Radboud University Nijmegen

Institute of Computing and Information Sciences

Exploring Query Re-Optimization

In a Modern Database System

Master Thesis Computing Science

Supervisor:
Prof. dr. ir. Arjen P. de Vries

Author:
BSc Laurens N. Kuiper

Second reader:
Prof. dr. ir. Djoerd Hiemstra

May 2021

Abstract

The standard approach to query processing in database systems is to optimize before executing. When available statistics are accurate, optimization yields the optimal plan, and execution is as quick as it can be. However, when queries become more complex, the quality of statistics degrades, which leads to sub-optimal query plans, sometimes up to several orders of magnitude worse. Improving statistics for these queries is infeasible because the cardinality estimation error propagates exponentially through the plan. This problem can be alleviated through re-optimization, which is to progressively optimize the query, executing parts of the plan at a time, improving plan quality at the cost of additional overhead. In this work, re-optimization is implemented and simulated in DuckDB, a main-memory database system designed for analytical workloads, and evaluated on the Join Order Benchmark. Total plan cost of the benchmark is reduced by up to 44% using this method, and a reduction in end-to-end query latency of up to 20% is observed using only simple re-optimization schemes, showing the potential of this approach.

Acknowledgements

The past year has been a challenging and exciting year. From breaking my leg to starting a job, and finally finishing my thesis. I would like to thank everyone who supported me during this period, and helped me get here.

First of all, I would like to thank my parents for supporting me all the way, cheering me on, and never doubting my studying and researching abilities.

Next, I would like to thank Prof. dr. ir. Arjen P. de Vries, for his interesting courses that gave me a sense of direction in my Master's when I needed it, supervising this thesis, and of course for introducing me to Spinque, where I have worked for over a year now.

Finally, I would like to thank Spinque B.V., for having a wonderful place to work and learn, for being awesome colleagues, and providing me with the hardware required to run the experiments for this research.

I hope you will enjoy reading this work as much as I did creating it.

Laurens

Abbreviations

CE	Cardinality Estimation
DB	Database
DBMS	Database Management System
IMDB	Internet Movie Database
JOB	Join Order Benchmark
SQL	Standardized Query Language
TPC	Transaction Processing Performance Council

Contents

1 Introduction

1.1 Background

A *database management system* (DBMS) is a software system that enables users to define, create, maintain and control access to a *database* (DB) [1]. It is a highly complex and sophisticated piece of software of which the core functionality is to provide its users the ability to store, retrieve, and update data in the database. The data can be viewed at different levels of abstraction. At the *external level*, which is the user's view of the database, implementation details are hidden. The *internal level*, the physical implementation of the database, describes *how* the data is stored. Between these is the *conceptual level*, which describes *what* data is stored, and the relationships among the data.

1.1.1 Query Processing

Users interact with the database through a high-level query language, typically the *standardized query language* (SQL). Queries are processed into a efficient query plan, expressed in a low-level language, before they are executed to retrieve the required data. In general, a query is processed by the following components:

Parser Transforms the query string into a parse tree representation

Logical Planner Converts the parse tree into a *logical query plan*, an operator tree

Optimizer Translates the logical query plan into a logically equivalent, more efficient query plan, by performing various optimizations

Physical Planner Converts the logical query plan into a *physical query plan* that describes the physical operations that need to be performed to execute the query (e.g. choosing between a nested loop join or a hash join)

The physical plan will yield the user results when executed. Without a doubt the optimizer is the most complex component. It has a difficult task, and there are many possible optimizations that can be done, such as expression rewriting, filter pushdown, and *join order enumeration.* The problem of finding the optimal join order is NP-complete [2], and one of the most studied problems in the field of database research.

Join Order Enumeration When a DBMS receives a query that addresses multiple tables, it has to combine their rows through a *join.* Usually the query contains *join predicates*, that specify how to combine the rows of the queried tables. A join without a predicate is called a *cartesian* or *cross join*: each row of a table is matched with each row of another table. Cross joining tables T_i, T_j with *cardinalities* $|T_i|, |T_j|$ results in a table with cardinality $|T_i \times T_j| = |T_i| \cdot |T_j|$. An *inner join* combines rows based on a column equality predicate: a row of a table is only matched with a row of another table if the specified column is equal. The cardinality of join $|T_i \bowtie_{a=b} T_j|$ depends on how many times T_i's column a is equal to T_j's column b, up to $|T_i| \cdot |T_j|$ if columns a and b consist of the same, single value.

Suppose the logical planner has translated a query into a logical plan that can be expressed as $T_1 \bowtie T_2 \bowtie_{a=b} T_3$. With only three input tables, the *plan space* is small, and the optimizer only has to choose between two alternatives: $(T_1 \bowtie T_2) \bowtie_{a=b} T_3$ or $T_1 \bowtie (T_2 \bowtie_{a=b} T_3)$[1]. Larger intermediate results translate into more work, therefore the second alternative, which delays the cross join, is more efficient. This is perhaps the optimizer's most important task: to find the query plan with an optimal join order. In the example it is easy to determine the optimal join order because the entire search space can be searched. However, the join ordering problem is NP-hard [2], therefore this task becomes infeasible as the number of input tables grows.

Despite the complexity, even for large queries with difficult join predicates, the search space can be fully examined with dynamic programming and clever tricks. The *DPhyp* algorithm [3], for instance, models the query graph as a hypergraph, which reducing the search space before searching for the optimal plan. PostgreSQL, a open-source DBMS that is widely used in the commercial world, does this for queries with less than 12 relations, then switches to genetic algorithms for larger queries to efficiently approximate the optimal plan. For bigger queries other techniques can be employed, e.g. *search space linearization*, which allows optimal or near-optimal solutions to be found for chain queries with up to 100 relations that have a linear query graph [4].

Rsearchers have proposed to apply reinforcement learning to prevent the optimizer from making the same mistakes when the same queries are repeatedly issued [5]. Preliminary results show that it matches or outperforms the PostgreSQL optimizer. However, it takes many training iterations to reach this level of performance. More fundamentally, such an approach is likely to suffer from updates to the DB.

Cardinality Estimation In the example, the plan that minimizes the cardinality of the intermediate results is considered optimal. This is an example of a *cost model*. In general, a cost model assigns a cost to each database operation, using the cardinalities of the tables as its principal input. The objective is to choose the cheapest query plan, which should result in the lowest runtime. However, the actual cardinalities of the intermediate tables are not known until the plan is executed, therefore the optimizer relies on *cardinality estimation* (CE). Theoretically, if the cost model and cardinality estimates are accurate, the plan found by the dynamic programming algorithm is optimal. In practice, optimizers are found to produce sub-optimal

[1]Actually, there is a third alternative which cross joins the three tables, and then applies the filter $a = b$, but this is trivially less efficient

query plans frequently.

Virtually all industrial-strength database systems estimate cardinalities using histograms coupled with statistical assumptions of uniformity and independence. This approach is simple to implement, computationally efficient, and tends to work well for workloads found in standard benchmarks for evaluating query engine performance such as those by the *Transaction Processing Performance Council*[2] (TPC). The researchers at TUM point out that TPC generates its data using the very same simplifying assumptions that most query optimizers make, while real-world datasets such as IMDB are full of correlations and skewed data distributions. Therefore they argue that while TPC benchmarks have proven their value for query engine evaluation, they are not useful for evaluating the cardinality estimation component of query optimizers.

In 2015, researchers at the *Technical University Munich* (TUM) introduced the *Join Order Benchmark* (JOB), and used it to investigate the quality of the main components of industrial-strength query optimizers [6]. The three main components of a query optimizer are the cardinality estimation module, the cost model, and the join order enumeration technique. A novel methodology was used to isolate the influence of each individual component on query performance. JOB consists of 113 analytical SQL queries over real data: the *Internet Movie Database*[3] (IMDB). It has a challenging, diverse, and realistic workload. Their experiments reveal that the cost model has much less impact on query performance than cardinality estimates. Simple cost models achieve similar performance to those used in industrial-strength database systems, while cardinality misestimations by a factor of 1000 or more were routinely observed in all tested systems. Small errors are exacerbated by propagating exponentially through the query plan [7], increasing the runtimes of the longest-running queries by up to several orders of magnitude. Finally, they show that it is worthwhile to fully examine the search space using dynamic programming rather than using a heuristic approach, despite the large cardinality misestimations.

An alternative to histogram-based cardinality estimation is sampling. By taking samples of the base tables, a probability distribution can be computed over the *selectivity* of a predicate (the fraction of rows matching the predicate). These distributions are multiplied to compute a distribution over the cardinality of the joined tables, instead of just a single point estimate. This can be used to make the query optimization process more robust, by avoiding plans that appear to have a slightly lower cost, but carry a high risk of underestimation [8]. Sampling has been studied for several decades, and although it produces much more accurate estimates, it is rarely used in practice. One reason is that the many I/O operations required to take a sample cause too much overhead, especially in disk-based systems. Nowadays, many databases reside in main memory, significantly

[2]`http://www.tpc.org/`

[3]`http://www.imdb.com/`

reducing the amount of overhead, making sampling a more viable alternative as demonstrated on JOB [9]. Despite all of these reasons in favour sampling, it is almost exclusively found in cutting-edge research database systems such as HyPer[4], not yet in commercial systems. An approach that uses machine learning poses another alternative to estimate cardinalities on JOB [10]. It addresses some of the weak spots of sampling, but suffers from generalization issues.

1.1.2 Re-Optimization

JOB has brought to light the weakest component in query optimization: cardinality estimation. Since the benchmark's inception, researchers have worked on improving this component, and with some success. While improving cardinality estimates will certainly improve runtime, another strategy has been proposed that attempts to avoid or react to inefficient query plans that were selected due to poor estimates, rather than facing the problems with CE head-on. All of the methods discussed so far share the same approach when it comes to query processing, which is to *plan-first execute-next*. This is why problems with cardinality estimates arise. Even when cardinality estimates become more accurate, small errors propagate exponentially through the query plan, causing huge misestimations as the number of joins increases, sometimes leading to disastrous plans, especially in the presence of skewed and correlated data distributions.

A technique called *re-optimization* attempts to overcome this problem by interleaving the planning and execution phases. It does this by executing a portion of an optimized plan, measuring the true cardinalities, and optimizing the plan again before continuing execution. A simple re-optimization scheme was simulated on JOB, demonstrating that it can improve the end-to-end latency of the top 20 longest running queries in the benchmark by 27% in PostgreSQL [11]. Another big takeaway from this research is that, without re-optimization not much improvement was observed on this workload, even when perfect cardinality estimates for joins of 3 or fewer tables were injected into the optimizer. When the cardinalities of joins with 4 or fewer tables were injected, improvements were observed, showing the challenge that CE must overcome to improve performance on JOB.

1.2 Objectives

The aim of this research is to explore query re-optimization in more detail, and evaluate its effectiveness in a practical setting, beyond simulation. Based on experiments with a real-world dataset, conclusions are drawn about re-optimization in order to help the field of query processing forward in making the right steps forward in improving end-to-end query latency.

[4]`https://hyper-db.de/`

The main objective of this research can be formulated as:

"To research query re-optimization schemes and evaluate their strengths and weaknesses compared to the plan-first execute-next approach, in order to expose problems with current query processing, and explore alternatives."

1.3 Research Questions

To achieve the main objective, the following central research question is defined:

"When do re-optimization schemes improve end-to-end latency, balancing the trade-off between increased planning and materialization cost, versus the benefit of improving the query plan, and how to define such schemes?"

In order to answer the central question, the following sub questions are defined:

1. What does it mean to re-optimize a query plan?
   a. Which methods for re-optimization exist, and what do they aim to do?
   b. How can a query optimizer decide to re-optimize a plan effectively?
2. How can re-optimization be implemented?
   (a) If not implemented fully, can schemes be simulated? If so, is the simulation realistic?
3. When does re-optimization improve end-to-end query latency?
   a. How big is the trade-off between costs and benefits?
   b. What is the difference between real and simulated schemes?
4. How does re-optimization make query processing robust against cardinality misestimations?
   a. To what extent does it reduce plan cost?

Question 1 is aimed at gaining a greater understanding of the subject as a whole, and question 2 is aimed at learning the internals of a DBMS. The goal of questions 3 and 4 is to obtain quantitative evidence on the effects of re-optimization.

1.4 Structure

An overview of related work as well as a theoretical framework can be found in section 2. This is where question 1 is answered: through studying both the history of query processing, as well as literature on re-optimization. Question 1b differs per DBMS, therefore the answers found in literature will differ from what is presented here. Section 3 explains how re-optimization is implemented in this work, and is where question 2 is answered. This section also introduces various re-optimization schemes, and goes into the setup of the experiments. The next section, section 4, presents the results of the experiments. These are discussed in section 5, where questions 3 and 4 are answered. In the same section, question 2b is revisited, to answer the second part of the question. Finally, the central research question is answered in section 6. Additional results can be found in the appendices.

2 Theory

This chapter reviews the history and background of developments in the field of query processing and re-optimization (section 2.1), which forms the basis of a theoretical framework (section 2.2).

2.1 Related Work

Many research papers on query processing and optimization focus on the industry standard benchmarks by TPC. As explained in the introduction, these are not appropriate for evaluating the CE component of optimizers, because simple estimators work unrealistically well on them. Re-optimization pays off more when cardinalities are difficult to predict. The oversimplified benchmarks, and the fact that modern query execution engines incorporate new technologies that make implementing re-optimization more complex together may provide the reason why most of the research on re-optimization was conducted over a decade ago, despite interesting ideas [12], and calls to change the execution model that is prevalent today [13].

2.1.1 Optimization

Before SQL and relational databases were the norm, querying a DB was more challenging. Users had to define how their query was to be executed, and the difference between a well- and poorly optimized query was huge. SQL is a high level query- and data manipulation language, in which requests are stated non-procedurally, without reference to access paths, designed for a relational database management system. This implies that the optimization challenge is left to the DBMS. The first implementation of SQL was IBM System R. Many of its design choices influenced later relational systems and still hold up to this day, notably the decision to optimize a query at compile time using dynamic programming, as described by one of IBM's influential publications [14].

For a while, most of the research on optimization focused on this plan-first execute-next approach. In these years it became clear that many of the assumptions that were made during planning do not hold up during execution, leading to sub-optimal plans. This lead to the development of new techniques that attempt to deal with such limitations by changing the plan at some point during query processing. These are referred to as *adaptive*, *dynamic* or *re-optimization* approaches, but often mean different things. In this section, from section 2.1.3 onward, these approaches are discussed.

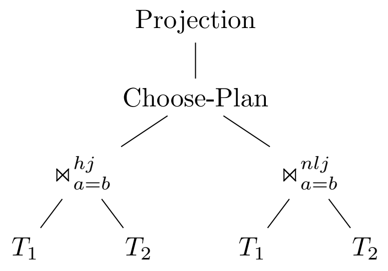


Figure 1: A dynamic query evaluation plan in Volcano.

2.1.2 Volcano Model

Before going into re-optimization, background information is discussed to understand how the evaluation of a query plan is modelled. The *Volcano Model* [15] describes what is now known as the 'classical' evaluation strategy of a query. It is still influential to date, with many modern databases implementing a variant of this model.

Tuple Streams Formerly known as the Iterator Model, the Volcano Model implements operators using an interface that produces an iterable stream of tuples, using the *open-next-close* protocol. Queries are expressed as query plans in which the operators are executable processing algorithms. When *next* is called on the root of a query plan, the operator tree is propagated recursively, calling *next* on its children until a tuple can be produced at the root. This entails that tuples can flow freely through the plan as if it were a pipeline i.e. operators do not need to process all input tuples before being able to output them. That is, unless the operator is said to be *blocking*. An example of this is the `MAX` operator: one cannot be sure of the maximum value until all tuples are processed.

Choose-Plan If a query evaluation evaluation plan is used repeatedly over an extended period of time, while the underlying data changes, the assumptions that were made when the plan was optimized lose their validity, which could cause the plan to be sub-optimal. At this point the plan should be re-optimized. Note that this is a different kind of re-optimization than in the introduction: re-optimization takes place between queries, rather than mid-query. To enable this, Volcano includes the *choose-plan* meta-operator that allows delaying optimization decisions until run-time, which creates a *dynamic query evaluation plan*. This can avoid a costly re-optimization of the plan as follows: consider the query plan shown in Figure 1. The choose-plan operator decides at run-time, based on some policy, to join tables T_1, T_2 using a *hash join* (hj) or a *nested loop join* (nlj), rather than having a static join method that is chosen during optimization.

The choose-plan operator could be used to change the join order of a query plan, but only if its children are logically equivalent plans with different pre-defined join orders. As mentioned in the introduction, join order enumeration is NP-hard, therefore it is infeasible to include all the different join orders using choose-plan operators. The query plan is dynamic, but the alternatives are planned before execution. Clearly, this differs from the plan-first execute next approach, but it does not offer a lot of flexibility, especially for complex queries with many joins.

2.1.3 Distributed Databases

The limitations of static query processing approaches are especially noticeable in distributed database systems, due to the fluctuating characteristics of resources. Therefore, adaptive approaches have gained significant attention in this field.

Initial Delay *Query scrambling* [16], [17] attempts to deal with unexpected delays in accessing remote sources, using two methods called *rescheduling*, and *operator synthesis*. Rescheduling does not alter the query plan, but rather the order of execution. For example, consider the *bushy* query plan qp_b that joins tables T_{1-5}: $qp_b = (T_1 \bowtie T_2) \bowtie (T_3 \bowtie (T_4 \bowtie T_5))$. If processing starts at $T_4 \bowtie T_5$, but there is a big delay in accessing T_3, the result of $T_4 \bowtie T_5$ is materialized in a temporary table, and $T_1 \bowtie T_2$ is processed while waiting for the delay. If no progress can be made anymore using this method, *operator synthesis* is invoked, which creates new operators (e.g. a join between two relations that were not directly joined in the original plan), significantly modifying the shape of the plan. Introducing new operators will likely increase the overall cost, but allows processing to continue instead of waiting.

The first method works best on bushy trees, which could have many sub-trees that can be processed as is, independent of other nodes in the tree. However, optimizers often produce *linear* trees (*left-deep*, *right-deep*, or *zig-zag*) that do not have sub-trees that can be processed independently. For example, a reordering of query plan qp_b could lead to a left-deep plan $qp_l = T_1 \bowtie (T_2 \bowtie (T_3 \bowtie (T_4 \bowtie T_5)))$, which is logically equivalent to the bushy one, but has only one sub-tree ($T_4 \bowtie T_5$) that can be processed as is, therefore no progress can be made using rescheduling.

Continuous Adaptation An influential paper in this field is Eddies [18]. It describe a query processing mechanism that continuously reorders operators in a query plan as it runs. This is achieved by merging multiple unary and binary operators into a single n-ary operator (an *eddy*), wherein each tuple has a flexible ordering of the query operators. Eddies implement Volcano's open-next-close protocol. Suppose an eddy performs the join $T_1 \bowtie T_2 \bowtie T_3 \bowtie T_4$. Tables T_{1-4} come from different sources, and all have different initial delays and different bandwidths, possibly also varying over the course of execution. If the join order was chosen statically based on cardinality estimates, e.g. $T_1 \bowtie (T_2 \bowtie (T_3 \bowtie T_4))$, but data source T_4 has a low bandwidth, we encounter a *synchronization barrier*: processing cannot continue until all of $T_3 \bowtie T_4$ is finished. Eddies solve this problem by allowing each tuple that they have to process to be routed individually through its operators, such that joins T_1, T_2, T_3 can be processed while waiting for data from T_4 to arrive.

Eddies are not only able to reorder joins mid-execution, which is an example of *logical plan re-optimization*, they are also able to adaptively change the behaviour of join algorithms. They have chosen specifically for non-blocking join algorithms that have many *moments of symmetry* that allow the left and right sides of the join to be swapped during execution. This adaptation pertains to the physical operator, therefore this is an example of *physical plan re-optimization.*

Limitations The methods discussed in this section are able to change the query plan mid-execution, partly by means of a flexible join order. It would that these could alleviate the join ordering problem that was introduced in the previous section, but they suffer from the same limitations, because they are aimed so specifically at the distributed database setting. Their flexibility deals with lack of certainty in delay and throughput, misestimated cardinalities not as much. The initial query still plan heavily influences the join order, which is based on cardinality estimates. Neither offers a complete reordering of the join operators in a plan, therefore neither are suitable for the CE problem.

2.1.4 Mid-Query Re-Optimization

The first work on re-optimization that allows for a complete reordering of joins dates back to 1998 [19]. The authors state that optimizers often produce sub-optimal query plans due to out-of-date statistics and exponential error propagation, but also due to filter predicates correlation, causing histogram-based approaches to be inaccurate[5]. Furthermore, filter predicates may be fuzzy (e.g. the SQL `LIKE` operator), or contain a user-defined function, in which case there is no way for the DBMS to (accurately) estimate selectivity. They the *dynamic re-optimization* algorithm that, through collecting statistics, detects sub-optimalities during execution and attempts to correct them.

Statistics Collection Sub-optimalities are detected by inserting a *statistics collector* operator at key points in the query plan. An example is given in Figure 2. After the filter on T_1, the selectivity, but also statistics about attributes that appear in predicates in the remainder of the plan $(T_1.a_2, T_1.a_3)$ are collected. If the collected statistics indicate that the current query plan

[5]The authors specifically separate experiments on the TPC-D benchmark from experiments on skewed datasets

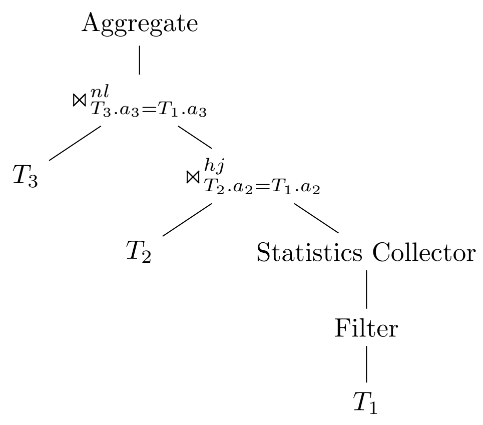

Aggregate
$\bowtie^{nl}_{T_3.a_3=T_1.a_3}$
T_3
$\bowtie^{hj}_{T_2.a_2=T_1.a_2}$
T_2
Statistics Collector
Filter
T_1

Figure 2: A plan with run-time statistics collection, adapted from Kabra & DeWitt [19].

is sub-optimal, the remainder of the plan is re-optimized , incurring a small cost. Re-optimization possibilities include reordering joins, changing join strategy, or swapping the left and right sides of a join.

Trade-Off There is a trade-off between gaining information and improving the query plan versus letting the execution plan run its course. Dynamic re-optimization has two parameters can be set to define a policy that weighs up the costs and benefits. The authors chose to materialize the intermediate result as a temporary table, which is easier to implement than suspending the query, but incurs slightly more cost due to I/O. Nonetheless, a significant improvement in execution time was achieved for complex queries. Note that this could unnecessarily slow down simple queries for which the additional overhead of re-optimizing is large compared to the already low of the plan.

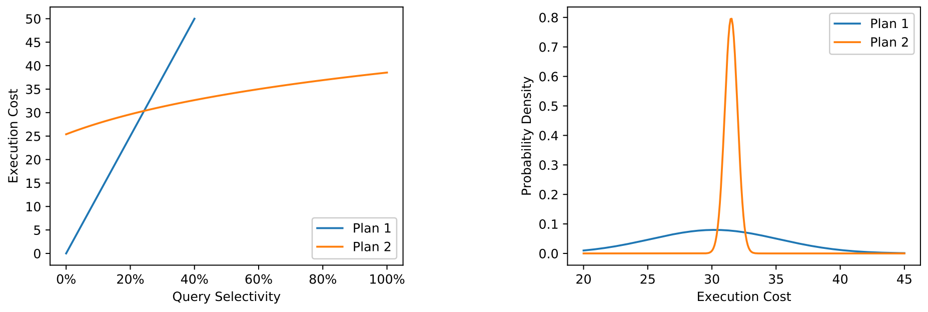
POP In practice, dynamic re-optimization only speeds up processing by re-optimizing hash joins (most hash join implementations are blocking, and need to be fully materialized anyway), and only if query results were not pipelined. Researchers at IBM raised these problems and improved on the approach with *progressive query optimization* (POP) [20], claiming to be more generally applicable. They define CHECK operators, checkpoints that collect statistics much like the statistics collector operator (although only cardinality was measured in their experiments). CHECK additionally has a *check range* parameter that succeeds when the measured cardinality is within a certain range. The range is determined during join order enumeration, and is set such the remaining plan at the checkpoint is guaranteed to be sub-optimal with respect to the optimizer's cost model when CHECK fails. They also observe that intermediate results should not always be reused: continuing execution from a sub-optimal initial choice of join order could incur more cost than restarting from scratch. The speedup of this approach on a real-world dataset was much more significant than on the TPC-H benchmark.

2.1.5 Avoiding Risks

Generally speaking, cardinality estimates are rough, uncertain, single-point estimates. Cardinality is usually estimated using histograms, in which the *attribute value independence* (AVI) assumption is used, which rarely holds up in practice due to the correlations present in real-world data. These estimates are the principal input of the cost model, which treats them as precise and accurate. The optimizer selects the plan with the lowest cost, as determined by this model. An important quality of the estimates is overlooked, namely their *certainty*.

Robust Plans Babcock & Chaudhuri propose Robust Cardinality Estimation (RCE) [8], a trade-off between predictability and performance. Their method combines sampling and Bayesian inference to compute a probability distribution over the possible selectivities. A *confidence threshold* T parameter can be set, which specifies the percentile value of the distribution to take as input for the cost model. Taking Figure 3 as an example, setting the confidence threshold to $T = 50\%$ (equal to the expected value) will cause the optimizer to select Plan 1. However, the cost of Plan 1 grows drastically as the selectivity of the query increases: it is a risky plan. By increasing the confidence threshold, e.g. $T = 80\%$, Plan 2 is selected, which has a higher expected cost, but is much more robust to an unexpected change in selectivity.

By selecting robust plans over (seemingly) optimal plans with regard to the cost model, more predictable query times can be achieved, at the cost an overhead incurred by sampling. The authors claim that their method avoids degradation in estimation quality due to the propagation of estimation errors. However, this is only true for estimates on selectivities on the base



(a) Execution Costs (b) Probability Density Function

Figure 3: Two Hypothetical Plans, adapted from from Babcock & Chaudhuri [8].

tables, because they do not consider techniques that join random samples. Errors in join cardinality estimation propagate through the remainder of the plan. Therefore, error propagation is still a real issue for RCE.

Robust Re-Optimization Re-optimization could benefit from from RCE: robust plans are less likely to be sub-optimal, therefore incurring the cost of re-optimization is less often needed. The authors of *Rio* [21], a *Proactive Re-Optimization* approach, recognize this and combine the ideas of robustness and re-optimization. Rio computes bounding boxes around the size of relations, similar to the CHECK operator in POP. These bounding boxes are considered when selecting the plan, like in RCE, yielding more robust plans. When there are multiple close-to-optimal plans, a switchable plan is created, like the choose-plan operator in Volcano.

Whereas the re-optimization approaches discussed so far are reactive, Rio also has a proactive component. As an example, assume that a check on the selectivity σ of a filter operation fails when 25% of the records in relation R are read. Re-optimization is triggered, a new plan is created, with a new bounding box. The check on the new bounding box might fail when 80% of the records in R are read, triggering re-optimization again. This is costly, and will lead to slow processing. Rio deals with this problem by treating the records that are read so far as a sample, and extrapolating the selectivity of the filter operation to the remainder of the records. This way the re-optimization mechanism is given an estimate of an operator's selectivity before it has finished processing. However, extrapolation is done under the assumption that tuples in R are randomly ordered, which is questionable.

2.1.6 Re-Optimization Revisited

After being largely ignored for over a decade, the introduction of JOB has sparked new interest in re-optimization. Rather than fully implementing re-optimization, researchers simulated its effects in PostgreSQL [11], by using the `CREATE TEMPORARY TABLE` to materialize intermediate results. Materialization is triggered when the Q-error, the factor by which the estimated cardinality of an intermediate table is larger or smaller the actual cardinality, is greater than a threshold. Total planning time was measured by summing the planning time of the original query, and all generated `SELECT` queries. Total execution time was measured by summing the execution time of each `CREATE TEMPORARY TABLE` command and the final `SELECT` query, excluding parsing and optimization time (in most graphs/tables). The researchers believe that this is a reasonable approximation of a simplistic re-optimization scheme.

Re-Optimization vs. CE The effects of this scheme were compared to the effects of improved cardinality estimation, by injecting the true cardinalities of up to n joins or fewer into the optimizer. This showed that re-optimization improves end-to-end latency of the 20 longest running queries in JOB just as much as having perfect cardinality estimates for joins of up to 4 tables. Many short queries saw a large relative increase in runtime, but this is negligible when looking at the absolute increase, and can be avoided with a more sophisticated re-optimization scheme. Combining re-optimization with perfect cardinality estimates of joins with up to 4 tables continued to reduce execution time, but only slightly.

The speedup on JOB with perfect cardinality estimates for joins of up to 4 tables is comparable to that of the simple simulated re-optimization scheme. Estimated cardinality perfectly for joins with this many tables is a difficult challenge. With clever sampling techniques plan quality on JOB can be improved significantly [9], if foreign key indices are available. However, due to the way the results are presented (geometric rather than arithmetic mean, relative rather than true runtimes), it is hard to judge what the total speedup on the benchmark is. The findings of the cardinality injection experiment suggest that sampling improves queries with a small number of joins much more than a large number of joins; exponential error propagation continues to be a problem Given these observations, re-optimization is likely to lead to the biggest gains in end-to-end latency, rather than improving CE.

2.2 Theoretical Framework

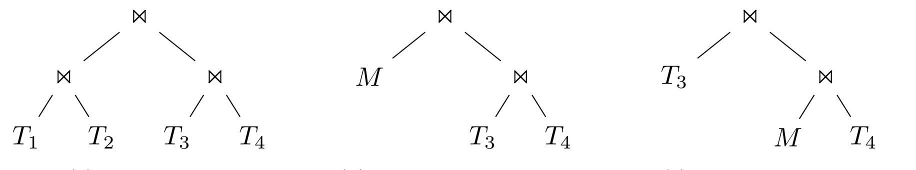
The field of data management has evolved to consider increasingly complex queries, for which traditional statistics-based cardinality estimation techniques fall short due to exponential error propagation [7]. Query optimizers rely on these estimates to select the optimal query plan. Inaccurate estimates can lead to a significant degradation of performance, and it has been acknowledged that this happens often in practice [6]. Distributed database systems ran into similar issues due to statistics being less available, and due to data coming from remote sources being delayed, for which adaptive approaches have been deployed successfully [16]–[18]. In the non-distributed setting, research has focused on other ways to improve processing speeds, e.g. by increasing bandwidth through switching to columnar rather than row storage to increas pipeline throughput, faster (de)compression [22], automatic parameter configuration based on CPU characteristics, and sampling-based cardinality estimation [8], [9].

These developments have significantly reduced query latency, but have not addressed the problem of exponential error propagation in cardinality estimates. Since the introduction of JOB [6], shortcomings in query optimization gained more attention, but most of the research has been confined to the plan-first execute-next paradigm [5], [9], [10]. One piece of work has shown with a simulation experiment that a simple re-optimization scheme can already yield better results on JOB in PostgreSQL [11]. Re-optimization requires materialization of intermediate results, which would usually be too expensive in a disk-based DBMS like PostgreSQL, but the researchers configured the system to cache all tables in memory. The simulation experiment showed improvement especially for long-running queries. Many of the adaptive techniques were proposed over a decade ago [19]–[21], but none of the widely-used database systems have adopted them, despite the increasing need to evaluate complex queries. Since then, hardware has evolved, and main-memory database systems have gained traction. In these systems, the cost of reading and writing tables is less, therefore materialization is cheaper, which makes a strong case for re-optimization.

These observations indicate that there is likely much to be gained from re-optimization. Exploratory research is needed in order to determine its strengths and limitations in different settings:

1. Row vs. Column storage
2. Disk-based vs. Main-memory
3. Volcano-style vs. Other execution models

Throughout the rest of this thesis, the term *re-optimization* refers to the style of approach described by Dynamic Re-Optimization and POP, in which statistics of intermediate results are measured and used to re-optimize the remaining plan. A simple example is sketched in Figure 4. Re-optimization is evaluated in DuckDB, a main-memory DBMS that uses column storage and has a Volcano-style execution model, in order to answer the research questions posed in the previous section. Many of the findings presented in this work will pertain only to this setting, with some exceptions.



(a) Initial plan

(b) Plan with a materialized intermediate result

(c) Re-optimized plan

Figure 4: An example of re-optimization where a join in the initial plan (a), $T_1 \bowtie T_2$, is materialized into a temporary table M (b). Based on the measured statistics of M, the decision to re-optimize the remaining plan is made, resulting in a different join order (c).

3 Approach

This section introduces various re-optimization schemes, as well as describing implementation details and experimental setup.

3.1 Implementation

The first step in this research has been to implement re-optimization in DuckDB [23], a novel DBMS in development by the Database Architectures group of the Centrum Wiskunde & Informatica in Amsterdam. DuckDB has been designed to execute analytical SQL queries while embedded in another process e.g. interactive data analysis using tools such as R or Python. DuckDB is written in C++, operates in main-memory, uses columnar storage, and executes queries in a so-called *Vector-Volcano* model, pulling chunks at a time, rather than tuples. Due to the early stage of development, DuckDB does not yet keep track of table statistics, and *lacks a CE module.*

As remarked by Perron et al. [11], modern query execution engines have different storage and execution models than a decade ago when re-optimization was more actively researched, making the issue more complex. DuckDB is a good example of this, but was created with a plan-first execute-next model of query execution in mind, not re-optimization. Therefore, a distinction has to be made between the implementation of re-optimization in DuckDB, and a simulation of its performance were it created with re-optimization in mind. Both approaches require roughly the same components.

3.1.1 General Design

Re-optimization is a constant back and forth between executing parts of the logical plan, measuring statistics, and evaluating whether to re-optimize the remainder of the plan. DuckDB executes the physical plan in a pipelined query engine, which cannot easily be suspended. Therefore, the decision was made to manipulate only the logical query plan and execute parts of it. Re-optimization can be realized in this paradigm if the following three operations can be carried out:

1. Select a node in the logical plan
2. Materialize the selected node as a table
3. Replace the selected node with the table

By repeating these steps in order, a query plan can be incrementally executed. The implementation of each operation is discussed in detail[6].

[6]The implementation is publicly available at: `https://github.com/lnkuiper/duckdb`

Node Selection A node is selected by traversing the operator tree of the logical query plan. The choice of node depends on the re-optimization scheme (see section 3.2). It is important to select operators for which it is difficult to estimate cardinality (aggregate, join, distinct, and filter), rather than those for which it is trivial (project, sort, get, etc.). Without issuing any queries, the only information available in DuckDB about nodes in the query plan is the structure of the operator tree, the parameters of the operators, as well as the types and cardinalities of the base tables at the leaves of the tree.

Intermediate Result Materialization Parameters are extracted from the operator tree T' with the selected node as the root node. Using these parameters, a SQL query $Q_{T'}$ that yields a logically equivalent plan to T' is constructed. The node is materialized into a temporary table by issuing the query `CREATE TEMPORARY TABLE AS (`$Q_{T'}$`);`. Within the parameters, tables and columns are referred to by *column bindings*. These bindings are unique to each plan, consisting of a tuple of (*table index*, *column index*), assigned at the leaf nodes of the plan by the table scan operator. Because column bindings cannot be used in a SQL query, these must be mapped back to their original names.

The cost of writing intermediate results to a table are expensive, therefore care must be taken as to not materialize more columns than needed. Only the columns that are used in the remainder of the plan (excluding the selected node) should be materialized. This information is collected by recursing through the logical operator tree and keeping track of occurrences of column bindings.

Node Replacement The selected node is replaced with a table scan operator on the materialized result. Replacing one operator with another is trivial, but requires modifications to the remainder of the plan. Because the table scan operator is tasked with assigning column bindings, the materialized node receives different bindings. All references to the original column bindings in the remaining plan must be replaced with the new column bindings.

3.1.2 Procedure

Using the described components, re-optimization is defined in algorithm 1. The re-optimization module obtains the plan after it has been parsed, planned, and optimized. In the following section, re-optimization schemes are explained by means of this procedure.

Algorithm 1 Re-optimization algorithm.

```
procedure ReOptimize($p$)                              ▷ Input: optimized logical plan $p$
    while $\neg$ Materialized($p$) do
        $n \leftarrow$ SelectNode($p$)                 ▷ Selection depends on scheme
        $t \leftarrow$ MaterializeNode($n$)
        $p$.ReplaceNode($n, t$)
        if DecideOptimize($p, n, t$) then              ▷ Decision depends on scheme
            $p \leftarrow$ Optimize($p$)
        end if
    end while
    return $p$                                         ▷ Fully materialized plan
end procedure
```

3.2 Re-Optimization Schemes

Two types of re-optimization schemes are considered. *Real schemes* have been implemented into DuckDB, and follow the steps in algorithm 1. *Simulated schemes* are simulations of schemes that would be possible if DuckDB could be suspended mid-execution.

3.2.1 Real Schemes

Schemes introduced in this section allow for basic analysis of the benefits and costs of re-optimization. Join ordering cannot be optimized when there is only one join in the remaining plan. Therefore, none of these schemes will choose to materialize a join that leaves the remaining plan with only one join.

Baseline The baseline scheme is to not re-optimize, equivalent to selecting the root node for materialization, no additional optimization, ending the procedure after 1 iteration. This is DuckDB's regular mode of operation, using the classical plan-first execute-next approach.

Filters Only This scheme selects Filter operation nodes to materialize until there are none left. The plan is optimized once after all Filter nodes have been materialized. Then, the root node is selected for materialization, ending the procedure after $f + 1$ iterations, where f is the number of filters in the plan.

N-Join For $N \in 2, ...,$ N-Join selects nodes that join N tables at a time. Note that for $N > 2$, e.g. $N = 3$ it is not always possible to find a node that joins exactly 3 tables, as shown in Figure 5. In this case the algorithm chooses the node that joins the smallest number of tables greater than N.

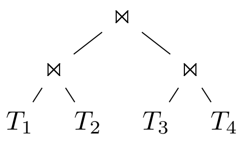


Figure 5: A plan in which there is no join possible between 3 tables.

One Step This scheme starts the same as *Filters Only*, but after the Filter nodes have been exhausted and the plan is optimized, it selects Join nodes that combine only two tables at a time, optimizing after each join. This continues until three tables are left, at which point the root node is selected, ending the procedure in $f + t - 2$ iterations, where t is the number of base tables in the plan.

Smart Step The error in the estimated cardinality of a certain node in the query plan is the product of the errors of its children times the error of that node. Errors in cardinality estimation occur only in nodes that require estimation, which are in aggregate, join, distinct, and filter operations. Assuming a constant error rate ϵ, the error rate of a node v is ϵ^n where n is the number of aggregate, join, distinct, and filter operations in the sub-tree with v as the root node.

The join, being a binary operator, is interesting because its error rate combines that of its children. Therefore, join nodes with large subtrees as children form a large risk, because their error rate has a high exponent. Smart Step searches for the join node with the smallest error rate exponent $n > N$, where N is a parameter, and selects its child with the largest error rate exponent for materialization. With this selection a moderate risk is taken, rather a large one. An example with parameter $N = 3$ is given in Figure 6.

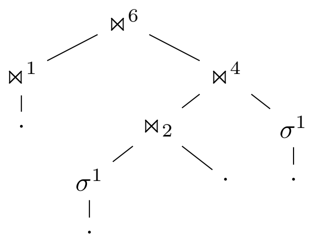


Figure 6: Logical plan with error annotated error rate exponent. Smart Step with $N = 3$ searches for the join node with smallest exponent greater than 3, finding $\bowtie_4$. Its child with the largest exponent is then chosen for materialization, which is $\bowtie_2$.

3.2.2 Simulated Schemes

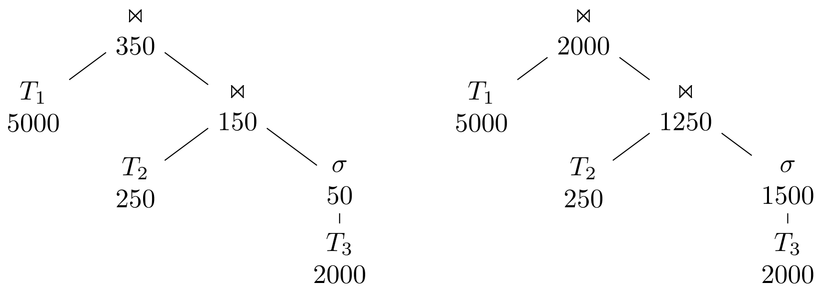
A DBMS that is able to suspend its execution can select a node to materialize not *before*, but rather *during* execution. This can be simulated using of the components described in section 3.1.1. Simulation causes additional overhead that would not be there were this a true implementation. This overhead is not added to the total query time, explained in section 3.3.1. Simulated re-optimization schemes are defined by means of algorithm 2.

Algorithm 2 Simulated re-optimization algorithm.

```
 1: procedure SimulateReOptimize($p$)                ▷ Input: optimized logical plan $p$
 2:     while ¬ Materialized($p$) do
 3:         for $n$ in $p$ do                            ▷ Loop from leaf nodes to root
 4:             $n$.cardinality ← TrueCardinality($n$)       ▷ Inject measured cardinality
 5:             if DecideMaterialize($p, n$) then           ▷ Decision depends on scheme
 6:                 break
 7:             end if
 8:         end for
 9:         $t$ ← MaterializeNode($n$)                   ▷ Final $n$ before loop was broken
10:         $p$.ReplaceNode($n, t$)
11:         $p$ ← Optimize($p, n$)    ▷ Always optimize: decision is already made at line 5
12:     end while
13:     return $p$                                        ▷ Fully materialized plan
14: end procedure
```

Cardinality Q-Error This scheme defines the `DecideMaterialize`(p, n) function to yield true when the the q-error between the estimated and measured cardinality of a node exceeds a threshold value T, otherwise false. This corresponds to the simulated re-optimization procedure described by Perron et al. [11].

Cost Q-Error A large q-error in cardinality estimation does not always indicate a sub-optimal query plan, as demonstrated in Figure 7. In theory, if the cost of the remaining plan does not change much given the measured cardinality, the remaining plan should theoretically run for as long as it was expected to. This scheme defines `DecideMaterialize`(p, n) to yield true when the q-error between the cost of the remaining plan with the estimated cardinality of node n and the cost of the remaining plan with the true cardinality of node n exceeds a threshold value T, false otherwise.



(a) Estimated cardinalities (b) Actual cardinalities

Figure 7: An example of a plan where a large q-error in cardinality does not indicate a sub-optimal query plan. The estimate for σ is off by orders of magnitude, but σ still much smaller than T_1, and therefore should be joined with T_2 first to keep intermediate results small.

3.3 Experimental Setup

This section describes the setup employed in the experiments of this research.

3.3.1 Query Latency

The duration of the planning and execution phases, time to write materialized results to a table, as well as the duration of re-optimization 'tooling' (e.g. fixing column bindings after replacing a node in the plan) and simulation overhead were recorded using DuckDB's `PRAGMA enable_profiling;` statement. The following steps were performed five times to measure an average:

1. Connect to a fresh in-memory database
2. Initialize tables
3. Execute query

The database was initialized for each query because executing the same query multiple times in a row can cause a speedup due to data being 'hot': residing in cache rather than main memory.

Re-Optimization For the real schemes the total query time is measured as the total end-to-end latency. This is split up into three query profiling categories:

1. Planning
    - Initial planning time
    - Sub-query planning time
    - Re-optimization tooling
    - Additional calls to Optimize

2. Execution
    - Sub-query execution time
    - Root node execution time (after sub-queries are done)
3. I/O
    - Time to write materialized result to table

I/O profiling was measured differently from the other categories, because DuckDB's query profiler does not explicitly measure it. It was observed that for queries that create a table, the total time of a query is not fully covered by the phases that the profiler measures. This excess is assumed to be the cost of writing data to a table. This assumption is verified in section 4.3.

Some unnecessary overhead is incurred by planning sub-queries. This is added to the total to illustrate the difference between the real implementation and the simulation.

Simulated Re-Optimization For simulated schemes the same three profiling categories are measured. Sub-query planning time is taken out of the equation for these.

3.3.2 Plan Cost

The cost model that DuckDB uses to optimize join order is simple. Given a join tree T, the cost function C is defined as

$$C(T) = \begin{cases} 0 & \text{if } T \text{ is a leaf node} \\ |T| + C(T_1) + C(T_2) & \text{if } T = T_1 \bowtie T_2 \end{cases}.$$

In short, the cost model sums up the size of (intermediate) results, and ignores the cost of single relations as they have to be read anyway. The rationale is that larger intermediate results cause more work.

According to the cost model, the cost of a logical query plan is obtained by looping over its join nodes, and summing over their cardinalities. Node cardinalities are obtained by converting each one to a query, and executing `SELECT COUNT(*) FROM (<query>);`. The cost of a re-optimized plan is obtained by treating each selected node as a logical plan, computing its cost, and adding it to a total.

3.3.3 Benchmark

Query profiling and plan cost experiments are performed on the Join Order Benchmark (JOB) [6].

Join Order Benchmark This benchmark was introduced in 2015 by Leis et al. [6], and consists of 113 multi-join queries on a real-world dataset. The query set consists of 33 query structures, consisting of one *select-project-join* block, each with 2-6 variants. JOB is designed to test the cardinality estimation component of query optimizers, specifically to find a good join order. Each query has between 3 and 16 joins, with an average of 8.

3.3.4 Hardware

All experiments were carried out on a server machine provided by Spinque B.V.[7], with the following specifications:

CPU	Intel Xeon E5-1650 v4 @ 3.60GHz
Memory	Kingston 32GB DDR4 @ 2400MHz (x4)

DuckDB operates on a single thread, because it does not (yet) support intra-query parallelism. Having 128GB of memory is more than enough for JOB, therefore swapping is not an issue.

[7] `https://www.spinque.com/`

4 Results

This section reports the results of the query profiling and plan cost experiments, and some ad hoc analysis. Section 5 gives thorough analysis.

4.1 Query Latency

For each scheme, only the results of the most successful parameter configuration in terms of overall benchmark run-time are shown in the figures in this section. The complete results can be found in Appendix A.

Per Query The latency per query is shown in Figure 8. Queries are sorted by their latency in baseline DuckDB, with longer running queries appearing at the right side of the bar plot. Query runtime has a rather large range without re-optimization; some queries take a fraction of a second, and others take multiple seconds. Re-optimization schemes seem to have a more consistent query runtime.

The best overall performance is Cardinality Q-Error, the simulated scheme by Perron et al. [11]. Multiple parameter configurations were tried (see Appendix A), but the same threshold value ($N = 32$) was found to be optimal. The other simulated scheme, Cost Q-Error, did not perform nearly as well. This is likely due to the interaction between the underdeveloped cardinality estimation and the cost model. The interaction causes big differences in estimated and measured cardinality to not alter the cost of the remaining plan by much in many cases. But even if these components were fully developed, this scheme is not guaranteed to succeed, as remarked by Perron et al. [11] "*To arrive at a plan near optimal, all catastrophic plans must have higher costs than a good plan. This is difficult to guarantee when improving only a subset of cardinality estimates.*".

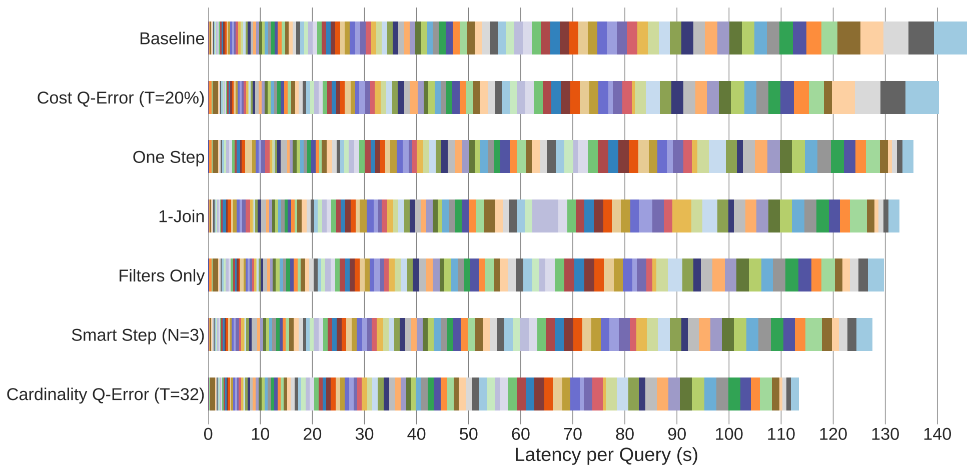


Figure 8: JOB end-to-end latency by query comparison of baseline DuckDB with re-optimization schemes.

4.1.1 Profiling

The overall performance on JOB with aggregated query profiling is shown in Figure 9. Profiling is split into three sections: execution, planning and I/O. Execution is measured as the time to process the data through the query plan. Planning is measured as the time to generate and optimize a query plan. I/O is measured as the time to write intermediate results to temporary tables. Even though DuckDB is a main-memory DBMS, I/O is still a factor. Writing data to memory is much cheaper than to disk, but not negligible, especially for schemes that materialize more often, such as One Step.

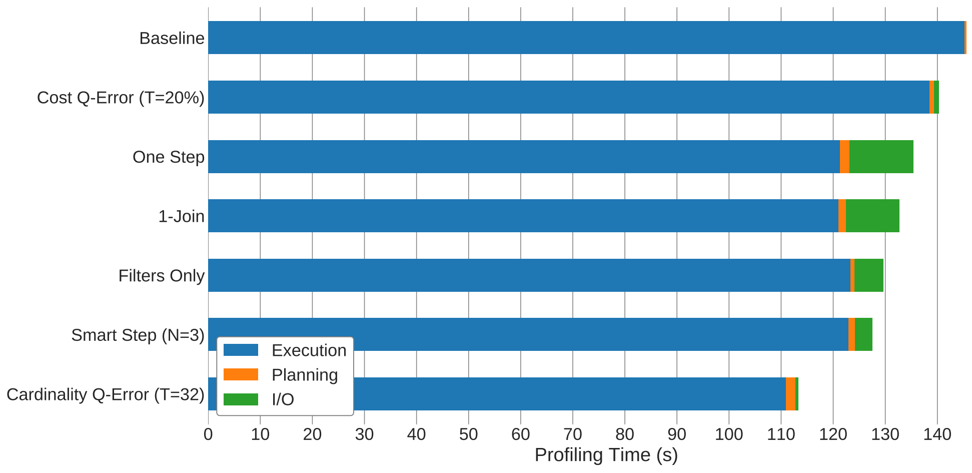


Figure 9: JOB query profiling of baseline DuckDB compared with re-optimization schemes.

Longest vs. Shortest Queries Query profiling of the 20 shortest and 20 longest running queries are shown in Figure 10 and Figure 11 respectively. Baseline DuckDB performs better than re-optimization on queries that take shorter to complete. Execution time is increased, as well as the additional overhead of planning and I/O. Re-optimization schemes that materialize less often perform better on these queries.

For queries that take longer to complete, re-optimization pulls ahead. Schemes that materialize more often perform best. Execution time is reduced, and the additional overhead of planning and I/O is not as significant because execution time dominates. The Cardinality Q-Error simulation performs only slightly better than the simple One Step and 1-Join schemes. The flexibility of being able to adjust more often pays off for longer queries, which was definitely not the case for shorter queries.

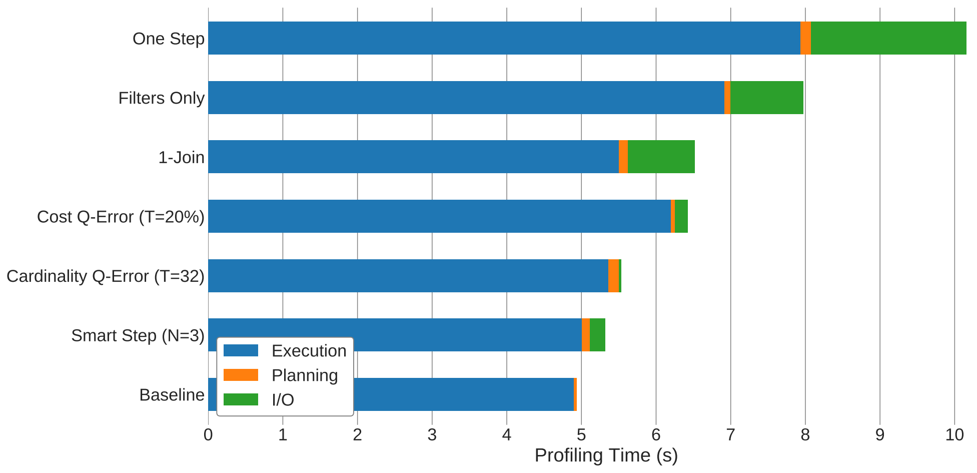


Figure 10: JOB query profiling of the 20 shortest queries in baseline DuckDB compared with re-optimization schemes.

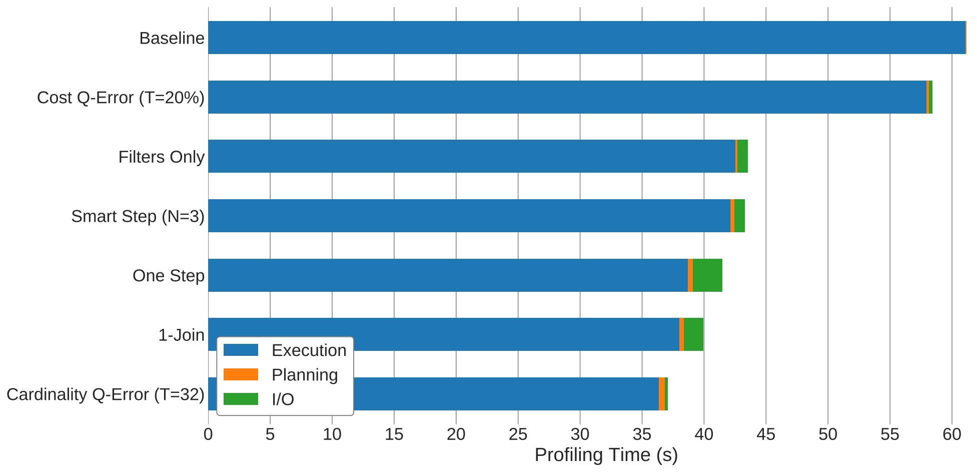


Figure 11: JOB query profiling of the 20 longest queries in baseline DuckDB compared with re-optimization schemes.

The difference in results between the longest and shortest queries raises the question how re-optimization performs on the 'average' query. By leaving out the longest- and shortest 20 queries (removing 'outliers'), a workload with 73 average queries remains. Results on this subset of queries is shown in Figure 12. There is less difference in performance between the schemes on this subset, and baseline DuckDB performs quite well. Cardinality Q-Error comes out on top.

The difference in performance between the different subsets of queries shows that no single scheme performs well in every setting. Creating a scheme that is able to perform well on a wide variety of workloads, perhaps parameterized, is a difficult challenge, and an interesting subject of future work.

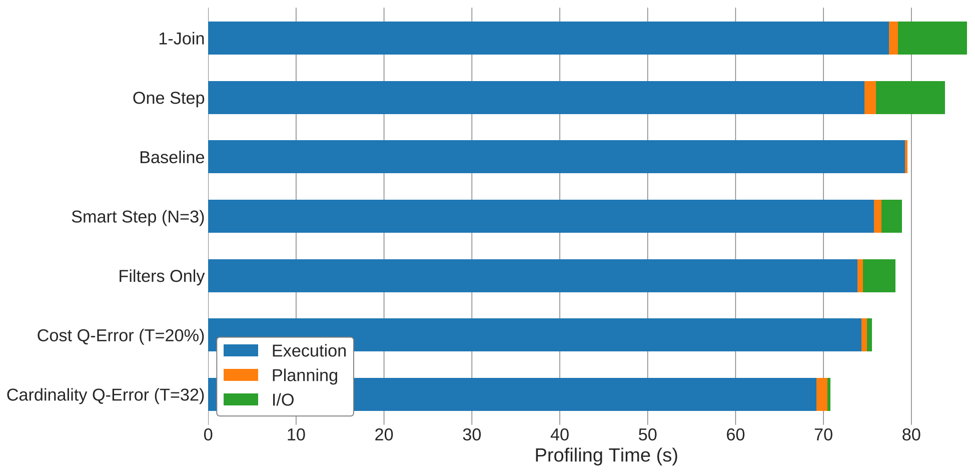


Figure 12: JOB query profiling of all queries except the longest- and shortest 20 in baseline DuckDB compared with re-optimization schemes.

4.2 Plan Cost

For each scheme, only the results of the most successful parameter configuration in terms of overall benchmark run-time are shown in the tables in this section. The complete results can be found in Appendix B.

Relative Cost The cost of generated query plans was compared to the lowest-cost plan that was found by at least one scheme. The percentage of queries falling in a relative cost range are shown in Table 1.

Despite having poor CE as its principal input, DuckDB's optimizer chose query plans with a much lower cost than re-optimization schemes several times. One reason for this is that cardinality estimates, even when they are inaccurate, can capture approximately the right cardinality ratio between relations, causing an efficient join order to be chosen. As intermediate results are materialized, their actual cardinality is measured, which can lead to a re-optimization that chooses a different, sub-optimal query plan. As such, re-optimization schemes were observed to pick plans with over 10 times the cost of the lowest-cost plan that was found at about the same rate as baseline DuckDB.

Total Cost Re-optimization schemes can lower the cost of JOB significantly, shown in Table 2. Baseline DuckDB has a higher overall cost than any re-optimization scheme. These results are in line with Figure 9, with the exception of Smart Step with parameter $N = 3$, which has the same cost as baseline, while it reduces runtime the most out of the real schemes. One explanation could be that while join order is not much better, choice of join strategy is improved. Smart Step, by materializing the child of a 'risky' join, allows the optimizer to make more informed decisions for these

Table 1: Percentage of queries with a relative execution plan cost compared to the lowest-cost plan that was found by at least one scheme.

Scheme	$< \mathbf{1.2}$	$[\mathbf{1.2}, \mathbf{1.5})$	$[\mathbf{1.5}, \mathbf{2})$	$[\mathbf{2}, \mathbf{5})$	$[\mathbf{5}, \mathbf{10})$	$> \mathbf{10}$
Baseline	48.7%	15.0%	14.2%	11.5%	3.5%	7.1%
Filters Only	66.4%	11.5%	8.8%	5.3%	0.9%	7.1%
One Step	73.5%	6.2%	6.2%	8.8%	0.0%	5.3%
1-Join	62.8%	8.0%	8.8%	12.4%	0.9%	7.1%
Smart Step (N=3)	48.7%	15.0%	13.3%	10.6%	2.7%	9.7%
Cardinality Q-Error (T=32)	71.7%	4.4%	9.7%	8.8%	0.9%	4.4%
Cost Q-Error (T=20%)	65.5%	15.9%	7.1%	3.5%	0.9%	7.1%

joins. Join strategy can make a big difference: Leis et al. found that disabling nested loop joins in PostgreSQL reduced JOB run-time significantly [6], because the cost of erroneously choosing this join (due to cardinality misestimations) strategy is big. Furthermore, the cost model does not take join strategy into account, therefore cost does not reflect join strategies chosen by the optimizer.

Table 2: Total cost of JOB in terms of the cost model (sum of intermediate result cardinalities).

Scheme	**Total Cost**
Baseline	631M
Filters Only	489M
One Step	364M
1-Join	467M
Smart Step (N=3)	625M
Cardinality Q-Error (T=32)	352M
Cost Q-Error (T=20%)	560M

4.3 Materialization Characteristics

For each scheme, the number and cardinality of temporary tables created during re-optimization were recorded. Table 3 shows the averages for the most successful parameter configuration in terms of overall benchmark runtime. The complete results are found in Appendix C.

The Cardinality Q-Error scheme materializes small intermediate results in comparison with other schemes. This is explained by DuckDB's simplistic CE, which is to estimate a node's cardinality to be the max of its children, therefore always estimating a selectivity of 1. In many cases the selectivity, and therefore the cardinality, are much lower, resulting in a high cardinality q-error, triggering materialization of the filter node. As such, this scheme tends to materialize small tables.
By combining these results with I/O profiling results, the assumption from section 3.3.1, that the excess in total query time, that is not covered by the

Table 3: Materialization characteristics of re-optimization schemes. *Materializations* are the per-query average number of temporary table creations, whereas *Cardinality* is the average size of these tables.

Scheme	**Materializations**	**Cardinality**
Baseline	0.00	$0.0K$
Filters Only	5.57	$290.9K$
One Step	11.21	$337.2K$
1-Join	5.65	$877.9K$
Smart Step (N=3)	3.03	$320.2K$
Cardinality Q-Error (T=32)	5.15	$39.3K$
Cost Q-Error (T=20%)	1.65	$326.9K$

profiling categories, belongs to the time it takes to write data to a table, can be tested. Figure 13 plots these against each other, with a correlation of $r = 0.96$, supporting the assumption.

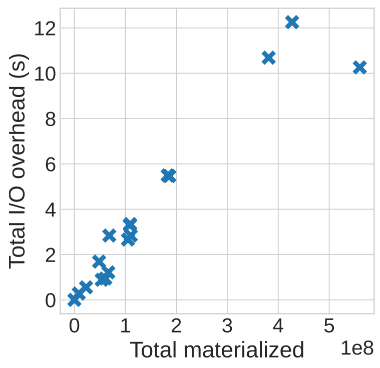


Figure 13: Total I/O overhead of all schemes against total size of materialized intermediate tables.

4.4 Query Plan Analysis

This section describes how the Cardinality Q-Error ($N = 32$) re-optimization scheme executes two interesting query plans.

Query 5a Out of all queries in JOB, the execution time of query 5a was slowed down the most by this re-optimization scheme, from $0.17s$ to $0.94s$, more than 5 times slower than the baseline duration. The cost of both the baseline plan and re-optimized plan are $154K$. Figure 14 shows the plan initially generated by the optimizer. Above it query 5a is shown. In the figure, the true cardinality of a node in the plan is denoted with T, the predicted cardinality with P, and the cardinality of base tables with C.

```
SELECT MIN(t.title) AS typical_european_movie
FROM company_type AS ct,
     info_type AS it,
     movie_companies AS mc,
     movie_info AS mi,
     title AS t
WHERE ct.kind = 'production_companies'
  AND mc.note LIKE '%(theatrical)%'
  AND mc.note LIKE '%(France)%'
  AND mi.info IN ('Sweden',
                  'Norway',
                  'Germany',
                  'Denmark',
                  'Swedish',
                  'Denish',
                  'Norwegian',
                  'German')
  AND t.production_year > 2005
  AND t.id = mi.movie_id
  AND t.id = mc.movie_id
  AND mc.movie_id = mi.movie_id
  AND ct.id = mc.company_type_id
  AND it.id = mi.info_type_id;
```

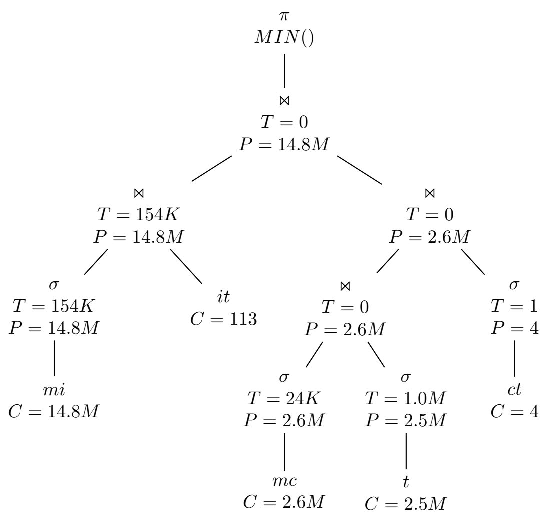


Figure 14: Initial plan for JOB query 5a.

The scheme re-optimized query 5a three additional times. The query plan after each iteration is found in Appendix D, as well as the queries that were used to materialize the intermediate results. Even when ignoring planning and I/O overhead, the execution time of this query is much slower with re-optimization than without.

The join between the filtered base tables *mc* and *t* yields no matches. Without re-optimization, execution starts by pulling chunks from these base tables up the pipeline. No chunk are emitted by the join, therefore no chunks ever need to be pulled from tables *mi*, *it* and *ct*, because there is nothing to join them with. As such, base tables *mi* and *it* are never read, and the join between them is not performed. In reality the plan cost is much less than the cost model predicts.

Re-optimization on the other hand chooses to materialize a filter node due to a large q-error, before re-optimizing. The re-optimized plan has a different join order, causing more base tables to be read as well as causing an intermediate result of $154K$ to be materialized in the second iteration, incurring a high cost that is avoided when executing the plan normally. This is partially caused by an implementation detail: re-optimization prioritizes materialization of the right children of joins first, rather than e.g. the deepest join.

Query 16a Out of all queries in JOB, the execution time of query 16a was sped up the most by this re-optimization scheme, from 4.45s to 0.65s, more than 7 times faster than the baseline duration. The baseline plan cost is $37M$, while the cost of the re-optimized plan is only 964K. Figure 15 shows the plan initially generated by the optimizer. Above it query 5a is shown.

```
SELECT MIN(an.name) AS cool_actor_pseudonym ,
       MIN(t.title) AS series_named_after_char
FROM aka_name AS an,
     cast_info AS ci ,
     company_name AS cn ,
     keyword AS k,
     movie_companies AS mc,
     movie_keyword AS mk,
     name AS n,
     title AS t
WHERE cn.country_code ='[us]'
  AND k.keyword ='character-name-in-title'
  AND t.episode_nr >= 50
  AND t.episode_nr < 100
  AND an.person_id = n.id
  AND n.id = ci.person_id
  AND ci.movie_id = t.id
  AND t.id = mk.movie_id
  AND mk.keyword_id = k.id
  AND t.id = mc.movie_id
  AND mc.company_id = cn.id
  AND an.person_id = ci.person_id
  AND ci.movie_id = mc.movie_id
  AND ci.movie_id = mk.movie_id
  AND mc.movie_id = mk.movie_id;
```

The scheme re-optimized query 16a six additional times. In the figure, the true cardinality of a node in the plan is denoted with T, the predicted cardinality with P, and the cardinality of base tables with C. The query plan after each iteration is found in Appendix D, as well as the queries that were used to materialize the intermediate results.

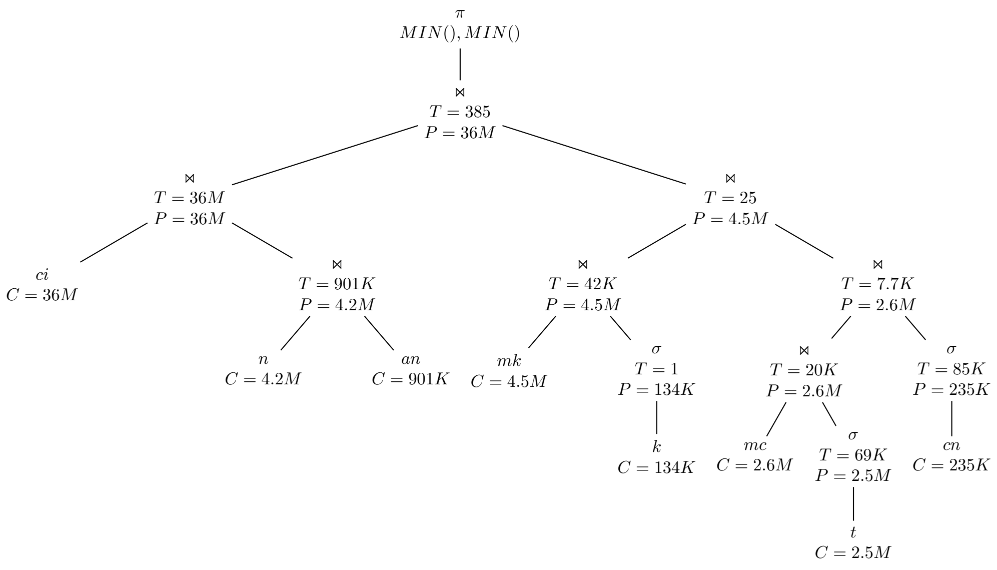

π
$MIN(), MIN()$
$\bowtie$ $T = 385$ $P = 36M$
$\bowtie$ $T = 36M$ $P = 36M$
ci $C = 36M$
$\bowtie$ $T = 901K$ $P = 4.2M$
n $C = 4.2M$
an $C = 901K$
$\bowtie$ $T = 25$ $P = 4.5M$
$\bowtie$ $T = 42K$ $P = 4.5M$
mk $C = 4.5M$
σ $T = 1$ $P = 134K$
k $C = 134K$
$\bowtie$ $T = 7.7K$ $P = 2.6M$
$\bowtie$ $T = 20K$ $P = 2.6M$
mc $C = 2.6M$
σ $T = 69K$ $P = 2.5M$
t $C = 2.5M$
σ $T = 85K$ $P = 235K$
cn $C = 235K$

Figure 15: Initial plan for JOB query 16a.

The $37M$ cost of the initial plan is dominated by the join between *ci*, *n*, and *an*, which costs $36M$. This is a *disastrous* plan, caused by one *disastrous* join. After materializing small intermediate results in iterations 1–5, this join is avoided by the re-optimization scheme after the sixth iteration.

5 Discussion

In this section, results are analysed in order to gain insights and answer research questions. Based on the analysis, and on insights gained through working with DuckDB, ideas for future work are suggested.

5.1 Result Analysis

The answers to research questions 3 and 4, and their sub-questions, are given by analyzing the results.

Trade-Off As shown in Figure 10, re-optimizing the shortest running queries slows down processing. In theory, re-optimization should decrease execution time due to an improved query plan, at the cost of increasing planning and I/O time. Increases in planning and I/O time are observed, but a reduction in execution time is not. The reason for this is that materializing intermediate results breaks the execution pipeline that would otherwise speed up inter-operator processing. For these queries, the I/O costs are huge in comparison with the execution costs, especially with schemes that materialize many/all filter nodes (One Step, Filters Only, Smart Step with $N = 2$).

On the contrary, re-optimizing the longest running queries always results in a reduced query time, shown in Figure 11. Here, the real schemes that materialize the most often (One Step, 1-Join) perform the best, almost as well as the best simulated schemes, despite their high I/O cost. Although the trade-off between the costs and benefits of re-optimization is huge for short queries, there is not much of a trade-off for long queries: re-optimization is almost always worth it.

Furthermore, in a DBMS that was built with re-optimization in mind that would be able to suspend execution mid-query, intermediate results would not have to be materialized as a temporary table, but would rather be kept as an object in the stack (e.g. `DataChunk` in DuckDB). This would entail less movement of data, meaning less I/O costs, and a higher performance for schemes that materialize often.

Real vs. Simulated The top simulated schemes perform best on long queries by a margin (Figure 11), and only slightly worse than baseline on short queries (Figure 10). Because long queries have a much larger impact on overall benchmark performance, simulated schemes come out on top (Figure 9). Simulated schemes are able to *react* to differences in estimated and measured statistics, deciding when to optimize on the fly, in contrast with real schemes, which *statically* select these moments. Therefore, it is no surprise that simulated schemes perform better.

However, the margin by which these perform better for the longest queries in the benchmark is lower than this author expected. Some of the real schemes that perform poorly when looking at overall benchmark performance, perform very well on long queries. By combining these schemes with a policy that only re-optimizes long queries, their overall performance could be closer to the top simulated schemes. This shows that most types of re-optimization, even such naive schemes as 1-Join, can improve end-to-end performance on JOB, even if re-optimization is an afterthought rather than a main objective of a DBMS.

Robustness From Table 2 it is clear that re-optimization reduces the total cost of JOB by a significant amount, down to almost half the cost of baseline DuckDB depending on the scheme. This is in line with the latency experiments, reinforcing the notion that re-optimization reduces query latency by improving join order.

Table 1 shows that for many queries, re-optimized query plans are often closer to the optimal plan. This is in line with the reduced number of 'outliers' seen in Figure 8: disastrous plans are consistently avoided. It can be concluded that re-optimization has a more robust performance on this workload.

5.2 Simulation: Realistic?

The simulation proposed by Perron et al. [11] (described in section 3.2.2) simulates a DBMS that can suspend execution after each operator. However, database systems that have a Volcano-style execution engine do not have to process all of the data through an operator before starting the next operator, unless it is a blocking operator e.g. hash-join, sort, etc. Instead, tuples are pulled through the operators row by row "tuple-at-a-time". This approach is known as pipelined query execution, which avoids the need to fully materialize intermediate results that may not fit in memory and would therefore significantly slow down processing due to the high costs of writing them to disk. It also allows queuing multiple instructions (for multiple operators down the pipeline) that are to be executed on a single tuple sequentially, allowing the CPU to keep the tuple in cache. This speeds up processing, if there are not too many cache misses (mispredicted CPU instructions).

DuckDB has a vector-Volcano model, which does a better job at exploiting modern CPUs. Data is stored in columns rather than rows, and is processed using fixed-size vectors (chunks of a column) at a time, similar to PAX [24]. Because a single instruction usually has to process an entire chunk, and not just one tuple, it can be loaded once for the chunk, allowing for more efficient CPU cycle usage. This also reduces the number of cache misses, speeding up execution even further. This, however, complicates the simulation, because the simulation does not take the benefits of pipelining into account at all.

DuckDB's execution model is more heavily pipelined than that of PostgreSQL. The difference lies in PostgreSQL's tuple-at-a-time execution model, in contrast with DuckDB's chunk-at-a-time that uses CPU cycles more efficiently. Therefore, the difference in execution time of processing a query plan one operator at a time (as is assumed in the simulation) rather than as a pipelined whole are smaller in PostgreSQL than in DuckDB, because the performance gain of pipelining in PostgreSQL is not as significant. Perron et al. acknowledge pipelining, but they believe that their simulation provides a reasonable approximation for an upper bound of the cost of re-optimization schemes. It could be argued that they cannot be sure of this because the pipeline is not respected in their simulation, but this author is inclined to side with them for the reasons given.

Since the effect of pipelining is more noticeable for more heavily pipelined column-stores, the simulation seems however less representative in the case of DuckDB. To make it realistic, pipelining could for instance be completely disabled, forcing operator-at-a-time processing. A reasonable estimation of the performance of this modified simulation would lie somewhere between the simulated and real re-optimization schemes (these do not differ much for long queries, see Figure 11), which is a significant speedup compared to the plan-first execute-next approach.

5.3 Parallelism

Parts of query plans that are independent, could be evaluated concurrently, raising the question what effect parallelism has on re-optimization. Antoshenkov's competition model [25], [26] starts multiple evaluation plans of the same query at the same time, stopping sub-optimal plans when it becomes clear that one plan is better than the others. An obvious application of the competition model in this context is to let the plan-first execute-next approach compete with the re-optimization approach. This ensures that short queries are never slowed down.

The approach can also be extended to intra-query parallelism. Independent parts of a query plan can be materialized at the same time. When the first part is materialized, re-optimization might be triggered. At this time, the other threads are still processing, but the cardinality estimation of the part that they are processing is becoming more precise. An informed decision can be made whether the other independently processed parts should be used in the re-optimized plan, or dumped.

Parallelism may also be introduced within operators, where e.g. threads process one chunk of data. Ignoring the fact that it could prove difficult to preserve row order in such a setting, this author does not believe that this kind of parallelism has a very different effect on the re-optimization vs. plan-first execute-next paradigms. Both the speedup of pipelined execution and that of operator-at-a-time execution should increase linearly with the number of threads.

5.4 Limitations

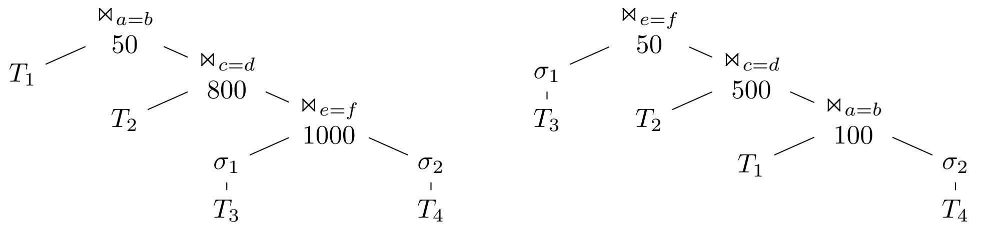
The results in this work showcase the potential of re-optimization, even though the implementation has some limitations. Some of these limitations can be alleviated by building on ideas that have been proposed in related work, which are discussed here.

Plan-First Execute-Next Although the real re-optimization schemes presented in this work improve DuckDB's performance on JOB, they essentially exist out of multiple plan-first execute-next steps. Ideally, statistics are measured during execution, and execution is suspended whenever the optimizer decides it should re-optimize the remainder of the plan. This was not possible because re-optimization was implemented into DuckDB as an afterthought, rather than as an essential part of its execution engine. Building a DBMS with this quality is outside of the scope of this work, and remains a subject of future research.

Cut Your Losses One limitation is that this implementation cannot fully recover from some of the mistakes made by the optimizer. In some instances it is better to discard a materialized intermediate result, if the cost of continuing execution using it is higher than starting over without it, having gained statistical information about the intermediate table. An example of this is illustrated in Figure 16. Suppose σ_1, σ_2 are filter operations that have a low selectivity, but highly correlate in their respective selections of rows from T_3, T_4. Joining these yields a large intermediate result of 1000, with the remaining plan costing 850 (following the cost model from section 3.3.2). Knowing this information, a plan with a cost of 650 can be constructed. Clearly, starting over would be the right choice. This is realised by injecting cardinalities into the optimizer, which could be the subject of future work.

In a pipelined execution model $\bowtie_{e=f}$ may not ever be fully materialized if it is non-blocking. However, the large error in cardinality estimation could be detected early by measuring how many tuples flow through the operator. Partial work further down the pipeline could be lost by switching plans.

Quantifying Risk The choice of which node to materialize is highly important, and influences the rest of the plan. Some joins carry more 'risk' than others, which is not taken into account by any of the schemes apart from SmartStep, which identifies risky joins using the structure of the query plan. The results show that this is more effective than blindly picking joins with the NJoin scheme, but ideally not only the structure of the plan, but also the data that is being processed should be taken into account when quantifying risk.



(a) Chosen plan (b) Optimal plan

Figure 16: A plan where the initial choice of join should be discarded.

The cardinality estimation method proposed by Babcock & Chaudhuri [8] uses probability distributions instead of a single-point estimate to generate *robust* query plans. This cardinality estimation technique could be used to more accurately identify risky joins that have an uncertain cardinality estimate, which would almost certainly improve the performance of SmartStep. The uncertainty of the join would be more important for the re-optimization mechanism than the actual cardinality estimate in such a scheme.

5.5 Generalization

At the time of writing, DuckDB is still in development, and some of its components are unfinished. Among these are the cardinality estimation module and the cost model. Currently, there is almost no CE: a node's cardinality is estimated to be the maximum cardinality of its children. The cost model used for join order enumeration, explained in section 3.3.2, minimizes intermediate results, but is too simplistic because it does not take e.g. cost of join strategy into account. It is scheduled for improvement along with the CE component. This raises the question whether the results presented in this work generalize to a DBMS that does have finished components.

The question can be answered in two parts. First, consider the exponentiality of error propagation through the query plan. All five commercial database systems tested in the article that introduced JOB [6] were observed to produce cardinality estimates that were off by several orders of magnitude as the number of joins increased. Likely, there is room for improvement in this area, but on the other hand this result seems inevitable; small errors in each step of the plan can result in large misestimations when the number of joins is high. Second, the results by Perron et al. [11], where perfect cardinality estimates are given PostgreSQL's query optimizer by an oracle, show us that even with perfect cardinality estimates for joins with up to 4 tables, re-optimization still yields a reduced execution time. Given these observations, it is safe to say that the result that query re-optimization significantly reduces run-time on this workload, will generalize to a DBMS that has finished CE and cost model components.

5.6 Research Questions

Research question 1 has been answered by reviewing related literature, apart from 1b, which has partly been answered by experimentation in this work. Re-optimization schemes vary in their effectiveness, but for long queries, any re-optimization is better than none.

Research question 2 has been answered by implementing re-optimization in DuckDB. Of course, there are many design decisions to be made, and therefore many ways to implement re-optimization. It was decided to keep the implementation simple, sacrificing some performance for ease of implementation. Additionally, it was discovered that a simulation has some caveats, which have been discussed in this section.

Research questions 3 & 4 have been answered in-depth through experimentation and result analysis in this section, and the previous section. It was discovered that for long-running queries the trade-off between costs and benefits is not a difficult one, while for short-running queries it is, leaving room for interesting re-optimization policies. Across JOB, robustness was improved by avoiding disastrous plans with a high cost.

6 Conclusion

In this work it has become evident that re-optimization significantly improves end-to-end latency in a workload that stresses the CE component of the query optimizer. Delaying the decision on join order can help produce query plans that are closer to optimal regarding join order, by using statistics that are collecting during execution. This approach works better for long running queries, in which case simple schemes that process one operator at a time provide a reduced runtime.

In DuckDB, with the workload used in this research, the cost of optimizing a plan multiple times and writing intermediate results to a temporary table pales in comparison to the benefit of a better plan. This is true even for simple re-optimization schemes that materialize each intermediate result. The reason for this is that DuckDB is a main-memory DBMS: the cost of reading and writing data is much lower than for a disk-based system.

Implementing re-optimization in an existing DBMS is challenging, and brings limitations that prevent it from reaching its potential. A true implementation would require an overhaul of the execution engine, which is one of the reasons it has not been adopted in commercial database systems. Depending on the execution model, a close-to-realistic simulation can be made to get an approximation of how a true implementation would perform. Unfortunately, the more a system benefits from pipelined execution, the less realistic the simulation becomes.

Clearly, the next step in re-optimization is to fully implement it by building a new DBMS or overhauling an existing execution engine. This would allow for more sophisticated schemes that have more advanced decision making behaviour like POP [20] or Rio [21]. Because it performs better on longer queries, re-optimization will likely have even more benefit in the distributed main-memory DBMS setting where more data is processed, such as Spark SQL. Reducing the intermediate results could greatly reduce the need to move data between machines, which is a costly operation.

As a final remark, this author would like to discuss why this research has become important especially now, after re-optimization has not been researched actively for more than 10 years. Developments in the database field, as well as advances in hardware, have alleviated the need for adaptive query optimization, by drastically increasing data bandwidth. This can explain why the plan-first execute-next approach has remained popular. However, over these years, data has been collected in larger quantities, and there has been an increasing demand for database technology in analytical processing. This use-case is precisely the scenario where the adaptive approach excels, and is precisely why re-optimization is becoming more important.

References

[1] T. Connolly and C. Begg, *Database Systems: A Practical Approach to Design, Implementation, and Management*, English, 6th. United Kingdom: Pearson Education Limited, 2015, ISBN: 9781292061184.

[2] T. Ibaraki and T. Kameda, "On the Optimal Nesting Order for Computing N-relational Joins", *ACM Trans. Database Syst.*, vol. 9, no. 3, pp. 482–502, Sep. 1984, ISSN: 0362-5915. DOI: `10.1145/1270.1498`. [Online]. Available: `http://doi.acm.org/10.1145/1270.1498`.

[3] G. Moerkotte and T. Neumann, "Dynamic Programming Strikes Back", in *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '08, Vancouver, Canada: ACM, 2008, pp. 539–552, ISBN: 978-1-60558-102-6. DOI: `10.1145/1376616.1376672`. [Online]. Available: `http://doi.acm.org/10.1145/1376616.1376672`.

[4] T. Neumann and B. Radke, "Adaptive Optimization of Very Large Join Queries", in *Proceedings of the 2018 International Conference on Management of Data*, ser. SIGMOD '18, Houston, TX, USA: ACM, 2018, pp. 677–692, ISBN: 978-1-4503-4703-7. DOI: `10.1145/3183713.3183733`. [Online]. Available: `http://doi.acm.org/10.1145/3183713.3183733`.

[5] R. Marcus and O. Papaemmanouil, "Deep Reinforcement Learning for Join Order Enumeration", in *Proceedings of the First International Workshop on Exploiting Artificial Intelligence Techniques for Data Management*, ser. aiDM'18, Houston, TX, USA: ACM, 2018, 3:1–3:4, ISBN: 978-1-4503-5851-4. DOI: `10.1145/3211954.3211957`. [Online]. Available: `http://doi.acm.org/10.1145/3211954.3211957`.

[6] V. Leis, A. Gubichev, A. Mirchev, P. Boncz, A. Kemper, and T. Neumann, "How Good Are Query Optimizers, Really?", *Proc. VLDB Endow.*, vol. 9, no. 3, pp. 204–215, Nov. 2015, ISSN: 2150-8097. DOI: `10.14778/2850583.2850594`. [Online]. Available: `http://dx.doi.org/10.14778/2850583.2850594`.

[7] Y. E. Ioannidis and S. Christodoulakis, "On the Propagation of Errors in the Size of Join Results", *SIGMOD Rec.*, vol. 20, no. 2, pp. 268–277, Apr. 1991, ISSN: 0163-5808. DOI: `10.1145/119995.115835`. [Online]. Available: `https://doi.org/10.1145/119995.115835`.

[8] B. Babcock and S. Chaudhuri, "Towards a Robust Query Optimizer: A Principled and Practical Approach", in *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '05, Baltimore, Maryland: ACM, 2005, pp. 119–130, ISBN: 1-59593-060-4. DOI: `10.1145/1066157.1066172`. [Online]. Available: `http://doi.acm.org/10.1145/1066157.1066172`.

[9] V. Leis, B. Radke, A. Gubiches, A. Kemper, and T. Neumann, "Cardinality Estimation Done Right: Index-Based Join Sampling", in *Proceedings of the 2017 8th Biennial Conference on Innovative Data Systems Research*, ser. CIDR '17, Chaminade, California, USA, Jan. 2017.

[10] A. Kipf, T. Kipf, B. Radke, V. Leis, P. A. Boncz, and A. Kemper, "Learned Cardinalities: Estimating Correlated Joins with Deep Learning", *CoRR*, vol. abs/1809.00677, 2018. arXiv: `1809.00677`. [Online]. Available: `http://arxiv.org/abs/1809.00677`.

[11] M. Perron, Z. Shang, T. Kraska, and M. Stonebraker, "How I Learned to Stop Worrying and Love Re-optimization", in *Proceedings of the 2019 IEEE 35th International Conference on Data Engineering (ICDE)*, Apr. 2019, pp. 1758–1761. DOI: `10.1109/ICDE.2019.00191`.

[12] T. Neumann and C. Galindo-Legaria, “Taking the Edge off Cardinality Estimation Errors using Incremental Execution”, in *Datenbanksysteme für Business, Technologie und Web (BTW) 2019*, V. Markl, G. Saake, K.-U. Sattler, G. Hackenbroich, B. Mitschang, T. Härder, and V. Köppen, Eds., Bonn: Gesellschaft für Informatik e.V., 2013, pp. 73–92.

[13] S. Chaudhuri, “Query Optimizers: Time to Rethink the Contract?”, in *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD ’09, Providence, Rhode Island, USA: Association for Computing Machinery, 2009, pp. 961–968, ISBN: 9781605585512. DOI: `10.1145/1559845.1559955`. [Online]. Available: `https://doi.org/10.1145/1559845.1559955`.

[14] P. G. 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*, ser. SIGMOD ’79, Boston, Massachusetts: Association for Computing Machinery, 1979, pp. 23–34, ISBN: 089791001X. DOI: `10.1145/582095.582099`. [Online]. Available: `https://doi.org/10.1145/582095.582099`.

[15] G. Graefe, “Volcano - An Extensible and Parallel Query Evaluation System”, *IEEE Trans. on Knowl. and Data Eng.*, vol. 6, no. 1, pp. 120–135, Feb. 1994, ISSN: 1041-4347. DOI: `10.1109/69.273032`. [Online]. Available: `https://doi.org/10.1109/69.273032`.

[16] L. Amsaleg, A. Tomasic, M. J. Franklin, and T. Urhan, “Scrambling Query Plans to Cope with Unexpected Delays”, in *Fourth International Conference on Parallel and Distributed Information Systems*, Dec. 1996, pp. 208–219. DOI: `10.1109/PDIS.1996.568681`.

[17] T. Urhan, M. J. Franklin, and L. Amsaleg, “Cost-Based Query Scrambling for Initial Delays”, in *Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD ’98, Seattle, Washington, USA: Association for Computing Machinery, 1998, pp. 130–141, ISBN: 0897919955. DOI: `10.1145/276304.276317`. [Online]. Available: `https://doi.org/10.1145/276304.276317`.

[18] R. Avnur and J. M. Hellerstein, “Eddies: Continuously Adaptive Query Processing”, *SIGMOD Rec.*, vol. 29, no. 2, pp. 261–272, May 2000, ISSN: 0163-5808. DOI: `10.1145/335191.335420`. [Online]. Available: `https://doi.org/10.1145/335191.335420`.

[19] N. Kabra and D. J. DeWitt, “Efficient Mid-query Re-optimization of Sub-optimal Query Execution Plans”, in *Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD ’98, Seattle, Washington, USA: ACM, 1998, pp. 106–117, ISBN: 0-89791-995-5. DOI: `10.1145/276304.276315`. [Online]. Available: `http://doi.acm.org/10.1145/276304.276315`.

[20] V. Markl, V. Raman, D. Simmen, G. Lohman, H. Pirahesh, and M. Cilimdzic, “Robust Query Processing through Progressive Optimization”, in *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD ’04, Paris, France: Association for Computing Machinery, 2004, pp. 659–670, ISBN: 1581138598. DOI: `10.1145/1007568.1007642`. [Online]. Available: `https://doi.org/10.1145/1007568.1007642`.

[21] S. Babu, P. Bizarro, and D. DeWitt, “Proactive Re-Optimization”, in *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD ’05, Baltimore, Maryland: ACM, 2005, pp. 107–118, ISBN: 1-59593-060-4. DOI: `10.1145/1066157.1066171`. [Online]. Available: `http://doi.acm.org/10.1145/1066157.1066171`.

[22] P. Boncz, M. Zukowski, and N. Nes, “MonetDB/X100: Hyper-Pipelining Query Execution”, *2nd Biennial Conference on Innovative Data Systems Research, CIDR 2005*, Jan. 2005.

[23] M. Raasveldt and H. Mühleisen, “DuckDB: An Embeddable Analytical Database”, in *Proceedings of the 2019 International Conference on Management of Data*, ser. SIGMOD ’19, Amsterdam, Netherlands: ACM, 2019, pp. 1981–1984, ISBN: 978-1-4503-5643-5. DOI: `10.1145/3299869.3320212`. [Online]. Available: `http://doi.acm.org/10.1145/3299869.3320212`.

[24] A. Ailamaki, D. J. DeWitt, and M. D. Hill, “Data page layouts for relational databases on deep memory hierarchies”, *The VLDB Journal*, vol. 11, no. 3, pp. 198–215, Nov. 2002, ISSN: 1066-8888. DOI: `10.1007/s00778-002-0074-9`. [Online]. Available: `https://doi.org/10.1007/s00778-002-0074-9`.

[25] G. Antoshenkov, “Dynamic query optimization in rdb/vms”, in *Proceedings of IEEE 9th International Conference on Data Engineering*, 1993, pp. 538–547.

[26] ——, “Dynamic optimization of index scans restricted by booleans”, in *Proceedings of the Twelfth International Conference on Data Engineering*, 1996, pp. 430–440.

A Query Latency Results

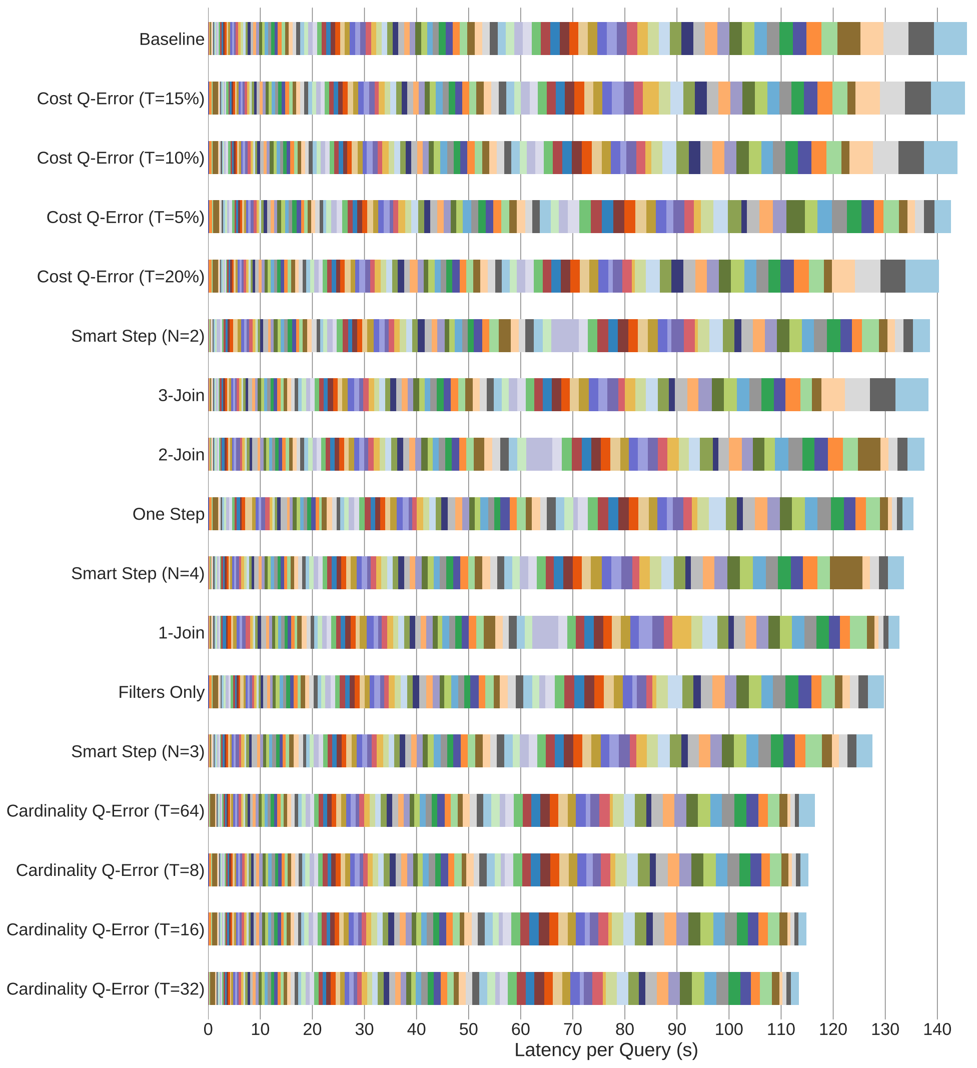


Figure 17: JOB end-to-end latency by query comparison of baseline DuckDB with all re-optimization scheme parameter configurations.

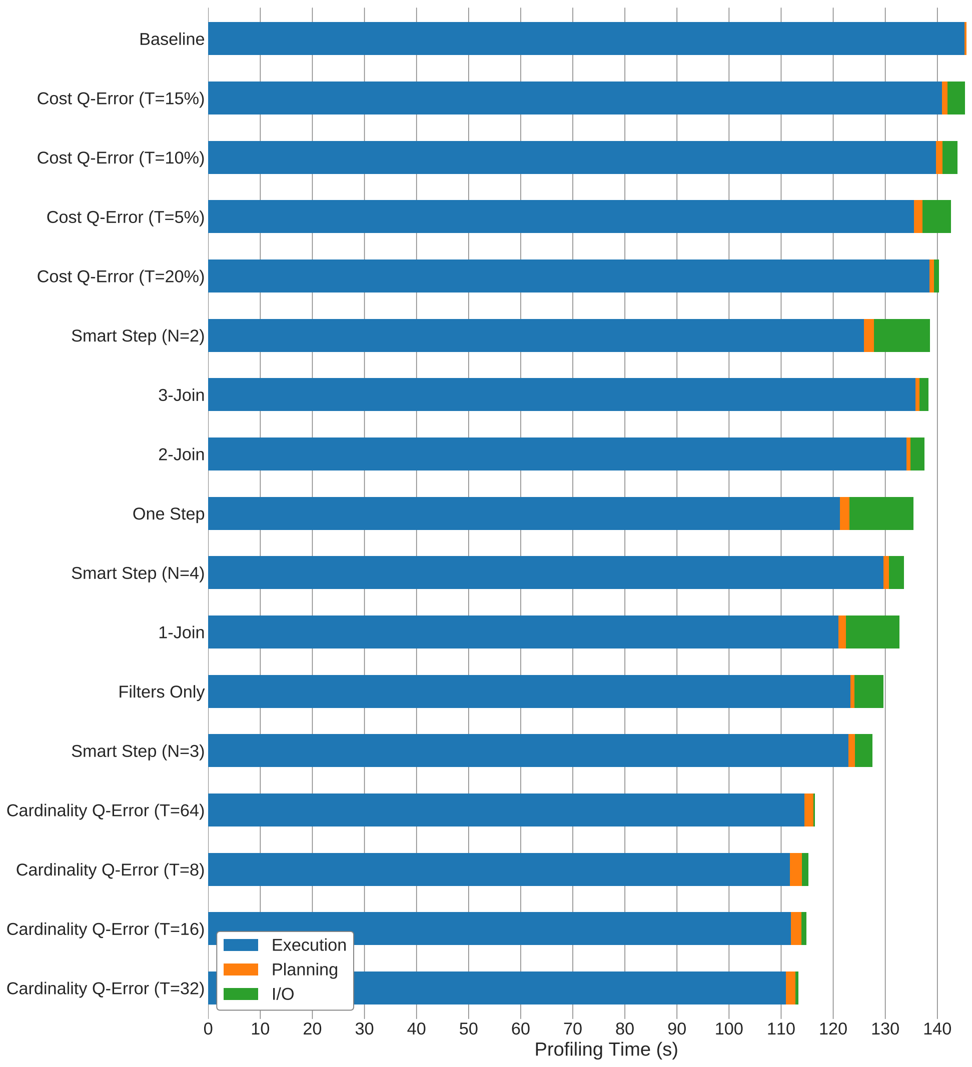


Figure 18: JOB query profiling of baseline DuckDB with all re-optimization scheme parameter configurations.

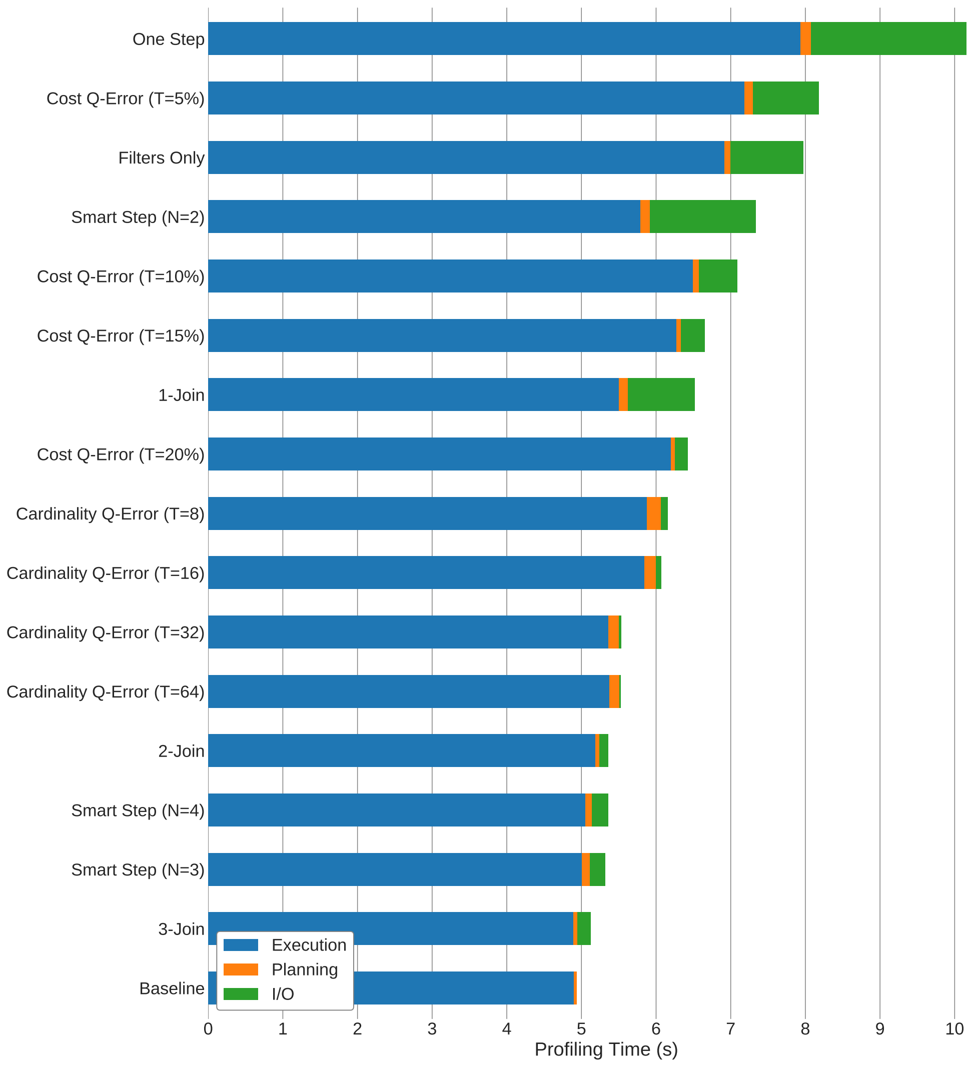


Figure 19: JOB query profiling of the 20 shortest queries in baseline DuckDB compared with all re-optimization scheme parameter configurations.

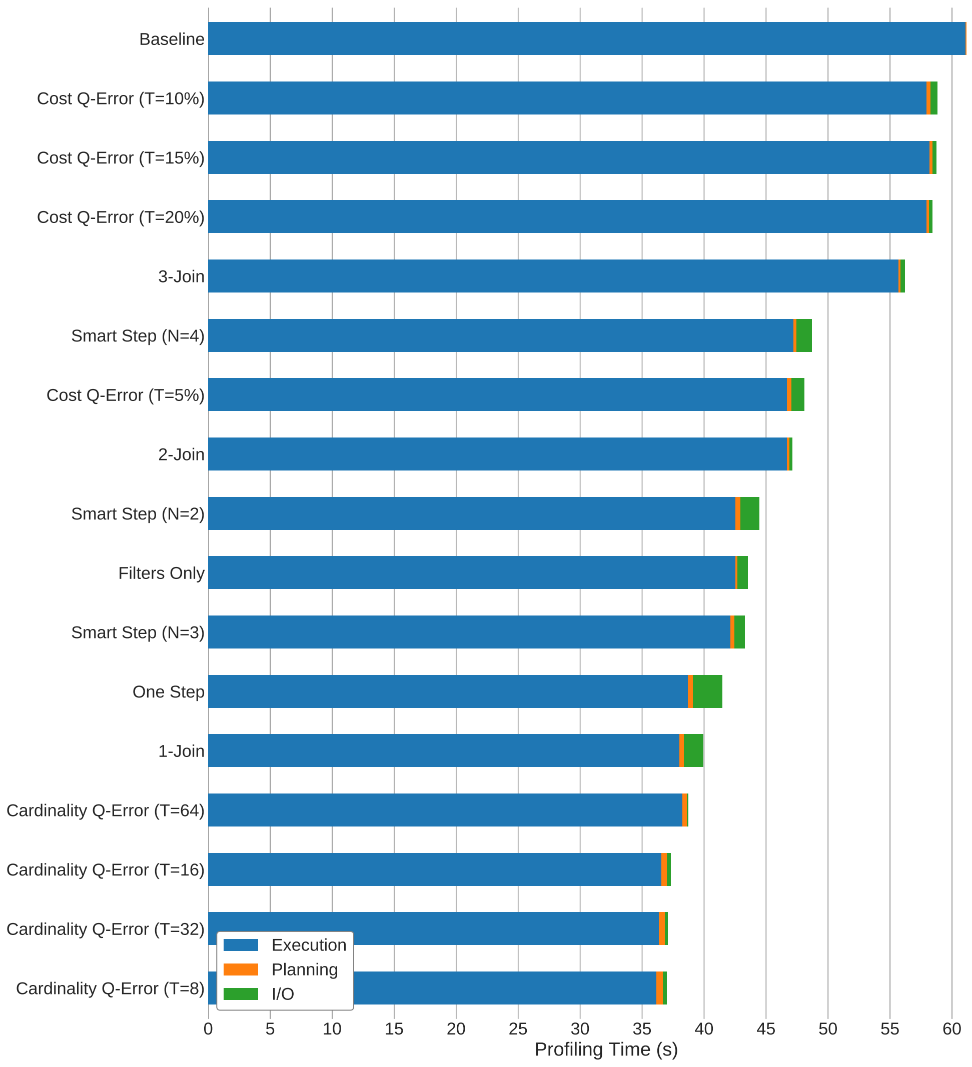


Figure 20: JOB query profiling of the 20 longest queries in baseline DuckDB compared with all re-optimization scheme parameter configurations.

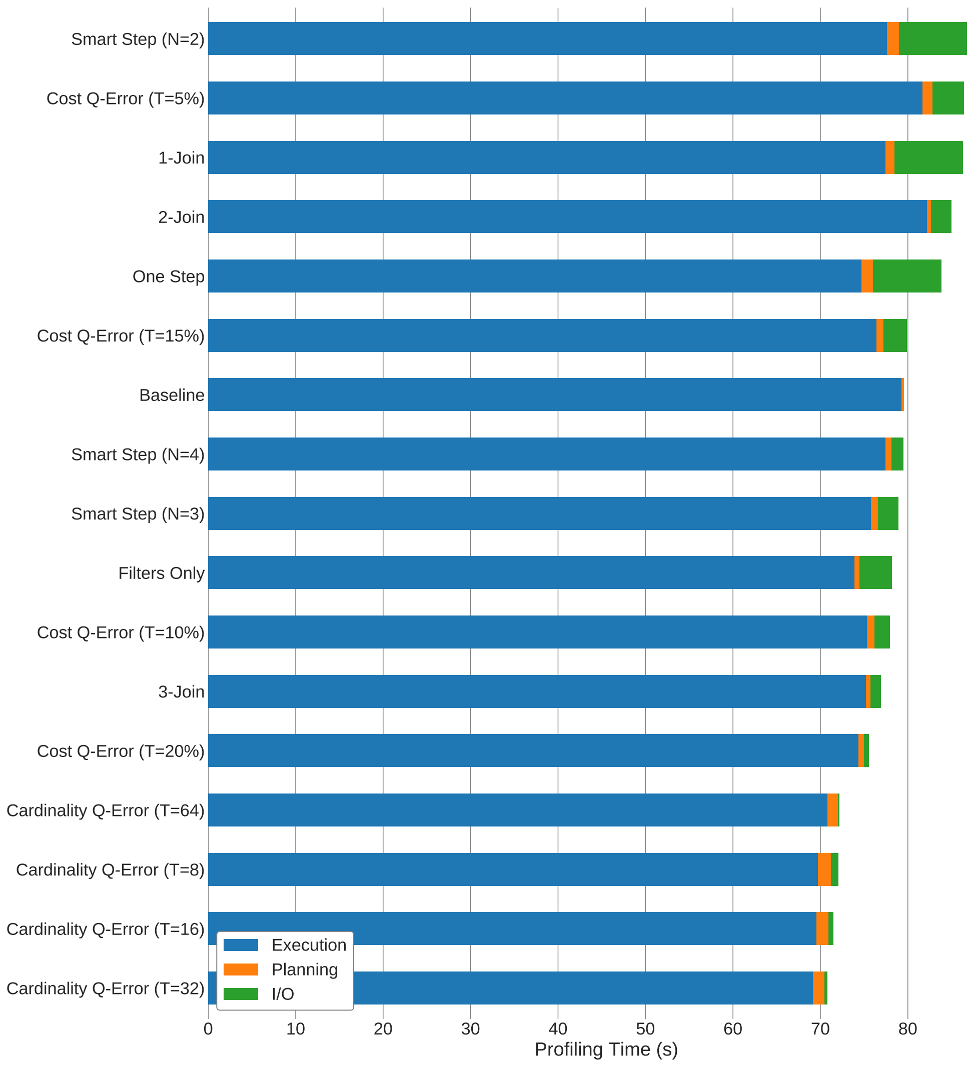


Figure 21: JOB query profiling of all queries except the longest- and shortest 20 in baseline DuckDB compared with all re-optimization scheme parameter configurations.

B Plan Cost Results

Table 4: Percentage of queries with a relative execution plan cost compared to the lowest-cost plan that was found by at least one scheme, all parameter configurations.

Scheme	**< 1.2**	**[1.2, 1.5)**	**[1.5, 2)**	**[2, 5)**	**[5, 10)**	**> 10**
Baseline	48.7%	15.0%	14.2%	11.5%	3.5%	7.1%
Filters Only	66.4%	11.5%	8.8%	5.3%	0.9%	7.1%
One Step	73.5%	6.2%	6.2%	8.8%	0.0%	5.3%
1-Join	62.8%	8.0%	8.8%	12.4%	0.9%	7.1%
2-Join	51.3%	13.3%	12.4%	9.7%	2.7%	10.6%
3-Join	52.2%	12.4%	15.0%	9.7%	3.5%	7.1%
Smart Step (N=2)	61.9%	15.0%	8.8%	4.4%	0.9%	8.8%
Smart Step (N=3)	48.7%	15.0%	13.3%	10.6%	2.7%	9.7%
Smart Step (N=4)	49.6%	14.2%	15.0%	10.6%	3.5%	7.1%
Cardinality Q-Error (T=8)	81.4%	1.8%	3.5%	7.1%	0.9%	5.3%
Cardinality Q-Error (T=16)	78.8%	2.7%	6.2%	6.2%	0.9%	5.3%
Cardinality Q-Error (T=32)	71.7%	4.4%	9.7%	8.8%	0.9%	4.4%
Cardinality Q-Error (T=64)	67.3%	6.2%	10.6%	9.7%	0.9%	5.3%
Cost Q-Error (T=5%)	51.3%	14.2%	10.6%	12.4%	1.8%	9.7%
Cost Q-Error (T=10%)	61.1%	12.4%	7.1%	8.0%	0.9%	10.6%
Cost Q-Error (T=15%)	60.2%	14.2%	10.6%	3.5%	2.7%	8.8%
Cost Q-Error (T=20%)	65.5%	15.9%	7.1%	3.5%	0.9%	7.1%

Table 5: Total cost of JOB in terms of the cost model (sum of intermediate result cardinalities), all parameter configurations.

Scheme	**Total Cost**
Baseline	631M
Filters Only	489M
One Step	364M
1-Join	467M
2-Join	626M
3-Join	629M
Smart Step (N=2)	560M
Smart Step (N=3)	625M
Smart Step (N=4)	626M
Cardinality Q-Error (T=8)	381M
Cardinality Q-Error (T=16)	357M
Cardinality Q-Error (T=32)	352M
Cardinality Q-Error (T=64)	390M
Cost Q-Error (T=5%)	615M
Cost Q-Error (T=10%)	606M
Cost Q-Error (T=15%)	616M
Cost Q-Error (T=20%)	560M

C Materialization Characteristics

Table 6: Materialization characteristics of re-optimization schemes, all parameter configurations. Materializations are the per-query average number of temporary table creations, cardinality is the average size of these tables.

Scheme	**Materializations**	**Cardinality**
Baseline	0.00	0.0K
Filters Only	5.57	290.9K
One Step	11.21	337.2K
1-Join	5.65	877.9K
2-Join	1.52	921.6K
3-Join	1.01	282.4K
Smart Step (N=2)	5.42	622.9K
Smart Step (N=3)	3.03	320.2K
Smart Step (N=4)	2.03	299.2K
Cardinality Q-Error (T=8)	6.35	92.9K
Cardinality Q-Error (T=16)	5.73	82.8K
Cardinality Q-Error (T=32)	5.15	39.3K
Cardinality Q-Error (T=64)	4.64	16.5K
Cost Q-Error (T=5%)	4.17	395.8K
Cost Q-Error (T=10%)	2.98	328.5K
Cost Q-Error (T=15%)	2.32	414.7K
Cost Q-Error (T=20%)	1.65	326.9K

D Query Plan Analysis

This appendix describes the full execution of query 5a and 16a by the Cardinality Q-Error scheme with parameter ($N = 32$). In figures, the true cardinality of a node in the plan is denoted with T, the predicted cardinality with P, and the cardinality of base tables with C.

Query 5a The query plan after iterations 1–3 is shown in Figure 22 – Figure 24, followed by the queries that were used to materialize the intermediate results.

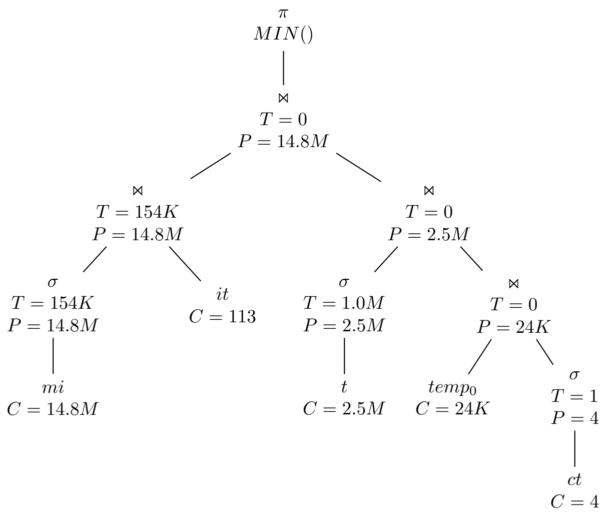


Figure 22: Plan for JOB query 5a after one iteration.

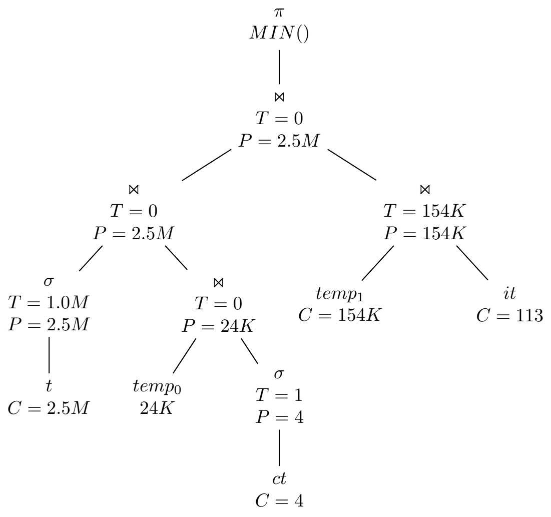


Figure 23: Plan for JOB query 5a after two iterations.

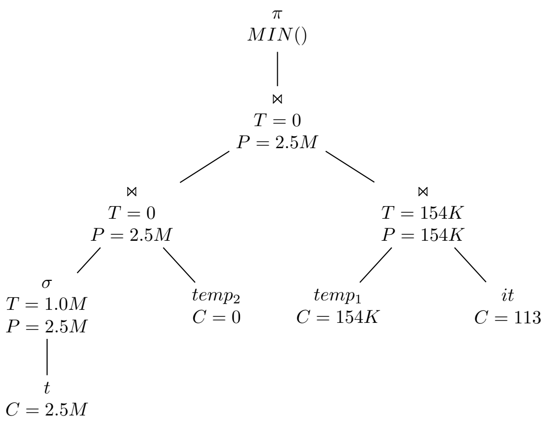


Figure 24: Plan for JOB query 5a after three iterations.

In the generated temporary table queries, the table index is appended to column names to avoid reoccurencen of column names, as various tables have overlapping column names.

```
CREATE TEMPORARY TABLE main._reopt_temp_8096346510212993050_0 AS (
  SELECT t8.movie_id AS movie_id8,
         t8.company_type_id AS company_type_id8
  FROM main.movie_companies AS t8
  WHERE t8.note LIKE '%(theatrical)%'
    AND t8.note LIKE '%(France)%'
);

CREATE TEMPORARY TABLE main._reopt_temp_8096346510212993050_1 AS (
  SELECT t9.movie_id AS movie_id9,
         t9.info_type_id AS info_type_id9
  FROM main.movie_info AS t9
  WHERE t9.info IN ('Sweden',
                    'Norway',
                    'Germany',
                    'Denmark',
                    'Swedish',
                    'Denish',
                    'Norwegian',
                    'German')
);

CREATE TEMPORARY TABLE main._reopt_temp_8096346510212993050_2 AS (
  SELECT t8.movie_id8 AS movie_id88
  FROM main._reopt_temp_8096346510212993050_0 AS t8,
       main.company_type AS t6
  WHERE t8.company_type_id8 = t6.id
    AND t6.kind = 'production companies'
);
```

Query 16a The query plan after iterations 1–6 is shown in Figure 25 – Figure 30, followed by the queries that were used to materialize the intermediate results.

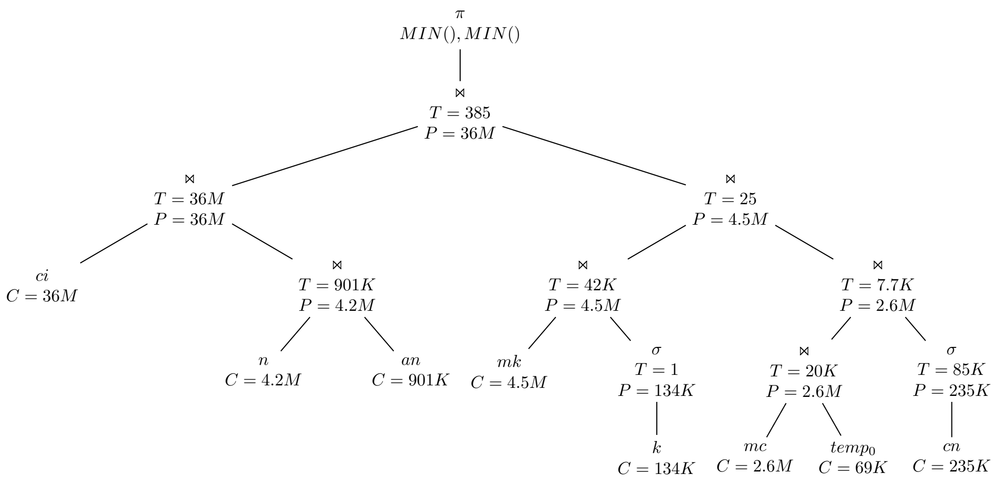

π
$MIN(), MIN()$
$\bowtie$ $T = 385$ $P = 36M$
$\bowtie$ $T = 36M$ $P = 36M$
$\bowtie$ $T = 25$ $P = 4.5M$
ci $C = 36M$
$\bowtie$ $T = 901K$ $P = 4.2M$
$\bowtie$ $T = 42K$ $P = 4.5M$
$\bowtie$ $T = 7.7K$ $P = 2.6M$
n $C = 4.2M$
an $C = 901K$
mk $C = 4.5M$
σ $T = 1$ $P = 134K$
$\bowtie$ $T = 20K$ $P = 2.6M$
σ $T = 85K$ $P = 235K$
k $C = 134K$
mc $C = 2.6M$
$temp_0$ $C = 69K$
cn $C = 235K$

Figure 25: Plan for JOB query 16a after one iteration.

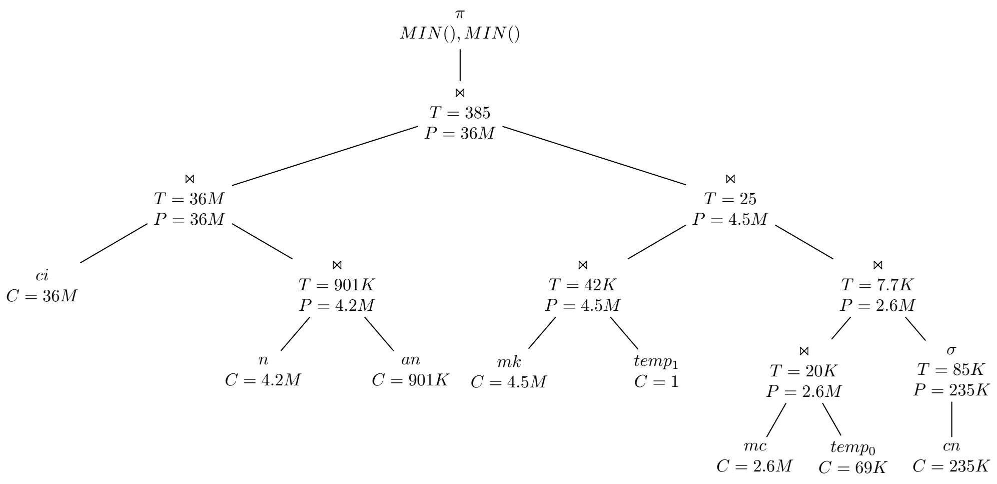

π
$MIN(), MIN()$
$\bowtie$ $T = 385$ $P = 36M$
$\bowtie$ $T = 36M$ $P = 36M$
$\bowtie$ $T = 25$ $P = 4.5M$
ci $C = 36M$
$\bowtie$ $T = 901K$ $P = 4.2M$
$\bowtie$ $T = 42K$ $P = 4.5M$
$\bowtie$ $T = 7.7K$ $P = 2.6M$
n $C = 4.2M$
an $C = 901K$
mk $C = 4.5M$
$temp_1$ $C = 1$
$\bowtie$ $T = 20K$ $P = 2.6M$
σ $T = 85K$ $P = 235K$
mc $C = 2.6M$
$temp_0$ $C = 69K$
cn $C = 235K$

Figure 26: Plan for JOB query 16a after two iterations.

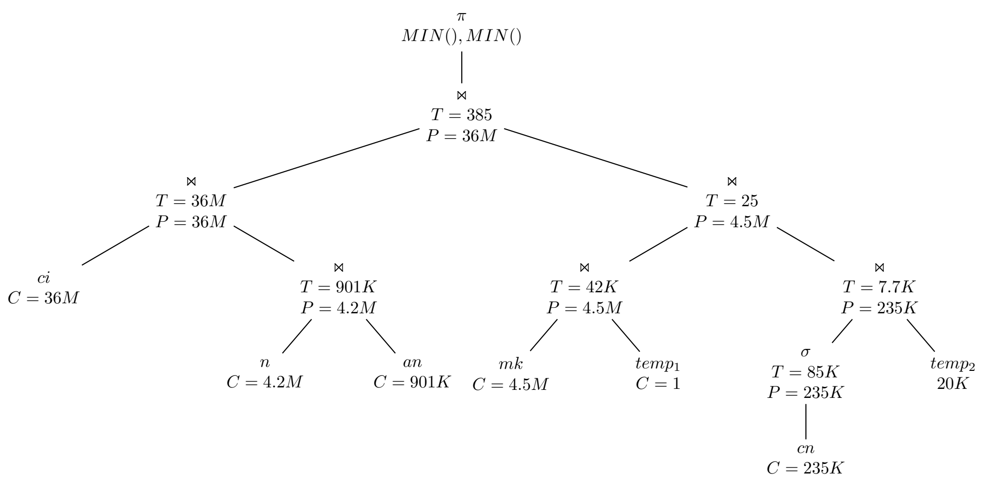

π
$MIN(), MIN()$
$\bowtie$ $T = 385$ $P = 36M$
$\bowtie$ $T = 36M$ $P = 36M$
ci $C = 36M$
$\bowtie$ $T = 901K$ $P = 4.2M$
n $C = 4.2M$
an $C = 901K$
$\bowtie$ $T = 25$ $P = 4.5M$
$\bowtie$ $T = 42K$ $P = 4.5M$
mk $C = 4.5M$
$temp_1$ $C = 1$
$\bowtie$ $T = 7.7K$ $P = 235K$
σ $T = 85K$ $P = 235K$
cn $C = 235K$
$temp_2$ $20K$

Figure 27: Plan for JOB query 16a after three iterations.

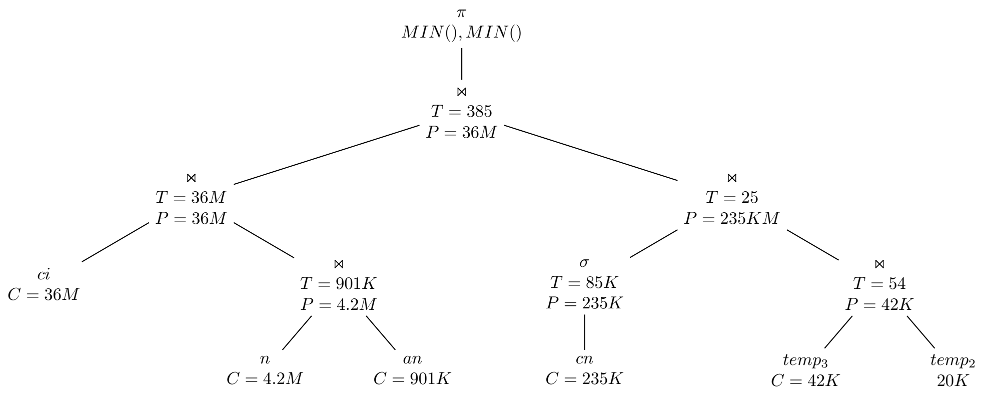

π
$MIN(), MIN()$
$\bowtie$ $T = 385$ $P = 36M$
$\bowtie$ $T = 36M$ $P = 36M$
ci $C = 36M$
$\bowtie$ $T = 901K$ $P = 4.2M$
n $C = 4.2M$
an $C = 901K$
$\bowtie$ $T = 25$ $P = 235KM$
σ $T = 85K$ $P = 235K$
cn $C = 235K$
$\bowtie$ $T = 54$ $P = 42K$
$temp_3$ $C = 42K$
$temp_2$ $20K$

Figure 28: Plan for JOB query 16a after four iterations.

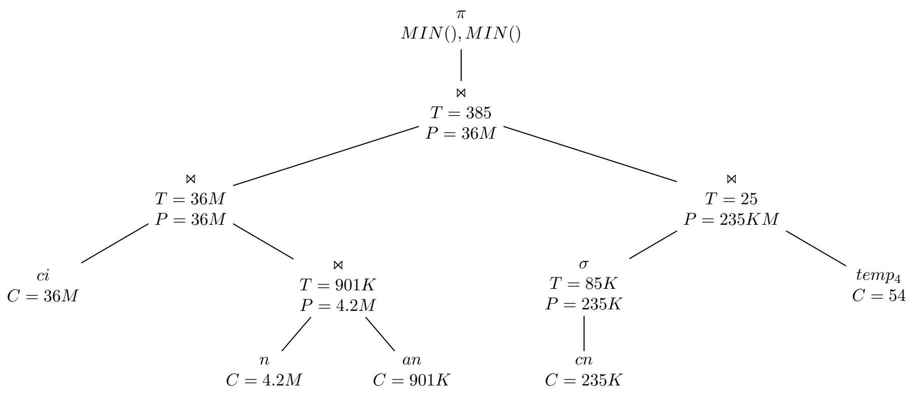

π
$MIN(), MIN()$
$\bowtie$ $T = 385$ $P = 36M$
$\bowtie$ $T = 36M$ $P = 36M$
ci $C = 36M$
$\bowtie$ $T = 901K$ $P = 4.2M$
n $C = 4.2M$
an $C = 901K$
$\bowtie$ $T = 25$ $P = 235KM$
σ $T = 85K$ $P = 235K$
cn $C = 235K$
$temp_4$ $C = 54$

Figure 29: Plan for JOB query 16a after five iterations.

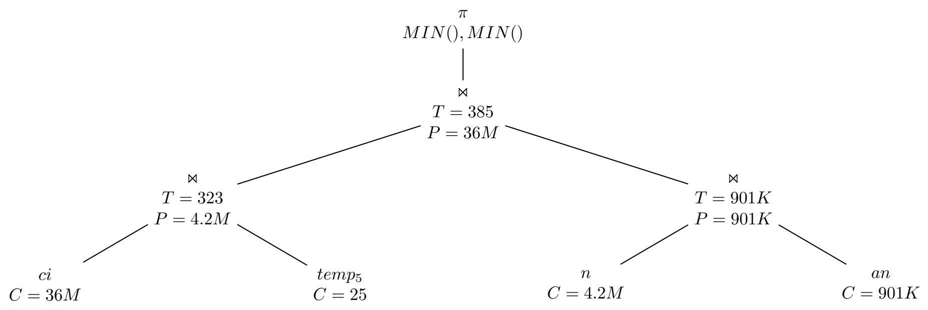


Figure 30: Plan for JOB query 16a after six iterations.

```
CREATE TEMPORARY TABLE main._reopt_temp_13213788151736028524_0 AS (
  SELECT t13.id AS id13,
         t13.title AS title13
  FROM main.title AS t13
  WHERE t13.episode_nr BETWEEN 50 AND 100
);

CREATE TEMPORARY TABLE main._reopt_temp_13213788151736028524_1 AS (
  SELECT t9.id AS id9
  FROM main.keyword AS t9
  WHERE t9.keyword = 'character-name-in-title'
);

CREATE TEMPORARY TABLE main._reopt_temp_13213788151736028524_2 AS (
  SELECT t10.movie_id AS movie_id10,
         t10.company_id AS company_id10,
         t13.title13 AS title1313
  FROM main.movie_companies AS t10,
       main._reopt_temp_13213788151736028524_0 AS t13
  WHERE t10.movie_id = t13.id13
);

CREATE TEMPORARY TABLE main._reopt_temp_13213788151736028524_3 AS (
  SELECT t11.movie_id AS movie_id11
  FROM main.movie_keyword AS t11,
       main._reopt_temp_13213788151736028524_1 AS t9
  WHERE t11.keyword_id = t9.id9
);

CREATE TEMPORARY TABLE main._reopt_temp_13213788151736028524_4 AS (
  SELECT t11.movie_id11 AS movie_id1111,
         t13.company_id10 AS company_id1013,
         t13.title1313 AS title131313
  FROM main._reopt_temp_13213788151736028524_3 AS t11,
       main._reopt_temp_13213788151736028524_2 AS t13
  WHERE t11.movie_id11 = t13.movie_id10
);

CREATE TEMPORARY TABLE main._reopt_temp_13213788151736028524_5 AS (
  SELECT t13.movie_id1111 AS movie_id111113,
         t13.title131313 AS title13131313
  FROM main.company_name AS t8,
       main._reopt_temp_13213788151736028524_4 AS t13
  WHERE t8.id = t13.company_id1013
    AND t8.country_code = '[us]'
);
```