
06f	5	0.299	0.432	0.000	0.000
07a	8	0.069	0.109	0.014	0.001
07b	8	0.065	0.069	0.014	0.001
07c	8	0.413	0.292	0.014	0.001
08a	7	0.087	0.103	0.003	0.001
08b	7	0.086	0.085	0.003	0.001
08c	7	1.374	0.722	0.003	0.001
08d	7	1.089	0.577	0.003	0.001
09a	8	0.167	0.167	0.013	0.001
09b	8	0.126	0.128	0.013	0.001
09c	8	0.172	0.175	0.013	0.001
09d	8	0.429	0.203	0.013	0.001

Summary of end-to-end execution times for queries 10a to 17f of the Join Order Benchmark when run on base tables. Results are in seconds.

query	relations	duckdb-tdoms-parquet total time	duckdb-old-parquet total time	duckdb-tdoms-parquet optimizer time	duckdb-old-parquet optimizer time
10a	7	0.107	0.159	0.002	0.000
10b	7	0.103	0.156	0.002	0.000
10c	7	0.180	0.201	0.002	0.000
11a	8	0.032	0.048	0.007	0.001
11b	8	0.028	0.029	0.007	0.001
11c	8	0.097	0.144	0.007	0.001
11d	8	0.319	0.141	0.007	0.001
12a	8	0.061	0.065	0.007	0.001
12b	8	0.068	0.076	0.007	0.001
12c	8	0.112	0.080	0.007	0.001
13a	9	0.088	2.449	0.018	0.002
13b	9	0.074	0.060	0.018	0.002
13c	9	0.065	0.053	0.018	0.002
13d	9	0.147	2.166	0.018	0.002
14a	8	0.088	0.129	0.005	0.001
14b	8	0.080	0.070	0.005	0.001
14c	8	0.109	0.245	0.005	0.001
15a	9	0.089	0.105	0.018	0.003
15b	9	0.079	0.088	0.018	0.003
15c	9	0.138	0.124	0.018	0.003
15d	9	0.146	0.111	0.018	0.003
16a	8	0.068	0.185	0.011	0.001
16b	8	0.579	0.521	0.011	0.001
16c	8	0.180	0.240	0.011	0.001
16d	8	0.177	0.229	0.011	0.001
17a	7	0.195	0.249	0.004	0.001
17b	7	0.276	0.225	0.004	0.001
17c	7	0.279	0.214	0.004	0.001
17d	7	0.284	0.223	0.004	0.001
17e	7	0.247	0.502	0.004	0.001
17f	7	0.310	0.313	0.004	0.001

Summary of end-to-end execution times for queries 18a to 26c of the Join Order Benchmark when run on base tables. Results are in seconds.

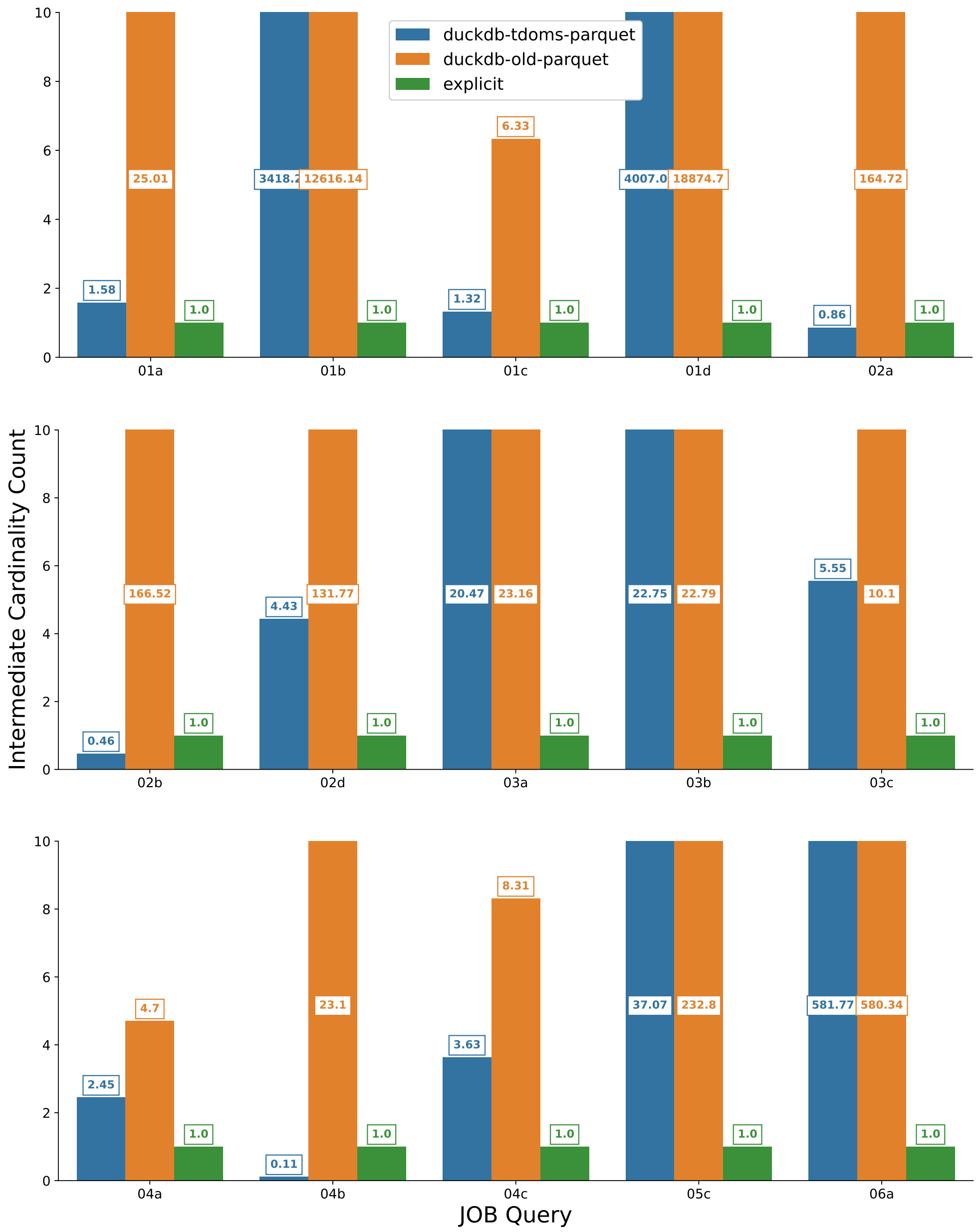
query	relations	duckdb-tdoms-parquet total time	duckdb-old-parquet total time	duckdb-tdoms-parquet optimizer time	duckdb-old-parquet optimizer time
18a	7	0.222	0.642	0.004	0.001
18b	7	0.142	0.144	0.004	0.001
18c	7	0.196	0.232	0.004	0.001
19a	10	0.222	0.205	0.022	0.003
19b	10	0.173	0.150	0.022	0.003
19c	10	0.256	0.236	0.022	0.003
19d	10	0.399	0.691	0.022	0.003
20a	10	0.164	0.468	0.020	0.003
20b	10	0.132	0.092	0.020	0.003
20c	10	0.128	0.348	0.020	0.003
21a	9	0.096	0.093	0.018	0.003
21b	9	0.068	0.071	0.018	0.003
21c	9	0.098	0.143	0.018	0.003
22a	11	0.122	0.087	0.022	0.003
22b	11	0.101	0.084	0.022	0.003
22c	11	0.240	0.123	0.022	0.003
22d	11	0.383	0.135	0.022	0.003
23a	11	0.121	0.072	0.022	0.003
23b	11	0.121	0.053	0.022	0.003
23c	11	0.126	0.106	0.022	0.003
24a	12	0.268	0.241	0.028	0.003
24b	12	0.197	0.214	0.028	0.003
25a	9	0.194	0.518	0.018	0.003
25b	9	0.152	0.154	0.018	0.003
25c	9	0.304	0.669	0.018	0.003
26a	12	0.162	0.237	0.026	0.003
26b	12	0.115	0.212	0.025	0.003
26c	12	0.209	0.220	0.026	0.003

Summary of end-to-end execution times for queries 27a to 33c of the Join Order Benchmark when run on base tables. Results are in seconds.

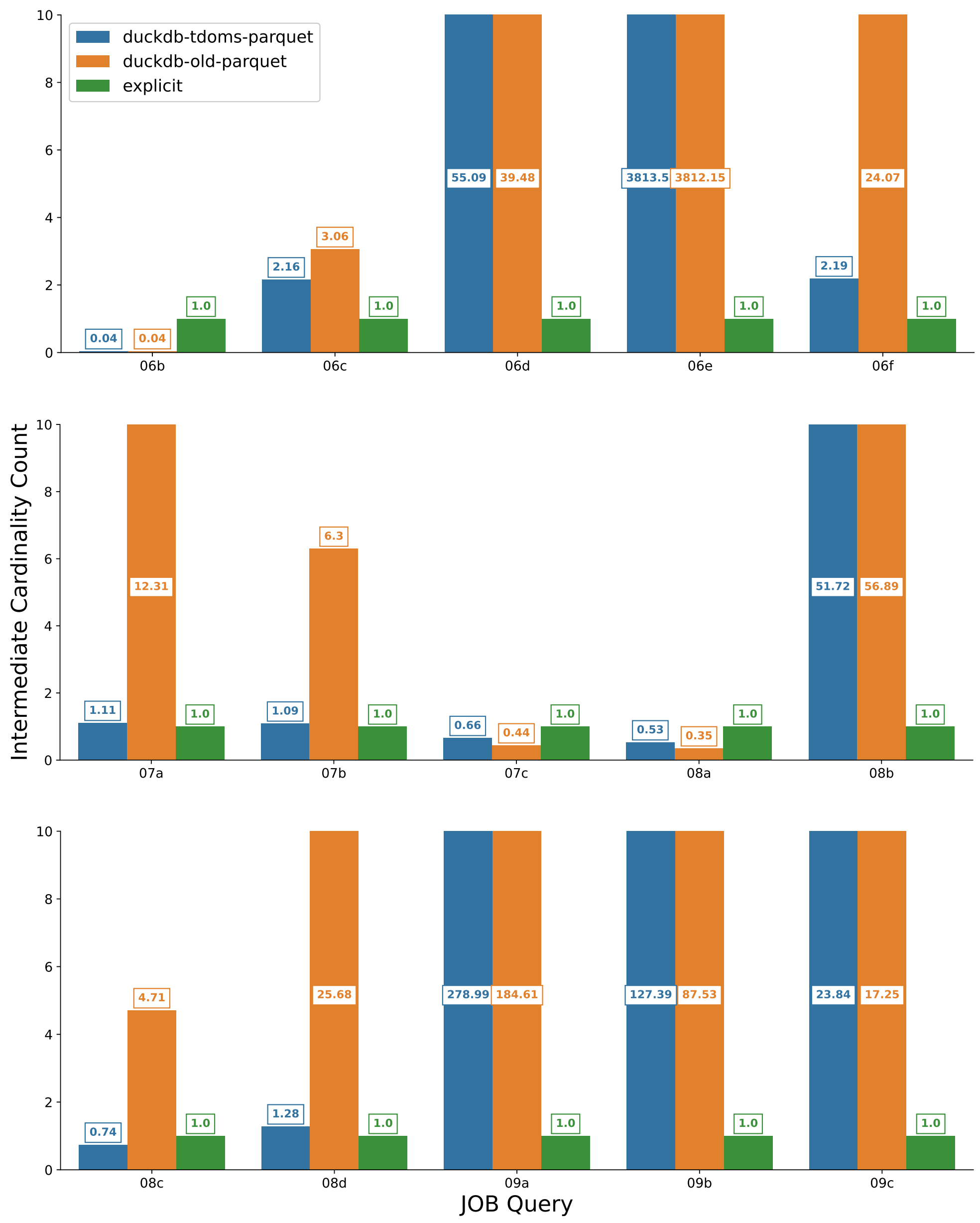
query	relations	duckdb-tdoms-parquet total time	duckdb-old-parquet total time	duckdb-tdoms-parquet optimizer time	duckdb-old-parquet optimizer time
27a	12	0.085	0.053	0.027	0.003
27b	12	0.084	0.051	0.027	0.003
27c	12	0.109	0.071	0.027	0.003
28a	14	0.200	0.093	0.032	0.004
28b	14	0.127	0.071	0.032	0.004
28c	14	0.226	0.092	0.032	0.004
29a	17	0.272	0.191	0.106	0.004
29b	17	0.257	0.155	0.107	0.004
29c	17	0.317	0.235	0.124	0.004
30a	12	0.174	0.154	0.026	0.003
30b	12	0.172	0.153	0.027	0.003
30c	12	0.257	0.187	0.027	0.003
31a	11	0.196	1.018	0.023	0.003
31b	11	0.188	0.171	0.023	0.003
31c	11	0.246	1.359	0.023	0.003
32a	6	0.005	0.068	0.001	0.000
32b	6	0.026	0.068	0.001	0.000
33a	14	0.063	0.031	0.029	0.003
33b	14	0.062	0.027	0.030	0.003
33c	14	0.066	0.032	0.029	0.003

Appendix JOB on parquet data

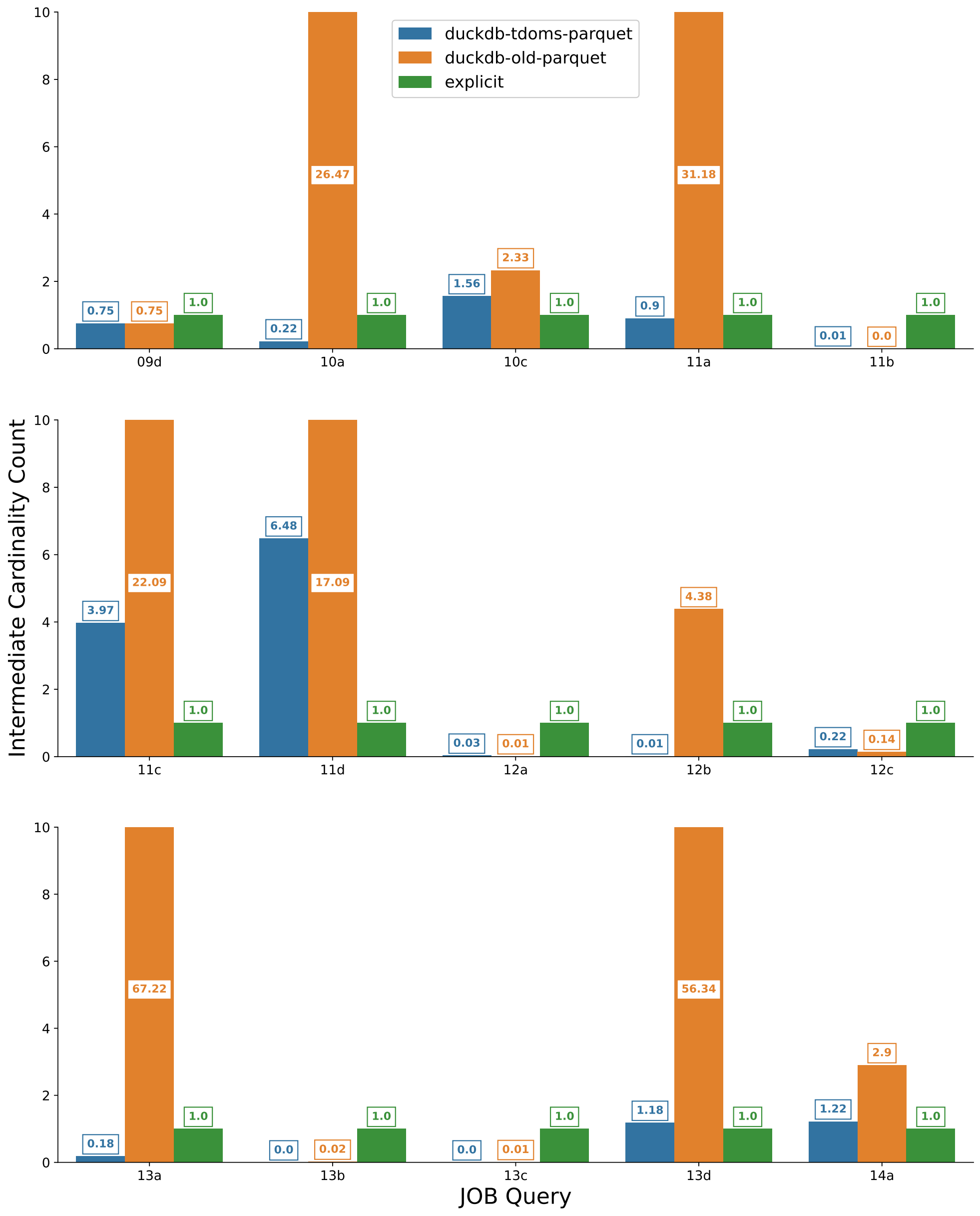
Intermediate cardinality counts for join order benchmark on parquet files.



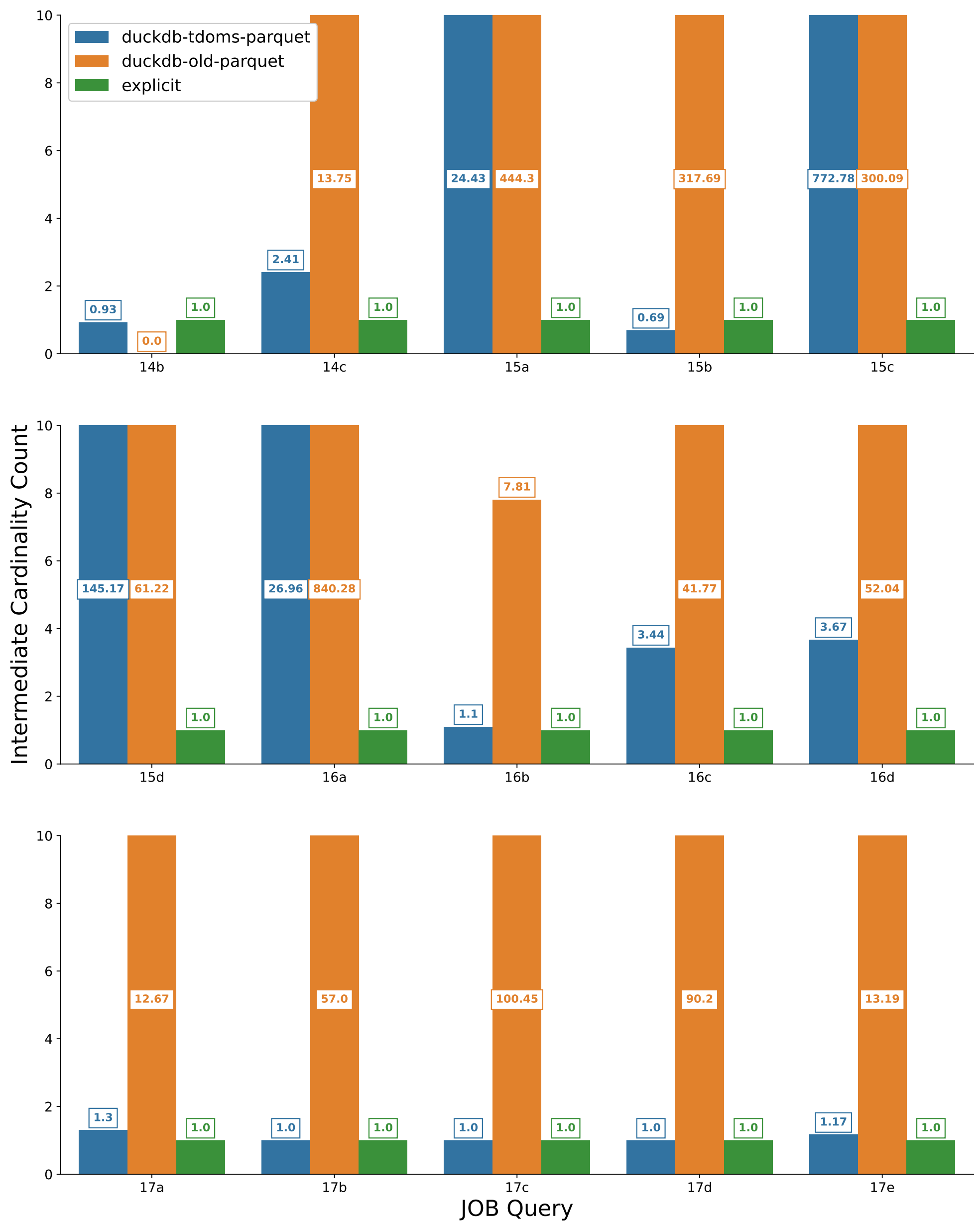
Intermediate Cardinality Counts of JOB queries 01a to 06a run on parquet files.



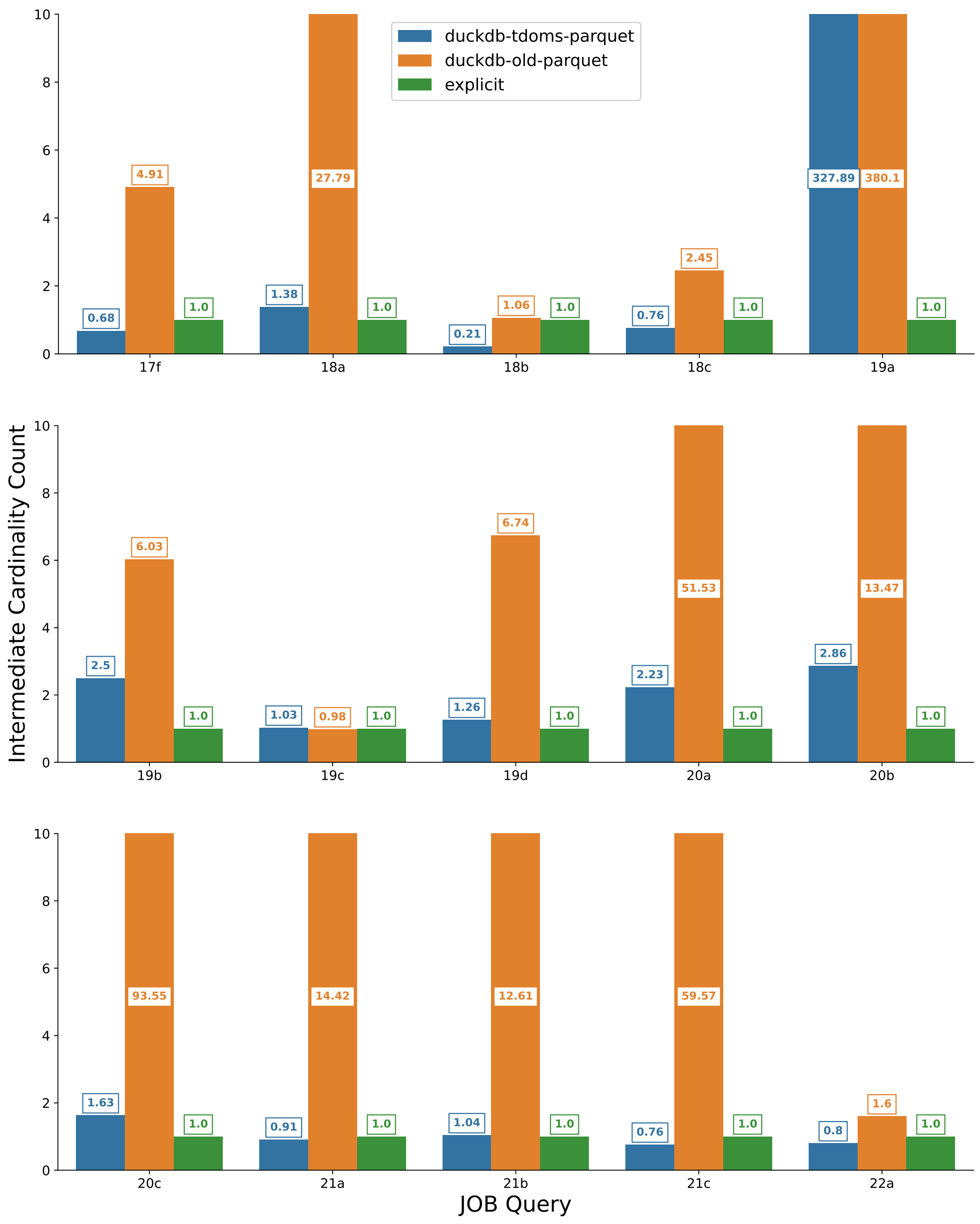
Intermediate Cardinality Counts of JOB queries 06b to 09c run on parquet files.



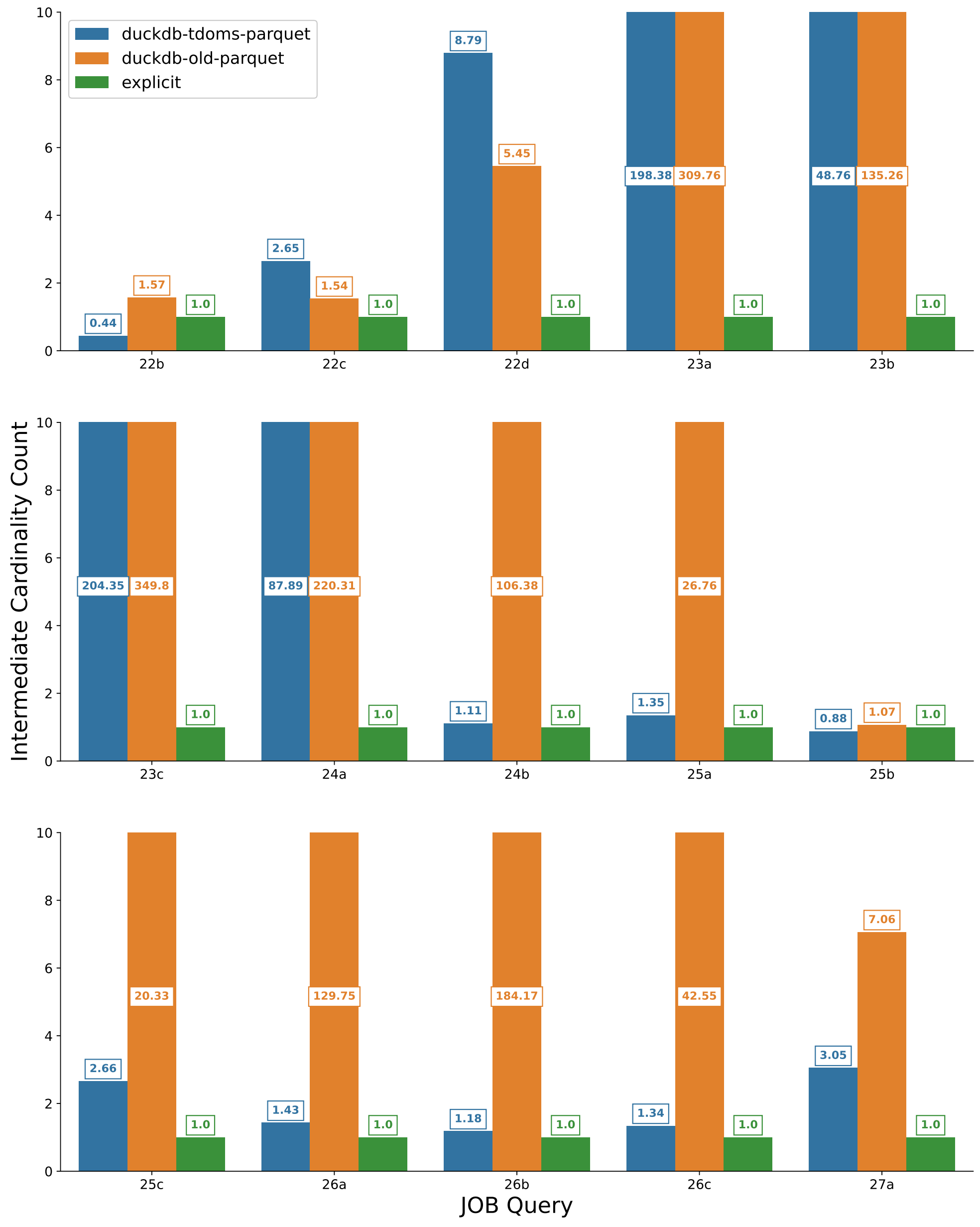
Intermediate Cardinality Counts of JOB queries 09d to 14a run on parquet files.



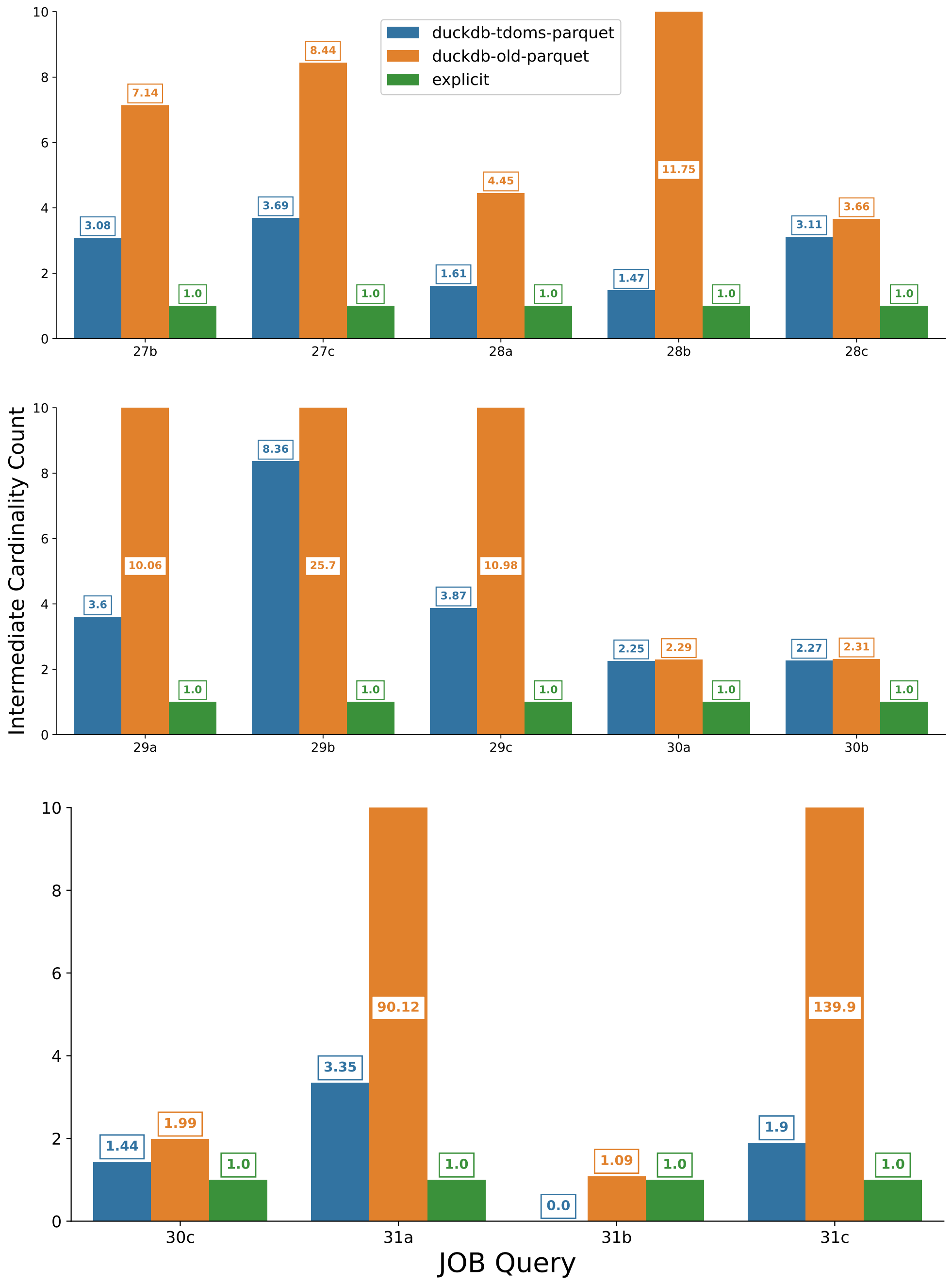
Intermediate Cardinality Counts of JOB queries 14b to 17e run on parquet files.



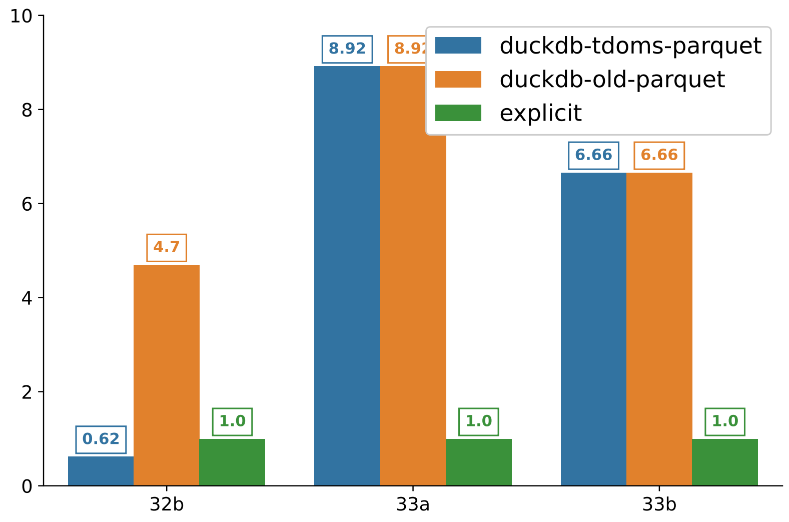
Intermediate Cardinality Counts of JOB queries 17f to 22a run on parquet files.



Intermediate Cardinality Counts of JOB queries 22b to 27a run on parquet files.



Intermediate Cardinality Counts of JOB queries 27b to 31c run on parquet files.



Intermediate
Cardinality Counts of JOB queries 32b to 33b run on parquet files.

Summary of end-to-end execution times for queries 01a to 09d of the Join Order Benchmark when run on parquet files. Results are in seconds.

query	relations	duckdb-tdoms-parquet total time	duckdb-old-parquet total time	duckdb-tdoms-parquet optimizer time	duckdb-old-parquet optimizer time
01a	5	0.073	0.115	0.000	0.000
01b	5	0.077	0.104	0.000	0.000
01c	5	0.063	0.072	0.000	0.000
01d	5	0.080	0.117	0.000	0.000
02a	5	0.068	0.217	0.000	0.000
02b	5	0.063	0.220	0.000	0.000
02c	5	0.025	0.218	0.000	0.000
02d	5	0.094	0.224	0.000	0.000
03a	4	0.328	0.342	0.000	0.000
03b	4	0.288	0.278	0.000	0.000
03c	4	0.341	0.382	0.000	0.000
04a	5	0.092	0.089	0.000	0.000
04b	5	0.061	0.058	0.000	0.000
04c	5	0.103	0.120	0.000	0.000
05a	5	0.068	0.321	0.000	0.000
05b	5	0.055	0.291	0.000	0.000
05c	5	0.351	0.335	0.000	0.000
06a	5	0.158	0.151	0.001	0.000
06b	5	0.168	0.165	0.001	0.000
06c	5	0.156	0.158	0.001	0.000
06d	5	0.355	0.168	0.001	0.000
06e	5	0.168	0.167	0.001	0.000
06f	5	0.405	0.511	0.001	0.000
07a	8	0.248	0.220	0.014	0.001
07b	8	0.213	0.165	0.014	0.001
07c	8	0.296	0.419	0.014	0.001
08a	7	0.292	0.267	0.003	0.001
08b	7	0.287	0.262	0.003	0.001
08c	7	0.518	0.803	0.003	0.001
08d	7	0.278	0.667	0.003	0.001
09a	8	0.423	0.382	0.013	0.002
09b	8	0.355	0.326	0.014	0.002
09c	8	0.429	0.377	0.013	0.002
09d	8	0.520	0.400	0.013	0.001

Summary of end-to-end execution times for queries 10a to 17f of the Join Order Benchmark when run on parquet files. Results are in seconds.

query	relations	duckdb-tdoms-parquet total time	duckdb-old-parquet total time	duckdb-tdoms-parquet optimizer time	duckdb-old-parquet optimizer time
10a	7	0.331	0.335	0.002	0.001
10b	7	0.272	0.332	0.002	0.001
10c	7	0.396	0.391	0.002	0.001
11a	8	0.095	0.096	0.007	0.001
11b	8	0.082	0.071	0.007	0.001
11c	8	0.101	0.178	0.007	0.001
11d	8	0.145	0.181	0.007	0.001
12a	8	0.334	0.315	0.007	0.001
12b	8	0.317	0.339	0.007	0.001
12c	8	0.379	0.327	0.007	0.001
13a	9	0.366	2.538	0.018	0.002
13b	9	0.133	0.115	0.018	0.002
13c	9	0.133	0.102	0.018	0.002
13d	9	0.222	2.233	0.018	0.002
14a	8	0.352	0.355	0.005	0.001
14b	8	0.339	0.323	0.005	0.001
14c	8	0.374	0.458	0.005	0.001
15a	9	0.360	0.353	0.018	0.003
15b	9	0.353	0.342	0.018	0.003
15c	9	0.343	0.371	0.018	0.003
15d	9	0.174	0.180	0.018	0.003
16a	8	0.209	0.273	0.011	0.002
16b	8	0.535	0.581	0.011	0.001
16c	8	0.309	0.324	0.011	0.001
16d	8	0.308	0.305	0.011	0.002
17a	7	0.330	0.348	0.004	0.001
17b	7	0.254	0.344	0.004	0.001
17c	7	0.251	0.330	0.004	0.001
17d	7	0.245	0.335	0.004	0.001
17e	7	0.386	0.578	0.004	0.001
17f	7	0.309	0.421	0.004	0.001

Summary of end-to-end execution times for queries 27a to 33c of the Join Order Benchmark when run on parquet files. Results are in seconds.

query	relations	duckdb-tdoms-parquet total time	duckdb-old-parquet total time	duckdb-tdoms-parquet optimizer time	duckdb-old-parquet optimizer time
18a	7	0.530	0.970	0.005	0.001
18b	7	0.556	0.553	0.005	0.001
18c	7	0.549	0.572	0.005	0.001
19a	10	0.664	0.624	0.023	0.003
19b	10	0.581	0.580	0.023	0.003
19c	10	0.661	0.629	0.023	0.003
19d	10	0.602	0.880	0.022	0.003
20a	10	0.463	0.572	0.018	0.003
20b	10	0.380	0.246	0.018	0.003
20c	10	0.379	0.492	0.018	0.003
21a	9	0.372	0.363	0.018	0.003
21b	9	0.348	0.339	0.018	0.003
21c	9	0.381	0.404	0.018	0.003
22a	11	0.409	0.378	0.022	0.003
22b	11	0.391	0.372	0.022	0.003
22c	11	0.520	0.403	0.022	0.003
22d	11	0.664	0.408	0.022	0.003
23a	11	0.359	0.361	0.022	0.003
23b	11	0.338	0.323	0.022	0.003
23c	11	0.363	0.392	0.022	0.003
24a	12	0.697	0.679	0.029	0.004
24b	12	0.683	0.649	0.029	0.004
25a	9	0.577	0.810	0.018	0.003
25b	9	0.540	0.526	0.018	0.003
25c	9	0.655	0.959	0.018	0.003
26a	12	0.342	0.387	0.026	0.003
26b	12	0.267	0.353	0.026	0.003
26c	12	0.278	0.354	0.026	0.003

End-to-End Execution Times for queries 27a to 26c of the Join Order Benchmark on parquet files. Results are in seconds.

query	relations	duckdb-tdoms-parquet total time	duckdb-old-parquet total time	duckdb-tdoms-parquet optimizer time	duckdb-old-parquet optimizer time
27a	12	0.356	0.315	0.027	0.003
27b	12	0.358	0.315	0.027	0.003
27c	12	0.382	0.346	0.027	0.003
28a	14	0.444	0.381	0.033	0.004
28b	14	0.410	0.352	0.033	0.004
28c	14	0.416	0.376	0.033	0.004
29a	17	0.790	0.668	0.125	0.004
29b	17	0.756	0.620	0.126	0.004
29c	17	0.817	0.679	0.126	0.004
30a	12	0.578	0.541	0.027	0.003
30b	12	0.572	0.538	0.027	0.003
30c	12	0.599	0.572	0.027	0.003
31a	11	0.593	1.268	0.023	0.003
31b	11	0.635	0.569	0.023	0.003
31c	11	0.655	1.577	0.023	0.003
32a	6	0.064	0.121	0.001	0.000
32b	6	0.096	0.119	0.001	0.000
33a	14	0.161	0.110	0.028	0.004
33b	14	0.146	0.104	0.028	0.004
33c	14	0.166	0.112	0.028	0.004