Vrije Universiteit Amsterdam Universiteit van Amsterdam

Master's Thesis

Cost-Based Hybrid Query Optimization in MotherDuck

Author: Jeewon Heo (2787818)

1st supervisor: Peter Boncz
daily supervisor: Boaz Leskes (MotherDuck)
2nd reader: Stefan Manegold

A thesis submitted in fulfillment of the requirements for the joint UvA-VU Master of Science degree in Computer Science

November 4, 2024

Abstract

MotherDuck is a cloud-based data warehouse built based on DuckDB, a database system optimized for analytical workloads. It offers hybrid query execution, where parts of a query are executed on a client and others on a server. MotherDuck's current query optimizer's site selection strategy relies on a heuristic, often resulting in suboptimal query plans. To address this, we develop a cost-based hybrid query optimizer that uses a dynamic programming approach to find a globally optimal plan. We define cost as the amount of data transferred over the network, which we approximate with cardinality estimates. Inspired by System-R's interesting order concept, we retain plans per interesting site, preserving plans that are not immediately the cheapest but whose result site may benefit subsequent operations. To support the cost-based optimizer, we have introduced changes in MotherDuck and DuckDB to 1) correctly utilize statistics for cardinality estimation and 2) preserve the cardinality estimation. The experimental results with the latest DuckDB show a 1.22x average speedup for TPC-H and 2.06x for TPC-DS. The long-running queries benefit more from the cost-based optimization, with some reaching over 17x speedup. The short-running queries are more prone to slowdowns and generally experience less degree of speedup. The preliminary evaluation with a cost model with constant latency cost does not introduce significant changes in performance overall, although some JOB queries gain over 38x speedup.

Contents

List of Figures

List of Tables

1 Introduction

In recent years, developments in the data world have been driven by the need to accommodate big data. Businesses increasingly rely on large-scale data to generate insights, train machine learning models, etc. [7]. This led to the rise of cloud-based data warehouses like Snowflake[1] and BigQuery[2] [8].

However, only a handful of companies generate and use big data. As elaborated in the article “Big Data is Dead”, in practice, big data is not necessary for many scenarios [9]. A medium-sized business with a few thousand customers putting in daily orders would generate megabytes of data. This sums to only a few gigabytes of data a year. Moreover, many applications, such as dashboards, only require a recent piece of total data. That is, while the amount of logged data may increase linearly over time, the required compute power remains steady.

Under these observations, MotherDuck[3] set out to provide *hybrid query execution* built based on DuckDB[4], an analytical database system. Hybrid query execution (or hybrid execution) is a new way of processing queries to database systems. MotherDuck's hybrid execution comes with access and storage of data on the cloud as existing data warehouses do. It differentiates itself by delegating parts of query processing to the client machine, where modern personal computers are powerful enough to handle such workload. This allows queries to be executed in *hybrid* mode, partially local and remote. By utilizing client resources, the hybrid execution model can cut down the cost and response time of queries.

[1] `https://www.snowflake.com/`
[2] `https://cloud.google.com/bigquery/`
[3] `https://motherduck.com/`
[4] `https://duckdb.org/`

In database systems, query optimization plays a critical role, as an optimized query could greatly reduce the response time compared to its non-optimized counterpart. In hybrid execution, the optimizer has an additional challenge – deciding where to run which parts of queries. The decision on execution sites of operations dictates where the transitions between client and server-side execution will be within a query. Whenever there is a transition, (1) the data from one site should be transferred to the other, and (2) the query is executed at the new site until another transition occurs. Therefore, assigning execution sites affects various aspects, including the amount of data sent over the network, the usage of compute resources on the cloud, and response time to the user.

The decision must be made when an operation involves data on the server and the client. Currently, MotherDuck's optimizer operates greedily by always moving "smaller" data. In doing so, the optimizer aims to minimize the amount of data transferred over the network. However, the heuristic it uses to pick out the "smaller" is often incorrect. Moreover, the greedy approach does not lead to a global minimum.

In this research, we develop a *cost-based* hybrid query optimizer that operates.. The optimizer aims to assign the execution sites that lead to a globally optimal strategy. As with the heuristic optimizer, the goal is to minimize the amount of data transfer. We take a Dynamic Programming (DP) approach, which recursively generates (sub-) plans starting from the bottom-most operation. The plans are evaluated by a cost model, and the optimal plans are retained. With the new cost-based optimizer, we expect to see a decrease in query execution time. This leads to our research questions:

RQ 1. How can we make the hybrid query optimizer cost-based?

- **RQ 1-1.** How do we formulate cost-based site selection as a Dynamic Programming problem?
- **RQ 1-2.** How do we implement it in the current MotherDuck architecture?

RQ 2. Does the cost-based optimizer have improvements over the heuristic optimizer?

In Chapter 2, we provide a general overview of database systems, query processing, and query optimization. We also look deeper into DuckDB's query optimizer, MotherDuck's hybrid query execution, and challenges related to statistics in DuckDB that we discovered. Chapter 3 reviews relevant literature on hybrid query processing and cost models in distributed database systems. In Chapter 4, we describe how MotherDuck's current hybrid query optimizer works. The section also covers how we addressed the challenges regarding cardinality estimates. In Chapter 5, we motivate the need for a cost-based hybrid query optimizer and formulate it into a DP problem. The algorithm of the optimizer is explained. Chapter 6 evaluates the cost-based optimizer against the heuristic optimizer and reports the relative speedups. We lay out possible reasons for the slowdown of certain queries. Preliminary results with a modified cost model with constant latency cost are also given. Next, We discuss the limitations of our work and list possible future works in Chapter 7. Finally, we conclude our thesis in Chapter 8.

2

Background

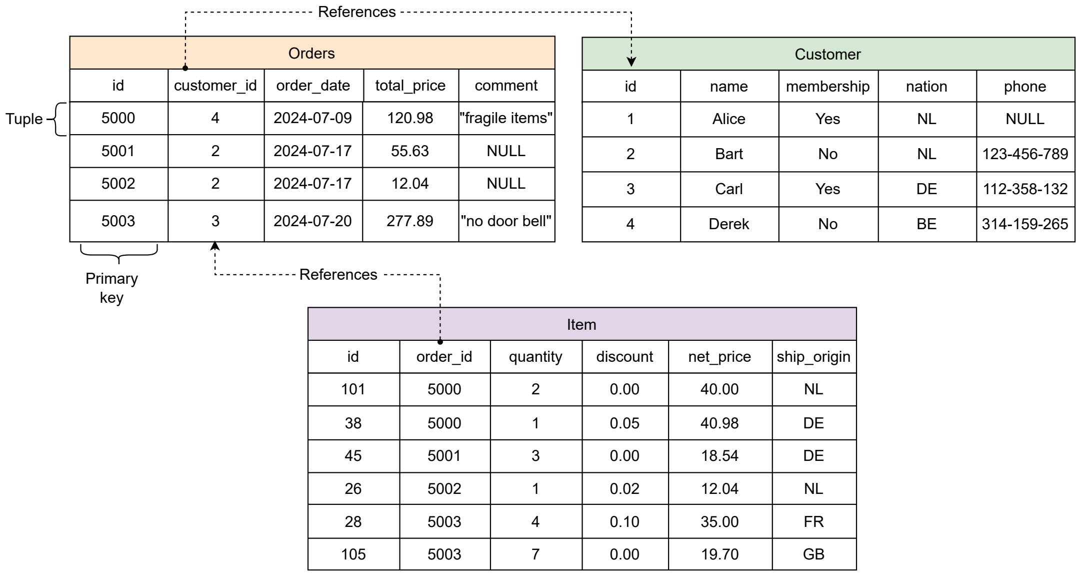
2.1 Relational Database Management Systems

Before the development of database systems, data used to be stored directly on top of file systems. However, this approach came with several drawbacks. For example, redundancy or inconsistency in data stored in different file formats and the inconvenience of writing a task-specific program for each new operation [10, pp. 5–8]. In response, Database Management Systems (DBMS) have emerged as a solution that many software applications now rely on. With DBMS, users can retrieve and store data from databases, which are interrelated collections of data. DBMS provides a convenient and efficient interface for accessing and manipulating data by abstracting details that handle data organization, integrity, security, and more [10, pp. 5–8].

A data model defines how data are organized and related to each other in a DBMS. One of the most widely used data models is the relational model. In a relational model, data are organized in tables (or relations) with fixed numbers of attributes (or columns) identified by unique names. Each row (or tuple) in a table is a record with the attributes specified in the table. Each row also has a unique identifier called a primary key, which can be used to reference tuples in other tables in a database. A DBMS that has a relational model is called a relational database management system (RDBMS). An example of an RDBMS is shown in Figure 2.1.

2.1.1 Relational Algebra and SQL

Queries to RDBMS can be represented using relational algebra [11]. In relational algebra, a relation is considered a set of homogeneous tuples $R = \{(r_{i1}, r_{i2}, ..., r_{in}) \mid i \in 1...m\}$,



Orders				
id	customer_id	order_date	total_price	comment
5000	4	2024-07-09	120.98	"fragile items"
5001	2	2024-07-17	55.63	NULL
5002	2	2024-07-17	12.04	NULL
5003	3	2024-07-20	277.89	"no door bell"

Customer				
id	name	membership	nation	phone
1	Alice	Yes	NL	NULL
2	Bart	No	NL	123-456-789
3	Carl	Yes	DE	112-358-132
4	Derek	No	BE	314-159-265

Item					
id	order_id	quantity	discount	net_price	ship_origin
101	5000	2	0.00	40.00	NL
38	5000	1	0.05	40.98	DE
45	5001	3	0.00	18.54	DE
26	5002	1	0.02	12.04	NL
28	5003	4	0.10	35.00	FR
105	5003	7	0.00	19.70	GB

Figure 2.1: Three relations in a hypothetical webshop database. *Customer* contains information about the shop's customers, and *Orders* is the information on customer orders. There may be multiple orders per customer. *Item* has information for each item in an order.

where $|R| = m$ is the number of tuples (or cardinality), and n is the number of columns (or degrees). An operator is a function that transforms k relations into an output relation.

The five primitives of relational algebra are [10, pp. 48–53]:

- Selection (or filter) $\sigma_{\varphi}(R)$ selects tuples that satisfy predicate φ
- Projection $\Pi_L(R)$ selects all tuples in R with attributes limited to $L = \{a_1, ..., a_n\}$
- Cartesian product $R \times S$ between p-degree relation R and q-degree relation S returns all combination of tuples in R and S. The output relation has degree of $p + q$ and cardinality of $|R| \times |S|$
- Set union $R \cup S$ between the relations with the same attributes returns an output relation that has tuples from both
- Set difference $R - S$ between the relations with the same attributes returns an output relation with tuples from R but not in S

From here, we can extend to other RDBMS operators:

- Natural join $R \bowtie S$ gives an output relation which contains tuples that are equal on their common attribute(s) $C = \{c_1, ..., c_n\}$. Then, the result relation is

$$T = \sigma_{R.c_1 = S.c_1, ... R.c_n = S.c_n}(R \times S),$$

where $R.c$ and $S.c$ denote the attribute c of relation R and S, respectively

- Theta join $R \bowtie_\theta S$ joins two relations where the predicate θ is satisfied. θ is a binary predicate between an attribute a and value v or an attribute b. θ can have one of the following operators: $=$, $\neq$, $<$, $>$, $\leq$, and $\geq$. The output is $T = \sigma_\theta(R \times S)$

- Equijoin is a specific type of theta join, where the predicate is an equality constraint

- Semi-join selects rows from one side of the join only if there is at least one matching row on the other side. If the attributes from the left relation are returned, it is called a left semi-join ($\ltimes$) and right semi-join ($\rtimes$) otherwise

Other operators, such as anti-join and outer join, can be defined using relational algebra.

Structured Query Language (SQL) is a language developed to query database systems, and many RDBMSs use a variation of SQL. SQL syntax can be translated into relational algebraic operators. For instance, `WHERE` clause to σ_φ, `SELECT (...) FROM` to Π_L, and `R JOIN S ON (...)` to $R \bowtie S$. One difference between relational algebra and SQL is that SQL treats a relation as a multiset instead of a set. Therefore, to eliminate duplicate tuples, SQL comes with a new operator `DISTINCT`.

Say we want to find the names of customers who ordered items shipped from the Netherlands (NL) in the webshop database (Figure 2.1). To express the query in relational algebra, we need to join the three relations, filter by the shipping origin of the items, and select the name column from the `Customer`.

$$\Pi_{name}(\sigma_{ship\_origin='NL'}(Customer \bowtie (Item \bowtie Orders))) \tag{2.1}$$

The query can be written in SQL as in Listing 2.1.

```sql
SELECT name
FROM Customer
    JOIN Orders ON Customer.id = Orders.customer_id
    JOIN Item ON Orders.id = Item.order_id
WHERE Item.ship_origin = 'NL';
```

Listing 2.1: SQL query for the names of customers who ordered items shipped from the Netherlands (`'NL'`).

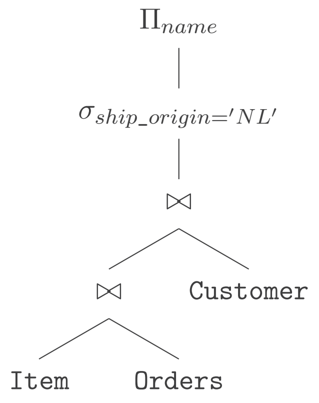
2.2 Query Processing

In RDBMS, a query goes through five phases in its lifecycle [10, pp. 689–691].

Parsing A query written in a human-readable language (likely in SQL) is first translated into the internal data structures of the database system. The parser produces a parse tree and checks its lexical and syntactic validity.

Binding The names in the parse tree are resolved to the underlying catalog objects like relations, tables, views, and columns. In doing so, the binder checks the semantic validity of the query.

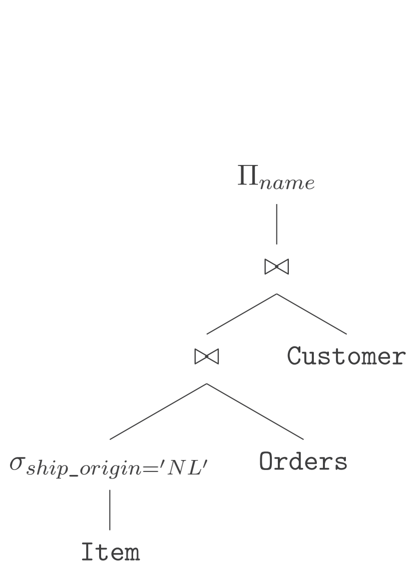
Planning The parse tree is transformed into a logical query plan, a tree-like structure of relational algebra operators. For example, a logical query plan of Listing 2.1 is:



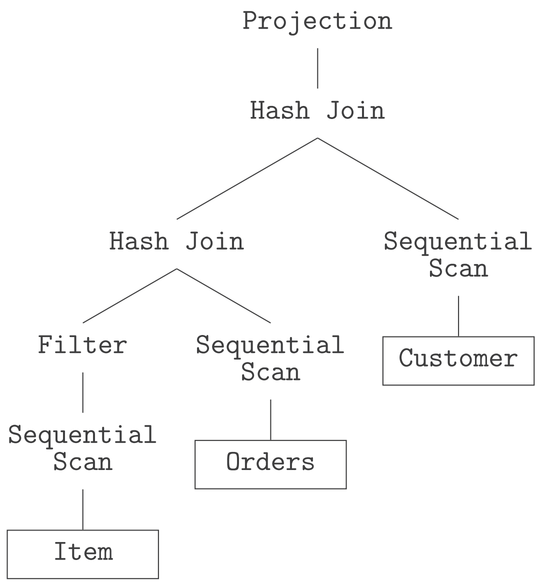
Optimization The query optimizer rewrites the logical plan into an equivalent but more efficient form (see Figure 2.2a). It also determines an efficient implementation for each logical operator. This implementation is called a physical operator. For instance, it chooses how to execute a join $R \bowtie_a S$. Some of the widely used join implementations are [12]

- Nested-loop join: an outer loop iterates over tuples from R and checks for each tuple in S whether the attribute a matches
- Hash join: a hash table is built from values $R.a$ and probed by tuples in S
- Sort-merge join: R and S are first sorted on a then scanned to find matching tuples

As a result, a physical plan Figure 2.2b is generated. Section 2.3 discusses query optimization in more detail.



(a) Optimized logical plan.



(b) Physical Plan

Figure 2.2: Optimized logical and physical plans for Listing 2.1

Execution Finally, a physical plan is evaluated. One approach is to *materialize* the resulting relation from each operation and pass it along to the next. Unless very small, the materialized results are written to disk, meaning there is potentially large read and write overhead [10, pp. 724–725].

An alternative approach is *pipelining*, where a sequence of operations is evaluated without materialization by passing on the results of one operator to the next. For instance, the topmost join and projection in Figure 2.2b can be combined. When the join produces a result tuple, instead of waiting for the rest of the results to generate, it can be passed on immediately to the projection operator to be processed. Pipelining can reduce the amount of data that is materialized and improve the efficiency of query execution.

Pipelined operators are run concurrently, allowing a greater degree of intra-query parallelism. A pipeline ends at a *blocking operator*, which requires a complete result from its child to output any result. Examples of blocking operators are sorting and hash join operations. A hash join can be pipelined with the build side relation, as probing depends on the hash table. Figure 2.3 illustrates how pipelines are defined in a query.

Pipelining can be implemented either pull-based or push-based. A *pull-based* approach is demand-driven. In this approach, a parent operator makes requests to its child, which,

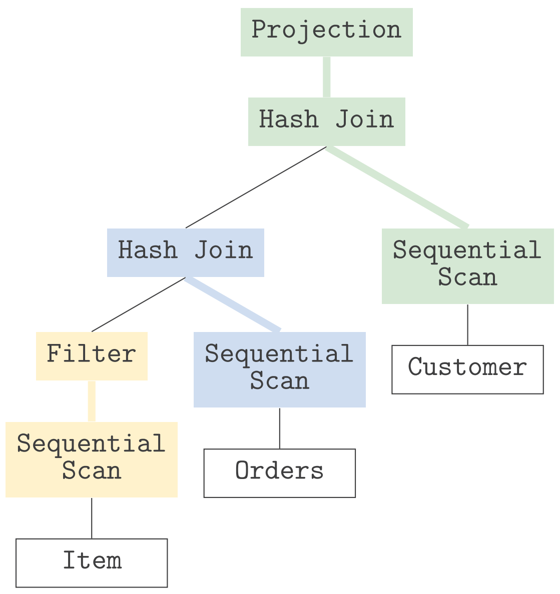


Figure 2.3: Pipelines in the physical plan in Figure 2.2b

in turn, computes a new tuple and returns it to the parent. A *push-based* approach is producer-driven. A child operator eagerly generates tuples that fill up a buffer. Its parent fetches from the buffer when it sees new tuples are present [10, pp. 726–727].

One implementation of pipelining is the Volcano model, where physical operators are evaluated tuple-at-a-time in a demand-driven way [13]. Many function calls to fetch a single tuple each time generate extra overhead, which makes the volcano model inefficient. This is improved by employing a vectorized execution, where a chunk of tuples (thousands) are evaluated at a time [14].

2.3 Query Optimization

The role of a query optimizer is to find the optimal evaluation plan (or physical plan) for a given query. In practice, finding the optimal plan can be time-consuming, especially for complex queries. In such cases, selecting a 'good enough' plan is acceptable. Optimization takes place on two levels. At the relational algebra level, the optimizer searches for the most efficient logical plan among many logically equivalent candidates. At the execution strategy level, the optimizer looks for *physical* details that maximize the efficiency. This includes the execution algorithm of operators, the use of indexes, and more [10, p. 743]. Query optimization is a crucial step in query processing, as the difference between the execution times of optimized and unoptimized query plans is often significant.

Query optimizer requires three components: a set of logically equivalent query plans that can be used to execute the query (*search space*), a technique to estimate the cost of query plans (*cost estimation*), and an algorithm to explore the search space (*plan enumeration*) [15]. A search space is generated using the equivalence rules that transform a query plan. An efficient plan enumerator is necessary to explore potentially very large search space. The cost estimation relies on statistics and cardinality estimation to evaluate the cost of the enumerated plans. We will discuss the three components in detail in the following sections.

2.3.1 Query Plan Transformation

A set of equivalence rules transforms a query plan into a different but logically equivalent form. The resulting alternative plans constitute a search space. Here, we describe a few of the most commonly applicable equivalence rules.

ER1. Natural and theta joins are commutative, *i.e.,* $R_1 \bowtie R_2 \equiv R_2 \bowtie R_1$.

ER2. Natural and theta joins are associative, *i.e.,* $(R_1 \bowtie R_2) \bowtie R_3 \equiv R_1 \bowtie (R_2 \bowtie R_3)$.

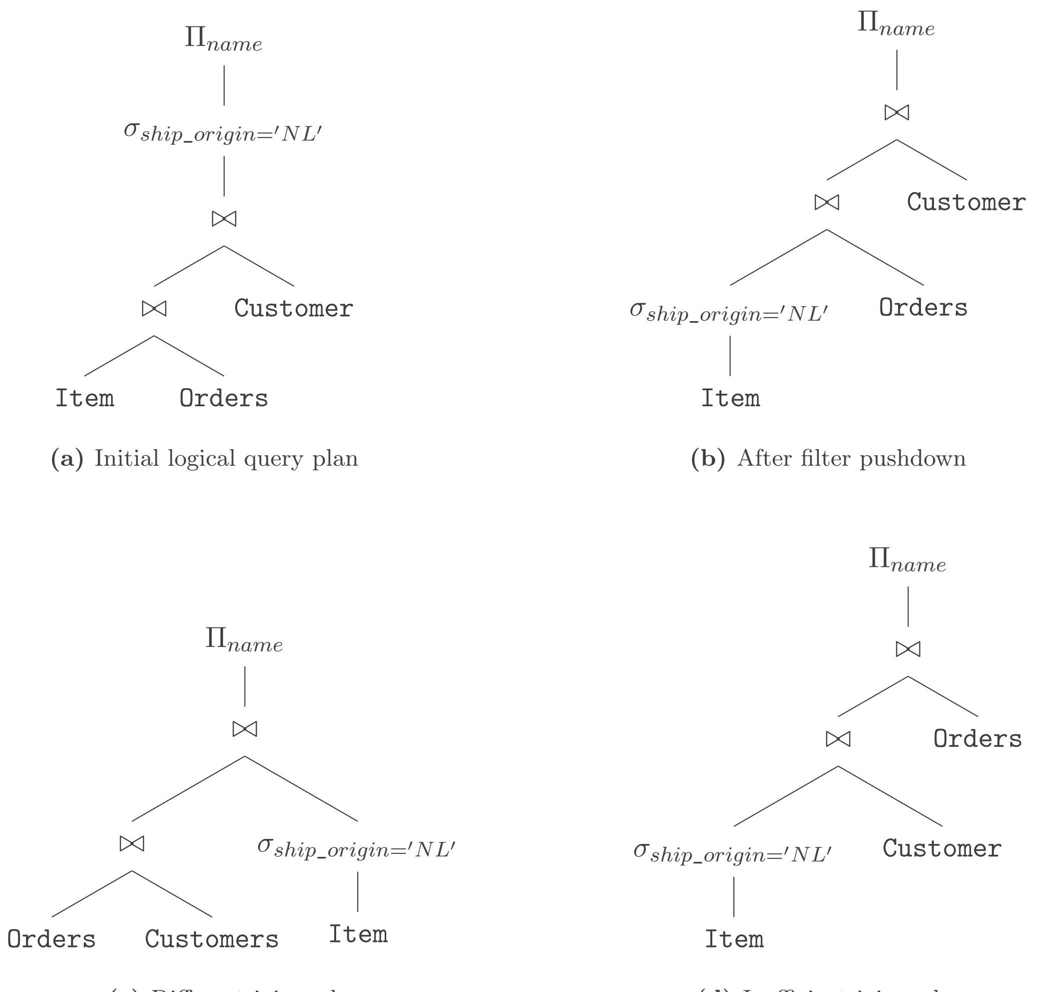
ER3. Selection is distributive over joins, *i.e.,* $\sigma_{\varphi_1 \wedge \varphi_2}(R_1 \bowtie R_2) \equiv (\sigma_{\varphi_1}(R_1)) \bowtie (\sigma_{\varphi_2}(R_2))$, given φ_1 only has attributes from R_1 and φ_2 from R_2.

ER4. Projection is distributive over joins, *i.e.,* $\Pi_{L_1 \cup L_2}(R_1 \bowtie R_2) \equiv (\Pi_{L_1}(R_1)) \bowtie (\Pi_{L_2}(R_2))$, given L_1 only has attributes from R_1 and L_2 from R_2.

Filter pushdown is a common optimization rule based on the ER3 introduced above. The selection operator can reduce the size of the intermediary result and save the resources required for subsequent operations. Figure 2.4b shows the logical plan after a filter $\sigma_{ship\_origin='NL'}$ is pushed down below the join between $Item$ and $Orders$. By returning fewer tuples from $Item$, the filter pushdown makes the join operation more efficient.

Join ordering is another crucial optimization step, made possible by the commutativity and associativity of joins (ER1 and ER2). The initial join order is $(Item \bowtie Orders) \bowtie Customer$. However, $Item \bowtie Orders$ is likely to be a large relation, as it will create a tuple for every line of items. On the other hand, $Orders \bowtie Customer$ will be a smaller relation as it is likely that $Orders$ have fewer tuples than $Item$, and $Customer$ have even fewer. Therefore, a better join order would be $(Orders \bowtie Customer) \bowtie Item$. Now, bringing the filter pushdown into the picture, it becomes

$$(Orders \bowtie Customer) \bowtie (\sigma_{ship\_origin='NL'}(Item)). \qquad \text{(Figure 2.4c)}$$



(a) Initial logical query plan **(b)** After filter pushdown **(c)** Different join order **(d)** Inefficient join order

Figure 2.4: Logical query plans with different transformation applied on query Listing 2.1).

Since $\sigma_{ship\_origin='NL'}(Item)$ has fewer tuples, it is possible that keeping the initial order generates smaller intermediate results, *i.e.,*

$$((\sigma_{ship\_origin='NL'}(Item)) \bowtie Orders) \bowtie Customer. \qquad \text{(Figure 2.4b)}$$

An example of inefficient join order is

$$((\sigma_{ship\_origin='NL'}(Item)) \bowtie Customer) \bowtie Orders \qquad \text{Figure 2.4d}$$

where *Item* and *Customer* have no common attribute, resulting in a large Cartesian product. These examples showcase the decisions that the optimizer has to make based on its cost estimation.

2.3.2 Cost Estimation

Cost model evaluate the execution costs of query plans based on the statistics, execution methods, etc. The basis of cost estimation in DBMSs is proposed in System-R [11, 16]:

1. Gather statistics from base tables.
2. For each operation, determine the output statistics given the statistics of its inputs.
3. Estimate the execution cost of the operation.

Steps 2-3 are executed iteratively as the optimizer traverses the tree of logical query plan bottom-up, typically starting from reading the base tables. The base table statistics are usually stored as catalog information in many DBMSs. Information such as the number of tuples, the number of distinct values, and attribute histograms is available. A *histogram* divides the values in a column into several buckets. It allows the optimizer to exploit the data distribution to derive output statistics for operations such as filters. Multidimensional histograms can be created to capture the joint distribution of correlated attributes. A more simple statistic, *min/max*, can be used in place of histograms, though it does not provide an insight into the data distribution as histograms do.

The cost estimation of an operation depends on the cardinality of its operands. Therefore, it is crucial to have an accurate cardinality estimation function. Filter and join are two operators that can significantly affect the sizes of intermediate data and their subsequent operators. Let us define n_R as the number of tuples in relation R and $V(A, R)$ as the number of distinct values of attribute A in relation R. Then, assuming uniform distribution, $\sigma_{A=v}(R)$ will result in size $\frac{n_R}{V(A,R)}$. The selectivity of a filter is defined as the probability that a tuple satisfies the filter condition, which is $\frac{1}{V(A,R)}$ in this case. If there is a histogram available, the size is $\frac{\text{frequency count in bucket } b}{\text{size of bucket } b}$, where b is the bucket where the value v belongs. A more complex approach is needed for non-equality predicates or combined predicates.

Let us consider a natural join $R \bowtie S$. Without common attributes, the result size is simply a Cartesian product, $R \times S$, with size $= n_R \cdot n_S$. If there is a common attribute A, and it is a primary key of R, the maximum size of the result is n_R. If A is a foreign key of S referencing R, then $|R \bowtie S| = |S|$. In the case A is not a primary key of either relation, we estimate the result size as

$$\frac{n_R \cdot n_S}{\max\left(V(A,R), V(A,S)\right)}. \tag{2.2}$$

The numerator (Cartesian product) is the maximum size of the join, whereas the denominator is its selectivity. Join selectivity here assumes a uniform distribution of A and that a tuple in one relation is less likely to find a matching tuple if A in the other relation has more distinct values. More concretely, if $V(A, R) = 5$ and $V(A, S) = 100$, we estimate the probability that a specific $R.A$ value is present in S is $1/100$.

Theta join $R \bowtie_\theta S$ can be rewritten as $\sigma_\theta(R \times S)$. The size is then estimated by multiplying the filter selectivity by the cardinality of the Cartesian product $(n_R \cdot n_S)$. The join cardinality estimation can be refined with histograms or *min/max*.

Many cost estimation techniques rely on a column's distinct count of values. Therefore, having an accurate distinct count estimate is crucial. HyperLogLog (HLL) is the state-of-the-art probabilistic algorithm for distinct count estimation in a multiset, which provides memory-efficient approximation called *sketches* [17]. HLL is based on the fact that the distinct count of uniformly distributed random numbers can be estimated using the maximum number of leading zeros in their binary representations. HLL first applies a hash function to the elements in a multiset and tracks the maximum length n of 0-prefix in the hash values. Then, the distinct count is approximated as 2^n. To account for the variance in the multiset, the multiset is divided into m disjoint subsets, and HLL takes the harmonic mean of the results as the distinct count of the entire set.

Note that statistics are not perfect. Cardinality estimation often assumes a uniform distribution of data and struggles to account for correlations between attributes. However, real-world data are frequently skewed or highly correlated [18]. For example, a filter on one of the two highly correlated attributes will affect the distribution of the other. Many algorithms have been proposed to detect correlations – by constructing multivariable histograms [19, 20], or by using data-oriented knowledge [21]. Umbra [1] combines HLL sketches with sampling, which enables efficient estimation of distinct count and corrects the correlation bias among multi-column estimates from sampling. [22]. Although there are many efforts to improve cardinality estimations, errors are inevitable. And the error can propagate or even explode as the optimizer proceeds to subsequent operations.

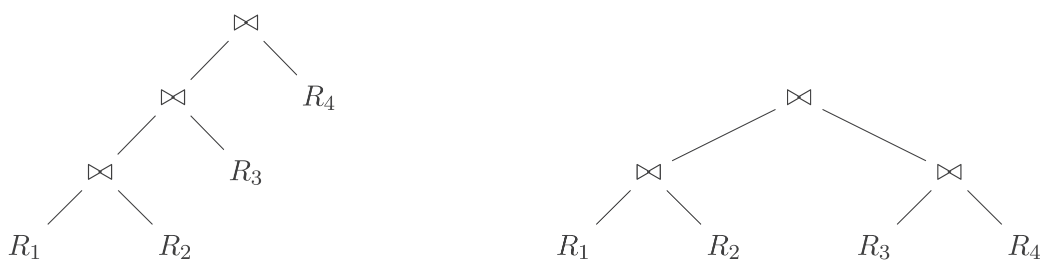
In the case of joins, the cardinality estimation errors grow exponentially with the number of joins involved in a query [23]. In addition, when multiple predicates are present in a join, some optimizers assume independence of the predicates, while others rely on the most selective predicate for their estimations. When relevant statistics are unavailable, some databases (*e.g.,* System-R and DuckDB) use constant selectivity values. The assumptions and heuristics of the optimizer are far from perfect.

[1]`https://umbra-db.com/`

Summary Cost estimation is necessary for the optimizer to determine the most cost-effective plan among many logical equivalents. One of the most important factors in a plan's cost is the intermediate result size (cardinality), which can be estimated using statistics. Accurately estimating the cardinalities of filters, joins, and distinct counts is difficult. Optimizers use statistics such as histograms and *min/max* but fall short regarding correlations, data distribution, and complex predicates. The errors in statistics could escalate as they propagate through a query tree. Good cost estimation is the core of query optimization, as "optimization is only as good as its cost estimates" [15].

2.3.3 Plan Enumeration

Now, the optimizer needs an efficient algorithm to explore the search space. Earlier query optimizers such as System-R focused primarily on joins and inspired many following query optimizers with their concepts of dynamic programming and interesting order [16]. It also restricts the shape of join orders to a left-deep model Figure 2.5. And, as with any dynamic programming problem, it imposes the principle of optimality. Then, to obtain the optimal solution for $\{R_1, R_1, R_3, R_4\}$, it is sufficient to consider the optimal orderings of the sub-problems, namely, $(\{R_1, R_2, R_3\} \bowtie R_4)$, $(\{R_1, R_2, R_4\} \bowtie R_3)$, $(\{R_1, R_3, R_4\} \bowtie R_2)$, and $(\{R_2, R_3, R_4\} \bowtie R_1)$, where the left-hand side of the join is in some optimal order.



(a) Left-deep joins **(b)** Bushy joins

Figure 2.5: Different join shapes

Solutions to a (sub-) problem are a set of plans that are optimal out of their *logical* equivalents. Note that logical equivalents may vary in their physical properties. Including the physical properties of a query plan increases the search space, as a logically equivalent plan can have multiple physical variants. The tuple order of a join output is one physical property that System-R considers. Take

$$R_1 \bowtie R_2 \bowtie R_3$$

joined on common attribute A. For a sub-problem $R_1 \bowtie R_2$, joining via a nested-loop join may be cheaper than a sort-merge join. But with a sort-merge join, the output is sorted on A, which significantly decreases the cost of the join with R_3 later. Therefore, System-R retains the plans for all *interesting orders*, including the ones that are not immediately the cheapest but could potentially provide a cheaper overall cost. Another physical property System-R takes into account is *access path*, whether to perform sequential or index scan of a table. The cost of an access path is estimated based on the CPU and disk utilization,

$$\text{cost} = (\text{page fetches}) + W \cdot (\text{RSI calls}), \tag{2.3}$$

where RSI is System-R's internal storage system and W is a weighting factor between CPU and disk. There are outer and inner tables for the join methods used in System-R. The outer table "drives" the join process. Each tuple in the outer table is evaluated against the inner table, which is called repeatedly. The cost of a join operation becomes

$$C_{outer}(path1) + N \cdot C_{inner}(path2), \tag{2.4}$$

where N is the product of all selectivity factors and cardinalities of relations joined so far. The cost of a join order is the sum of the costs of all intermediate joins. Plans with the same logical and physical properties but higher costs are pruned [16].

R* is an extension of System-R for distributed database systems. Due to its distributed settings, R* considers additional decision variables, which increase its search space. The new concerns are 1) where to execute joins and 2) how to transfer the participating relations over the network [24]. The cost is now a linear combination of the intra-site costs (CPU and I/O) and network costs (number of messages and bytes transferred over the network). Like System-R, plans with higher costs are pruned.

$$\begin{aligned}\text{cost} = {} & W_{CPU} \cdot (\#\ instructions) + W_{I/O} \cdot (\#\ I/Os) \\ & + W_{MSG} \cdot (\#\ messages) + W_{BYT} \cdot (\#\ bytes)\end{aligned} \tag{2.5}$$

Later, extensible plan enumerators such as Starburst [25] and Volcano [13] emerged to allow easier extensions of optimizer architecture [15]. Extensible optimizers can extend to address non-inner joins, new join methods, or other operators (*e.g.,* aggregation).

2.3.4 DPhyp Plan Enumerator

DPhyp is a state-of-the-art plan enumerator proposed by Moerkotte *et al.* [26]. It has its basis in previous DP-based approaches, including DPsize (derived from [27]) and DPccp [28]. DPhyp supports an efficient way of handling complex join predicates and the most number of non-inner join types. It does so by modeling join relations as a hypergraph – where nodes are relations and edges connect joined relations.

Definition 2.3.1 (Hypergraph)**.** *A hypergraph is a pair $H = (V, E)$ such that*

1. *V is a non-empty set of nodes and*

2. *E is a set of hyperedges, where a hyperedge is an unordered pair (u, v) of non-empty subsets of $V (u \subset V, v \subset V)$ and $u \cap v = \emptyset$.*

A non-empty subset of V is a hypernode, and the nodes in V are ordered in some arbitrary relation $\prec$.

Definition 2.3.2 (Subgraph)**.** *Let $H = (V, E)$ be a hypergraph and $V' \subseteq V$ subset of nodes. Subgraph $G|_{V'}$ is defined as $G|_{V'} = (V', E')$ where $E' = \{(u, v) \mid (u, v) \in E,\ u \subseteq V',\ v \subseteq V'\}$.*

Definition 2.3.3 (Connected)**.** *Let $H = (V, E)$ be a hypergraph. H is connected if $|V| = 1$ or there exists a partitioning V', V'' of V and a hyperedge $(u, v) \in E$ such that $u \subseteq V'$, $v \subseteq V''$, and both $G|_{V'}$ and $G|_{V''}$ are connected.*

Definition 2.3.4 (csg-cmp-pair)**.** *Let $H = (V, E)$ be a hypergraph and S_1 and S_2, two subsets of V such that $S_1 \subseteq V$, $S_2 \subseteq (V - S_1)$, and $G|_{S_1}$ and $G|_{S_2}$ are connected subgraphs. $G|_{S_2}$ is then called a connected complement of $G|_{S_1}$. If there is a hyperedge $(u, v) \in E$ such that $u \subseteq S_1$ and $v \subseteq S_2$, then (S_1, S_2) is a csg-cmp-pair (ccp).*

All csg-cmp-pairs must be explored to find the optimal solution. The enumerator must do so in an efficient and ordered way (all csg-cmp-pairs (S_1', S_2') where $S_1' \subseteq S_1$ and $S_2' \subseteq S_2$ are enumerated before (S_1, S_2)). The main idea behind generating csg-cmp-pairs is to iteratively extend connected subgraphs by new nodes in its *neighborhood*. To find the neighborhood of connected subgraph S, we first define $E_\downarrow$ as the non-subsumed minimal subset of E such that for all $(u, v) \in E$ there is $(u', v') \in E_\downarrow$ with $u' \subseteq u$ and $v' \subseteq v$. And hypernodes u' and v' must not be part of the set X, which contains nodes to exclude from neighborhood candidates. Then the neighborhood is $\mathcal{N}(S, X) = \bigcup_{v \in E_{\downarrow(S,X)}} \min(v)$. To avoid potential redundancies, removing subsumed edges ensures that $E_\downarrow$ contains the "smallest" hyperedges.

The driver function for the DPhyp algorithm is shown in Algorithm 1, which is based on the four general principles outlined by the authors:

1. csg-cmp-pairs are constructed by enumerating csg's

2. csg's and their cmp's are created by recursive graph traversals

3. some nodes are forbidden (belongs in exclusion set X) to avoid duplicates
4. connected subgraphs are extended by following edges to the neighborhood nodes. The connecting hyperedge is $n : 1$, from the hyper node v (where $|v| = n$) to $\min(u) \in E_{\downarrow}$ (where $|\min(u)| = 1$)

Algorithm 1 Pseudocode for driver function `Solve()` in DPhyp [26]

```
1: for each $v \in V$
2:     $dpTable[\{v\}]$ = plan for $v$              ▷ initialize $dpTable$
3: for each $v \in V$ descending according to $\prec$
4:     EmitCsg($\{v\}$)                         ▷ process singleton sets and
                                                   call EnumerateCmpRec recursively to find cmp's.
5:     EnumerateCsgRec($\{v\}, \mathcal{X}_v$)     ▷ expand singleton sets.
                                                   internally call (1) EmitCsg to extend the node with
                                                   each neighbor and (2) EnumerateCsgRec to recur-
                                                   sively extend the connected subgraph given the ex-
                                                   clusion set $\mathcal{B}_v = \{w \mid w \prec v\} \cup \{v\}$.
   return $dpTable[V]$
```

The only fully reorderable join type is inner join ($\bowtie$). Other join types – outer join, anti-join, semi-join, nestjoin, and dependent join [1] – are not. Let $\mathcal{LOP}$ be a set of left variants of non-fully reorderable joins, and let $\xrightarrow{1}$ and $\xrightarrow{2}$ be operators in $\mathcal{LOP}$. The following equivalence rules can be derived to transform the queries:

$$(R \xrightarrow[\theta_{RS}]{1} S) \xrightarrow[\theta_{RT}]{2} T = (R \xrightarrow[\theta_{RT}]{2} T) \xrightarrow[\theta_{RS}]{1} S \tag{2.6}$$

$$(R \bowtie_{\theta_{RS}} S) \xrightarrow[\theta_{ST}]{2} T = R \bowtie_{\theta_{RS}} (S \xrightarrow[\theta_{ST}]{2} T) \tag{2.7}$$

$$(R \bowtie_{\theta_{RS}} S) \xrightarrow[\theta_{RT}]{2} T = R \bowtie_{\theta_{RS}} (S \xrightarrow[\theta_{RT}]{2} T) \tag{2.8}$$

Rule 2.6 can be rewritten with right variant of $\xrightarrow{1}$ as

$$(S \xleftarrow[\theta_{RS}]{1} R) \xrightarrow[\theta_{RT}]{2} T = S \xleftarrow[\theta_{RS}]{1} (R \xrightarrow[\theta_{RT}]{2} T)$$

Equivalent query plans for non-inner joins are derived from applying these equivalence rules. However, not all reorderings output the same result as the original plan. Thus, a technique to restrict the search space to remove "conflicting reorderings" is required. DPhyp does so using extended eligibility list (EEL) proposed in [29]. DPhyp extends EEL with syntactic eligibility list (SES) and total eligibility list (TES) for more efficient conflict detection and reporting.

[1] A dependent join is where one side of the join is dependent on the evaluation of the other. Defined as $R \bowtie\!\!\lhd_{\theta} = \{r \circ s \mid r \in R, s \in S(r),\ \theta(r, s)\}$

Summary DPhyp is a plan enumerator algorithm based on DP and hypergraphs. It recursively constructs csg-cmp-pairs by incrementally extending connected subgraphs with their neighbors. It supports complex join predicates and provides much more efficient reordering of non-inner joins than existing approaches.

2.4 DuckDB

DuckDB is an in-process database optimized for Online Analytical Processing (OLAP) workloads [30]. Much like SQLite, DuckDB runs embedded in a host process without a need for servers. This enables fast data transfer and sharing and avoids the disadvantages of traditional stand-alone database systems, which are hard to set up and restricted by client protocols [30, 31]. DuckDB's embeddability and simplicity also suit recent use cases of data analytics work, where many data scientists run queries on their local PCs [32]. Moreover, to support faster execution of OLAP workloads, characterized by long-running, complex queries over a large amount of data [33], DuckDB employs state-of-the-art techniques such as columnar storage and vectorized execution combined with morsel-driven parallelism [34] among others.

DuckDB is also extensible with new functionalities. For instance, JSON or Parquet extensions allow users to work on different file formats. The extension modules are allowed a wide range of changes, including data types, optimizer rules, and parser [35]. One such extension module is MotherDuck's hybrid query execution, which will be discussed in Section 2.6

2.4.1 Join Order Optimizer

DuckDB follows the query processing steps outlined in Section 2.2. In the optimization phase, enumerating the entire search space is expensive. Therefore, many optimizers apply heuristics to reduce the search space. An example is how some database systems limit the join shape to a left-deep tree [16, 36]. In DuckDB, the optimizer applies a series of rules early on before starting join ordering. These include filter pushdown, sub-expression rewriting, and more. Then comes the join ordering, followed by a few more optimization rules, including removing unused columns. Here, we delve into their join order optimizer.

An overview of DuckDB's join order optimization process is illustrated in Figure 2.6. DuckDB uses DPhyp as its enumerator, which emits pairs of relations to be joined. Then, the cardinality and cost are estimated. Finally, the *join node*, consisting of the cost and reference to the child joins, is stored in *DP table*. In the following sections, we explain how

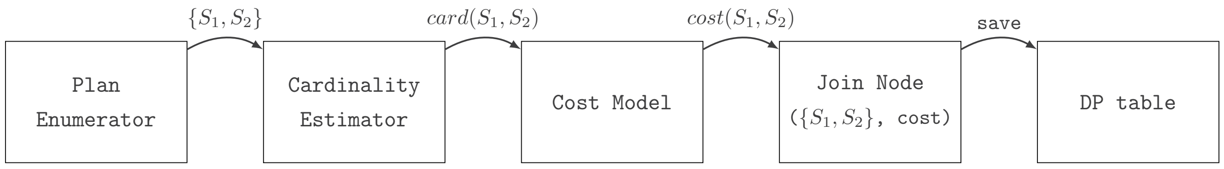


Figure 2.6: Join order optimization process in DuckDB. Diagram adapted from [1].

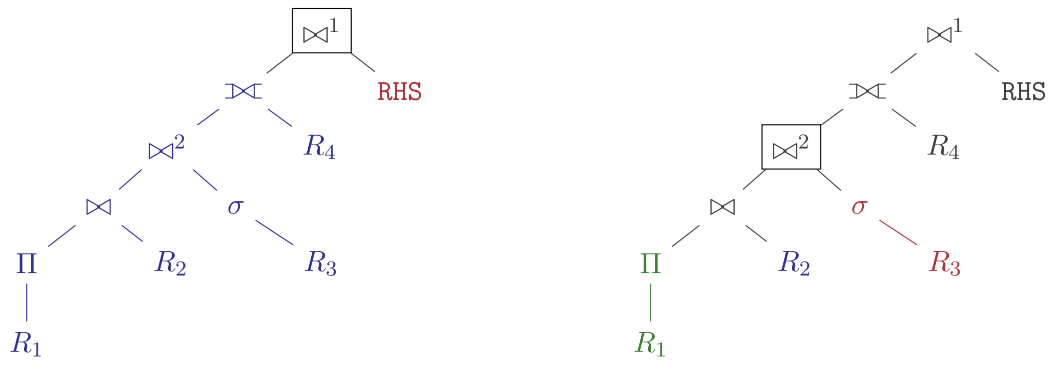
DuckDB identifies reorderable operators, then discuss how it computes the cardinality and the cost of joins.

2.4.1.1 Building hypergraphs

This section describes how DuckDB identifies a set of relations to reorder, which form the nodes in a hypergraph. To better explain DuckDB's logic, we introduce a concept of *reorderable block* as a unit in a join order problem. A reorderable block is a subtree of a query plan, and a join order problem involves multiple reorderable blocks. Reordering happens on a block level – while the blocks can be reordered during optimization, their internal operators are fixed. Reorderable blocks are constructed recursively in a bottom-up style, where the sub-reorderable blocks are considered to have already been optimized or cannot (further) be optimized. A hypergraph is built from the identified reorderable blocks, considering each block as a *node*, and it is passed on to the DPhyp enumerator to find the optimal join order. A block is *finalized* when it is in an optimal join order.

An example is illustrated in Figure 2.7. Looking from the root of the plan ($\bowtie^1$), there are two reorderable blocks – marked in blue and red (see Figure 2.7a). Let us focus on the left-hand side (the blue block). The outer join (⟗) is not reorderable in DuckDB, but its left and right subtrees can still be optimized. The right child (R_4) is naturally optimized, but the left-hand side ($\bowtie^2$) needs further optimization. From the $\bowtie^2$ subtree, we can identify three sub-reorderable blocks $\Pi(R_1)$, R_2, and $\sigma(R_3)$ (see Figure 2.7b). A hypergraph is created from the three blocks, and the optimal join order is found. When the blocks are rearranged, $\bowtie^2$ is finalized. This means that its parent, ⟗ block, is also finalized as both children ($\bowtie^2$ and R_4) are optimized. Returning to the root $\bowtie^1$, the optimizer reorders ⟗ and `RHS` blocks, finalizing the root block and completing the join order optimization.

Let us discuss how DuckDB implements this logic internally. DuckDB's `JoinOrderOptimizer` has a `QueryGraphManager` responsible for extracting relations and building hypergraphs. `QueryGraphManager` has a function `ExtractJoinRelations`, which recursively constructs a set of relations to be reordered. The function receives an operator `input_op` as an input and returns a boolean for whether the `input_op` block is reorderable. The returned



(a) Two reorderable blocks at $\bowtie^1$ **(b)** Three reorderable blocks at $\bowtie^2$

Figure 2.7: Reorderable blocks, each identified by color

boolean informs the `JoinOrderOptimizer` whether to start the optimization process of the `input_op` block.

`ExtractJoinRelations` starts by traversing a given query tree until it encounters an operator `op` it cannot pass through (*e.g.,* join or projection). There are a few different cases to handle.

- If `op` is not reorderable, the subtree headed by `input_op` subtree must be treated as one unit – like the $\bowtie$ block in Figure 2.7a. However, `op`'s child subtrees may still be reorderable; therefore, new `JoinOrderOptimizer`s are called for the children. When `op`'s children are finalized, then `input_op` block is also finalized. The block as a whole can participate in a join ordering of some parent join, so the return value for this case is `true`.

- If `op` is a reorderable join, we should look further below the tree (like the $\bowtie^2$ block in Figure 2.7b). Therefore, `ExtractJoinRelations` is recursively called with its children as inputs and returns the sub-reorderable blocks. Once all sub-reorderable blocks are identified, `JoinOrderOptimizer` starts the subsequent hypergraph building, enumeration, and cost-based join order decision process (see Figure 2.6). Once the process is completed, `input_op` block is finalized. However, `input_op` block itself can only participate in join ordering when both children return `true`, *i.e.,* the return value for this case is `ExtractJoinRelations(left) AND ExtractJoinRelations(right)`.

- If `op` is a projection, aggregate, or window operator, a new join optimization starts for its child. When `op`'s child is finalized, `input_op` block is also finalized.

- If `op` is a get operator, `input_op` block is already finalized.

- If `op` is some other operator, the `op` subtree is not reorderable. `input_op` block is finalized and informs and returns `false`.

The DPhyp enumerator explores possible join orders for each identified reorderable block and populates the *DP Table* with the cost and cardinality. The join order optimizer selects the optimal plan and generates a new subtree with optimal ordering. The reordered (and finalized) block is returned to the caller, where the statistics of `op` are extracted and propagated so the parent block can access relevant and up-to-date information during its optimization.

2.4.1.2 Cardinality estimator

This section describes how DuckDB estimates the cardinality of join outputs. Compared to other state-of-the-art database systems, DuckDB has limited statistics. One reason is that data can be sourced directly from parquet or CSV files. Additionally, the cost of maintaining detailed statistics of large data used in OLAP is significant. Gathering, storing, and maintaining up-to-date statistics introduce overhead, which overwhelms the performance gains. Therefore, DuckDB's current cost estimation relies on little to no statistics [1].

DuckDB assumes that the equi-joins are foreign-key primary-key (FK-PK) joins, which ensures the *containment of value sets* [1]. This is the assumption that in $A \bowtie_{A.Y_{FK}=B.Y_{PK}} B$, every value of foreign key $A.Y_{FK}$ exists in primary key $B.Y_{PK}$. Then, the cardinality of the join output can be estimated from the properties of join attributes, such as their selectivity or duplicity. To this end, DuckDB introduces the following concepts:

- Multiplicity: the average frequency of each distinct value in a column
- Selectivity: the probability a column domain value is found in a column
- Current Domain (*cdom*): the number of distinct values in a column
- Total Domain (*tdom*): the number of distinct values in the column's total domain

The selectivity and multiplicity of some column $A.x$ can be defined as:

$$sel(A.x) = \frac{cdom(A.x)}{tdom(A.x)} \tag{2.9}$$

$$mul(A.x) = \frac{card(A)}{cdom(A.x)} \tag{2.10}$$

Note that the term *relation* in this section essentially refers to a node in a hypergraph (or reorderable block). Let us consider the join of two relations A and B on $A.w = B.x$. The number of tuples z in A resulting in $A \bowtie B$ is

$$\begin{aligned} card(A,B)_A &= card(A) \cdot P(z \in B.x) \cdot mul(B.x) \\ &= card(A) \cdot sel(B.x) \cdot mul(B.x) \\ &= \frac{card(A) \cdot card(B)}{tdom(B.x)} \end{aligned} \tag{2.11}$$

Similarly,

$$card(A,B)_B = \frac{card(B) \cdot card(A)}{tdom(A.w)} \tag{2.12}$$

Since DuckDB assumes the *containment of value sets*, $tdom(A.w) = tdom(B.x)$, making Equations (2.11) and (2.12) equivalent. Therefore, the join selectivity is

$$jsel(A.w, B.x) = \frac{1}{tdom(A.w)}$$

and the join cardinality is

$$\begin{aligned} card(A,B) &= \frac{card(A) \cdot card(B)}{tdom(A.w)} \\ &= card(A) \cdot card(B) \cdot jsel(A.w, B.x) \end{aligned} \tag{2.13}$$

The formula can extend for joins with n relations with an arbitrary number of predicates (noted as $*$). For instance, the cardinality of three-way join is

$$card(C \bowtie (A \bowtie B)) = card(A) \cdot card(B) \cdot card(C) \cdot \tag{2.14}$$

$$jsel(C.*, B.*) \cdot jsel(C.*, A.*) \cdot jsel(A.*, B.*) \tag{2.15}$$

This is similar to how System-R computes join sizes – multiplying the product of all cardinalities of participating relations with the product of all join selectivities Section 2.3.3. However, previous studies point out that the cardinalities are underestimated [18, 23]. DuckDB addresses this problem by formulating it as a graph problem. Then, a graph $G = (\mathcal{R}, \mathcal{J})$ is constructed, where the nodes are a set of relations $\mathcal{R} = \{R_1, ..., R_n\}$ and edges are a set of join selectivities $\mathcal{J}$ where for all $jsel(A.*, B.*) \in \mathcal{J}$, $A \in \mathcal{R}$ and $B \in \mathcal{R}$. A minimum spanning tree of the graph finds a path with join selectivities such that each relation has at most one applicable join selectivity and the product of the join selectivities is minimal (*i.e.,* most selective). The cardinality equation is

$$card(\mathcal{R}) = \Gamma(\mathcal{R}, \mathcal{J}) \cdot \prod_{R_i \in \mathcal{R}} card(R_i), \tag{2.16}$$

where $\Gamma(\mathcal{R}, \mathcal{J})$ is the product of the join selectivities in the minimum spanning tree of $G = (\mathcal{R}, \mathcal{J})$ [1].

Equation (2.16) assumes that distinct value counts are available for the relations which DuckDB estimates using HLL on 10% of values in a column of a base table. However, a different approach is needed if the estimation is not available (*e.g.,* data is read directly from Parquet or CSV files). Relations in graph G share the same *tdom* value, which is initialized based on two cases. 1) If distinct counts are known for a subset of join attributes, the highest distinct count will be the *tdom*. 2) Otherwise, the lowest cardinality among the join attributes will be the *tdom*. For $A \bowtie KB$ with $card(A) > card(B)$, DuckDB assumes that K is a foreign key in A and a primary key in B. Due to the containment of value sets, all values of $A.K$ are found in $B.K$ (*i.e.,* $set(A.K) \subseteq set(B.K)$). So, $tdom(K)$ is $card(B)$.

The join selectivity $jsel$ is then the inverse of the *tdom*. Moreover, the availability of distinct count statistics affects how DuckDB determines the base table filter's selectivity. By default, DuckDB applies its ad-hoc selectivity of 20%. However, if HLL statistics are present, a more accurate estimation is made for the filter. For instance, the selectivity of filter $A.x = 100$ will be $\frac{card(A)}{tdom(A.x)}$.

2.4.1.3 Cost model

As suggested by [18], DuckDB calculates the cost of a join plan as the sum of the estimated cardinalities of all intermediate joins. As examples,

$$\begin{aligned}
card(A \bowtie B) &= card(A) \cdot card(B) \cdot jsel(A.*, B.*) \\
cost(A \bowtie B) &= card(A \bowtie B) \\
\\
card(C \bowtie (A \bowtie B)) &= \ card(A) \cdot card(B) \cdot card(C) \cdot \\
&\quad\ jsel(C.*, B.*) \cdot jsel(C.*, A.*) \cdot jsel(A.*, B.*) \\
cost(C \bowtie (A \bowtie B)) &= card(A \bowtie B) + card(C \bowtie (A \bowtie B)) \\
&= card(A \bowtie B) \cdot (1 + card(C) \cdot jsel(B.*, C.*))
\end{aligned}$$

2.4.1.4 Deciding left and right relations

DuckDB performs 'left-right optimization' as the last stage of join order optimization. It decides which relations to be on the left and right side of the join ($A \bowtie B$ or $B \bowtie A$). The purpose of this optimization is to improve efficiency when executing the join. For hash joins, building a hash map from a smaller table is more efficient due to faster building and probing time. During execution, DuckDB will take the right side of the join as the build side and the left as the probe side. The left-right optimization rearranges the relations such that a smaller relation is placed on the right. DuckDB as ad-hoc heuristics to decide when to 'flip' the relations. For inner and outer joins, the relations are flipped when $card(A) < card(B)$. For left/right outer, semi-, and anti-joins, they are flipped when $card(A) < card(B) * 2$.

DuckDB v1.1, they introduce an improved optimization (renamed `BUILD_PROBE_SIDE_-OPTIMIZER`). Unlike the previous optimization, which was based purely on cardinality, the new rule is based on *build size*. The build size is $cardinality \times row\_width$, where row_width is estimated as the number of columns, plus the penalties of variable, recursive, or nested data types. DuckDB prefers the side with a join in its subtree to be the probe side. The reason is that the tuples from the subtree will already have been built or are on the flight, making [re]building faster. If the left subtree has a join, then DuckDB penalizes the right subtree by increasing the cost by 15%.

2.4.2 Limitations in Cardinality Estimation

Distinct Counts DuckDB performs HLL for distinct count estimates on 10% sample of a column. Since DuckDB v1.1, it has a new implementation of HLL, which is more memory efficient but less accurate[1][2]. With the new implementation, the sampling ratio of integer columns is increased to 30% to compensate for the inaccuracy. However, as discovered in [37], random sampling incurs significant error for distinct count estimates on at least some data distributions unless it is for a large fraction of data. Therefore, DuckDB's distinct count is prone to producing large errors for some data.

Joins DuckDB v1.0 only supports the reordering of inner joins. In their latest v1.1 release, DuckDB also supports reordering left semi- and anti-joins. DuckDB has logic for estimating cardinalities of non-reorderable join types. For right semi-joins and anti-joins, the cardinalities are estimated as the cardinalities of their right child. The cardinalities of other join types are estimated as the max cardinality among their children.

Filters If a filter is not an equality comparison or distinct count statistics of associated columns are unavailable, DuckDB applies the filter selectivity of 20%.

[1] `https://github.com/duckdb/duckdb/pull/12355`
[2] `https://github.com/duckdb/duckdb/pull/13489`

Other Operators By default, other operators pick the maximum cardinality among their children as their cardinality. If distinct count statistics are available for an aggregate operation, the cardinality is estimated as the highest among the grouping attributes. This estimation assumes the correlation among grouping attributes, *e.g.,* car's make and model. Therefore, the most selective column's distinct count (*e.g.,* car's model) will be used to estimate the distinct count of the aggregation. This is a more conservative approach to how other database systems estimate the aggregate counts. Many database systems assume independence of the grouping attributes, where they estimate the aggregate counts as the product of distinct counts of the attributes. If distinct count statistics are unavailable, the aggregate count is set to be half of its child's cardinality.

Loss of Cardinality Information When statistics are extracted during the join order optimization, the cardinality of a reorderable block is set at the head. The internal operators are not guaranteed to have their respective cardinality information. Moreover, subsequent rules following join order optimization do not rely on the cardinalities of operators. Therefore, the cardinality information is often lost during or after join order optimization. The following are the cases where cardinality information is lost.

- Join order optimization:
    - As mentioned above, extracted statistics of `op` are propagated to the head (`input_op`), but not set at the `op` itself (see *relation_ manager.cpp*).
    - Does not initialize the cardinality of mark joins. Mark joins are special joins introduced to handle nested sub-queries in the seminal paper on joins in Hyper system [38]. They create an attribute that marks whether the tuple has a join partner or not. It is useful when there is a subquery with quantified comparisons (*e.g.,* `EXISTS`). (see *relation_ statistics_ helper.cpp*).
    - When the optimal join order is selected, the cardinality of the head of the reorderable block is set with the respective entry found in the *DP Table*. When a reorderable block is of form σ(`GET`), the entry's cardinality is pushed all the way down to the `GET`, prematurely reducing the cardinality of the base table (see *query_ graph_ manager.cpp*).
- Statistics propagation (*statistics_ propagator.cpp*):
    - Updates statistics based on operator information. For instance, if there is a filter $A.w \leq 100$, we can update the *max* statistics of the column to 100.

  - Refines filter or join using available statistics. Say there is a join between relations A and B on $A.w = B.x$ and *min/max* statistics ($1 \leq A.w \leq 100$ and $10 \leq B.x \leq 200$) are available. Then, a stricter range condition is introduced by inserting filters below the join operator. Adding filters triggers a new round of filter pushdown (see *propagate_join.cpp*). When a filter is placed on top of an inner join, DuckDB constructs a Cartesian product from the children and reconstructs a new inner join with updated join conditions (see *pushdown_-inner_join.cpp*). If updating the join conditions is impossible, a new filter operator will be created on top of the join. The cardinality information is lost in the process, and the new inner join operator will not have its cardinality initialized.

    Moreover, if filter expressions are always true, the filter is completely erased; if they are always false, the filter is replaced with an empty intermediate result. Similarly, if join predicates are always true, the join is converted into a Cartesian product (without initializing its cardinality); if they are always false, the join is replaced with an empty intermediate result.

  - Applies compressed materialization, which creates "sandwich" `projection`s around operators whose outputs are to be materialized. The cardinalities of new `projection`s are not initialized (see *compressed_materialization.cpp*).

- Unused columns removal: Removes unused columns by inserting a new `projection` operator with uninitialized cardinality (see *remove_unused_columns.cpp*)

- Common Sub-expression extraction: Removes common sub-expressions from its children and inserts a new `projection` operator on top with the extracted expression and uninitialized cardinality (see *cse_optimizer.cpp*

We ran TPC-H and TPC-DS queries to gauge the amount of missing cardinality information. Table 2.1 shows the types and numbers of operators with unset cardinality from the benchmarks. Incomplete queries refer to the queries that have at least one operator with unset cardinality. The percentages of incomplete queries are 86.36% and 85.96% for TPC-H and TPC-DS, respectively. When excluding projections, which do not alter the cardinalities of intermediate results, they decrease to 36.36% and 89.90%, which still represent a significant portion of the benchmarks.

	TPC-H	TPC-DS
Projection	80	398
Top N	5	80
Filter	0	27
Aggregate	1	17
Window	0	11
Inner Join	2	11
Mark Join	4	8
CTE Scan	-	13
Disticnt	-	1
# Total operators	357	3029
# Missing operators	92	566
% incomplete queries	86.36%	95.96%
% incomplete queries (excluding projections)	36.36%	89.90%

Table 2.1: Statistics on operators missing cardinality information in queries from TPC-H and TPC-DS

2.5 Architectural Models in Distributed Database Systems

A distributed database is a collection of logically interrelated databases that may reside in different physical locations. A distributed DBMS (DDBMS) is a software system that manages a distributed database and provides users with the experience of having a single logical database despite the physical distribution of data [2, p. 2]. Data distribution provides scalability, availability, and reliability that are challenging to achieve with centralized systems [2, pp. 7–13]. However, the distributed nature comes with challenges in managing data consistency across sites, concurrent data access, executing complex operations over multiple sites, etc. [2, pp. 13–17].

This section introduces four major architectural models of distributed DBMSs. They can be categorized by varying levels of autonomy, distribution, and heterogeneity (see Figure 2.8).

Client-Server In a client-server architecture, the database system is divided into two components: server and client. The server hosts DBMS(s), which process queries invoked by the client. The client contains the application layer, which serves as an interface to the server. In traditional client-server models, entire query processing, transaction, and storage management take place on the server [2, p. 20]. Users submit queries on the client machine,

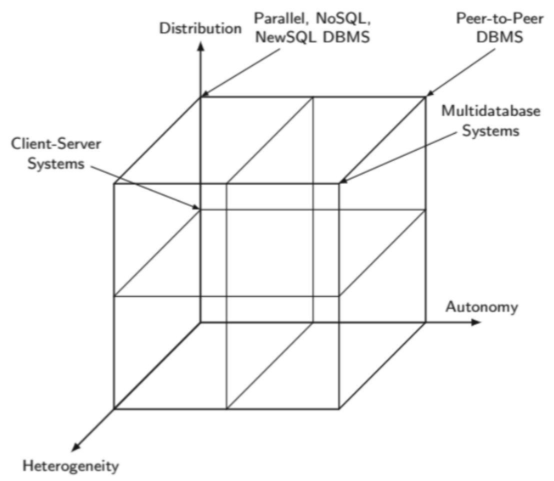


Figure 2.8: Dimensions in distributed DBMS architectures. Different architectural models can be characterized by their level of autonomy, distribution, and heterogeneity [2, pp. 18].

which are sent to the server for processing. Results generated by the server are returned and displayed on the client. This model is characterized by the *thin* client, as the responsibilities of the clients are simple, and only minimal computing power is required. With the ever-increasing capabilities of client machines (*e.g.,* personal laptops), *thick* clients emerged as an alternative. They partake in more responsibilities and, depending on their design, can perform a large portion of query processing.

Peer-2-Peer Peer-2-Peer (P2P) is an architecture where there is no distinction of functionalities between each site (clients or servers). A centralized view of data (global conceptual schema or GCS) is partitioned among the peers, where they are stored as local conceptual schema (LCS). A global query gets translated into multiple local queries, which are processed at different sites [2, pp. 22–24].

Multi-database Distributed multi-database systems consist of individual DBMSs that are fully autonomous and potentially unaware of other DBMSs [2, pp. 25–27]. Each site has fully functioning DBMSs that may be heterogeneous in terms of data model, schema, query language, and more. To integrate the individual DBMSs and provide a unified view to the users, a layer of software runs on top of the DBMSs. The most popular implementation of the layer is *mediator/wrapper* (or *middleware*). Mediators are in charge of providing a global view of databases (global conceptual schema) and coordinating the processing of user queries. Wrappers map DBMSs to the mediators' view. For instance, mapping relational DBMS to object-oriented mediator. The exact roles of mediators and wrappers may vary for different implementations.

2.6 Hybrid Query Execution in MotherDuck

MotherDuck is a DuckDB extension that enables *hybrid query processing*, which allows queries to run partly on the cloud and partly on the client [35]. This section describes the architecture of MotherDuck and the modifications made on DuckDB to power hybrid query execution.

2.6.1 Architecture

Unlike traditional client-server architecture, MotherDuck has a thick client (see Section 2.5). The client has a local DuckDB instance, fully equipped to process queries. On top of that, the MotherDuck extension provides a server that hosts another DuckDB instance (called *duckling*) on the cloud. This allows users to create and share DuckDB databases in the cloud and query cloud-hosted data [35]. Unlike many client-server models, which store all data primarily on the server, MotherDuck clients can have local databases or files. Queries are executed locally, remotely, or in a hybrid fashion, depending on data placement. If a query involves data from both client and server, then some part of the query is executed locally and others remotely.

2.6.2 Adaptations from DuckDB

Here, we outline the adaptations made on top of DuckDB to support MotherDuck's hybrid query execution.

Parsing MotherDuck extends DuckDB's parser to support table functions to read from Parquet, JSON, and CSV files. Also, sharing of remote databases is made possible by introducing `CREATE SHARE` and `ATTACH <share url>` statements.

Binding During the binding phase, catalog information resolves parsed names to database objects. MotherDuck's remote catalog (or *virtual catalog*) is built based on DuckDB's extensible catalog and provides visibility on remote database objects to the local DuckDB instance. Queries that modify the remote catalog (*e.g.,* `CREATE DATABASE remote_db`) are run fully on the server. The server notifies the client of the changes, who is long-polling on a background process for remote catalog updates. Remote catalogs contain information

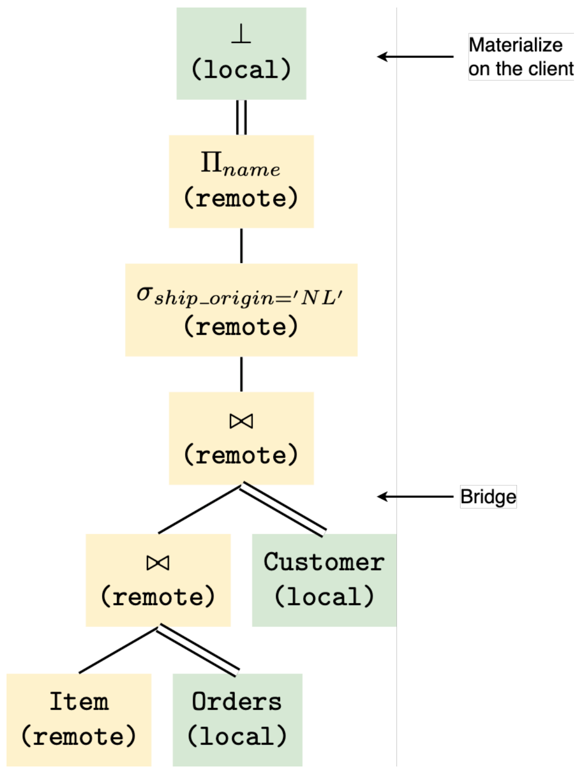


Figure 2.9: MotherDuck-optimized logical query plan of Listing 2.1. Operators are marked as `local` and `remote`.

on database locations and statistics, which are used during the subsequent planning and optimizing phases.

Planning and Optimizing Optimizations from extension modules are run at the end of the optimization phase after all DuckDB optimization rules are run. MotherDuck's optimizer has to plan where to execute the logical operators in a given query plan. Using the catalog information, the optimizer marks the locations of database objects in the query; then, it traverses the query tree to determine the execution sites of each operator. The optimizer employs a heuristic which aims to choose a site that incurs the least data transfer cost. Moreover, MotherDuck has to decide where to materialize the final results, noted with the ⊥ symbol. `SELECT` statements (Data Query Language (DQL)) must display the results on the client side. Data Definition Language (DDL) or Data Manipulation Language (DML) depend on the location of the involved database object. A new logical operator called *bridge* is introduced to handle data transfer between the client and server. Bridges are inserted between the operators with different execution sites to transfer the output from the child operation to the parent. MotherDuck's optimization logic is described in more detail in Chapter 4.

Figure 2.9 shows the MotherDuck-optimized logical query plan. Bridge operators are inserted when a transition from client to server is required. Also, the result of the query must be returned to the client. We do this by adding a bridge from the final remote projection (Π) to local materialization ($\perp$).

Execution Bridge operators come in pairs – a source and a sink. A bridge source is a blocking operator, which completes the pipeline and transfers the data to its matching bridge sink, which is the starting operator of the subsequent pipeline.

Summary Hybrid query execution allows running parts of a query locally and other parts remotely. It powers efficient reading from remote data sources, creating and sharing remote databases. Adaptations were made on DuckDB to support the hybrid query execution, including remote catalog and bridge operators. Hybrid execution adds a new dimension to plan for – the execution site. This decision is made in MotherDuck's optimizer, which is applied as a rule at the end of DuckDB's optimization phase. MotherDuck's optimizer is discussed in detail in the following section.

3

Literature Review

This chapter reviews relevant literature on hybrid query processing and cost definitions in distributed DBMSs. First, we list examples of implementations of hybrid query processing in client-server and widely distributed systems. Next, we study how the cost is defined in distributed DBMSs and present different optimization objectives and the factors considered in cost models.

3.1 Hybrid Query Processing

Hybrid query processing can be characterized by its flexibility in choosing the site of execution. It allows operations in a query plan to be executed at different sites based on criteria like data placement, the query processing capability of sites, workload, etc. We list examples of hybrid query processing implementation in different architectural models. For each implementation, we identify:

1. what is being shipped across sites (*e.g.,* data, query, code, etc.)

2. site-selection mechanism (*i.e.,* how it determines the execution sites for operations)

3.1.1 Adoptions in Client-Server Systems

Here, we describe different hybrid query processing policies found in the client-server architecture. The concepts behind these policies are also used for hybrid query processing in other architectural models, which will be covered in the upcoming section.

Hybrid Shipping One of the first proposals of hybrid query processing is by Franklin *et al.* in [39]. Previously, the two predominant paradigms of the client-server model were data shipping and query shipping. **Data shipping**, generally used in object-oriented DBMSs (ODBMSs), moves data from the server to the client, where it is potentially cached. A query is entirely processed on the client, where the client resources (CPU, memory, and disk) are exploited. In contrast, **query shipping** processes a query on the server, sending only the result back to the client. Franklin *et al.* present a new model, **hybrid shipping**, which dynamically decides where to execute individual operations in a query. It combines the advantages of data shipping and query shipping by providing the flexibility to choose the *site* based on client and server conditions (*e.g.,* server load, caching capacity, and memory allocation).

The performance of hybrid shipping is evaluated in a simulated environment with varying query complexity, caching capacity, server load, etc. The experiments show that neither data shipping nor query shipping wins all scenarios. Data shipping reduces the load on the server but tends to incur high communication costs. Query shipping has low communication costs but neglects client resources. Hybrid shipping makes flexible use of client resources and caches, performing at least as well as the best out of the two "pure" approaches and outperforming them in most cases.

The authors also note that hybrid shipping requires more complex query optimization due to the extra decision variable, the operator site. They propose a two-step optimization where an incomplete query plan (including join ordering) is generated at compile time. The site selection occurs during runtime when the optimizer can consider the latest system state. The site selection works in two stages: 1) generation of a randomized plan and 2) iterative application of transformation rule and simulated annealing. The optimizer selects a plan that minimizes the response time, which is estimated based on data volume and client/server conditions.

Enhanced Client-Server (Transaction Shipping) Another early idea of hybrid query processing, **enhanced client-server (ECS)**, is found in [40] and evaluated in [41] against existing client-server DBMS configurations.

The client-server architecture here is defined as a model where all database operations are done on the server (like query shipping above). Another model the paper compares is RAD-UNIFY (RU) [42]. In RU, query processing is done on the client, while the server maintains low-level DBMS operations and data manager. Data is sent to the client cache for planning and execution. With the client-side cache, RU performs better than the traditional client-server model when it comes to small/medium retrieval operations. Yet, with heavy updates (I/O), the data manager on the server is overloaded and becomes a bottleneck.

Enhanced client-server (ECS) extends traditional client-server by incorporating client-side caching and incremental updates. The cached query results form a local database, a partial replica of the server-side database, preventing repeated server access for the same data. ECS can be classified as a **transaction shipping** model, where transactions are shipped to the location where data is. When a client submits a query, the server initially processes the query if it involves data that is not cached on the client. If the relevant data is cached on the client, the client performs query processing. This model relieves server load by delegating both CPU and I/O load to clients. However, in case of updates, a propagation method (incremental update) is required to keep local and remote databases in sync. From experiments with simulated workloads and networks, ECS proved to have significant performance improvements over traditional client-server or RU systems, especially under light update rates.

Load-Sharing Real-Time Database System Kanitkar and Delis introduce a *hybrid* load-sharing algorithm that performs either transaction shipping or data shipping (or both) to address the potential delays of transaction shipping under large updates [43]. The algorithm is implemented in their load-sharing client-server real-time database system (LS-CS-RTDBS). It selects the transaction site based on two heuristics: H_1) processing load and H_2) data availability of the site. A site is preferable if the estimated queuing time of a new transaction is lower (H_1) and if it is waiting for fewer conflicting locks on data (H_2). If a site other than the query's origin is dimmed more preferable, the transaction is shipped to the new site. Then, if needed, relevant data is shipped to the chosen site. The experiments show that LS-CS-RTDBS completes more transactions in a given time (*i.e.,* higher throughput) and transfers less data than the traditional client-server system.

KRISYS KRISYS is a research prototype of an object-oriented database system built in a client-server architecture [3]. It adopts a hybrid query processing approach, which dynamically determines whether to execute query fragments on the client or delegate them to the server. The architecture is displayed Figure 3.1.

The client has a query processing engine called KOALA Processing System, which is responsible for generating execution plans and has the capability to execute them. During runtime, it interacts with the Context Manager to determine which parts of the query can be executed locally. The Context Manager consults the Working Memory, a buffer storing

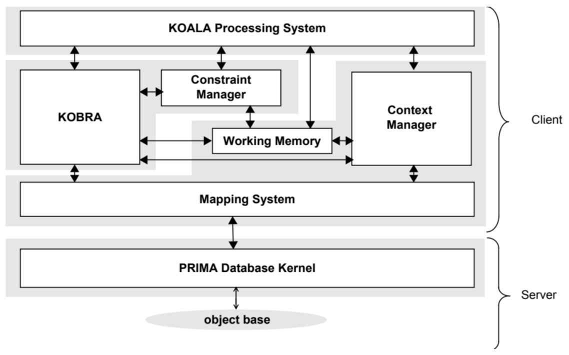


Figure 3.1: Architecture of KRISYS [3].

cached objects from previous queries or data fetched from the server. If necessary data is already available in the Working Memory, the client can carry out relevant operations. Otherwise, the operations may be forwarded to the server. If the server is not equipped to process the operations (*e.g.,* complex joins or aggregates), the required data will be transferred to the client for processing.

KRISYS' hybrid query processing resembles hybrid shipping. It performs dynamic site selection depending on the client's runtime context. By utilizing the already available data on the client and delegating parts of execution to the server, KRISYS reduces the data transfer across the client-server boundary.

Predator Mayr and Seshadri introduce an efficient way to execute client-site user-defined functions (UDFs) in Cornell Predator, a client-server database system [44]. They argue that existing approaches for server-site UDFs are not efficient for client-site UDFs due to network latency, security concerns, and unavailability of client-specific data to the server. To address this, they propose executing the client-site UDFs directly on the client. These UDFs are modeled as joins and executed as either semi-joins or client-site joins. The join types differ in the amount of data transferred from the server to the client and the client to the server. Moreover, Predator factors network asymmetry into account, allowing it to choose a join implementation that minimizes network cost.

Predator's query processing model is similar to hybrid shipping. To process client-site UDFs, data is shipped to the client, where the UDF results are computed. This temporarily ships the query processing task to the client as well. The task is given back to the server along with the UDF results for executing non-client-site operations.

MotherDuck As described in Section 2.6, MotherDuck is a cloud-based analytics database with a client-server architecture. It has a full-fledged DBMS instance (DuckDB) on the client, which has the capability to store data and process queries, unlike traditional client-server database systems.

MotherDuck employs hybrid query execution to utilize client-side resources. During optimization, the execution sites are determined under two concepts. 1) Queries are processed close to data. That is, the optimizer will prefer to execute a sequence of operations at the data source site for as long as possible. For instance, operations involving only client data will continue on the client side. 2) A site with a larger estimated relation is preferred. The motivation for the decision is to minimize the network costs. When an operation involves both the server and client data, MotherDuck favors executing the operation at the site where a larger relation resides. This heuristic is based on the assumption that DuckDB's join order optimizer places a relation with a larger cardinality on the right (which is often not true). Hence, MotherDuck always transfers data from the right-hand side to the left-hand side. While MotherDuck optimizer aims to minimize the number of tuples transferred over the network, due to the greedy nature of this approach, the resulting execution plan may not be optimal.

3.1.2 Adoptions in Widely Distributed Systems

This section lists implementations of hybrid query processing in widely distributed systems, mostly as mediator/wrapper models in multi-database architecture.

Garlic Garlic is a middleware system that provides an integrated view of heterogeneous data sources [45]. Each data source, or *repository*, has a wrapper that hides the details of underlying DBMS and models the data in Garlic's object-oriented data model.

In Garlic, wrappers participate in query planning to complement the central query processor, which has no knowledge about the query processing capabilities of individual data sources [46]. During the planning phase, the query processor identifies a query fragment relevant to a particular data source and forwards it to the wrapper. The wrapper then determines how much of the query fragment it can process and returns a set of query plans that implement it. The query processor continues the planning, incorporating the returned plans. It adds operators to fill the "gaps" that the repository is not able to handle after all. During execution, the wrapper executes the fragment it agreed to process. If there are operators that are not processed by the wrappers (the gaps), relevant data is transferred to the central query execution engine, and the operators are executed there.

The Garlic model is a variant of hybrid shipping; both query and data can be shipped based on the processing capability of data sources. Query fragments are executed locally in individual data sources as much as their processing capabilities allow. The operations that are not supported locally are executed in the central query execution engine, to which relevant data is shipped.

MOCHA (Code Shipping) MOCHA is a middleware system designed to interconnect a large number of heterogeneous DBMSs [47]. It has a self-extensible architecture where user-defined functionality, such as custom query operators and data types, are deployed *automatically* and *dynamically*. This is realized by **code shipping** Java component to the site that requires the functionality and migrating the code from site to site on demand.

Moreover, code shipping provides more flexibility in operator site selection. In other hybrid processing policies (and, by extension, traditional processing models), execution sites for an operation containing user-defined functions are limited to those where the functions are already installed. For instance, MotherDuck's remote functions can only be executed on the server (later discussed in Section 4.1.1). Similarly, in Predator, client-site UDFs are confined to the client. Also, a wrapper in the Garlic system will refuse to carry out parts of a query fragment if it includes some unimplemented functions. These restrictions may hinder the optimizer from selecting a query plan with better execution sites simply because the function is absent. While some middleware systems (*e.g.,* TSIMMIS [48] and DISCO [49]) allow manual installations to deploy the function to *all* sites, it does not scale in large-scale systems.

Figure 3.2 illustrates the architecture of MOCHA. The query processing coordinator (QPC) manages query processing logic, including parsing, optimizing, scheduling, and execution. The data access provider (DAP) is a wrapper that maps remote site DBMSs to QPC. It can also load and execute user-defined functions shipped in Java code.

Code shipping enables site selection for efficient query processing; *data-reducing* operations are pushed close to the data source (DAP), and *data-inflating* operations close to the client (QPC). This allows to minimize the movement of data over the network. MOCHA employs an DP approach in R* [24], but enhanced with a new cost metric, volume reduction factor (VRF), defined as:

$$VRF(\Omega) = \frac{VDT}{VDA},$$

where Ω is an operator, VDA the size of input relation, and VDT the size of the output. Operator Ω is data-reducing if $VRF < 1$ and data-inflating otherwise. The total cost of a plan is a cumulative sum of VRF of all operators.

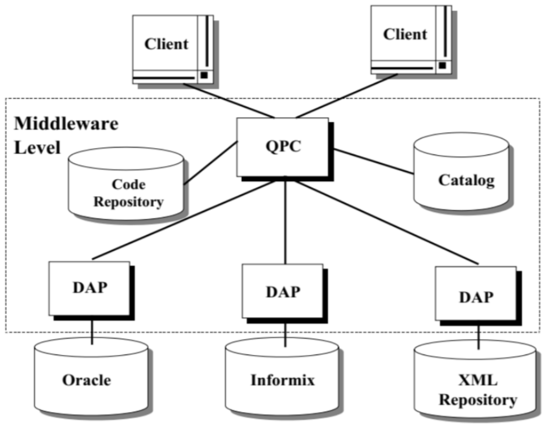


Figure 3.2: MOCHA architecture. QPC is the query processing coordinator, which manages the entire query processing. DAP is the data access provider, a wrapper that provides QPC with uniform access to remote sites. Additionally, it can load and execute user-defined functions.

Mutant Query Plans (Combined Shipping) In their work on mutant query plans (MQPs), Papadimos and Maier introduce an XML representation of query plans that adapt dynamically to data availability and server workload [50]. It is designed specifically to address challenges in querying internet resources, where servers do not have a complete view of the catalog or access to all data required to process a given query.

MQP reduces network costs by processing operators near the data source. That way, it can potentially transfer a reduced amount of data among the servers. MQPs are transferred as XML documents, and servers parse them into query operators. Based on the local catalog, a server resolves URNs and decides how much of the plan it can evaluate locally. The optimizer (re)optimizes the plan and forms sub-plans to be evaluated on the server. The policy manager then determines whether to accept or reject the sub-plans. It may reject the plan if the server has a high workload or the mutant plan's cost is too high. The evaluated sub-plans are substituted with their results, formatted in XML. Hence, a new *mutant* query is created. If the plan is not fully evaluated, it is passed on to another server that can resolve at least one URN.

Note that MQPs embed data; they are simultaneously queries and data. As authors call it, this concept can be thought of as **combined shipping**, which merges query shipping and data shipping into one.

Distributed ORION Distributed ORION (referred to as ORION from here) is an object-oriented database designed for distributed environments [51]. It employs a dataflow execution model that decomposes a query into subqueries that are executed concurrently at multiple sites. Data are distributed across the sites, so executions are synchronized by data exchange among the participating sites.

ORION represents each query as a query graph where nodes represent classes, attributes, predicates, or logical connectives (`AND`, `OR`, `NOT`). During the optimization phase, the query graph is split into clusters of nodes. ORION provides flexibility to choose *traversal method* and *execution site* of each cluster that minimize the cost.

Three traversal methods are available: forward traversal, reverse traversal, and mixed traversal. The forward traversal processes nodes in a top-down manner, sending identifiers (UIDs) from parent nodes to their children. It requires a second phase where UIDs of qualified child instances are percolated up to the parents; the child UIDs are used to filter the parent node's instances. In contrast, the reverse traversal begins at the leaf nodes and proceeds to the root. The mixed traversal combines the two methods, where some parts are forward traversed and some reverse traversed.

The optimizer also evaluates three site-selection strategies. First is the parent-site/children-site approach. This strategy directs data flows according to the traversal direction: parent-site for reverse traversal and child-site for forward traversal. Here, data from each resident site of the source node are sent to each resident site of the destination node. Therefore, it requires $p \times q$ messages, where p and q are site counts of the source and destination node, respectively. The second strategy is the originating-site approach. Subquery results of each node are always sent to the query's originating site, where the partial results are merged and forwarded to the resident sites of the next node (parent or child). This reduces the required messages to $p+q$. The last strategy, the largest-site approach, selects the site with the largest estimated data in a cluster. This strategy minimizes inter-site communication by avoiding transferring large volumes of data to multiple sites.

Finally, the optimizer evaluates plans by local processing and communication costs. The local processing cost involves the CPU and I/O costs, and the communication cost includes CPU time and communication delay. ORION then selects the plan with the lowest estimated response time.

ORION's query processing resembles hybrid shipping. It allows flexible site-selection balancing processing and network costs. Parent-site/child-site and originating-site strategies can be considered as data shipping. Data from the source node's resident sites are forwarded to the site(s) where execution takes place. On the other hand, the largest-site strategy aligns more with query shipping. While the largest site does receive small volumes of data from other sites, a significant portion of relevant data is already available.

Therefore, this strategy can be considered delegating query processing task, hence, query shipping.

3.1.3 Summary

We have reviewed various database systems and their implementation of hybrid query processing. Two key aspects of hybrid query processing–what is being shipped and site selection criteria–are for each system listed in Table 3.1.

	What is shipped	Site selection criteria
Hybrid Shipping	data, query	server load, data placement, memory allocation
Enhanced Client-Server	transaction	data placement
LS-CS-RTDBS	transaction, data	server load, data placement
KRISYS	data, query	data placement, operation complexity
Predator	data, query	client-site UDF
MotherDuck	data, query	data placement, data sizes
Garlic	data, query	data placement, processing capability
MOCHA	code	volume reduction factor
Mutant Query Plans	combined data+query	server load, data placement
ORION*	data, query	local processing costs, communication costs

Table 3.1: Comparison of hybrid query processing in different systems. (*ORION employs local processing costs and communication costs to determine the specific site selection strategy (and traversal method) for each cluster. The available site selection strategies are: parent-site/child-site, originating-site, and largest-site approaches.)

3.2 Defining Cost in Distributed DBMSs

The cost model directs the optimizer in selecting a plan from a search space. They play a critical role in identifying a plan that maximizes the objective of the optimizer. In the following section, we explore different optimization objectives and factors considered in cost models.

3.2.1 Optimization Objectives

Optimization objectives define how cost models are constructed in database systems. Cost models evaluate the quality of plans by quantifying how well they meet the specific optimization objectives. Total time and response time are the most widely used objectives in database systems. **Total time** (or total cost) is the sum of all time components (*e.g.,* processing and network), and **response time** is the elapsed time between the query initiation and completion [2, p. 157].

Total time is the objective employed in the R* system [24, 52]. R* is among the first systems to introduce a detailed cost model, considering local processing and communication costs. The cost function for total time follows the form:

$$\textit{total time} = T_{CPU} \cdot \#insts + T_{I/O} \cdot \#I/Os + T_{MSG} \cdot \#msgs + T_{TR} \cdot \#bytes$$

The first two terms compute the local processing time. T_{CPU} is the time per a CPU instruction, $T_{I/O}$ is the time of a disk I/O. The next two terms measure the communication cost. T_{MSG} is the latency, the time it takes to send and receive a message, and T_{TR} is the time it takes to transfer a data unit.

The coefficients can be weighted to reflect machine capacity or network conditions. If the machine is heavily loaded, higher weights could be assigned to T_{CPU} and $T_{I/O}$. In distributed systems, networks usually become the bottleneck [2, p. 158]. Therefore, many early database cost models neglected local costs and focused on minimizing the network costs.

To evaluate the response time, the parallelization of local processing and communication must be considered. Sequential executions are usually the dominant factor. Therefore, the maximal number of sequential operations of each component (CPU instructions, disk I/O, messages, and bytes) are factored into the cost function.

$$\begin{aligned} \textit{response time} = {} & T_{CPU} \cdot \#seq\_insts + T_{I/O} \cdot \#seq\_I/O \\ & + T_{MSG} \cdot \#seq\_msgs + T_{TR} \cdot \#seq\_bytes \end{aligned}$$

Lowering response time can be achieved through increased parallelization. However, this does not necessarily reduce the total time. In fact, the total may increase because of a higher number of parallel local processing and network communication. Systems like Distributed INGRES handle this trade-off with a dynamic approach, where the objective is to minimize a weighted combination of total time and response time [53].

An alternative optimization objective is monetary cost. With the growing popularity of cloud-based databases, the monetary cost has become another important component in the cost model. While the two cost functions above calculate the cost in terms of time, the cost can be converted into monetary expenses using the rates set by cloud service providers for computation, data storage, and data egress. Cost models in the cloud environment will be discussed in Section 3.2.2.2.

3.2.2 Cost Factors

Here, we outline various factors that contribute to costs or are considered by the cost model.

3.2.2.1 Network

Network cost is one of the most critical factors in any distributed system. In [52], Mackert and Lohman compare the relative importance of the four components of the cost model in R*: CPU, I/O, messages, and bytes. The results in a simulated network environment show that the importance of network costs is marginal compared to local processing costs under a high-speed network. The relative network costs increased in a medium-speed network, although local processing remained a significant contributor to total time.

Johansson, in their paper "On the impact of network latency on distributed systems design", focuses on the **impact of latency in distributed databases**[54]. They evaluated two cost models: 1) model proposed in [55], which does not consider network latency and 2) an enhanced version of 1) with added network latency cost. The experiments were conducted under varying network speeds:

1. 56 Kbps (representative of WANs of the 1980s to early 1990s)
2. 1.544 Mbps, T1 class network (representative of WANs of the mid-to-late 1990s)
3. 44.736 Mbps, T3 class network (representative of high-speed networks in the late 1990s)

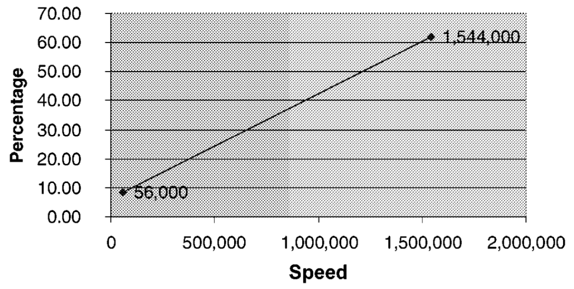


Figure 3.3: Latency relative to the total network response time against the network speed (bps).

The results show that in a 56 Kbps network, transmit time still overwhelms queuing delay or latency. However, in T1 and T3 networks, the transmit time decreases significantly, and latency becomes the dominant factor. Figure 3.3 illustrates that the latency takes up over 60% of the total network response time with 1.544 Mbps (T1) as opposed to 10% with

56 Kbps network. The results suggest that the faster the network is, the more influence latency has. As stated in [56], "at high speeds, we are latency limited, not bandwidth limited."

More recently, as widely distributed databases came into play, network topology among participating sites became influential in query processing performance. Ahmad and Çetintemel propose a **network-aware query processing**, which considers underlying network characteristics in their cost model [57]. They propose three models that reflect the network characteristics in the operator placement algorithm.

1. Edge: only considers the source and proxy sites
2. Edge+: Edge, but considers the network distance between the sites (*i.e.,* latency)
3. In-Network: Edge+, but considers a larger set of candidate sites

Say there are two operators m and n that are mapped to site $\lambda(m)$ and $\lambda(n)$, respectively. Then, the cost of processing the edge between m and n in Edge is:

$$c(m,n) = \begin{cases} 0 & \text{if for } \lambda(m) = \lambda(n) \\ \beta(m,n) & \text{otherwise,} \end{cases}$$

where $\beta(m,n)$ is the product of bandwidths and selectivity of the input operators. In Edge+, the cost function changes to factor in the distance d between the two operator sites:

$$c(m,n) = \begin{cases} 0 & \text{if for } \lambda(m) = \lambda(n) \\ \beta(m,n) \cdot d(\lambda(m), \lambda(n)) & \text{otherwise,} \end{cases}$$

The cost $c(m,n)$ increases as the distance (or latency) between the two sites increases. The In-Network placement uses the same cost function as Edge+. The difference is that it considers promising sites other than source or proxy, allowing the algorithm to dynamically choose sites based on network topology. The experiments show that Edge+ and In-Network placement outperformed Edge in a high-delay constraint environment under uniform, cluster, and star network topologies. Also, as the average distance grows between the servers, the bandwidth consumption of Edge+ and In-Network placement increases. This result aligns with the cost model, where cost increases with the distance; the algorithm reduces the cost at the expense of bandwidth consumption. The paper shows that network awareness in cost models can improve the performance of query processing.

3.2.2.2 Cloud

Cloud service providers offer various heterogeneous instance types. This raises a new challenge for cloud-native DBMSs, as the performance and monetary costs depend on hardware configuration. As [58] argues, the trade-offs between the cost and performance can present unpleasant surprises to the customers; *e.g.,* lower hourly rate instances cost more than expensive ones. Leis and Kuschewski propose a **model-based approach** to estimate the cost of a particular workload on some specific hardware setup [4]. The model can guide how to best configure instances, taking workload, hardware, and billing into account. The ultimate goal is to bring the current system close to a theoretical limit of processing throughput per dollar, as shown in Figure 3.4.

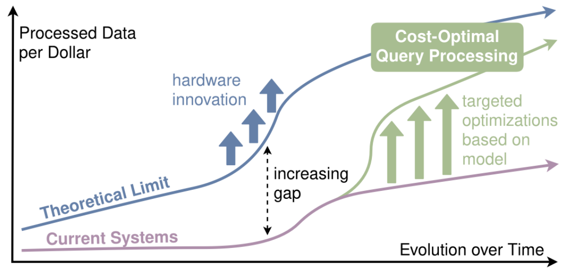

Processed Data per Dollar
hardware innovation
Cost-Optimal Query Processing
targeted optimizations based on model
increasing gap
Theoretical Limit
Current Systems
Evolution over Time

Figure 3.4: Outlook of model-based hardware configuration in the cloud [4].

They propose three models, incrementally adding extra decision variables:

1. M1: CPU hours + scanned data.
2. M2: M1 + data caching
3. M3: M2 + scalability fraction (how much of the workload can be scaled)

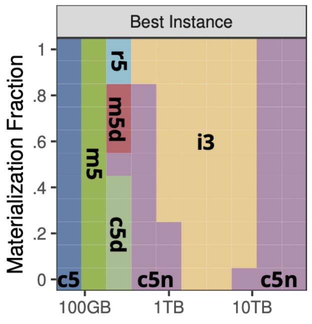


Figure 3.5: Best instance type for varying materialization fraction and scanned data [4].

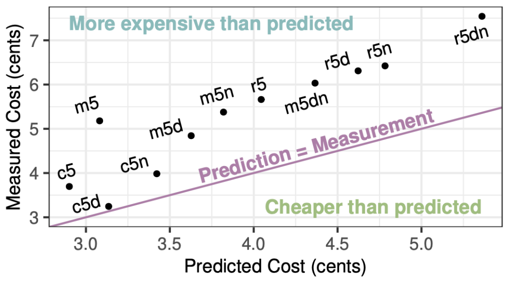


Figure 3.6: Measure cost vs prediction [4].

The experiments are carried out in AWS using various instance types. Evaluation with M2 proved that optimal instance type heavily depends on workload; there were clear winning instance types for different materialization fractions (how much of the scanned data has to materialize) and volumes of scanned data (see Figure 3.5). In the comparison between M3 prediction and measured cost,

With the increasing use of cloud-based databases, optimizing for cloud costs has become a key concern. To address this, some systems are designed to explicitly balance monetary cost with execution time. Karampaglis *et al.* propose a **bi-objective approach** in a cloud environment in [5], combining both total execution time and monetary cost. This model guides the allocation of query fragments to cloud instances. The total time on a particular cloud instance is estimated by summing the processing times of operators (based on [59]) and transfer times of data (based on [60]). The monetary cost is estimated using knowledge of the cloud service provider's charging policy.

For each query fragment, the model aims to maximize *user satisfaction*. The authors assume that each user provides their own function of worst acceptable trade-off. Figure 3.7 illustrates the cost vs time curves of 1) cloud instances and 2) user-defined threshold. An instance that creates the maximal difference between the two curves is selected as the optimal execution site for the fragment.

While the paper establishes a framework for the bi-objective cost model, it lacks experimental results to demonstrate its effectiveness. The impact and efficiency of this approach in balancing monetary cost and performance remain to be tested.

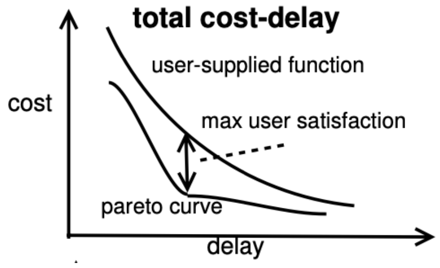


Figure 3.7: Cost vs time curves of 1) cloud instances and 2) user-defined function. The point with maximal difference generates max user satisfaction [5].

3.2.2.3 Cache

Distributed database systems can obtain performance gains through dynamic caching. However, a complication arises when dynamic caching is combined with flexible site selection (*e.g.,* hybrid shipping). A circular dependency emerges: cache is a byproduct of operator placement, but the optimizer selects operator sites based on data and cache location. Hence, the optimizer may make a decision that is not beneficial in the long run. To address this problem, Kossmann *et al.* propose **cache investment**, a novel approach to integrating query optimization and data caching for data placement that benefits performance in a long-term [6]. Cache Investment may intentionally choose a suboptimal plan for better data placement for the subsequent queries.

The cache investment scheme is integrated into an existing cost-based query optimizer without disrupting its internal logic. Traditional query optimizers work as illustrated in Figure 3.8. With the cache investment, the optimizer receives an *augmented* state of available caches as in Figure 3.9. The optimizer assumes that reported items are already present in the cache.

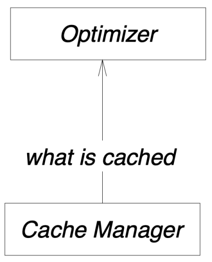


Figure 3.8: Traditional optimizer [6].

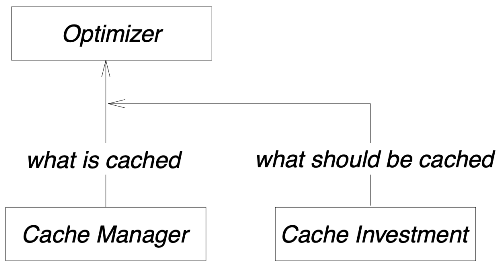


Figure 3.9: Integration with cache investment [6].

A decision to cache or not cache is a trade-off between the cost of caching and the potential benefit, quantified as *investment cost* and *ROI* (return of investment), respectively. An item is considered to be a candidate if it meets the following criteria:

1. the ROI is higher than the investment cost
2. the net saving of caching the item (ROI - investment cost) is higher than the ROI of the item(s) that would be evicted

There are two policies to determine the cache candidates: *Reference-couting* and *Profitable*. Since both policies are history-based, the *values* of data items are tracked. The value V of data item t at client c after execution of query q is:

$$V_t^c(q) = v_t^c + \alpha \cdot V_t^c(q-1),$$

where $0 \leq \alpha \leq 1$ is an aging factor that discounts the value after each query execution and $v_t^c(q)$ is a contribution of query q to the value V by using the item t. If query q does not access item t, $v_t^c(q) = 0$; otherwise, it is ≥ 0.

In Reference-counting policy, $\alpha = 0$ and $v_t^c(q) = 1$ for any query. In other words, it simply tracks the frequency of access for each data item and prioritizes items with higher counts for caching.

In contrast, the Profitable policy explicitly calculates ROI and investment costs of investment candidates. The investment cost is computed as the cost difference between a plan that brings specific data to the cache and the best execution plan for a query. The ROI is the value of item t at the time of optimizing q. The contribution of some query q to the value V is the estimated saving by having t cached. It is, $cost(executing\ q\ without\ t\ cached) - cost(executing\ q\ with\ t\ cached)$.

The experimental results show that under most types of workload, cache investment reduced the response time compared to conservative (existing caching scheme, no cache investment) or optimistic caching (brings all un-cached data to clients). There was no clear winner between the Reference-counting and Profitable policies, and the policy selection should depend on workloads, query types, and heterogeneity of servers.

History-aware query optimizer (Hawc) is developed by Perez *et al.* in [61]. Hawc is an enhanced cost-based query optimizer that utilizes history information to identify queries that could produce (intermediate) results that could benefit the subsequent query executions. Hawc maintains two additional data structures: *history pool* $\mathcal{H}$ and *view pool* $\mathcal{V}$. The history pool is a set of recent queries. For each query in the pool, a set of alternative plans generated during optimization is stored. The view pool contains a set of tables created as (intermediate) results of previous queries.

Hawc does the following in the optimization phase:

1. Query Q is optimized using views from the view pool$\mathcal{V}$ for the entire history pool $\mathcal{H}$. If some plan p for Q creates (intermediate) results that would have reduced the costs of plans in the history pool, the savings are credited to p when p is evaluated
2. The view pool is updated
3. The history pool is updated

Queries in the history pool and view pool are continuously updated. The *hypothetical query cost* for each query in the history pool is computed. The hypothetical query cost is an answer to the question, "what would have been the cost of query Q if the views in V were materialized when Q was optimized?" [61]. The `CostReplace(`V, p`)` algorithm carries out the view matching for plan p, assuming all views in V are materialized, and updates each query $h \in \mathcal{H}$. Each query h in the history pool has a set of query plans $h.P$. Then, the optimal cost for query h given view set V is:

$$cost(h) = \min_{p \in h.P} cost(\texttt{CostReplace}(V, p))$$

During the optimization of query Q, the sum of hypothetical costs of the history pool is added to the cost of plan p that is being evaluated. The history pool is first updated through view matching p's (intermediate) results with the queries. For each old query h, an updated set of query plans $h^*.P$ are generated:

$$h^*.P = \{\texttt{CostRepalce}(\mathcal{V}, p) \mid p \in h.P\}$$

Then, the final cost of plan p is:

$$\mathit{cost}(p)' = \mathit{cost}(p) + \sum_{h \in \mathcal{H}^*} h^*.w \times \mathit{cost}(h^*),$$

where $\mathit{cost}(p)$ is the base cost of p and h^* is weight assigned to each query $h^* \in \mathcal{H}^*$.

Hawc's performance was evaluated against two alternative approaches: 1) optimizer without any caching and 2) Hawc without history-awareness, where a query is not optimized for the history pool. The results demonstrated that Hawc outperformed both optimizers with speedups ranging from 1.4x to 3.5x for varying workload types.

3.2.2.4 Other factors

There are other cost factors that are noteworthy in cost models. Farnan *et al.* propose a **preference-aware optimizer** that provides a framework allowing declarative, user-defined constraints on optimizers [62]. Another growing area of interest is **energy consumption** in database systems. Given the increasing focus on environmental impact, there are many research for energy-efficient database management. For example, Dembele *et al.* conducted a comprehensive study on what is required for energy reduction in distributed DBMSs [63]. Guo *et al.* propose an *energy cost model* that balances trade-offs between energy costs and performance [64]. Zhou *et al.* developed GreenDB, an energy-efficient system that couples "hot nodes" in database clusters with "cold nodes". This coupling keeps cold nodes in low-power mode, reducing the energy-saving overheads [65].

4

MotherDuck's Query Optimizer

This chapter explains the current implementation of MotherDuck's query optimizer. We describe the heuristics on which the optimizer is based, as well as its decision-making process. Next, we point out the discrepancy between available statistics in MotherDuck and DuckDB, and how we address them and preserve the cardinality information.

4.1 Heuristic Query Optimizer

MotherDuck's optimizer is applied after DuckDB's optimizer has run. Its role is to decide where to execute operators given a DuckDB-optimized query plan – on the client (`local`) or server side (`remote`). For DDL and DML, the execution site of the plan is determined by the placement of the associated catalog entry. The query is executed locally if the entry is on the client side. Otherwise, the query is executed remotely. DQL, on the other hand, requires more intricate decision-making.

MotherDuck's current optimization for DQL operates on a simple heuristic that assumes that smaller relations are always on the right-hand side of operators. It strictly prefers to transfer the right-hand side relation to the location of the left-hand side. Therefore, MotherDuck's current approach can be described as *greedy*. MotherDuck's heuristic is based on DuckDB's preference to place a smaller relation as a build-side of joins. While this is a preferred arrangement, the assumption does not always hold (*e.g.,* in the case of non-inner joins). Furthermore, DuckDB v1.1 introduces another decision variable for deciding the build and probe side – the presence of the join operator in the child subtree (see Section 2.4.1.4). This change increases the chance that MotherDuck's assumption does not hold.

4.1.1 Site Constraints

While most operator types can be executed on the client or the server side, there are three cases where the options are restricted.

1. **Scan operators** are responsible for reading data from a file system or a table. Therefore, their execution sites are pre-determined by the placement of the data source.

2. **Operator pairs that share a memory state** must be colocated.

   - A *duplicate elimination join* (`DELIM_JOIN`) removes duplicate tuples from either the left or right side of the join. It is paired with a special scan operator, the duplicate elimination scan (`DELIM_GET`), which supports keeping track of the tuples to eliminate.
   - A *recursive CTE* (`REC_CTE`) applies a query recursively to its own result set. A CTE scan (`CTE_SCAN`) retrieves these intermediate results during the recursion process.

   These operator pairs must be executed at the same site and within the same execution pipeline. Consequently, all operators on the path between a colocated pair must also be colocated. Note that the colocated operators still have the flexibility to be executed locally or remotely; the constraint is that when a decision is made, it is applied collectively to all colocated operators. An example of colocation is shown in Figure 4.1.

3. **Remote scalar functions** can only be executed on the server. MotherDuck currently offers `embedding` function [66], which transforms text to a numeric representation in vector space and `prompt` function [67], which allows users to prompt large/small language models.

4.1.2 Decision Process

MotherDuck's heuristic optimizer makes two traversals through the query plan. The first traversal creates a linked list of colocated operators (*colocated group*). In the linked list, a *colocated child* has a reference to its immediate *colocated parent*, and the root of the colocated operators refers to itself. In the example of TPC-H query `q04`, (see Figure 4.1), the colocated group looks like: `Delim Get` $\rightarrow$ `Comparison Join` $\rightarrow$ `Projection` $\rightarrow$ `Delim Join` $\circlearrowleft$.

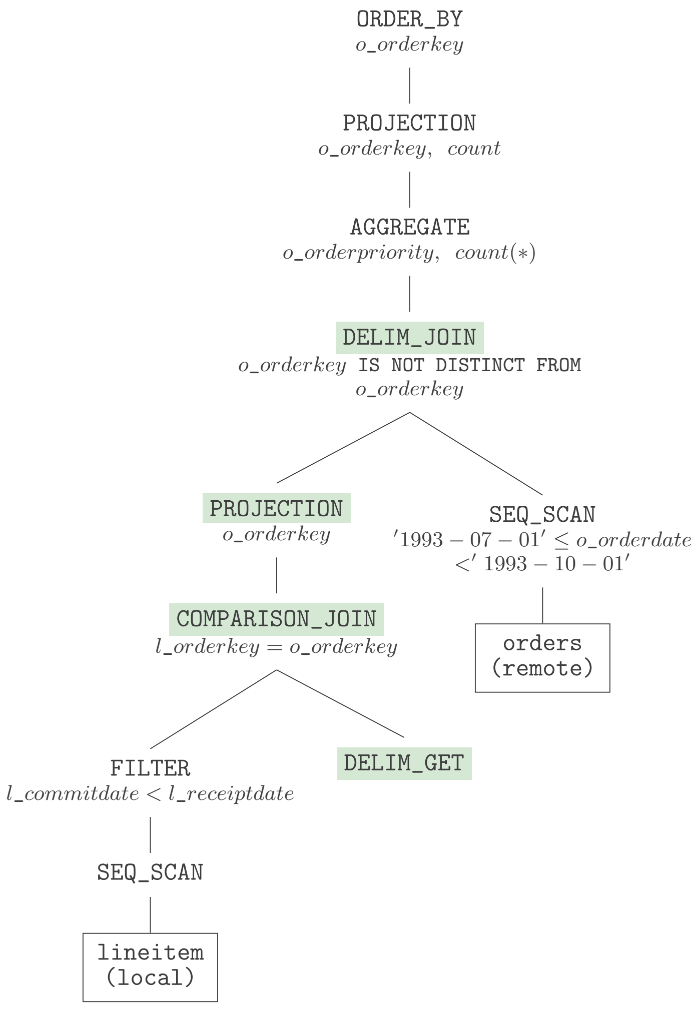


Figure 4.1: Query plan of TPC-H query number 4 (Listing 8.1). `Delim Join` and `Delim Scan` are colocated, as are `Comparison Join` and `Projection` which lie between the two.

The optimizer assigns the execution sites in the next traversal. The decisions are made greedily in a bottom-up fashion. The optimizer first processes all child nodes of the current node before deciding on the execution site. Therefore, a node's execution site is determined based on the (resolved) execution sites of its children. During the optimization, we introduce `agnostic` as a placeholder execution site for the nodes whose execution sites are not yet determined. Most non-leaf operators are `agnostic`, but even some leaf nodes can be `agnostic` as well. For example, the `range` operator is typically a leaf node but is `agnostic`. Unlike the `Get` operator, which scans data, the `range` operator generates data and not bound to any specific location. This allows the operator to be executed in either client or server.

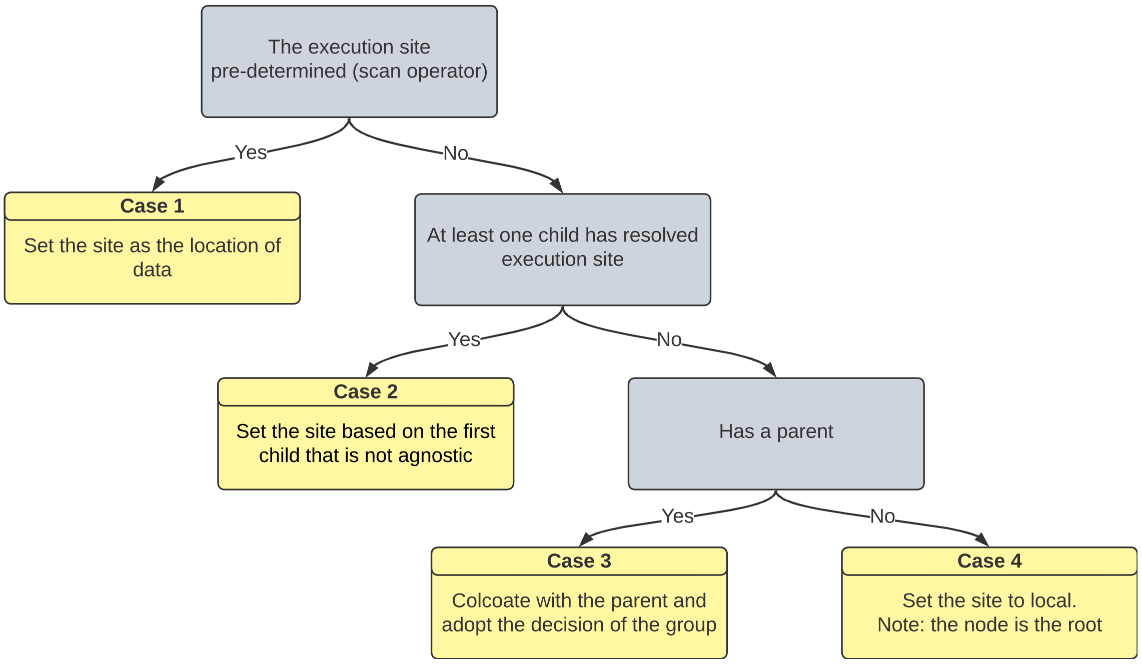

The execution site pre-determined (scan operator)
Yes
No
Case 1
Set the site as the location of data
At least one child has resolved execution site
Yes
No
Case 2
Set the site based on the first child that is not agnostic
Has a parent
Yes
No
Case 3
Colcoate with the parent and adopt the decision of the group
Case 4
Set the site to local.
Note: the node is the root

Figure 4.2: Decision tree of the heuristic optimizer.

At each operator, a decision is made following the decision tree illustrated in Figure 4.2. There are four possible outcomes at each node:

Case 1. In the case of a scan operator, resolve to the location of the data.

Case 2. Resolve to one of its children's execution sites. Here is where the *greedy* characteristic of the optimizer comes into play. The optimizer always prefers to side with the execution site of the node's first child, based on the assumption that DuckDB's join order optimizer places a "bigger" table on the left. If the left child's execution site is `agnostic`, and the right child's is not, the node follows the execution site of the right child.

Case 3. Defer decision and colocate with its parent. If none of the node's children has a resolved execution site (both `agnostic`), the node is colocated with its parent and adopts the execution site of the colocation group. A new colocation group (`node` $\rightarrow$ `parent`) is created if neither the node nor the parent is part of an existing group. If the execution site of the colocation group has been determined, the node is assigned the same site.

Case 4. Resolve to a local execution if cases 1-3 are not applicable. In other words, an `agnostic` root operator will be executed locally.

The optimizer traverses and processes the query plan in Figure 4.1 as follows.

1. `SEQ_SCAN (lineitem)` is `local` (case 1).
2. `FILTER` is `local` because its only child (`SEQ_SCAN`) is `local` (case 2).
3. `DELIM_GET` is `agnostic` and colocated with its parent `COMPARISON_JOIN` (case 3).
4. `COMPARISON_JOIN` is `local` because its left child (`FILTER`) is `local` (case 2).
5. `PROJECTION` is `local` because it is colocated with `COMPARISON_JOIN` (case 3).
6. `SEQ_SCAN (orders)` is `remote` (case 1).
7. `DELIM_JOIN` is `local` because it is colocated with `COMPARISON_JOIN` (case 3).
8. `AGGREGATE` is `local` because its only child (`DELIM_JOIN`) is `local` (case 2).
9. `ORDER_BY` is `local` because its only child (`AGGREGATE`) is `local` (case 2).

The execution site of `DELIM_GET` will be `local`, following the decision of its colocated group (determined at `COMPARISON_JOIN`).

4.2 Bridging the Gap in Statistics

4.2.1 Differences in Query Plans by DuckDB and MotherDuck

We observed that DuckDB produces differently shaped plans for the same query, depending on data placement (fully local vs hybrid placement). There are three causes for this phenomenon. First, due to an error in MotherDuck's serialization logic, the cardinalities of operators were cleared and uninitialized. Second, MotherDuck does not support *min/max* statistics for remote tables, which means that the statistics propagation rule in DuckDB optimizer (see Section 2.4.2) is not triggered when remote tables are involved. A hybrid query may be eligible for refinements such as a new filter operator, stricter join conditions, removal of ineffective filter or join operators, etc. However, due to the absence of *min/max*, the changes are not applicable and potentially result in a different query plan than its fully local counterpart. Third, DuckDB applies the table filter selectivity only when the `Get` operator's table function is named `seq_scan`. This limits the possible usage of table statistics, including failing to apply table filter selectivity on MotherDuck's remote tables, whose table function names are changed. We were able to fix the first and third issues. Support for *min/max* statistics remains to be addressed.

Fix 1. Cardinality information is correctly serialized in MotherDuck[1].

Fix 2. Table filter selectivity is applied if statistics are available regardless of the table function name. This ensures that the table filters are applied to remote tables as well [2].

4.2.2 Preserving Cardinality Information

We resolved the issue of losing cardinality information in DuckDB, discussed in Section 2.4.2, and the fixes are merged into DuckDB v1.1[3]. Here, we describe the fixes and their limitations.

Fix 3. Statistics extracted at the leaf operator `op` of a reorderable block are used to set the cardinality of `op`. This still does not guarantee that cardinalities operators in the middle (other than `op` and `input_op`) are estimated accurately. However, this guarantees a better estimate for `op`.

Fix 4. Mark join's cardinality is set to that of its right child. A mark join creates a new column to mark whether the tuple on the right-hand side has a join partner on the left-hand side. The output of the join is the right-hand side result concatenated with the new column.

Fix 5. The cardinality found in `QueryGraphManager`'s *DP Table* does not overwrite the cardinality of `GET` when the reorderable block is of form σ(`GET`). `GET` already has cardinality estimates available due to Fix 3.

Fix 6. Cartesian product created in `QueryGraphManager` has uninitialized cardinality. This is resolved by multiplying the cardinalities of left and right children.

Fix 7. Within the statistics propagation rule, a new Cartesian product operator created from a join operator adopts the cardinality of the join. If the Cartesian product is later converted to new operator(s), the cardinality is again passed on to them. One downside of this is that if the Cartesian product converts to (`FILTER` + `JOIN`), both operators will have the same cardinalities. As a future work, the filter's cardinality can be narrowed further by considering *min/max* statistics.

[1]Fix 1: https://github.com/motherduckdb/mono/pull/4943
[2]Fix 2: https://github.com/duckdb/duckdb/pull/12424
[3]Fixes 3-9: https://github.com/duckdb/duckdb/pull/13517

Fix 8. Uninitialized cardinality in newly created projection operators is solved by simply pulling up the cardinality of their child operators.

Fix 9. `Top N` [1] operator's cardinality is set to the smaller value between N and its child's cardinality.

With these fixes, every query in TPC-H and TPC-DS have no operators with uninitialized cardinality (see Table 2.1 for comparison). Fix 1 is available since MotherDuck's June 11, 2024 release, and Fixes 2-9 since DuckDB 1.1.0.

[1] Order By followed by Limit can merge into a Top N operator.

5

Cost-Based Query Optimizer

This chapter discusses the cost-based optimizer in MotherDuck. First, we motivate the need for a cost-based optimization that leverages cardinality information. Then, we formulate the optimization as a DP problem, defining the action space, state space, and cost function. Lastly, we present an overview of our algorithm through pseudocode and visualization of the decision-making process. We end the chapter with time complexity analysis and validation of the correctness and optimality of the algorithm.

5.1 Motivating Examples

We first look at the cases where cost-based query optimization can help make more efficient decisions.

Example 1: Expanding operation Some operators can produce outputs that are bigger than their children. A common example is when a projection adds new columns to a table. In (a), projection Π returns all existing plus new columns (new_cols) from a remote table, increasing the output size. Rather than downloading the results after the projection, it would be more cost-efficient to transfer the table early to reduce the amount of data that needs to be downloaded (see (a$'$)).

Certain operations, such as union or join, expand the output row-wise, meaning they could produce outputs with larger cardinality than their children. In (b), MotherDuck's heuristics chooses to transfer the local table on the right-hand side to the server. However, considering the results should be downloaded back to the client, preemptively downloading the remote table is more efficient before the $\cup_{all}$ operation expands the output size (see (b$'$)).

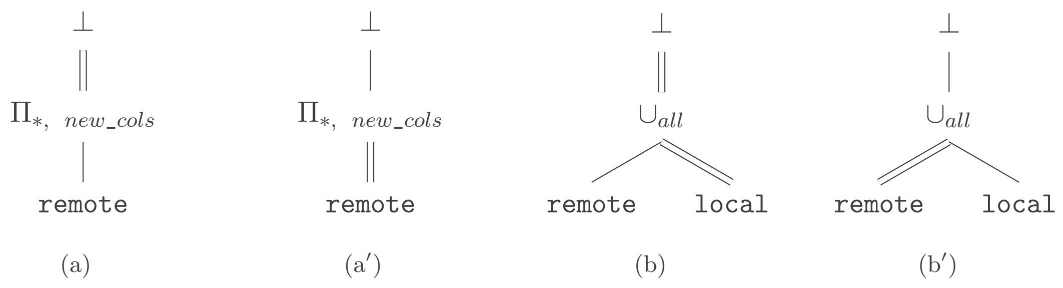


Figure 5.1: Queries with expanding operations. MotherDuck's heuristic plans (a, b) and cost-efficient plans (a′, b′). Note the difference in bridge placement (||).

Example 2: Multiple joins among local/remote relations In (c), the heuristics greedily set the join sites to that of the join's larger child. This approach moves $600K = 50K + 400K + 100K + 50K$ rows over the network. We can improve this with a cost-based approach where we select the plan that minimizes the total cost. As in (c′), we can transfer the larger table `A`, which is twice as big as table `B`. The benefit of this decision is that the join result and all other data involved in the subsequent operations are ready on the server. The remaining two joins are executed remotely, and the final result of 50K rows is transferred to the client. The total number of rows moved in the plan is $150K = 100K + 50K$, which is only 1/4 of what plan (c) moves.

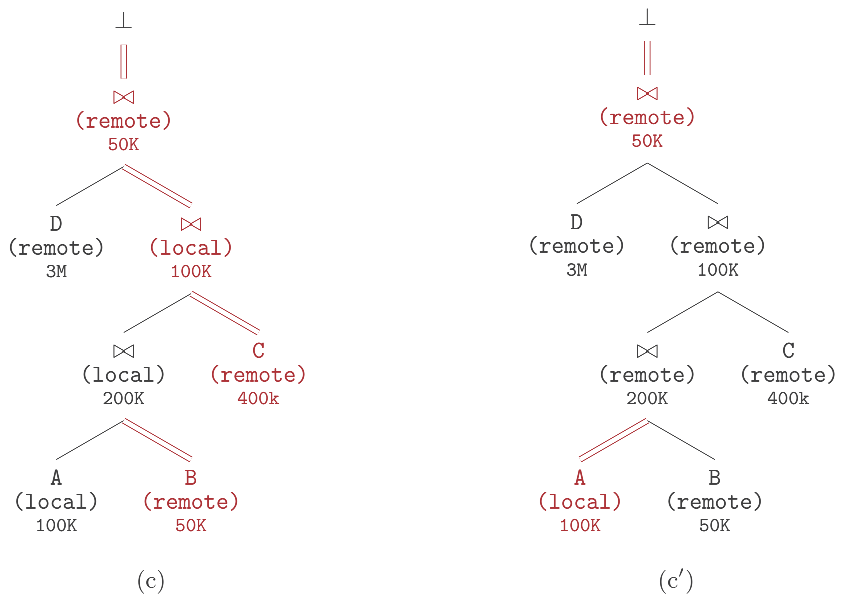


Figure 5.2: Query plans with multiple joins among local and remote relations. The numbers represent the estimated cardinalities. Edges and nodes are marked red when transferred to different sites. MotherDuck's heuristic plan is shown in (c) and the cost-efficient plan in (c′).

Example 3: Network and compute asymmetry External factors such as network and compute resources also contribute to the execution cost. In MotherDuck's client-server model, there are asymmetries in network speed and computation power. Consider a case where a user has a much faster download speed than upload and comparable computing power as the server. Say the user queries `remote` ⋈ `local`, where the remote table is larger than the local one. As opposed to MotherDuck's heuristics ((d)), transferring the large remote table to the client for the join operation would exploit the fast download speed of the client ((d′)).

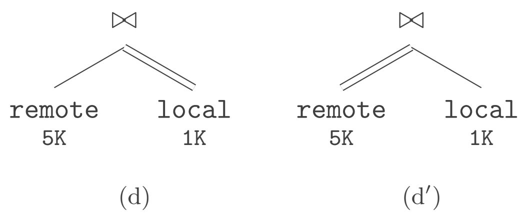


Figure 5.3: MotherDuck's heuristic plan (d) and cost-efficient plan (d′) considering asymmetric network and compute power.

Summary The examples show that utilizing the sizes of tables and intermediate results can improve the quality of query plans. Moreover, query plans could further benefit from exploiting network and compute asymmetry.

Instead of relying on a heuristic as the existing optimizer does, we could leverage available information (results sizes, network conditions, etc.) to create a cost model, which quantifies the efficiency of candidate query plans. A cost model allows us to select a plan with a *globally* minimal cost.

5.2 Formulation of DP Problem

Our goal is to determine operators' execution sites in a query plan that minimizes the global cost. Our approach draws inspiration from the join ordering mechanism of System-R [27]. During join ordering, System-R considers *interesting orders* of tuples. For each join plan, two options are retained – one where the output relation is ordered (sort-merge join) and one where it is not (nested-loop join). The idea is to preserve plans that are not immediately the cheapest to execute (*e.g.,* sort-merge join) but whose physical property (ordered-ness) could reduce the costs of the subsequent joins.

Similarly, we would like to preserve a plan per *interesting sites*. For each operator, (at most) two options are available – `local` and `remote`. In our case, the execution site of an operator dictates a physical property, the *result site*. Executing an operator on one site may initially be more expensive than on the other. However, the result site could turn out to be more favorable for the following operations. Hence, we retain a plan per possible execution site.

We employ DP approach as System-R, where we build plans bottom-up. We traverse the query tree from the leaf nodes until we reach the root. At each operator, two sub-plans (`local` and `remote` executions) and their costs are computed. Eventually, at the root, we pick the best plan that produces results locally, assuming the query is DQL, where results must be presented to the user who is on the client side. For other query types (DML and DDL), the root execution site depends on where relevant data resides.

For costing plans, we only consider *internal* information, such as the sizes of tables and estimated intermediate results. We exclude external factors, such as network and CPU conditions, from the optimization. We define the cost as the amount of data we transfer over the network, approximated with cardinalities. With our DP framework, however, the cost model can easily extend to incorporate additional cost factors, such as network latency, bandwidth, CPU, and I/O costs.

This section formulates hybrid query optimization as a DP decision problem, defining the action space, state space, and cost function.

Action space There are two possible actions for each operator – to execute locally or remotely. However, scan operators can only have one option because it must be executed where the data source is. All other types of operators are agnostic, meaning they can be executed either locally or remotely. Let us define the action space as $\mathcal{A} = \{\texttt{local},\ \texttt{remote}\}$ and possible actions per operator *op* are:

$$\alpha_{op} = \begin{cases} \{\texttt{local}\} & \text{if } op \text{ scans local data,} \\ \{\texttt{remote}\} & \text{if } op \text{ scans remote data or is a remote function,} \\ \{\texttt{local},\ \texttt{remote}\} & \text{otherwise.} \end{cases}$$

State space Each operator in a query plan has an associated state, which consists of its result size, subplan per execution site, and colocation parent if it has one. Formally, the state space $\mathcal{X}$ of query plan q is defined as

$$\mathcal{X} = \{(result\_size,\ colocation\_parent,\ per\_site\_plan) \mid \forall \text{ operators in } q\}\,. \tag{5.1}$$

In addition to the operators in a query plan, we consider the materialization operator $\perp$ to compute the cost of materializing the results on each location.

result_size is the size of an output relation of *op*, which we approximate as the cardinality of the intermediate results. *colocation_parent* is a reference to its colocated parent. *per_site_plan* contains the optimal cost and policy for its children. An optimal policy is a sequence of actions to be taken in a given state to achieve a minimal cost [68, 197].

Cost function Our objective is to minimize the total estimated *amount of data* that is transferred over the network during the query execution. The execution cost of *op* in site *a* is the sum of 1) the children's execution cost so far and 2) the potential transfer cost from the children's execution site to the parent's. Formally, the *optimal* execution cost of *op* at *a* is:

$$C(op,\ a) = \begin{cases} 0 & \text{if } op \text{ is a leaf node,} \\ \sum\limits_{c \in op.children} \left(\min\limits_{ca \in A(c,\ op,\ a)} \{C(c,\ ca) + T(c,\ ca,\ a)\} \right) & \text{otherwise,} \end{cases} \tag{5.2}$$

where $A(\cdot)$ returns an action space of a child:

$$A(child,\ op,\ a) = \begin{cases} \{a\} & \text{if } op \text{ and } child \text{ are colocated,} \\ \alpha_{child} & \text{otherwise,} \end{cases} \tag{5.3}$$

and $T(\cdot)$ returns the transfer cost of moving the result produced by a child:

$$T(child,\ from,\ to) = \begin{cases} child.result\_size & \text{if } from \neq to, \\ 0 & \text{otherwise.} \end{cases} \tag{5.4}$$

Finally, we compute the cost of remote and local materialization. For DQL statements, the result of a query should be present on the client side. Therefore, if the root operator is executed remotely, a final download cost $T(root,\ \texttt{remote},\ \texttt{local})$ is added to the execution cost.

5.3 Algorithm

5.3.1 Overview

Algorithm 2 show important data structures we use in `Optimizer`, `State`, and `ExecutionPlan` objects. `Optimizer` has *states* which maps each logical operator to a `State`. A `State` contains information on the operator's result size, possible execution sites (α_{op}), its colocated

Algorithm 2 Key variables in Optimizer, State, and ExecutionPlan objects

```
1: class Optimizer
2:     map<operater, State> states
3: class State
4:     int result_size
5:     vector<ExecutionSite> possible_execution_sites
6:     operator colocation_parent
7:     map<ExecutionSite, ExecutionPlan> per_site_plan
8: struct ExecutionPlan
9:     int cost
10:    vector<ExecutionSite> optimal_child_sites
```

Algorithm 3 Recursive function to find optimal execution plans

```
1: function PlanExecution(op)
2:     for each child of op
3:         PlanExecution(child)
4:     state ← states[op]
5:     for each site in state.possible_execution_sites
6:         execution_cost ← 0
7:         optimal_child_sites ← [ ]
8:         for each child of op
9:             child_state ← states[child]
10:            colocated ← (child_state.colocated_parent = op)
11:            optimal_site, optimal_cost ← OptimalChildSiteAndCost(child_state, site, colocated)
12:            execution_cost ← execution_cost + optimal_cost
13:            append(optimal_child_sites, optimal_site)
14:        if op is root ∧ site is remote then
15:            total_cost ← total_cost + TransferCost(op, remote, local)
16:        SaveExecutionPlan(site, ExecutionPlan(cost, child_sites))
```

parent, and execution plans for children. An *ExecutionPlan* is built for each possible execution site and stores the cost of the plan and optimal sites for the children.

Our DP algorithm consists of two traversals of a query plan in a depth-first-search (DFS) fashion. The first traversal initializes `State` per node and marks colocated operators (Algorithm 2). A `State` contains *result_size* (cardinality), *possible_execution_sites* (α_{op}), *colocation_parent* (an optional reference to the colocated parent), and *per_site_plan* (optimal cost and associated optimal policy for children, initially empty). Note that, in our implementation, the action space (α_{op}) is merged into the state.

Algorithm 4 Cost function

```
1: function OptimalChildSiteAndCost($child\_state$, $parent\_site$, $colocated$)
2:     if $colocated$ then
3:         ▷ colocated child must follow the execution site of the parent
4:         return $parent\_site$, GetExecutionCost($child\_state$, $parent\_site$)
5:     $optimal\_site \leftarrow parent\_site$
6:     $optimal\_cost \leftarrow$ INF
7:     for each $child\_site$ in $child\_state.possible\_execution\_sites$
8:         ▷ cost is the sum of the execution cost so far and transfer cost
9:         $cost \leftarrow$ GetExecutionCost($child\_state$, $child\_site$)
10:                + TransferCost($child\_state$, $child\_site$, $parent\_site$)
11:        if $child\_cost < optimal\_cost$ then
12:            $optimal\_site \leftarrow child\_site$
13:            $optimal\_cost \leftarrow child\_cost$
14:    return $optimal\_site$, $optimal\_cost$
```

Algorithm 5 Transfer cost

```
1: function TransferCost($state$, $source\_site$, $target\_site$)
2:     if $source\_site = target\_site$ then
3:         $transfer\_cost \leftarrow 0$
4:     else
5:         $transfer\_cost \leftarrow state.result\_size$
6:     return $transfer\_cost$
```

In the second traversal, optimal execution plans are constructed bottom-up (Algorithm 3). We iterate over the *possible_execution_sites* of *op* (line 5). Given *site*, we find the optimal execution site and cost per child (lines 8-13). The *cost* of *op* at *site* is the sum of the child costs (line 12). The optimal execution site per child is added to *optimal_child_sites* (line 13). The execution plan for *op* at *site* are saved in *per_site_plan*[*site*] (line 16).

The optimal execution site and cost of a child (line 11) are determined in Algorithm 4. If the child is colocated with its parent (*i.e., colocated* is *true*), then the only option is to execute the child at *parent_site*. The cost is simply what has been accrued so far without any additional transfer cost (lines 2-4). The GetExecutionCost function retrieves the accumulated cost of the child, which is memoized as part of *child_state*.

If the child is not colocated, we iterate over the *possible_execution_sites* of the child and compare the costs (lines 7-14). *cost* of a child at *child_site* includes potential TransferCost on top of the execution cost so far (from GetExecutionCost). The TransferCost (Algorithm 5) checks whether the *child_site* and *parent_site* are the same. If so, no cost is added; otherwise, the child's $result\_size$ becomes the transfer cost.

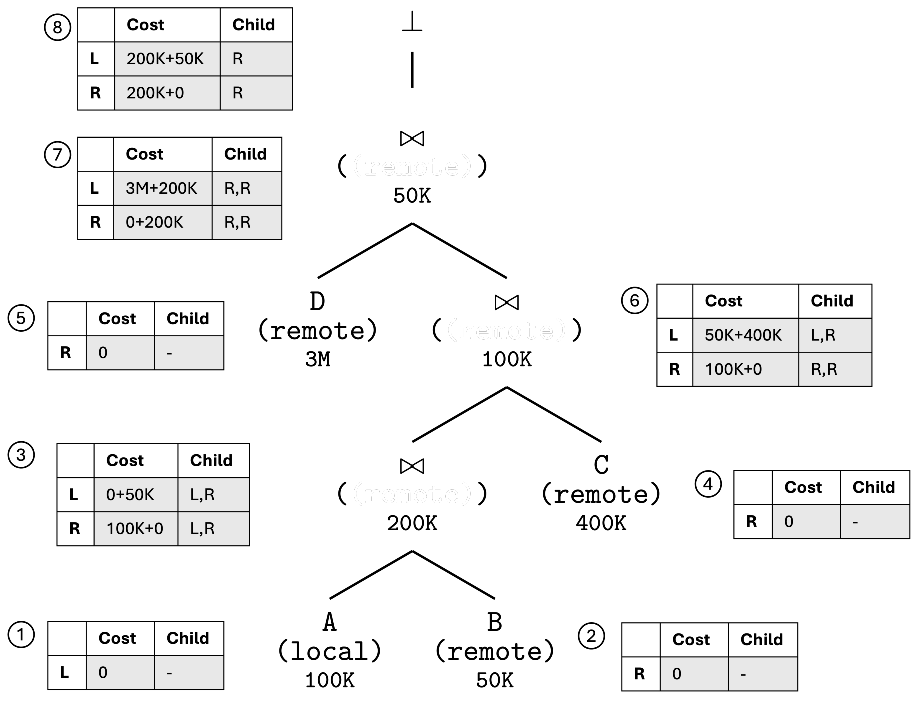


Figure 5.4: Bottom-up plan construction, in order from operator 1 to 8. L and R stand for `local` and `remote` executions, respectively. The *per_site_plan* is displayed as a table at each node. The Cost column stands for execution cost at L or R. The Child column stands for the optimal execution sites of the children (from left child to right).

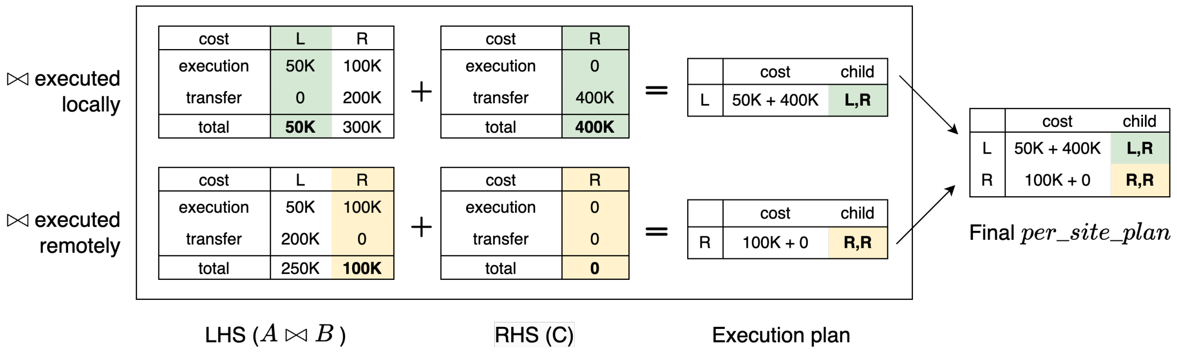


Figure 5.5: Visualization of subplan construction in operator 6 (Figure 5.4).

Figure 5.4 demonstrates the bottom-up plan construction of the query plan previously seen in Figure 5.2. The scanning operators (1, 2, 4, and 5) only have one subplan for where the data is located. Other operators (3, 6, 7, and 8) have two subplans each, one more `local` execution and another for `remote`.

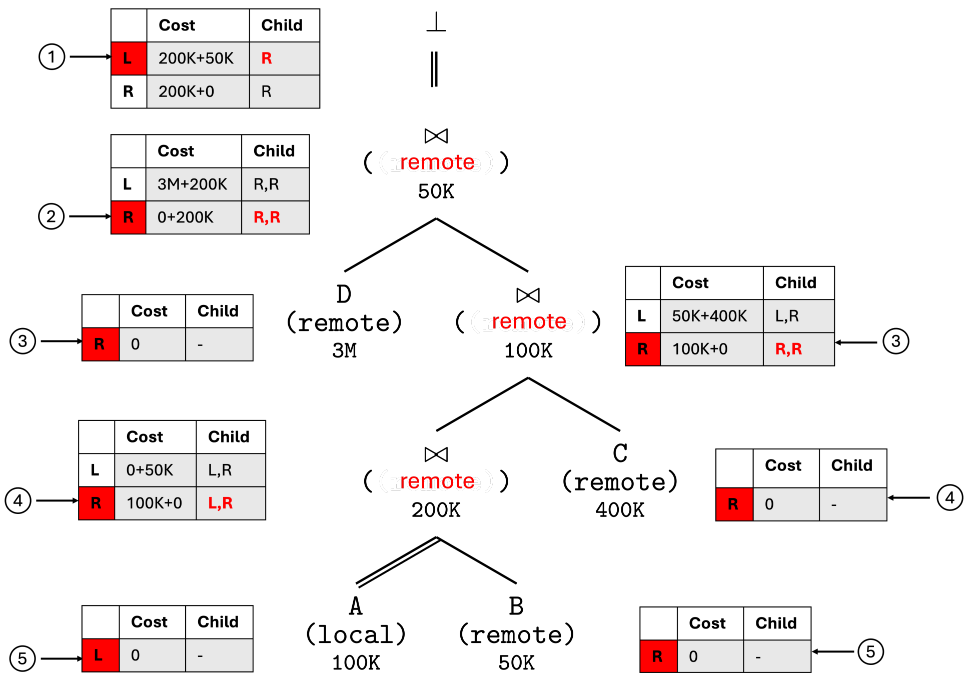
As an example, the process of subplan construction at operator 6 (the join between $(A \bowtie B)$ and C) is visualized in Figure 5.5. Two scenarios are considered.

1. If the join is executed locally, the best execution site for LHS join is `local`, with a cost of 50K. For the RHS scan of table `C`, `remote` is the only option. Since the table `C` is stored on the server, the transfer cost is 400K.

2. If the join is executed remotely, LHS join must be executed remotely to achieve the minimal cost of 100K. This way, the LHS result is prepared on the server, saving the transfer cost. For the RHS, again, the only option is to execute remotely. Since the table does not have to be transferred, the RHS cost is 0.

Now, operator 6 has the execution plans for local and remote execution scenarios.

At the top, the materialization cost is computed. Since the query is a DQL statement, we pick the local execution plan. Finally, as shown in Figure 5.6, the optimal child execution sites are propagated downwards. Cost-based optimization is completed, and execution sites for all operators are assigned.



①

	Cost	Child
L	200K+50K	R
R	200K+0	R

②

	Cost	Child
L	3M+200K	R,R
R	0+200K	R,R

③

	Cost	Child
R	0	-

③

	Cost	Child
L	50K+400K	L,R
R	100K+0	R,R

④

	Cost	Child
L	0+50K	L,R
R	100K+0	L,R

④

	Cost	Child
R	0	-

⑤

	Cost	Child
L	0	-

⑤

	Cost	Child
R	0	-

Figure 5.6: Top-down decision propagation. The decision is made at the top and relayed to the children.

5.3.2 Time Complexity

Our optimizer traverses a given tree of operators twice – first to initialize states and mark colocations and second to determine optimal execution sites recursively. The time complexity of DFS traversal of a tree is $O(N)$ where N is the number of nodes (*i.e.,* operators).

Our optimizer performs two depth-first traversals of a given operator tree. The first traversal initializes states and marks colocations, while the second determines optimal execution sites recursively. The time complexity of a DFS traversal of a tree is $O(N)$, where N is the number of nodes (operators). As described in Algorithm 3,

- The outer loop iterates over at most two execution sites, $O(2)$.
- The inner loop iterates over the children of an operator, also $O(2)$, as each operator has at most two children.
- Inside the inner loop, the cost function `OptimalChildSiteAndCost` is called, iterating over possible execution sites for each child, which is again $O(2)$.

Thus, the second traversal involves $O(8N)$ function calls, or $O(N) \cdot O(2) \cdot O(2) \cdot O(2)$. The total time complexity is $O(9N) = O(N) + O(8N)$, which simplifies to $O(N)$.

5.3.3 Validation

We validated the correctness and optimality of the algorithm. The *correctness* was tested by verifying whether the intermediate results in *states* match the manually computed values. The *optimality* was tested against a brute-force optimizer, which generates all possible plans for a given query. If a query has n agnostic operators, a brute-force optimizer creates 2^n plans and picks the cheapest plan(s) among them. There may be multiple cheapest plans. We verify that the plan from our optimizer is one of the cheapest plans selected by the brute-force optimizer.

6

Experiments

We evaluate the performance of the cost-based optimizer against the heuristic optimizer. Starting with the experimental setup, we present the comparisons between the two optimizers and the influence of fixes described in Section 4.2. Then, we follow up by laying out possible reasons that may contribute to the slowdown of some queries.

6.1 Setup

6.1.1 Benchmarks

We used TPC-H, TPC-DS, and Join Order Benchmark (JOB) to test the performance of the query optimizers. TPC-H [69] and TPC-DS [70] are widely used benchmarks for OLAP databases. They generate synthetic data sets assuming uniformity and independence. However, real-world data frequently have correlations and non-uniform distributions, which make cardinality estimation more challenging. For a more realistic benchmark on cardinality estimation, Leis *et al.* introduced JOB based on Internet Movie Data Base (IMDB) [18]. The JOB queries also closely resemble real-world use cases and have, on average, 8 joins per query.

To test the performance for hybrid queries, we ran the benchmarks with all possible table placements (`local` or `remote`). For instance, a query involving three base tables has $2^3 = 8$ variations. Including all possible table placements, the benchmark queries extend to:

- TPC-H: 452 queries (originally 22)
- TPC-DS: 18,156 queries (originally 99)
- JOB: 166,448 queries (originally 113)

.

6.1.2 Experimented Optimizers

We compare the performance of the heuristic and cost-based optimizers. Furthermore, we observe the effects of fixes (see Section 4.2). At the time of the experiments, the MotherDuck-side change, Fix 1 (change in MotherDuck serialization to preserve cardinality), was already released in the production environment. Therefore, we experiment with the influence of DuckDB-side cardinality fixes.

For DuckDB v1.0, the versions of optimizers in consideration are:

H_1. Heuristic optimizer with Fix 1 (MotherDuck fix)

H^*. Heuristic optimizers with Fixes 1-9 (MotherDuck and DuckDB fixes)

C_1. Cost-based optimizer with Fix 1 (MotherDuck and DuckDB fixes)

C^*. Cost-based optimizers with all fixes Fixes 1-9 (MotherDuck and DuckDB fixes)

We also compare the performance of the heuristic and cost-based optimizers with DuckDB v1.1. On top of the Fixes 1-9, DuckDB v1.1 has a few other changes that affect plan structures or cardinality estimates:

- Support for join reordering of left semi- and anti-joins (see Section 2.4.2)
- Different left-right optimization for joins. Instead of cardinalities, estimated build sizes of left and right relations are compared to decide the build side. Moreover, the relation with join in its subtree is preferred as the probe side. This means that there is an increased chance that a smaller relation is not on the build side (left) of the join (see Section 2.4.1.4).
- Less accurate but more memory efficient implementation of HLL, which is used for distinct count statistics (see Section 2.4.2).

We refer to the heuristic and cost-based optimizers with DuckDB v1.1 as $H_{v1.1}$ and $C_{v1.1}$, respectively. We also present preliminary results with a modified cost model with constant network latency cost. We refer to the optimizer with the new cost model as $C'_{v1.1}$.

Name	Type	DuckDB	Fixes
H_1	heuristic	v1.0	Fix 1
C_1	cost-based	v1.0	Fix 1
H^*	heuristic	v1.0	all fixes
C^*	cost-based	v1.0	all fixes
$H_{v1.1}$	heuristic	v1.1 (incl. all fixes)	-
$C_{v1.1}$	cost-based	v1.1 (incl. all fixes)	-
$C'_{v1.1}$	cost-based (+ constant latency cost)	v1.1 (incl. all fixes)	-

Table 6.1: List of optimizers by type, base DuckDB version, and applied fixes.

6.1.3 Environments

We conducted our experiment on an AWS's c6gd.4xlarge EC2 instance, which has 16 CPUs and 32GiB memory [1]. We queried MotherDuck's production environment, which assigns 12 core instances per user. To match the number of CPUs used on the client and server sides, we limited our instance's CPU usage to 12 when executing the queries.

6.2 Heuristic vs Cost-Based Optimizer without DuckDB Fixes

We first present the results of H_1 and C_1 – the heuristic and cost-based optimizers without DuckDB-side fixes.

6.2.1 Results

We plot the execution time speedup with the cost-based optimizer against the execution time of the heuristic optimizer. The x-axis in the plot is the execution time in seconds. The y-axis is the speedup of a query relative to its baseline speed (*i.e.,* execution time with heuristic optimizer). Also, we only compare the queries whose plans are changed by the cost-based optimizer. A query is omitted from the comparison if the heuristic and cost-based optimizer generate the same plan.

The numbers of queries with different plans are:

- TPC-H: 96 different plans
- TPC-DS: 10,314 different plans
- JOB: 18,202 different plans

[1] `https://aws.amazon.com/ec2/instance-types/c6g/`

	TPC-H	TPC-DS	JOB
speedup (count, avg)	40, 1.33x	5,817, 1.11x	5,214, 1.58x
slowdown (count, avg)	56, 0.82x	4,497, 0.89x	12,988, 0.24x
total (count, avg)	96, 1.03x	10,314 1.02x	18,202, 0.62x

Table 6.2: Counts and average speedups of queries that speed up and slow down (H_1 vs C_1).

Table 6.2 shows that for all benchmarks, more queries slow down than speed up. For TPC-H and TPC-DS, while the overall average speedup is close to 1.0, some queries become especially slow. Examples are the red point (TPC-H q01, bottom-most) in Figure 6.1 and the cluster of purple points (TPC-DS 78, bottom-left) in Figure 6.2.

For JOB, the queries slow down 0.62x on average. In Figure 6.3, we see two distinct clusters. Most queries in the upper cluster speed up, with some reaching 32x speedup. The lower cluster suffers from slowdown. In both clusters, short-running queries are more likely to slow down. Some queries slow down to almost 1/64x the heuristic optimizer's speed. This trend is also visible in TPC-H and TPC-DS. Short-running queries experience regression, while some long-running queries speed up.

Without DuckDB-side fixes, cardinality estimates for some operators are not preserved or are inaccurate. Loss of cardinality estimates results in C_1 optimizing based on the "incorrect" values, which leads to inefficient plans for many queries. Below, we present an example illustrating how C_1 makes suboptimal decisions due to poor cardinality estimates.

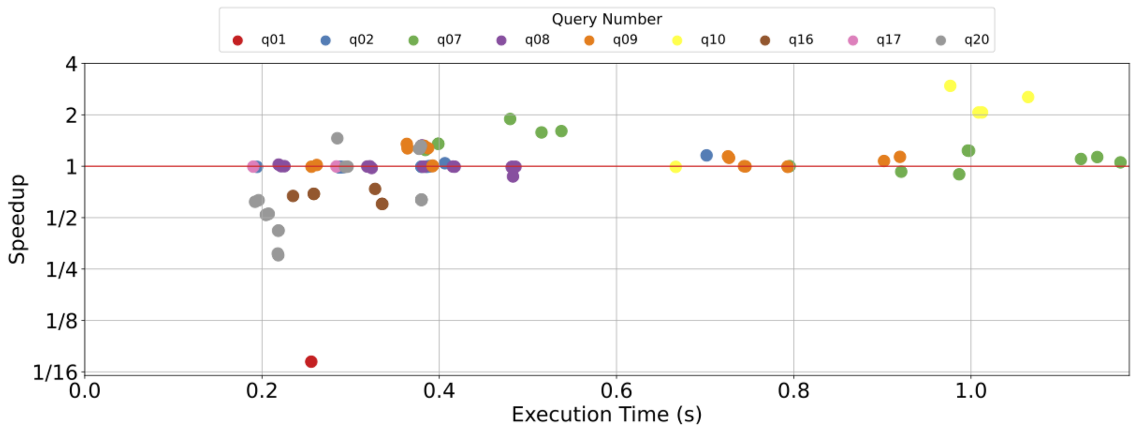


Figure 6.1: TPC-H speedup without fixes (H_1 vs C_1).

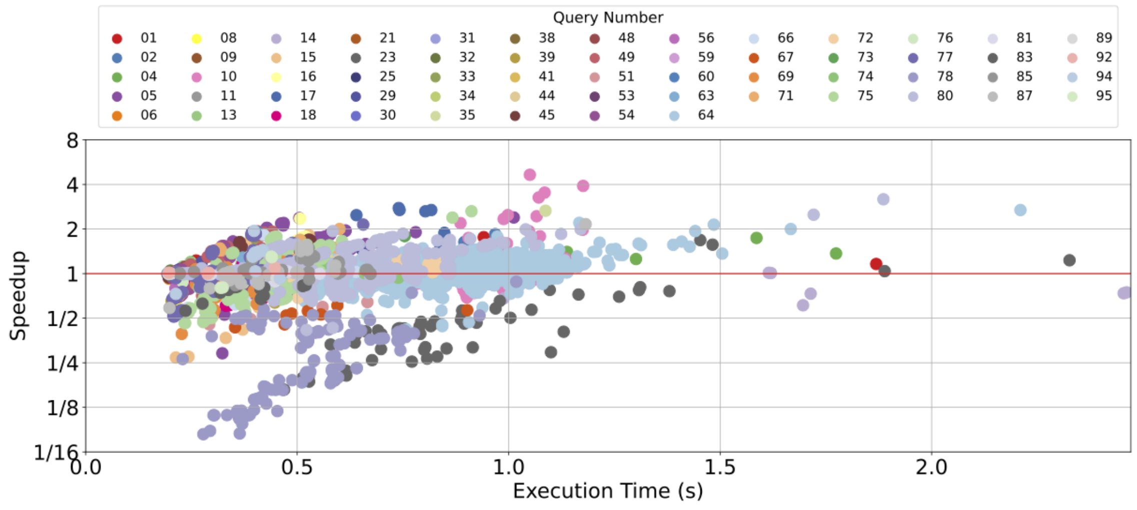


Figure 6.2: TPC-DS speedup without fixes (H_1 vs C_1).

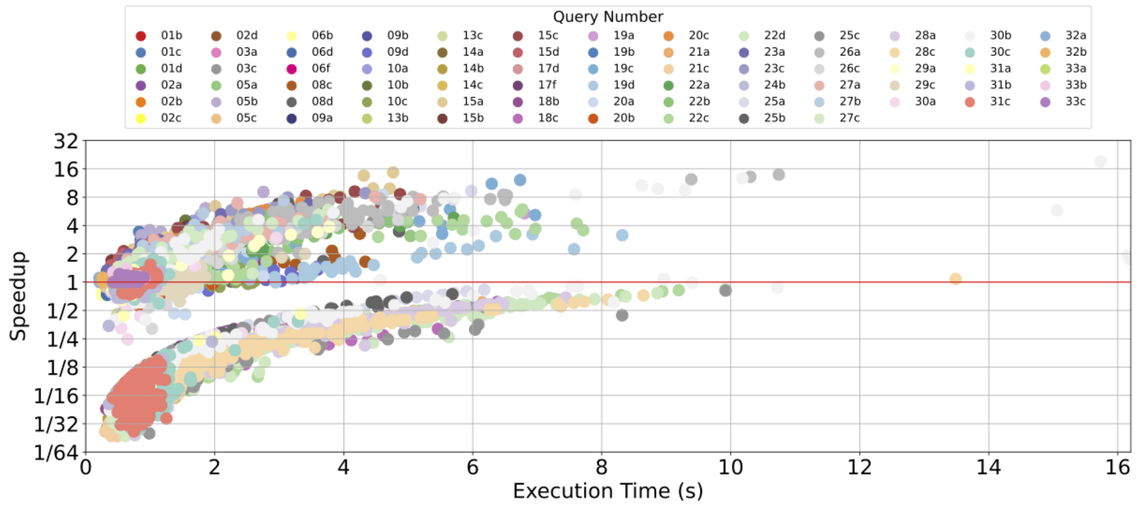


Figure 6.3: JOB speedup without fixes (H_1 vs C_1).

6.2.2 Example

Among TPC-DS queries, query 78 (variation 17 [1]) has the worst slowdown, 0.08. A partial query plan is shown in Figure 6.4. The plan demonstrates how missing cardinality information negatively affects the decision-making of the cost-based optimizer. The cardinality of the second `HASH_JOIN` is 0. It was initially set to $1,441,548$ during the join order optimization but later cleared out. The cost-based optimizer (C_1) inserts a bridge between

[1]Table placement: catalog_returns at `local`, catalog_sales at `local`, date_dim at `remote`, store_returns at `local`, store_sales at `local`, web_returns at `local`, and web_sales at `local`

HASH_JOIN and FILTER, where, according to the provided information, the transition from local to remote execution is the cheapest (0 cost). In reality, the cost-based optimizer ends up having to transfer a large HASH_JOIN result, which has $1,441,548$ rows.

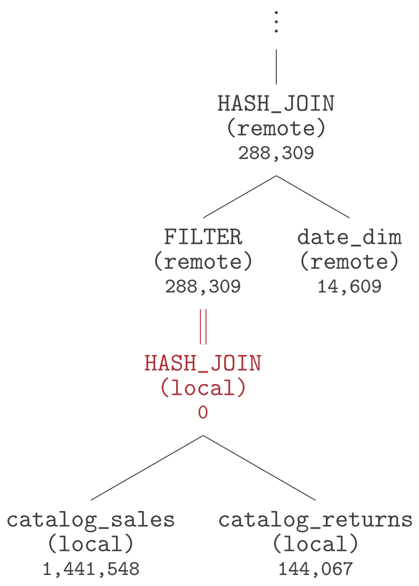


Figure 6.4: Partial query plan of variation 17 of TPC-DS query 78.

6.3 Heuristic Optimizer with and without DuckDB Fixes

Before comparing the performance of the heuristic and cost-based optimizers with fixes, we gauge how the fixes affect the baseline of the heuristic optimizer. To this end, we compare the performance of the heuristic optimizer before and after the DuckDB fixes are applied – H_1 and H^*. H_1 has Fix 1 applied since the fix became available in production during this experiment. H^* has all the fixes applied (Fixes 1-9).

Recall that the heuristic optimizer does not rely on cardinality information. Therefore, fixes that preserve cardinality after DuckDB's join order optimization (Fixes 3-9) are irrelevant in this experiment. Only Fixes 1 and 2 are relevant here. Fix 1 corrects MotherDuck logic for serializing cardinality, and Fix 2 applies table filter selectivity to remote table cardinality, thereby affecting the DuckDB join order optimizer.

The numbers of queries with different plans for H_1 and H^* are:

- TPC-H: 70 different plans
- TPC-DS: 5,489 different plans
- JOB: 126,536 different plans

6.3.1 Results

With Fixes 1 and 2, MotherDuck optimizer receives plans with intended join orders by DuckDB. Hence, we expected an improvement in execution time, as DuckDB is optimizing with more accurate estimates for remote table cardinalities. From Table 6.3, more queries speed up than slow down. However, there are no significant improvements, with average speedups of 1.13x for TPC-H, 1.09x for TPC-DS, and 1.05x for JOB. Around 30% of queries in TPC-H and TPC-DS actually slow down. In JOB, the percentage increases to 46%.

We also report the results specifically of queries that affect the baseline in H^* vs C^* comparison. These are the ones that have different plans for H^* and C^*. Therefore, their speedups (or slowdowns) from H_1 to H^* cause the baseline to shift in the H^* vs C^* comparison. The statistics are shown in the last line of Table 6.3 – “baseline changes”. TPC-DS and JOB have an increase in baseline to 1.10x and 1.05x, respectively. Only one query from TPC-H changes the baseline, with a speedup of 1.02x.

	TPC-H	TPC-DS	JOB
speedup (count, avg)	45, 1.31x	3,850, 1.16x	67,535, 1.20x
slowdown (count, avg)	25, 0.81x	1,639, 0.90x	59,001, 0.87x
total (count, avg)	70, 1.13x	5,489, 1.09x	126,536, 1.05x
baseline changes (count, avg)	1, 1.02x	1,990, 1.10x	348, 1.05x

Table 6.3: Counts and average speedups of queries that speed up and slow down (H_1 vs H^*).

Figures 6.5 to 6.7 show the speedup of H^* against H_1. The colored queries are the ones that affect the baseline in the H^* vs C^* comparison. Other queries that do not affect the baseline are marked in gray.

In Figure 6.5, the speedup range in TPC-H is between 1/2x and 2x. And no queries participate in H^* vs C^* comparison, *i.e.,* the baseline remains unchanged. In Figure 6.6, the speedup in TPC-DS is between 1/4x and 4x. The short-running queries experience slowdown, but beyond the 0.5-second mark, we start to observe more speedups. Limiting to the colored dots, we see the short-running queries slow down to 1/2x the speed, which is not as much as the slowdown for the gray dots. JOB has a larger speedup range, 1/8x to 16x (Figure 6.10). While JOB also exhibits a trend where short-running queries slow down, the magnitude of the slowdown is especially bigger here.

Overall, H^* makes the query execution marginally faster. Also, it introduces a significant slowdown, which is especially evident in JOB queries.

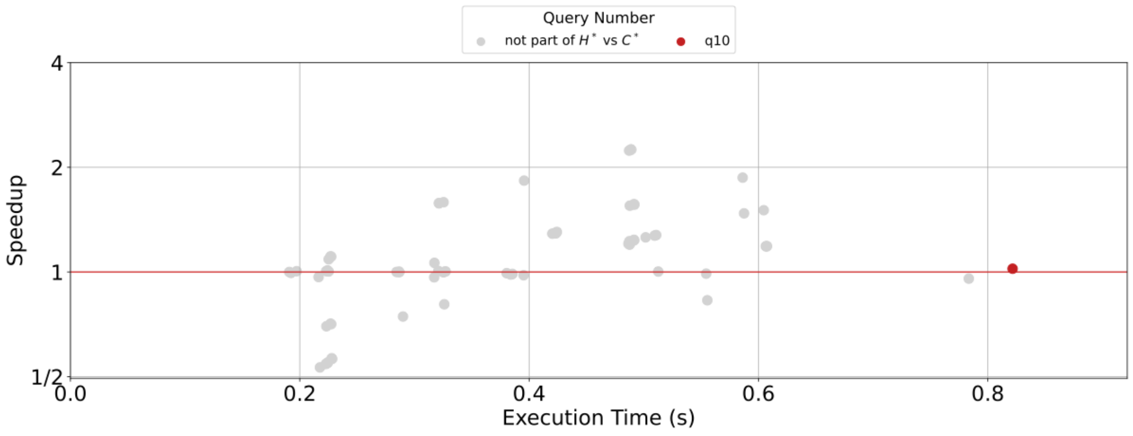


Figure 6.5: TPC-H speedup of the heuristic optimizer with all fixes (H_1 vs H^*)

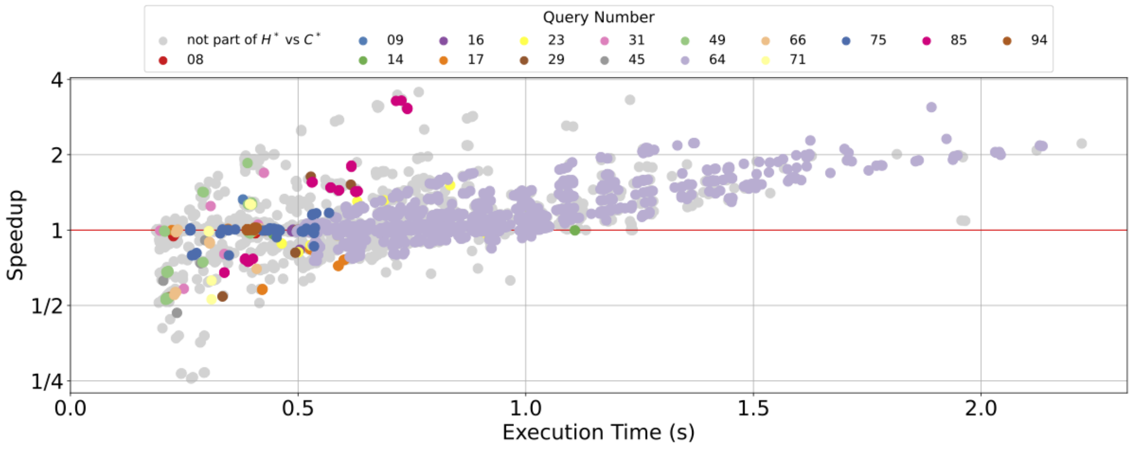


Figure 6.6: TPC-DS speedup of the heuristic optimizer with all fixes (H_1 vs H^*)

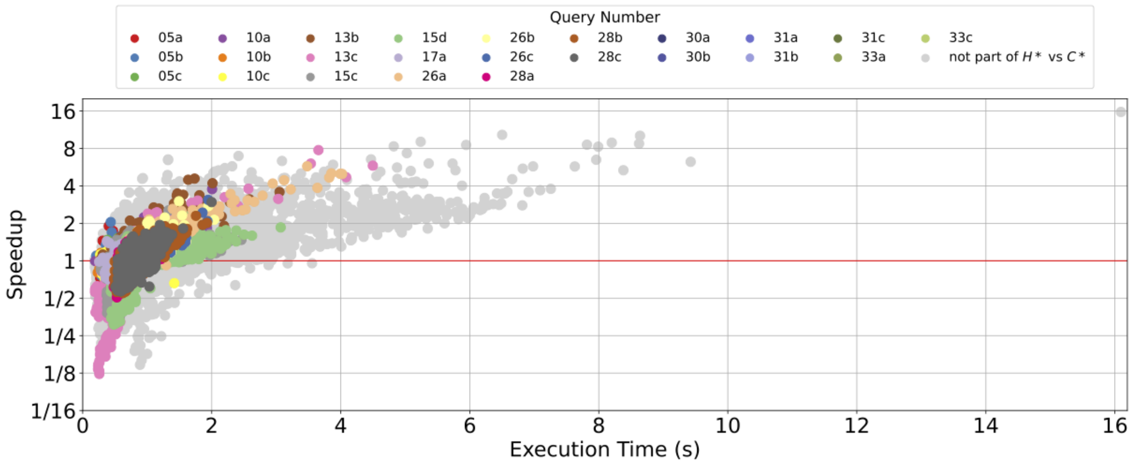


Figure 6.7: JOB speedup of the heuristic optimizer with all fixes (H_1 vs H^*).

6.4 Heuristic vs Cost-Based Optimizer with DuckDB Fixes

Now, we compare the heuristic and cost-based optimizers, both with all fixes (H^* vs C^*). Here, we report the experimental results and hypothesize possible reasons behind regressions.

In the comparison between H^* and C^*, the numbers of queries with different plans are:

- TPC-H: 44 different plans
- TPC-DS: 6,078 different plans
- JOB: 720 different plans

6.4.1 Results

	TPC-H	TPC-DS	JOB
speedup (count, avg)	32, 1.41x	3,456, 1.11x	476, 1.18x
slowdown (count, avg)	12, 0.60x	2,622, 0.92x	244, 0.98x
total (count, avg)	44, 1.19x	6,078, 1.03x	720, 1.11x

Table 6.4: Counts and average speedups of queries that speed up and slow down (H^* vs C^*).

Table 6.2 show that more queries speed up than slow down. Average speedup is the highest for TPC-H, with 1.19x. 32 queries experience a speedup with an average of 1.41x, while 12 queries show a slowdown with an average of 0.60x. TPC-DS and JOB exhibit lower average speed up overall – 1.11x and 1.18x, respectively. However, queries that slow down have a better average, 0.92x and 0.98x, respectively. This indicates that while fewer queries experience significant speedup, most regressions are contained to a smaller magnitude.

Looking at Figure 6.8 and Figure 6.9, we see that short-running queries in TPC-H and TPC-DS are slowing down. For instance, in TPC-DS (Figure 6.9), most queries that took less than 0.3 seconds with heuristic optimizer (H^*) experience slowdown. Between 0.3 and 0.6 seconds, the dots are distributed symmetrically along the baseline, meaning there are about an equal number of slowdowns and speedups. Going towards the right side of the graph, we observe more queries speed up. Some long-running queries get substantially faster. Most evident are the orange (query 10) and green points (query 17), with some variations gaining around 4x speed. For JOB, we see a different trend compared to TPC-H and TPC-DS. Both short- and long-running queries speed up in Figure 6.10. Moreover, there are only a few slowdowns, which still remain very close to the baseline.

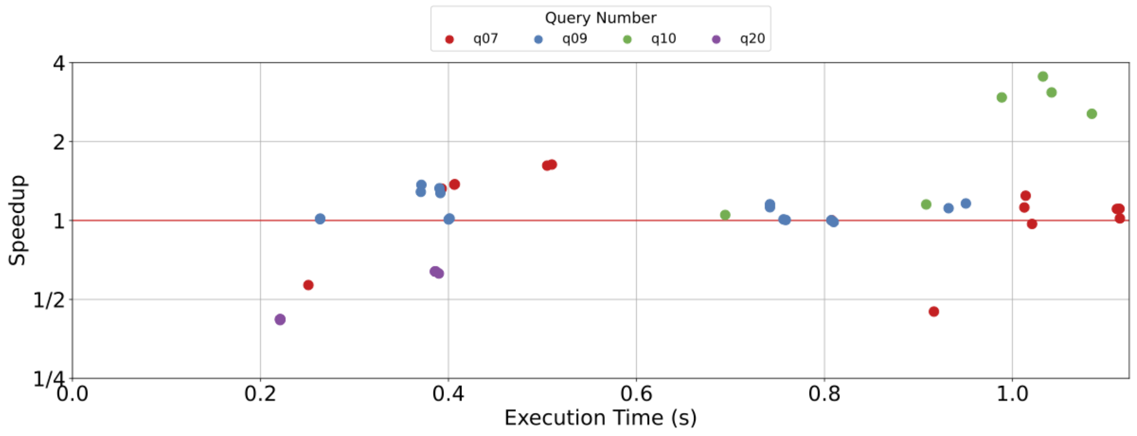


Figure 6.8: TPC-H speedup with fixes (H^* vs C^*).

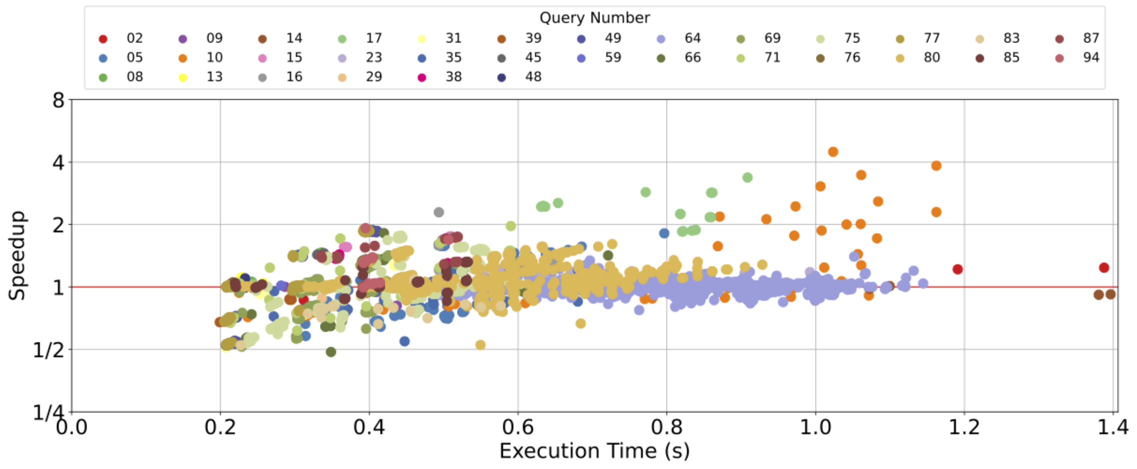


Figure 6.9: TPC-DS speedup with fixes (H^* vs C^*).

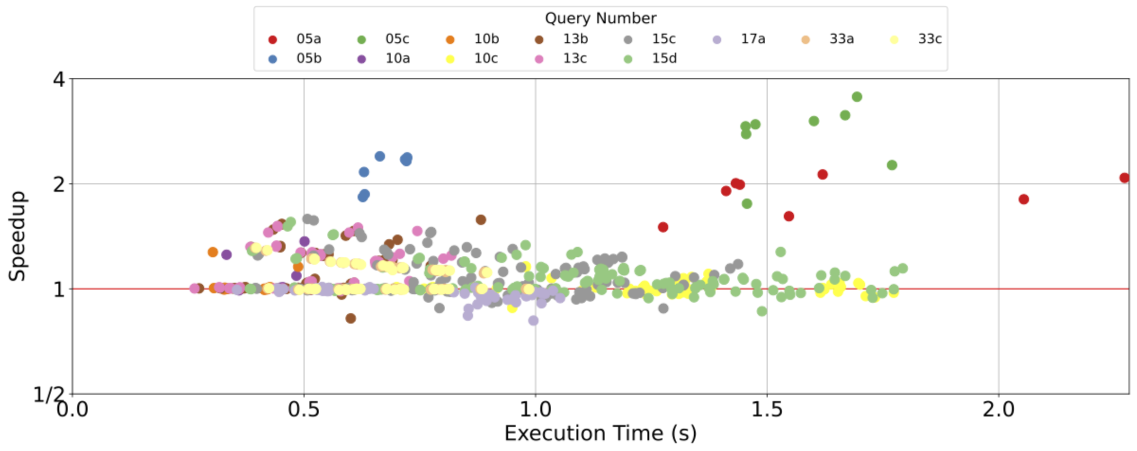


Figure 6.10: JOB speedup with fixes (H^* vs C^*).

The relative speedup is higher with fixes than without. Table 6.5 shows that for all benchmarks C^* has better improvements over H^* than C_1 does over H_1. JOB has the most improvement, going from 0.62x with C_1 to 1.11x with C^*, followed by TPC-H. TPC-DS shows a slight improvement. It is worth noting that C^* avoids some significant slowdowns observed in C_1 (see Figures 6.1 to 6.3). While C_1 introduces slowdowns as bad as 1/16 or 1/32, the C^* slowdowns are mostly contained within 1/2.

	TPC-H	TPC-DS	JOB
C_1 over H_1	1.03x	1.02x	0.62x
C^* over H^*	1.19x	1.03x	1.11x

Table 6.5: Average speedups of C_1 vs H_1 and C^* vs H^*.

Over the following two sections, we delve into possible reasons for the slowdown. As iterated in Section 2.3, optimization is only as good as cost estimates. Since the cost-based optimizer depends on cardinality estimates, poor estimates lead to suboptimal query plans, which are more likely to be found in non-reorderable joins. In addition, we examine latency introduced from an increased number of bridges in plans as another slowdown factor.

6.4.2 Reorderability of Joins

TPC-H and TPC-DS exhibit more slowdowns compared to JOB. A potential reason that the cost-based optimizer performs better for JOB is that JOB only contains inner joins, for which DuckDB v1.0 optimizes the join orders. In contrast, TPC-H and TPC-DS include other join types, such as semi- and anti-joins, which are not reorderable in DuckDB v1.0 (see 2.4.2). Hence, the estimated cardinality of such join types is not as accurate as inner joins'. The inaccuracy leads to the cost-based optimizer producing suboptimal query plans when these join types are involved.

We now analyze how non-reorderable joins contribute to the slowdown. 104 out of 452 variations of TPC-H and $7,554$ out of $18,156$ variations of TPC-DS have at least one non-reorderable join. Limiting to the queries with different query plans for heuristic (H^*) and cost-based (C^*) optimizers, we find that 8 out of 44 TPC-H queries and 2,164 out of 6,078 TPC-DS queries contain non-reorderable join.

Figure 6.11 categorizes queries by whether they are fully reorderable and reports the numbers of queries that become faster and slower. In TPC-H, most fully reorderable queries speed up, while all non-fully reorderable queries slow down. Also, there is a notable

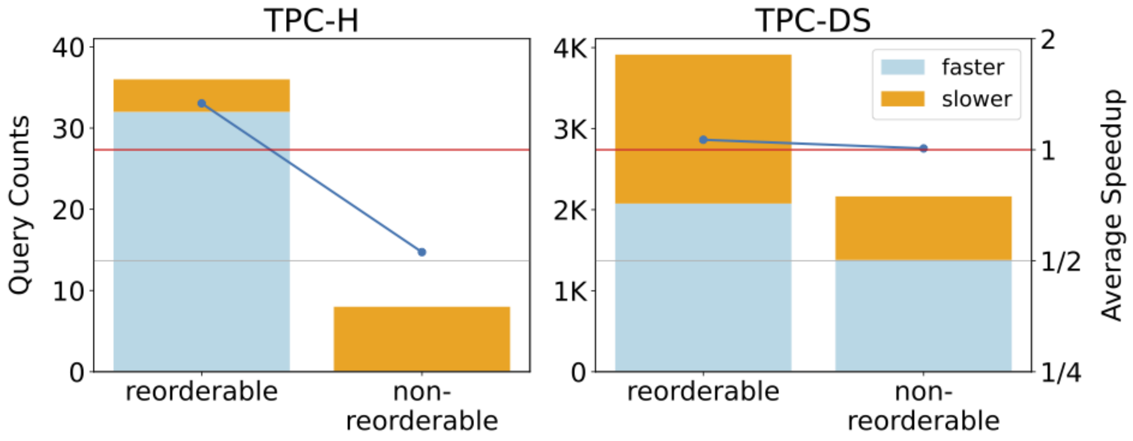


Figure 6.11: Counts and average speedup of queries, grouped by whether they include non-reorderable join types. The left y-axis shows the counts of queries that become slower (orange) and faster (blue). The right y-axis represents the average speedup of fully reorderable and non-fully reorderable queries.

difference in the average speedup: fully reorderable queries achieve a speedup of around 1.33x, while non-fully reorderable queries slow down to 0.55x speed.

In contrast, TPC-DS demonstrates a different pattern. Nearly half of fully reorderable queries, while about one-third of non-fully reorderable slow down. Interestingly, non-fully reorderable queries are more likely to speed up than fully reorderable queries. Moreover, the average speedup of fully reorderable queries is not significantly better than that of non-fully reorderable ones. This suggests that there are other factors contributing to the slowdown.

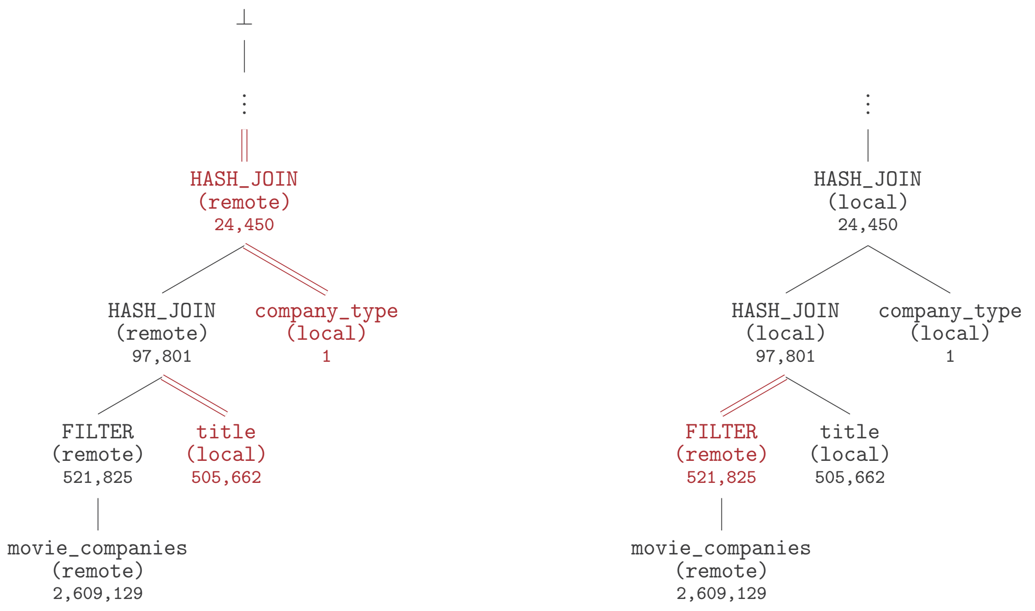
6.4.2.1 Examples

The following examples demonstrate how the cost-based optimizer behaves for queries with inner and non-inner joins.

Improvement: Inner Joins An example of a speedup is query 5c variation 5[1] from JOB. Its partial query plans by H^* and C^* are displayed in Figure 6.12.

H^* always opts to transfer relations on the right side – `title` and `company_type` – moving $505,663 = 505,662 + 1$ rows. Both hash joins shown in the partial plan are executed remotely. Hence, the results must later be downloaded to the client, adding $24,450$ rows to the cost.

[1]Table placement: `movie_info` at `local`, `movie_companies` at `remote`, `title` at `local`, `company_type` at `local`, and `info_type` at `local`



(a) Plan by H^*. **(b)** Plan by C^*.

Figure 6.12: Partial query plans of variation 5 of JOB query 5c.

C^* rather chooses to download the filtered result of `movie_companies`, which is estimated to have 521,825 rows. Although this choice moves more rows initially, it prepares all data on the client side, avoiding any future data transfers. As a result, C^* transfers less data overall than H^*.

Challenge: Semi- and Anti-Joins Among many join types that DuckDB v1.0 does not optimize, semi- and anti-joins are frequently found in benchmark queries. TPC-H query q20, variation 12[1]. is an example of how these joins impact the optimizer's decisions. Partial plans for the query are shown in Figure 6.13.

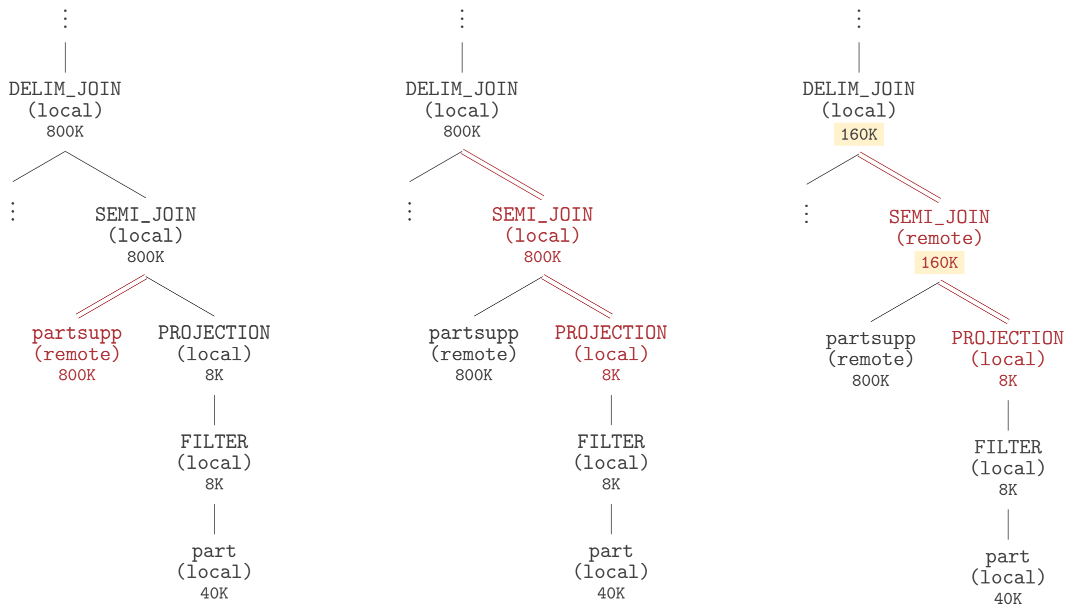
The query has a duplicate elimination join (delim join) that must be colocated with a delim scan (see Section 2.1.1). On delim join's left subtree (omitted in the plans), its matching delim scan is joined with a large local table `lineitem` (1.2M rows). Therefore, the colocated group follows the location of `lineitem`, and the delim join must be executed locally.

The optimizer must decide where to perform the left semi-join between `partsupp` and `PROJECTION`. The first option is to download `partsupp` with 800K rows (Figure 6.13a). The second option is to upload `PROJECTION` with 8K rows (Figure 6.13b). With the second

[1]Table placement: lineitem at `local`, nation at `remote`, part at `local`, partsupp at `remote`, and supplier at `remote`

option, the semi-join result (800K rows) has to be downloaded back to the client for the local execution of `DELILM_JOIN`. This brings the cost up to 880K. Comparing the two options, the cost-based optimizer takes the first option.

However, the semi-join cardinality is estimated inaccurately here. On the right subtree of the semi-join, there is a filter with selectivity of 20%. As explained in 2.1.1, semi-join selects rows from one relation if there is at least one matching row on the other relation. Therefore, when there is a filter that reduces the size of the matching relation, the semi-join cardinality should also decrease accordingly. Assuming uniform distribution as DuckDB does, a better estimate for the semi-join is 160K = 800K ×20%. Given the adjusted estimate, the cost-based optimizer would have chosen a better query plan (Figure 6.13c).



(a) Option 1: transfer `partsupp`. Plan chosen by C^*

(b) Option 2: transfer `PROJECTION` and `SEMI_JOIN`.

(c) Plan chosen by C^* given better semi-join cardinality estimation. Modified cardinality estimates are highlighted in yellow.

Figure 6.13: Partial query plans of variation 12 of TPC-H query q20.

6.4.3 Number of Bridges

Another possible reason for the slowdown is that the cost-based optimizer introduces more bridges than the heuristic optimizer for some queries. A bridge manages the transfer of data between the client and server. Even when the volume of data is small, each transfer adds latency to the overall execution time. As a result, if the optimizer generates a plan that introduces a lot of bridges, the latency may outweigh the benefits of transferring a smaller amount of data. The increased latency from additional bridges may explain why short-running queries are more susceptible to slowdown than long-running ones (see Figures 6.8 and 6.9). In other words, the latency can take up a large portion of the total execution time of short-running queries, making them slower. Moreover, in high-speed networks, latency becomes a dominant factor in network response time [54].

In this section, we do not focus specifically on short-running queries but study the general relationship between the number of bridges and speedup. We define *bridge delta* as

$$\text{bridge delta} \; = \; (\#\text{ bridges in } C^* \text{ plan}) - (\#\text{ bridges in } H^* \text{ plan}).$$

A negative value indicates that C^* uses fewer bridges than H^*, while a positive value means otherwise.

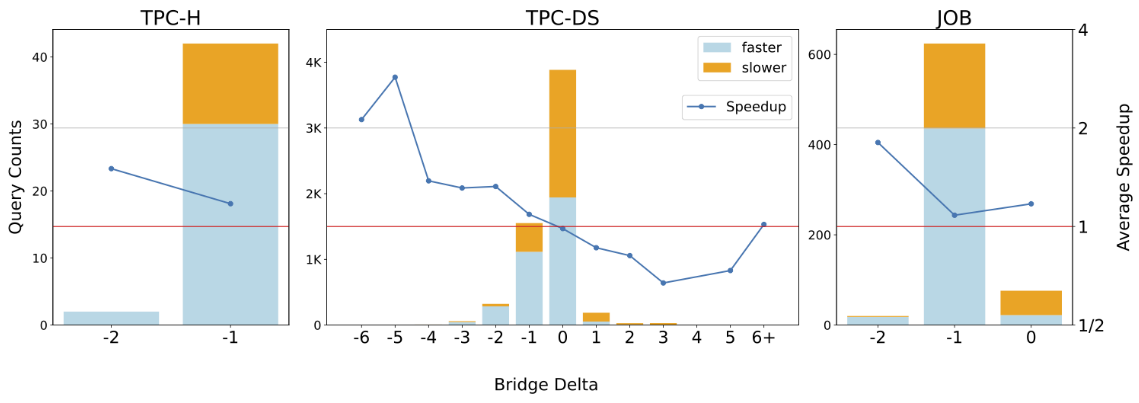


Figure 6.14: The x-axis is the bridge delta. A greater bridge delta means that the cost-based optimizer introduces more bridges. The query counts on the left y-axis show the number of queries that become slower (orange) and faster (blue). The right y-axis represents the average speedup of queries with a specific bridge delta.

Figure 6.14 depicts the number of queries that speed up and slow down per bridge delta and their average speedup. As the bridge delta increases, the ratio of queries that slow down also increases. For all benchmarks, there are little to no queries that slow down when the cost-based optimizer reduces the number of bridges by two or more. Additionally, there is a negative correlation between bridge delta and average speedup. The greater the bridge delta, the lower the average speedup. This trend is particularly evident in TPC-DS. With the bridge delta of -6 and -5, the average speed up is over 2x. It decreases until it meets the baseline of 1x when the bridge delta is 0 and falls further as the delta increases.

Interestingly, there is an increase in average speedup for bridge delta of 6+. This case is explained in more detail in the Section 6.4.3.1.

6.4.3.1 Examples

In this section, we present examples to analyze the relationship between bridge delta and speedup. We explore three cases: 1) query that slows down with more bridges, 2) query that slows down with fewer bridges, 3) query that speeds up with more bridges, and 4) query that speeds up with fewer bridges.

Case 1: Slowdown with More Bridges Query 71, variation 21 of TPC-DS as an example. This query is one of the short-running queries that experience regression described in Section 6.4.1. In this case, C^* slightly reduces both the estimated and actual number of rows transferred over the network but introduces two extra bridges in the plan. The query slows down to about 0.67x the original execution time. The slowdown implies that added latency from the extra bridges can outweigh the gains from transferring less data.

	# bridges	estimated cardinality	real cardinality	execution time (s)
H^*	3	57,133	86,373	0.2135
C^*	5	44,408	72,900	0.3123

Case 2: Slowdown with Fewer Bridges In rarer cases, some queries experience slowdown even when the cost-based optimizer introduces fewer bridges. One such example is TPC-DS query 83, variation 3. Based on the estimations, C^* transfers approximately 10% less data compared to H^*. However, the estimations are far off from the real cardinalities. The table below shows that the actual number of rows transferred in the C^* plan is double the estimated amount and more than three times what is transferred in the H^* plan. This means that the transfer time of data is significantly larger. As a result, the query takes almost twice the time to be executed. This example suggests that (1) when a significantly larger amount of data is transferred, the influence of latency becomes trivial, and (2) a large error in cardinality estimation leads to an inefficient plan, which has poor placement of bridges, which could lead to a slowdown even with a smaller number of bridges.

	# bridges	estimated cardinality	real cardinality	execution time (s)
H^*	4	166,148	93,360	0.2270
C^*	2	150,694	305,867	0.4333

Case 3: Speedup with More Bridges Some queries manage to speed up despite the cost-based optimizer introducing more bridges. For TPC-DS query 16, variation 5, C^* adds two bridges. However, the added latency is overcome by the fact that C^* transfers significantly less data than H^*. As shown in the table, the C^* plan transfers only 14% of the rows transferred by H^*. The speedup of the query is 1.31x.

	# bridges	estimated cardinality	real cardinality	execution time (s)
H^*	4	1,447,243	1,513,963	0.6490
C^*	6	150,804	218,841	0.4957

A more "extreme" example is TPC-DS query 2, variation 5, which speeds up even though the cost-based optimizer adds six extra bridges. Like the above example, the C^* plan transfers significantly less data – 55% of the rows moved by H^*. The time saved by transferring less data was enough to overcome the extra latency of the six new bridges.

	# bridges	estimated cardinality	real cardinality	execution time (s)
H^*	2	2,883,096	2,883,096	1.38775
C^*	8	2,210,058	1,586,326	1.12025

These examples support the previous observation that the influence of latency is marginal if there is a considerable difference in the amount of data transferred. The transfer time of large data will take up more of the execution time.

Case 4: Speedup with Fewer Bridges Finally, we look at an example where a query speeds up with a plan with fewer bridges. For TPC-DS query 10, variation 7, the cost-based optimizer requires only two bridges, while the heuristic optimizer requires seven. Moreover, there is a significant difference in the amount of data being transferred. H^* moves more than two million rows when C^* moves only 109,093. As a result, the query speeds up around 4.5x. The reduced number of bridges, combined with the smaller transfer amount, allows the query to be executed substantially faster.

	# bridges	estimated cardinality	real cardinality	execution time (s)
H^*	7	4,188,211	2,053,758	1.0235
C^*	2	2,881,788	109,093	0.2285

6.5 Heuristic vs Cost-Based Optimizer with DuckDB v1.1

Finally, we compare the performance of heuristic and cost-based optimizers with DuckDB v1.1 ($H_{v1.1}$ vs $C_{v1.1}$). The numbers of queries with different plans for $H_{v1.1}$ and $C_{v1.1}$ are:

- TPC-H: 38 different plans
- TPC-DS: 7,076 different plans
- JOB: 17,696 different plans

6.5.1 Results

	TPC-H	TPC-DS	JOB
speedup (count, avg)	29, 1.36x	3,746, 3.33x	10,314, 1.03x
slowdown (count, avg)	9, 0.78x	3,330, 0.64x	7,382, 0.97x
total (count, avg)	38, 1.22x	7,076, 2.06x	17,696, 1.01x

Table 6.6: Counts and average speedups of queries that speed up and slow down ($H_{v1.1}$ vs $C_{v1.1}$).

The relative speedups are shown in Figure 6.15. TPC-H queries have a similar level of speedup and slowdown as C^* (see Table 6.4 for comparison). 29 queries speed up with an average of 1.36x and 9 slow down with average of 0.78x (0.60x for C^*).

There is little improvement for JOB with $C_{v1.1}$. The average speedup is only 1.01x. Yet, from Figure 6.17, there are some queries ($<$ 4 seconds) that speed up more than 2x. The slowdowns are also mostly within 1/2x bound.

For TPC-DS, execution gets twice as fast on average. Just looking at the queries that become faster, the average speedup is 3.33x. Yet, the average slowdown (0.64x) had worsened from C^* (0.92x). From Figure 6.16, we can confirm that the magnitude of slowdown and speedup had increased. A cluster of long-running queries (green, query 64) experience significant speedup, some even reaching over 17x. Interestingly, shorter-running variants of the same query struggle from regression, some even experiencing 1/8x the execution speed (green cluster on the bottom left). This complex query has 16 inner joins, 1 mark join, 2 aggregates, and 1 CTE. The query does not have any non-reorderable operators, suggesting that there is another factor that makes the execution slow. Disregarding this cluster of queries that slow down, the average speedup increases from 2.06x to 2.70x.

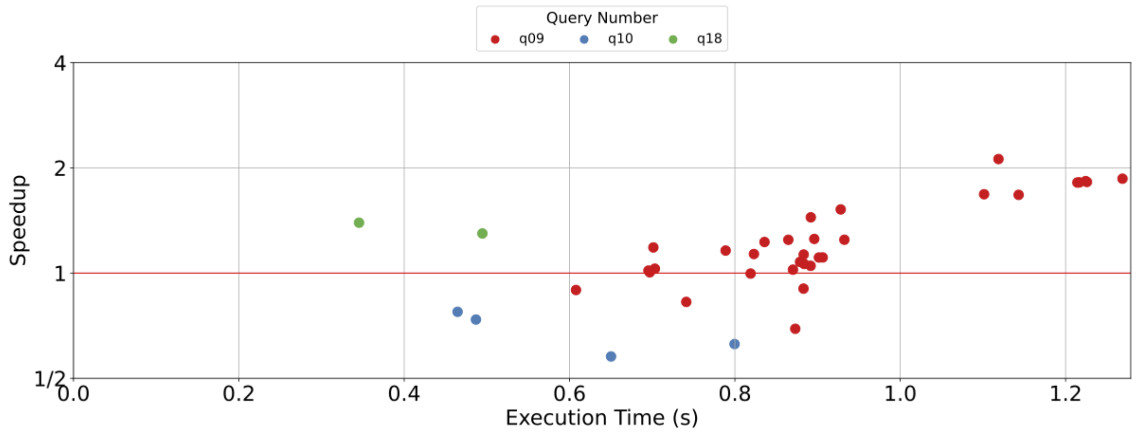


Figure 6.15: TPC-H speedup with DuckDB v1.1 ($H_{v1.1}$ vs $C_{v1.1}$).

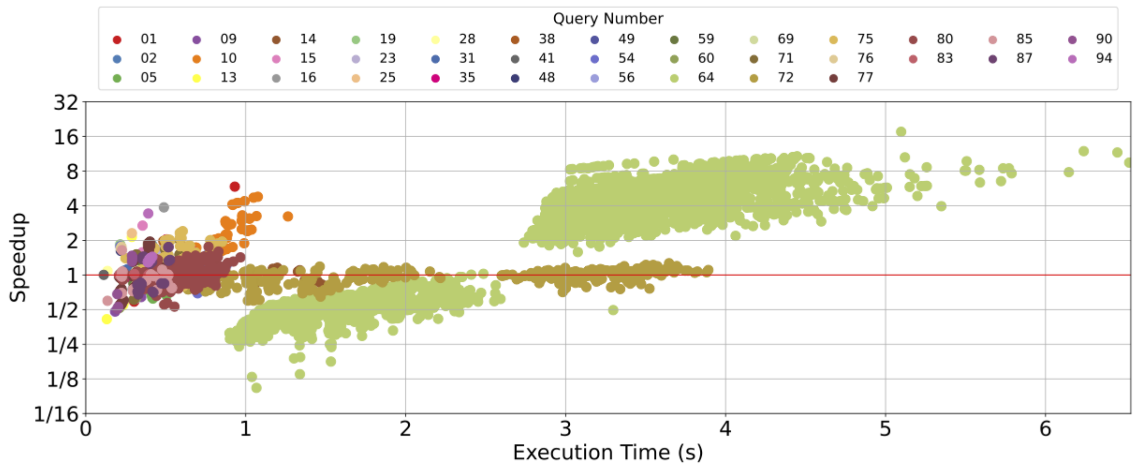


Figure 6.16: TPC-DS speedup with DuckDB v1.1 ($H_{v1.1}$ vs $C_{v1.1}$).

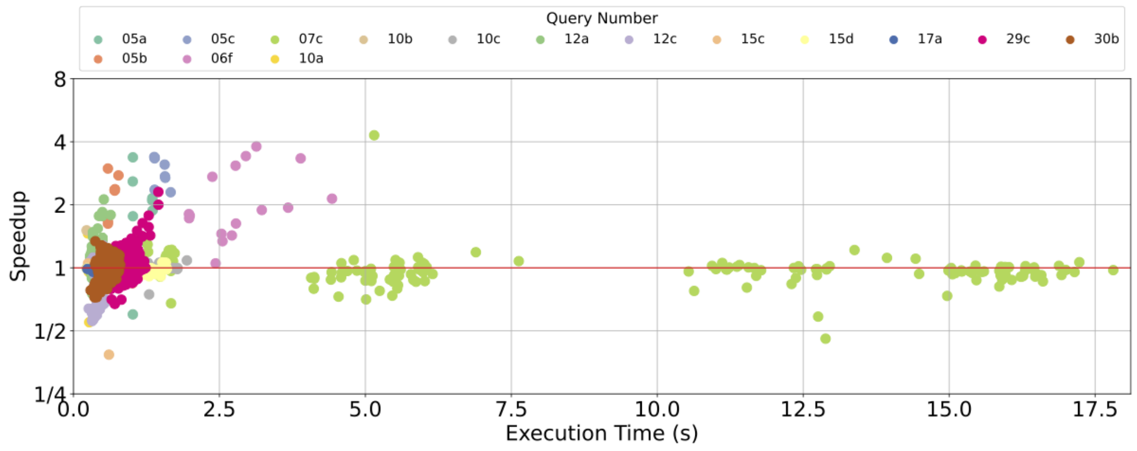


Figure 6.17: JOB speedup with DuckDB v1.1 ($H_{v1.1}$ vs $C_{v1.1}$).

On the upside, many short-running queries (< 1 second) speed up significantly, with some getting 8x faster. While there are some that slow down to 1/2x the speed, the general trend differs from what we observed in C^*, where most short-running queries slowed down.

6.6 Preliminary Results with Constant Latency Cost

We discussed the influence of latency in Section 6.4.3. To account for the latency in the cost model, we introduce a constant latency cost. This is the first step to accommodating network conditions in our cost model. The transfer cost is now calculated as follows:

$$T(child,\ from,\ to) = \begin{cases} child.result\_size + bridge\_cost & \text{if } from \neq to, \\ 0 & \text{otherwise.} \end{cases}$$

We set *bridge_cost* to 90,000. This is an ad-hoc value computed from a few individual query plans and their execution times. Let us take Case 1 as an example.

- H^* plan has 3 bridges and transfers 86,373 rows with an execution time of 0.2135 seconds.
- C^* plan has 5 bridges and transfers 72,900 rows with an execution time of 0.3123 seconds.

We derive two equations:

$$3 \cdot latency + \frac{86,373 \text{ rows}}{bandwidth} = 0.2135 \text{ seconds}$$

$$5 \cdot latency + \frac{72,900 \text{ rows}}{bandwidth} = 0.3123 \text{ seconds}$$

The equations yield $latency \approx 0.0536$ seconds and $bandwidth \approx 1,639,730$ rows per second. Since the unit of our cost function is cardinality, we have to convert the latency back to the number of rows. 0.0536 seconds is equivalent to the time it takes to transfer $87,889$ rows. For simplicity, we round to $90,000$. With the updated cost model, the cost-based optimizer will add a bridge only when it results in a net saving of over $90,000$ rows.

We compare the execution times of the heuristic optimizer and the cost-based optimizer with the adjusted cost model. The numbers of queries that have different plans than the heuristic optimizer's are:

- TPC-H: 165 different plans
- TPC-DS: 9,580 different plans
- JOB: 120,926 different plans

6.6.1 Results

	TPC-H	TPC-DS	JOB
speedup (count, avg)	95, 1.27x	5,528, 2.70x	103,169, 1.17x
slowdown (count, avg)	70, 0.88x	4,052, 0.65x	17,757, 0.92x
total (count, avg)	165, 1.11x	9,580, 1.83x	120,926 1.13x

Table 6.7: Counts and average speedups of queries that speed up and slow down ($H_{v1.1}$ vs $C'_{v1.1}$).

Table 6.7 shows that there are no additional improvements for TPC-H and TPC-DS with the adjusted cost model. In fact, the average speedups are lower than $C_{v1.1}$'s. Figures 6.18 and 6.19 also display similar patterns as $C_{v1.1}$. One difference for TPC-H is that there are more moderate-level speedups for short-running queries. For TPC-DS, the most evident change is the new cluster formed in the bottom left corner (mint, query 80), where some variants regress to 1/8x the speed. Although the same cluster exists in $H_{v1.1}$ vs $C_{v1.1}$ experiment, the slowdown of the cluster became more pronounced here.

JOB queries gain around 1.13x speedup, which is higher than in $C_{v1.1}$'s 1.01x. Figure 6.20 shows that a query 29a variant reaches over 50x speedup and some 07c variants reach 38x. The shorter-running queries (< 5 seconds) display a similar trend as in $C_{v1.1}$ (see Figure 6.17 for comparison); although speedup and slowdown range had increased, they are almost evenly distributed from 1/8x to 8x.

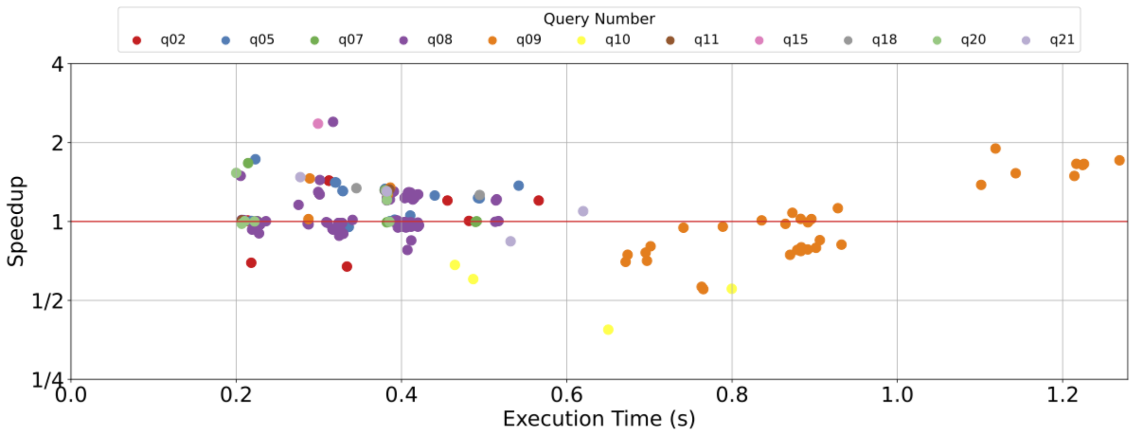


Figure 6.18: TPC-H speedup with the adjusted cost model ($H_{v1.1}$ vs $C'_{v1.1}$).

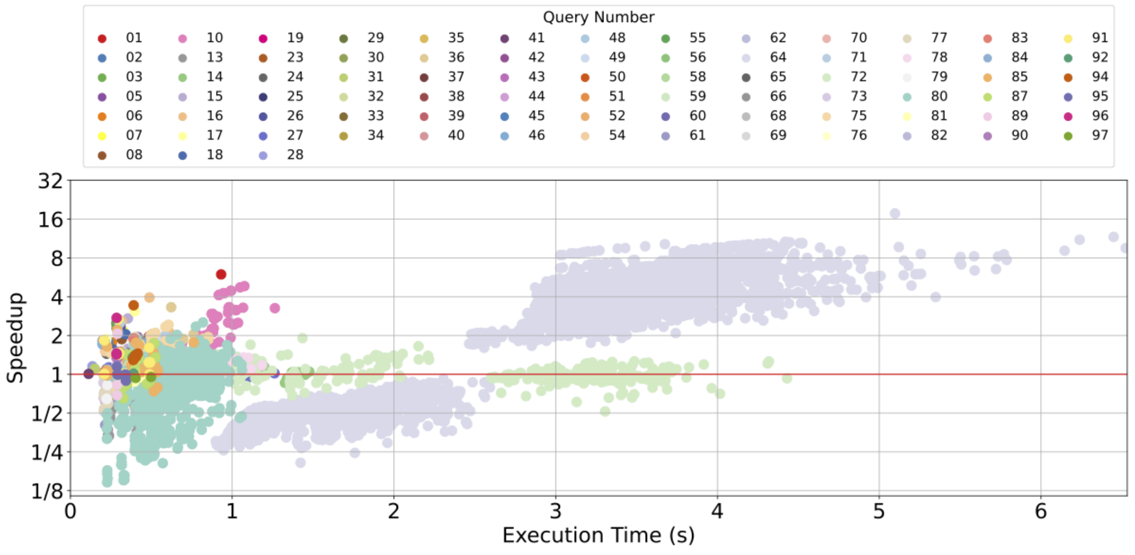


Figure 6.19: TPC-DS speedup with the adjusted cost model ($H_{v1.1}$ vs $C'_{v1.1}$).

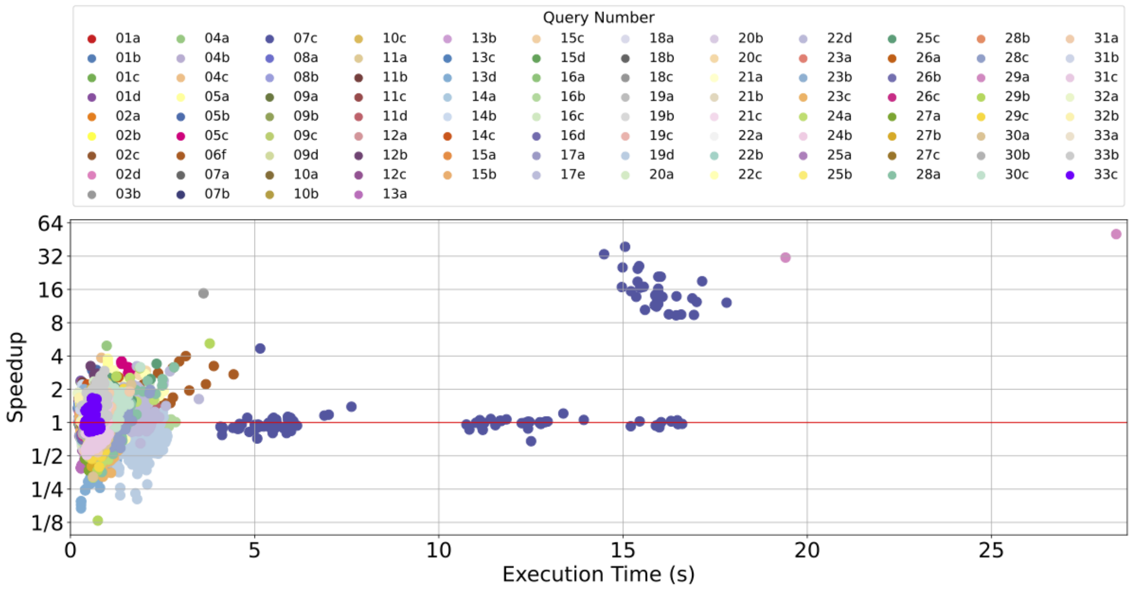


Figure 6.20: JOB speedup with the adjusted cost model ($H_{v1.1}$ vs $C'_{v1.1}$).

6.7 Summary

In this chapter, we compared the execution times of queries optimized by the heuristic and cost-based optimizers. We started by comparing the the heuristic and cost-based optimizers without DuckDB fixes applied (H_1 vs C_1, Section 6.2). Overall, C_1 was able to slightly outperform H_1 for TPC-H and TPC-DS but slowed down to 0.62x speed for JOB. Short-running queries experienced more slowdowns in greater magnitude than long-running queries, some JOB queries even reaching 1/32x slowdown. Without fixes on the DuckDB side, C_1 optimized with poor cardinality estimates, leading to suboptimal query plans.

Additionally, we examined how the fixes change the performance of the heuristic optimizer (H_1 vs H^*, Section 6.3). The purpose of the experiment was to observe the changes in the heuristic optimizer's baseline for the H^* vs C^* comparison. Overall, the baseline shifts upwards – 1.02x for TPC-H, 1.10x for TPC-DS, and 1.05x for JOB.

Next, we compared the heuristic and cost-based optimizers with all fixes (H^* vs C^*, Section 6.4). Overall, plans by C^* have faster execution times – 1.19x for TPC-H, 1.03x for TPC-DS, and 1.11x for JOB. Figure 6.10 shows that there are no significant slowdown in JOB; in fact, most queries that slow down lie very close to the baseline. Moreover, there are queries that reach around 4x speedup. TPC-H and TPC-DS exhibit a different pattern. These benchmarks have slow-down queries that take over twice the original execution time (Figures 6.8 and 6.9). Also, short-running queries are more prone to slowdown than long-running queries.

We laid out possible reasons for slowdown. First, we investigate how the reorderability of joins influences the execution time of queries. Unlike JOB, TPC-H and TPC-DS involve non-inner joins, which are not reorderable in DuckDB v1.0 and may have poor cardinality estimates. Figure 6.11 shows the average speedup of non-fully reorderable TPC-H queries is significantly lower compared to fully reorderable ones. On the other hand, the difference in average speedup is insignificant for TPC-DS. This suggests that there are other factors that contribute to the slowdown.

Another potential reason we explored is the increase in the number of bridges. Each added bridge introduces latency in the total execution time. Even if latency for each bridge is small, cumulated latency could slow down the overall query execution. For short-running queries, latency has a particularly negative impact, as it can take up a significant portion of the execution time. Figure 6.14 shows a negative correlation between the number of bridges and the average speedup. In other words, if the cost-based optimizer introduces more bridges than the heuristic optimizer, the average speedup tends to decrease.

By analyzing individual queries, we found the influence of latency depends on the amount of data being transferred. When the transfer amount is small, the latency has more influence over the execution time. Suppose the cost-based optimizer transfers only slightly less data than the heuristic optimizer but introduces more bridges. In that case, it has a high chance that the query will slow down if the optimizer introduces additional bridges (Case 1). On the other hand, if the cost-based optimizer transfers significantly less data, the impact of additional bridges becomes trivial, allowing a query to speed up despite a higher number of bridges (Case 3). It is important to note that if the error in cardinality estimation is big such that a significantly larger amount of data is transferred than expected, the effect of latency will, again, be trivial (Case 2).

Finally, we compared the performance of the optimizers using DuckDB v1.1 ($H_{v1.1}$ vs $C_{v1.1}$, Section 6.5). The cost-based optimizer achieved a higher average speedup than C^* for TPC-H and TPC-DS. TPC-DS especially had good improvements, where some queries speed up 17x. Moreover, unlike the trend in C^*, where most short-running queries slow down, $C_{v1.1}$ had many short-running queries that experience significant speedup. However, a cluster of queries does show a substantial slowdown, which warrants further investigation.

To factor in network latency in the cost model, we introduced a fixed latency cost per bridge ($H_{v1.1}$ vs $C'_{v1.1}$, Section 6.6). This adjustment did not produce significant changes for TPC-H and TPC-DS. A notable change is that for TPC-H, $C'_{v1.1}$ introduces more moderate-level speedups for short-running queries. The new cost model improves the performance of JOB queries; the average speedup of the cost-based optimizer increases from 1.01x to 1.13x. Moreover, some queries reach over 38x or 50x speedup. For all benchmarks, specific causes for slowdowns and speedups remain to be studied.

7

Limitation and Future Work

We have developed the new cost-based hybrid query optimizer and evaluated its performance against the existing heuristic optimizer. This chapter discusses the limitations of our approach and explores potential future work to improve the hybrid query optimizer. Additionally, we propose ideas for further analysis and experiments to gain more insights.

7.1 Understanding Slowdowns

While the cost-based optimizer has improved the overall query execution time, performance issues remain for some queries. We hypothesized that the reorderability of joins and the number of bridges influence the execution time and found correlations with slowdowns. However, we currently lack the analysis to quantify their contributions to slowdowns.

7.1.1 Reorderability of Joins

One approach to evaluating how join reorderability impacts the cardinality estimates is by comparing the estimation errors for fully and non-fully reorderable queries. This experiment can numerically show the difference in estimation quality between reorderable and non-reorderable joins. Furthermore, we can make this experiment more granular by gathering the estimation errors per the number of non-reorderable joins in a query. This would also allow us to observe whether having many non-reorderable joins explodes the estimation errors.

The experiments with DuckDB v1.1, which supports join order optimization for left semi- and anti-joins, show overall performance improvements for TPC-H and TPC-DS. In particular, many TPC-DS queries speed up significantly. It could be helpful to analyze how the performance changed for queries that used to be non-fully reorderable. We observed that many variants of one query slow down significantly in this experiment but do not contain any non-reorderable joins. We will discuss this case in more detail in Section 7.1.3.

7.1.2 Number of Bridges

We have identified a correlation between bridge delta and the number of slowdowns, as well as the average speedup. As the bridge delta increases, the likelihood of slowdown grows, and the average speedup decreases. Moreover, our findings suggest that the influence of bridges depends on the size of the data being transferred. Smaller data transfer amplifies the effect of latency on query execution time, while the effect becomes trivial in larger data transfer.

An important question is at what point latency starts becoming influential. To answer this question, we computed the break-even point between latency and transfer time and introduced it as a constant latency cost in the cost model. With the new cost model, the optimizer favors adding a bridge only if the transfer size exceeds the break-even. The initial results with DuckDB v1.1 show that the new cost model improves the performance of the cost-based optimizer for JOB. However, TPC-H and TPC-DS do not benefit from the new model.

Further analysis could yield more insights from this experiment. For example, we could examine how the number of bridges in plans changes with the new cost model. We can measure how conservative the optimizer gets with the added latency. Additionally, we could investigate why some JOB queries speed up significantly while some TPC-H and TPC-DS queries slow down. Moreover, a direct comparison between $C_{v1.1}$ and $C'_{v1.1}$ may provide more insights into the impact of the new cost model.

We can also devise a more systematic approach to compute latency cost. Now, latency cost is an ad-hoc value estimated from a few queries and their execution times. We can improve this by incorporating more cases into consideration. Yet, real-world network conditions fluctuate frequently. So, to maintain an accurate reflection of the real network latency in the cost model, we would need to adjust the cost dynamically rather than have it fixed. This is discussed further in Section 7.3.

7.1.3 Other Factors

There could be more factors that affect the performance of the cost-based optimizer that we have not found or analyzed so far. Here, we list a few possible avenues to discover unforeseen trends.

First, we can study a few sensitive queries that are especially prone to slow down or speed up. Such examples are TPC-DS queries 64 and 80. In $H_{v1.1}$ vs $C_{v1.1}$ comparison, query 64 variants are split into two clusters: one that speeds up significantly and one that regresses. The only differentiating factor between the two clusters is table placement. We could investigate if there is a shared pattern in table placements within each cluster.

TPC-DS query 80 generally becomes slower with the new cost model with fixed latency cost ($C'_{v1.1}$). In $H_{v1.1}$ vs $C_{v1.1}$ comparison, the query 80 cluster spans evenly from 1/2x to 2x speedup. In other words, a similar number of variants slow down and speed up. In contrast, with $C'_{v1.1}$, the speedup threshold remains at a similar level (2x), while the slowdown magnitude increases to 1/8x. The differentiating factor between the two experiments is the cost model. We could examine why adding a fixed latency cost makes this particular query slower.

Other factors we have not yet formally analyzed are the new DuckDB v1.1 changes beyond join order optimization. The two main relevant changes are the new implementations of HLL and left-right optimization. These two changes cause structural changes in query plans, *i.e.,* the shape of query plans may differ from DuckDB v1.0's. We can conduct an experiment to observe the baseline change between DuckDB v1.1 and v1.0.

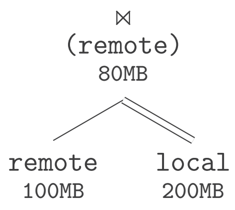
7.2 Cardinality Estimation: Ideal Scenario

Our current cost model is closely tied to the cardinality estimates provided by DuckDB. Hence, significant errors in the estimates are detrimental to the query plans generated by the cost-based optimizer. However, as we discovered throughout the thesis, there are multiple issues related to cardinality estimation in DuckDB. One of the challenges is that DuckDB does not support reordering for some join types; thus, precise join size estimations are unavailable for them. Even with DuckDB v1.1, certain join types like right semi-join, right anti-join, and outer joins are not supported.

Furthermore, DuckDB's distinct count statistics rely on HLL from sampled data (30% for integer columns and 10% for others). Cardinality estimations based on small samples, however, are prone to error as stated in [37]. Other limitations are discussed in more detail in Section 2.4.2.

Given these limitations, it is hard to assess the full potential of the cost-based query optimizer. To bypass them, we could impose an *ideal scenario*, where the cardinalities of all operators are known a priori [1]. In this setup, the optimizer would be able to generate

[1]Techniques for extracting true cardinalities from query plans have been proposed in [71].



plans that have truly globally minimal costs. This experiment would allow us to observe the maximum possible speedup achievable by the cost-based optimizer. Additionally, it would allow us to investigate the impact of estimation errors further. We could learn how estimation errors affect different types of queries by comparing the performance with perfect cardinalities to that with estimated cardinalities. These insights will be valuable for future improvements to the optimizer.

7.3 Cost Model

The performance can be further improved by extending the cost model. Currently, the cost of the plan is defined as the amount of data transferred over the network, which we approximate with the cardinality. However, we can utilize the schema information to estimate the **physical size of relations** (in bytes).

Moreover, as illustrated in an example in Section 5.1, taking advantage of **network and CPU asymmetry** can produce more efficient query plans. To this end, we can extend the cost model to include network latency, bandwidth, and CPU instructions per second (IPS).

These modifications can translate the logical cost (cardinality) into a physical one (time). The parameters of the new cost model will look like:

- $W_{latency}$, unit in seconds
- $W_{bandwidth}$, unit in seconds/byte
- W_{cpu}, unit in seconds/instruction

Then, given the data size in bytes, the cost function is:

$$cost = W_{latency} \times (\#\ bridges) + W_{bandwidth} \times (\#\ bytes) + W_{cpu} \times (\#\ instrcutions).$$

As a concrete example, let us consider a candidate query plan below. This plan chooses to transfer the local table (200MB) and remotely execute the join operation (5 million instructions). Say the latency is 10ms, the bandwidth is 100MB/sec, and the server CPU has 200K MIPS (million instructions per second). The weights are: $W_{latency} = 10ms$, $W_{bandwidth} = 10ms/MB$, and $W_{cpu} = 0.005ms/MI$. The cost is then

$$10ms + 10ms/MB \times 100MB + 0.005ms/MI \times 5MI = 1010.025\ ms.$$

The extended cost model has an advantage over the current model, which relies solely on logical estimates. It also goes beyond including a constant latency cost as in $C'_{v1.1}$. By incorporating factors like bandwidth and IPS, on top of latency, the extended model provides costs that better reflect real physical constraints. Furthermore, the cost represents the expected query execution time, which we ultimately aim to optimize. Hence, the extended model enables us to directly optimize the execution time.

Another benefit of the extended cost model is that it resolves the question raised in the previous section regarding the influence of latency relative to the transfer size (Section 7.1.2). The model explicitly includes latency as a cost factor ($W_{latency}$), along with the time required to transfer data ($W_{bandwidth} \times (\#\ bytes)$). It captures how the latency and data size interact and how much they attribute to the expected execution time.

Furthermore, the cost model can be enhanced to optimize for the **monetary cost**. The main cost factors of cloud-based services are computation time and egress cost. We can incorporate the hourly rate for computation and per-byte cost of outbound data transfer in the cost model. Doing so would reduce the cloud expenses charged to MotherDuck. To still guarantee timely execution, we can find a balance between the two metrics: *monetary cost* and *execution time*, optimizing for lower expenses while maintaining an acceptable timeframe for query processing. The trade-off between the two metrics has been explored in [5] and discussed in Section 3.2.2.2.

Our DP framework is extensible. It can accommodate the additional cost factors as new terms in the cost function. However, the challenges are 1) estimating the sizes of relations and 2) measuring the weights of the new cost factors. We need careful computation estimating the sizes of variable-length and nested data types (*e.g.,* VARCHAR and MAP). Moreover, the network conditions change frequently, meaning we need a way to monitor the network and adjust the cost model accordingly. As a proof of concept, we can run benchmarks in a controlled environment with fixed latency, bandwidth, and known IPS, then compare the execution times with the extended cost model against the current one.

8

Conclusion

In this work, we set out to make MotherDuck's hybrid query optimizer cost-based (**RQ 1**). The role of MotherDuck's optimizer is to assign execution sites – either `local` or `remote`– for each operator in a query. The existing query optimizer relies on a simple heuristic, which always transfers the right-hand side relation to the left. However, this greedy approach often leads to a suboptimal query plan, transferring more tuples than necessary. To find a *globally optimal* plan, we structured this task as a DP optimization problem. Inspired by the concept of *interesting order* in System-R optimizer [27], we retain plans for each *interesting site*. Our approach retains a plan even if it initially incurs a higher cost because it may provide a better data placement for subsequent operations and lead to a smaller overall cost.

The cost is defined as the amount of data transferred across the network, approximated as cardinality. Since the cost model depends on cardinality, every operator in a query plan needs a cardinality estimate according to DuckDB's statistics. However, our initial evaluation revealed several issues related to cardinalities on both the DuckDB and MotherDuck sides. Two critical issues are 1) loss of cardinality information following DuckDB's join order optimization (due to subsequent DuckDB optimization rules or MotherDuck's faulty serialization) and 2) neglecting table filter selectivity in remote table cardinality estimation (resulting in potentially suboptimal join orders). To address these, we implemented fixes to preserve cardinality estimates and better utilize DuckDB's statistics, which are now merged in MotherDuck and DuckDB code.

The experimental results with DuckDB v1.0 demonstrate that the cost-based optimizer improves the execution time; JOB saw a 1.11x speedup, TPC-DS 1.03x, and TPC-H 1.19x. Both TPC-H and TPC-DS exhibit a trend where long-running queries gain speedups, while short-running ones tend to slow down. JOB queries, on the other hand, achieve speedups regardless of their original execution time, with only a few slowdowns. Two possible causes of slowdowns were investigated: non-reorderability of joins and network latency. The non-reorderable joins in TPC-H and TPC-DS may result in inaccurate cardinality estimates, leading to poor query plans by the cost-based optimizer. We observed a negative correlation between the number of bridges and speedups. Through individual case studies, we found that latency is especially impacts short-running queries, where latency often dominates execution time.

The latest DuckDB version, v1.1, increased the average speedup of the cost-based optimizer to 1.22x for TPC-H and 2.06x for TPC-DS. For TPC-DS, we observed many long-running queries with substantial speedup (17x), and some short-running queries achieving speedups up to 8x. However, the average speedup for JOB declined to 1.01x, with noticeable slowdowns among short-running queries. The preliminary results using a cost model with a constant latency cost show improved performance of the cost-based optimizer for JOB queries, raising the average speedup to 1.13x. The adjusted cost model was particularly effective for long-running queries in JOB, where some reach over 38x or even 50x speedup. Yet, it did not provide significant changes for TPC-H and TPC-DS. The impact of DuckDB v1.1 updates and latency cost remains a topic for further investigation.

References

[1] Tom Ebergen. *Join Order Optimization with (Almost) No Statistics*. Master's thesis, Vrije Universiteit Amsterdam, 2022. www.cwi.nl/boncz/msc/2022-TomEbergen.pdf. v, 20, 22, 24

[2] M. Tamer Özsu and Patrick Valduriez. *Principles of Distributed Database Systems*. Springer Publishing Company, Incorporated, Gewerbestrasse 11, 6330 Cham, Switzerland, 4th edition, 2020. v, 28, 29, 42

[3] Stefan Dessloch, Theo Härder, Nelson Mattos, Bernhard Mitschang, and Joachim Thomas. **Advanced data processing in KRISYS: modeling concepts, implementation techniques, and client/server issues**. *The VLDB Journal*, **7**(2):79–95, May 1998. v, 35, 36

[4] Viktor Leis and Maximilian Kuschewski. **Towards cost-optimal query processing in the cloud**. *Proc. VLDB Endow.*, **14**(9):1606–1612, May 2021. v, 45

[5] Zisis Karampaglis, Anastasios Gounaris, and Yannis Manolopoulos. **A Bi-objective Cost Model for Database Queries in a Multi-cloud Environment**. In *Proceedings of the 6th International Conference on Management of Emergent Digital EcoSystems*, MEDES '14, page 109–116, New York, NY, USA, 2014. Association for Computing Machinery. vi, 46, 97

[6] Donald Kossmann, Michael J. Franklin, Gerhard Drasch, and Wig Ag. **Cache investment: integrating query optimization and distributed data placement**. *ACM Trans. Database Syst.*, **25**(4):517–558, December 2000. vi, 47

[7] Hemn Barzan Abdalla. **A brief survey on big data: technologies, terminologies and data-intensive applications**. *Journal of Big Data*, **9**(1), November 2022. 1

[8] Amir Gandomi and Murtaza Haider. **Beyond the hype: Big data concepts, methods, and analytics**. *International Journal of Information Management*, **35**(2):137–144, 2015. 1

[9] Jordan Tigani. **Big Data is Dead**, February 2023. Accessed: 2024-09-22. 1

[10] Abraham Silberschatz, Henry F. Korth, and S. Sudarshan. *Database System Concepts*. McGraw-Hill, 2 2019. 5, 6, 8, 9, 10

[11] E. F. Codd. **A relational model of data for large shared data banks**. *Commun. ACM*, **13**(6):377–387, jun 1970. 5, 13

[12] Priti Mishra and Margaret H. Eich. **Join processing in relational databases**. *ACM Comput. Surv.*, **24**(1):63–113, mar 1992. 8

[13] G. Graefe and W.J. McKenna. **The Volcano optimizer generator: extensibility and efficient search**. In *Proceedings of IEEE 9th International Conference on Data Engineering*, pages 209–218, 1993. 10, 16

[14] Peter A. Boncz, Marcin Zukowski, and Niels Nes. **MonetDB/X100: Hyper-Pipelining Query Execution.** In *CIDR*, pages 225–237. www.cidrdb.org, 2005. 10

[15] Surajit Chaudhuri. **An overview of query optimization in relational systems**. *Proceedings of the seventeenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, 1998. 11, 15, 16

[16] 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 23–34, New York, NY, USA, 1979. Association for Computing Machinery. 13, 15, 16, 19

[17] 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. 14

[18] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. **How good are query optimizers, really?** *Proc. VLDB Endow.*, **9**(3):204–215, nov 2015. 14, 23, 24, 69

[19] M. Muralikrishna and David J. DeWitt. **Equi-depth multidimensional histograms**. In *ACM SIGMOD Conference*, 1988. 14

[20] Viswanath Poosala and Yannis E. Ioannidis. **Selectivity Estimation Without the Attribute Value Independence Assumption**. In *Very Large Data Bases Conference*, 1997. 14

[21] Meihui Zhang, Marios Hadjieleftheriou, Beng Chin Ooi, Cecilia M. Procopiuc, and Divesh Srivastava. **Automatic discovery of attributes in relational databases**. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, SIGMOD '11, page 109–120, New York, NY, USA, 2011. Association for Computing Machinery. 14

[22] Michael J. Freitag and Thomas Neumann. **Every Row Counts: Combining Sketches and Sampling for Accurate Group-By Result Estimates**. In *Conference on Innovative Data Systems Research*, 2019. 14

[23] 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 268–277, New York, NY, USA, 1991. Association for Computing Machinery. 14, 23

[24] Guy M. Lohman, C. Mohan, Laura M. Haas, Dean Daniels, Bruce G. Lindsay, Patricia G. Selinger, and Paul F. Wilms. *Query Processing in R**, pages 31–47. Springer Berlin Heidelberg, Berlin, Heidelberg, 1985. 16, 38, 42

[25] L. M. Haas, J. C. Freytag, G. M. Lohman, and H. Pirahesh. **Extensible query processing in starburst**. *SIGMOD Rec.*, **18**(2):377–388, jun 1989. 16

[26] 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 539–552, New York, NY, USA, 2008. Association for Computing Machinery. 16, 18

[27] Edgar F Codd. **A relational model of data for large shared data banks**. *Communications of the ACM*, **13**(6):377–387, 1970. 16, 61, 99

[28] 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 930–941. VLDB Endowment, 2006. 16

[29] J. Rao, B. Lindsay, G. Lohman, H. Pirahesh, and D. Simmen. **Using EELs, a practical approach to outerjoin and antijoin reordering**. In *Proceedings 17th International Conference on Data Engineering*, pages 585–594, 2001. 18

[30] Mark Raasveldt and Hannes Mühleisen. **DuckDB: an Embeddable Analytical Database**. In *Proceedings of the 2019 International Conference on Management of Data*, SIGMOD '19, page 1981–1984, New York, NY, USA, 2019. Association for Computing Machinery. 19

[31] Mark Raasveldt and Hannes Mühleisen. **Don't hold my data hostage: a case for client protocol redesign**. *Proc. VLDB Endow.*, **10**(10):1022–1033, jun 2017. 19

[32] Mark Raasveldt and Hannes Mühleisen. **Data Management for Data Science - Towards Embedded Analytics**. In *10th Conference on Innovative Data Systems Research, CIDR 2020, Amsterdam, The Netherlands, January 12-15, 2020, Online Proceedings*. www.cidrdb.org, 2020. 19

[33] DuckDB. **Why DuckDB**. https://duckdb.org/why_duckdb. Accessed: 2024-08-30. 19

[34] Viktor Leis, Peter Boncz, Alfons Kemper, and Thomas Neumann. **Morsel-driven parallelism: a NUMA-aware query evaluation framework for the many-core age**. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, page 743–754, New York, NY, USA, 2014. Association for Computing Machinery. 19

[35] RJ Atwal, Peter Boncz, Ryan Boyd, Antony Courtney, Till Döhmen, Florian Gerlinghoff, Jeff Huang, Joseph Hwang, Raphael Hyde, Elena Felder, Jacob Lacouture, Yves LeMaout, Boaz Leskes, Yao Liu, Dan Perkins, Tino Tereshko, Jordan Tigani, Nick Ursa, Stephanie Wang, and Yannick Welsch. **MotherDuck: DuckDB in the cloud and in the client**. https://www.cidrdb.org/cidr2024/papers/p46-atwal.pdf, 2022. 19, 30

[36] 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):889–900, apr 2017. 19

[37] Moses Charikar, Surajit Chaudhuri, Rajeev Motwani, and Vivek Narasayya. **Towards estimation error guarantees for distinct values**. In *Proceedings of the Nineteenth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '00, page 268–279, New York, NY, USA, 2000. Association for Computing Machinery. 25, 95

[38] Thomas Neumann, Viktor Leis, and Alfons Kemper. **The Complete Story of Joins (in HyPer)**. In *Datenbanksysteme für Business, Technologie und Web*, 2017. 26

[39] Michael J. Franklin, Björn Thór Jónsson, and Donald Kossmann. **Performance tradeoffs for client-server query processing**. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, SIGMOD '96, page 149–160, New York, NY, USA, 1996. Association for Computing Machinery. 33

[40] Nick Roussopoulos and H. Kang. **Principles and techniques in the design of ADMS+((advanced data-base management system))**. *Computer*, **19**:19 – 23, 1986/// 1986. 34

[41] A. Delis and N. Roussopoulos. **Performance comparison of three modern DBMS architectures**. *IEEE Transactions on Software Engineering*, **19**(2):120–138, 1993. 34

[42] W. B. Rubenstein, M. S. Kubicar, and R. G. G. Cattell. **Benchmarking simple database operations**. *SIGMOD Rec.*, **16**(3):387–394, December 1987. 34

[43] V. Kanitkar and A. Delis. **Site selection for real-time client request handling**. In *Proceedings. 19th IEEE International Conference on Distributed Computing Systems (Cat. No.99CB37003)*, pages 298–305, 1999. 35

[44] Tobias Mayr and Praveen Seshadri. **Client-site query extensions**. In *Proceedings of the 1999 ACM SIGMOD International Conference on Management of*

Data, SIGMOD '99, page 347–358, New York, NY, USA, 1999. Association for Computing Machinery. 36

[45] Mary Tork Roth and Peter M. Schwarz. **Don't Scrap It, Wrap It! A Wrapper Architecture for Legacy Data Sources**. In *Proceedings of the 23rd International Conference on Very Large Data Bases*, VLDB '97, page 266–275, San Francisco, CA, USA, 1997. Morgan Kaufmann Publishers Inc. 37

[46] Laura M. Haas, Donald Kossmann, Edward L. Wimmers, and Jun Yang. **Optimizing Queries Across Diverse Data Sources**. In *Proceedings of the 23rd International Conference on Very Large Data Bases*, VLDB '97, page 276–285, San Francisco, CA, USA, 1997. Morgan Kaufmann Publishers Inc. 37

[47] Manuel Rodríguez-Martínez and Nick Roussopoulos. **MOCHA: a self-extensible database middleware system for distributed data sources**. *SIGMOD Rec.*, **29**(2):213–224, May 2000. 38

[48] Sudarshan Chawathe, Hector Garcia Molina, Joachim Hammer, Kelly Ireland, Yannis Papakonstantinou, Jeffrey Ullman, and Jennifer Widom. **The TSIMMIS Project: Integration of Heterogeneous Information Sources**. In *Proceedings of the 10th Meeting of the Information Processing Society of Japan (IPSJ 1994)*, pages 7–18, 1994. 38

[49] A. Tomasic, L. Raschid, and P. Valduriez. **Scaling heterogeneous databases and the design of Disco**. In *Proceedings of 16th International Conference on Distributed Computing Systems*, pages 449–457, 1996. 38

[50] Vassilis Papadimos and David Maier. **Mutant query plans**. *Information and Software Technology*, **44**(4):197–206, 2002. 39

[51] B. Paul Jenq, Darrell Woelk, Won Kim, and Wan-Lik Lee. **Query processing in distributed ORION**. In François Bancilhon, Constantino Thanos, and Dennis Tsichritzis, editors, *Advances in Database Technology — EDBT '90*, pages 169–187, Berlin, Heidelberg, 1990. Springer Berlin Heidelberg. 39

[52] Lothar F. Mackert and Guy M. Lohman. **R* optimizer validation and performance evaluation for local queries**. In *Proceedings of the 1986 ACM SIGMOD International Conference on Management of Data*, SIGMOD '86, page 84–95, New York, NY, USA, 1986. Association for Computing Machinery. 42, 43

[53] Michael Stonebraker. *The design and implementation of distributed INGRES*, page 187–196. Addison-Wesley Longman Publishing Co., Inc., USA, 1986. 42

[54] Jesper M. Johansson. **On the impact of network latency on distributed systems design**. *Inf. Technol. and Management*, **1**(3):183–194, July 2000. 43, 83

[55] Salvatore T. March and Sangkyu Rho. **Allocating Data and Operations to Nodes in Distributed Database Design**. *IEEE Trans. on Knowl. and Data Eng.*, **7**(2):305–317, April 1995. 43

[56] W. Richard Stevens. *TCP/IP illustrated (vol. 1): the protocols*. Addison-Wesley Longman Publishing Co., Inc., USA, 1993. 44

[57] Yanif Ahmad and Uğur Çetintemel. **Network-aware query processing for stream-based applications**. In *Proceedings of the Thirtieth International Conference on Very Large Data Bases - Volume 30*, VLDB '04, page 456–467. VLDB Endowment, 2004. 44

[58] Avrilia Floratou, Jignesh M. Patel, Willis Lang, and Alan Halverson. **When Free Is Not Really Free: What Does It Cost to Run a Database Workload in the Cloud?** In Raghunath Nambiar and Meikel Poess, editors, *Topics in Performance Evaluation, Measurement and Characterization*, pages 163–179, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg. 45

[59] Wentao Wu, Yun Chi, Shenghuo Zhu, Junichi Tatemura, Hakan Hacigümüs, and Jeffrey F. Naughton. **Predicting query execution time: Are optimizer cost models really unusable?** In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*, pages 1081–1092, 2013. 46

[60] Yahya Slimani, Faiza Najjar, and Najla Mami. **An Adaptive Cost Model for Distributed Query Optimization on the Grid**. In Robert Meersman, Zahir Tari, and Angelo Corsaro, editors, *On the Move to Meaningful Internet Systems 2004: OTM 2004 Workshops*, pages 79–87, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg. 46

[61] Luis L. Perez and Christopher M. Jermaine. **History-aware query optimization with materialized intermediate views**. In *2014 IEEE 30th International Conference on Data Engineering*, pages 520–531, 2014. 48

[62] Nicholas L. Farnan, Adam J. Lee, Panos K. Chrysanthis, and Ting Yu. **PAQO: Preference-aware query optimization for decentralized database systems**. In *2014 IEEE 30th International Conference on Data Engineering*, pages 424–435, 2014. 49

[63] Simon Pierre Dembele, Ladjel Bellatreche, and Carlos Ordonez. **Towards Green Query Processing - Auditing Power Before Deploying**. In *2020 IEEE International Conference on Big Data (Big Data)*, pages 2492–2501, 2020. 49

[64] Binglei Guo, Jiong Yu, Bin Liao, Dexian Yang, and Liang Lu. **A green framework for DBMS based on energy-aware query optimization and energy-efficient query processing**. *Journal of Network and Computer Applications*, **84**:118–130, 2017. 49

[65] Yi Zhou, Shubbhi Taneja, Chaowei Zhang, and Xiao Qin. **GreenDB: Energy-Efficient Prefetching and Caching in Database Clusters**. *IEEE Transactions on Parallel and Distributed Systems*, **30**(5):1091–1104, 2019. 49

[66] Till Döhmen. **Introducing the embedding() function: Semantic search made easy with SQL!**, August 2024. Accessed: 2024-10-21. 52

[67] Till Döhmen. **Introducing the prompt() Function: Use the Power of LLMs with SQL!**, October 2024. Accessed: 2024-10-21. 52

[68] Warren B. Powell. *Approximate Dynamic Programming*. Wiley, 8 2011. 63

[69] **TPC-H Homepage**. `https://www.tpc.org/tpch/`. Accessed: 2024-05-30. 69

[70] **TPC-DS Homepage**. `https://www.tpc.org/tpcds/`. Accessed: 2024-05-30. 69

[71] Surajit Chaudhuri, Vivek Narasayya, and Ravi Ramamurthy. **Exact cardinality query optimization for optimizer testing**. *Proc. VLDB Endow.*, **2**(1):994–1005, August 2009. 95

Appendix

```sql
SELECT
    o_orderpriority,
    count(*) AS order_count
FROM
    orders
WHERE
    o_orderdate >= CAST('1993-07-01' AS date)
    AND o_orderdate < CAST('1993-10-01' AS date)
    AND EXISTS (
        SELECT
            *
        FROM
            lineitem
        WHERE
            l_orderkey = o_orderkey
            AND l_commitdate < l_receiptdate)
GROUP BY
    o_orderpriority
ORDER BY
    o_orderpriority;
```

Listing 8.1: TPC-H query q04.