University of Minho
Computer Science

Development Report

2024/2025

SQL Optimizer Analysis

Eduardo Pereira (A70619)
Tomás Meireles (A100106)
Tiago Fernandes (A98983)

May 27, 2025

Abstract

This report outlines the development of a project carried out as part of the **Project** curricular unit in the third year of the **Computer Science** degree at the **University of Minho**.

The main objective of the project is to perform an in-depth analysis of a SQL query optimizer, through the graphical representation of the various phases involved in the optimization process of each query.

To accomplish these objectives, and following the recommendation of our academic advisor, Professor **José Orlando Pereira**, we made use of the educational database system **Rising-light**, which includes a functional query optimizer available through its **GitHub repository**.

The development process was structured into five main phases:

1. Familiarization with the available tools and initial preparations: Cloning of the repository to our GitHub repository to begin testing possibilities.
2. Analysis of the problem and definition of the project's goals.
3. Insertion of strategically placed print statements into the `optimizer.rs` file to trace the execution flow.
4. Logging and saving of relevant execution data to external csv files.
5. Generation of visual representations and bar charts based on the collected data.

This structured approach provided a deeper understanding of the inner workings of the optimizer and enabled a comprehensive analysis of its behavior across different stages.

Contents

List of Figures

1. Introduction

1.1. Objectives

The main objectives of this project are:

- To understand the inner workings of a SQL query optimizer through visual representation
- To analyze the different phases of the optimization process
- To identify patterns and bottlenecks in query optimization
- To visualize how different SQL queries are processed and optimized
- To provide insights that could lead to potential improvements in query optimization techniques

Our approach focused on instrumenting the Risinglight database system to collect detailed information about the optimization process, allowing us to generate visual representations that highlight key aspects of the optimizer's behavior.

1.2. Report structure

To ease the reading and comprehension of this document, in this section will be explained the structure and content of each chapter.

- **Chapter 1 – Introduction:** Contextualization of the project and its main objectives.
- **Chapter 2 – Description and Specification:** Informal description of the problem, project goals, and the data collected for analysis.
- **Chapter 3 – Optimization Stages and Rule Application:** Detailed description of the optimization stages, the rules applied in each stage, and the iterative nature of the optimization process.
- **Chapter 4 – Visual Representation and Analysis:** In-depth analysis of the optimizer's behavior, focusing on selected queries and the impact of each optimization stage.
- **Chapter 5 – Discussion of Results:** Discussion of the results obtained from the analysis, highlighting the differences in optimization processes and possible improvements.
- **Chapter 6 – Final Conclusions and Future Work:** Final remarks about the project and suggestions for future improvements.

2. Description and Specification

2.1. Informal Problem Description

A SQL query optimizer is a critical component of any database management system. It transforms user-written queries into efficient execution plans, aiming to minimize resource usage and execution time. The optimization process involves several complex stages, including:

1. Parsing the SQL query into a logical representation
2. Applying transformation rules to generate alternative execution plans
3. Estimating the cost of each plan
4. Selecting the plan with the lowest estimated cost

Understanding this process is challenging due to its complexity and the abstract nature of the operations performed.
Our project aims to make this process more tangible by analysing each stage and helping pinpoint the exact area which causes more inefficiency during query optimization.

To achieve this, we've instrumented the Risinglight query optimizer to capture detailed information about:

- Cost decrease for each query
- Number of nodes in the relational expression
- Expression groups generated during optimization
- Most used rules in a certain stage
- Insight into a specific optimizing technique: merge

This data is then processed and displayed in bar charts to provide further insight into the optimizer.

2.2. Database Schema

The Risinglight database system is designed to support the TPC-H benchmark, which is a widely used standard for evaluating the performance of database systems. The TPC-H schema consists of the following tables:

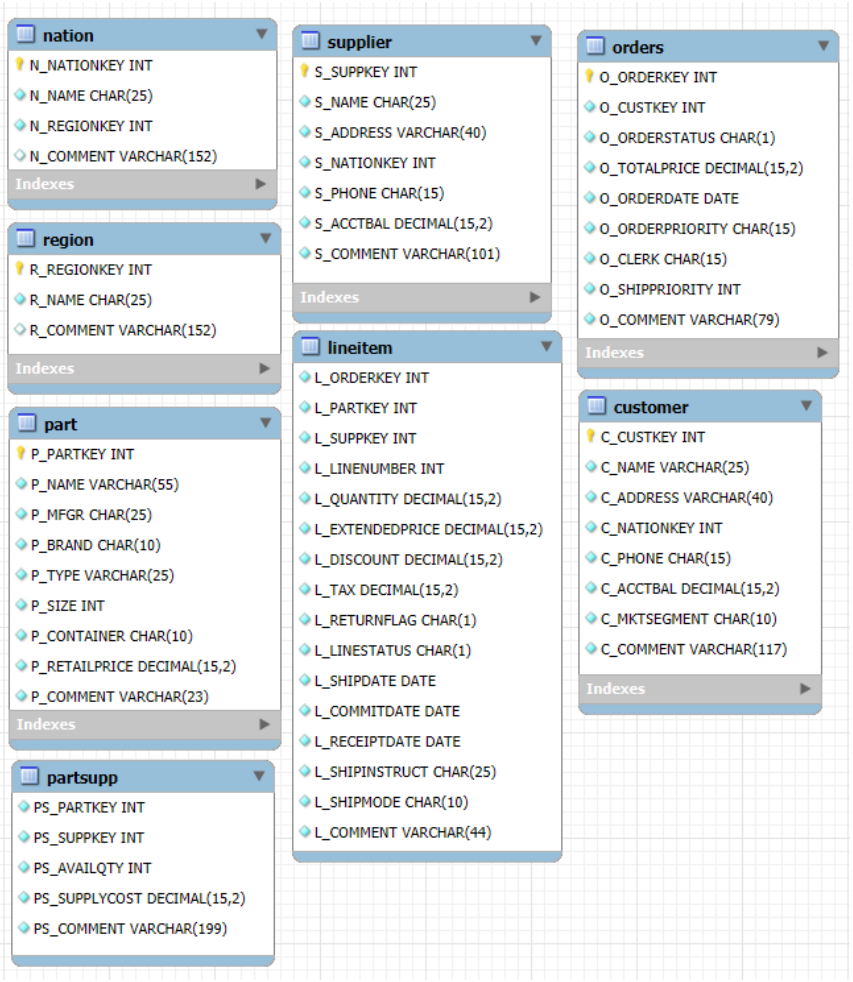

Figure 2.1.: Database Schema

3. Optimization Stages and Rule Application

The optimizer starts with the initial logical expression, which is a representation of the SQL query. This expression is then transformed through a series of stages, each applying a set of rules to optimize the query.
There are three main stages, each with its own set of rules and an unknown number of iterations, in order to target specific inefficiencies and improve the overall performance of the query execution plan.

Iterative Application: Each group of rules is applied iteratively within its stage. The optimizer may revisit the same rule multiple times as the query plan evolves, ensuring that all possible optimizations are explored before moving to the next stage.

Throughout this report, we will refer to the initial state of the expression as **stage 0**, since it is the state before any optimization has been applied. Below is a summary of the three stages and their respective rules:

3.1. Stage 1: Pushdown Apply and Join Transformation

In the first stage, the optimizer focuses on simplifying logical expressions and transforming subqueries into joins. The main goals are to:

- **Simplify Boolean Expressions:** Rules such as `and_rules` are used to simplify and normalize logical AND expressions, making the query plan easier to optimize in later stages.
- **Apply Always—Better Transformations:** The `always_better_rules` group contains general-purpose rules that are always beneficial, such as removing redundant operations or simplifying expressions.
- **Subquery to Join Conversion:** The `subquery_rules` transform certain subqueries into equivalent join operations, enabling further optimizations and improving execution efficiency.

3.2. Stage 2: Predicate and Projection Pushdown, Index Scan

The second stage aims to reduce the amount of data processed by pushing filters and projections as close as possible to the data sources. The rules applied include:

- **Expression Simplification:** General expression simplification rules (`expr::rules`) are applied to further normalize and reduce expressions.
- **Predicate Pushdown:** The `predicate_pushdown_rules` move filter conditions (WHERE clauses) down the query tree, so that irrelevant rows are filtered out early.
- **Projection Pushdown:** The `projection_pushdown_rules` ensure that only necessary columns are carried through each stage, minimizing data movement.
- **Index Scan Optimization:** The `index_scan_rules` attempt to replace full table scans with index scans when possible, improving query performance.
- **Always-Better Transformations:** These are re-applied to catch any new opportunities created by the previous rules.

3.3. Stage 3: Join Reordering and Hash Join

The third stage focuses on optimizing join operations, which are often the most expensive part of query execution. The rules in this stage include:

- **Join Reordering:** The `join_reorder_rules` change the order in which joins are performed, aiming to minimize intermediate result sizes and overall cost.
- **Hash Join Optimization:** The `hash_join_rules` convert suitable joins into hash joins, which are more efficient for large datasets.
- **Predicate and Projection Pushdown:** These rules are re-applied to ensure that any new opportunities for pushdown created by join reordering are exploited.
- **Order Optimization:** The `order_rules` optimize ORDER BY operations, potentially reducing sorting costs.
- **Boolean Simplification and Always-Better Transformations:** These are applied again to further refine the plan.

4. Visual Representation and Analysis

The goal of the optimizer is mainly to reduce the cost of the execution of certain queries, this is accomplished through a series of optimization stages that eventually lead to a more efficient database.
Since our goal is to analyse this optimizer, we thought it would be helpful to compare two different queries. Since we are dealing with cost-efficient optimization, we decided we should choose the ones with the most different variation in cost, which means we have to look for one query with little variation and one with a significant cost decrease.

We thought it would be helpful if we could see the cost for every query and compare it, before choosing the ones to further analyse. With this in mind and making use of the **Matplotlib library** in **Python**, we built a bar chart, in which is displayed the cost for every stage the optimizer goes through.

The initial cost is omitted, as it remains identical across all queries. This value is defined as the maximum possible value that can be represented by a 32-bit floating point number and it is assigned using the line `[let mut cost = f32::MAX]`. This number has a value of approximately 3.4×10^{38}.

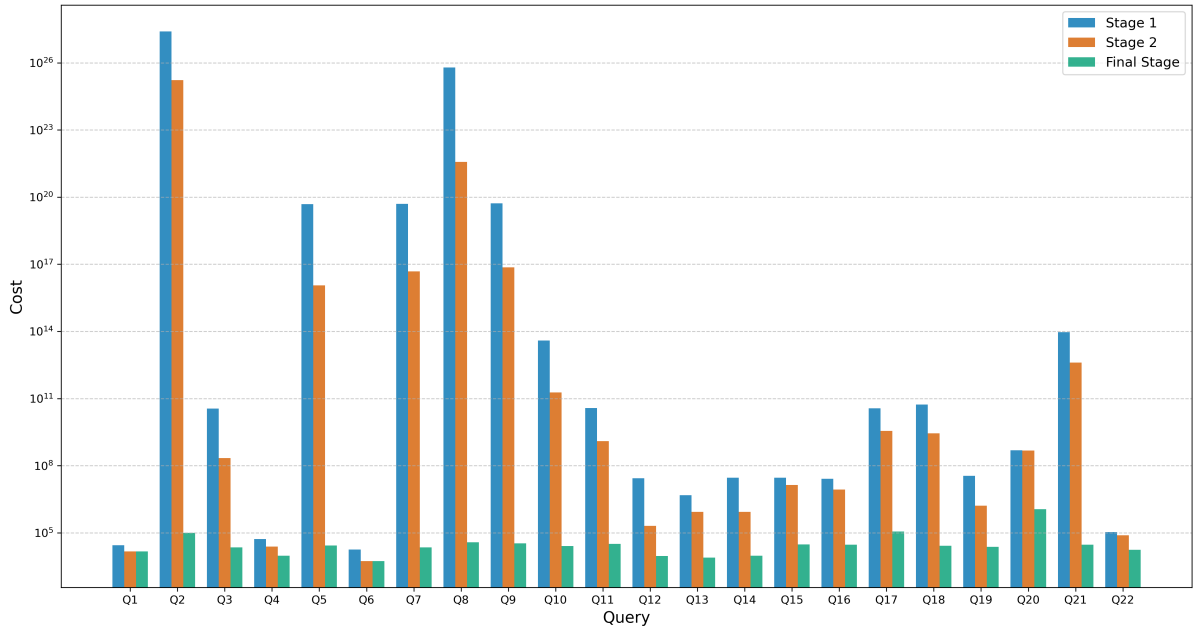

Figure 4.1.: Cost variation bar chart

The chart above does a good job at providing some insight as to what queries would be more advantageous to analyse with more depth.

To help understand the motivations behind this analysis, the table below has more detailed information about every query: initial and decreased costs alongside the stages.

Query	Stage 1	Stage 2	Final Stage	Absolute Reduction
Q1	28128.111	14618.112	14618.112	13509.999
Q2	2502958000000000091792343040.000	16926520000000000639631360.000	99146.390	2502958000000000091792343040.000
Q3	35436200000.000	215360400.000	22157.160	35436177842.840
Q4	52110.470	24255.469	9412.734	42697.736
Q5	49138130000000000000.000	11308230000000000.000	27351.363	49138129999999975424.000
Q6	17958.207	5469.458	5469.458	12488.749
Q7	50314906000000000000.000	48091680000000000.000	22097.984	50314905999999975424.000
Q8	62104804999999996569845760.000	3766529499999999885312.000	37279.418	62104804999999996569845760.000
Q9	52074470000000000000.000	72212870000000000.000	33519.906	52074469999999967232.000
Q10	39213236000000.000	189269080000.000	25337.281	39213235974662.719
Q11	37588554000.000	1263876700.000	31861.496	37588522138.504
Q12	27565862.000	201470.450	9076.412	27556785.588
Q13	4840745.500	867070.300	7853.546	4832891.954
Q14	29172502.000	850032.300	9406.975	29163095.025
Q15	29049016.000	13515014.000	29854.697	29019161.303
Q16	26056468.000	8582357.000	29108.352	26027359.648
Q17	36120883000.000	3601246000.000	112654.830	36120770345.170
Q18	54558280000.000	2775188700.000	26530.621	54558253469.379
Q19	34976370.000	1627012.800	23419.270	34952950.730
Q20	491925120.000	475881760.000	1130161.900	490794958.100
Q21	91870570000000.000	4082035900000.000	28986.838	91870569971013.156
Q22	107983.266	77013.290	17342.803	90640.463

Figure 4.2.: Cost variation table

Based on the images above, we come to the conclusion that some queries have a huge decrease in cost, while others barely evolve in efficiency.
The query with the most decrease in cost throughout the stages is query 2, with a decrease of 2.502958e+27, and on query 6 the decrease is barelly noticeable, having only a difference of 12488.749499999998 from stage 1 to stage 3.

During the development of this project, we had the chance to analyse all the TPC-H queries of the **Risinglight** database system, more specifically, the expression groups and the most used rules in each stage. We started detecting a pattern in a certain type of query: in queries with a similar cost variation, the number of expressions groups and the most used rules were also similar. As we will see further into this report, in queries with higher cost decrease, one of the most used rules in the last stage would always be related to **join operations**, for example join-reordering or join-rotating.
With this in mind, we thought analysing only **query 2** and **query 6** would be enough to understand the optimizer's behavior, since they are the most extreme cases in terms of cost variation and will reflect the general behaviour among the rest of the queries. These are the two queries we will focus on in this analysis, to understand the huge difference behind these two.

While this analysis is only focused on the 2 queries mentioned, the remaining queries' analysis will be available on our GitHub repository - **Charts folder** for the reader to explore.

To understand the steps forward in this analysis, it's useful to see the SQL query that we are going to analyse, as well as the relational expression generated.

Since these are quite complex, especially the expression generated, we will display them in the appendix, in the end of this report.

4.1. Query 6

4.1.1. Relational Expression

The relational expression is a representation of the SQL query in a tree-like structure, where each node represents an operation or a relation. The expression is generated by the optimizer and serves as an intermediate representation before the final execution plan is created. In the appendix, in section Query 6 (Appendix), we can see the SQL query, and the relational expression generated.

Judging by the relational expression alone, we can see we are dealing with a simple query, which already gives the hint that the optimizer won't have a lot of work to do with this one.

Nevertheless, the optimizer still manages to reduce the number of nodes from 51 in the initial expression to 31 in the final optimized form, while maintaining the semantic equivalence of the query.

When comparing the initial and final expressions, we find some reasons why the number of nodes is lower: the number of columns is brought down to only the necessary ones, in this case, four; the operations were also simplified to keep only the essential ones such as **filter**.

4.1.2. Expression Groups

The expression groups are a representation of the different equivalent expressions generated during the optimization process. Each group contains expressions that are semantically equivalent but may differ in their internal representation.

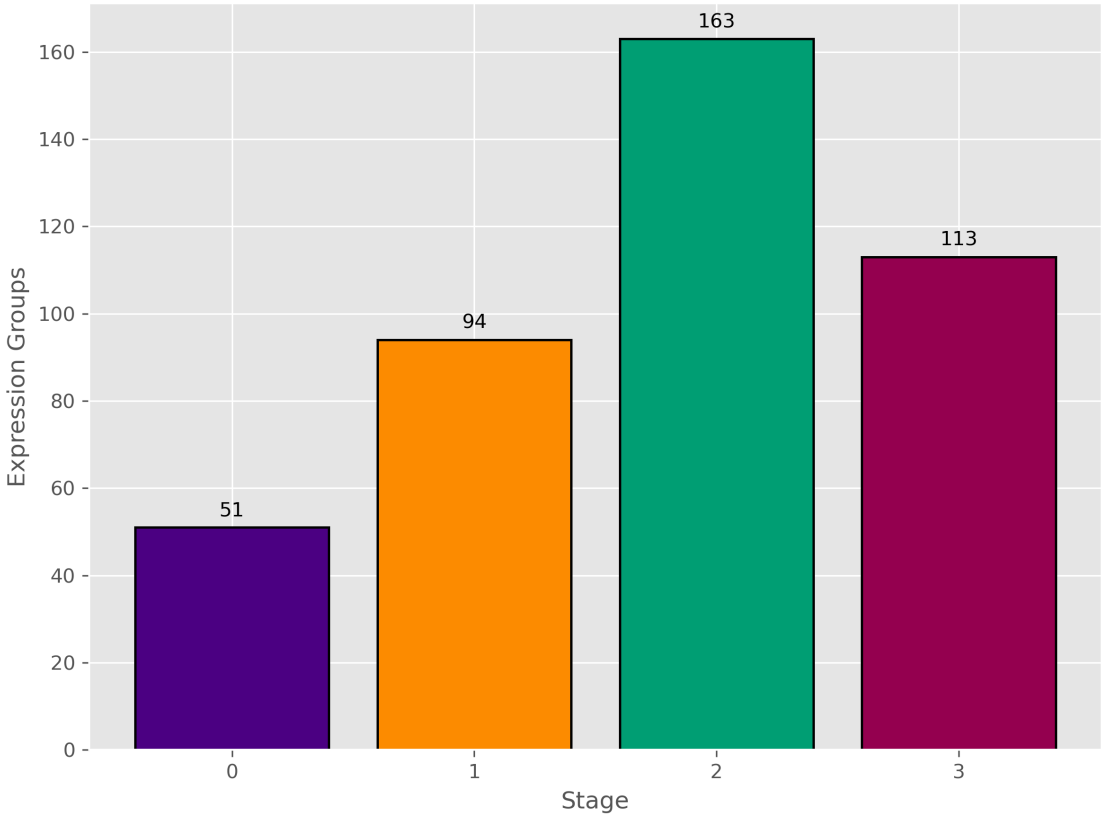

Figure 4.3.: Query 6 - Expression Groups

The chart above displays the number of expression groups generated at each stage of the optimization process. Within each group, there may be an unknown number of equivalent expressions.

This bar chart shows that the number of expression groups generated in stage 2 is significantly higher than in stage 1 and stage 3. This suggests that the optimizer is exploring a larger number of equivalent expressions in stage 2, likely due to the application of more complex rules and transformations.
This is also reflected in the cost variation chart (Figure 4.1), where we can see that the decrease in cost is more accentuated when we move from stage 1 to stage 2. Stage 2 is where the optimizer applies the predicate pushdown and projection pushdown rules.

4.1.3. Rule Application

Given these new results, we can't help but wonder what exactly is happening during this stage that might trigger this increase in the number of expression groups.
Looking at the code in the `optimizer.rs` file, it's possible to learn exactly what rules are being applied during this stage.

The rules are defined in the `src/planner/rules` folder, and they are grouped into different categories. The rules that are applied during stage 2 are defined in the `src/planner/rules/plan.rs` file.

```
static STAGE2_RULES: LazyLock<Vec<Rewrite>> = LazyLock::new(|| {
    let mut rules = vec![];
    rules.append(&mut rules::expr::and_rules());
    rules.append(&mut rules::plan::always_better_rules());
    rules.append(&mut rules::plan::predicate_pushdown_rules());
    rules.append(&mut rules::plan::projection_pushdown_rules());
    rules.append(&mut rules::plan::index_scan_rules());
    rules
});
```

These rules are critical for reducing data processing overhead. Predicate pushdown moves filter conditions (WHERE clauses) down the query tree closer to the data sources, allowing rows to be filtered out as early as possible before they participate in costly operations like joins. Projection pushdown ensures that only necessary columns are carried through each operation in the query plan, minimizing data movement between operations.

When joined with index scan rules, which attempt to utilize available indexes instead of performing full table scans, these transformations significantly reduce the amount of data that needs to be processed in subsequent operations, leading to substantial performance improvements.

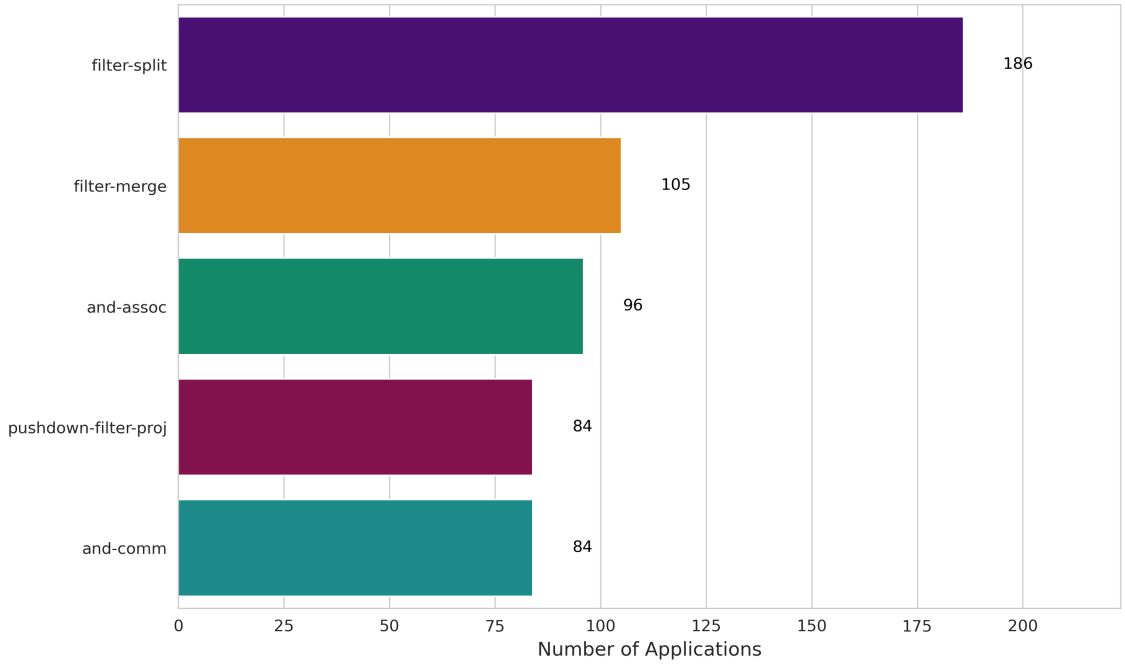

Figure 4.4.: Query 6 - Rules in Stage 2

The **filter-split** rules are optimization rules applied during query processing to simplify and improve the efficiency of query execution. These rules work by breaking down complex filter conditions (e.g., multiple conditions combined with AND) into separate, simpler filters.

Example:

```
SELECT *
FROM orders
WHERE order_date > '2023-01-01' AND total_amount < 100;
```

Initially, the optimizer represents the filter condition as a single node in the query plan:

```
(filter (and (> order_date '2023-01-01') (< total_amount 100)) orders)
```

After applying the filter-split rule, the condition is split into two separate filters:

```
(filter (> order_date '2023-01-01')
    (filter (< total_amount 100) orders))
```

This transformation has several benefits:

- **Predicate Pushdown:** Each filter can now be pushed closer to the data source, reducing the amount of data processed in subsequent stages.
- **Parallel Evatuation:** Independent filters can be evaluated in parallel, improving query performance.
- **Index Utilization:** Splitting filters allows the optimizer to use indexes more effectively, as each condition can be matched to a specific index.

The second most applied rule is the **filter-merge** rule. While this may seem counterintuitive, since it's the opposite of the filter-split rule, it is important to understand that the optimizer applies both rules in a way that they can be used interchangeably.
As we know, the optimizer visits these rules an indefinite amount of times, even within the same stage, which means, while the first rule splits the conditions on one iteration, the second rule may merge them back on the next iteration, if the optimizer sees fit.

This aligns with our understanding that stage 2 focuses heavily on optimizing the filter conditions and their order of execution.

4.1.4. Merge Operations

The merge operation is a critical step in the optimization process, as it allows the optimizer to combine multiple equivalent expressions into a single representation.

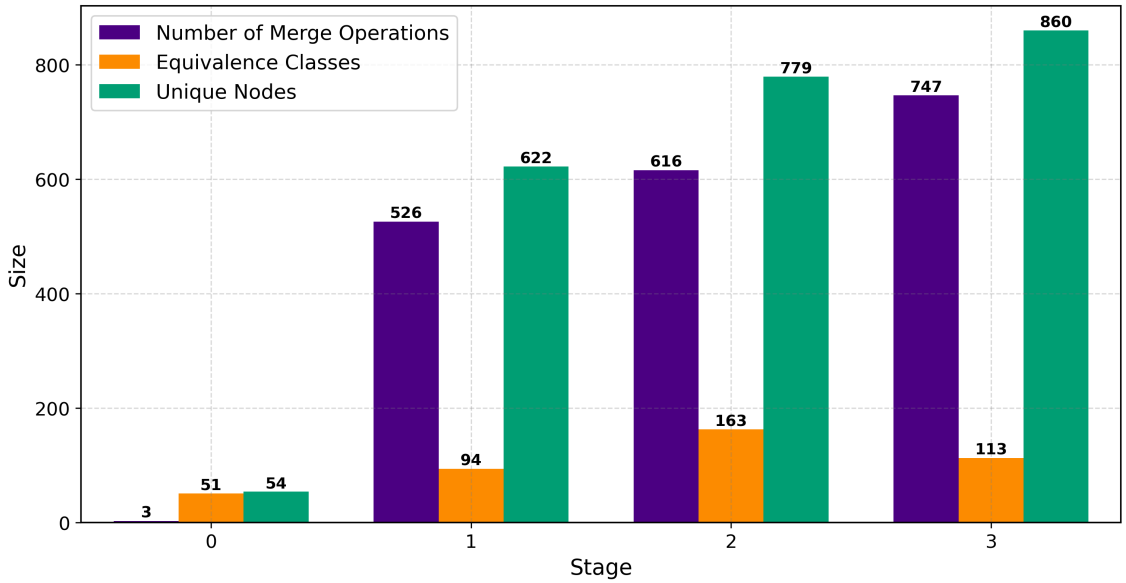

Figure 4.5.: Query 6 - Merge Operations

So far, regarding query 6, we were led to believe that stage 2 is where the optimizer does most of the work. However, when we look at the figure above, not only are they more frequent in stage 3, but we can also see the growing difference between the amount of unique nodes and equivalence classes.

Each equivalence class represents a group in which all the expressions are equivalent, while each node is an unique expression, for example, the expression `"a+b"` is a possible node.
However, `"b+a"` can also be a node, but since they represent the same operation, they can be merged into a single equivalence class.
This is the reason for the growing difference between the number of equivalence classes and the number of unique nodes.

4.2. Query 2

So now, we believe it will be interesting to see how the optimizer behaves with a more complex query, and how it manages to reduce the number of nodes in the expression.

4.2.1. Relational Expression

The stage 0 expression for query 2 proves that this query is far more complex than the previous query we analysed.

The expression is composed of 129 nodes, which is a significant increase compared to the 51 nodes in query 6. This complexity is expected, given the nature of the query, which involves multiple tables and subqueries.

One more reason to analyse the stage 0 expression is to understand how the optimizer starts with a very complex expression, and how it is gradually simplified throughout the stages.

If we compare the initial and final expression, it's possible to see some differences in which operations are present in the nodes. For example, the initial expression has **join** operations, while the final expression includes **hash-join** operations and specific join types, such as **inner-join** or **left-join**.

In a logical plan of a database, a **join** only indicates that two tables are being combined, but it does not specify how this combination is done.
In contrast, a **hash join** is a specific implementation of a join operation that uses a hash table to efficiently combine rows from two tables based on a common key. This is a more efficient way to perform joins, especially for large datasets, and it is one of the optimizations applied by the optimizer in this query.

Another important observation is the change from the general **join** operator to more specific join types like **inner-join**. This transformation reflects a crucial optimization step where the optimizer determines the most efficient type of join for each operation based on the data characteristics and query requirements.
For example, if only matching rows are needed from both tables, an inner-join is more efficient than a simple join. The choice of join type directly impacts the query's execution plan and overall performance.

As we can see in the Query 2 (Appendix), the optimizer managed to reduce the number of nodes from 129 in the initial expression to 109 in the final optimized form, while maintaining the semantic equivalence of the query.
This reduction in complexity is one of the factors that contributes to the overall cost decrease.

So now, a good question to ask is: **How does the optimizer manage to reduce the number of nodes?**

To help answering this question, it is helpful to analyse the expression groups generated during the optimization process.

4.2.2. Expression Groups

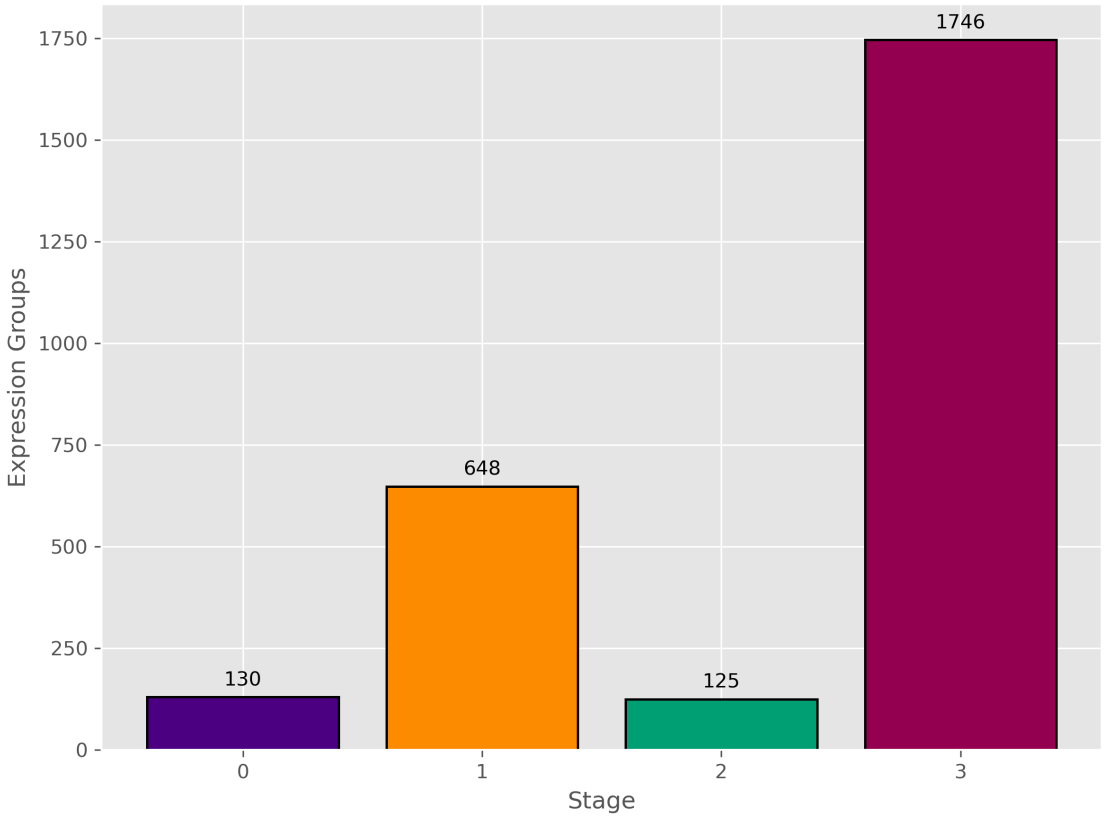

Figure 4.6.: Query 2 - Expression Groups

According to the **Cost Variation** chart (Figure 4.1), the decrease in cost is more accentuated when we move from stage 2 to stage 3. This is the stage where the optimizer applies the join reordering and hash join rules.
This is also reflected in the bar chart covering the expression groups (Figure 4.6), where we can see that the number of groups generated in stage 3 is significantly higher than in stage 2. This implies that the optimizer is, not only exploring a larger number of equivalent expressions in stage 3, but also being able to reduce the overall cost of the query.

This is a very interesting finding, as it suggests that the optimizer is capable of finding more efficient ways to represent the same operation, even when the number of equivalent expressions increases.

4.2.3. Rule Application

Given these new results, we can't help but wonder what exactly is happening during this stage that might trigger this increase in the number of expression groups.

Looking at the code in the `optimizer.rs` file, it's possible to learn exactly what rules are being applied during this stage.

The rules are defined in the `src/planner/rules` folder, and they are grouped into different categories. The rules that are applied during stage 3 are defined in the `src/planner/rules/plan.rs` file.

```
static STAGE3_RULES: LazyLock<Vec<Rewrite>> = LazyLock::new(|| {
    let mut rules = vec![];
    rules.append(&mut rules::expr::and_rules());
    rules.append(&mut rules::plan::always_better_rules());
    rules.append(&mut rules::plan::join_reorder_rules());
    rules.append(&mut rules::plan::hash_join_rules());
    rules.append(&mut rules::plan::predicate_pushdown_rules());
    rules.append(&mut rules::plan::projection_pushdown_rules());
    rules.append(&mut rules::order::order_rules());
    rules
});
```

The rules applied during stage 3 are similar to those in stage 2, but with a focus on join operations and hash joins. The optimizer applies the join reordering and hash join rules to optimize the join operations, which are often the most expensive part of query execution.
The join reordering rules change the order in which joins are performed, aiming to minimize intermediate result sizes and overall cost. The hash join rules convert suitable joins into hash joins, which are more efficient for large datasets.

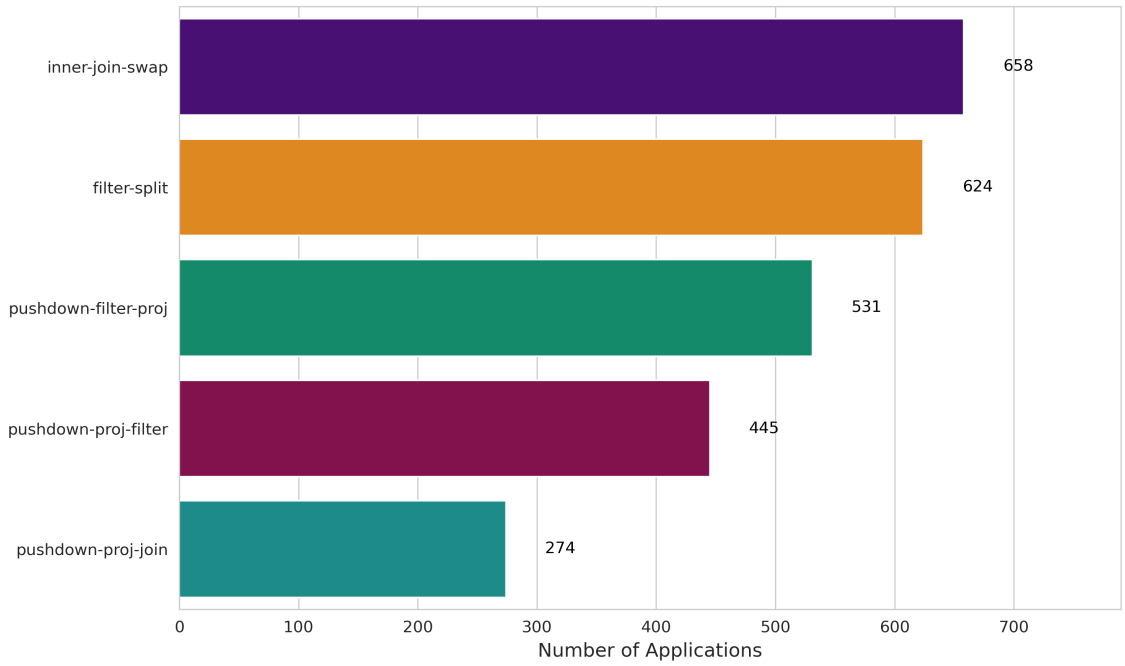

Figure 4.7.: Query 2 - Rules in Stage 3

Looking at the most popular rules applied during stage 3 (Figure 4.7), we can see that the most frequently applied rules are related to join operations and projection pushdown. This aligns with our understanding that stage 3 focuses heavily on optimizing joins and their order of execution.

The high frequency of join-related rule applications explains the increase in expression groups, as each new join order or implementation strategy generates new equivalent expressions. Despite this increase in complexity, the optimizer successfully reduces the query's cost through these transformations.

A rule `inner-join-swap` being the most frequently applied shows how significant join reordering is for query optimization.

Consider the basic associativity transformation: $(A \bowtie B) \bowtie C \Rightarrow A \bowtie (B \bowtie C)$. This seemingly simple change in join order can drastically impact performance. If table $B \bowtie C$ produces a smaller intermediate result than $A \bowtie B$, the right-side expression requires less memory and computation in subsequent joins.
This transformation alone was applied 658 times during stage 3 optimization of Query 2, suggesting how extensively the optimizer explores the space of possible join orders to find the most efficient execution plan.

Just like in query 6, the **filter-split** rule is also present in the most used rules in this query.

4.2.4. Merge Operations

During stage 3, the optimizer applies join reordering rules and other transformations that generate a large number of new equivalent expressions. Each new join order or execution strategy can result in different representations for the same logical operation. As a consequence, the number of expression groups grows rapidly.
However, many of these new expressions end up being semantically equivalent to each other. The merge mechanism of the optimizer serves precisely to identify and merge these equivalent expressions into a single group, reducing redundancy in the e-graph. Therefore, a significant increase in the number of merge operations is expected at this stage, reflecting the intense exploration of execution plan space and the subsequent consolidation of equivalent expressions.

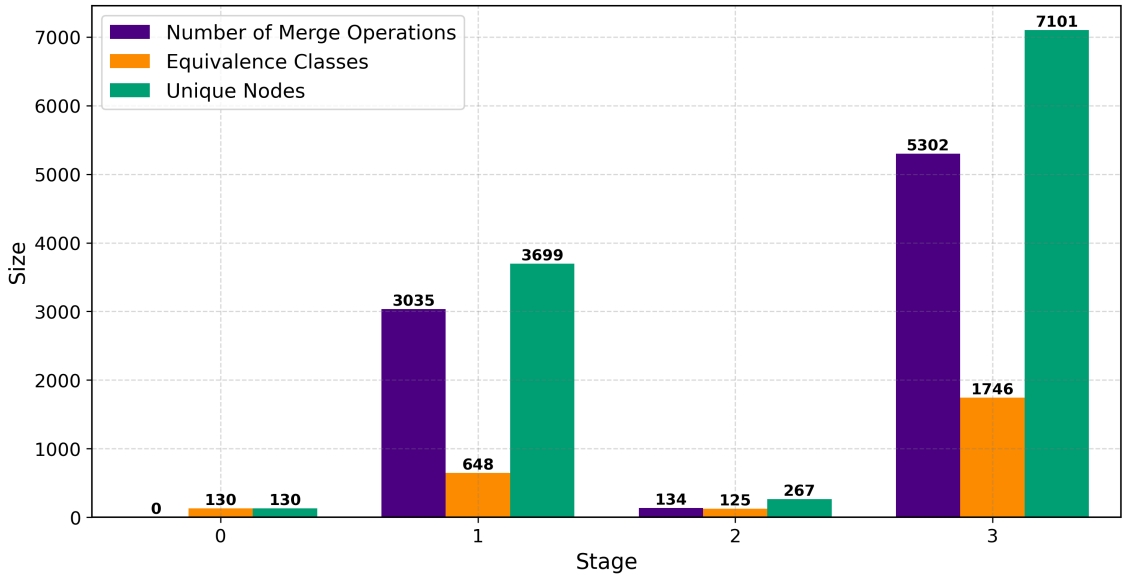

Figure 4.8.: Query 2 - Merge operations

Here we can see that the optimizer has the expected behavior: The number of merge operations greatly increases, which means that, despite the increase in the number of expression groups generated during stage 3, a large group of these are also being merged into a single representation.

Here we can also find some interesting information about the number of equivalence classes and the number of unique nodes. Just like in query 6, we can see that the number of equivalence classes is growing, while the number of unique nodes is decreasing. This is confirmation of what we mentioned before: that, before these operations, there are lot of equivalent expressions with the same semantic meaning.

5. Discussion of Results

In this chapter, we will discuss the results obtained from the analysis of the two queries, focusing on the differences in their optimization processes and the implications of these findings.

Some key findings from our analysis include:

- The relationship between query complexity and the number of expression groups generated
- Common transformation patterns applied by the optimizer for specific query constructs
- The significant impact of join reordering and hash join optimizations on higher cost queries
- The importance of merge operations in consolidating equivalent expressions and reducing redundancy

5.1. Relational Expression

The relational expressions for both queries show a significant difference in complexity. Query 6 has a relatively simple expression with 51 nodes, while query 2 has a much more complex expression with 129 nodes.

Based on the each of the relational expressions, we can conclude query 2 required more effort from the optimizer in order to improve its efficiency. However, there is something both expressions have in common: too much redundant or unnecessary nodes.

For what we saw in the chapters for each query, it seems the initial expression doesn't have a specific implementation for the necessary operations, instead, they have a more general representation, such as **join** or **filter**. Another problem is, for example in query 6, there are more columns being included in the expression than those absolutely needed.

This provides an opportunity for improvement in this optimizer, for example, if we could have a prediction on how the optimizer would behave for a specific type of query, then the initial expression would be greatly improved, and the initial cost would decrease significantly.

5.2. Query complexity and important stages

The complexity of a query is directly related to which stage will be more important for its optimization. In query 6, the optimizer does most of the work in stage 2, while in query 2, the most important stage is stage 3. This means that these two queries have different characteristics and could benefit from different optimization strategies.

5.3. Rule Application

Based on the most used rules during each stage, we can see big differences between the two queries, despite the fact that the same set of rules is being applied to both.

In query 6, despite the rules in stage 3 focusing on join reordering and hash join optimizations, the most applied rules are related to **filter** operations. This is very different from query 2, where the most applied rules in stage 3 are related to **join** operations, as expected.

This is another point where improvements could be made in this system through behaviour prediction. If it's detected that a query is going to be simple, like query 6, probably the last set of rules wouldn't need to be focusing on join operations. In some cases, maybe the optimizer could even skip this last stage altogether, since the cost decrease is not significant enough to justify the extra work. This could possibly improve the optimizer's overall performance.

5.4. Iterative Optimization

This feature also proved to be very important, being able to apply the same set of rules repeatedly is useful to garantee all the possible improvements in efficiency are being made, before moving to the next stage.

6. Final Conclusions and Future Work

This report describes our process of analyzing and visualizing the behavior of a SQL query optimizer through the instrumentation of the Risinglight database system.

When we started this project, we found it very difficult to navigate through this process, and took us some weeks to finally getting some idea of how it worked.
We believe our visual representations have made it possible to observe patterns and behaviors that would be difficult to discern from raw data alone.

While we believe we met the requirements initially set, there is still room for future work.
If we could work within each rule and get information about which nodes are being affected by them directly, that would be a great next step. Also, based on what we stated in the **discussion section**, we can see that there are some improvements that could be made to the optimizer, such as:

- Implementing a prediction mechanism to optimize the initial expression based on query characteristics
- Enhancing the optimizer's ability to deal with redundant stages for simple queries

One possible implementation for these features could rely on machine learning techniques. This approach would mean the optimizer could learn from previous queries and their optimization process, identifying patterns and behaviors that could be used to predict the best course of action for new queries, becoming more resourceful with each execution.

These enhancements could even be applied to other real-world systems using databases, for example, traffic control cameras in crosswalks, since some work with databases to analyse traffic flow in real time. Since this information has to be processed in real time, these improvements could be crucial to ensure the system is able to process the information quickly and efficiently, without unnecessary delays or resource usage.

Overall, this project has provided valuable insights into query optimization, specifically into the Risinglight database and its optimizer. We feel that, with more research and continued development, this project has potential to contribute to the improvement of query optimization in the future and its application in real-world systems.

A. Appendix

A.1. Query 2

```
select
    s_acctbal,
    s_name,
    n_name,
    p_partkey,
    p_mfgr,
    s_address,
    s_phone,
    s_comment
from
    part,
    supplier,
    partsupp,
    nation,
    region
where
    p_partkey = ps_partkey
    and s_suppkey = ps_suppkey
    and p_size = 15
    and p_type like '%BRASS'
    and s_nationkey = n_nationkey
    and n_regionkey = r_regionkey
    and r_name = 'EUROPE'
    and ps_supplycost = (
        select
            min(ps_supplycost)
        from
            partsupp,
            supplier,
            nation,
            region
        where
            p_partkey = ps_partkey
            and s_suppkey = ps_suppkey
            and s_nationkey = n_nationkey
```

```
            and n_regionkey = r_regionkey
            and r_name = 'EUROPE'
    )
order by
    s_acctbal desc,
    n_name,
    s_name,
    p_partkey
limit 100;
```

A.1.1. Relational Expression

Stage 0:

```
    RecExpr { nodes: [Constant(Bool(true)), Column($1.2(1)), Column($1.1(1)),
        Column($1.0(1)), List([3, 2, 1]), Table($1), Scan([5, 4, 0]), Column($0
        .3(1)), Column($0.2(1)), Column($0.1(1)), Column($0.0(1)), List([10, 9,
        8, 7]), Table($0), Scan([12, 11, 0]), Column($3.6(1)), Column($3.5(1)),
        Column($3.4(1)), Column($3.3(1)), Column($3.2(1)), Column($3.1(1)),
        Column($3.0(1)), List([20, 19, 18, 17, 16, 15, 14]), Table($3), Scan([22,
         21, 0]), Column($4.4(1)), Column($4.3(1)), Column($4.2(1)), Column($4
        .1(1)), Column($4.0(1)), List([28, 27, 26, 25, 24]), Table($4), Scan([30,
         29, 0]), Inner, Join([32, 0, 31, 23]), Join([32, 0, 33, 13]), Join([32,
        0, 34, 6]), Constant(String(""EUROPE"")), Eq([2, 36]), Eq([8, 3]), Eq
        ([17, 10]), Eq([20, 27]), Column($2.0), Eq([41, 28]), And([42, 40]), And
        ([43, 39]), And([44, 38]), And([45, 37]), Filter([46, 35]), Min(25), List
        ([48]), Agg([49, 47]), Filter([0, 50]), List([]), Order([52, 51]), Ref
        (48), List([54]), Proj([55, 53]), Constant(Int32(0)), Constant(Null),
        Limit([58, 57, 56]), Column($1.2), Column($1.1), Column($1.0), List([62,
        61, 60]), Scan([5, 63, 0]), Column($0.3), Column($0.2), Column($0.1),
        Column($0.0), List([68, 67, 66, 65]), Scan([12, 69, 0]), Column($4.4),
        Column($4.3), Column($4.2), Column($4.1), Column($4.0), List([75, 74, 73,
         72, 71]), Scan([30, 76, 0]), Column($3.6), Column($3.5), Column($3.4),
        Column($3.3), Column($3.2), Column($3.1), Column($3.0), List([84, 83, 82,
         81, 80, 79, 78]), Scan([22, 85, 0]), Column($2.8), Column($2.7), Column(
        $2.6), Column($2.5), Column($2.4), Column($2.3), Column($2.2), Column($2
        .1), List([41, 94, 93, 92, 91, 90, 89, 88, 87]), Table($2), Scan([96, 95,
         0]), Join([32, 0, 97, 86]), Join([32, 0, 98, 77]), Join([32, 0, 99, 70])
        , Join([32, 0, 100, 64]), LeftOuter, Apply([102, 101, 59]), Eq([72, 54]),
         Eq([61, 36]), Eq([66, 62]), Eq([81, 68]), Constant(String(""\%BRASS"")),
         Like([91, 108]), Constant(Int32(15)), Eq([90, 110]), Eq([84, 74]), Eq
        ([41, 75]), And([113, 112]), And([114, 111]), And([115, 109]), And([116,
        107]), And([117, 106]), And([118, 105]), And([119, 104]), Filter([120,
        103]), Filter([0, 121]), Desc(79), List([123, 67, 83, 41]), Order([124,
        122]), List([79, 83, 67, 41, 93, 82, 80, 78]), Proj([126, 125]), Constant
        (Int32(100)), Limit([128, 57, 127])] }
```

Stage 3:

```
RecExpr { nodes: [Constant(Bool(true)), Column($4.3(1)), Column($4.1(1)), Column(
    $4.0(1)), List([3, 2, 1]), Table($4), Scan([5, 4, 0]), Column($3.3(1)),
    Column($3.0(1)), List([8, 7]), Table($3), Scan([10, 9, 0]), Column($0.2(1)),
    Column($0.0(1)), List([13, 12]), Table($0), Scan([15, 14, 0]), Column($1.1(1)
    ), Column($1.0(1)), List([18, 17]), Table($1), Scan([20, 19, 0]), Constant(
    String(""EUROPE"")), Eq([17, 22]), Filter([23, 21]), List([18]), Proj([25,
    24]), List([12]), Inner, HashJoin([28, 0, 25, 27, 26, 16]), List([13]), Proj
    ([30, 29]), List([7]), HashJoin([28, 0, 30, 32, 31, 11]), List([8]), Proj
    ([34, 33]), List([2]), HashJoin([28, 0, 34, 36, 35, 6]), List([3, 1]), Proj
    ([38, 37]), Column($4.4), Column($4.3), Column($4.2), Column($4.1), Column($4
    .0), List([44, 43, 42, 41, 40]), Scan([5, 45, 0]), Column($3.6), Column($3.5)
    , Column($3.4), Column($3.3), Column($3.2), Column($3.1), Column($3.0), List
    ([53, 52, 51, 50, 49, 48, 47]), Scan([10, 54, 0]), Column($1.2), Column($1.1)
    , Column($1.0), List([58, 57, 56]), Scan([20, 59, 0]), Eq([22, 57]), Filter
    ([61, 60]), Column($0.3), Column($0.2), Column($0.1), Column($0.0), List([66,
     65, 64, 63]), Scan([15, 67, 0]), List([58]), List([64]), HashJoin([28, 0,
    70, 69, 68, 62]), List([50]), List([66]), HashJoin([28, 0, 73, 72, 71, 55]),
    List([43]), List([53]), HashJoin([28, 0, 76, 75, 74, 46]), List([53, 52, 51,
    50, 49, 48, 47, 44, 43, 42, 41, 40, 66, 65, 64, 63, 58, 57, 56]), Proj([78,
    77]), Column($2.8), Column($2.7), Column($2.6), Column($2.5), Column($2.4),
    Column($2.3), Column($2.2), Column($2.1), Column($2.0), List([88, 87, 86, 85,
     84, 83, 82, 81, 80]), Table($2), Scan([90, 89, 0]), Constant(Int32(15)), Eq
    ([83, 92]), Constant(String(""\%BRASS"")), Like([84, 94]), And([95, 93]),
    Filter([96, 91]), List([44]), List([88]), HashJoin([28, 0, 99, 98, 97, 79]),
    List([3]), LeftOuter, HashJoin([102, 0, 99, 101, 100, 39]), List([88, 87, 86,
     85, 84, 83, 82, 81, 80, 53, 52, 51, 50, 49, 48, 47, 44, 43, 42, 41, 40, 66,
    65, 64, 63, 58, 57, 56, 1]), Proj([104, 103]), Min(1), List([106]), List([88,
     87, 86, 85, 84, 83, 82, 81, 80, 53, 52, 51, 50, 49, 48, 47, 44, 43, 42, 41,
    40, 66, 65, 64, 63, 58, 57, 56]), HashAgg([108, 107, 105]), Ref(106), List
    ([88, 86, 52, 51, 49, 48, 47, 41, 65, 110]), Proj([111, 109]), Eq([41, 110]),
     Filter([113, 112]), List([88, 86, 52, 51, 49, 48, 47, 65]), Proj([115, 114])
    , Desc(48), List([117, 65, 52, 88]), Constant(Int32(0)), Constant(Int32(100))
    , TopN([120, 119, 118, 116]), List([48, 52, 65, 88, 86, 51, 49, 47]), Proj
    ([122, 121])] }
```

A.2. Query 6

```
    select
    sum(l_extendedprice * l_discount) as revenue
from
    lineitem
where
    l_shipdate >= date '1994-01-01'
    and l_shipdate < date '1994-01-01' + interval '1' year
    and l_discount between 0.08 - 0.01 and 0.08 + 0.01
    and l_quantity < 24;
```

A.2.1. Relational Expression

Stage 0:

```
RecExpr { nodes: [Constant(Bool(true)), Column($7.15), Column($7.14), Column(
    $7.13), Column($7.12), Column($7.11), Column($7.10), Column($7.9), Column
    ($7.8), Column($7.7), Column($7.6), Column($7.5), Column($7.4), Column($7
    .3), Column($7.2), Column($7.1), Column($7.0), List([16, 15, 14, 13, 12,
    11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1]), Table($7), Scan([18, 17, 0]),
    Constant(Int32(24)), Lt([12, 20]), Constant(Decimal(0.01)), Constant(
    Decimal(0.08)), Add([23, 22]), LtEq([10, 24]), Sub([23, 22]), GtEq([10,
    26]), And([27, 25]), Constant(Interval(Interval { months: 12, days: 0, ms
    : 0 })), Constant(Date(Date(8766))), Add([30, 29]), Lt([6, 31]), GtEq([6,
     30]), And([33, 32]), And([34, 28]), And([35, 21]), Filter([36, 19]), Mul
    ([11, 10]), Sum(38), List([39]), Agg([40, 37]), Filter([0, 41]), List([])
    , Order([43, 42]), Ref(39), List([45]), Proj([46, 44]), Constant(Int32(0)
    ), Constant(Null), Limit([49, 48, 47])] }
```

Stage 3:

```
RecExpr { nodes: [Constant(Bool(true)), Column($7.10), Column($7.6), Column($7.5)
    , Column($7.4), List([4, 3, 2, 1]), Table($7), Scan([6, 5, 0]), Constant(Date
    (Date(8766))), GtEq([1, 8]), Constant(Date(Date(9131))), Gt([10, 1]), And
    ([11, 9]), Constant(Int32(24)), Gt([13, 4]), And([14, 12]), Filter([15, 7]),
    List([3, 2]), Proj([17, 16]), Constant(Decimal(0.07)), GtEq([2, 19]),
    Constant(Decimal(0.09)), GtEq([21, 2]), And([22, 20]), Filter([23, 18]), Mul
    ([2, 3]), Sum(25), List([26]), Agg([27, 24]), Ref(26), List([29]), Proj([30,
    28])] }
```

A.3. README

In order to run the project, you can make use of the `run.sh` script we developed to ease the analysis process. However, you will need to work with our modified version of the **Egg library**. You will also have to change the **Cargo.toml** file in the Risinglight folder, in the line "`egg = { path = "/home/blackparkd/github/egg", features = ["deterministic"] }`" to point to your local egg directory. You can also use our remote branch of the egg library, replacing the command to the local directory with: "`egg = { git = "https://github.com/Blackparkd/egg", branch = "eggLcc" , features = ["deterministic"] }`"

The README file in our GitHub Repository: `https://github.com/Blackparkd/risinglight/blob/main/README.md` contains all the necessary information to navigate the reader through this script.

A.4. Important Links

- **Risinglight GitHub Repository:** `https://github.com/risinglightdb/risinglight`
- **Egg library:** `https://github.com/egraphs-good/egg`
- **Modified Risinglight GitHub Repository:** `https://github.com/Blackparkd/risinglight/tree/main`
- **Modified Egg library:** `https://github.com/Blackparkd/egg/tree/eggLcc`
- **Generated charts folder:** `https://github.com/Blackparkd/risinglight/tree/main/src/planner/outputs/bar_charts`
- **Python scripts:** `https://github.com/Blackparkd/risinglight/tree/main/src/planner/script`
- **Modified optimizer.rs file:** `https://github.com/Blackparkd/risinglight/blob/main/src/planner/optimizer.rs`

A.5. References

These are the most important concepts and tools we used while developing this project.
These concepts are essential for understanding the context of our work and the technologies used.

- **Risinglight:** Risinglight is an open-source SQL database system designed for educational purposes. It is built on top of the Rust programming language and aims to provide high performance and scalability for data processing tasks.
- **Egg library:** Egg is a Rust library for writing and manipulating e-graphs. It provides a framework for implementing various optimization techniques, including those used in query optimization.
- **E-graphs:** E-graphs are data structures used to represent and optimize expressions in a way that allows for efficient manipulation and transformation. They are particularly useful in the context of query optimization, where multiple equivalent expressions may exist.
- **SQL:** SQL (Structured Query Language) is a standard programming language used for managing and manipulating relational databases. It allows users to perform various operations, such as querying data, inserting records, updating information, and deleting entries.

- **Query Optimization:** Query optimization is the process of transforming a SQL query into an efficient execution plan. It involves analyzing the query structure, applying various optimization techniques, and selecting the best plan based on cost estimates.
- **Cost Model:** A cost model is a mathematical representation used to estimate the resource consumption (e.g., time, memory) of different query execution plans.
- **Relational Algebra:** Relational algebra is a formal system for manipulating relations (tables) in a database. It provides a set of operations, such as selection, projection, and join, that can be used to express queries and transformations on relational data.
- **TPC-H:** TPC-H is a widely used benchmark for evaluating the performance of database systems. It consists of a set of complex SQL queries that simulate real-world data processing scenarios, allowing researchers and developers to assess the efficiency of their systems.
- **Python:** Python is a high-level programming language known for its simplicity and readability. It is widely used for data analysis, visualization, and scripting tasks. In this project, Python was used to generate the charts present in this report and visualize the optimization process.
- **Matplotlib:** Matplotlib is a popular Python library for creating static, animated, and interactive visualizations. It provides a wide range of plotting functions and customization options, making it suitable for generating bar charts.
- **Rust:** Rust is a systems programming language that emphasizes safety, concurrency, and performance. It is used in the Risinglight database system to implement the query optimizer and other components.

A.6. Acknowledgments

We would like to express our gratitude to Professor **José Orlando Pereira** for his guidance throughout this project. His expertise and insights were invaluable in shaping our understanding of query optimization and the Risinglight database system.
You can find more information about his work and research on his personal page:
https://jopereira.github.io.